FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Konzepte moderner Programmiersprachen (KmPS)

(Wahlfach, Modulnummer 55685)

Wintersemester 2025/2026

Prof. Dr. Andreas Claßen
(zusammen mit Prof. Dr. Heinrich Faßbender)
Fachbereich 5 Elektrotechnik und Informationstechnik
FH Aachen

Nebenläufigkeit & Parallelismus in Programmiersprachen:

JavaScript,
Callbacks,
Promises / Futures,
`async` / `await`

} (JS & generell)

Event-Driven / Reactive Programming

- Laufzeitumgebung triggert Events.
- Programm registriert Callback-Funktionen für die Events.

Zum Beispiel:

- Events seitens Plattform:
  z.B. Web Browser / HTML events, I/O events (NW, disk, …), ...

"Higher order" event-driven programming: (Functional) Reactive Programming (FRP).

Reactive Programming

… als "higher order" event-driven programming:

Event Streams als "unendliche Listen".
Operationen auf Event Streams: **`map`**, **`filter`**, Kombination von Streams etc.

Beispiel mittels RxJS http://reactivex.io/ :

```
fromEvent(document, 'mousemove').pipe(
  map(event => event.clientX),
  throttleTime(300)
)
```

… in Zeitabständen von 300 ms

Die x-Koordinate der Mausposition "liefern" …

Wie bei Unix Pipes:
Der Event Stream wird im ersten Teil der Pipe "als Wert generiert" und dann als Wert durch die Pipe durchgereicht (d.h. potentieller impliziter Parameter aller weiteren Stufen der Pipe):

```
fromEvent(document, 'mousemove') | map(event => event.clientX) | throttleTime(300)
```

Quelle: https://rxviz.com/examples/mouse-move

Event-Driven Programming in JavaScript

JavaScript Code registriert Event Handler bei der Runtime.

JavaScript Runtime immer *single-threaded*: *Ein* Thread mit *einer* Event Loop.

Ausnahme:
Web Workers als wirklich parallele Einheiten im Runtime.
Allerdings wie getrennte Anwendungen.

Warum *nur eine* Event Loop in JavaScript?

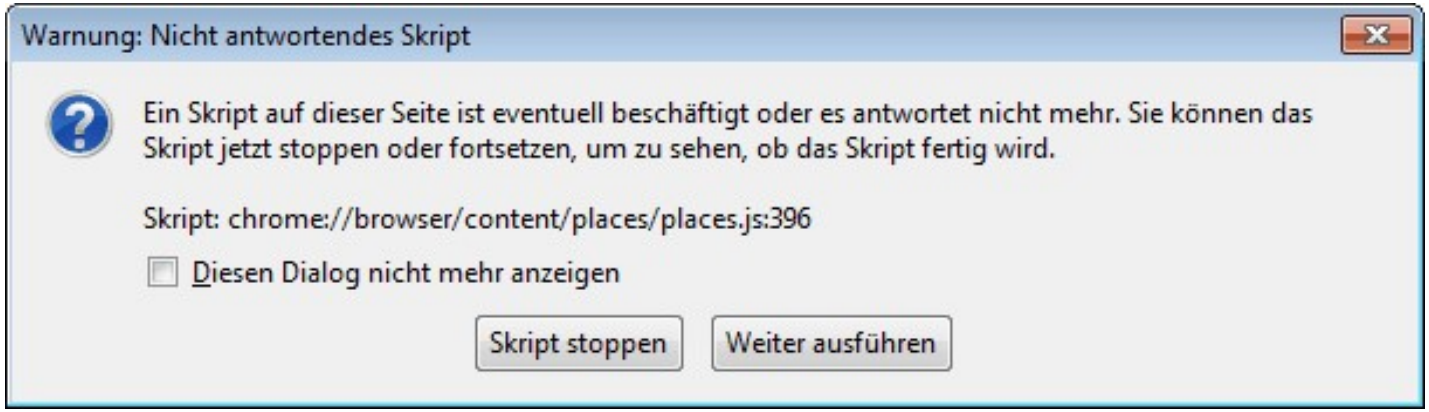
- Vermeide komplexe Anforderungen an Plattform (insbes. Web Browser).
- Seiteneffekte Teil von JavaScript. Parallele Event Loops könnten wegen Seiteneffekten Semantik der Programme ändern.
- Jeder Callback sequentiell programmierbar ohne Notwendigkeit, Shared Data vor konkurrierenden Zugriffen zu schützen.

