Promises
Asynchronität als Basis

Problem

- JavaScript ist single-threaded

```
// reading a file synchronously const fs =
require('fs');
const data = fs.readFileSync('/file.md'); // blocks here
until file is read
```

- Skripte, die blockierenden IO verwenden sind nicht performant!
- Die meisten JavaScript-Standard-Funktionen erwarten daher Callbacks, die das Ergebnis verarbeiten (kein return Wert):

```
// reading a file asynchronously
const fs = require('fs');
fs.readFile('/file.md', function(err, data){
        if (err) throw err;
});
```

Schlecht lesbarer Code, Programmablauf unklar!

Promises
Ausweg aus der Callback-Hell

```
chooseToppings(function(toppings) {
  placeOrder(toppings, function(order) {
    collectOrder(order, function(pizza) {
      eatPizza(pizza);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

```
chooseToppings()
.then(toppings => placeOrder(toppings))
.then(order => collectOrder(order))
.then(pizza => eatPizza(pizza))
.catch(failureCallback);
```

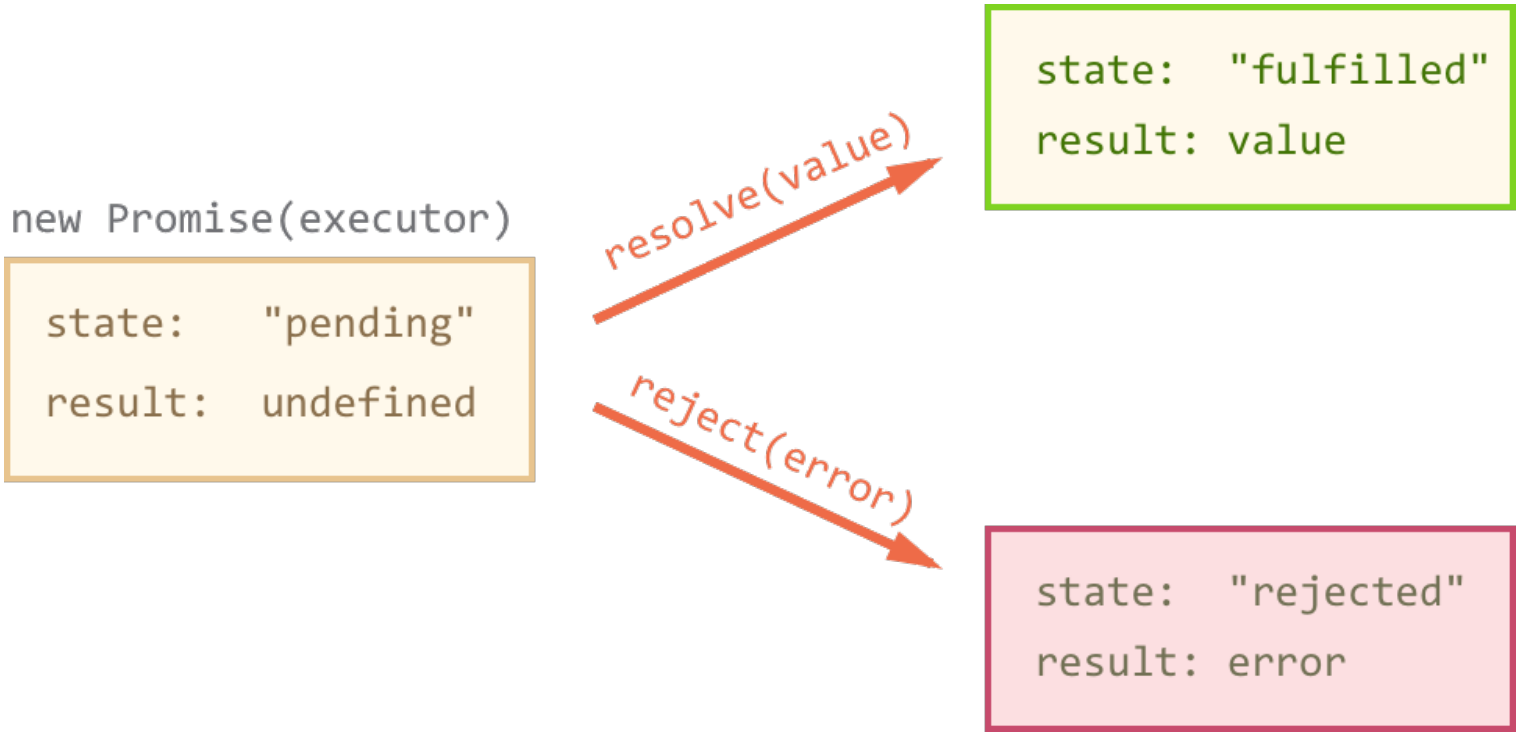
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises

Promises
Stellvertreter/Proxys für ein zukünftiges Resultat

- Das **Promise**-Interface repräsentiert einen Platzhalter (ähnlich zu future)
- Es gibt drei Zustände für den Platzhalter: Pending (Startzustand), Fullfilled und Rejected
- An die Zustandsübergänge werden im Promise zwei Funktionen (callbacks) geheftet
  - `resolve(value)`: erfolgreicher Ausgang - Platzhalter wird mit Wert gefüllt
  - `reject(reason)`: Misserfolg - Platzhalter wird mit Fehlermeldung gefüllt
- Nutzer der Platzhalter können die Zustandsübergänge mit deren Resultaten durch then, bzw. catch abfragen

Syntax
Asynchronität durch Promises

3 Zustände:



https://javascript.info/promise-basics

