Disclaimer

The content of this talk is derived from a term paper that was prepared during the winter semester 2016/2017 in the course "1908 – Seminar Moderne Programmiertechniken" at the FernUniversität Hagen.
The official presentation of the paper will be between 10^{th} and 12^{th} of March 2017 in Hagen.
The original title of the term paper is "Asnychrone Programmierung: Moderne Methoden in ECMAScript 6".

This presentation will be made available on `https://github.com/opt12/EnterThePromisedLand` and is licensed under the
Creative Commons Attribution Share Alike 4.0 license.

Enter the Promised Land

Modern Methods for Asynchronous Programming in ECMAScript 6

Felix Eckstein

JavaScript Hobbyist

HannoverJS – JavaScript User Group Meeting
January 26th, 2017

Enter the Promised Land

Enter the Promised Land
Modern Methods for
Asynchronous Programming in ECMAScript 6

Felix Eckstein

JavaScript Hobbyist

HannoverJS – JavaScript User Group Meeting
January 26th, 2017

- Wer von euch programmiert in JavaScript?
- Wer von Euch ist der Meinung, dass JavaScript eine ernstzunehmende Programmiersprache ist?
- Also programmieren alle von uns asynchronen Code
- Was habe ich über asynchrone Programmierung gelernt weshalb sind mit ECMAScript 6 (2015) wirklich tolle Features im Sprachkern angekomen sind
- Wege aus der Callback Hell
- Zunächst Grundlagen async Programmierung und Callbacks
- Promises für asynchrone Daten
- Generators für komplexe Kontrollflüsse
- Wer benutzt tagtäglich Promises?

Keine Angst, es wird Katzenfotos geben

Outline

1. Introduction
   - JavaScript: Good to Know
   - Basic Asynchronous Programming
2. Callbacks
   - Traditional Callback Approach
   - Downsides
3. Promises
   - Foundation
   - Mode of Operation
   - The Fun of Promises
4. Generators
   - Foundation
   - Promises and Generators

Modern Methods in ECMAScript 6
What's it all about?
Today we want to walk through this agenda:

- What is asynchronous programming?
- How is this done in traditional JavaScript?
- What modern methods were introduced in ECMAScript 6?
  - Promises to pass asynchronous results
  - Generators to ease the control flow
- Putting it all together

JavaScript: Good to Know

- JavaScript has functional properties
  - Functions are “First-Class Citizens” . . .
    . . . and hence can be passed around in variables
- JavaScript is single threaded
- Program execution is Event-Loop driven
- Program execution follows “Run-to-Completion” sematics

- JavaScript has functional properties
  - Functions are "First-Class Citizens" ...
    ...and hence can be passed around in variables
- JavaScript is single threaded
- Program execution is Event-Loop driven
- Program execution follows "Run-to-Completion" sematics

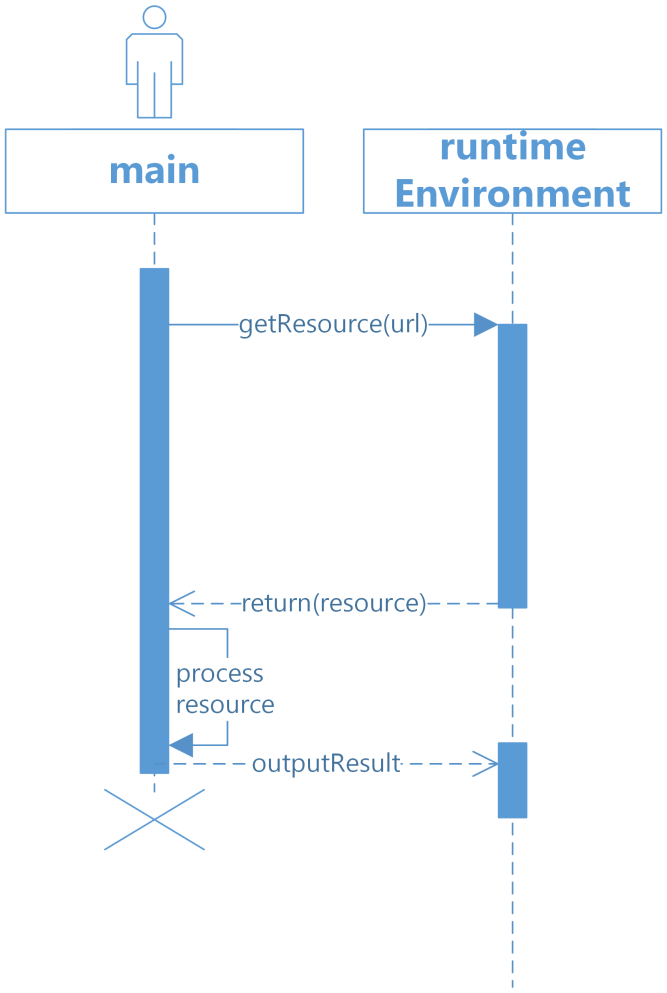
Run-to-Completion:

- Funktionsausführung wird durch Event-Loop angestoßen
- Damit wird ein neuer Stack angelegt mit Funktionsaufrufen
- Funktion wird bis zum Ende ausgeführt
- Bis der Call-Stack leer ist
- Erst dann kommt der nächste Tick der event-Loop und stößt auf leerem stack wieder neue Funktionsausführung an
- Es gibt keine Möglichkeit, dass eine Funktionsausführung von externen Ereignissen unterbrochen wird

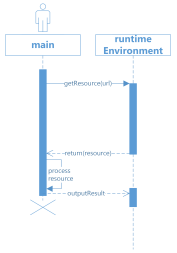
Function Calls in Single Threaded Execution

"Synchronous Programming"

- Synchronous functions
  - get called
  - calculate their result
  - return their result
- Function calls may block the execution
- Waiting for external events leads to inefficiency
- No concurrency possible



- Synchronous functions
  - get called
  - calculate their result
  - return their result
- Function calls may block the execution
- Waiting for external events leads to inefficiency
- No concurrency possible



- Das Hauptprogramm kann nicht weiterarbeiten
- Nebenläufigkeit ist nicht möglich

Asynchronous Programming

- Asynchronous Functions do it differently:
  - Async functions get called now, . . .
  - . . . return immediately and . . .
  - . . . calculate their result.
  - This result is retrieved later
- The calling code regains control directly
- The main program can do useful things while the async function still works on the result.
- This brings concurrency into single threaded execution

- Asynchronous Functions do it differently:
  - Async functions get called now, ...
  - ... return immediately and ...
  - ... calculate their result.
  - This result is retrieved later
- The calling code regains control directly
- The main program can do useful things while the async function still works on the result.
- This brings concurrency into single threaded execution

Erkläre hier, dass die asynchronen Funktionen eben doch in einem anderen Thread ablaufen können.
Eventuell sogar noch die Grafik Zeigen:

