

Web Engineering + Design 1

JAVASCRIPT

Michael Gfeller

Intro

- JavaScript ist mächtig
- JavaScript ist super
- JavaScript wird immer besser
- JavaScript ist gefährlich



Bild-Quelle: <http://engineering.wix.com/wp-content/uploads/2015/04/mluuwgx-1024x576.jpg>

Die Teilnehmer...

- ... können ihre Kenntnisse von Java auch auf JavaScript anwenden
- ... kennen die Grundlagen von JavaScript und können diese anwenden
- ... können die Spezialitäten von JavaScript erklären und nachvollziehen
- ... kennen den Unterschied zwischen ECMAScript und JavaScript
- ... kennen die Natur von JavaScript und können die Punkte an Beispiele erklären
- ... können Array-Funktionen anwenden und kennen die Unterschiede zwischen den verschiedenen Iterationsvarianten
- ... können Arrow / Lambda-Funktionen anwenden
- ... können die neuen Features von ECMAScript 6 anwenden
- ... können JavaScript interpretieren

Die Teilnehmer...

- ... können (automatische) Typenumwandlungen nachvollziehen und erklären
- ... können JavaScript Programmieren 😊

- **Einstieg**
- **JavaScript ausführen**
- **JavaScript**
- **Primitive Typen**
 - Booleans
 - Number
 - String
 - Rechnen mit primitives
 - The Abstract Equality Comparison Algorithm
 - null != undefined
- **Reference Typen**
 - Array
 - Simple Object
 - Functions Teil 1

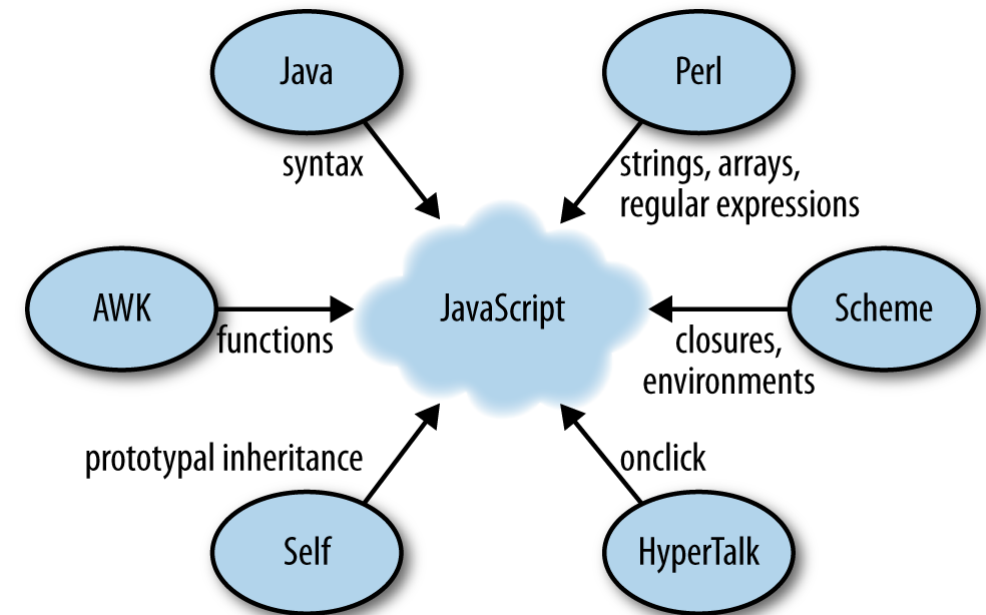
Inhaltsverzeichnis Woche 2

- **Functions Teil 2**
- **Scope**
- **Context**
- **Use Strict**
- **Arrow Function**
- **Constructor Function**
- **JavaScript Features**

EINSTIEG

JavaScript Erfolgsgeschichte

1995	JavaScript in 10 Tagen entwickelt
1997	Dynamic HTML: HTML kann geändert werden
1999	XMLHttpRequest: Nachträgliches laden von Daten vom Server
2001	JSON
2005	Google Maps erscheint: Perfektionierte Dynamic HTML und XMLHttpRequest. => AJAX
2006	JQuery
2007	WebKit
2008	V8 – Engine (Chrome)
2009	NodeJs
2009	PhoneGap / ChromeOS



ECMAScript vs. JavaScript

- **ECMAScript ist der offizielle Name für JavaScript**
- **Sun / Oracle haben die Rechte an dem Namen «JavaScript»**

Generell:

- **JavaScript wird für die Programmiersprache verwendet**
- **ECMAScript ist der Name für die Sprachdefinition**
 - z.B. Die aktuelle Version von JavaScript ist ECMAScript 6 (2015) und ECMAScript 7 wird entwickelt.