```
new Promise(executor);
new Promise(function(resolve, reject) { ... });
```

Syntax
Asynchronität durch Promises

```
1  let promise = new Promise(function(resolve, reject) {
2    // the function is executed automatically when the promise is constructed
3
4    // after 1 second signal that the job is done with the result "done"
5    setTimeout(() => resolve("done"), 1000);
6  });
```

https://javascript.info/promise-basics

```
new Promise(executor);
new Promise(function(resolve, reject) { ... });
```

- **executor entspricht einer Callback-Funktion** die üblicherweise zwei Aufrufe hat, für resolve (alles OK) und reject (nicht OK)
- Hinweis: Ein Promise kann als Return-Wert verwendet werden! Daher ist Chaining möglich

https://javascript.info/promise-basics

Syntax
Zugriff auf das Resultat

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});

promise.then(alert); // shows "done!" after 1 second
```

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

https://javascript.info/promise-basics

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);
```

Syntax
from Callback to Promise - Anwendung

```
const promise1 = new Promise(function(resolve, reject) {
  // Promises ermöglichen eine elegantere Verarbeitung
  // asynchroner Ereignisse. Hier ein Timeout
  setTimeout(function() {
    resolve('foo');
  }, 300);
});

promise1.then(function(value) {
  console.log(value);
  // expected output: "foo"
});


console.log(promise1);
// expected output: [object Promise]
```

Syntax
from Callback to Promise - Anwendung

```
const promise1 = new Promise(function(resolve, reject) {
  // Promises ermöglichen eine elegantere Verarbeitung
  // asynchroner Ereignisse
  setTimeout(function() {
    resolve('foo');
  }, 300);
});
```

```
promise1.then(value => console.log(value))
```

```
console.log(promise1);
// expected output: [object Promise]
```

Syntax
Technischer Ablauf

Was ist das Ergebnis auf der Konsole?

```
console.log( 'a' )

const p = new Promise( executor: function ( resolve, reject) {
    console.log( 'b' )
    setTimeout( handler: () => {
        console.log('D')
        resolve ( value: 'E')
    }, timeout: 0 );
} );

// Other synchronous stuff, that possibly takes a very long time to process
p.then(v => console.log(v))
console.log( 'c' )
```

a
b
c
D
E

Syntax
from Callback to Promise - Anwendung

fsreadfile mit Callback:

```
// reading a file asynchronously
const fs = require('fs');
fs.readFile(filePath, function(err, data){
        if (err) throw err;
        console.log('CONTENT:', data);
});
```

Anwendung von Promises am Beispiel von fsreadfile:

```
readFilePromisified(filePath, {encoding: 'utf8'})
.then(data => console.log('CONTENT:', data))
.catch(err => console.log('ERROR:', err))
```

Promises
Die Implementierung ist zum aktuellen Stand nicht ganz leicht

```
const fs = require('fs');

function readFilePromisified(filePath, options) {
  return new Promise(
    function (resolve, reject) {
      let callback = function (err, data) {
        if (err) reject(err); // führt zu .catch(...)
        resolve(data); // führt zu .then(...) }
      };
      fs.readFile(filePath, options, callback);
    });
}
```

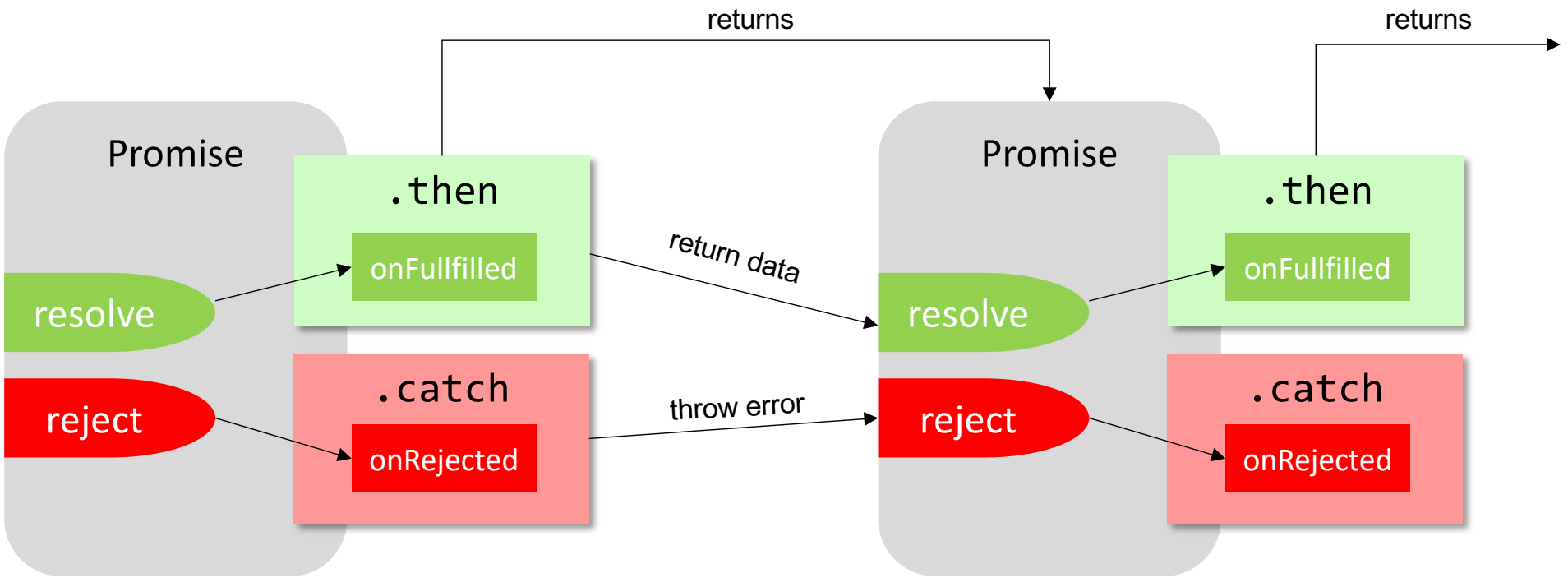
Wir werden noch sehen, dass gerade das Return einer Promise mit neuer Syntax vereinfacht wurde

Was machen eigentlich `resolve` und `reject`?

- `Promise.resolve(value)` und `Promise.reject(reason)` liefern tatsächlich einen Promise
- Formal ist also Chaining möglich
  - Beispiel: `fsProm(…).then(…).then(…).catch(…)`

Syntax
Technischer Ablauf

Chaining: Sowohl `then` als auch `catch` geben neue Promise Objekte zurück



Syntax
Promises kaskadieren