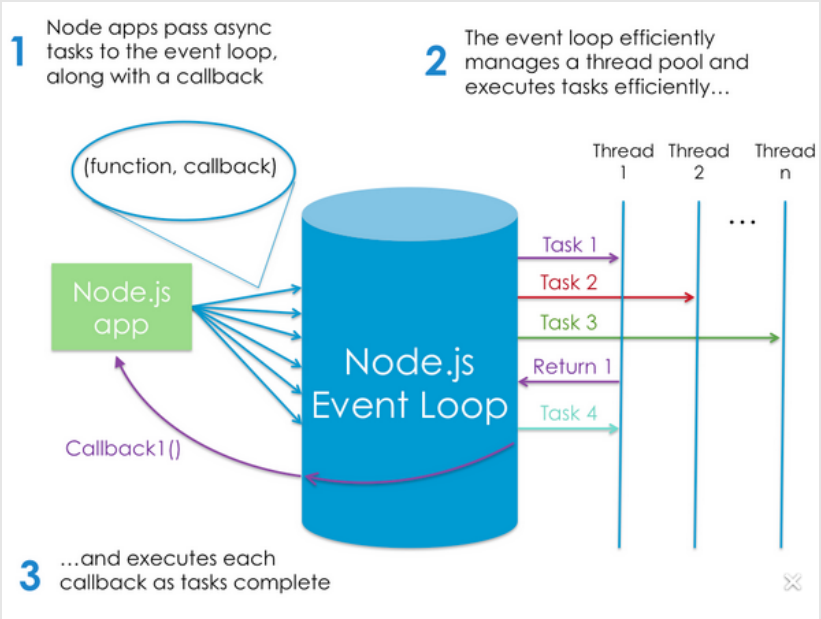


Figure: Asnychronous Functions are executed outside the JavaScript-Single-Thread

Retrieving Asynchronous Results

How can I retrieve the result of an asynchronous function call?

- Main program is divided into chunks
- The chunk placing the function call now runs to completion
- The availability of an async result is enqueued in the event queue
- This event is processed in a later tick of the Event-Loop
- The result is processed in a different execution context

In fact, the relationship between the now and later parts of your program is at the heart of asynchronous programming. [Simpson, 2015]

How can I retrieve the result of an asynchronous function call?

- Main program is divided into chunks
- The chunk placing the function call now runs to completion
- The availability of an async result is enqueued in the event queue
- This event is processed in a later tick of the Event-Loop
- The result is processed in a different execution context

In fact, the relationship between the now and later parts of your program is at the heart of asynchronous programming. [Simpson, 2015]

Trotz anderem Execution Context sind die Variablen vom Aufrufzeitpunkt verfügbar, da diese in einer Closure gehalten werden

Wir werden gleich sehen, wie das ablaufen kann.

Callbacks to Process Results

"Callback" functions may be used to process async results

- The async function receives a callback as parameter
- The callback processes the result of the async call
- The async function is responsible for calling the callback
  - This call can happen either directly (evil!)
  - It can be bound to an event which is enqueued (normal!)
- The callback function is usually executed in a different context than the caller function

The callback function acts as the "continuation" of the program that needs the result.

"Callback" functions may be used to process async results

- The async function receives a callback as parameter
- The callback processes the result of the async call
- The async function is responsible for calling the callback
  - This call can happen either directly (evil!)
  - It can be bound to an event which is enqueued (normal!)
- The callback function is usually executed in a different context than the caller function

The callback function acts as the "continuation" of the program that needs the result.

zunächst der Traditionelle ansatz seit Beginn von JavaScript:

Callbacks

eine Callbackfunktion wird aufgerufen um das ergebnis zu verarbeiten

Synchroner Aufruf: Stört die Run-To-Completion weil der Callback mit auf den Call-Stack kommt

Release of Zalgo: berühmter Blogpost Isaac Z. Schlueter: Designing APIs for Asynchrony [Schlueter, 2013]

Eine ´´coninuaion" als "Fortsetzung" des eigenen Programms

Example: Simple Callback-Code

Callback Invocations are Registered at Events of the Asynchronous Function


```
1  function loadImageCallback(url, cb) {
2      let image = new Image();
3
4      image.onload = function () {
5          cb(null, image);
6      }
7      image.onerror = function () {
8          let message = 'Could not load image at ' + url;
9          cb(new Error(message));
10     }
11     image.src = url;
12 };
13
14 loadImageCallback(logoName, function (err, logo) {
15     if (err) return errorMessage(err);
16     else processImage(logo);
17 });
18 console.log('main program finished');
```

```
function loadImageCallback(url, cb) {
    let image = new Image();

    image.onload = function () {
        cb(null, image);
    }
    image.onerror = function () {
        let message = 'Could not load image at ' + url;
        cb(new Error(message));
    }
    image.src = url;
};

loadImageCallback(logoName, function (err, logo) {
    if (err) return errorMessage(err);
    else processImage(logo);
});
console.log('main program finished');
```

