

JavaScript

eine moderne Einführung
für Informatiker
und Java-Programmierer

Manfred Kaul



JavaScript ist die populärste Technologie

Programming, Scripting, and Markup Languages



... das war nicht
immer so ...

Erst die verbesserte

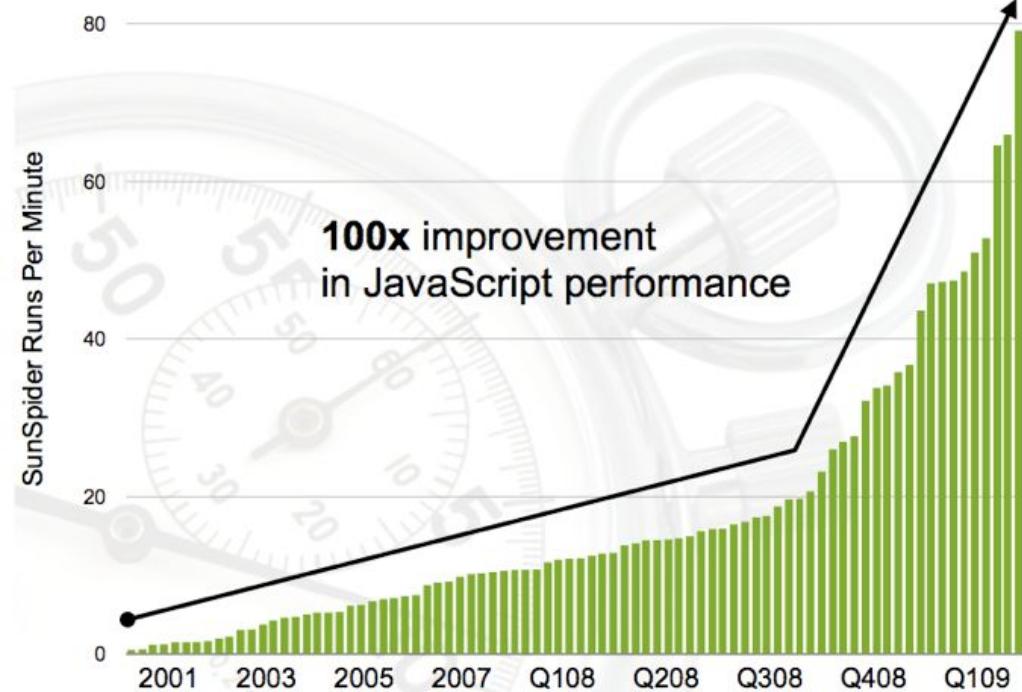
JavaScript Performance

hat JavaScript so attraktiv gemacht ...

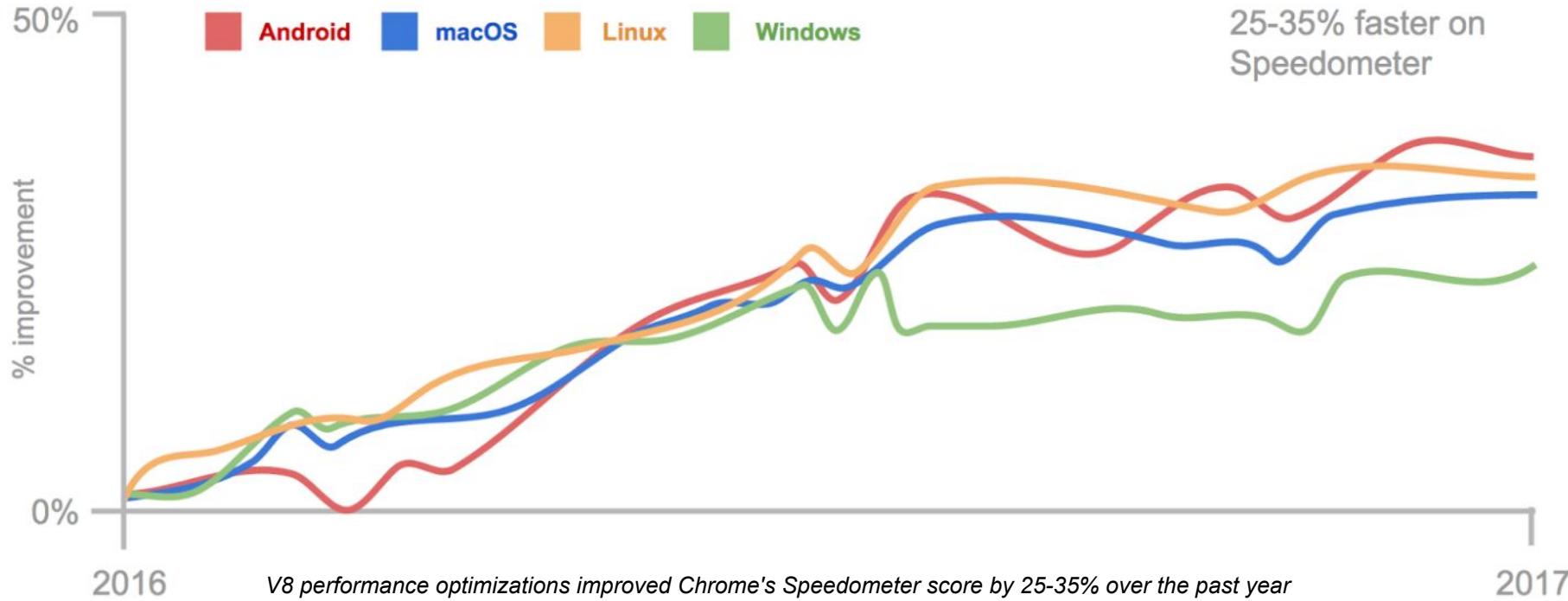


- JavaScript Engine Race
- Viele konkurrierende Implementierungen
- Heutiger Spitzenreiter Google V8

More Speed



https://en.wikipedia.org/wiki/JavaScript_engine



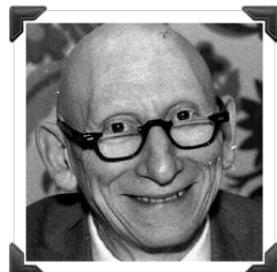
<https://blog.chromium.org/>

Warum JavaScript lernen?

- weil JavaScript die Sprache des Webs ist
 - weil JavaScript milliardenfach verbreitet ist
 - weil es der Standard des Webs seit 1995 ist
 - weil JavaScript gut zum Browser passt (e.g. DOM-Manipulation)
- weil JavaScript-Neuerungen die Browser-Innovation vorantreibt
 - höhere Sprachen, höhere Schichten hinken immer hinterher
- weil die Kombination aus Closures, Prototypen & Objekt-Literalen einzigartig ist
- weil JavaScript eine andere Art des Programmierens, Modellierens und Denkens ist. ⇒ Sapir-Whorf-Hypothese

Die Sapir-Whorf-Hypothese

- Die Sapir-Whorf-Hypothese besagt, Sprache forme Denken und Verhalten.
 - Wer in JavaScript wie in Java programmiert, hat die Sprache noch nicht verstanden.
- Die Hypothese ist bisher weder bewiesen noch widerlegt, hat aber viele wiss. Projekte stimuliert.



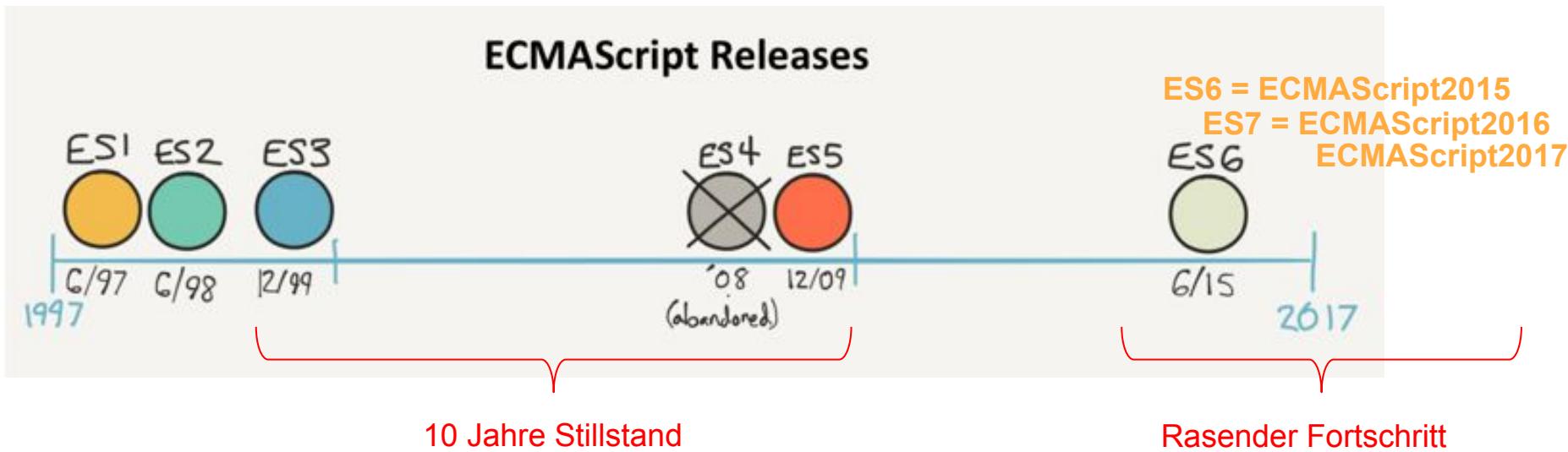
- Alan Perlis: "*A language that doesn't affect the way you think about programming is not worth knowing*"

aus "Epigrams on Programming"

Historie: JavaScript hat viele Namen:

~~LiveScript~~ = JavaScript = JS
= ECMAScript2015,16,... = ES6,7,... = ~~JScript~~

es ist im Kern immer die gleiche Sprache gemeint



<https://segment.com/blog/the-deep-roots-of-js-fatigue/>

Kompatibilitätstabelle für ES6

Wie stellt man Browser-Kompatibilität her?