Nachteil des "eine Event Loop" Ansatzes bei JavaScript

… wie beim kooperativen Multitasking:

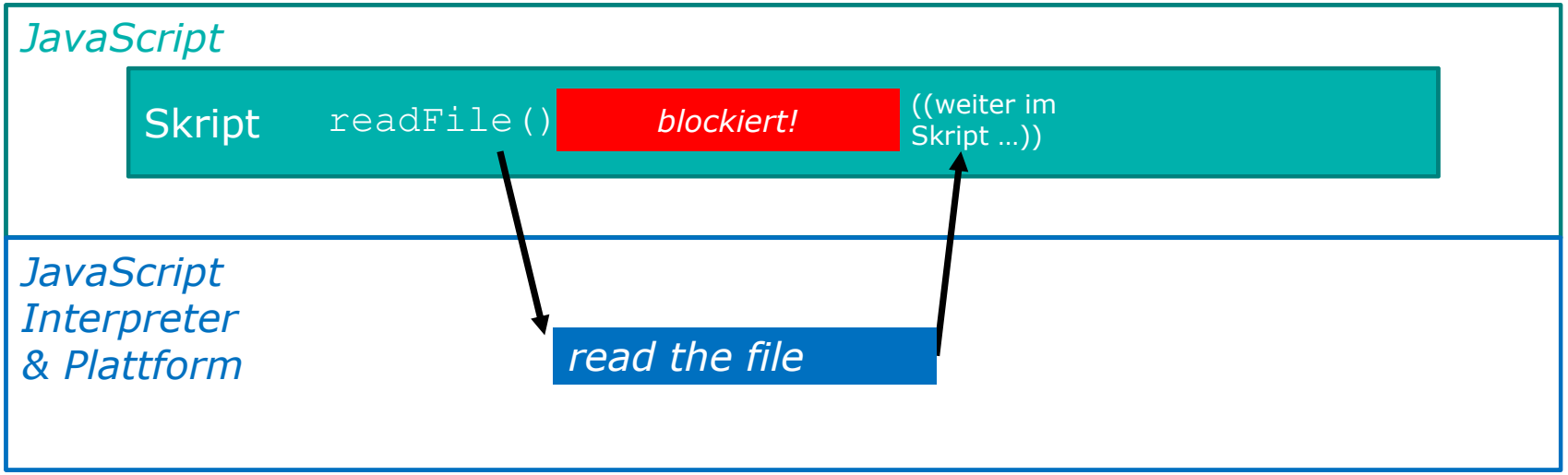
Während Event abgearbeitet wird, kann kein anderer Event abgearbeitet werden.
Bei Callbacks mit langer Abarbeitungszeit: JS Runtime nicht mehr responsive.

Bei Browsern oft: "Ein Skript antwortet nicht mehr".



