BACK TO JAVASCRIPT

NOW: ECMASCRIPT 6

INHALT

- Was war das nochmal?
- Scoping
- Arrow Functions
- Extended Parameter Handling
- Template Strings
- Enhanced Object Properties
- Internationalization & Localization

WAS WAR ECMASCRIPT NOCHMAL?

ECMASCRIPT (1997) VON ECMA INTERNATIONAL

- European Computer Manufacturers Association
- Scripting Language Specification in ECMA-262
- ECMAScript 6: Juni 2015
- WICHTIG: mindestens NodeJS Version 6.x muss istalliert sein
- (Besserer) Standard von JavaScript

SCOPING IN JAVASCRIPT

WHAT YOU NEED TO KNOW..

GLOBAL SCOPE

```
// global scope
var name = 'Todd';
```

- APIs und Module nutzen meist den Global Scope (jQuery)
- Bad thing: Namespace Clashes

LOCAL SCOPE

```
// Scope A: Global scope out here
var myFunction = function () {
  // Scope B: Local scope in here
};
```

- Lokale Variablen sind nicht Sichtbar ausserhalb der Funktion
- Lokale Variablen können aber mit neuen Scope definiert werden (Exposing)

FUNCTION SCOPE

- Jeder Scope wird **nur** mit einem Funktionsaufruf erstellt
- New functions = new scope

```
// Scope A
var myFunction = function () {
  // Scope B
  var myOtherFunction = function () {
    // Scope C
  };
};
```

LEXICAL SCOPE (CLOSURES)

- Jede innere Funktion hat Zugriff auf die Variablen des äusseren Scopes

```
var myFunction = function () {
  var name = 'Todd';
  var myOtherFunction = function () {
    console.log('My name is ' + name);
  };
  console.log(name);
  myOtherFunction(); // call function
};

// Will then log out:
// `Todd`
// `My name is Todd`
```

LEXICAL SCOPE (CLOSURES)

```
var name = 'Todd';
var scope1 = function () {
  // name is available here
  var scope2 = function () {
    // name is available here too
    var scope3 = function () {
      // name is also available here!
    };
  };
};
```

LEXICAL SCOPE (CLOSURES)

```
// name = undefined
var scope1 = function () {
  // name = undefined
  var scope2 = function () {
    // name = undefined
    var scope3 = function () {
      var name = 'Todd'; // locally scoped
    };
  };
};
```

DAS WORT THIS

- Jeder Scope bindet unterschiedliche Werte an 'this'
- Abhängig von dem Scope indem 'this' gecalled wird
- Default: window oder node

```
var myFunction = function () {
          console.log(this);
          // this = global, [object Window]
   };
myFunction();

var myObject = {};
   myObject.myMethod = function () {
        console.log(this); //this = Object { myObject }
   };

var nav = document.querySelector('.nav');
// <nav class="nav">
var toggleNav = function () {
        console.log(this); // this =  <nav> element;
        };
nav.addEventListener('click', toggleNav, false);
```

PROBLEM:

```
var nav = document.querySelector('.nav');
var toggleNav = function () {
  console.log(this); // <nav> element
  setTimeout(function () {
    console.log(this); // [object Window]
  }, 1000);
};
nav.addEventListener('click', toggleNav, false);
```

PROBLEM:

- Ein neuer Scope (mit setTimeout() welcher nicht von dem Eventhandler stammt)
- setTimeout ist default bei dem window-Object
- Wir müssen das 'richtige this' vorher speichern

LÖSUNG:

```
var nav = document.querySelector('.nav');
var toggleNav = function () {
  var that = this;
  console.log(that); // <nav> element
  setTimeout(function () {
    console.log(that); // <nav> element
  }, 1000);
};
nav.addEventListener('click', toggleNav, false);;
```

LÖSUNG: (BETTER WAY)

.call() und .apply()

```
var links = document.querySelectorAll('nav li');
for (var i = 0; i < links.length; i++) {
  (function () {
    console.log(this);
  }).call(links[i]);
}
```

.CALL() UND .APPLY()

- Bindet das 'richtige this' zu dem zugehörigen Scope
- .call(scope, arg1, arg2, arg3)
- .apply(scope, [arg1, arg2])

WTF?????