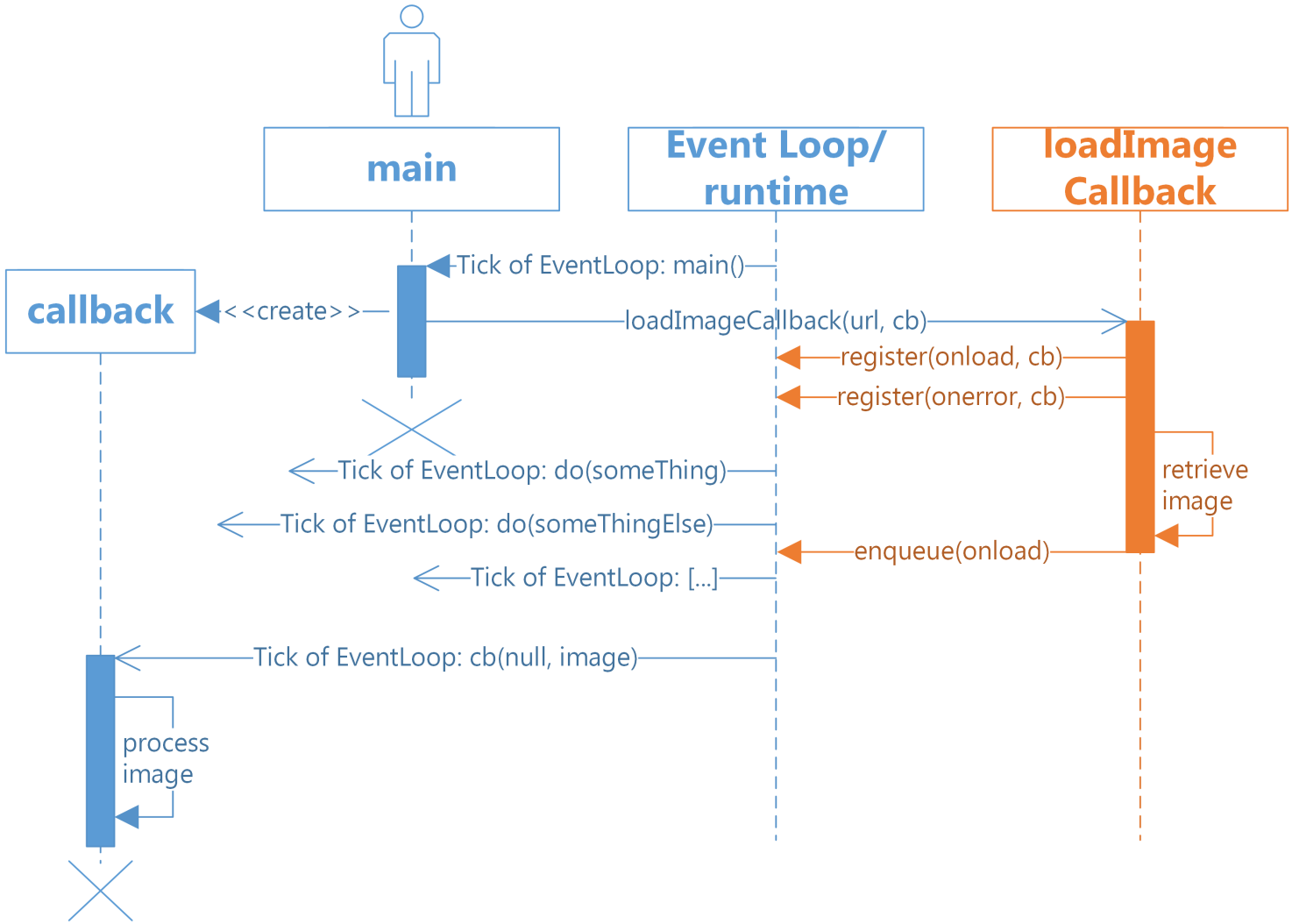
- Aufruf der Asynchronen Funktion
- Callback, der übergeben wird inkl. Error-Handling
- Registrieren des Callbacks zum onLoad-event
- Registrieren des Callbacks zum onError-event
- Verknüpfen der url mit der src Property und damit starten der eigentlichen asynchronen Aktion

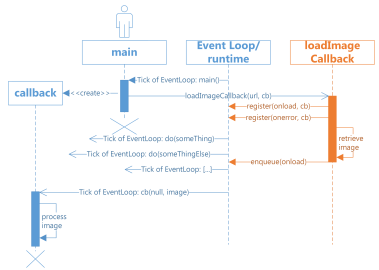
Erkläre kurz Error-Handling

Dann kommt noch ein Sequenzdiagramm

Asynchronous Image Retrieval

Sequence Diagram for Asynchronous Image Retrieval with Callback





- Event Loop startet Hauptfunktion
- Es wird eine CallbackFunktion zum Verarbeiten des Ergebnis on-the-fly erzeugt
- Asynchroner aufruf inklusive Callback
- Sofortige Rückgabe und weiterführung des Hauptprogramms
- Ausserhalb von JavaScript Land:
- Registrieren des Callbacks zum onLoad-event
- Registrieren des Callbacks zum onError-event
- Einreihen des Ergebnisevents onload
- Zwischendrin sind viele andere Sachen passiert in JavaScript Land
- Aufruf der Callback Funktion durch Event-Loop zur Verarbeitung des Ergebnis

Let's get our hands dirty!

Example Using Nested Callbacks

- Load a Logo
- Load a cute Cat
- Apply the Logo as a Watermark
- Show the Image on a Web-Page

```
function watermarkSeq([logo, cat]) {
    loadImageCallback(logo, function (err, logoImg) {









    });
}
```

```
function watermarkSeq([logo, cat]) {
    loadImageCallback(logo, function (err, logoImg) {
        if (err) return errorMessage(err);
        loadImageCallback(cat, function (err, catImg) {
```

- Load a Logo
- Load a cute Cat
- Apply the Logo as a Watermark
- Show the Image on a Web-Page

```
function watermarkSeq([logo, cat]) {
    loadImageCallback(logo, function (err, logoImg) {
    });
}
```

```
function watermarkSeq([logo, cat]) {
    loadImageCallback(logo, function (err, logoImg) {
        if (err) return errorMessage(err);
        loadImageCallback(cat, function (err, catImg) {
```

Das sehen wir uns gleich in echt an.

Und ja, endlich kommt die Katze. Danke für die Geduld!

Eigentlich alles schön!

Warum ist der Vortrag noch nicht zu Ende?

Diese Methode mit Callbacks hat entscheidende Nachteile

Downsides

Callbacks Have Quite some of Downsides

- The “Pyramid of Doom” renders code unreadable
- Nested callbacks are a pain in the brain
- Explicit error handling is needed in each step
  There’s no way for errors to bubble up
- Passing a continuation of your program relies on trust
  Bad things can happen in foreign code:
  - Callback is too early (sync instead of async)
  - Callback is too late or completely omitted
  - Callback is called more than once
  - Async function silently swallows Errors
- Effectively an “Inversion of Control”
  takes place and you have to trust unknown code
  about your program continuation

What if instead of handing the continuation of our program to another party, we could expect it to return us a capability to know when its task finishes, and then our code could decide what to do next? [Simpson, 2015, S. 39]

What if instead of handing the continuation of our program to another party, we could expect it to return us a capability to know when its task finishes, and then our code could decide what to do next? [Simpson, 2015, S. 39]

Gibt es eine besser Möglichkeit ohne diese Nachteile?

Was wäre denn besser?

Schöner wäre, wenn wir

- Nicht einen fremden, nicht vertrauenswürdigen Stück Code die Fortsetzung unseres Programms überlassen
- sondern eine Möglichkeit bekommen selber herauszufinden wenn asynchrone Funktion zu Ende ist
- Wann das Ergebnis vorliegt
- Selber zu entscheiden, was wir tun wollen

There is a Better Way: "...I Promise"

In ECMAScript 6 a new object type was introduced which is called "Promise":
A "Promise" is?

- A placeholder for a future value
- An object that can be used
  by the receiving function now
- An object that can be assigned a value
  by the constructing (async) function later

There is a Better Way: "...I Promise"

In ECMAScript 6 a new object type was introduced which is called "Promise":
A "Promise" is?

