

thecodecampus</>
w1k



JavaScript von Morgen
schon heute

- Ausgangssituation
- Setup
- Features

ÜBER MICH

- Philipp Burgmer
 - Software-Entwickler, Trainer
 - Fokus: Frontend, Web-Technologien
 - burgmer@w11k.de
- w11k GmbH
 - Software Design, Entwicklung & Wartung
 - Consulting, Schulungen & Projekt Kickoff
 - Web-Apps, Mobil-Apps, Rich Clients
 - AngularJS, TypeScript, Eclipse RCP

AUSGANGSSITUATION

ECMAScript 5

- Standard seit 2009
- Kompabilität
 - Aktuelle Browser 100%
 - IE8 11% ohne Polyfill
 - IE9 97% (nur kein strict-Mode)
 - Node 100%
 - io.JS 100%

Quellen:

- [ES5 Compat Table](#)

AUSGANGSSITUATION

ECMASCRIPT 6

- Standard wurde Juni 2015 verabschiedet (ES 2015)
- Implementierungen schon gegen Entwurf
- Kompabilität
 - IE10 4%
 - Chrome 43 41%
 - Firefox 38 60%
 - Edge 63%
 - Node 19%
 - io.JS 39%

Quellen:

- [ES6 Compat Table](#)

AUSGANGSSITUATION

ECMASCRIPT 7

- Ferne Zukunft? Auch ES 2016 genannt!
- Entwürfe und Vorschläge
- Interessante neue Features
 - Pattern Matching
 - Traits
 - Guards und Trademarks
 - Object.observe

PROBLEME

- Alte Umgebungen / Browser
 - IE8 immer noch relevant
 - IE9 wird uns noch Jahre begleiten (derzeit > IE10 [1])
 - Node 0.10?
- Nicht alle Engines implementieren alles gleich
Verschiedene Versionen von Proposals und Drafts

Quellen:

- [1] <https://www.netmarketshare.com/browser-market-share.aspx?qprid=2&qpcustomd=0>

NEUE FEATURES IN ES

- Neu in Standard-Bibliothek (Promises, Maps, Sets, ...)
- Syntactic Sugar (Klassen, Module, ...)
- Erweiterungen der Sprache und Engine
 - Typed Arrays
 - Proxies
 - Erben von vordefinierten Konstruktor-Funktionen
 - ...

SHIMS / POLYFILLS

- Rüsten fehlende Teile der Standard-Bibliothek nach
- es-shims
 - es5-shim (44%)
 - es6-shim (17%)
 - es7-shim (33%)
- core-js (48% von ES6)
 - *Modular compact standard library for JavaScript*
 - Shims für ES5, ES6 und ES7 proposals
 - dictionaries, extended partial application, console cap, date formatting
- ...

Quellen:

- ES Compat Table
- core-js (ohne Babel) Wert über eigene Erweiterung von *ES6 Compat Table*

TRANSPILER

- Übersetzen *Syntactic Sugar* von ES6 in ES5 Code
- Bringen meist Polyfills gleich mit
- Google Traceur
- Babel (früher 6to5)
- Microsoft TypeScript

TypeScript

- Viele ES6 Features + viele weitere
 - Optionale statische Typen zur Compile-Zeit
 - Klassen, Module
 - Generics, Interfaces, Mixins
- Interoperabel da Superset von ES5
- Bewegt sich mit 1.5 und 1.6 mehr zum ES6 Standard (z.B. Module)

TypeScript

- Für neues Projekt sehr zu empfehlen
- Migration teilweise schwierig
 - Nicht jeder ES5 Code ist gültiger TypeScript Code (Type Inference)
 - Aber jeder ES5 Code kann in TypeScript eingebunden werden (separate Datei mit Typ-Deklarationen notwendig)

SETUP

SETUP

PACKAGE MANAGER

- Optional!
- Lädt Abhängigkeiten von externen Quellen
 - Für Entwicklung auf lokalen Rechner
 - Für Betrieb auf Server
- Verwaltet Versionsnummern und Konflikte
- Bower, NPM, JSPM, ...