Shims und Polyfills

Shims and polyfills are libraries that retrofit newer functionality on older JavaScript engines:

- A *shim* is a library that **brings new features to an older environment**, using only the means of that environment.
- A *polyfill* is a shim for a **browser API**. It typically checks if a browser supports an API. If it doesn't, the polyfill installs its own implementation. That allows you to use the API in either case. The term *polyfill* comes from a home improvement product; according to [Remy Sharp](#):

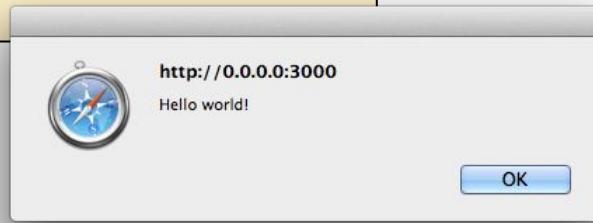
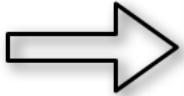
Polyfilla is a UK product known as Spackling Paste in the US. With that in mind: think of the browsers as a wall with cracks in it. These [polyfills] help smooth out the cracks and give us a nice smooth wall of browsers to work with. => Deutsch: “Moltofill”

Examples include:

- [“HTML5 Cross Browser Polyfills”](#): A list compiled by Paul Irish.
- [es5-shim](#) is a (**nonpolyfill**) shim that retrofits **ECMAScript 5 features** on ECMAScript 3 engines. It is purely language-related and makes just as much sense on Node.js as it does on browsers.

Historie von JavaScript

```
<html>  
<h1 onclick="alert('Hello World')>Header</h1>  
</html>
```



- Skriptsprache für HTML
- gedacht für kleine Event-Handler
 "onclick", "onchange", ...
- kein Java!
- Vorgänger LiveScript von Brendan Eich
erfunden bei der Fa. Netscape im Jahre
1995:

Erster Prototyp in 10 Tagen !!!

1995 



Quelle: https://de.wikipedia.org/wiki/Brendan_Eich

Event-Handler

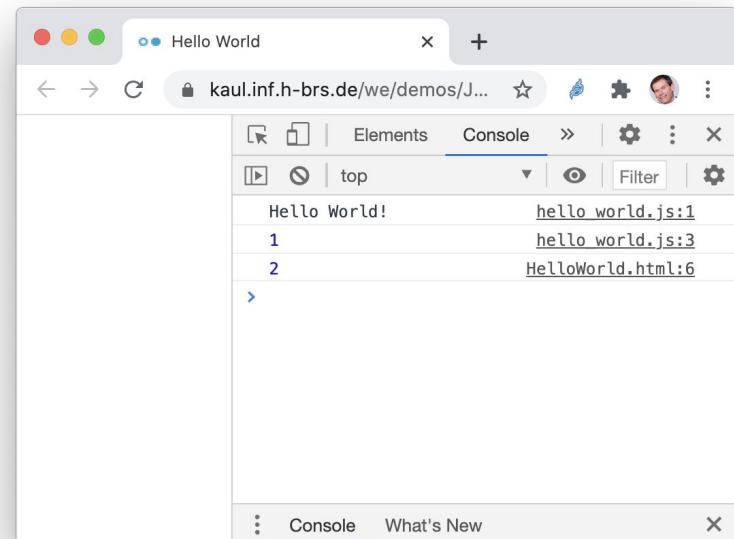
- ↓ [Allgemeines zu Event-Handlern](#)
- ↓ [onabort](#) (bei Abbruch)
- ↓ [onblur](#) (beim Verlassen)
- ↓ [onchange](#) (bei erfolgter Änderung)
- ↓ [onclick](#) (beim Anklicken)
- ↓ [ondblclick](#) (bei doppeltem Anklicken)
- ↓ [onerror](#) (im Fehlerfall)
- ↓ [onfocus](#) (beim Aktivieren)
- ↓ [onkeydown](#) (bei gedrückter Taste)
- ↓ [onkeypress](#) (bei gedrückt gehaltener Taste)
- ↓ [onkeyup](#) (bei losgelassener Taste)
- ↓ [onload](#) (beim Laden einer Datei)
- ↓ [onmousedown](#) (bei gedrückter Maustaste)
- ↓ [onmousemove](#) (bei weiterbewegter Maus)
- ↓ [onmouseout](#) (beim Verlassen des Elements mit der Maus)
- ↓ [onmouseover](#) (beim Überfahren des Elements mit der Maus)
- ↓ [onmouseup](#) (bei losgelassener Maustaste)
- ↓ [onreset](#) (beim Zurücksetzen des Formulars)
- ↓ [onselect](#) (beim Selektieren von Text)
- ↓ [onsubmit](#) (beim Absenden des Formulars)
- ↓ [onunload](#) (beim Verlassen der Datei)
- ↓ [javascript:](#) (bei Verweisen)

Ausführung von JavaScript in DevTools

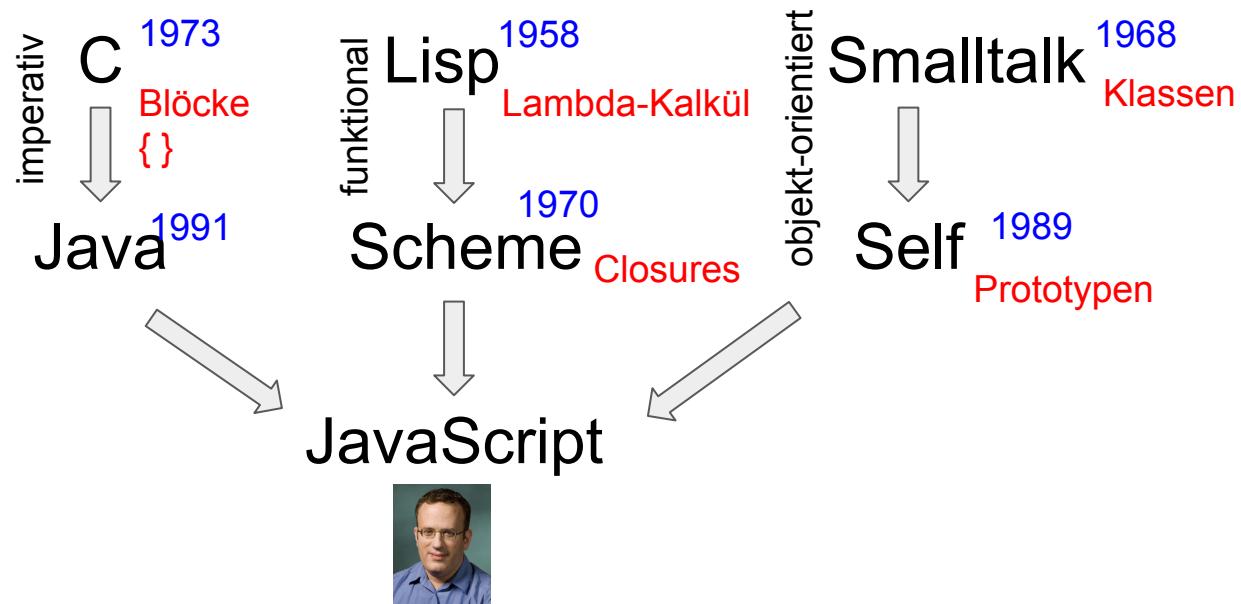
```
<!DOCTYPE html>
<html lang="de">
<head>
  <title>Hello World</title>
  <script src="hello_world.js"></script>
  <script>x += 1; console.log( x ); </script>
</head>
</html>
```

hello_world.js

```
console.log( "Hello World!" );
var x = 1;
console.log( x );
```



Programmiersprachen als Ideengeber für JavaScript



Die Kombination von **Closures** und **Prototypen** in JavaScript ist **einzigartig**.