- A placeholder for a future value
- An object that can be used by the receiving function now
- An object that can be assigned a value by the constructing (async) function later

- Ich möchte ein Haus kaufen
- Ich habe aber kein Geld
- Ich gehe zur Bank
- Dort bekomme ich jetzt eine Kreditzusage
- Damit kann ich mir schon ein Haus suchen und verhandeln
- Diese Zusage muss erst eingelöst sein, wenn ich später beim Notar sitze und den Vertrag unterschreibe
- Obwohl ich den Wert (das Geld) noch nicht habe, kann ich schon mit der Promise (der Zusage) arbeiten

The States of a Promise

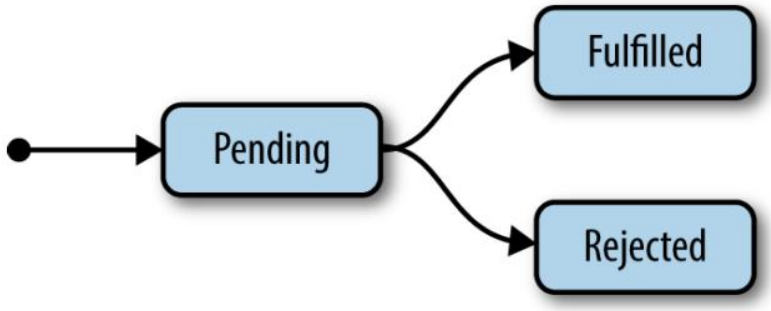


Figure: The three states of a Promise. (from [Parker, 2015, S. 16])

Any Promise object is in one of three mutually exclusive states: ***fulfilled***, ***rejected***, *and* ***pending****:*

- *A promise* `p` *is fulfilled if* `p.then(f, r)` *will immediately enqueue a Job to call the function* `f`*.*
- *A promise* `p` *is rejected if* `p.then(f, r)` *will immediately enqueue a Job to call the function* `r`*.*
- *A promise is pending if it is neither fulfilled nor rejected.*

A promise is said to be ***settled*** *if it is not pending, i.e. if it is either fulfilled or rejected. [Ecma TC39 und Wirfs-Brock, 2015, § 25.4]*

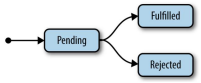


Figure: The three states of a Promise. (from [Parker, 2015, S. 16])

*Any Promise object is in one of three mutually exclusive states: **fulfilled**, **rejected**, and **pending**:*

- *A promise `p` is fulfilled if `p.then(f, r)` will immediately enqueue a Job to call the function `f`.*
- *A promise `p` is rejected if `p.then(f, r)` will immediately enqueue a Job to call the function `r`.*
- *A promise is pending if it is neither fulfilled nor rejected.*

*A promise is said to be **settled** if it is not pending, i.e. if it is either fulfilled or rejected. [Ecma TC39 und Wirfs-Brock, 2015, § 25.4]*

- Eine Promise hat genau drei Status
- Pending: Wenn noch kein Ergebnis vorliegt. Aber das Promise Objekt ist schon da
- Resolved: Ein Ergebnis liegt vor und die asynchrone Operation ist erfolgreich abgeschlossen
- Rejected: Die Asynchrone Funktion hat einen Fehler gehabt und daher gibt es kein Ergebnis
  sondern eine Fehlermeldung
- Eine Promise ist settled oder pending
- Dn status der Promise kann ausschließlich die erzeugende Funktion ändern
- (oder das Laufzeitsystem im Fall von Fehlern)

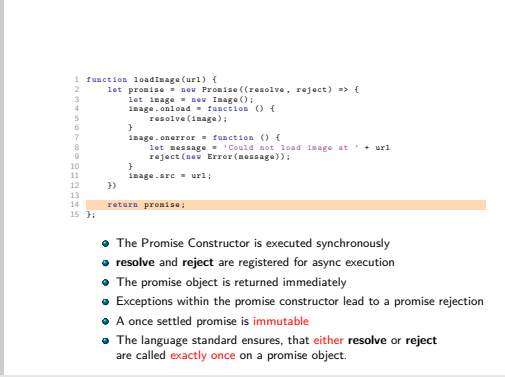
To make beneficial use of Promises, a few basics need to be understood

- Create a Promise
- Resolve a Promise
- Error Handling
- Promise Chaining

Example: Creating a Promise


```
1  function loadImage(url) {
2      let promise = new Promise((resolve, reject) => {
3          let image = new Image();
4          image.onload = function () {
5              resolve(image);
6          }
7          image.onerror = function () {
8              let message = 'Could not load image at ' + url
9              reject(new Error(message));
10         }
11         image.src = url;
12     })
13
14     return promise;
15 };
```

- The Promise Constructor is executed synchronously
- **resolve** and **reject** are registered for async execution
- The promise object is returned immediately
- Exceptions within the promise constructor lead to a promise rejection
- A once settled promise is immutable
- The language standard ensures, that either **resolve** or **reject** are called exactly once on a promise object.



- Ausführen des Konstruktors
- Rückgabe der Promise
- resolve bzw. reject asynchrone aufrufe

The language standard ensures, that either **resolve** or **reject** are called exactly once on a promise object.
Afterwards, the promise object gets immutable

Den status einer Promise kann nur die erzeugende Funktion ändern.

Der Status kann genau einen Übergang von pending zu settled haben.

Resolving a Promise

Getting a Hold of the Value Using `p.then([f],[r])`

The receiver of a promise can register handler functions for fulfillment and rejection of a promise.