Blockierende I/O Plattform-Operationen

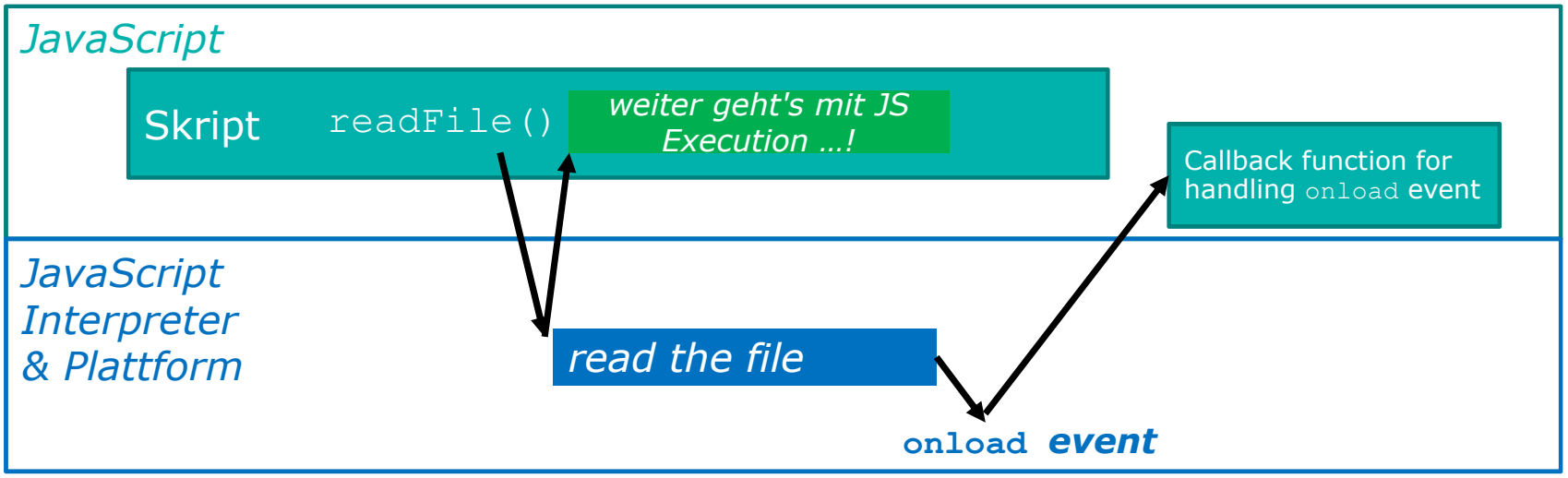
I/O Operationen sind bei JS Anwendungen das, was viel "Wartezeit" beinhaltet.
Sind aber Teil der Plattform (Browser, JavaScript Engine), auf JS Ebene nur Funktionsaufruf …



Google V8 JavaScript Engine

… *als "Trendsetter / Durchbruch":*

Erste JavaScript Runtime mit *event-driven Architektur für "Plattform-I/O-Operationen"*, d.h. I/O Operationen non-blocking und als event generators.
Dadurch Geschwindigkeitsvorteil des Chrome Browsers zu der Zeit …



Node.js JavaScript Plattform

… nutzt V8 non-blocking I/O Plattform. Realisiert eine komplett in event-driven Architektur aufgebaute Plattform für server-seitiges JavaScript.

Weitere interessante Features u.a.:
Web Server API.
`npm` Package Manager.

Inspiriert von event-driven Architektur / non-blocking I/O der NGINX Web Server & Proxy Software.

Quellen: https://nodejs.org/en/about/ , https://de.wikipedia.org/wiki/Datei:Node.js_logo.svg , Dezember 2018.

Node.js Beispiel: Non-Blocking, **Event-Driven Callbacks**

Synchrone Funktionsaufrufe blockieren.
Callbacks werden **asynchron ausgeführt** und blockieren dadurch nicht.

Synchroner file read (via synchronem Funktionsaufruf):

```
const fs = require('fs');
const data = fs.readFileSync( '/file.md' ); // blocks here until file is read
console.log(data);
moreWork();
```

data dependency

`console.log()` wird vor `moreWork()` aufgerufen.

Asynchrone Version (via Callback):

```
const fs = require('fs');
fs.readFile( '/file.md', (err, data) => { if (err) throw err; console.log(data); } );
moreWork();
```

data dependency

`moreWork()` wird vor `console.log()` aufgerufen!!

`fs.readFile()` ist non-blocking.