Programmiersprachenparadigmen von JavaScript

1. interpretiert wie LISP und Smalltalk
 2. schwache, dynamische Typisierung
 - a. keine statische Typisierung
 3. multi-paradigmatisch
 - a. imperativ wie C und Java
 - i. gehört zur Familie der C-basierten Programmiersprachen
 - b. funktional wie Scheme
 - i. mit lexical Closures wie in Scheme
 - c. objekt-orientiert wie Self
 - i. prototyp-basiert: dynamische Bindung entlang der Prototypen-Kette
 - ii. vor ES6 hatte JS *keine Klassen*
 - d. Ereignis-getriebene (*event driven*) Funktionen
 - i. ideal zum Schreiben von Event-Handler
- **dynamisch** ⇒ zur Laufzeit Abfrage mit `typeof x`
 - **schwach** ⇒ nur grobe Einteilung (z.B. kein Typ "Array"), die sich zur Laufzeit ändern kann, z.B. `var x = 1; x = "string"; x = [1, 2];`

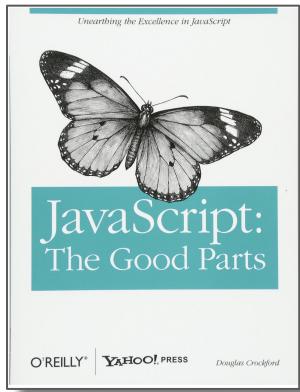
<https://en.wikipedia.org/wiki/JavaScript>



Douglas Crockford

Was ist der Kern des Verstehens von JavaScript?

JavaScript ist eine
funktionale Sprache.



JavaScript ist Lisp mit
dem Aussehen von C.



Douglas Crockford

Was ist der Kern des Verstehens von JavaScript?

JavaScript ist eine
objekt-orientierte Sprache.

You make **prototype** objects, and then ... make new instances. Objects are mutable in JavaScript, so we can augment the new instances, giving them new fields and methods. These can then act as prototypes for even newer objects. **We don't need classes to make lots of similar objects... Objects inherit from objects.**

What could be more object oriented than that?

<http://javascript.crockford.com/prototypal.html>

Wie baut man eine **kinderleichte** Sprache für alle? durch **Minimalismus**: nur Funktionen & Objekte

Funktionen

```
function f(x) {  
    return x*x;  
}
```

Objekte

```
{ a: 1, b: 2 }
```

Minimalismus: Was kann man weglassen?

Klasse ⇒ Objekt

Methode ⇒ Funktion

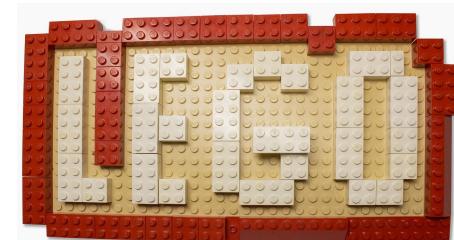
Konstruktor ⇒ Funktion

Array ⇒ Objekt

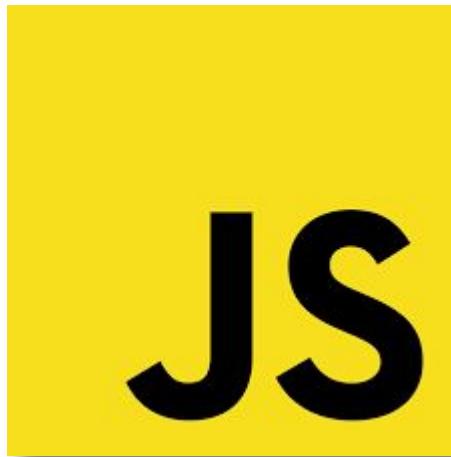
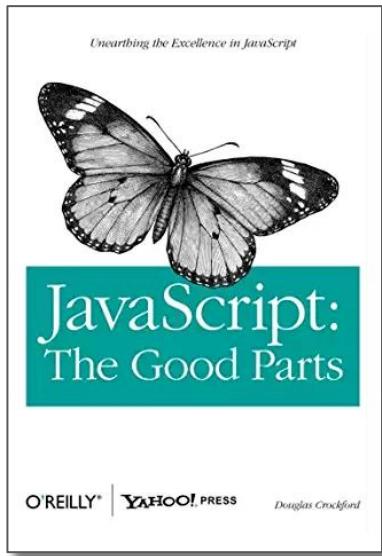
Objekt-Kapsel ⇒ Function Scope

Prinzipien:

- A. Minimalismus
- B. Universalität
- C. Rekursivität



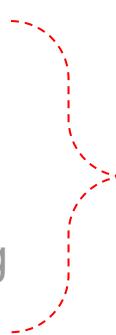
Die Programmiersprache JavaScript



The Bad Parts:

- Stillschweigende Globalisierung
- Automatic Semicolon Insertion
- this
- new
- Abstract Equality ==
- reserved words
- keine statische Typsicherheit

Gliederung

1. Typen und Operatoren
 2. Funktionen
 3. Variablen und Sichtbarkeit (Scoping)
 - a. Hoisting
 - b. this
 4. Objekte
 5. Klassen (ab ES6)
 6. Closures
 7. Funktionale Programmierung
- 
- erst nächste Woche*

Duck Typing

- "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."
- **Dynamische Typisierung:** Prüfung zur Laufzeit, ob Attribut und Methode vorliegt
- erlaubt **einheitliche Verarbeitung** trotz unterschiedlicher Typen, falls gemeinsame Attribute und Methoden vorliegen
- Duck Typing gibt es in JavaScript, Ruby, Python, Groovy, PHP

Funktion als Konstruktor

```
function Hund(name) {  
    this.name = name;  
    this.sprich = function () {  
        console.log( this.name + ' Wau' );  
    }  
}  
  
function Katze(name) {  
    this.name = name;  
    this.sprich = function () {  
        console.log( this.name + ' Miau' );  
    }  
}  
  
var list = [ new Hund('Bello'), new Katze('Garfield') ];  
  
for (var x in list){  
    list[x].sprich();  
}
```

1. Typen und der `typeof`-Operator

1. Sechs primitive Datentypen

1. Number
2. String
3. Boolean
4. Symbol (neu in ES 6)
5. **Undefined**
6. **Null**

Alles was **kein** primitiver Datentyp ist, ist ein Objekt.

2. Ein komplexer Datentyp: Object

1. user-defined
2. system-defined
3. Function
4. Array (`Array.isArray(x)`)
5. Error

```
var x = 1; x = "string"; x = [ 1, 2 ];
typeof x
```

Der Operator `typeof` liefert einen `String`

Typ	Rückgabewert
Undefined	"undefined"
Null	"object" (see below)
Boole'scher Wert	"boolean"
Zahl	"number"
Zeichenkette	"string"
Symbol (neu in ECMAScript 2015)	"symbol"
Host-Objekt (von der JS-Umgebung bereitgestellt)	implementierungsabhängig
Funktionsobjekt (implementiert [[Call]] nach ECMA-262)	"function"
Alle anderen Objekte	"object"

typeof - Operator

- Typ zur Laufzeit prüfen
Ergebnis nicht immer wie erwartet
- Unterscheidung Objekt vs. Array
Array.isArray([1,2,3])
- Erkennung Integer
Number.isInteger(123)
- Symbol erst ab ES6

Ausdruck	Ergebnis als String
typeof {}	'object'
typeof []	'object'
var f = function(){}; typeof f	'function'
typeof 'abc'	'string'
typeof 123	'number'
typeof 0.1	'number'
typeof true	'boolean'
typeof null	'object'
typeof undefined	'undefined'
typeof Symbol()	'symbol'

instanceof - Operator (nur für Objekte)

Ausdruck	Ergebnis
<pre>function Hund(name) { this.name = name; } bello = new Hund('Bello'); bello instanceof Hund</pre>	true
<pre>[1,2,3] instanceof Array</pre>	true
<pre>({a:1,b:2}) instanceof Object</pre>	true
<pre>(function() {}) instanceof Function</pre>	true
<pre>123 instanceof Number</pre>	false

1. Typen: 1.1. Der JavaScript-Typ Number

- nur ein Datentyp für Zahlen: Number
 - (in Java: integer, float, double, ...)
 - nur doppelpräzise 64-bit Werte im IEEE 754 Format: $\pm 2^{\text{Exponent}} \cdot \text{Mantisse}$

in Java "double"

IEEE 754 Converter (JavaScript), V0.22

Besonderheiten bei Number

Rundungsfehler

- $0.1 + 0.2 \neq 0.3$
- $0.1 + 0.2 === 0.3000000000000004$
- $9007199254740992 + 1 === 9007199254740992$

15 Nullen

Besondere Number-Werte: `NaN`, `Infinity` und `-Infinity`

- `NaN` === $1 * "x"$
- `Infinity` === $1/0$
- `-Infinity` === $-1/0$

Number

```
⋮ Console What's New Rendering
[ ] ⚡ top ▾ | ⚡ | Filter

> 3. === 3
<- true
> 3.0 === 3
<- true
> 3.toString()
✖ Uncaught SyntaxError: Invalid or unexpected token
> (3).toString()
<- "3"
> typeof 3
<- "number"
> typeof (3)
<- "number"
> typeof( new Number(3) )
<- "object"
> typeof( Number(3) )
<- "number"
> Number(3) === Number(3)
<- true
> new Number(3) === new Number(3)
<- false
```



```
1 Number('123') // 123
2 Number('12.3') // 12.3
3 Number('123e-1') // 12.3
4 Number('') // 0
5 Number('0x11') // 17
6 Number('0b11') // 3
7 Number('0o11') // 9
8 Number('foo') // NaN
9 Number('100a') // NaN
```

```
var x = (3).
  m toFixed(fractionDigits?: number) string
  m toExponential(fractionDigits?: number) string
  m toLocaleString(locales?: string | string[], o...) string
  m toPrecision(precision?: number) string
  m toString(radix?: number) string
  m valueOf() number
```