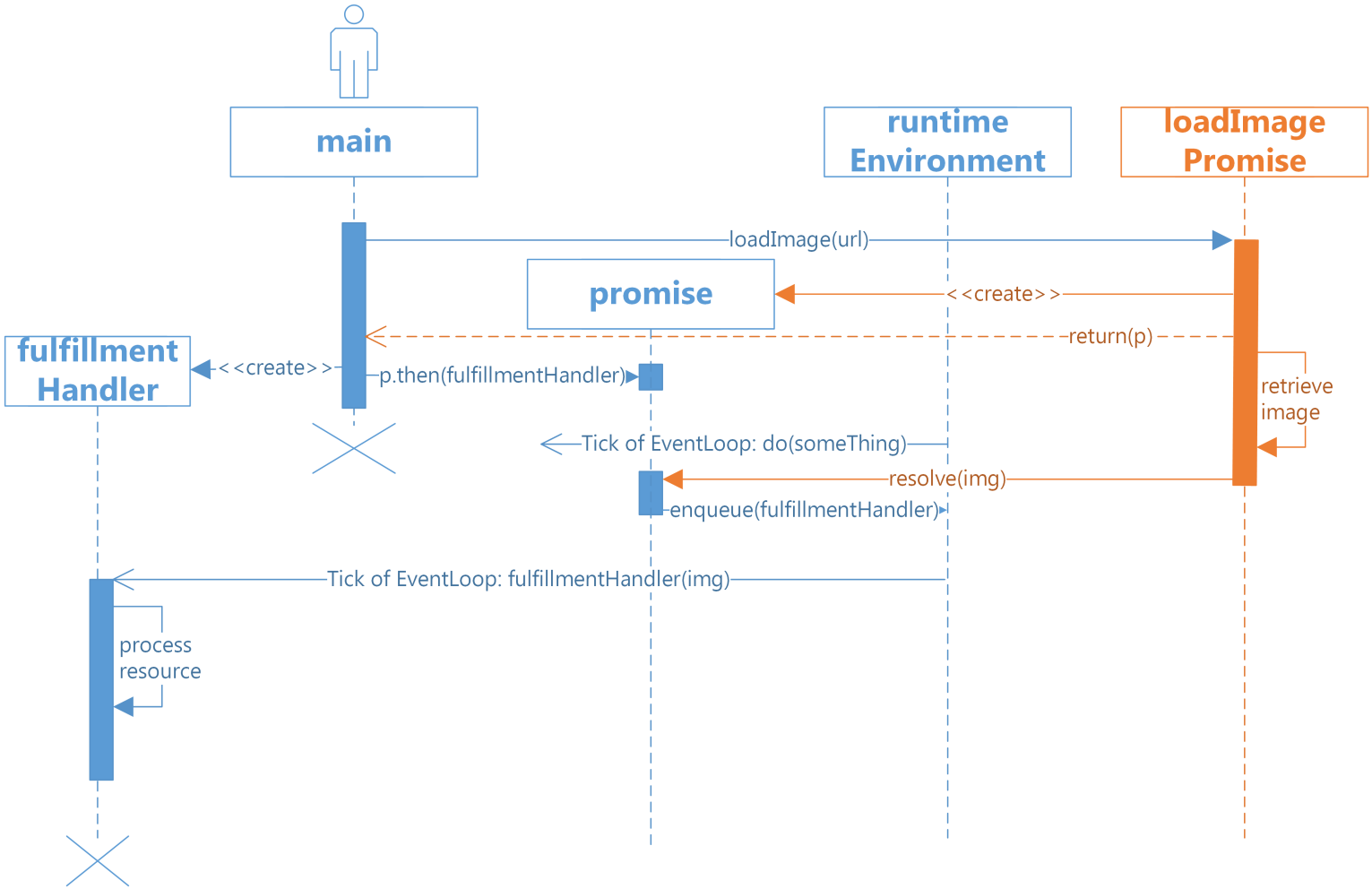
- each promise has a method `p.then([f],[r])` `returns Promise`
  - `f` and `r` are handler functions
  - `f` is called when the promise state is “fulfilled”
  - `r` is called when the promise state is “rejected”
  - either of the function parameters `f` and `r` is optional
- an arbitrary number of handlers may be registered
- calls to `f` or `r` are always enqueued
  and hence executed later

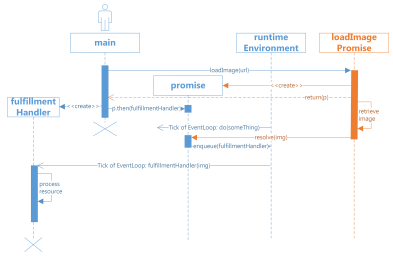
The receiver of a promise can register handler functions for fulfillment and rejection of a promise.

- each promise has a method `p.then([f],[r]) returns Promise`
  - `f` and `r` are handler functions
  - `f` is called when the promise state is "fulfilled"
  - `r` is called when the promise state is "rejected"
  - either of the function parameters `f` and `r` is optional
- an arbitrary number of handlers may be registered
- calls to `f` or `r` are always enqueued and hence executed later

- Jede Funktion, die eine Referenz auf die Promise hat kann
- Handler Funktionen registrieren
- Handler können das Ergebnis bearbeiten (fulfilled)
- oder Fehler verarbeiten (rejected)
- Es können beliebig viele Handler registriert werden (immutable)
- Auch wenn die Promise bei Registrierung schon settled ist wird trotzdem asynchron der Handler ausgeführt

Sequence Diagram of Promises





- Das Promise Objekt wird von der asynchronen Funktion synchron erzeugt
  aber es wird noch kein value zugewiesen
- Die Promise wird sofort zurückgegeben
- Der Empfänger kann nach Gusto Handler erzeugen und darauf registrieren
- Oder aber die Promise an andere Funktionen weiterreichen
- Die Async Funktion settled die Promise sobald sie fertig ist.
- Wenn die Promise settled ist, werden die passenden Handler ausgeführt.
  genau ein Mal
  immer asynchron
- Die asynchrone Funktion kennt die Handler nicht

Promise Chaining

Multiple promise resolutions can follow one after the other:

- the return value of `p.then()` is a new promise that settles to the return value of the handler
- a call to `p.then()` with no registered handler just returns the underlying promise value wrapped in another promise
- an exception within `p.then()` yields a rejected promise

This behavior enables effective
functional composition called Promise Chaining

Multiple promise resolutions can follow one after the other:

- the return value of `p.then()` is a new promise that settles to the return value of the handler
- a call to `p.then()` with no registered handler just returns the underlying promise value wrapped in another promise
- an exception within `p.then()` yields a rejected promise

This behavior enables effective
functional composition called Promise Chaining

Eine weitere wichtige Eigenschaft

Damit werden Promises erst wirklich spassig

Error Handling with Promises

Error handling gets easy with chained promises

- every exception leads to a rejected promise
- if no rejection handler is registered, `p.then()` just returns a new rejected promise
- unhandled rejections propagate through the promise chain

> *Rejections and errors propagate through promise chains. When one promise is rejected all subsequent promises in the chain are rejected in a domino effect until an onRejected handler is found. In practice, one catch function is used at the end of a chain [. . .] to handle all rejections. [Parker, 2015, S. 20]*

Error handling gets easy with chained promises

- every exception leads to a rejected promise
- if no rejection handler is registered, `p.then()` just returns a new rejected promise
- unhandled rejections propagate through the promise chain

Rejections and errors propagate through promise chains. When one promise is rejected all subsequent promises in the chain are rejected in a domino effect until an onRejected handler is found. In practice, one catch function is used at the end of a chain [...] to handle all rejections. [Parker, 2015, S. 20]

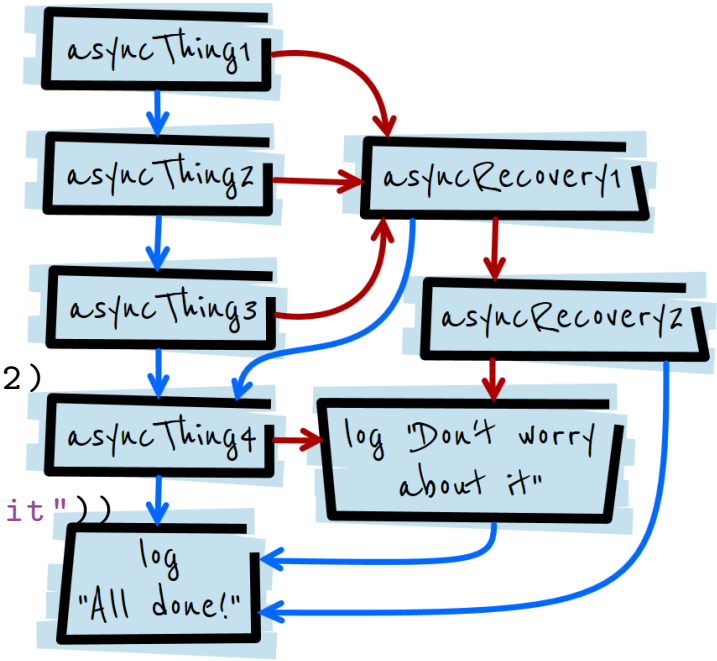
Fehlerbehandlung wird durch Promises sicher