Quelle: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

"Callback Hell"

Programmierung mittels Callbacks führt bei abhängigen Programmteilen zu tief verschachteltem, unübersichtlichem Programmcode.

(Pseudo-) Beispiel: Asynchrones Lesen dreier abhängiger Werte nacheinander, mittels Callbacks

```
get_data1(
   (data1) => { get_data2(
                   data1,      // … welche Werte für data2 erlaubt sind, hänge von data1 ab
                   (data2) => { get_data3(
                                   data2, // … welche Werte für data3 erlaubt sind, hänge von data2 ab
                                   (data3) => { console.log(data3);}
                                   );
                               }
                );
             }
    );
```

Diese Einrückungen nach rechts nennt man manchmal auch "Pyramid of Doom" …

Nebenläufigkeit "im imperativen Kontrollfluss"

… Nebenläufigkeit im imperativen Kontrollfluss "verstecken" …

… wodurch der Code seine "imperative Natur der Programmierung" behält.

Nebenläufigkeit "im imperativen Kontrollfluss": **Promises / Futures**

… ermöglichen "sequentielle Programmierung" voneinander (daten-) abhängiger asynchroner Programmteile.

Future / Promise (engl. ‚Versprechen'): Platzhalter-Objekt für Berechnungsergebnis, wobei Berechnung ggfs. noch nicht beendet.

Platzhalter kann von folgenden (vom Wert abhängigen) Programmteilen entgegengenommen werden, ohne dass vorherige asynchrone Berechnung schon beendet sein muss *(Wert vielleicht erst "später" benötigt …)*

JavaScript Promises: Erstes Beispiel

```
let promise_val = fs.readFileAsync('/file.md');
// … triggering an async operation, promise as immediate return value
// … promise as placeholder for file contents

moreWork(); // … (maybe) not yet using promise / file contents at all

promise_val.then(
   data => { console.log(data); } // … requires the file content data
 );

andSomeMoreWork(); // … (maybe) not using promise / file contents
```

Promises in JavaScript

… gibt es seit ECMAScript 6 (ECMAScript 2015).

Promise Objekte:

- Repräsentieren *einen* zukünftigen Ergebniswert.
  Kann aber *ein* Objekt mit *mehreren* Attributen sein.
- Zustand `pending`, wenn Ergebniswert noch nicht fertig berechnet.
  Ansonsten `settled`. Dabei entweder `resolved` (Ergebnis erfolgreich verfügbar) oder `rejected` (Fehler während der asynchronen Operation; Fehlerwert als Ergebnis).

Promises in JavaScript: Syntax

Variante mit Arrow Function Schreibweise …

Beispiel:

```
function async_fun(…) {
    // …
    return new Promise(
```

Executor

```
  ( resolve , reject ) =>
     {
        // Operationen, ggfs. asynchroner Art,
        // zur "Berechnung" des konkreten Ergebniswerts
        if ("async. Berechnung erfolgreich") resolve(…);
        else reject(…);
     }

);
```

```
}

var promise1 = async_fun(…);
```

Alternativ, ohne Arrow-Notation:

```
function async_fun(…) {
    // …
    return new Promise( function( resolve , reject ) {
                            // Operationen, ggfs. asynchroner Art,
                            // zur "Berechnung" des konkreten Ergebniswerts
                            if ("async. Berechnung erfolgreich") resolve(…);
                            else reject(…);
                        } );
}
```

Promises in JavaScript

Beispiel:

```
function resolves_after_5_s() {
  return new Promise( (resolve, reject) => {
      // Executor wird sofort ausgeführt …
      console.log("Executor started.");

      // Nach 5 Sekunden (asynchron) wird das Promise Objekt auf den Wert 42 resolved
      setTimeout( () => { resolve(42); console.log("Resolved.");} , 5000);
    } );
}

let promise2 = resolves_after_5_s();

console.log("Script done.");
```