```
readFilePromisified("bla.txt", {encoding: 'utf8'})
.then(function(data){
        console.log('CONTENT:', data);
}).then(function(data){
        console.log("2nd then", data);
})
```

Output:

```
CONTENT: This is the file contents!
2nd then undefined
```

→ Chaining nur halb funktional, da Daten-Parameter nicht durchgereicht werden, console.log ist eben keine promise!

Syntax
Promises kaskadieren

Lösung:

```
readFilePromisified("bla.txt", {encoding: 'utf8'})
.then(function(data){
        console.log ("1st then ", data);
        return Promise.resolve(data);
}).then(function(data){
        console.log("2nd then ", data);
});
```

→ Programmablauf einfacher nachzuvollziehen

Output:

```
1st then This is the file contents!
2nd then This is the file contents!
```

→ Hinweis: Ein return data hätte beim 1. gereicht, da return zu einem Promise wird

Promises statt Callback-Hell

```
chooseToppings(function(toppings) {
  placeOrder(toppings, function(order) {
    collectOrder(order, function(pizza) {
      eatPizza(pizza);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

```
chooseToppings()
.then(function(toppings) {
  return placeOrder(toppings);
})
.then(function(order) {
  return collectOrder(order);
})
.then(function(pizza) {
  eatPizza(pizza);
})
.catch(failureCallback);
```

```
chooseToppings()
.then(toppings => placeOrder(toppings))
.then(order => collectOrder(order))
.then(pizza => eatPizza(pizza))
.catch(failureCallback);
```

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises

Syntax
Asynchronität durch async / await

Verständlicher wird der Code durch das Deklarieren von asynchronen Funktionen

Beispiel:

```
async function read1st() {
  data = await readFilePromisified("bla.txt", { … });
  console.log("1st then ", data);
  return data; //wird zu Promise.resolve(data)
}
async function read2nd() {
  data = await read1st();
  console.log("2nd then ", data);
}

read2nd();
```

Output:

```
1st then This is the file contents!
2nd then This is the file contents!
```

Syntax
Asynchronität durch async/await

Was machen eigentlich `async` und `await`?

- `async` kann als Schlüsselwort vor dem Schlüsselwort `function` verwendet werden und **zeigt an, dass es asynchrone Abschnitte in der Funktion gibt.**
- **Eine async Funktion liefert einen Promise! Ein `return` wird automatisch in ein `Promise.resolve()` gewandelt**
- Innerhalb einer asynchronen Funktion kann `await` genutzt werden um auf einen Promise zu "warten"
- **`await` darf nur in `async functions` verwendet werden.** Ansonsten wird ein `SyntaxError` geworfen
- Um das `catch`-Verhalten eines Promises nachzubilden muss ein `try/catch` benutzt werden

```
async function read1st() {
  try{
    data = await readFilePromisified("bla.txt", { … });
    console.log("1st then ", data);
    return data; //wird zu Promise.resolve(data)
  }catch(e){ throw e }//wird zu Promise.reject(e)
}
```

Achtung: Asynchrone Funktionen sollten trotzdem schnell sein

```
// ---------------------------------------------------------------------
// these functions simulate requests that need to run async
// ---------------------------------------------------------------------
function asyncThing1() {
    return new Promise( executor: (resolve) => {
        setTimeout( handler: () => resolve( value: 'Thing 1 is done!'),  timeout: 2000);
    });
}

function asyncThing2() {
    return new Promise( executor: (resolve) => {
        setTimeout( handler: () => resolve( value: 'Thing 2 is done!'),  timeout: 2000);
    });
}

// ---------------------------------------------------------------------
// this is the code that actually loads data
// ---------------------------------------------------------------------
function doThings() {
    asyncThing1().then((thing1) => {
        // do something with the first response
        console.log(thing1);
    });

    asyncThing2().then((thing2) => {
        // do something with the second response
        console.log(thing2);
    });
}
```

https://www.learnwithjason.dev/blog/keep-async-await-from-blocking-execution#promises-are-a-wonderful-powerful-tool