Fehler sprudeln auf bis zum Ende
und können zentral behandelt werden

Es ist gute Praxis am ende einer Promise Chain einen Fehlerhandler anzubringen

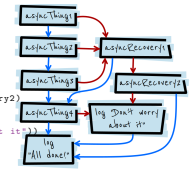
Example: Error Handling in Promise Chains


```
1  //code and illustration taken from
2  //https://developers.google.com/web/
3  //  fundamentals/getting-started/primers/promises
4
5
6  asyncThing1()
7      .then(asyncThing2)
8
9      .then(asyncThing3)
10
11     .catch(asyncRecovery1)
12
13     .then(asyncThing4(), asyncRecovery2)
14
15     .catch(err =>
16        console.log("Don't worry about it"))
17     .then(res =>
18        console.log("All done!"));
19 })
```



Enter the Promised Land
Promises
Mode of Operation
Example: Error Handling in Promise Chains

```
//code and illustration taken from
//https://developers.google.com/web/
//  fundamentals/getting-started/primers/promises


asyncThing1()
   .then(asyncThing2)

   .then(asyncThing3)

   .catch(asyncRecovery1)

   .then(asyncThing4(), asyncRecovery2)

   .catch(err =>
      console.log("Don't worry about it"))
   .then(res =>
      console.log("All done!"));
})
```



Hier ein Konstrukt Promise.catch()
Das ist eine Abkürzung für Promise.then(null, r)
Es wird lediglich der Rejection Handler registriert, kein Fulfillment Handler

Bei Fulfillment wird die Promise einfach durchgereicht.

There are Callbacks All Over

Didn't You just Tell Me This is a Bad thing?

> *You've no doubt noticed that Promises don't get rid of callbacks at all. They just change where the callback is passed to. Instead of passing a callback to foo(..) , we get something (ostensibly a genuine Promise) back from foo(..), and we pass the callback to that somethinginstead. [Simpson, 2015, S. 52]*

- The async function uses a callback to settle the promise
- The calling function registers handler callbacks to handle the result
- Promises still make heavy use of callbacks

The callback is no longer executed by foreign code
The callback execution is guaranteed by the language definition.
They cannot be invoked in an inappropriate way!

You've no doubt noticed that Promises don't get rid of callbacks at all. They just change where the callback is passed to. Instead of passing a callback to foo(..) , we get something (ostensibly a genuine Promise) back from foo(..), and we pass the callback to that somethinginstead. [Simpson, 2015, S. 52]

- The async function uses a callback to settle the promise
- The calling function registers handler callbacks to handle the result
- Promises still make heavy use of callbacks

The callback is no longer executed by foreign code
The callback execution is guaranteed by the language definition.
They cannot be invoked in an inappropriate way!

Der wesentliche Unterschied zur Herausgabe eines Callbacks an eine asynchrone Funktion:

- Der Sprachstandard und die Laufzeitumgebung sorgen zuverlässig dafür, dass von der asynchronen Funktion auf einer Promise entweder resolve() oder reject() aufgerufen werden. Weitere Aufrufe haben keine Wirkung mehr
- Der Handler-Callback wird auf dem Promise-Objekt registriert und hat eine durch den Sprachstandard vorgegebene Semantik
- Der Handler (die Continuation) wird nicht mehr an fremden Code herausgereicht, sondern lediglich auf dem wohldefinierten Promise Objekt registriert.
- Eine Handler-Funktion wird immer asynchorn aufgerufen, niemlas synchron
- Eine Promise ist nach dem Settlement immutable und daher ist es sicher sie als Objekt durch den eigenen Code zu reichen und

Do Promises Solve all of the Callback Problems?

- No continuation is passed to untrustable code.
- Everything happens on the well defined promise object under control of the JavaScript runtime
  - No early calling of handler function
    Guaranteed to be async
  - Easy timeout implementation using `Promise.race(it)`
    No late calling
  - Handler functions execute exactly once
- No silent swallowing of errors
  Exceptions lead to rejected promise
- Structured error handling becomes possible
  Errors are propagated through the promise chain
- No “Pyramid of Doom”. The code gets readable again

Example: Cats Again!

Promises Enable Concise and Easy to Read Code


```
1 function watermarkProm([logo, cat]) {
2     let logoP = loadImage(logo);
3     let catP = loadImage(cat);
4
5     Promise.all([catP, logoP])
6         .then(watermarkHelper)
7         .then(addImg)
8         .catch(errorMessage)
9 }
```

```
function watermarkProm([logo, cat]) {
    let logoP = loadImage(logo);
    let catP = loadImage(cat);

    Promise.all([catP, logoP])
        .then(watermarkHelper)
        .then(addImg)
        .catch(errorMessage)
}
```

Hier ein Konstrukt Promise.race()

Das wartet, bis alle Promises im Array gesettkled sind

We are almost happy with promises,
however, synchronous code is still easier to read.

ECMAScript 6 offers another great feature
to ease the control flow of asynchronous functions:
Generators

Generators enable writing asynchronous code
as if it was synchronous.

Enter the Promised Land

Generators

There's More to Come

We are almost happy with promises,
however, synchronous code is still easier to read.

ECMAScript 6 offers another great feature
to ease the control flow of asynchronous functions:
Generators

Generators enable writing asynchronous code
as if it was synchronous.

Wir haben eigentlich was wir brauchen
Eine sichere Methode zum asynchronen Progerammieren

Und der Talk ist immer noch nicht zu Ende
Sorry!

Es gibt in ECMAScript 6 Generators,
die bei komplexen Kontrollflüssen noch ein bisschen mehr Spaß bringen
und die Lesbarkeit noch weiter verbessern.

- Lots of collections are `Iterables`
- `Iterators` are functions iterating over an `Iterable`
- `Iterators` have a simple interface
  - `it.next()` retrieves the next element `el`
  - `el` has two properties: `el.done` and `el.value`

- Lots of collections are `Iterables`
- `Iterators` are functions iterating over an `Iterable`
- `Iterators` have a simple interface
  - `it.next()` retrieves the next element `el`
  - `el` has two properties: `el.done` and `el.value`

Generators wurden nicht zur Steuerung des Kontrollflusses erfunden

Daher zunächste die graue Theorie

Die für Freunde funktionaler Programmierung mindestens genauso spannend ist.