ECMAScript Versionen

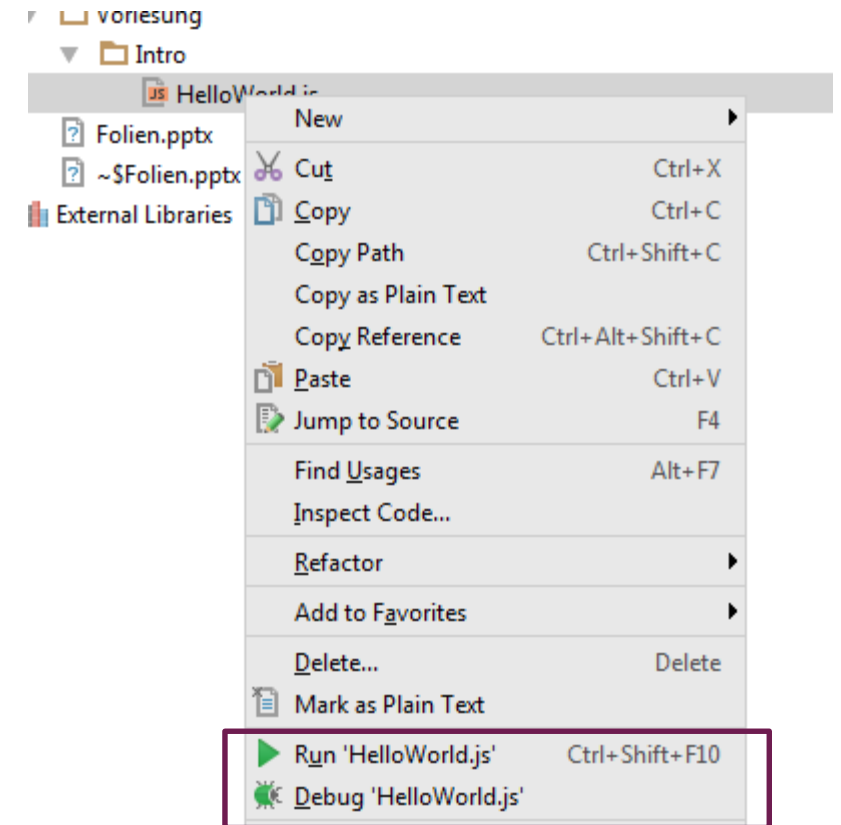
Version	publiziert am	Unterschiede zur Vorgängerversion
1	Juni 1997	erste Version
2	Juni 1998	Änderungen zwecks Kompatibilität zum internationalen Standard ISO/IEC 16262
3	Dezember 1999	Neu sind reguläre Ausdrücke, bessere Verarbeitung von Zeichenketten, Kontrollfluss, Fehlerbehandlung mit try/catch , bessere Fehlerbehandlung, bessere Formatierung bei der Ausgabe von Zahlen usw.
4	abgebrochen	Wegen Uneinigkeit in Bezug auf die Zukunft der Sprache wurde die weitere Entwicklung des komplexen Entwurfes zu ECMAScript 4 eingestellt. Einige Ideen werden in ES6 wieder aufleben.
5	Dezember 2009	Im „strict mode“ wird eine erweiterte Fehlerprüfung eingeschaltet. Unklare Sprachkonstrukte von ECMAScript 3 werden entschärft und neue Features wie getter- und setter-Methoden, Unterstützung von JSON usw. hinzugefügt.
5.1	Juni 2011	Entspricht dem internationalen Standard ISO/IEC 16262:2011, Version 3
6	Juni 2015	Neue Syntax für komplexe Applikationen wie Klassen und Module , die aber mit ähnlicher Terminologie wie in ECMAScript 5 (strict mode) definiert werden können. Neue Sprachbestandteile wie for/of-Schleifen, teilweise an Python angelehnte Syntax usw. Der Codename lautet „Harmony“ und wurde bis kurz vor Verabschiedung als „ECMAScript 6“ bezeichnet
7	in Arbeit	Weiterführung der mit ES6 begonnenen Änderungen

Quelle: <https://en.wikipedia.org/wiki/ECMAScript#Versions>

- **Grösster Schritt von JavaScript**
- **Teil der Vorlesung**
- **Browser Support wird besser und besser.**
 - <https://kangax.github.io/compat-table/es6/>
- **ECMAScript 6 kann schon jetzt verwendet werden mit sogenannten «JavaScript Compiler»**
 - <https://babeljs.io/>
 - <https://github.com/google/traceur-compiler>
- **Zusammenfassung der ECMAScript 6 Features mit Beispiele**
 - <http://es6-features.org/>

JAVASCRIPT AUSFÜHREN

- Nutzt die V8 Engine von Chrome
- Kann JavaScript ausführen
- Kann JavaScript **debuggen**
- Installation <https://nodejs.org/en/>
- Wird von WebStorm genutzt um JavaScript auszuführen
- Kann über die Command-Line angesteuert werden
- Folgende Parameter aktivieren ES6 Features
 - --harmony
 - --harmony_destructuring
 - Aktuelle liste der Features: <https://nodejs.org/en/docs/es6/>
- Node.js kann als Web Server fungieren => später in der Vorlesung



Hello World

helloWorld.js

```
console.log("Hello World");
```


Hello World 2

- JavaScript ist (in den meisten Fällen) gut lesbar
- Was gibt folgender Code aus?

```
var y = 3;  
var x = 3 + y;  
var value = add(x, y);  
  
(value > 10) ? console.log("Big") : console.log("Small");  
  
function add(a, b) {  
    return a + b;  
}
```

JAVASCRIPT

Die Natur von JavaScript

- **It's dynamic**

- Objekte können verändert werden z.B. Methoden überschrieben werden

- **It's dynamically typed**

- Variablen können den Type ändern – je nach Inhalt

- **It's functional and object-oriented**

- **It fails silently**

- Bei Fehler wird oft keine Exception geworfen sondern läuft weiter. z.B. $0/0 = \text{NaN}$ (Not a Number)

