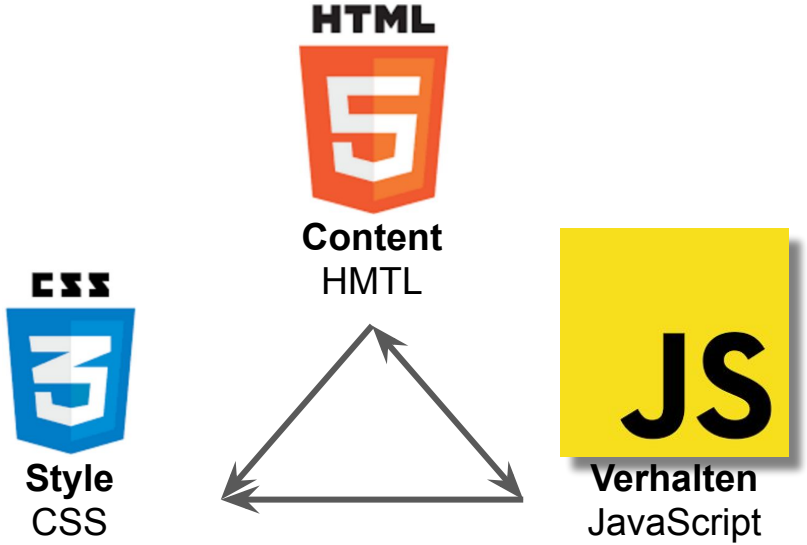
JavaScript

eine moderne Einführung für Informatiker und Java-Programmierer

Manfred Kaul



Eingangsfrage:

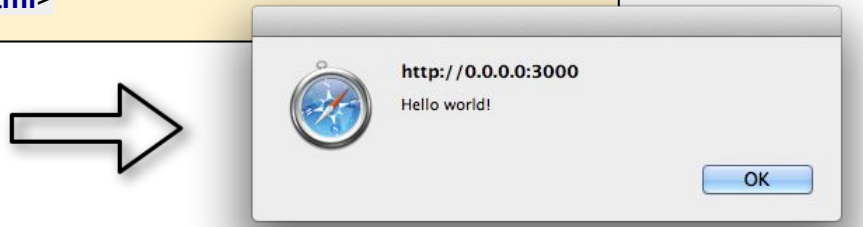
Wie baut man eine Skriptsprache für das World-Wide Web?