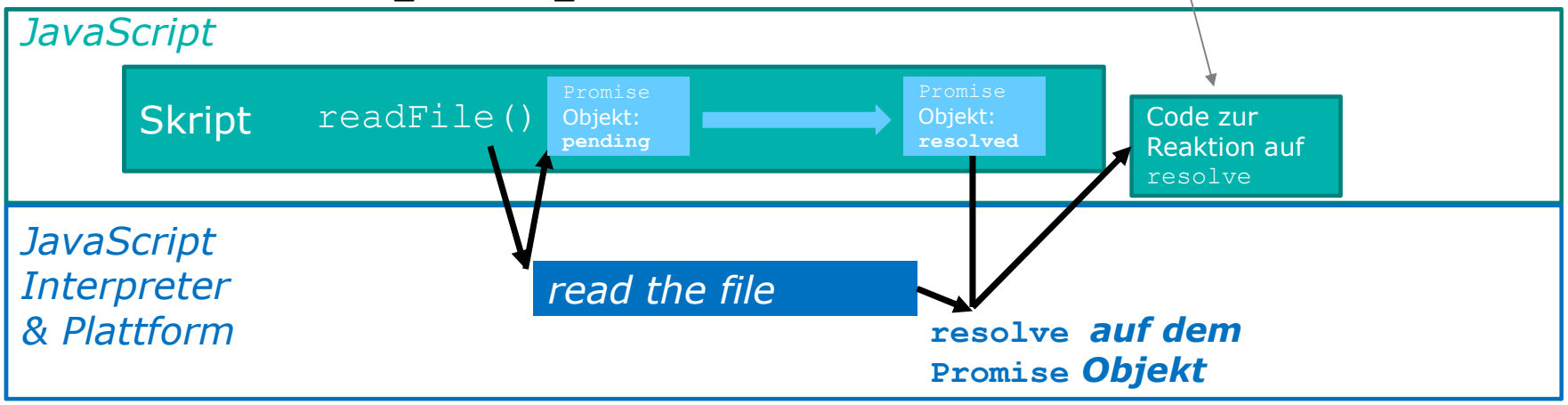
Output:

```
Executor started.
Script done.
Resolved.
```

Plattform-(I/O)-Operationen geben Promises zurück

Symbolisches Beispiel: (symbolisch, da diese Funktion `readFile()` ja eigentlich "low-level in der Plattform" realisiert ist und nicht als JavaScript code …)

```
function readFile() {
  return new Promise( (resolve, reject) => {
      // Liest die Datei. Wenn fertig gelesen:
      if /(* read o.k. */) resolve( <file content> ) else reject( <error code> ) ;
    } );
}
let promise2 = readFile();
promise2.then( file_content_data => { … code … } );
```



Promises in JavaScript: Executor Function

… als Parameter-Funktion des Promise Konstruktors. Wird sofort ausgeführt.

Hat zwei "abstrakte" Parameter: Funktionen, vom JavaScript System parametrisiert.

Nur der Rumpf des Executors wird programmiert, inkl. dortiger Aufrufe von `resolve()` bzw. `reject()`.

```
function async_fun(…) {
    // …
    return new Promise( ( resolve , reject ) =>
                        {
                          // Operationen, ggfs. asynchroner Art,
                          // zur "Berechnung" des konkreten Ergebniswerts
                          if ("async. Berechnung erfolgreich") resolve(…) ;
                          else reject(…) ;
                        }
                      );}
var promise1 = async_fun(…)
```

Promises in JavaScript: `resolve()` und `reject()`

… Callback-Funktionen mit *einem* Parameter. Tragen diesen als Ergebniswert bzw. Fehler-Wert ins Promise Objekt ein.

Im Executor aufrufen, ohne sie vorher definieren zu müssen …

Beispiel:

```
function resolves_after_5_s() {
  return new Promise( (resolve, reject) => {
    // Nach 5 Sekunden (asynchron) wird das Promise Objekt auf den Wert 42 resolved
    setTimeout( () => { resolve(42); console.log("Resolved.");} , 5000);
  } );
}
```

Promises in JavaScript: `.then()` und `.catch()`

`.then()` gibt daten-abhängigen "Nachfolgecode" in Callback-Form an.