- **It's deployed as source code**

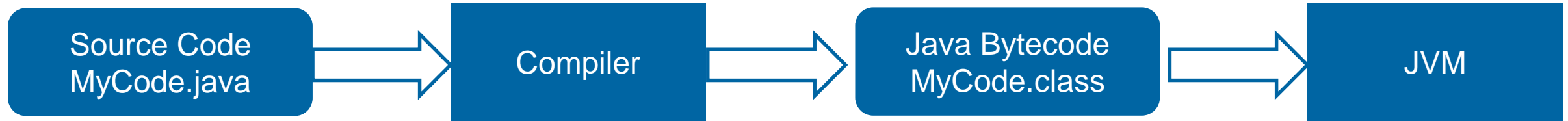
- JavaScript wird erst beim Ziel (z.B. Browser) interpretiert bzw. kompiliert

- **It's part of the web platform**

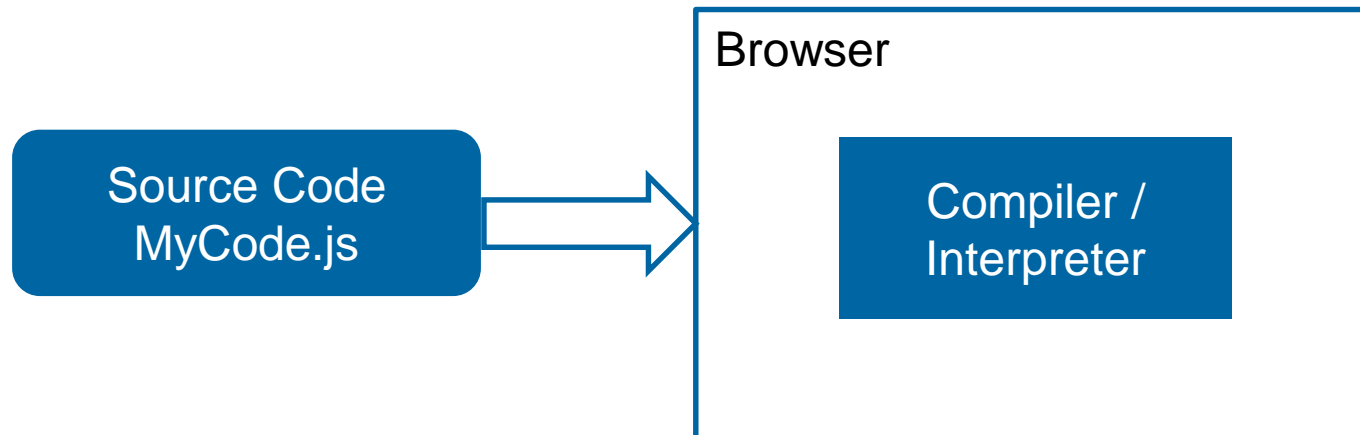
- Auch ohne Browser lauffähig

It's deployed as source code

Java Code



JavaScript Code



JAVASCRIPT-TYPEN

It's dynamically typed

JavaScript is a dynamically typed language.

- Variablen benötigen keine Typendeklaration
- Die gleiche Variable kann über die Zeit unterschiedliche Typen beinhalten
- `typeof()` kann genutzt werden um den Type der Variable abzufragen
- Wichtig: Für Nachvollziehbarkeit sollte einer Variable immer nur Werte vom gleichem Typen zu gewissen werden.

```
var foo;  
foo = "Michael";  
console.log(foo + " is a " + typeof(foo));  
foo = 42;  
console.log(foo + " is a " + typeof(foo));  
foo = true;  
console.log(foo + " is a " + typeof(foo));
```

Output:

```
Michael is a string  
42 is a number  
true is a boolean
```


Wie Java unterscheidet auch JavaScript zwischen Primitives und Objekten

Primitive Typen:

- **string; number; boolean; null; undefined; symbol (ECMAScript 6)**
- **Compared by value**
- **Always immutable**

Objekte:

- **Alles andere: Plain Objects, Arrays, Regular Expressions, Functions**
- **Compared by reference**
- **Mutable by default**

Typeof

typeof(type)	Result
Undefined	'undefined'
Null	'object'
Boolean	'boolean'
Number	'number'
String value	'string'
Function	'function'
Symbol (ECMAScript 6)	'symbol'
All other	'object'

PRIMITIVES

- **true** und **false**
- **Jeder Wert kann in ein boolean gewandelt werden**
 - `!!(null) => false`
 - `Boolean(null) => false`
- **Logische Operatoren**
 - And: `&&`
 - Or: `||`
- **Prefix Operatoren**
 - Not: `!`
- **Comparison operators**
 - `===, !==, ==, !=`
 - `>, >=, <, <=`

Falsy / Truthy

false Werte:

- false
- 0 (zero)
- "" (empty string)
- null
- undefined
- NaN

true Werte:

- Alles andere
 - Z.B. "0" "false" [] {}

```
console.log(Boolean(undefined)); //false
console.log(Boolean(0));          //false
console.log(Boolean(3));          //true
console.log(Boolean({}));        //true
console.log(Boolean([]));        //true
```

- Nach Definition sind alle Zahlen «floats» (Gleitkommazahl)...
- Die Engines versuchen die floats auf integers abzubilden – falls möglich

```
console.log(0.3333333333333333 * 3 == 1);    //true
var x = 1/3;
console.log(x + x + x);                      //1
console.log(0.3 + 0.7);                     //1
console.log(Number.isInteger(x + x + x));    //true
console.log(Number.isInteger(0.3));          //false
console.log(0.1 + 0.2);                     //0.30000000000000004
```

- Funktioniert nicht immer. Grund: Ist ausserhalb vom gültigen Bereich

```
console.log(9999999999999999); //10000000000000000000
Number.isSafeInteger(9999999999999999); //false
```

- Definition: http://de.wikipedia.org/wiki/IEEE_754
- Beschreibung: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Number

■ NaN («not a number»)