Konvertierung von Typen

explizit

implizit

Coercion

via Funktionen:

- `Number("3") === Number(3)`
- `Boolean(0) === false`
- `String(3) === "3"`

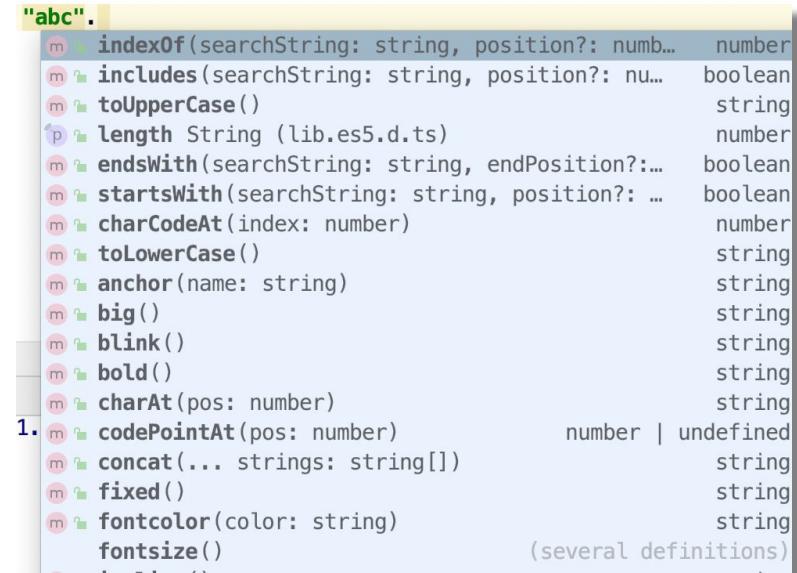
```
> 3*"4"  
< 12  
> "3"*"4"  
< 12  
> 3+"4"  
< "34"  
> 1 && true  
< true  
> true + false  
< 1  
>
```

1. Typen: 1.2. Der JavaScript-Typ String

```
var s1 = 'abc';
var s2 = "def";
s2 += `beide ${s1} ${s2} auch mehrzeilig`; // Backtick

console.log( "abc" === "abc" ); // true
console.log( "a" + "b" === "ab" ); // true
console.log( "abc".indexOf("b") === 1 ); // true
console.log( "abc".includes("b") ); // true
console.log( "abcd".slice(1,3) === "bc" ); // true
```

"Template String"



1. Typen: 1.2. Der JavaScript-Typ Boolean

true / false

truthy / falsy

```
var x = true;  
console.log( typeof x ); // "boolean"
```

```
x = 1;  
if ( x ) console.log( "1 is truthy" );
```

```
x = null;  
if ( !x ) console.log( "null is falsy" );
```

```
x = undefined;  
if ( !x ) console.log( "undefined is falsy" );
```

```
x = undefined;  
if ( x || ( x = 3 ) ) console.log( x ); // 3
```

truthy	falsy
true	false
1	0
' '	''
" "	""
{}	``
[]	NaN
-Infinity	null
Infinity	undefined
function(){} [] + []	

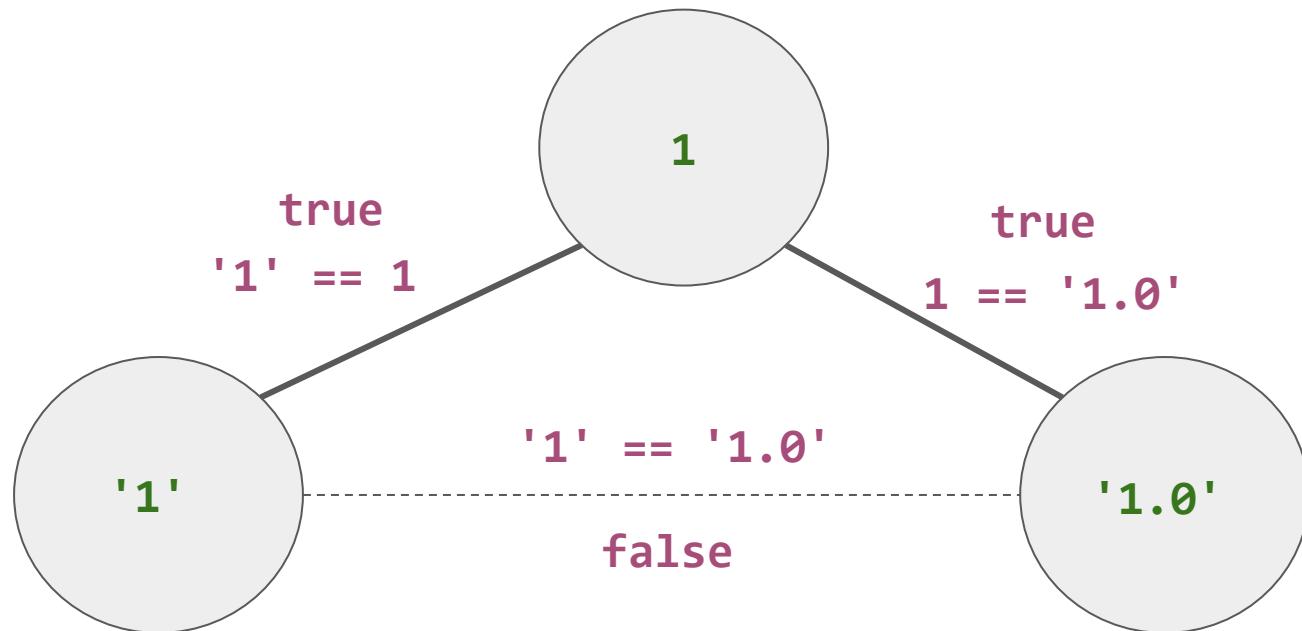
1. Typen: 1.3. Vergleichsoperatoren == und ===

==	===
abstrakte Gleichheit	strikte Gleichheit
intransitiv	transitiv
vor dem Vergleich der Werte wird eine Typangleichung (type coercion) durchgeführt	gleiche Typen und gleiche Werte
<pre>// true console.log(0 == '0'); console.log(false == '0'); console.log(null == undefined); console.log('\t\r\n' == 0); console.log(0 == "0");</pre>	<pre>// false console.log(0 === '0'); console.log(false === '0'); console.log(null === undefined); console.log('\t\r\n' === 0); console.log(0 === "0");</pre>

Empfehlung: Verwenden Sie möglichst nur die strikte Gleichheit ("===")

https://developer.mozilla.org/de/docs/Web/JavaScript/Vergleiche_auf_Gleichheit_und_deren_Verwendung

Intransitiver abstrakter Gleichheitsoperator ==



1. Typen: 1.3. Strikter Gleichheitsoperator ===

The comparison `x === y`, where `x` and `y` are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If `Type(x)` is different from `Type(y)`, return **false**.
2. If `Type(x)` is Number or BigInt, then
 - a. return `Number::equal(x, y)` or return `BigInt::equal(x, y)`.
3. If `Type(x)` is String, then
 - a. If `x` and `y` are exactly the same sequence of code units (same length and same code units at corresponding indices), return **true**; otherwise, return **false**.
4. If `Type(x)` is Boolean, then
 - a. If `x` and `y` are both **true** or both **false**, return **true**; otherwise, return **false**.
5. If `Type(x)` is Symbol, then
 - a. If `x` and `y` are both the same Symbol value, return **true**; otherwise, return **false**.
6. If `x` and `y` are the **same Object value**, return **true**. Otherwise, return **false**.

<https://tc39.es/ecma262/#sec-strict-equality-comparison>

1. Typen: 1.4. Abstrakter Gleichheitsoperator ==

The comparison `x == y`, where `x` and `y` are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If `Type(x)` is the same as `Type(y)`, then
 - a. Return the result of performing Strict Equality Comparison `x === y`.
2. If `x` is `null` and `y` is `undefined`, return **true**.
3. If `x` is `undefined` and `y` is `null`, return **true**.
4. If `Type(x)` is Number and `Type(y)` is String, return the result of the comparison `x == !ToNumber(y)`.
5. If `Type(x)` is String and `Type(y)` is Number, return the result of the comparison `!ToNumber(x) == y`.
6. If `Type(x)` is BigInt and `Type(y)` is String, then
 - a. Let `n` be `!StringToBigInt(y)`.
 - b. If `n` is `NaN`, return **false**.
 - c. Return the result of the comparison `x == n`.
7. If `Type(x)` is String and `Type(y)` is BigInt, return the result of the comparison `y == x`.
8. If `Type(x)` is Boolean, return the result of the comparison `!ToNumber(x) == y`.
9. If `Type(y)` is Boolean, return the result of the comparison `x == !ToNumber(y)`.
10. If `Type(x)` is either String, Number, BigInt, or Symbol and `Type(y)` is Object, return the result of the comparison `x == ToPrimitive(y)`.
11. If `Type(x)` is Object and `Type(y)` is either String, Number, BigInt, or Symbol, return the result of the comparison `ToPrimitive(x) == y`.
12. If `Type(x)` is BigInt and `Type(y)` is Number, or if `Type(x)` is Number and `Type(y)` is BigInt, then
 - a. If `x` or `y` are any of `NaN`, `+∞`, or `-∞`, return **false**.
 - b. If the mathematical value of `x` is equal to the mathematical value of `y`, return **true**; otherwise return **false**.
13. Return **false**.

