Workshop: JavaScript für Java-Entwickler

Oliver Zeigermann | http://zeigermann.eu

Online Version: http://bit.ly/1EKba2b

Oliver Zeigermann

- Entwickler, Architekt, Berater und Coach
- http://zeigermann.eu
- Buch: JavaScript für Java-Entwickler
- Arbeitet bei *embarc Software Consulting GmbH*

Inhalte

- Basiswissen
- Funktionen
- Objekte und Klassen
- Module
- Optional: Unit-Tests
- Optional: jQuery
- Optional: Die Zukunft von JavaScript

Hello World

Hello World #1

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World!</title>
    <script>
      alert("Hello World");
    </script>
  </head>
  <body>
  </body>
</html>
```

Run

Hello World #2

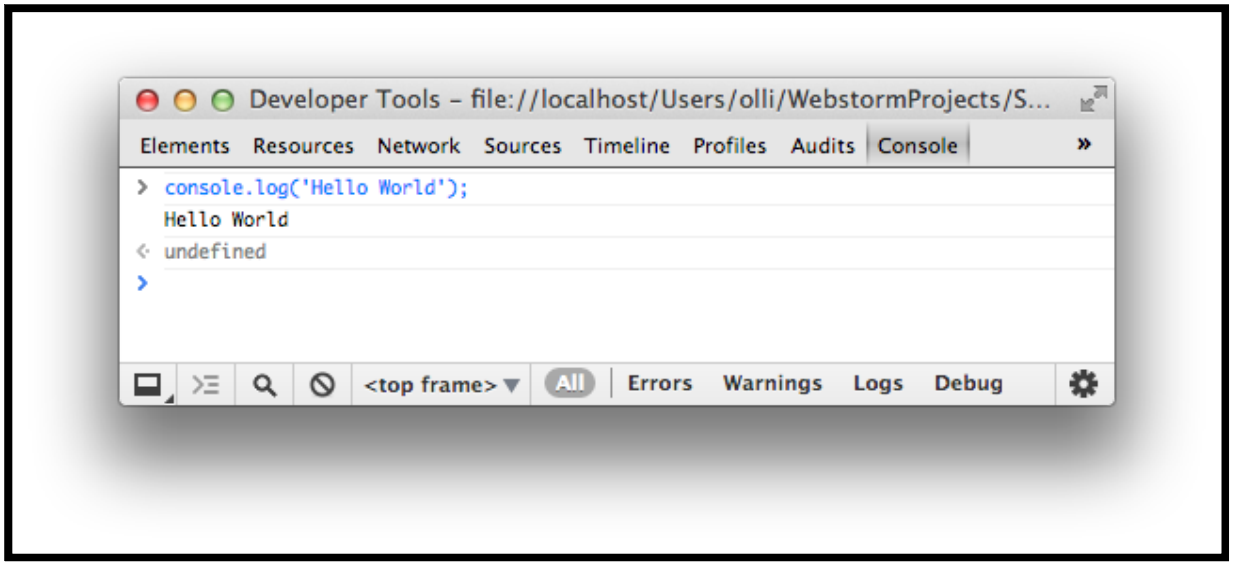
```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World!</title>
  </head>
    <body>
      <div id="log"></div>
      <script>
          var element = document.getElementById("log");
          element.innerHTML = "<h1>Hello World</h1>";
      </script>
    </body>
</html>
```

Run

Hello World #3

```
node -e "console.log('Hello World');"
```

Hello World #4



Hello World #5

JS Bin Öffnen

Object

```
var map = {
    feld1: 'Huhu',
    zweites$Feld: "Auch sowas geht!"
};
console.log(typeof map === "object"); // true
console.log(map.feld1); // Huhu
console.log(map["zweites$Feld"]); // Auch sowas geht!