- Ist ein Error-Wert z.B. $0/0 \Rightarrow \text{NaN}$
- Hat auch den Type «number»
- $\text{NaN} == \text{NaN}$ ist immer false
 - `isNaN()` zum Überprüfen

■ Infinity

- Unendlich
- Kann auch negativ sein

```
var div0 = 0 / 0;
console.log( div0 );           //NaN
console.log( typeof(div0) );   //number
console.log( parseInt("abc") ); //NaN
console.log( div0 == NaN );    //false
console.log( isNaN(div0) );    //true

console.log( 3 / 0 );           //Infinity
console.log( Math.pow(2,10000) ); //Infinity
console.log( -Math.pow(2,10000) ); //-Infinity
```

■ Jeder Wert kann in eine Zahl verwandelt werden

- `+(true) => 1` bzw. `Number(true) => 1`
- `Number(null) => 0`
- `Number(«abc») => NaN`
- *Ausnahme: Symbol*

■ `parseInt(«string»)` / `parseFloat(«string»)`

- Parst bis zum ersten Fehler

```
console.log( +(true) );           //1
console.log( +(false) );          //0
console.log( +("1ab") );          //NaN
console.log( +("123") );          //123
console.log( +([]) );              //NaN
console.log( parseInt("1.2ab") );  //1
console.log( parseFloat("1.2ab") );//1.2
console.log( parseInt("abc") );    //NaN
```

- Mit "Text" oder 'Text'
- Escape mit «\»
- Typische Properties / Methoden vorhanden
 - length
 - slice()
 - trim()
 - indexOf()

```
'abc'  
"abc"
```

```
'Did she say "Hello"?'  
"Did she say \"Hello\"?"
```

```
'That\'s nice!'  
"That's nice!"
```

```
'Line 1\nLine 2' // newline  
'Backslash : \\'
```

String – Template (ECMAScript 6)

■ ECMAScript 6 bringt 'Template strings'

- Ermöglicht Strings mit Placeholders (und mehr)
- https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/template_strings

■ Es gibt einen neuen String-Typ welcher mit `umschlossen` wird

- Inhalt innerhalb von `\${ ... }` wird interpretiert und durch das Resultat ersetzt
- Linebreaks und Leerzeichen werden beibehalten.

```
var name = "Michael";  
var hobby = "Hike";  
  
console.log("Mein Name ist: "+name +  
            "\nHobby: " + hobby);
```

```
console.log(`Mein Name ist: ${name}  
Hobby: ${hobby}`);  
  
//kann auch rechnen  
var a = 4;  
var b = 5;  
console.log(`${a} + ${b} = ${a + b}`);
```

■ Was geben folgende Ausdrücke aus?

"4" / "2"

"4" - "2"

"4" * "2"

"4" + "2"

10 * 3 + "px"

"px" + 1 - 2

1 / 0

"3px" + 3 * 2 + "3px"

"foo" + "abc"

"2" - -1

- Punkt vor Strich
- Von Links nach Rechts aufgelöst

Spezielle:

- `String + Value = String`
- `Value + String = String`

Ansonsten:

- `Value [Numerische Operator] Value = Number`
 - d.h. + - * / %

The Abstract Equality Comparison Algorithm

```
console.log([] == false);           //true
console.log("" == false);           //true
console.log(null == false);         //false
console.log(0 == "0");               //true
console.log(null == undefined);      //true
console.log([1,2] == "1,2");         //true
console.log(NaN == NaN);             //false
console.log([] == ![]);              //true
```

The Abstract Equality Comparison Algorithm

■ The Abstract Equality Comparison Algorithm (==)

Definition:

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

Beschreibung:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Using_the_Equality_Operators

■ The Abstract Relational Comparison Algorithm (<)

Definition:

<http://www.ecma-international.org/ecma-262/5.1/#sec-11.8.5>

The Abstract Equality Comparison Algorithm

■ ===

- Verhindert die typen Umwandlung von Primitives
- Für Objekte nicht notwendig
- Im Zweifelsfall immer verwenden

```
console.log(false === false); //true
console.log(4 === 4); //true
console.log(false === false); //true
console.log([] === false); //false
console.log("" === false); //false
console.log(null === false); //false
console.log(0 === "0"); //false
console.log(null === undefined); //false
console.log([1,2] === "1,2"); //false
console.log(NaN === NaN); //false
console.log([] === ![]); //false
```

null != undefined

undefined:

- Variable ist nicht definiert. Z.B. `var a` wurde vergessen
- Variable ist nicht initialisiert. Z.B. `a = 1234` wurde vergessen

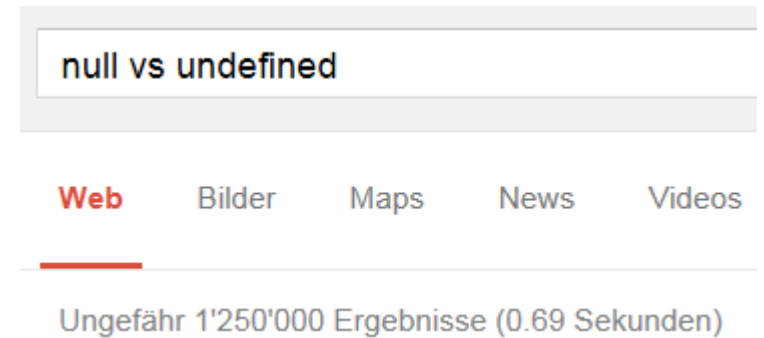
null:

- Ist ein Wert von einer Variable.