Achtung: Asynchrone Funktionen sollten trotzdem schnell sein

```
// 🚫 don't do this in your real code; it's slow!
async function doThings() {
    const thing1 = await asyncThing1();
    console.log(thing1);

    const thing2 = await asyncThing2();
    console.log(thing2);
}

doThings();
```

https://www.learnwithjason.dev/blog/keep-async-await-from-blocking-execution#promises-are-a-wonderful-powerful-tool

Promise.all!

```
// ✅ do this — async code is run in parallel!
async function doThings() {
    const p1 = asyncThing1();
    const p2 = asyncThing2();

    const [thing1, thing2] = await Promise.all( values: [p1, p2]);

    console.log(thing1);
    console.log(thing2);
}

doThings();
```

https://www.learnwithjason.dev/blog/keep-async-await-from-blocking-execution#promises-are-a-wonderful-powerful-tool

await – Blockierend?

await kann nur in Funktionen genutzt werden, die mit async ausgezeichnet sind!

(async () => {...} möglich

async Funktionen sind eigentlich Promises und der return-Wert ist ein automatisch fullfiled promise

await ist faktisch eine Schreibweise für ein then()

Damit wird die Thread nicht blockiert!

await ist syntaktischer Zucker für ein Chaining von promises. Anschaulich

werden also callbacks auf die Job-Queue gelegt

await – Blockierend?

```
function foo() {
    console.log("Hi, ");
    return Promise.resolve(1);
}

foo().then(result => console.log(result))
console.log("Volker, ");
```

```
Hi,
Volker,
1
```

```
async function foo() {
    console.log("Hi, ");
    return 1;
}

async function bar() {
    const result = await foo();
    console.log(result);
}

bar();
console.log("Volker, ");
```

await – Blockierend?

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
  // expected output: "resolved"
}

(async() => { await asyncCall();
                console.log("Wow");
            })()

console.log("Hui");
```

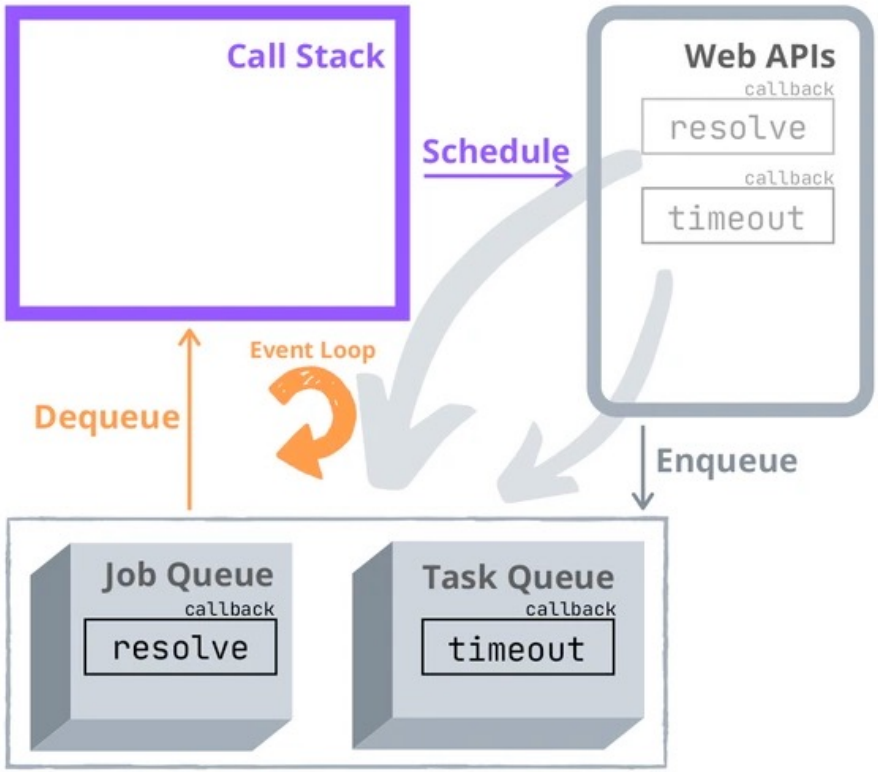
```
> "calling"
> "Hui"
> "resolved"
> "Wow"
```

Promises sind schneller als Timeouts!

```
setTimeout(function timeout() {
    console.log('Timed out!');
}, timeout: 0);

Promise.resolve( value: 1).then(function resolve() {
    console.log('Resolved!');
});

// logs 'Resolved!'
// logs 'Timed out!'
```



https://dmitripavlutin.com/javascript-promises-settimeout/

Promises sind schneller als Timeouts!

```
async function foo() {
    setTimeout(() => console.log('Timed out'), 0);
    await Promise.resolve().then(() =>
console.log('Resolved'))
    console.log('Wird ausgeführt')
}
foo();
console.log('Hello')
```

```
Hello
Resolved
Wird ausgeführt
Timed out
```

Datenbankzugriff

Datenhaltung mittels Dateien nicht zu empfehlen

- Datenbanken!

Die meisten gängigen Datenbanken lassen sich mit Node.js nutzen

- (Oracle) MySQL
- MariaDB
- Oracle Database
- SQLite
- MS SQL
- PostgreSQL
- IBM Informix
- MongoDB (NoSQL)

Wir nutzen SQLite

Weitere Informationen (nicht wirklich…): http://howfuckedismydatabase.com/

Sequelize.js

- Third Party Node.js Modul für Datenbankzugriffe
- **ORM** (Datenbankzugriff ohne SQL-Queries schreiben zu müssen)
- Raw Queries auch möglich (nutzen wir auf Basis von Promises)
- Verbindungsaufbau:

```
const Sequelize = require('sequelize');

// Option 1: Passing parameters separately
const sequelize = new Sequelize('database', 'username', 'password', {
    host: 'localhost',
    dialect: /* one of 'mysql' | 'mariadb' | 'postgres' | 'mssql' */
});

// Option 2: Passing a connection URI
const sequelize = new
Sequelize('postgres://user:pass@example.com:5432/dbname');

// SQLite
const sequelize = new Sequelize({
    dialect: 'sqlite',
    storage: 'path/to/database.sqlite'
});
```

Sequelize.js

- SQL-Abfrage mittels sequelize.query

```
sequelize.query("SELECT * FROM user WHERE id = " + id ,
                        {type: sequelize.QueryTypes.SELECT })
    .then( userinfos => {
        if(userinfos.length != 0){
            console.log(userinfos["0"].user)
        }
        else{
            console.log("no users")
        }
    })
```

- anfällig für SQL Injections
- sollte für Abfragen nicht benutzt werden

Alternative: Prepared Statements

SQL Injection

Exkurs: SQL Injection

- Fehlende Validierung/Maskierung von Benutzereingaben führt zu ungewollten Datenbankstatements
- Beispiel:

	Erwarteter Aufruf
Aufruf	`http://example.com/page?id=42`
Erzeugtes SQL	`SELECT titel, text FROM artikel WHERE ID=42;`

	Angriff mittels SQL-Injection
Aufruf	`http://example.com/page?id=42;DROP+TABLE+user`
Erzeugtes SQL	`SELECT titel, text FROM artikel WHERE ID=42;DROP TABLE user;`

Exkurs: SQL Injection

- Zweites Beispiel:

	Erwarteter Aufruf

Aufruf	`http://example.com/login (POST)`
	
	`user=admin&pw=honigkuchen`

Erzeugtes SQL	`SELECT id FROM user WHERE name = 'admin'` `  AND pw = 'cee199b741247512eb298a34c1fa18f5ca704bae'`

	Angriff mittels SQL-Injection

Aufruf	`http://example.com/login`
	
	`user=admin'+OR+1='1'--&pw=asdf`

Erzeugtes SQL	`SELECT id FROM user WHERE name = 'admin' OR 1='1'` `  --AND pw = '3da541559918a808c2402bba5012f6c60b27661c'`

Prepared Statements

Prepared Statements

- Vorbereitete Anweisung wird an die Datenbank gesendet
  - > Platzhalter für Variablen

- Ausführung des Statements nach Einsetzen der Variablen
  - > Datenbanksystem kümmert sich um das Escapen
  - > Verhindert SQL-Injections und prüft die Gültigkeit von Parametern

- Geschwindigkeitsvorteil bei mehrfachen Ausführen
  - > Statement liegt dem Datenbanksystem bereits vor

Sequelize.js
Prepared Statements

„Replacements“ (ggf. mit await davor)

- Ersetzung des Parameters per
  - „?“-Notation (unnamed Parameter)

  oder
  - „:“-Notation (named Parameter)

```
sequelize.query("SELECT * FROM user WHERE id = ?" ,
                    {replacements: [req.params.id],
                     type: sequelize.QueryTypes.SELECT })
    .then( userinfos => {
        if(userinfos.length != 0){
            console.log(userinfos["0"].user)
        }
        else{
            console.log("no users")
        }

    })
```

Sequelize.js

„:"-Notation

```
sequelize.query("SELECT * FROM user WHERE id = :id" ,
                {replacements: { id: req.params.id},
                 type: sequelize.QueryTypes.SELECT })
    .then( userinfos => {
        if(userinfos.length != 0){
            console.log(userinfos["0"].user)
        }
        else{
            console.log("no users")
        }

    })
```

Sequelize.js

Array-Replacements

```
sequelize.query("SELECT * FROM user WHERE id IN(:ids)" ,
                {replacements: { ids: [1, 2, 3, 4]},
                 type: sequelize.QueryTypes.SELECT })
    .then( userinfos => {

        […]

    })
```

%-Operator

```
        replacements: { search_name: 'an%'}
```

> Suche aller Namen, die mit ‚an‘ beginnen

Sequelize.js
Gerne auch mit await for der Query

```
const results = await sequelize.query("SELECT * FROM " +
                "user WHERE id = ?" ,
                   {replacements: [req.params.id],
                    type: sequelize.QueryTypes.SELECT })
console.log(results) // gibt die Zeilen aus
```

```
const [results, metadata] = await sequelize.query("…")

// metadata gibt bei einigen DB-Systemen zusätzliche

// Informationen an
```

Sequelize.js
ORM!!!