Zwei Callbacks als Parameter: Für Erfolg bzw. Fehlschlag der Berechnung.
Beispiel:

```
promise1.then(  result => { console.log(result); } ,
                error  => { console.log("Error:", error); } );
```

Zweiter Parameter kann weggelassen werden. Dann keine Fehlerbehandlung.

Alternativ: `.catch()` für Fehler-Callback.
Beispiel:

```
var promise1 = do_it_async();  // … gibt Promise Objekt zurück
// …
promise1
  .then(  result1 => { console.log(result1); } )
  .catch( err => { console.log(err); } );
```

Promises in JavaScript: `.then()`

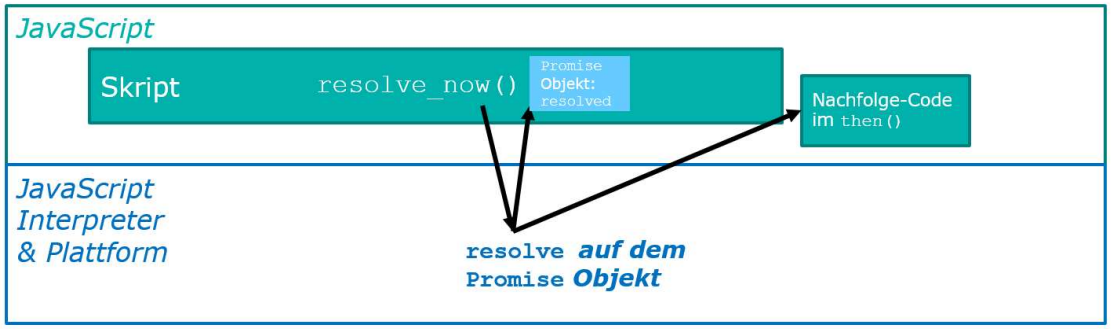
`.then()` Callback *immer* verzögert ausgeführt, auch wenn Promise schon Wert hat.
Dadurch klarer bestimmtes Programmverhalten.

Beispiel:

```
function resolve_now() { return new Promise( (resolve, reject) => { resolve(42); } ); }
let promise1 = resolve_now(); // sofortiges resolve() der Promise
promise1.then( result => { console.log("Resultat: " + result); } );
console.log("Skript Ende!");
```

Output:

```
Skript Ende!
Resultat: 42
```



Interna des Schedulings in JavaScript: Jobs, Job Queues

Zwei Job Queues:

- **`ScriptJobs`**: vollständige Ausführung eines Skripts.
- **`PromiseJobs`**: durch `resolve()` / `reject()` ausgelöste Jobs
  => Code im `then()` / `catch()`.

Innerhalb der JavaScript Event Loop:

- Wenn Jobs in **`ScriptJobs`** Queue: Führe den ersten Job dort aus.
  Dieser hängt u.U. über `then()` / `catch()` neue Jobs an die **`PromiseJobs`** Queue an.
- Wenn aktueller Job aus **`ScriptJobs`** Queue abgearbeitet:
  Arbeite **`PromiseJobs`** Queue ab, ggfs. bis diese komplett leer.
  Jobs der **`PromiseJobs`** Queue hängen u.U. neue Jobs an diese Queue an. Diese ggfs. auch abarbeiten.
- Dann zurück zur Event Loop. Holt nächsten Job aus **`ScriptJobs`**.

Promise Chaining: `.then()` gibt immer Promise zurück

`.then()` gibt als Rückgabewert immer ein Promise Objekt zurück.
Nicht-Promise-Rückgabewert wird automatisch in Promise "eingepackt" …

Dadurch können `.then()` verkettet werden (**chaining**).