Animation zum abstrakten Gleichheitsoperator

Code on [Github](#)

JavaScript Equality Algorithms

Made by nem035

X

Primitive Non Primitive

null

Y

==

Primitive Non Primitive

undefined

Run

Algorithm Animation

```
null == undefined
```

Checking if x and y have the same type
Checking if x is null and y is undefined
true



<https://nem035.github.io/js-equality-algorithms/>

JavaScript Equality Algorithms

Made by nem035

X

Primitive Non Primitive

0

Y

Primitive Non Primitive

'0'

Run

Algorithm Animation

```
0 == '0'
```

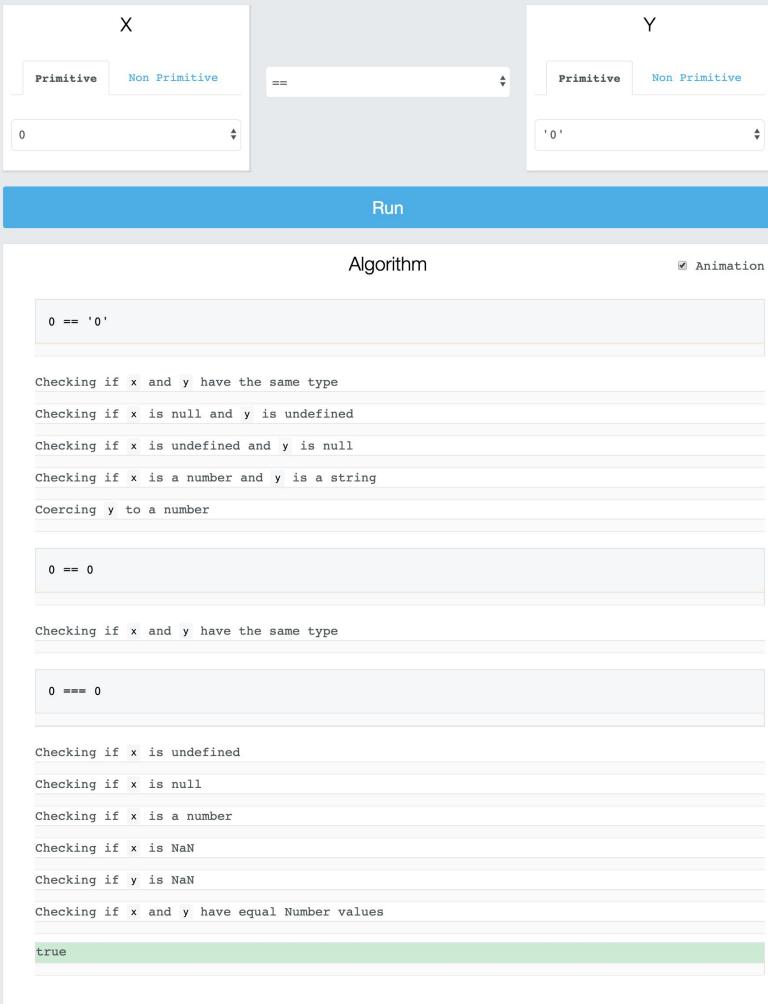
Checking if x and y have the same type
Checking if x is null and y is undefined
Checking if x is undefined and y is null
Checking if x is a number and y is a string
Coercing y to a number

```
0 == 0
```

Checking if x and y have the same type

```
0 === 0
```

Checking if x is undefined
Checking if x is null
Checking if x is a number
Checking if x is NaN
Checking if y is NaN
Checking if x and y have equal Number values
true



JavaScript "==" Equality Table

The figure is a heatmap illustrating the results of the == operator for various JavaScript values. The x-axis and y-axis both list the same set of values: null, undefined, NaN, Infinity, -Infinity, true, false, 1, 0, -1, "1", "0", "-1", "", "true", "false", and strings. A green square indicates that the comparison a == b is true, while a white square indicates it is false. The heatmap shows that most values are considered equal to themselves and to NaN, but not to other types or values like infinity or -infinity.

2. Funktionen

Funktionen

```
function f(x) {  
    return x*x;  
}
```

Funktionen gehören zu den erfolgreichsten Abstraktionen der Mathematik

$$f(x) = x^2$$

Wie würde eine einfachste
funktionale Programmiersprache aussehen?

2.1. Funktionsparameter

- Parameter sind optional

```
function f(a,b){  
    return a + ( b || 0 );  
}  
f(1); // -> 1
```

- Übergabe "pass by reference" bei Objekten, sonst "pass by value"

```
var x = {};  
function set(y) {  
    y.a = 2;  
}  
set(x); // -> { a: 2 }
```

```
var x = 1;  
function set( y ) {  
    y = 2;  
}  
set( x ); // -> 1
```

Nur 1 Parameter vom Typ Object

- Keyword-Parameter

→ Idiom

→ Entwurfsmuster

```
function pay( param ) {  
    payFrom( param.sender ).to( param.receiver ).amount( param.amount );  
}  
  
pay({ sender: 1147, receiver: 4812, amount: 1370 });
```

Beim Aufruf werden die Keywords genannt. Damit wird die Verwechslungsgefahr vermindert. Das wird z.B. in APIs eingesetzt. Reihenfolge egal.

2.2. Funktionen als Werte

Funktion

- als Wert einer Variablen

```
var sinn_des_universums = function (){ return 42; };
```

- als Wert eines Parameters

```
function twice( f ) { return function( x ){ return f( x, x ); } }
```

- als Rückgabewert einer Funktion

```
function sinn() { return function () { return 42; } }
```

2.3. Funktionsdeklarationen haben 2 Aufgaben:

- 1. Aufgabe: eine Berechnungsvorschrift für eine Funktion zu speichern
- 2. Aufgabe: einen Namensraum (Scope) zu definieren

*Function
Scoping*

```
function f(x){  
    var y = 14;  
}  
  
console.log( y );
```

ReferenceError: y is not defined

- Der Function Scope wird nur bei der **function**-Syntax erzeugt, nicht bei der " $=>$ " - Notation.

2.4 IFFE Pattern

Eine IIFE (Immediately Invoked Function Expression) ist eine JavaScript-Funktion, die ausgeführt wird, sobald sie definiert ist. Sie dient nur dazu, einen neuen **Namensraum** zu öffnen und zu kapseln.

```
1 (function () {  
2     var aName = "Barry";  
3 })();  
4 // Variable aName is not accessible from the outside scope  
5 aName // throws "Uncaught ReferenceError: aName is not defined"
```

Name der Funktion unwichtig

neuer Namensraum

<https://developer.mozilla.org/de/docs/Glossary/IIFE>

<https://addyosmani.com/resources/essentialjsdesignpatterns/book/#detailnamespacing>

2.5 JavaScript-Interpreter arbeitet in 2 Phasen

- 1. Phase: Deklarationen sammeln und merken.
- 2. Phase: Anweisungen ausführen.

korrekt

```
foo(); // 5  
  
function foo() {  
    console.log( 5 );  
}
```

Deklaration

Fehler:

```
fun(); // ReferenceError: fun is not defined  
  
var fun = function() {  
    console.log( 5 );  
};
```

Das ist eine Anweisung und keine Deklaration!

2.6 Hoisting

- Sämtliche Deklarationen werden unsichtbar an den Anfang ihres Sichtbarkeitsbereichs (*scope*) verschoben.
- Der Programmierer braucht dies nicht zu tun. (Hoisting ist gedacht als Arbeitserleichterung.)
- Konsequenz: Man kann seine Hilfsfunktionen auch ans Ende stellen.

```
function foo() {  
    bar();  
    var x = 1;  
    function bar(){  
        console.log( x );  
    }  
}
```



// is actually interpreted like this:

```
function foo() {  
    var x;  
    var bar = function(){  
        console.log( x );  
    };  
    bar();  
    x = 1;  
}
```

Deklarationen werden an den Anfang verschoben.

Übung zu Hoisting

Welche Ausgabe auf der Konsole?

```
var foo = 1;

function bar() {

    var foo = foo ? 1 : 2;

    console.log( foo );

}

bar();
```



Lösung zu Hoisting

```
var foo = 1;

function bar() {
    var foo = foo ? 1 : 2;
    console.log( foo );
}

bar();
```

Hoisting

Konsole: 2

```
var foo = 1;
```

```
function bar() {
```

```
    var foo = undefined;
```

```
    foo = foo ? 1 : 2;
```

```
    console.log( foo );
```

```
}
```

```
bar();
```

falsey

Konsole: 2

3. Objekte

Objekte gehören zu den erfolgreichsten Abstraktionen in Programmiersprachen.