map.hund = "Ganz neu geht auch";
```

Typen

```
var string = "String";
typeof string === "string";

var int = 1;
typeof int === "number";

var float = 1.0;
typeof float === "number";

var bool = true;
typeof bool === "boolean";

var func = function() {};
typeof func === "function";

typeof michGibtEsNicht === "undefined";
```

Array

```
var array = ["a", "b", "c"];
var el = array[2];
array[1] = 20;
typeof array === "object";
// fügt die 4 am Ende hinzu
array.push(4);
```

Kontrollstrukturen

for

```
for (var i=0; i < array.length; i++) {
    console.log(i + ": " + array[i]);
}

// Durch alle Feld-Namen iterieren
// Geht für Map / Object und für Array!
for (var i in map) {
    console.log(i + ": " + map[i]);
}
```

Kontrollstrukturen ansonsten wie in Java / C#

- if / else
- while / do
- switch
- break / continue
- Referenz

Funktionen

```
function f2() {
    console.log("Called!");
}
var result2 = f2();
result2 === undefined;

var f1 = function(p1, p2) {
    return p1 + p2;
};
var result1 = f1(1,2);
result1 === 3;
```

Optionale Parameter

```
function f1(p1) {
    if (typeof p1 === 'undefined') {
        return null;
    } else {
        return p1;
    }
}

var result1 = f1(1);
console.log(result1 === 1);

var result2 = f1();
console.log(result2 === null);
```

Varargs

```
function summe() {
    var sum = 0;
    for (var a in arguments) {
        sum += arguments[a];
    }
    return sum;
}

var result5 = summe(1,2,3);
console.log(result5 === 6);
```

Scopes

So nicht!

```
{
    var huch = "Ich bin noch da";
}

console.log(huch); // Ich bin noch da
```

So!

```
(function () {
    var achso = "Ich bin weg";
}());

console.log(achso); // ReferenceError
```

Immediately-Invoked Function Expression (IIFE)

Übung: Basiswissen

- Erzeuge ein Array mit Personen-Objekten mit mindestens den Eigenschaften
  - Name
  - Alter
  - Geschlecht
- Gib alle Personen nacheinander aus
  - Erstelle eine Funktion, die eine Person auf der Console ausgibt

Objekte, die Zweite

- Objekte können auch Funktionen als Properties haben
- Diese funktionieren dann wie Methoden, d.h. `this` ist an das Objekt gebunden über das sie aufgerufen werden

```
var obj = {
    field: 10,
    log: function() {
        console.log(this.field);
    }
};

obj.log(); // 10
```

Call / Apply

- call und apply sind Methoden auf Funktionen
- Erlauben das freie Binden an einen anderen Kontext

```
var field = "Reingelegt";
obj.log.call(this);
// => ???
```

Klassen mit JavaScript

- Klassen und Konstruktoren sind Mechanismen, um mehrere, strukturell gleiche oder ähnliche Objekte zu erzeugen
- Auch in JavaScript können eigene Klassen definiert werden
- Einfachvererbung ist ebenso möglich
- Der Mechanismus ist nicht direkt in die Sprache eingebaut
- Stattdessen benutzen wir Best-Practice-Patterns
- Grundlage ist die prototypische Vererbung

Prototypen

- Jedes Objekt hat zusätzlich eine Referenz auf seinen Prototyp
  - `Object.getPrototypeOf()` in neueren Browsern
- `Object` hat keinen Prototypen, ist aber Prototyp aller anderen Objekte
- Lesende Property-Zugriffe können transitiv an Prototypen delegiert werden
- Dies heißt prototypische Vererbung

Setzen des Prototypen aka das Typen-System

Der Prototyp kann nicht direkt, aber durch Aufruf von `new` gesetzt werden

```
/** @constructor */
function Person(name) {
    this.name = name;
}

// Methode
Person.prototype.getName = function() {
    return this.name;
};


var olli = new Person('Olli');
olli.getName() === 'Olli';
```

Ablauf eines Konstruktoraufrufs mit `new`

1. ein leeres, neues Objekt wird erzeugt
2. die Konstruktor-Funktion hat ein Property `prototype`, dies wird als Prototyp des neuen Objekts verwendet
3. `this` wird an dieses neue Objekte gebunden
4. die Konstruktor-Funktion wird aufgerufen (mit `this` gebunden)
5. das neue Objekt wird implizit zurückgegeben (wenn die Funktion kein explizites `return` hat)

”Typsystem”

Ein Objekt ist `instanceof` aller seiner Prototypen

