JavaScript-Sprachgrundlagen für C#-Entwickler

Eine Einführung

Oliver Zeigermann / @DJCordhose

Online-Version der Slides: http://djcordhose.github.io/serious-javascript/slides/js_csharp_basta_2014.html

Code dieser Session: http://djcordhose.github.io/serious-javascript/coding-sessions/basta2014.html

Joe McCann
@joemccann

Folgen

"By 2017, JavaScript will be the most in-demand language skill in application development (AD)."

— Forrester Research 2014

Übersetzt aus dem Englisch von bing Translator

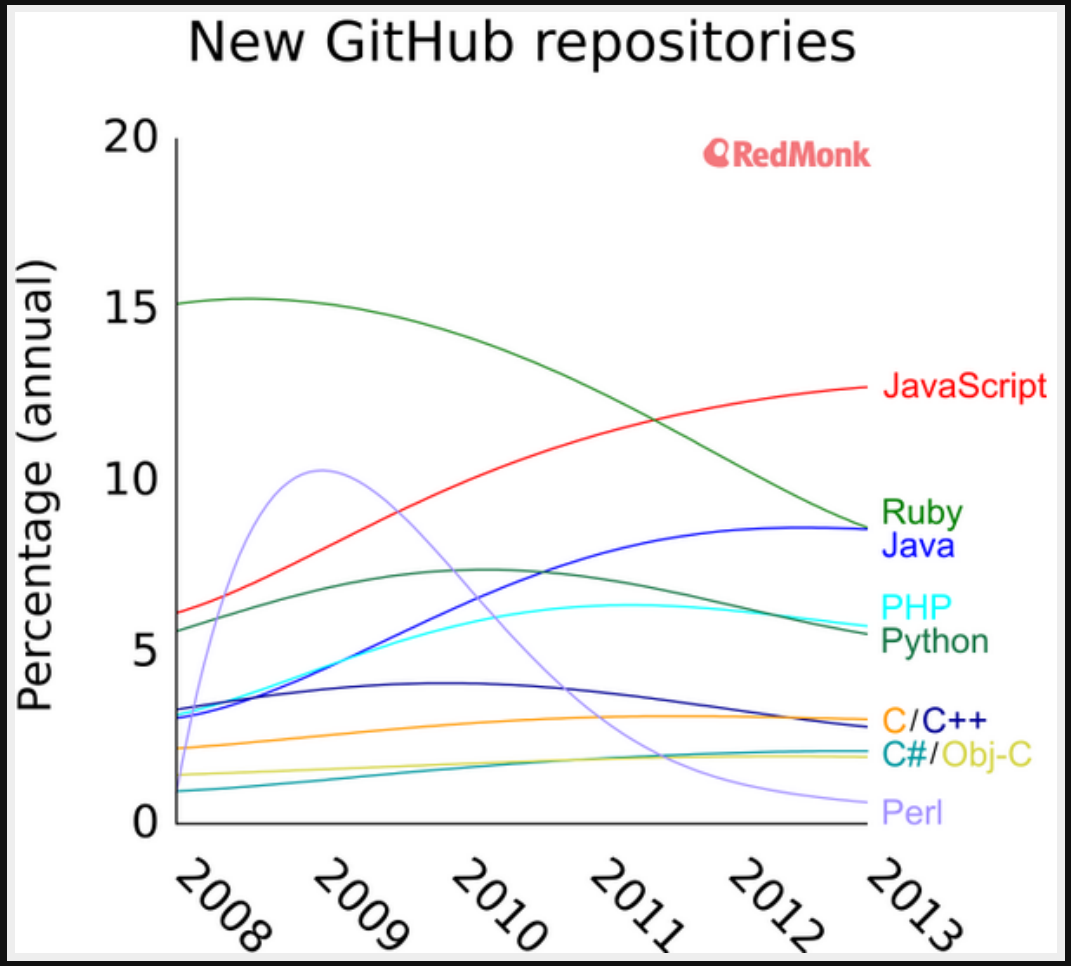
"Bis zum Jahr 2017 werden JavaScript die am meisten gefragte Sprachfertigkeit in der Anwendungsentwicklung (AD)".

— Forrester Research 2014

Antworten Retweetet Favorisiert ••• Mehr

RETWEETS	FAVORITEN
264	125

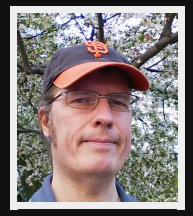
20:29 - 16. Apr. 2014



http://redmonk.com/dberkholz/2014/05/02/github-language-trends-and-the-fragmenting-landscape

Oliver Zeigermann

- Entwickler bei http://graylog2.org/
- @DJCordhose
- https://github.com/DJCordhose
- javatojavascript.blogspot.de
- JavaScript und TypeScript für C#-Entwickler



Signierstunde

Mittwoch, 10:00 Uhr

„JavaScript und TypeScript für C#-Entwickler“

mit **Oliver Zeigermann**

entwickler.press

Themen Live-Coding

- var
- console.log
- environments: node und chrome
- object
- array
- for
- function
- iife
- foreach
- class pattern
- module pattern
- CommonJs
- TypeScript