Umgekehrt wird beim Chaining für den Aufruf der Callback-Fkt auch der Ergebnis-/Error-Wert automatisiert aus der durch den vorherigen Schritt gelieferten Promise Promise "ausgepackt".

Beispiel:

```
var p = new Promise( (resolve, reject) => { resolve(1); } );


p.then( value => { console.log( value ); return value + 1; } )
 .then( value => { console.log( value ); });
```

Promises in JavaScript: `.catch()`

… für `reject()` im Executor bzw. Fehler in einem `.then()` Schritt.

Sowohl `reject()` als auch Exceptions werden durch `.catch()` gefangen.

Also einheitliches Error Handling bei Promises. Keine verstreuten `try` / `catch` Blöcke.

Beispiel:

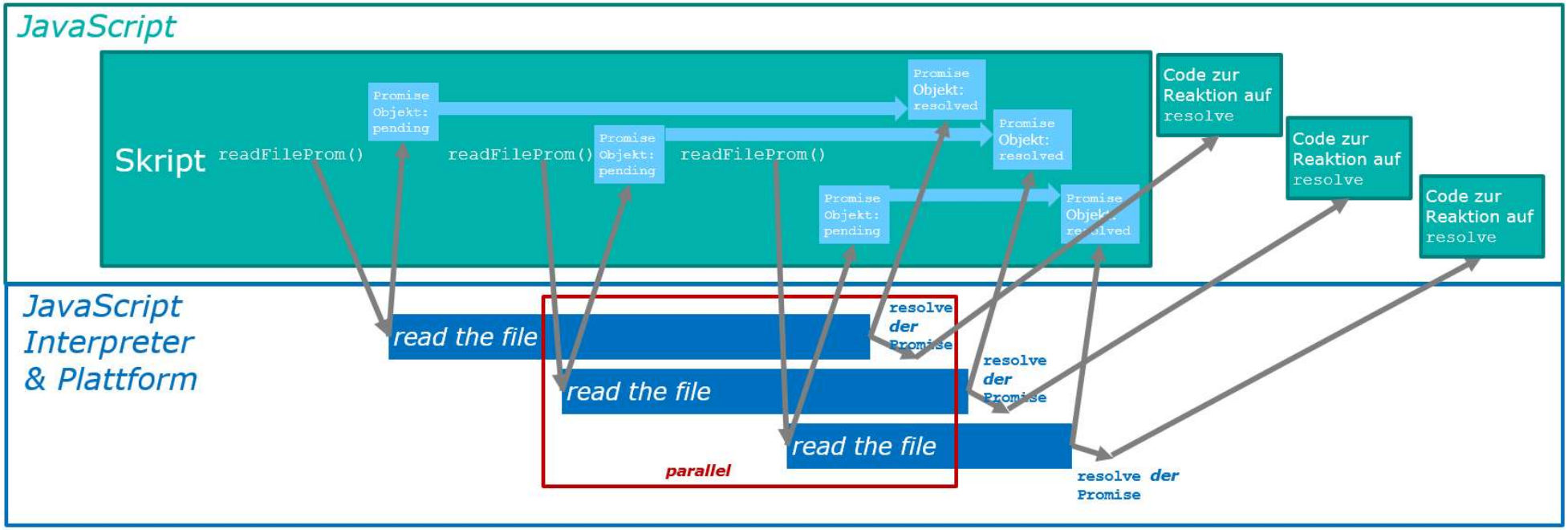
```
function throwingFunction(num) { return new Promise( (resolve, reject) => {
    if (typeof num !== 'number') throw new Error('invalid num');
    // do something else asynchronously and call resolve(result) }; }

function rejectingFunction(num) { return new Promise( (resolve, reject) => {
    if (typeof num !== 'number') reject( new Error('invalid num') );
    // do something else asynchronously and call resolve(result) }; }

var resultThrowing = throwingFunction(num)
    .then(  result => { console.log(result); } )
    .catch( error  => { console.log(error);  } );

var resultRejecting = rejectingFunction(num)
    .then(  result => { console.log(result); } )
    .catch( error  => { console.log(error);  } );
```

`Promise.all` parallelisiert asynchrone Plattform-Operationen




```
let filenames = ['datei1.html', 'datei2.html', 'datei3.html'];
Promise.all( filenames.map(readFilePromisified) );
```

Promisification

… bezeichnet Umwandlung einer Library vom Callback-Style in Promise-Style.