=> undefined ist ein Zustand und null ist ein Wert

Achtung: `null == undefined` resultiert in `true`

Hinweis: In alten Browsern ist undefined überschreibbar.



null != undefined

Wie würden Sie überprüfen ob eine Variable undefined ist?
Z.B. myVariable

Wie würden Sie überprüfen ob ein Wert auf einem Objekt undefined ist?
z.B. myVariable.a

null != undefined

Wie würden Sie überprüfen ob eine Variable undefined ist?
Z.B. myVariable

Wie würden Sie überprüfen ob ein Wert auf einem Objekt undefined ist?
z.B. myVariable.a

A:

```
typeof myVariable == 'undefined';
```

B:

```
myVariable.a == undefined;
```

Combined:

```
typeof (myVariable) != 'undefined' && myVariable.a == undefined;
```

Performance: <http://jsperf.com/undefined-null-typeof>

ARRAY

Array

```
var arr = [ 'a', 'b', 'c' ];  
arr[0] = 'x';  
arr.push("d");  
console.log(arr);  
console.log(arr.length);
```

■ Wichtige Methoden:

https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Array#Methods_of_array_instances

Array Methoden & Iterieren

■ Liste aller Varianten:

https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Array#Iteration_methods

```
var arr = [ 'a', 'b', 'c' ];
arr.forEach(function(elem, index) {
    console.log(index + ":" + elem);
});

var array = [1, 2, 3, 4].map(function (x) { return x*x });
console.log(array);

var filteredArray = array.filter(function(elem) {
    return elem > 5;
});
console.log(filteredArray);

console.log(array.every( function(elem) { return elem > 5 }));
```

Array Iterieren for of/in

■ Normale

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

■ For-In iteriert über die Property Namen

```
for(var x in arr)
{
    console.log("for in", x + ":" + arr[x]);
}
```

```
var dummy = {name: "Hallo", date : Date.now()};
```

```
for(var x in dummy)
{
    console.log("dummy has property", x);
}
```

■ For-Of iteriert über die Werte

```
for(var y of arr)
{
    console.log("for of", y);
}
```

OBJECT

Simple Object

- Ein Object ist eine Sammlung von Properties.
- Die Properties werden mit einem Set (HashSet) verwaltet
 - Key = String
 - Value= Value
 - boolean / function / string / ...
- Objekt können als «object literals» erstellt werden und/oder nachträglich mit Properties ergänzt werden:

```
var person = {  
  name : "Michael",  
  hallo : function() {  
    return "Hallo "+this.name;  
  }  
};  
  
console.log(person.hallo());  
  
person.name = "Bob";  
console.log(person.hallo());
```

```
var mySimpleObject = {};  
mySimpleObject.name = "Michael";  
mySimpleObject.hallo = function() {  
  return "Hallo "+this.name;  
};  
  
console.log(mySimpleObject.hallo());
```

Simple Object: It's dynamic

- Properties und Methoden können hinzugefügt / verändert werden:

```
var person = {  
  name : "Michael",  
  hallo : function() {  
    return "Hallo " + this.name;  
  }  
};  
person.hobby = "Hike";  
person.hallo = function() {  
  return "Hallo " + this.name + " Hobby: " + this.hobby;  
};  
console.log(person.hallo());
```

- Auch von «Standard»-Objekten. *Wichtig: Standardfunktionalität sollte nie verändert werden.*

```
console.log("X");  
console.log = function(value) {  
  
};  
console.log("X");
```

FUNCTIONS

JavaScript Funktionen

- Funktionen können in Variablen abgespeichert werden.
- Funktionen können als Parameter übergeben werden.
- Funktionen besitzen eine offene Parameter-Liste
 - Es können mehr oder weniger als die deklarierte Anzahl an Parameter übergeben werden.
 - Alle Parameter werden in `arguments` abgelegt.
- Funktionen besitzen Properties
- Eine Funktion erzeugt einen eigenen Scope.

```
function helloWorld(a){  
    console.log(a || "No Data");  
}
```

```
function helloWorld2(){  
    console.log(arguments[0]);  
}
```

```
var sayHello = function(fnOutput)  
{  
    fnOutput("Hallo")  
}
```

```
sayHello(helloWorld);  
sayHello(helloWorld2);
```

Funktionen definieren

```
//Funktionen können definiert werden  
function hallo(){  
    console.log("Hallo");  
}  
hallo();  
  
//Funktionen können einer Variable zugewiesen werden  
var hallo2 = function() {  
    console.log("Hallo2");  
};  
hallo2();  
  
//Funktionen können einer Variable zugewiesen werden  
var foo = hallo;  
foo();
```


Funktionen können auch als Parameter übergeben werden

```
function add(a, b) {  
    return a + b;  
}  
  
function minus(a, b) {  
    return a - b;  
}  
  
function calc(fn, a, b) {  
    console.log(fn(a, b));  
}  
  
calc(add, 3, 4);  
calc(minus, 3, 4);
```

Funktionen besitzen eine offene Parameter-Liste

```
function foo(name) {  
    console.log(name);  
    console.log(arguments.length);  
    // Ausgabe aller Parameter  
    for(var i = 0; i<arguments.length; i++)  
    {  
        console.log(i, arguments[i]);  
    }  
    // oder als Array  
    var array = Array.prototype.slice.call(arguments);  
    console.log(array.join("\n"));  
}  
  
foo("Michael", "Gfeller", "HSR", "IFS");
```

- arguments beinhaltet alle Parameter, welche der Funktion übergeben wurden
- arguments ist kein Array
 - arguments to array wie folgt:
`Array.prototype.slice.call(arguments);`