```
const Note = sequelize.define('notes',
    { note: Sequelize.TEXT, tag: Sequelize.STRING })

sequelize.sync({ force: true })
   .then(() => {
       console.log(`Database & tables created!`)
       Note.bulkCreate([
          { note: 'after work party', tag: ‘fun' },
          { note: 'Scrum Meeting', tag: 'work’ },
          ]).then(function() {
              return Note.findAll(); }).then(
              (notes) => console.log(notes})
       };
app.get('/notes', function(req, res) {
       Note.findAll().then(notes => res.json(notes)); });

app.get('/notes/:id', function(req, res) {
       Note.findAll({ where: { id: req.params.id }
       }).then(notes => res.json(notes)); });
```

JavaScript im Einsatz

Prototypen und ECMAScript

JS Exceptions

- Standardweg für Exception Handling
- Wenn ein Problem auftritt, wird eine Exception geworfen
- Klassische Statements: `throw/try/catch`
- **Erzeugung** über einen exception constructor (optionaler Übergabeparameter: `message`)
  - `const rangeException = new RangeError([message]);`
  - `const syntaxException = new SyntaxError([message]);`
- **Werfen** der Exception per throw:
  - `throw rangeException;`

JS Exceptions

- Gängige Exception-Typen in Javascript
  - `Error` – genereller Fehler (Basis-Klasse)
  - `ReferenceError` – nicht valide Referenz
  - `TypeError` – nicht valider Typ
  - `SyntaxError` – falsche Syntax von Quellcode
  - `RangeError` – Parameter außerhalb valider Grenzen
- Eigenschaften der Error-Klasse:
  - `name`, `message` (Beschreibung der Fehlermeldung), `stack` (stack trace)

JS Exceptions

- Exception Handling
  1. Fangen der Exception
     - `try-catch`
  2. Fehlerbehebung
     - Anzeigen der Fehlermeldung; Wiederholen, Default-Aktion ausführen
  3. Ausführung der Applikation fortsetzen
     - Applikation wird in der ersten Zeile nach dem `catch`-Block fortgeführt

```
try {
    // code that can throw an exception
} catch (ex) {
    // this code is executed in case of exception
    // and ex holds the info about the exception
}
```

Try-Blöcke werden oftmals nicht von der JS-Engine optimiert, so dass hier Besser keine durchsatzkritischen Maßnahmen getroffen werden sollten

JS Exceptions

Handlen von mehreren unterschiedlichen Exceptions

- **Jedes try-catch kann nur einen catch-Block enthalten**
- Filtern des Exception-Types wie in Java oder C++ nicht möglich (multiple Catches – abhänge catch-Abschnitte sind nicht Teil der Standardisierung und sollten nicht verwendet werden)
- Die ECMAScript-Spezifikation erlaubt nur folgende Syntax:

```
try {
    // Some code causing different types of exceptions
} catch (ex) {
    if (ex instanceof TypeError) { … }
    else if (ex instanceof ReferenceError) { … }
    else if (ex instanceof SyntaxError) { … }
}
```

Vererbung in JavaScript ist anders
Konstante können geändert werden

```
const vs = {name: "Volker Sander", beruf: "Professor"}

vs.name = "V.Sander"

console.log(vs)
```

Vererbung in JavaScript ist anders
Prototypen

Klassenlose Vererbung in JavaScript

- Über Prototypen (Objekte erben von Objekten)
  - > Jedes Objekt und jede Funktion besitzt in JavaScript ein privates Attribut welches auf das sogenannten Prototypobjekt zeigt
  - > Auch das Prototypobjekt besitzt ein solches, bis auf die Mutter aller Objekte, das Object-Objekt. Hier ist das Attribut null
  - > Ein Prototypobjekt kann Attribute und Methoden(Funktionen) haben
  - > Vererbung wird anschaulich auf Basis der Verlinkung der Prototypobjekte realisiert

Basis für unsere Klassen, Objekte sind zunächst Funktionen

Funktionen können über die Prototypen Attribute erhalten

Objekte können damit Attribute und Funktionen vom Prototypobjekt „ererben“

Prototypen
Basis für Vererbung

Klassenlose Vererbung in JavaScript

- Über Prototypen (Objekte erben von Objekten)
  - > Beim Erstellen eines neuen Objektes mittels „new“ wird immer die **Verlinkung mit dem Prototypen** hergestellt
  - > Im **Prototypobjekt können jederzeit Attribute und Methoden gespeichert werden**, auf die auch neu erzeugte Objekte zugreifen können, als ob sie unmittelbar Eigenschaften von ihnen selbst sind.
  - > Auch Prototypen haben einen Prototypen, wobei das Object-Objekt keinen Prototypen hat (null)
    - – Es ergibt sich eine „Prototype-Chain“

Zugriff auf ein Attribut – Ablauf

- Beispiel: **alert(obj.a);**
  - > Besitzt obj ein Attribut a? Falls ja, nehmen wir den Wert
    - – Sonst: Besitzt der Prototype von obj ein Attribut a? Falls ja, nehmen wir den Wert
      - • Sonst: Besitzt der Prototype des Prototypes von obj ein …
  - > Solange durchführen bis ein Attribut a gefunden ist (sonst undefined)

Prototypen
Zugriff auf die Prototypen

Prototype und __proto__

- **[Funktion]**.prototype
  - > „Magisches“ Attribut prototype, das jede Funktion besitzt
  - > Grundlage für die Vererbung
  - > Enthält gemeinsam nutzbare Methoden und Attribute, auf die die neu erzeugten Objekte zugreifen können
- **[Objekt]**.__proto__
  - > „Magisches“ Attribut __proto__, das jedes Objekt besitzt
  - > Referenziert die Eigenschaft prototype des Objekts (nach Erzeugung mittels new)

Zugriff auf ein Attribut – Ablauf

- Beispiel: `alert(obj.a);`
  - > Besitzt obj ein Attribut a? Falls ja, nehmen wir den Wert
    - – Sonst: Besitzt der obj.__proto__ ein Attribut a? Falls ja, nehmen wir den Wert
      - • Sonst: Besitzt obj.__proto__.__proto__ ein …
  - > Solange durchführen bis ein Attribut a gefunden ist (sonst undefined)

Weitere Informationen: https://developer.mozilla.org/de/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

Einsatz
Prototypen