Ggfs. systematisch mittels Wrapper Funktion **`promisify(…)`**.

Schematisch & vereinfacht könnte diese so aussehen:

```
let function_promisified = promisify(orig_cb_function);

// Use the new, promisified function: ...
function_promisified(…).then(…);
```

Nebenläufigkeit "im imperativen Kontrollfluss": **`async` / `await`**

… gibt es seit ECMAScript 7 (ECMAScript 2017).

Asynchrone Funktion als neue Funktions-Art. Im Rumpf mittels **`await`** auf Resultate anderer asynchroner Funktionen warten.
`await` nur innerhalb von `async` Funktionen und auf oberster Ebene in Modulen.
`async` / `await` ergänzt den Promise Mechanismus.

Beispiel:

```
async function getUserAsync(name)
{
  let response = await fetch(`https://api.github.com/users/${name}`);
  let json = await response.json();
  console.log("JSON:", json);
  return json;
}

let data = await getUserAsync('docker');
console.log("Promise settled, so async function done:", data));
```

Oder mittels Promise-Operationen:

```
getUserAsync('docker')
  .then(data => console.log("Promise settled, so async function done:", data));
```

`async` Funktionen

… geben immer ein Promise Objekt als Rückgabewert zurück.
"Nicht-Promise" Rückgabewert würde automatisch als `resolve()` Wert in Promise "eingepackt"!

Beispiel:

```
async function getUserAsync(name)
{
  let response = await fetch(`…`);
  let json = await response.json(); // ... "json" ist Resultatwert, nicht Promise
  console.log("JSON:", json);
  return json;
}

getUserAsync('docker')
  .then(data => console.log("Promise settled, so async function done:", data));
```

… `.then()` ist immer auf dem Rückgabewert aufrufbar.
Also ist dieser eine Promise!

Fehlerbehandlung bei `async` / `await`

… mittels des normalen `try` / `catch` Exception Handlings, wie beim sequentiellen Programmieren.
Kein "neuartiges" spezielles `.catch()` mit seinem Callback-Parameter nötig …

Beispiel:

```
async function get_doc(db, docid) {
  // …
  let doc;
  try {
    doc = await db.get(docid);
    console.log(doc);
    // return doc;
  }
  catch (err) { throw err; }
}
```

(Nur) mit Promises:
Mittels `.catch()` Error Handling der Promises, nicht mittels des "normalen" `try` / `catch` …

```
function get_doc(db, docid) {
  // …
  db.get('docid')
  .catch( err => { throw err; } )
  .then(  doc => { console.log(doc); } )
}
```

Bis auf das **`await`** identischer Code zur sequentiellen Programmierung.

`async` Funktionen

… entsprechen einem `new Promise (...);` und ihr Rumpf entspricht dann dem Executor dieser neuen Promise.

`await` wartet auf das Settlement einer Promise, d.h. die Zeilen hinter einem **`await`** entsprechen der Parameter-Funktion eines `then()`.

Beispiel:

```
async function getUserAsync(name)
{
  let response = await fetch(`…`);
  let json = await response.json()
  console.log("JSON:", json)
  return json;
}

let data = await getUserAsync('docker');
console.log("Promise settled:", data));
```

*versus **Promise** Variante:*

```
function getUser(name){
  return new Promise( (resolve, reject) => {
    fetch(`…`) // ... gibt Promise zurück
      .then( response => { return response.json(); } )
      .then( json => { console.log("JSON:", json); resolve(json); } )
  });
}
getUserAsync('docker')
  .then(data => console.log("Promise settled:", data));
```