Rest Parameters

```
function foo(name, ...params){  
    console.log(1, name);  
    console.log(2, params.join(";"));  
}  
  
foo("Michael", "Gfeller", "HSR", "IFS");
```

- Mit «...» kann man den letzten Parameter als «Rest-Parameter» definieren
- Dieser Parameter wird mit allen restlichen Werten abgefüllt
- Keine restlichen Parameter erzeugt ein leeres Array

Funktionen besitzen Properties

```
var fn = function() { return "Michael"};
console.log(fn.name);           //""
fn = function name() { return "Michael"};
console.log(fn.name);           //"name"
console.log(fn.length);         //0

fn = function name(name) { return name};
console.log(fn.length);         //1
```

- `.name` beinhaltet der Name der Funktion
 - Anonyme Methoden besitzen kein Name
 - Dieser Name wird für den Stacktrace genutzt
 - Moderne Browser loggen, falls kein Name angegeben wurde, den Variablen Name
- `.length` beinhaltet die Anzahl Parameter der Funktion

Funktionen toString()

- toString() gibt den Source Code von der Funktion zurück.
- Muss explizit aufgerufen werden:

```
function sayHallo(name) {  
    return "Hallo " + name;  
}  
console.log(1, sayHallo);  
console.log(2, sayHallo.toString());
```

```
1 [Function: sayHallo]  
2 'function sayHallo(name){\r\n    return "Hallo " + name;\r\n}'
```

SCOPE

Scope

- Unter «Scope» versteht man die Sichtbarkeit von Variablen und Funktionen
- Jede Funktion generiert einen neuen Scope
- Innerhalb von einem Scope kann man auf
 - dessen Variablen
 - globale Variablen
 - Variablen aller «Parent»-Scopes zugreifen.
 - Dieses Feature nennt sich Closure
 - Diese Werte bleiben erhalten
- Funktionen verhalten sich gleich.
- Wichtig: Ein Script Tag erzeugt keinen Scope

```
var funcA = function() {  
  var a = 1;  
  var funcB = function() {  
    var b = 2;  
    console.log(a, b); // outputs: 1, 2  
  };  
  funcB();  
  console.log(a, b); // Error! b is not defined  
};  
funcA();  
funcB(); // Error! funcB is not defined
```

Scope

Variable im globalen Scope

- Ohne «var»
- Im Browser über `window.myGlobalVariable` zugreifbar. In node.js: `global.myGlobalVariable`

Variablen im lokalen / Funktionen Scope

- Benötigt ein «var»
- Nur innerhalb des Scopes zugreifbar oder dessen «nested» Scopes
 - Closures

«Module Scope»: (node.js)

- Node.js kennt noch ein «Module Scope». Nicht **explizit** globale Variablen werden auf den «module Scope» gelegt. Dieser ist File basiert.

«Block Scopes»

- Mit [ECMA6](#) möglich

Variablen Scope

```
a = "A"; //wird auf das globale Objekt gelegt
var b = "B"; // wird auf den aktuellen Scope gelegt

function foo(){
    c = "C"; //wird auf das globale Objekt gelegt
    var d = "D"; // wird auf den aktuellen Scope gelegt
}
foo();

var globalObject = typeof(global) === "undefined" ? window : global; // browser oder node js
console.log(globalObject.a);
console.log(globalObject.b);
console.log(globalObject.c);
console.log(globalObject.d);
```

Node

A
undefined
C
undefined

Browser

A
B
C
undefined

Block Scope (ECMAScript 6)

- 'use strict' Modus notwendig
- «let» als neues Keyword

```
'use strict'
for(let x = 1; x < 10; ++x){
  console.log(x);
  if(x > 4){
    let x = 10; //wird vom for-loop ignoriert
    // var x = 10 // Identifier 'x' has already been declared
  }
}
//console.log(x); //ReferenceError: x is not defined
```

Babeljs:

```
'use strict';
for (var _x = 1; _x < 10; ++_x) {
  console.log(_x);
  if (_x > 4) {
    var _x2 = 10; //wird vom for-loop ignoriert
  }
}
console.log(x); //ReferenceError: x is not defined
```

CONTEXT

«this» Context

«this» ist der aktuelle Context.

«this» referenziert je nach Aufrufart ein anders Objekt:

- Falls ein eine Funktion als Methode von einem Objekt aufgerufen wird. Ist this = objekt
Beispiel: **object.foo()**;
- Falls eine Funktion mit new() aufgerufen wird. Wird «this» mit einem neu erstellten Objekt abgefüllt.
Beispiel. **new foo()**;
- Falls eine «unbound» Funktion aufgerufen wird. Zeigt «this» auf das globale Objekt.

Jeder Funktion kann mit apply() oder call() den Context gesetzt werden. In diesem fall werden die oben genannten Regeln ignoriert.

Z.B. **foo.call({ counter : 123});**

Jeder Funktion kann mit bind einen Context vorgeben werden. In diesem fall werden die oben genannten Regeln ignoriert.
Diese Regel wird ignoriert falls die «gebundene Funktion» mit new aufgerufen wird.

var boundFoo = foo.bind({counter : 11});
boundFoo();

USE STRICT

Strict Mode / 'use strict'

'use strict' hat folgende Ziele

- **Eliminiert «fails silently»**

- Falls eine Variable ohne «var» definiert wird
- Falls Read-Only Werte gesetzt werden
 - Ohne Strict Mode: Wird einfach ignoriert
- ...