Generators are special functions implementing the interfaces `Iterable` and `Iterator`

- The values retrieved by `gen.next()` are not predefined but computed on the fly
- New keyword `yield` to send out values
  The generator function pauses at `yield`,
  till the value right to `yield` is retrieved by `gen.next()`
- Additional methods and functionality
  - `gen.next(val)` injects `val` at the position of `yield`
  - `gen.throw(err)` injects an exception
  - `gen.return(val)` ends the generator which returns `val`

Generators behave like “Coroutines” and break the strict Run-to-Completion semantics of JavaScript

Generators are special functions implementing the interfaces `Iterable` and `Iterator`

- The values retrieved by `gen.next()` are not predefined but computed on the fly
- New keyword `yield` to send out values
  The generator function pauses at `yield`, till the value right to `yield` is retrieved by `gen.next()`
- Additional methods and functionality
  - `gen.next(val)` injects `val` at the position of `yield`
  - `gen.throw(err)` injects an exception
  - `gen.return(val)` ends the generator which returns `val`

Generators behave like "Coroutines" and break the strict Run-to-Completion semantics of JavaScript

- Hier noch erklären, dass durch
  - `gen.next(val)` und
  - `gen.throw(err)`
- eine zwei Wege Kommunikation stattfinden kann.
  Exceptions können entweder innerhalb gefangen werden oder beenden den Generator
- Durch das Pausieren werden Endlosschleifen im Generator möglich.
  Das ist wie bei Streams und anderen unendlichen Datenstrukturen in funktionalen Sprachen
- Das Brechen der Run-to-Completion semantics ist nicht schlimm für Synchronisation, da die Abgabe der Kontrolle immer freiwillig erfolgt und nicht präemptiv.
- Shallow Coroutines (seicht, flach)

Example: Generator Counting Up

Generator functions can be recognized by the star*


```
1  //Example of an increment generator with variable steps
2
3  function* increment() {
4      let count = 1;
5      let step = 1;
6
7      while (true) { //generate forever
8          step = (yield count) || step; //yield the count and adjust the step
                if new step provided
9          count += step;
10         console.log("\tgenLog: count: ", count, " step: ", step);
11     }
12 }
13
14 var it1 = increment(); //get the iterator
15 console.log("first value: ", it1.next());
16 console.log("value: ", it1.next(4));
17 console.log("value: ", it1.next(-2));
18 console.log("last value:", it1.return(42));
19 console.log("another it1.next(): ", it1.next());
```

results in

```
1 //first value:  { value: 1, done: false }
2 //  genLog: count:  5  step:  4
3 //value:  { value: 5, done: false }
4 //  genLog: count:  3  step:  -2
5 //value:  { value: 3, done: false }
6 //last value: { value: 42, done: true }
7 //another it1.next():  { value: undefined, done: true }
```

Enter the Promised Land
Generators
Foundation
Example: Generator Counting Up

```
//Example of an increment generator with variable steps

function* increment() {
    let count = 1;
    let step = 1;

    while (true) { //generate forever
        step = (yield count) || step; //yield the count and adjust the step
                if new step provided
        count += step;
        console.log("\tgenLog: count: ", count, " step: ", step);
    }
}

var it1 = increment(); //get the iterator
console.log("first value: ", it1.next());
console.log("value: ", it1.next(4));
console.log("value: ", it1.next(-2));
console.log("last value:", it1.return(42));
console.log("another it1.next(): ", it1.next());
```

results in

```
//first value:  { value: 1, done: false }
//  genLog: count:  5  step:  4
//value:  { value: 5, done: false }
//  genLog: count:  3  step:  -2
//value:  { value: 3, done: false }
//last value: { value: 42, done: true }
//another it1.next():  { value: undefined, done: true }
```

Wir können uns das auch kurz ansehen.

Ist aber nicht spannend

Promises and Generators

Promises and Generators can be combined in a very interesting way:

- On the right hand side of `yield`, a promise is retrieved
- This promise is yielded out to the caller
- The caller registers handlers on the yielded promise
  - When the promise is fulfilled, the handler injects the value back into the generator
  - When the promise is rejected, the handler throws into the generator
- The caller takes the next promise out of the generator
- Or finishes execution when the generator is exhausted

Using this pattern, inside the generator there are async function calls, but left hand of `yield`, the fulfillment values can be used directly like if they were synchronous.

Example: Generated Cats

Yielding Out the Cat from a Generator Using a stupid Runner


```
1  function* watermarkGen([logo, cat]) {
2      let logoPromise = loadImage(logo);
3      let catPromise = loadImage(cat);
4
5      try {
6          let logoImage = yield logoPromise; //1
7          let catImage = yield catPromise; //2
8          let wmImg = yield watermarkHelper([catImage, logoImage]); //3
9          addImg(wmImg); //4
10     } catch (e) {
11         console.log("error occured inside Generator: ", e);
12         errorMessage(e);
13     } finally {
14         console.log('Generator finished');
15     }
16 }
17
18 function watermarkRunner() {
19     let it = watermarkGen([logoName, catName])
20
21     it.next().value                              //1
22         .then(p => return it.next(p).value) //2
23         .then(p => return it.next(p).value) //3
24         .then(p => return it.next(p).value) //4
25         .catch(p => return it.throw(p))
26     console.log('watermarkRunner() function finished');
27 }
```

The Code inside the generator reads like sync and brings back `try-catch-finally` to asynchronous programming.

Enter the Promised Land
Generators
Promises and Generators
Example: Generated Cats

```
function* watermarkGen([logo, cat]) {
    let logoPromise = loadImage(logo);
    let catPromise = loadImage(cat);

    try {
        let logoImage = yield logoPromise; //1
        let catImage = yield catPromise; //2
        let wmImg = yield watermarkHelper([catImage, logoImage]); //3
        addImg(wmImg); //4
    } catch (e) {
        console.log("error occured inside Generator: ", e);
        errorMessage(e);
    } finally {
        console.log('Generator finished');
    }
}

function watermarkRunner() {
    let it = watermarkGen([logoName, catName])

    it.next().value                         //1
        .then(p => return it.next(p).value) //2
        .then(p => return it.next(p).value) //3
        .then(p => return it.next(p).value) //4
        .catch(p => return it.throw(p))
    console.log('watermarkRunner() function finished');
}
```

The Code inside the generator reads like sync and
brings back `try-catch-finally` to asynchronous programming.

in reality do not program out the runner, but use co
this enables code written like sync and is especially useful for complex control flows
code is written in sync including try-catch-finally, but is executed efficiently like async

this pattern is regarded as important that it is awaited (pun intended) in ECMAScript 2017 and can be used now with babel or node.js

Writing Async Code in Sync Style

The Runner function for Generators and Promises is Easily Factored Out

- In real world code, the runner function for a generator yielding promises is easily factored out
- A lot of ready made runner utilities is available; `co` is most popular (see [Holowaychuk, 2013])
- Generators in conjunction with a runner utility enable the programmer to write async code in a synchronous fashion
- This makes complex control flows very concise and readable
- The async advantage does not get lost

The Future of Generators: `async` and `await`

Async control flow gets easy in future ECMAScript versions

The pattern of Promises, Generators and a runner utility was considered important enough, that it will be (most likely) part of the ECMAScript 2017 standard