```
var olli = new Person('Olli');
Object.getPrototypeOf(olli) === Person.prototype;
olli instanceof Object;
olli instanceof Person;
```

Warnung

Douglas Crockford findet Klassen und *this* doof

Douglas Crockford hatte vor einigen Jahren großen Einfluss auf die Entwicklung von JavaScript

Douglas Crockford schrieb das Buch „JavaScript the good parts"

Die Zukunft von JavaScript gibt ihm allerdings nicht Recht (später mehr)

Vererbung

1. Klassen-Hierarchien und Instanzen nutzen beide Prototypische Vererbung
2. Klassen-Hierarchien werden einmal aufgebaut und als Prototypen der Instanzen verwendet
3. Klassen-Hierarchien werden ebenso über Prototypen erstellt
4. Object.create erzeugt ein neues Object mit einem anderen Objekt als Prototypen
5. Aufruf von Super-Konstruktoren und Super-Methoden über call / apply

Vererbung #1

```
function Person(name, gender) {
    this.name = name;
    this.gender = gender;
}
Person.prototype.getName = function() {
    return this.name;
};

function Male(name) {
    Person.call(this, name, "Male"); // super call
}
Male.prototype = Object.create(Person.prototype);
```

Vererbung #2

```
Male.prototype.getName = function() {
    // super call
    return "Mr " + Person.prototype.getName.call(this);
};

var olli = new Male('Olli');
olli.getName() === 'Mr Olli';
olli.gender === 'Male';
olli instanceof Male;
olli instanceof Person;
olli instanceof Object;
```

Vererbungshierarchien - revisited

- Das verwirrendste Thema in JavaScript
- Alternative zu Klassen-Hierarchien sind Mixins
- Viele Bibliotheken nehmen sich dieses Themas an
- Bei Mixins werden alle Properties eines Prototypen in einen anderen hineinkopiert

Beispiel

```
function Programmer(name, language) {
    Person.call(this, name);
    this.language = language;
}
_.extend(Programmer.prototype, Person.prototype);
// instead of
//Programmer.prototype = Object.create(Person.prototype);

Programmer.prototype.code = function() {
    return this.getName() + " codes in " + this.language;
};
var programmer = new Programmer('Erna', 'JavaScript');
console.log(programmer.getName()); // Erna
console.log(programmer.code()); // Erna codes in JavaScript
console.log(programmer instanceof Programmer); // true
// true for Object.create, false for _.extend
console.log(programmer instanceof Person);
```

Underscore

- _.*extend* eine von vielen Hilfsfunktionen der Bibliothek *Underscore*
- http://underscorejs.org/
- Versucht Unzulänglichkeiten der Sprache als Bibliothek auszugleichen
- Ein Klassiker unter den JavaScript-Bibliotheken
- wird typischerweise als _ importiert (daher der Name)
- https://lodash.com/ ist die moderne Variante
- Neues JavaScript-Versionen enthalten bereits eine ganze Reihe der Funktionen dieser Bibliotheken

Übung: Klassen

1. Schreibe eine Klasse für Person
   - Lasse im Konstruktor die drei bekannten Parameter für `name`, `alter` und `geschlecht` zu
   - Mache aus allen Funktionen, die auf Personen arbeiten, Methoden
2. Erzeuge ein Objekt vom Typ Person und rufe Methoden darauf auf

Optionaler Zusatzübung: Vererbung

1. Schreibe die Klasse Customer
   - Customer soll von Person erben
   - Berechne im Konstruktor aus den Parametern zumindest ein zusätzliches Feld, das den vermuteten bevorzugten Kaufgegenstand angibt
   - Rufe aus dem Customer-Konstruktor den Person-Konstruktor auf
   - Überschreibe die Methode `getName`
   - Füge die Methode `shop` hinzu, die den bevorzugten Gegenstand ausgibt
2. Erzeuge mindestens ein Objekt vom Typ Customer und rufe Methoden darauf auf

Module

Module in JavaScript werden über `Closures` realisiert

Closure in einem Satz

Eine innere Funktion hat immer Zugriff auf alle Variablen und Parameter ihrer äußeren Funktion, *auch wenn diese äußere Funktion bereits beendet ist*.

Frei nach *Douglas Crockford*

Beispiel Closure