- **Eliminiert «Probleme» welchen es Compiler verunmöglicht den Code zu optimieren**

- Viele ECMAScript Features funktionieren nur im Strict Mode

- **Security wird «leicht» verbessert**

- Z.B. wird bei «unbound» Funktion «this» nicht auf das globale Objekt gelegt.

Wichtig: das Laufzeitverhalten ändert sich!

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

Strict Mode aktivieren

- 'use strict' im JavaScript aktiviert den Strict Mode
 - Erste Linie einem File
 - Ganzes File wird in Strict Mode geschaltet;
 - Erste Linie in einer Function
 - Funktion und nested Funktionen werden in Strict Mode geschaltet;
- Strict Mode wird vererbt
 - Alle «nested» Scopes werden auch Strict
 - Funktionen werden nicht in Strict Mode geschaltet obwohl Sie von einer Strict Funktion aufgerufen werden
- Wichtig
 - Strict Mode kann nicht mehr entfernt werden
 - Strict Mode muss pro File aktiviert werden
 - «Strict Mode» JavaScript sollte nicht mit «Non Strict Mode» JavaScript zu einem JavaScript File vereint werden (concatenate)

```
function a() {  
    let a = 1;  
}  
  
function c() {  
    'use strict';  
  
    function b() {  
        let b = 1;  
    }  
    b();  
    a();  
}  
c();
```

ARROW FUNCTION / ARROW

ECMAScript 6 Funktionen

- **ECMAScript 6 erlaubt es Funktionen mit einem «Arrow-Syntax» zu definieren**
 - Oft «Lambda» bzw «Lamba-Syntax» genannt (C# / Scala)
- **Sinnvoll für (sehr) kleine Funktionen z.B. als Filter Parameter**

```
[ 'a', 'b', 'c' ].forEach((elem, index) => console.log(index + ":" + elem));  
  
var array = [1,2,3,4].map(x => x*x);  
  
var filteredArray = array.filter( elem => elem > 5);  
  
console.log(array.every( elem => elem > 5));  
  
console.log(() => {  
    var x = 9;  
    var y = 11;  
    return x + y;  
})());
```

- Bei Lambda-Funktionen ist der Context immer auf das selbe Objekt gebunden.
- Praktisch für Constructor Funktion

```
function Point(x, y) {  
  this.x = x;  
  this.y = y;  
  this.area = () => this.x + this.y;  
}  
  
var areaFn = new Point(10, 50).area;  
console.log(areaFn());
```

babeljs

```
function Point(x, y) {  
  var _this = this;  
  
  this.x = x;  
  this.y = y;  
  this.area = function () {  
    return _this.x + _this.y;  
  };  
}
```

CONSTRUCTOR FUNCTION

Konstruktor Funktion

- JavaScript unterstützt Klassen und Inheritance
 - Teil von WED2
- Falls eine JavaScript Funktion mit «new» aufgerufen wird, wird diese zu einem Konstruktor.
 - new erzeugt ein leeres Objekt und setzt dieses als Context.
- Starten mit einem gross Buchstaben
- Fragen:
 - Weshalb ist y private?
 - Was passiert wenn man Rectangle ohne new aufruft
 - Im Strict Mode?

```
function Rectangle(width, height){  
    this.x = width; // public  
    var y = height; // private  
    this.area = () => this.x * y;  
    this.toString = () => { return `X: ${this.x} Y: ${y}` }  
}  
  
var rect = new Rectangle(5, 5);  
  
console.log(rect.area());  
rect.x = 100;  
rect.y = 100;  
console.log(rect.area());  
  
console.log(""+rect);
```

JAVASCRIPT FEATURES

Variante 1:

```
"use strict";  
var a  
a = 1;  
console.log("0", a);
```

Variante 2:

```
"use strict";  
a = 1;  
console.log("0", a);  
var a;
```

- Dieses Feature nennt sich «Hoisting»
- JavaScript verschiebt alle Deklarationen von Methoden / Variablen an den Anfang des Scopes.
- Initialisierungen werden nicht ge-«hoisted»
- Kann zu Bugs führen.
- Funktionen-Definitionen werden auch verschoben.
- «let» Variablen sind vom Hoisting nicht betroffen
- Achtung: Bei einer «Function-Expressions» wird die Zuweisung nicht verschoben.

Hoisting

	Hoisting	Scope	Creates global properties
var	Declaration	Function	Yes
let	Temporal dead zone	Block	No
const	Temporal dead zone	Block	No
function	Complete	Block	Yes
class	No	Block	No
import	Complete	Module-global	No

Quelle: http://exploringjs.com/es6/ch_variables.html#_ways-of-declaring-variables

Jedes Objekt ist eine HashTable

■ Jedes Objekt ist eine HashTable

- Ausnahme: null, undefined

■ Zugriff auf Eigenschaften vom Objekt mit

- `obj[PropertyName]` bzw. `obj[PropertyName] = "A "`

■ Objekte

- Wandeln den Index Wert immer in einen String.
 - `Sample[1] == Sample["1"]`

■ Array

- Unterstützt ganzzahlige Nummern als Indexer
- Nicht ganzzahlige Nummern werden zu Strings gewandelt.
 - Das Array verhält sich wie ein normales Objekt
- `[].length` beachtet nur die echten Array Einträge.
- => Ein Array sollte wie ein Array verwendet werden.

■ Functions

- Können wie Objekte mit Properties ergänzt werden

Syntax

```
while (condition) {  
  statement  
}
```

condition

An **expression** evaluated before each pass through the loop. If this condition evaluates to true, statement is executed. When condition evaluates to false, execution continues with the statement after the while loop.