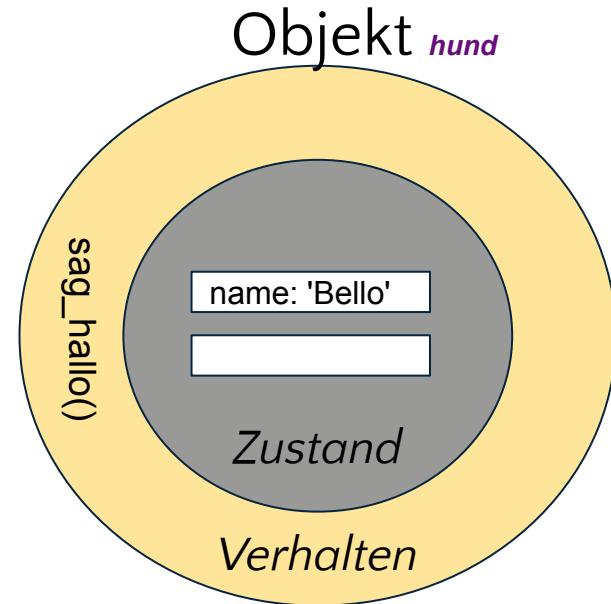
3.1. Objektliterale

```
var x = {  
  a: 1,  
  b: 2,  
  c: [ 3, 4, { d: 5 } ]  
};
```

```
var hund = {  
  name: "Bello",  
  sag_hallo: function () {  
    return "Bello sagt " + "Wau";  
  }  
};  
  
console.log( hund.sag_hallo() );
```

nested arrays & objects

Objekt ohne Klasse



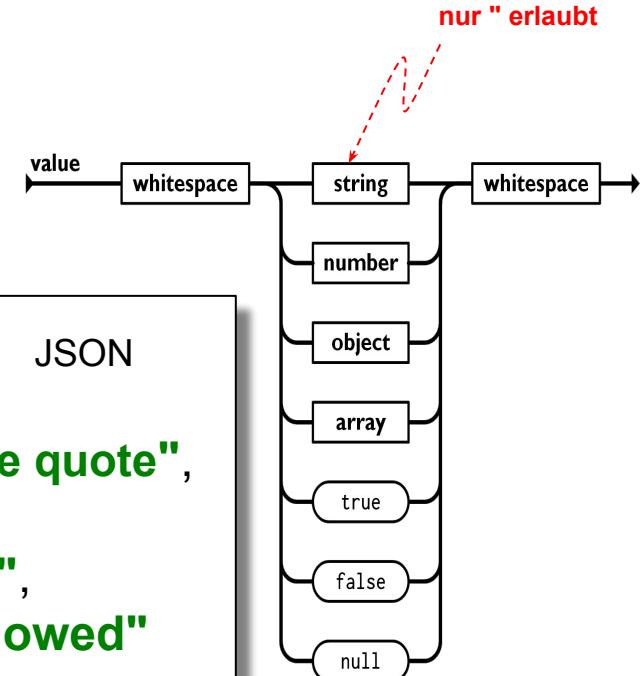
3.2. Objekt-Literale versus JSON

```
var world = "World";  
  
var x = {  
  a: 1,  
  b: 'strings',  
  c: [ 'arrays', 3, 4, {  
    d: "nested objects",  
    e: `backtick strings`  
  }  
],  
f1: function( param ){  
  return "Hello " + param;  
},  
"f2": () => "Goodbye " + world  
};
```

JavaScript

```
{  
  "a": 1,  
  "key": "keys with double quote",  
  "c": [ "arrays", 3, 4, {  
    "d": "nested objects",  
    "e": "backtick not allowed"  
  }  
],  
  "f": "functions not allowed"  
}
```

Datei-Format



<https://www.json.org/>

3.3. Fabrikfunktionen zur Erzeugung von Objekten

Statt Konstruktor eine Funktion,
die ein Objekt berechnet

Methoden sind Funktionen
des berechneten Objektes

Konstruktion eines neuen Hunde-Objekts

Die Erzeugung von
Objekten geht auch
ohne Klassen.

```
function makeDog( name ) {  
  var new_dog = { name: name };  
  new_dog.toString = function () {  
    return "Hund " + new_dog.name;  
  };  
  return new_dog;  
}  
  
var bello = makeDog( "Bello" );  
console.log( bello.name );  
bello.name = "Bolle";  
console.log( bello.toString() );
```

erzeugtes Hunde-Objekt

name kann von außen
überschrieben werden.

Primitive sind immutable. Objekte sind mutable.

3.4. Kapselung mittels Function Scope

name wird über Parameter gesetzt, geheim gemerkt und in der Methode "toString()" lesend verwendet. Ein nachträgliches Überschreiben von "name" ist unmöglich.

=> Privatisierung von "name"

Objekt-Kapselung
ohne Klassen

```
function makeDog( name ) {  
    return {  
        toString: function(){ return "Hund " + name }  
    };  
}
```

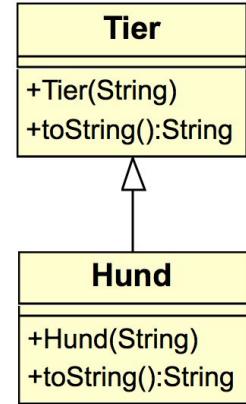
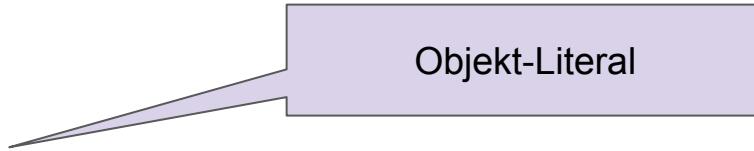
```
var bello = makeDog( "Bello" );  
console.log( bello.toString() ); // Hund Bello
```



Douglas Crockford

3.5. Vererbung ohne Klassen

```
function Tier(name) {  
    return {  
        name: name,  
        toString: function () {  
            return "Tier(" + name + ")";  
        }  
    };  
}  
  
function Hund(name) {  
    var that = Tier(name);  
    that.toString = function () {  
        return "Hund(" + name + ")";  
    };  
    return that;  
}  
  
var bolle = Hund("Bolle");  
console.log( bolle.toString() );
```



```
function Funktionaler_Konstruktor( param ) {  
    var that = innerMaker( param );  
    var secret = ...  
    that.method = function () {  
        do_something( param, that, secret );  
    };  
    return that;  
}
```

Douglas Crockford's
"Power Constructor"

3.6. this

this → verweist auf den Besitzer der Funktion

- in globalen Funktionen:
 - das globale Objekt = die Umgebung
 - im Browser: → "window"
- in Konstruktoren:
 - → das konstruierte Objekt
- in Methoden:
 - → das besitzende Objekt
- in onclick-Methoden in HTML:
 - → das HTML-Element
- in eval
 - → Besitzer im Kontext

```
> function Point(x, y) {  
  this.x = x;  
  this.y = y;  
}  
var p = new Point(7, 5);  
console.log(p);  
  
▼Point {x: 7, y: 5} ⓘ  
  x: 7  
  y: 5
```

```
function Point(x, y) {  
  this.x = x;  
  this.y = y;  
}  
var p = Point(7, 5); // we forgot new!  
console.log(p === undefined); // true  
  
// Global variables have been created:  
console.log(x); // 7  
console.log(y); // 5
```

<http://2ality.com/2014/05/this.html>

3.6. **this** ist dynamisch gebunden

- wird an den **Besitzer** der Funktion **zur Laufzeit** gebunden

```
1 var o = {  
2     prop: 37,  
3     f: function() {  
4         return this.prop;  
5     }  
6 };  
7  
8 console.log(o.f()); // logs 37
```

```
1 var o = {prop: 37};  
2  
3 function independent() {  
4     return this.prop;  
5 }  
6  
7 o.f = independent;  
8  
9 console.log(o.f()); // logs 37
```

this in Event-Handlern

```
<!DOCTYPE html>
<html lang="de">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Title</title>
</head>
<body>

<h1 onclick="this.style.color='red';">Überschrift wird bei Click rot</h1>

</body>
</html>
```

Überschrift wird bei Click rot

Überschrift wird bei Click rot

this
vom Typ HTMLElement



<https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>

this in Event-Handlern

```
<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Title</title>
  <script>
    function redify( elem ){
      elem.style.color='red';
    }
  </script>
</head>
<body>

<h1 onclick="redify( this );">Überschrift wird bei Click rot</h1>

</body>
```

Überschrift wird bei Click rot

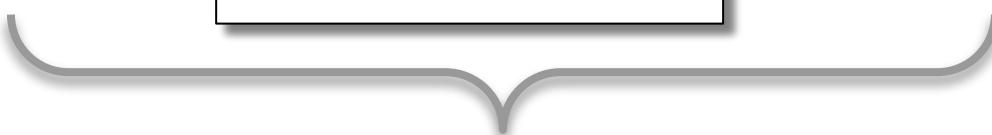
Problem:

Ohne Vererbung sind Objekte speicherintensiv

```
var bello = {  
  name: "Bello",  
  sag_hallo: function () {  
    return "Bello sagt Wau";  
  }  
};
```

```
var rocky = {  
  name: "Rocky",  
  sag_hallo: function () {  
    return "Rocky sagt Wau";  
  }  
};
```

```
var fifi = {  
  name: "Fifi",  
  sag_hallo: function () {  
    return "Fifi sagt " + "Wau";  
  }  
};
```