```
function outer() {
    var used = "Olli";
    var unused = "Weg";
    return (function() {
        return "Text: " + used;
    });
}

var inner = outer();
console.log(inner());
```

Closure Definition

Eine Closure ist eine spezielle Art von Objekt, welche zwei Dinge kombiniert

1. Eine Funktion
2. die Umgebung in welcher diese Funktion erzeugt wurde - diese Umgebung besteht aus allen lokalen Variablen und Parametern, die sichtbar waren als die Closure erzeugt wurde

Aus der Definition auf MDN

Revealing Module Pattern

```
var humanresources = (function () {
    function InternalStuff() {
    }

    function Person(name) {
        this.name = name;
        // uses internal stuff
    }

    return {
        Person: Person
    };
})();
```

Sichtbarkeit bei Revealing Module Pattern

```
var olli = new humanresources.Person('Olli');
olli.name === 'Olli';
// TypeError: undefined is not a function
new humanresources.InternalStuff();
```

Es gibt zwei grundsätzlich unterschiedliche Modul-Systeme als Defacto-Standard

- AMD
  - Module werden nicht blockierend und potentiell asynchron geladen
  - Module werden über bower oder manuell installiert
  - Verwendung auf Client-Seite
  - Default-Implementierung: RequireJS
- CommonJS
  - Module werden blockierend und synchron geladen
  - Module werden über npm installiert
  - Verwendung auf Server-Seite
- Beide Modul-Systeme können über r.js / Browserify auch auf Server / Client benutzt werden

Definieren eines commonJS Moduls bei node

```
// in der Datei "eater.js"
var eatThis = function (name) {
    console.log(name);
};
exports.eatThis = eatThis;
```

Benutzen eines commonJS Moduls bei node

```
var eaterModule = require("eater");

eaterModule.eatThis(name);
```

Definieren eines AMD Moduls mit requireJS

```
// Ein Modul pro Datei:
// js/modules/accounting.js
define(function () {
    return {
        getIdNumberForName: function(name) {
            // ...
        }
    };
});
```

Benutzen eines AMD Moduls mit requireJS

```
require(['js/modules/accounting'], function (Accounting) {
    var name = // ...
    var id = Accounting.getIdNumberForName(name);
    // ...
});
```

Übung: Module

1. Schreibe ein Modul, in das du die vorhandenen Typendefinitionen verschiebst. Dieses Modul soll
   - nach außen nur die `Customer`-Klasse exportieren
2. Schreibe den aufrufenden Code so um, dass er mit den neuen Modulen arbeitet

Testen mit Jasmine

Grundlagen Jasmine

- weit verbreitetes JavaScript Unit-Testframework
- Stil BDD orientiert, etwas anders als JUnit
- Ausführung im Browser
- `describe`: eine komplette Test-Suite
- `it`: ein einzelner Testfall
- `expect`: erwartetes Ergebnis ausdrücken
- `beforeEach / afterEach`: Vorbereitung / Aufräumarbeiten
- Hauptseite Jasmine

Test-Suite

```
describe("Calculator", function () {
    var data;
    beforeEach(function () {
        data = calculateMortgage(200000, 10, 7.5, 30);
    });
    afterEach(function() {
    });

    it("principle", function () {
        expect(data.principle).toEqual(199990.00);
    });
    it("invalid", function () {
        expect(function () {
            calculateMortgage(1, 10, 7.5, 30);
        }).toThrowError("You do not need any money");
    });
});
```

Beispiel-Test-Lauf

Tests laufen lassen

Spies (am Beispiel von Ext Js)