Statement oder Expression

■ Expression

- Erzeugt einen Wert
- Kann als Parameter einer Funktion übergeben werden
- Beispiele:
 - `myfunc("a", "b")`
 - `3 + x`
 - `myVar`

```
var x = (y >= 0 ? y : -y);
```

■ Statement

- Führt etwas aus (Sprachelemente)
- Beispiele:
 - `If`
 - `Loops`
 - `Var`

```
var x;  
  
if (y >= 0)  
{  
    x = y;  
}  
else {  
    x = -y;  
}
```

Statement oder Expression

```
function testExpression( a ) {  
    console.log(a);  
}  
var i = 10  
testExpression("A");  
testExpression(a = i > 0 ? i : -i );  
testExpression(var a = i > 0 ? i : -i );  
testExpression((var a = i > 0 ? i : -i) );
```

<https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/var>

Semicolon insertion

Details: <http://people.mozilla.org/~jorendorff/es6-draft.html#sec-rules-of-automatic-semicolon-insertion>

- A semicolon is inserted before, when a Line terminator or "}" is encountered that is not allowed by the grammar.
 - Ein Semikolon wird eingefügt, falls die zusammengefügte Zeile zu einem Fehler führen würde.

```
var x = 10
var y = 20;
console.log(x + "," + y);
```

```
var a = 2 * 4
(a).toString();
```

```
var x = 10 var y = 20;    // Fehler
console.log(x + "," + y);
```

```
var a = 2 * 4(a).toString();
```

Semicolon insertion

- A semicolon is inserted at the end, when the end of the input stream of tokens is detected and the the parser is unable to parse the single input stream as a complete program.
 - Semicolon am Ende eines Programmes
- A semicolon is inserted at the end, when a statement with restricted productions in the grammar is followed by a line terminator.
 - Statement wird mit dem nächsten “line terminator” beendet.
 - PostfixExpressions (++) and --); continue; break; return;

```
function createUser(name)
{
    return
    {
        name : name
    }
}
```

```
function createUser(name)
{
    return;
    {
        name : name
    }
}
```

Semicolon insertion

Aufgabe:
Welcher Wert wird «foo» beinhalten?

```
var foo
var bar

var baz = function(data) {
  return data +
    1
}
bar = 1

foo = bar + baz
(bar + bar) + baz(bar)
```

```
var foo;
var bar;

var baz = function(data)
{
  return data +
    1;
};

bar = 1;
foo = bar + baz(bar + bar) + baz(bar);
```

Fazit: Semicolon insertion

- Falls Semikolons weggelassen werden, fügt der JavaScript-Interpreter diese ein.
- Automatisch gesetzte Semikolons können zu Bugs führen.
- Programmierer sollten sich NIE auf automatisch gesetzte Semikolons verlassen.
- Aber: Für bessere Übersicht sinnvoll:

```
var result = [4,2,1,5].map(function(x) { return x - 3 }).filter(function(x) { return x > 0 }).sort();  
console.log(result);
```

```
var result = [4,2,1,5]  
    .map(function(x) { return x - 3 })  
    .filter(function(x) { return x > 0 })  
    .sort();
```

UTILITIES

Ein regulärer Ausdruck (englisch regular expression, Abkürzung RegExp oder Regex) ist in der theoretischen Informatik eine Zeichenkette, die der Beschreibung von Mengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient.

Quelle: Wikipedia

Ausführlich: <http://www.ecma-international.org/ecma-262/5.1/#sec-15.10>

- Teil der Sprachdefinition
- Pattern-Matching
- Search & Replace

Syntax: */ muster / flags bzw. RegExp(muster, flags)*

Muster: Ausdruck, welcher gesucht werden sollte. Kann spezielle Zeichen beinhalten

Flags: g, i, m

Spezielle Zeichen: . \d \D \s ...

Weitere Informationen: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/RegExp

Regex - Beispiele

Ersetze alle 'JS' durch 'JavaScript'

```
"Das ist ein Text über JS".replace(/JS/i, "JavaScript")
```

Erkennen ob der RegEx matched

```
/JS/i.test("Das ist ein Text über JS")
```

```
"Das ist ein Text über JS".match(/JS/)
```

Compile von Regex-Ausdrücken

```
var compiled = /JS/;  
compiled.compile();
```

<http://jsperf.com/regexcompile>

Date Objekt

- Datums-Funktionalitäten
- `Date()` > erzeugt ein Datums-String
- `new Date()` => erzeugt ein Datums-Objekt
- `new Date([YEAR], [MONTH], [DAY],...)` => erzeugt ein Datums-Objekt mit den Daten
 - `new Date(2015, 20, 10)` => 10 Sept. 2016

Date to String

- Manuell mit `getMonth()` `getDay()` `getFullYear()`
- `new Date().toLocaleDateString()`
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/toLocaleDateString
- Package: <http://momentjs.com/>

Math Objekt

- **Mathe-Funktionalitäten**
- **Mathe-Konstanten wie PI**
- **Winkel-Funktionen in Radian**
- **Implementation ist Browser-spezifisch. D.h. unterschiedliche Genauigkeit je nach Browser.**

ES6

■ <http://es6-features.org/>

- Enhanced Object Properties
 - Default Parameter Values
 - Rest Parameter
 - Nicht mit REST-API zu verwechseln!
 - Spread Operator
- Template Literals
- Destructuring Assignment
- Classes
- Symbol Type
- Iterator & For-Of Operator
- Internationalization & Localization

ZUM NACHLESEN

Zum Nachlesen

- <http://exploringjs.com/es6/>
- <http://speakingjs.com/es5/index.html>