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";
map.f = function() { return "Aha!" };

delete map.hund; // löschen
console.log(map.hund); // 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);
// An Position 1 werden 2 Elemente entfernt und zurück gegeben
array.splice(1, 2);
// An Position 1 werden 0 Elemente entfernt und zurück gegeben
// Zudem wird an Position 1 "x" hinzugefügt
array.splice(1, 0, "x");
```

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";
```

Iterieren über Array-Elemente

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

// Achtung: Iteriert über Indizes, nicht über die Elemente!
for (var i in array) {
    console.log(i + ": " + array[i]);
}
```

Kontrollstrukturen ansonsten fast wie in C#

- if / else
- while / do
- switch (Achtung: case-Block erfordert kein *break* am Ende)
- break / continue
- Referenz

Funktionen

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

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

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 #1

```
function f2() {
    // "arguments" enthält immer alle Argumente der Funktion
    if (typeof arguments[0] === 'undefined') {
        return null;
    } else {
        return arguments[0];
    }
}

var result3 = f2(1);
console.log(result3 === 1);

var result4 = f2();
console.log(result4 === null);
```

Varargs #2

```
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);
```

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)

Arrays revisited

```
var array = [1,2,3];

array.forEach(function(e) { console.log(e); });

array.filter(function(e) { return e > 2; });

array.map(function(e) { return e + 100; });

array.reduce(function(reduced, current) {
    return reduced + current;
}, 100);
```

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 global = {
    field: "Reingelegt"
};
obj.log.call(global);
// => ???
```

Klassen mit JavaScript

- Klassen und Konstruktoren sind Mechanismen, um mehrere, strukturell gleiche oder ähnliche Objekte zu erzeugen
- Auch in JavaScript können eigene Typen 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
  - `__proto__` oder `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

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');
olli.__proto__ === Person.prototype;
olli.constructor === Person;
olli instanceof Object;
olli instanceof Person;
```

Vererbungshierarchien

- Das verwirrendste Thema in JavaScript
- Klassen-Hierarchien und Instanzen nutzen beide Prototypische Vererbung
- Klassen-Hierarchien werden einmal aufgebaut und als Prototypen der Instanzen verwendet
- Klassen-Hierarchien werden ebenso über Prototypen erstellt
- Aufruf von Super-Konstruktoren und Super-Methoden über call / apply
- Alternative zu Klassen-Hierarchien sind Mixins

Mixins

```
function Logable(name) {
    this.name = name;
}
Logable.prototype.log = function(msg) {
    console.log(this.name + ": " + msg);
}

function Person(name, age) {
    Logable.call(this, name);
    this.age = age;
}
_mixin(Person, Logable);

var olli = new Person('Olli');
console.log(olli instanceof Person); // true
console.log(olli instanceof Logable); // false
olli.log('Hi');
```

Implementierung von _mixin

```
function _mixin(_sub, _super) {
    for (var p in _super.prototype) {
        if (p === 'constructor') {
            continue;
        }
        _sub.prototype[p] = _super.prototype[p];
    }
}
```

Klassen-Hierarchien über Prototypen

```
function Logable(name) {
    this.name = name;
}
Logable.prototype.log = function(msg) {
    console.log(this.name + ": " + msg);
}

function Person(name, age) {
    Logable.call(this, name);
    this.age = age;
}
_extends(Person, Logable);

var olli = new Person('Olli');
console.log(olli instanceof Person); // true
console.log(olli instanceof Logable); // true
olli.log('Hi');
```

Implementierung von _extends

```
function _extends(_sub, _super) {
    _sub.prototype = new _super();
}
```

Achtung: Vereinfachte Version mit Schwächen

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;
        this.inner = new InnerStuff();
    }

    humanresources.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();
```

Namespaces

```
var eu = eu || {};
eu.zeigermann = eu.zeigermann || {};
eu.zeigermann.person = eu.zeigermann.person || {};
(function (person) {
    // Constructor
    function Person(name) {
        this.name = name;
    }

    // Factory
    function create(name) {
        return new Person(name);
    }

    // Export der Factory-Methode
    person.createPerson = create;
})(eu.zeigermann.person);
```

Import

```
eu.zeigermann.main = {};
(function (main, person) {
    function run() {
        var olli = person.createPerson("Olli");
        console.log(olli.name); // => "Olli"

        console.log(person.Person); // => undefined
    }
    main.run = run;
})(eu.zeigermann.main, eu.zeigermann.person);

eu.zeigermann.main.run();
```

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);
    // ...
});
```

Ausblick

ECMAScript 6 / Projekt Harmony

- Projekt Harmony ist ECMAScript >=6
- Ab IE9 sind wir bei ECMAScript 5
- Spec für ES6 beinahe final
- i18n API
- Module
- Klassen
- let, const
- Fat arrow **=>** für besseres Binding von **this**
- vararg, optional, spread operator
- keine statischen Typen
- Collection Types
- In 5 Jahren breit verfügbar (lauf Wirfs-Brock, Editor der Spec)
- Vieles bereits heute nutzbar über diverse Transpiler

Vielen Dank

Fragen / Diskussion

Oliver Zeigermann / @DJCordhose