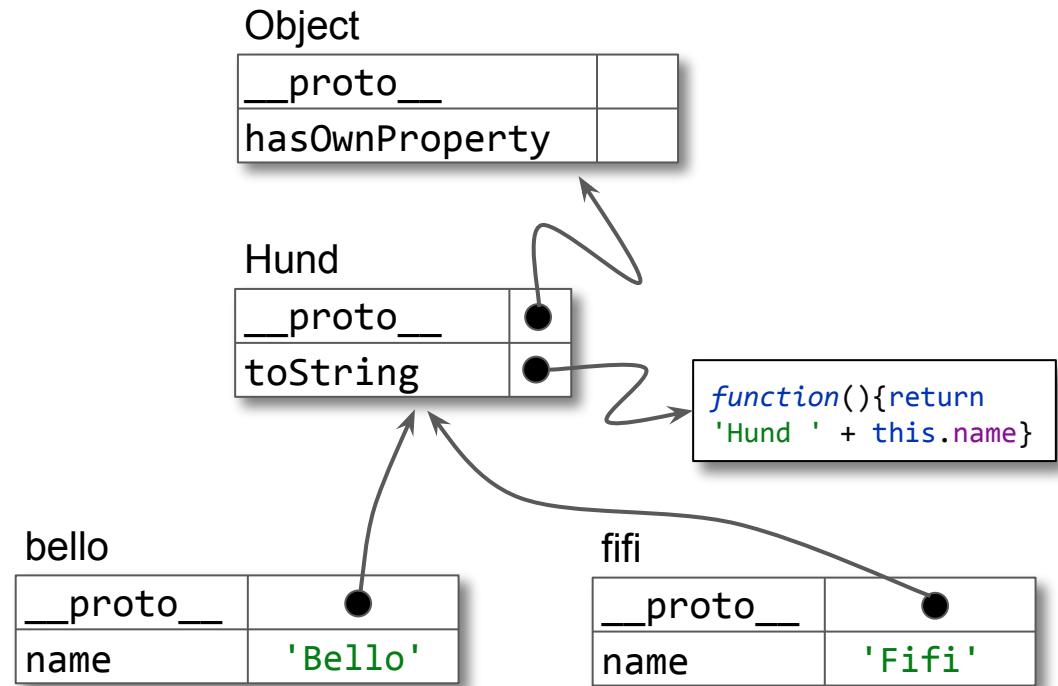
Die Methode "sag_hallo" belegt **3-mal** Speicherplatz.

bessere Lösung mit Prototyp:

```
var Hund = {  
  sag_hallo: function () {  
    return this.name + " sagt Wau";  
  }  
};  
  
var bello = {  
  __proto__: Hund,  
  name: 'Bello',  
};  
  
var fifi = {  
  __proto__: Hund,  
  name: 'Fifi',  
};
```

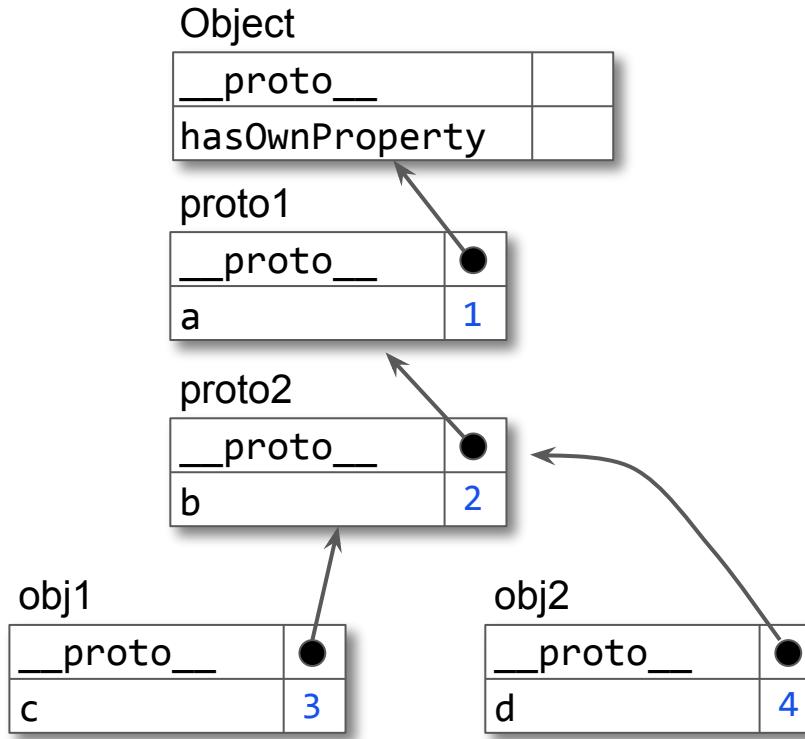
3.7 Vererbung entlang der Prototypenkette

```
var Hund = {  
    toString: function() {  
        return 'Hund ' + this.name;  
    }  
};  
  
var bello = {  
    __proto__: Hund,  
    name: 'Bello',  
};  
  
var fifi = Object.create( Hund );  
fifi.name = "Fifi";
```



3.7 Vererbung entlang der Prototypenkette

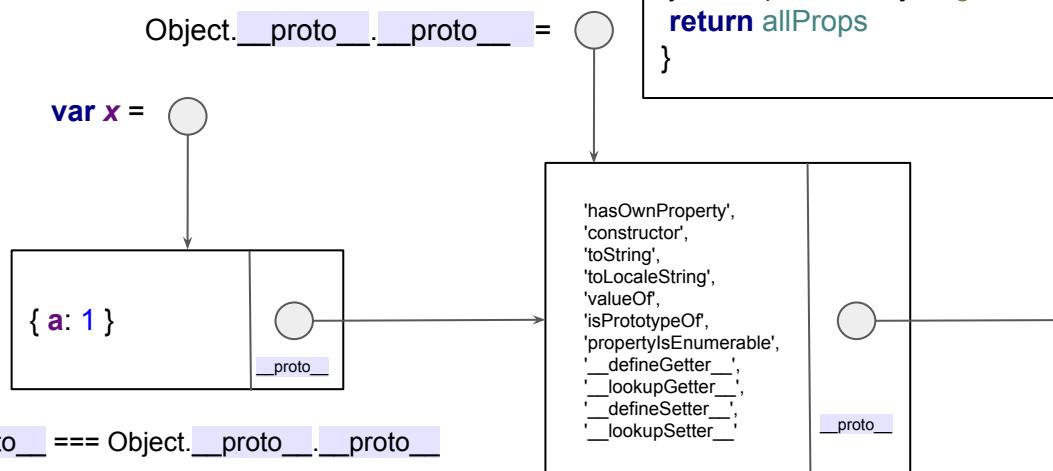
```
var proto1 = {  
  a: 1,  
};  
  
var proto2 = {  
  __proto__: proto1,  
  b: 2,  
};  
  
var obj1 = {  
  __proto__: proto2,  
  c: 3  
};  
  
var obj2 = {  
  __proto__: proto2,  
  d: 4  
};
```



```
> obj1.c  
< 3  
> obj2.c  
< undefined  
> obj1.b  
< 2  
> obj2.b  
< 2  
> obj1.a  
< 1  
> obj2.a  
< 1  
> obj1.hasOwnProperty("c")  
< true  
> obj1.hasOwnProperty("a")  
< false
```

Iteration über Prototypen-Kette

- Aufsammeln aller PropertyNames entlang der Prototypen-Kette



```
var x = { a: 1 };
console.log( Object.getPrototypeOf( x ) );
console.log( getAllProperties( x ) );

function getAllProperties(obj){
  var allProps = []
    , curr = obj
  do {
    allProps.push( Object.getOwnPropertyNames(curr) )
  } while( curr = Object.getPrototypeOf(curr) )
  return allProps
}
```

```
{}
[[ 'a'],
 [ 'hasOwnProperty',
  'constructor',
  'toString',
  'toLocaleString',
  'valueOf',
  'isPrototypeOf',
  'propertyIsEnumerable',
  '__defineGetter__',
  '__lookupGetter__',
  '__defineSetter__',
  '__lookupSetter__',
  '__proto__']]
```

Funktionen

```
function f(x) {  
    return x*x;  
}
```

Objekte

```
{ a: 1, b: 2 }
```

Zusammenfassung: JavaScript ist gut für
"Funktionale Programmierung mit Objekten"

1. Objekte und Funktionen
 - a. einfach und gleichzeitig skalierbar
2. Lego-Prinzip bei Objekten, Funktionen
 - a. Objekte mit Funktionen
 - b. Funktionen für Objekt-Konstruktion
 - c. Funktionen für Objekt-Verarbeitung

Grundregel beim Programmieren:

Nicht raten, sondern
messen und prüfen!