SETUP

MODULE LOADER

- Optional!
- Läuft im Browser/Client
- Läd benötigten Code vom Server in den Client
- Nicht Teil von ES6, extra Standard/Entwurf [1]
- RequireJS, Browserify, SystemJS, WebPack, ...
- Bundling nicht vergessen

Quellen:

- [1] [ES6-Module-Loader Polyfill](#)

SETUP

TRANSPILER

- Optional!
- Übersetzt ES6 in ES5
- Dynamisch im Client, on demand
- Kann in Module-Loader integriert werden
- Statisch im Build-Prozess

SETUP

DEMO

ES6 FEATURES

UND WIE SIE UNSER LEBEN VEREINFACHEN

PROMISES

- Besserer Umgang mit Asynchronität
- Callback-Hell vermeiden
- Weit verbreitet als Bibliotheken

"Pyramid of Doom"

```
1 step1(function (value1) {  
2   step2(value1, function(value2) {  
3     step3(value2, function(value3) {  
4       // Do something with value3  
5     });  
6   });  
7 });
```

Flache Struktur mit Promises

```
1 promisedStep1()  
2 .then(promisedStep2)  
3 .then(promisedStep3)  
4 .then(function (value3) {  
5   // Do something with value3  
6 })
```

PROMISES

KRIS KOWAL'S Q

- Weit verbreitete Bibliothek (z.B. in AngularJS)
- Andere API als ES6 Promises: Promise & Deferred
- In AngularJS 1.3 `$q` als Factory-Funktion, ähnlich wie ES6

Deferred API of Q

```
1 function asyncFunc() {  
2   var deferred = Q.defer();  
3   setTimeout(function () {  
4     deferred.resolve('result'); // or deferred.reject('error');  
5   }, 500);  
6  
7   return deferred.promise;  
8 }
```

Factory API of \$q in AngularJS

```
1 function asyncFunc() {  
2   return $q(function(resolve, reject) {  
3     $timeout(function () { resolve(); /* or reject(); */ }, 500);  
4   });  
}
```

PROMISES

ES6

- Keine Bibliothek notwendig, Teil der Standard-Bibliothek
- Vordefinierter Konstruktor

```
1 function asyncFunc() {  
2   return new Promise(function(resolve, reject) {  
3     setTimeout(function () {  
4       resolve(); // or reject();  
5     }, 500);  
6   })  
}
```

MODULE

IN ES5

- Kein natives Modul-System
- Community Alternativen: AMD und CommonJS
- Viele Module-Loader (RequireJS, Browserify, SystemJS, Webpack, ...)
- Nicht jede Bibliothek für jedes Modul-System vorhanden

MODULE

IN ES6

- Natives Modul-System (Beste aus beiden)
 - Statisches und dynamisches Laden von Code
 - Kontext für Variablen, keine globalen Variablen mehr
 - Ein Modul pro Datei, eine Datei pro Modul
 - Struktur auf Datei Ebene
 - Zirkuläre Abhängigkeiten erlaubt
- Viele Bibliotheken noch nicht als ES6 Modul verfügbar

ES6 MODULE

NAMED EXPORT & IMPORT

lib.js mit Named-Export

```
1 var hidden = 0;  
2 export function state() {  
3   return hidden++;  
4 }
```

Gezielt importieren per Namen

```
1 import { state } from './lib';  
2 state(1);
```

Alles importieren mit Namespace

```
1 import * as lib from './lib';  
2 lib.state(1);
```


ES6 MODULE

DEFAULT EXPORT & IMPORT

- Export `lib.js`

```
1 var hidden = 0;  
2 export default function() {  
3   return hidden++;  
4 }
```

- Import

```
1 import state from './lib';  
2 state(1);
```

MODULE

ANGULARJS

- AngularJS 1.x hat eigenes Modul-System
- Hat praktisch keinen Mehrwert (mehr) [1]
- Wird man nicht los, zumindest nicht ganz
- Zwei Möglichkeiten
 - Modul-System über Modul-System
 - AngularJS Module-System weitestgehend ignorieren

Infos:

- [1] [This AngularJS modules/dependencies thing is a lie](#)

MODULE

ANGULARJS

ES6 Modul-System über AngularJS Module-System

```
1 import angular from 'angular';
2 import 'angular-ui-router';
3
4 import {homeIndexModule} from './index/home-index';
5 import {authRequiredModule} from 'app/routes/auth-required';
6
7 export var homeModule = angular.module('app.route.home', [
8   'ui.router',
9   authRequiredModule.name,
10  homeIndexModule.name
11 ]);
12 homeModule.config(function homeRoute($stateProvider) {
13   $stateProvider.state('app.auth-required.home', {
14     abstract: true,
15     template: '<ui-view></ui-view>'
16   });
17 }
18 );
```

Quellen:

- [gocardless/es6-angularjs](https://github.com/gocardless/es6-angularjs)

MODULE

ANGULARJS

Nur ein AngularJS Modul

```
1 import angular from 'angular';
2 import 'angular-ui-router';
3
4 export var app = angular.module('app', ['ui.router']);
```