- `async` and `await` are already reserved keywords
- Async functions implement the presented pattern
- At each `await` keyword, the right hand promise is resolved before the async execution continues
- Promise specialties like `Promise.all()` or `Promise.race()` are still available
- This takes away the burden of an additional runner utility

Example: Async Cats


```
1  async function watermarkAsync([logo, cat]) {
2      try {
3          let logoImage = await loadImage(logo);
4          let catImage = await loadImage(cat);
5          let watermarkedImg = await
               watermarkHelper([catImage, logoImage]);
6          addImg(watermarkedImg);
7      } catch (e) {
8          console.log("error occured inside Generator: ", e);
9          errorMessage(e);
10     } finally {
11         console.log('Async function finally');
12     }
13     console.log('watermarkAsync() function finished');
14 }
```

```
async function watermarkAsync([logo, cat]) {
    try {
        let logoImage = await loadImage(logo);
        let catImage = await loadImage(cat);
        let watermarkedImg = await
            watermarkHelper([catImage, logoImage]);
        addImg(watermarkedImg);
    } catch (e) {
        console.log("error occured inside Generator: ", e);
        errorMessage(e);
    } finally {
        console.log('Async function finally');
    }
    console.log('watermarkAsync() function finished');
}
```

Das liest sich wie synchroner code.
Try-Catch-finally funcktionieren wie gewohnt
Alles ist schön effizient und async.
ABER Beachte: Der “console.log()” kommt natürlich auch erst nach dem “finally” That’s it
SUMMARY

Lots of brave coders got lost fighting "Callback Hell"

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

pikabu.ru

- Promises are a suitable relief and give back control. They
  - are safe to use
  - enable structured error handling

Source: http://cs6.pikabu.ru/images/big_size_comm/2015-07_5/1437839925179784091.jpg

Thank's a Lot!

Source: cat taken from http://wallpapersinhq.online/20777-smiling_kitten_cats_tabby_cat_hd_wallpaper/

Bibliography

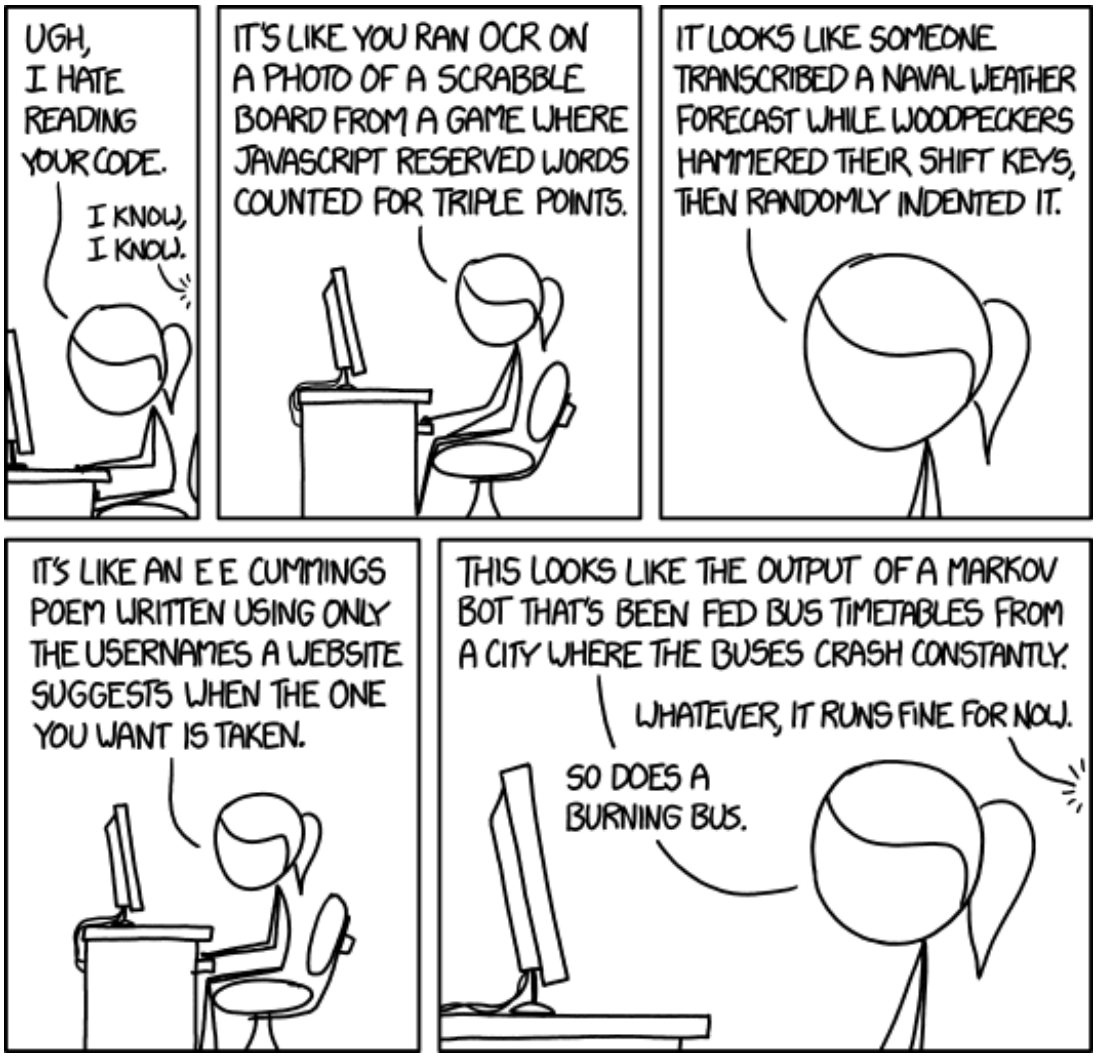
[Ecma TC39 und Wirfs-Brock 2015] Ecma TC39 ; Wirfs-Brock, Allen ; Ecma International (Hrsg.): *ECMAScript 2015 Language Specification*. 2015. – URL `http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf`. – Zugriffsdatum: 2016-12-26

[Holowaychuk 2013] Holowaychuk, T. J.: *co: generator async control flow goodness*. 2013. – URL `https://www.npmjs.com/package/co`. – Zugriffsdatum: 2016-12-03

[Parker 2015] Parker, Daniel: *JavaScript with promises*. First edition. Sebastopol, CA : O'Reilly Media, 2015. – URL `http://proquest.tech.safaribooksonline.de/9781491930779`. – ISBN 9781449373214

[Schlueter 2013] Schlueter, Isaac Z.: *Designing APIs for Asynchrony*. 8 2013. – URL `http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony`. – Zugriffsdatum: 2016-11-25

[Simpson 2015] Simpson, Kyle: *Async & performance*. First edition. Beijing and Cambridge and Farnham and Köln and Sebastopol and Tokyo : O'Reilly, 2015 (You don't know JS). – ISBN 9781491904220



Source: `http://xkcd.com/1695/`