- Variablen (`var`) sind Function Scoped
- Jede Variable die nicht gebunden ist, ist frei zum Outer Scope

UND ES KOMMT NOCH HÄRTER: HOISTING

- Variablen werden so ausgewertet als wären sie am Anfang des Scopes
- Zuerst werden alle Variablendeklarationen gesucht 'var'
- Diese sind default undefined
- Siehe lectureExamples/ecmaScriptExamples/hoisting.js

KLEINE ABHILFE IN ECMASCRIPT 6

- Einführung von Block-scoped Variablen
- Neue Variablendeklaration: **let** und **const**
- geschweifte Klammen definieren hier den Geltungsbereich (und nur für den)

CONST

- Deklaration einer Variable mit konstantem Wert
- Der Wert dieser Variablen kann nicht geändert werden
- Muss initialisiert werden, wenn sie deklariert wurde.
- Wird für das require der (npm) Module verwendet:

```
const express = require('express');
const socketio = require('socketio');
```

LET

- Deklaration einer Variablen, deren Bereich auf den Block beschränkt ist
- kann nicht vor ihrer Deklaration verwendet werden
- Default als undefined.
- Wird für das require der (npm) Module verwendet:
- Siehe lectureExamples/ecmaScriptExamples/letconst.js

```
var  l = 10;
{
    let l = 2;
   // At this point, l = 2.
}
// At this point, l = 10.
```

ARROW FUNCTIONS

SYNTACTIC SUGAR

- Kürzerer Syntax als 'bekannte' function expressions
- Sind immer anonym

```
var a2 = a.map(function(s){ return s.length });

var a3 = a.map( s => s.length );
```

SYNTACTIC SUGAR 2.0

```
nums.forEach(v => {
   if (v % 5 === 0)
       fives.push(v)
})

nums.forEach(function (v) {
   if (v % 5 === 0)
       fives.push(v)
})
```

EXTENDED PARAMETER HANDLING

DEFAULT PARAMETER VALUES

```
//ECMAScript
function f (x, y = 7, z = 42) {
        return x + y + z;
}

//JavaScript
function f (x, y, z) {
    if (z === undefined) {
        z = 42;
        }
    return x + y + z;
};
```

REST PARAMETER

```
//ECMAScript
function f (x, y, ...a) {
        //a.length == 3
        return (x + y) * a.length;
}

//JavaScript
function f (x, y) {
    var a = Array.prototype.slice.call(arguments, 2);
    return (x + y) * a.length;
};

f(1, 2, "hello", true, 7) === 9;
```

TEMPLATE STRINGS

TEMPLATE-STRINGS

- sind in back-ticks (` `) eingeschlossen (kein doppelten "" oder einfach ")
- erlauben einfache mehrzeilige Strings
- erlauben Platzhalter (Dollarsymbol gefolgt von geschweiften Klammern)

MEHRZEILIGE STRINGS

```
//Vorher:
console.log("string text line 1\n"+
"string text line 2");

//Als Template String:
console.log(`string text line 1
string text line 2`);
```

KEINE EXPLIZITE KONKATINATION

```
var a = 5;
var b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
```

RAW STRINGS

String.raw()

```
String.raw(`Hi\n${2+3}!`);
// "Hi\\n5!"
```

ENHANCED OBJECT PROPERTIES

PROPERTY SHORTHAND

Reduzierter Syntax für das definieren von Objekten

```
//ECMAScript
obj = { x, y };

//JavaScript
obj = { x: x, y: y };
```

COMPUTED PROPERTY NAMES

Reduzierter Syntax für das definieren von Objekten

```
//ECMAScript
let obj = {
foo: "bar",
[ "baz" + quux() ]: 42
};

//JavaScript
var obj = {
    foo: "bar"
};
obj[ "baz" + quux() ] = 42;
```

PROPERTY SHORTHAND

Reduzierter Syntax für das definieren von Objekten

```
//ECMAScript
obj = {
        foo (a, b) {
        …
        }
};

//JavaScript
obj = {
        foo: function (a, b) {
        …
        }
};
```

INTERNATIONALIZATION & LOCALIZATION

INTERNATIONALIZATION & LOCALIZATION

- Automatische Formatierung für Zeit und Datum
- Datum USA: 2016-01-07, Datum Deutschland: 01.07.2016
- Zahlenangaben England: 1,234,567.89, Deutschland: 1.234.567,89

NUMBER FORMATTING

```
var l10nEN = new Intl.NumberFormat("en-US");
var l10nDE = new Intl.NumberFormat("de-DE");
l10nEN.format(1234567.89) === "1,234,567.89";
l10nDE.format(1234567.89) === "1.234.567,89";
```

CURRENCY FORMATTING

```
var l10nUSD = new Intl.NumberFormat("en-US", { style: "currency", curr
var l10nGBP = new Intl.NumberFormat("en-GB", { style: "currency", curr
var l10nEUR = new Intl.NumberFormat("de-DE", { style: "currency", curr
l10nUSD.format(100200300.40) === "$100,200,300.40";
l10nGBP.format(100200300.40) === "£100,200,300.40";
l10nEUR.format(100200300.40) === "100.200.300,40 €";
```

DATE/TIME FORMATTING

```
var l10nEN = new Intl.DateTimeFormat("en-US");
var l10nDE = new Intl.DateTimeFormat("de-DE");
l10nEN.format(new Date("2015-01-02")) === "1/2/2015";
l10nDE.format(new Date("2015-01-02")) === "2.1.2015";
```