```
it('that click calls calculate method', function() {
    spyOn(formCtrl, 'calculateMortgage');
    var btn = formCtrl.getCalculateButton();
    btn.fireEvent('click');

    expect(formCtrl.calculateMortgage).toHaveBeenCalled();
});

it('that fired event is received', function() {
    spyOn(formCtrl, 'openMortgageEditor');

    // main controller fires event that should be called by form controller
    mainCtrl.fireEvent('edit', mortgage);

    expect(formCtrl.openMortgageEditor).toHaveBeenCalledWith(mortgage);
});
```

Übung: Teste deine eigenen Klassen

- Nutze dafür dieses JsBin-Projekt als Grundlage http://jsbin.com/jijuri/2/edit?js,output
- Clone das Projekt und verschiebe deine Klassen in den JavaScript-Bereich
- Schreibe mit Jasmine sinnvollen Spezifikationen für deine Klassen
- Achtung: Aus technischen Gründen, läuft Jasmine hier in der älteren Version 1.3.1

jQuery

Hello World jQuery

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World jQuery!</title>
    <body>
      <div id="log"></div>
      <script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.min.js"

      <script>
        $(document).ready(function(){
            $("#log").html("<h1>Hello World</h1>");
        });
      </script>
    </body>
</html>
```

Run

Überblick

- Standard-JavaScript-Bibliothek
- Fast überall zu finden
- Adressiert Probleme bei der Programmierung des DOMs
- Flexible Trennung von View und Logik
- Funktionalität unterteilbar in "Auswahl" und "Manipulation"
- Core, UI und Mobile Bibliothek vorhanden
- Viele Plugins vorhanden, z.B. jcarousel

Selectors

- Zur Auswahl von Elementen
- Spucken immer ein Array von Elementen aus
- nach CSS-Klasse: `$(".class")`
- nach Tag: `$("tag")`
- nach id: `$("#id")`
- nach Attribut: `$("[name='value']")` bzw. `$("tag[name='value']")`
- Hierarchie: `$("form input")`

Manipulation

- Operieren auf der Ausgabe der Selektoren
- anzeigen `show()` / verbergen `hide()`
- Wert des ersten Elements holen `val()` - für Inputs in Form
- Wert des ersten Elements setzen `val(value)` - für Outputs in Form
- Z.B.

```
log.show();
```

Listener

- Listener werden auf der Ausgabe von Selektoren definiert
- `on(eventName, handler)` Listener registrieren
- Events z.B. `click` oder `submit`
- Z.B.

```
var log = $("#log");
log.on('click', function(event) { // ... });
```

Tiefere Einsichten in jQuery

- `$` ist ein kürzerer Name für die globale Funktion `jQuery`
- `$` hat statische Methoden, z.B. `getJSON`
- `$` erzeugt Objekte vom Typ `$`
- CSS-Selektoren sind denen von jQuery sehr ähnlich
- HTML5 bietet Selektoren mit ähnlicher Mächtigkeit, z.B. `document.querySelector('#log');`

Plugins

`jQuery` kann um eigene Funktionalität erweitert werden

$ als Factory

```
var log = $("#log");
Object.getPrototypeOf(log) === $.prototype; // true
log instanceof $; // true
```

Erweiterung der Funktionalität von $ über Plugins

```
$.fn === $.prototype;
$.fn.myPlugin = function() {
  // ...
};
log.myPlugin();
```

Die 50 nützlichsten Plugins 2013

Code-Beispiel: Einfaches Plugin

Alle durch den Selektor gefundenen Elemente ausgeben

```
var log = $("#log");
$.fn.dump = function() {
    // iterate over all elements found by selector
    this.each(function() {
        // this is a DOM element,
        // make it a $ object if desired
        console.log($(this));
    });
    // return $ element for chaining
    return this;
};
// chaining does work
log.dump().show();
```