Debugger-Demo

JavaScript: The Bad Parts & Designfehler

Douglas Crockford bezeichnet z.B. die **automatische Semikolon-Einfügung** als einen „furchtbaren Designfehler“ von JavaScript und empfiehlt von wenigen Ausnahmen abgesehen, das Semikolon immer explizit anzugeben.

```
function test1() {  
    return  
    {  
        problem: 'ERROR'  
    }  
}
```

test1() gibt undefined zurück.

SILENT ERROR !!!

```
function test2() {  
    return {  
        problem: 'ERROR'  
    }  
}
```

test2() gibt das Objekt zurück.

⇒ Empfehlung: Geschweifte Klammer rechts!

Semikolon-Regeln in JavaScript

7.9.1 Rules of Automatic Semicolon Insertion

There are three basic rules of semicolon insertion:

1. When, as the program is parsed from left to right, a token (called the *offending token*) is encountered that is not allowed by any production of the grammar, then **a semicolon is automatically inserted** before the offending token if one or more of the following conditions is true:
 - The offending token is separated from the previous token by at least one *LineTerminator*.
 - The offending token is `}`.
2. When, as the program is parsed from left to right, the end of the input stream of tokens is encountered and the parser is unable to parse the input token stream as a single complete ECMAScript *Program*, then a semicolon is automatically inserted at the end of the input stream.
3. When, as the program is parsed from left to right, a token is encountered that is allowed by some production of the grammar, but the production is a *restricted production* and the token would be the first token for a terminal or nonterminal immediately following the annotation “[no *LineTerminator* here]” within the restricted production (and therefore such a token is called a restricted token), and the restricted token is separated from the previous token by at least one *LineTerminator*, then a semicolon is automatically inserted before the restricted token.

<http://ecma-international.org/ecma-262/5.1/#sec-7.9>

https://de.wikipedia.org/wiki/Automatic_semicolon_insertion

Gleichheitsprüfung in JavaScript mit ==

11.9.3 The Abstract Equality Comparison Algorithm

The comparison `x == y`, where `x` and `y` are values, produces `true` or `false`. Such a comparison is performed as follows:

1. If `Type(x)` is the same as `Type(y)`, then
 - If `Type(x)` is Undefined, return `true`.
 - If `Type(x)` is Null, return `true`.
 - If `Type(x)` is Number, then
 - i. If `x` is `Nan`, return `false`.
 - ii. If `y` is `Nan`, return `false`.
 - iii. If `x` is the same Number value as `y`, return `true`.
 - iv. If `x` is `+0` and `y` is `-0`, return `true`.
 - v. If `x` is `-0` and `y` is `+0`, return `true`.
 - vi. Return `false`.
 - If `Type(x)` is String, then return `true` if `x` and `y` are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return `false`.
 - If `Type(x)` is Boolean, return `true` if `x` and `y` are both `true` or both `false`. Otherwise, return `false`.
 - Return `true` if `x` and `y` refer to the same object. Otherwise, return `false`.
2. If `x` is `null` and `y` is `undefined`, return `true`.
 3. If `x` is `undefined` and `y` is `null`, return `true`.
 4. If `Type(x)` is Number and `Type(y)` is String,
 - return the result of the comparison `x == ToNumber(y)`.
 5. If `Type(x)` is String and `Type(y)` is Number,
 - return the result of the comparison `ToNumber(x) == y`.
 6. If `Type(x)` is Boolean, return the result of the comparison `ToNumber(x) == y`.
 7. If `Type(y)` is Boolean, return the result of the comparison `x == ToNumber(y)`.
 8. If `Type(x)` is either String or Number and `Type(y)` is Object,
 - return the result of the comparison `x == ToPrimitive(y)`.
 9. If `Type(x)` is Object and `Type(y)` is either String or Number,
 - return the result of the comparison `ToPrimitive(x) == y`.
 10. Return `false`.

```
' ' == '0' // false  
0 == '' // true  
0 == '0' // true  
false == 'false' // false  
false == '0' // true  
false == undefined // false  
false == null // false  
null == undefined // true  
' \t\r\n ' == 0 // true
```

== ist nicht transitiv

Type Coercion



Empfehlung:
Verwenden Sie nur ===

<http://www.ecma-international.org/ecma-262/5.1/#sec-11.9.3>

Vergleichsoperatoren ==, === und Object.is(x, y)

Sameness Comparisons

x	y	==	===	Object.is
undefined	undefined	true	true	true
null	null	true	true	true
true	true	true	true	true
false	false	true	true	true
'foo'	'foo'	true	true	true
0	0	true	true	true
+0	-0	true	true	false
0	false	true	false	false
""	false	true	false	false
""	0	true	false	false
'0'	0	true	false	false
'17'	17	true	false	false

[1, 2]	'1,2'	true	false	false
new String('foo')	'foo'	true	false	false
null	undefined	true	false	false
null	false	false	false	false
undefined	false	false	false	false
{ foo: 'bar' }	{ foo: 'bar' }	false	false	false
new String('foo')	new String('foo')	false	false	false
0	null	false	false	false
0	NaN	false	false	false
'foo'	NaN	false	false	false
NaN	NaN	false	false	true

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness

JavaScript Schlüsselworte "*reserved words*"

break
case
catch
class
const
continue
debugger
default
delete
do
else
export
extends
finally
for

function
if
import
in
instanceof
let
new
return
super
switch
this
throw
try

typeof
var
void
while
with
yield

Mit Ergänzungen für
andere Versionen von
JavaScript

https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Lexical_grammar

Hoisting in Funktionen

Alle Variablen-Deklaration
werden vom Interpreter an den
Anfang der Funktion verschoben.

```
function f() {  
    for (var i = 0; i<3; i++){  
        for (var i = 0; i<3; i++){  
            console.log( i );  
        }  
    }  
}  
  
f(); // -> 0,1,2
```

Warum wird die Schleife
nur 3-mal durchlaufen?

```
var scope = "global";  
function f() {  
    console.log(scope); // Prints "undefined", not "global"  
    var scope = "local"; // Variable initialized here, but defined everywhere  
    console.log(scope); // Prints "local"  
}  
f()
```



verschieben

```
var scope = "global";  
function f() {  
    var scope = undefined;  
    console.log(scope); // Prints "undefined", not "global"  
    scope = "local"; // Variable initialized here, but defined everywhere  
    console.log(scope); // Prints "local"  
}  
f()
```

Stillschweigende Globalisierung

```
var x = 2;  
f(4); // Funktion mit Seiteneffekt !!!  
console.log( x ); // -> 3
```

```
function f(y) {  
    x = 3; // -> var fehlt, daher wird das globale x genommen  
    console.log( x ); // -> 3  
}
```

**Vergessen von var
wird hart bestraft!**

- Wenn **var** fehlt, wird global angenommen, (wird stillschweigend globalisiert)
d.h. **var** x wird zu einer Property des globalen Objektes
- z.B. im Browser heißt das globale Objekt "**window**"
und repräsentiert den Bereich eines Tabulators im Browser

Bei Fragen: Stackoverflow

2.106.651 JavaScript questions

Newest 'javascript' Questions

Questions tagged [javascript]

For questions regarding programming in ECMAScript (JavaScript/JS) and its various dialects/implementations (excluding ActionScript). This tag is rarely used alone but is most often associated with the tags [node.js], [jquery], [json], and [html].

Learn more... Top users Synonyms (14) javascript jobs

2,106,651 questions

Ask Question

0 votes 0 answers 2 views

0 votes 0 answers 4 views

Uncatched: TypeError .unon not a function

So I'm currently working one of our corporate websites to jQuery v3.0+ standard from jQuery v1.8.1 to meet the WordPress 5.6 update in december which will deprecate the jQuery-migrate integration. ...

asked 52 secs ago by SLE 618 ● 1 ● 10

javascript jquery wordpress

how can i get values from xml namespace tags using php [duplicate]

<Invoice xmlns="urn:oasis:names:specification:ubl:schema:xsd:Invoice-2" xmlns:cac="urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2" xmlns:cbc="urn:...>

asked 2 mins ago by laiu ks 1

javascript php json xml codeigniter

The Overflow Blog

- The Overflow #43: Simulated keyboards
- How to communicate more deliberately and efficiently when working remotely

Featured on Meta

- Responding to the Lavender Letter and commitments moving forward

best it GmbH & Co. KG

Digital Agency

Fullstack Developer Siegburg, Deutschland €45k - €55k

javascript php

Fullstack Developer Siegburg, Deutschland €45k - €55k

javascript php

Fullstack Developer

Kopieren Sie jedoch nicht Code aus Stackoverflow, ohne diesen zu verstehen!
Stackoverflow enthält auch fehlerhaften Code!

Literatur

- Mozilla Developer Network (MDN)
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript
 - <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- Douglas Crockford: JavaScript: The Good Parts
 - <http://javascript.crockford.com/>
 - <http://crockford.com/>
 - <http://crockford.com/javascript/encyclopedia/>
 - <http://javascript.crockford.com/survey.html>
 - <http://javascript.crockford.com/tdop/tdop.html>
- Kyle Simpson
 - [You don't know JS](#)
- Dr. Axel Rauschmayer
 - <http://2ality.com>
 - <http://exploringjs.com>
- Addy Osmani: Learning JavaScript Design Patterns
 - <https://addyosmani.com/resources/essentialjsdesignpatterns/book>