```
let f = function () {
  // .this ordnet die Attribute dem aufrufenden Objekt zu
  this.a = 1;
  this.b = 2;
}
let o = new f(); // {a: 1, b: 2}
f.prototype.b = 3;
f.prototype.c = 4;
console.log(o.a); // 1
console.log(o.b); // 2 und nicht etwa 3! Suchhierarchie!!!
console.log(o.__proto__.b)
console.log(o.c); // 4
console.log(o.d); // undefined
f.e = 5; // ist jetzt ein Attribut der Funktion f und nicht
des Objekts!
console.log(f.e); // 5
console.log(o.e); // undefined
```

Syntax
Prototypen

```
let f = function () {
  this.a = 1;
  this.b = 2;
}

let o = new f(); // {a: 1, b: 2}

f.prototype.summe = function () {
  return this.a + this.b;
};

console.log(o.summe());  // 3
```

Anwendung bei der Vererbung

```
function Fahrzeug(speed) {
    this.speed = speed; // Eigenschaften werden an das
    this.distance = 0;  // Objekt selber gebunden
}

// Funktion für Objekte des Typs Fahrzeug
Fahrzeug.prototype.go = function(times) {
    this.distance += this.speed * times;
};

// porsche.__proto__ === Fahrzeug.prototype (auch für opel)
var porsche = new Fahrzeug(260);
var opel    = new Fahrzeug(140);
opel.go(2);

alert(opel.distance);                    // 280
```

Anwendung bei der Vererbung

```
function Fahrzeug(speed) {
    this.speed = speed;
    this.distance = 0;
};
Fahrzeug.prototype.go = function(times) {
    this.distance += this.speed * times;
};
var fahrzeug1 = new Fahrzeug(120);
fahrzeug1.go(2);

function Auto(speed, fabrikat) {
    Fahrzeug.apply(this, arguments);
    this.fabrikat = fabrikat;
}

// Wir erzeugen ein Objekt, dass die prototype-Funktionen von Fahrzeug enthält
Auto.prototype = new Fahrzeug();
Auto.prototype.hupen = function() { // Wir hängen eine weitere Funktion an
    alert('Möööp!');
}

var auto = new Auto(150, 'Corsa');
auto.hupen();          // Gibt aus: 'Möööp!'
auto.go(1);
alert(auto.distance); // 150
```

Anwendung bei der Vererbung

Object.create

- Einfache Möglichkeit Prototyping zu nutzen
- Beispiel 1

```
var foo = {
    eins : 1,
    zwei : 2,
    pi : 3.14159,
};

var bar = Object.create(foo); //Objekt mit foo als Prototyp
bar.drei = 3;

// Eigenschaften von bar:
bar.eins;   // 1 (vom Prototyp geerbt)
bar.zwei;   // 2 (vom Prototyp geerbt)
bar.drei;   // 3
bar.pi;     // 3.14159 (vom Prototyp geerbt)
```

Object.create

- Einfache Möglichkeit Prototyping zu nutzen
- Beispiel 2

```
var foo = {
    eins : 1,
    zwei : 2,
    pi : 3.14159,
};

var bar = Object.create(foo);//Objekt mit foo als Prototyp
bar.drei = 3;
bar.pi   = 4;
// Eigenschaften von bar:
bar.eins;   // 1 (vom Prototyp geerbt)
bar.zwei;   // 2 (vom Prototyp geerbt)
bar.drei;   // 3
bar.pi;     // 4 (Eigenschaft vom Prototypen überdeckt)
```

Anwendung bei der Vererbung

Object.create: Beispiel 2

```
var obj1 = {
    a : 1
};
var obj2 = Object.create(obj1);
obj2.b = 2;
var obj3 = Object.create(obj2);
obj3.c = 3;


obj1;   // {a: 1 }
obj2;   // {a: 1, b: 2 }
obj3;   // {a: 1, b: 2, c: 3 }
```

Object.create

```
Object.create(proto [, propertiesObject])
```

- Erstellt ein Objekt mit *proto* als __proto__
  - > Wird *null* übergeben, so wird ein Objekt ohne Prototyp erstellt
- Per *propertiesObject* können Eigenschaften des Objektes, sowie Getter- und Setter-Funktionen bestimmt werden
  - > Hier zunächst nicht weiter berücksichtigt

Welche Möglichkeiten kennen wir nun Objekte zu erstellen?

- Object.create

```
var obj = Object.create(null);
obj.foo = 1;
obj.bar = 2;
```

- JSON-like-Notation