Tutorial jQuery-Plugins

Übung: jQuery

- Nutze dafür dieses JsBin-Projekt als Grundlage http://jsbin.com/zofure/1/edit?html,js,console,output
- Clone das Projekt für und führe die dort beschriebenen Anweisungen aus
- Ziel ist es, einen Textbereich ein- und auszublenden
- Falls du lieber ein bisschen spielen möchtest, kannst du auch hier mit der Lösung beginnen http://jsbin.com/rutede/3/edit?html,js,console,output

Optionale Zusatz-Übung: jQuery

- Zeige eine Instanz einer deiner Klassen an
- Mache alle Eigenschaften (Properties) per Click sichtbar oder unsichtbar
- Beginne mit der Lösung der ersten Aufgabe, wenn du eine Grundlage brauchst http://jsbin.com/rutede/3/edit?html,js,console,output

Die Zukunft von JavaScript

- JavaScript gewinnt immer mehr an Bedeutung
- Wer für den Browser entwickelt, kommt an JavaScript nicht vorbei
- JavaScript hat allerdings viele Klippen
  - keine deklarierte Typen
  - Klassen und Vererbung umständlich
  - keine lexikalischen Scopes
  - Bindung von *this* kann verwirrend sein
  - Kein Einheitliches Modul-Konzept
- Die nächste Version von JavaScript wird diese Klippen entfernen (außer Typen)
- Es gibt Ansätze für deklarierte Typen in JavaScript

JavaScript vs ECMAScript 6 vs ECMAScript 2015 vs Projekt Harmony

- ECMAScript ist der Standard und JavaScript ist die Implementierung
- Ab IE9 sind wir bei ECMAScript 5 (2009)
- ECMAScript 2015 ist der neue Name für ECMAScript 6
- Projekt Harmony ist ECMAScript >=6
- Spec für ES6 beinahe final
- In 5 Jahren breit verfügbar (lauf Wirfs-Brock, Editor der Spec)
- Vieles bereits heute nutzbar

Neue Spracheigenschaften

keine statischen Typen

- Module
- Klassen
- Destructuring (Pattern Matching)
- let, const
- Fat arrow **=>** für besseres Binding von **this**
- vararg, optional, spread operator
- Collection Types (Map, Set, WeakMap, WeakSet)

Gesprengte Klippe #1: Lexikalisches Scoping

```
{
  let a = 10;
  // or
  const a = 10;
  a = 20;
  // when a is const:
  // => TypeError: Assignment to constant variable.
}

console.log(a);
// => ReferenceError: a is not defined
```

Gesprengte Klippe #2: Klassen

```
class Person {
    constructor(name) {
        this.name = name;
    }
    getName() {
        return this.name;
    }
}
class Programmer extends Person {
    constructor(name, language) {
        super(name);
        this.language = language;
    }
    code() {
        return this.getName() + " codes in " + this.language;
    }
}
const programmer = new Programmer('Erna', 'JavaScript');
```

Weitere gesprengte Klippen

- Binden von this für Callbacks etc. mit =>
- Module
- Unicode support
- Reflection API
- Tail Recursion

Wo läuft ES 2015 schon jetzt?

- Chrome
- Firefox
- *node --harmony --use-strict*
- io.js https://iojs.org/en/es6.html
- Transpiler
  - https://babeljs.io/, ehemals 6to5, wird von JsBin unterstützt
  - https://github.com/google/traceur-compiler
  - http://www.typescriptlang.org/ (plus statische Typen)
  - Liste weiterer Transpiler https://github.com/addyosmani/es6-tools
- Wie viel läuft?: http://kangax.github.io/es5-compat-table/es6

Übung: ES 2015 Klassen

1. Schreibe deine Klassen auf die neue Klassensyntax von ES 2015 um
2. Nutze dafür den Babel-Übersetzer in JsBin: http://jsbin.com/jobaci/1/edit?js,console