ES6 Modul-System statt AngularJS Modul-System

```
1 import ngModule from 'app/module';
2 import 'angular-ui-router';
3
4 ngModule.config(function ($stateProvider) {
5     $stateProvider.state('app.auth-required.home', {
6         abstract: true, template: '<ui-view></ui-view>'
7     });
8 }
9 );
```

KLASSEN

- Andere Schreibweise für existierende Funktionalität
- Kann fast vollständig in ES5 übersetzt werden
- Vererbung per Transpiler nicht zu 100% möglich

Klasse in ES6

```
1 class Point {  
2   constructor(x, y) { this.x = x; this.y = y; }  
3   toString() { return `(${this.x}, ${this.y})`; }  
4 }
```

Klasse übersetzt in ES5

```
1 var Point = (function () {  
2   function Point(x, y) { this.x = x; this.y = y; }  
3   Point.prototype.toString = function () {  
4     return "(" + this.x + ", " + this.y + ")";  
5   };  
6   return Point;  
7 })();
```

SERVICES IN ANGULARJS

OHNE KLASSEN

- Services in AngularJS: allgemeine Objekte, die Funktionalität kapseln
- Werden vom Framework instanziiert und verwaltet

Module#service mit Konstruktor-Funktion

```
1 angular.module("myModule").service("myService", function ($http) {  
2     this.doSomething = function () { };  
3 });  
4 // service = new myService();
```

Module#factory mit Factory-Funktion

```
1 angular.module("myModule").factory("myService", function ($http) {  
2     var service = {}; // var service = function () {};  
3     service.doSomething = function () { };  
4     return service;  
5 });  
6 // service = myService();
```

SERVICES IN ANGULARJS

MIT KLASSEN

Service in AngularJS mit ES6 Klasse

```
1 var http = Symbol();
2 class MyService {
3
4   constructor($http) {
5     this[http] = $http; // pseudo private
6   }
7
8   doSomething() {}
9 }
10 MyService.$inject = ['$http'];
11 angular.module("myModule").service("myService", MyService);
```

ARROW FUNCTIONS

- Kurzschreibweise für Funktionen
- `(param) => { statement; }` statt `function (param) { statement; }`
- Aber: kein eigener lexikalischer Kontext, `this` vom äußeren Kontext

Funktion mit gleichem Kontext

```
1 class Person {  
2   constructor(name, friends) { this.name = name; this.friends = friends; }  
3   printFriends() {  
4     this.friends.forEach(friend =>  
5       console.log(this.name + " knows " + friend));  
6   }  
7 }
```


ARROW FUNCTIONS

ANWENDUNGSFÄLLE

- Promise Handler
- Event Handler

Funktion mit gleichem Kontext

```
1 function TestCtrl ($rootScope, DataService) {  
2     // no more var that = this;  
3     this.data = []; this.lastEvent = undefined;  
4     DataService.get().then(data => { this.data = data; });  
5     $rootScope.$on('eventName', event => { this.lastEvent = event.name});  
6 }  
7 angular.module('myModule').controller('TestCtrl', TestCtrl)
```

UND NOCH EIN BISSCHEN
TypeScript

STATISCHE TYPEN

- Typen nur zur Entwicklungszeit vorhanden
- Werden von Compiler gelöscht
- Vorteile: Refactorings, Code Completion und Navigation
- Basis Type: Boolean, Number, String, Array, Enum, Any, Void
- Eigene Typen über Klassen und Interfaces

Compile-Error

```
1 function greeter(person: string) {  
2     return "Hello, " + person;  
3 }  
4  
5 var user = [0, 1, 2]; // user: array  
6  
7 document.body.innerHTML = greeter(user);
```

TYPE INFERENCE

- Typen müssen nicht überall angegeben und wiederholt werden
- Compiler kennt Typen aus Kontext

```
1 var x = 3; // x: Number
2
3 window.onmousedown = function(mouseEvent) {
4     console.log(mouseEvent.buton); // Error
5 };
```

STRUKTURELLE TYPEN

- Auch *Duck-Typing* genannt
- Typen sind kompatibel wenn geforderte Properties da sind
- *Was ist da, und nicht woher kommt es*

```
1 interface Named { name: string; }  
2 class Person { name: string; }  
3  
4 var p: Named;  
5 // OK, because of structural typing  
6 p = new Person();
```

TypeScript

FEATURES ZUSÄTZLICH ZU ES6

- Interfaces
- Generics
- Mixins
- Optionale und Default-Parameter
- Enums

NOCH MEHR DAZU ...

- Exploring ES6 von Axel Rauschmayer
- Deploying ES6 und Using ES6 today
- ES6 Features (kurz und bündig)
- TypeScript Dokumentation

Philipp Burgmer
burgmer@w11k.de

www.w11k.de
www.thecodecampus.de