```
var obj = {
    foo : 1,
    bar : 2
};
```

- Konstruktorfunktion

```
var obj = new Object();
obj.foo = 1;
obj.bar = 2;
```

Modernes JavaScript

ES6/ES7, Event Loop

ECMAScript

JavaScript basiert auf ECMAScript-Standard

- Ständige Weiterentwicklung durch Gruppe aus Browserherstellern
  - > Meist oberstes Gebot: Abwärtskompatibilität
- JavaScript erweitert ECMAScript um Features, die nicht im Standard vorgesehen sind

- Heute werden Features oft in Browsern integriert bevor der Standard diese überhaupt vorsieht
  - > Erlaubt einfachere Erkennung von Problemen bevor der Standard festgehalten wird
  - > Aber Entwickler müssen nicht standardisierte APIs benutzen

ECMAScript

Entwicklung

Edition	Release	Name
1	1997	Zuerst „Mocha“, dann „LiveScript“, dann „JavaScript“ genannt
2	1998	Kleinere Änderungen
3	1999	Diverse Erweiterungen (z.B. Regular Expressions)
4	-	Nie veröffentlicht
5	2009 (!)	Einführung des strict modes, Fokus auf Kompatibilität
6	2015	Klassen, Arrow functions, Viele Erfahrungen aus Dialekten (z.B. CoffeeScript) übernommen
7	2016	Weiterentwicklung der Sprache, await async
8	2017	Parallelität und Shared Memory für Worker: atomics

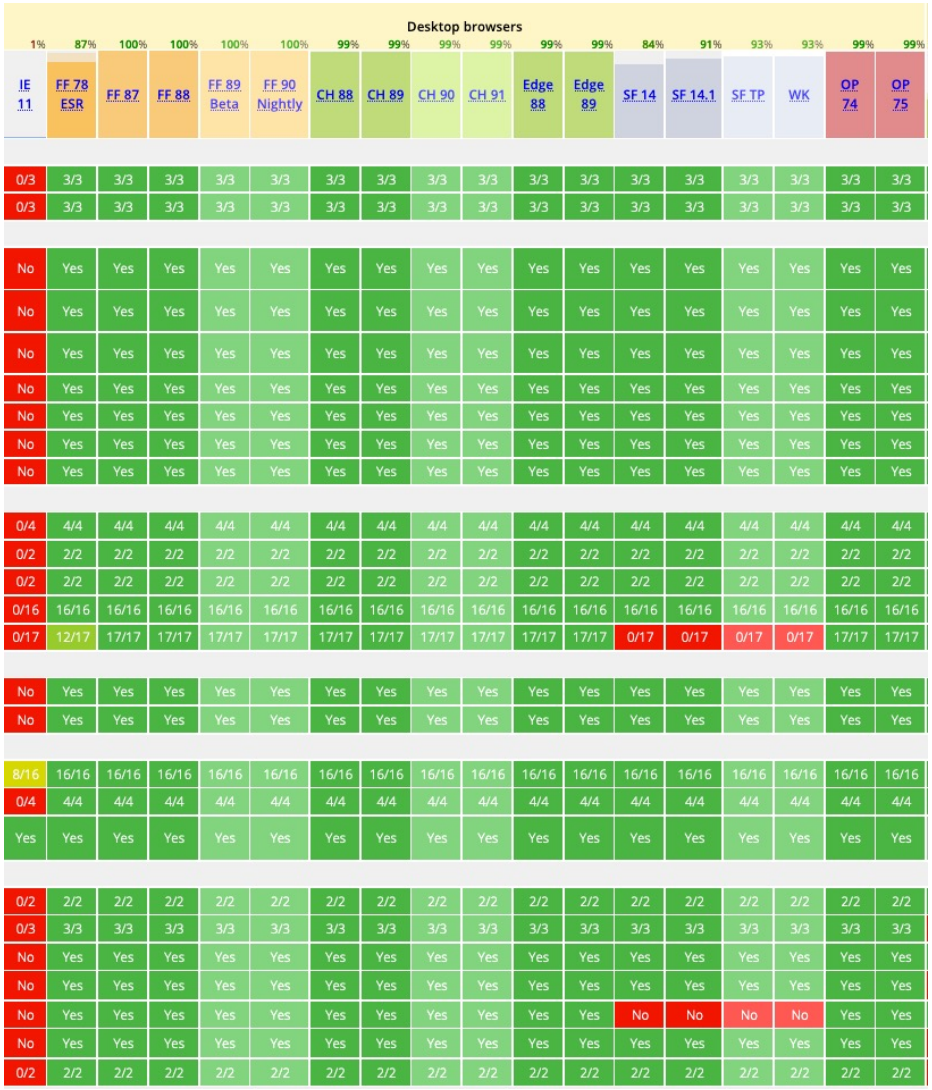
ECMAScript 7

Veröffentlicht Juni 2016

- Viel geplant, am Ende wenig Neues
  - > Math.pow-Kurzschreibweise: x ** y
  - > Array.prototype.includes
  - > Async await als wesentliche Neuerung
- Für die nächste Version mehr Features geplant

→ **bisher passabler Browser Support**

(Stand Mai. 2021)



Quelle: http://kangax.github.io/compat-table/es2016plus/

Transpilation: Babel
Babel is a JavaScript compiler.

BABEL JS

https://babeljs.io/

- It turns ES2015:

```
const adding = (a, b) => a + b
```

- into old JavaScript:

```
'use strict';
var adding = function adding(a, b) {
    return a + b;
};
```

Und im Zusammenhang mit Node.js...

https://www.toptal.com/nodejs/top-10-common-nodejs-developer-mistakes

Mistake #1: Blocking the event loop
Mistake #2: Invoking a Callback More Than Once
Mistake #3: Deeply Nesting Callbacks
Mistake #4: Expecting Callbacks to Run Synchronously
Mistake #5: Assigning to “exports”, Instead of “module.exports”
Mistake #6: Throwing Errors from Inside Callbacks
Mistake #7: Assuming Number to Be an Integer Datatype
Mistake #8: Ignoring the Advantages of Streaming APIs
Mistake #9: Using Console.log for Debugging Purposes
Mistake #10: Not Using Supervisor Programs