JavaScript-Geschichte

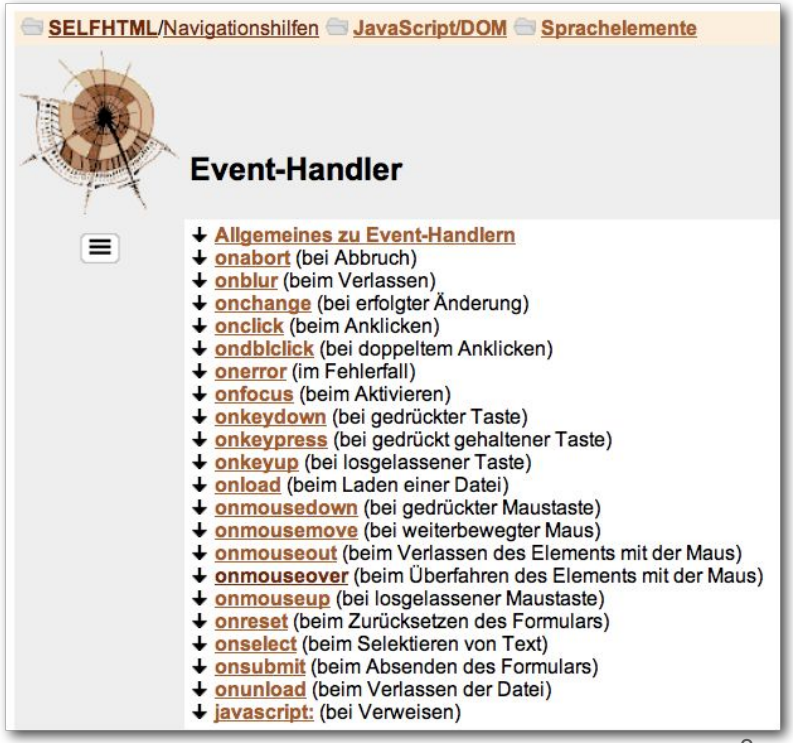
1995

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

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

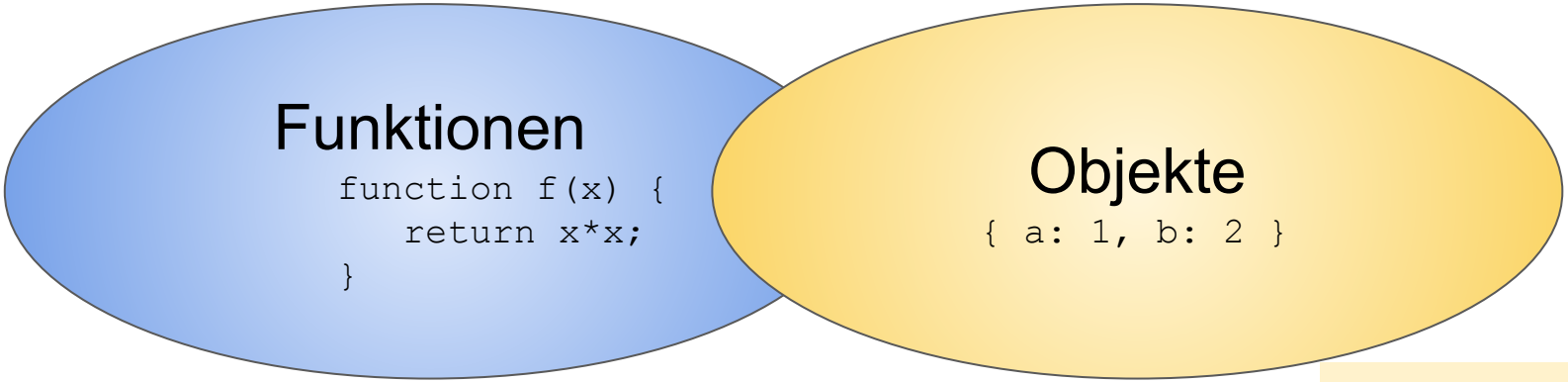


- Skriptsprache für HTML
- gedacht für kleine Event-Handler
  "onclick", "onchange", ...
- kein Java!
- Vorgänger LiveScript von Brendan Eich erfunden bei der Fa. Netscape im Jahre 1995:
  Erster Prototyp in 10 Tagen !!!



de.selfhtml.org/javascript/sprache/eventhandler.htm

… durch Minimalismus: nur Funktionen & Objekte

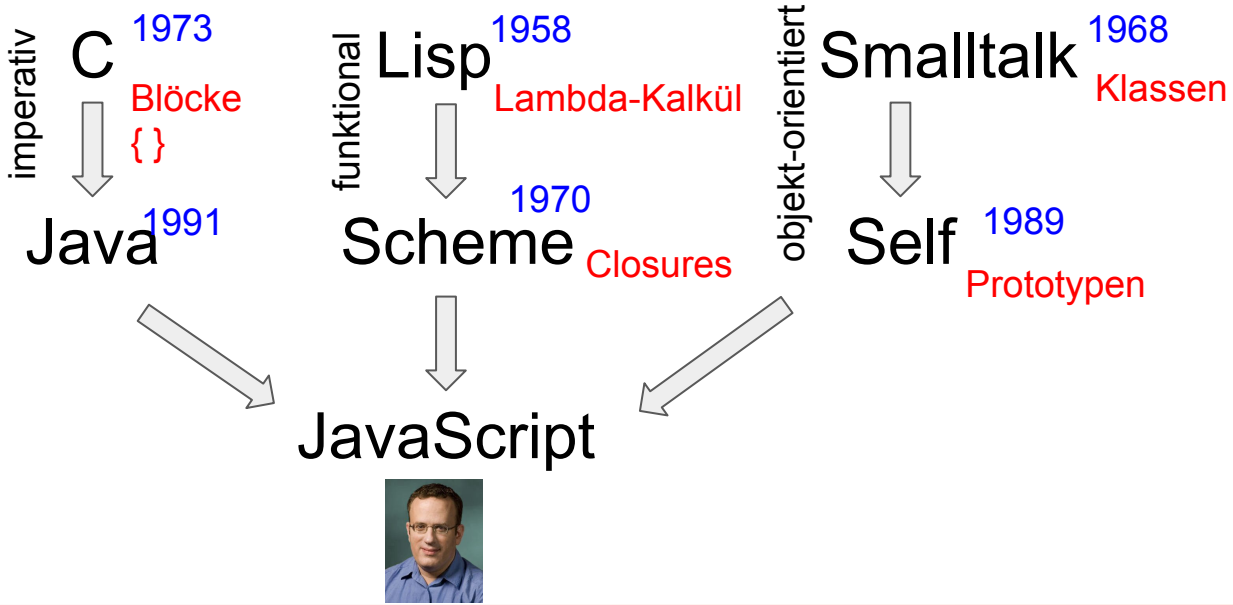


Minimalismus: Was kann man weglassen?

Klasse	⇒	Objekt
Methode	⇒	Funktion
Konstruktor	⇒	Funktion
Array	⇒	Objekt
Objekt-Kapsel	⇒	Function Scope

Prinzipien:
A. Minimalismus
B. Universalität
C. Rekursivität

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

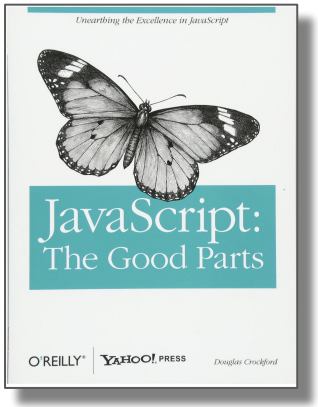
- **dynamisch** ⇒ zur Laufzeit Abfrage mit **typeof x**
- **schwach** ⇒ nur grobe Einteilung (z.B. kein Typ "Array"), die sich zur Laufzeit ändern kann, z.B. var x = 1; x = "string"; x = [1, 2];

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

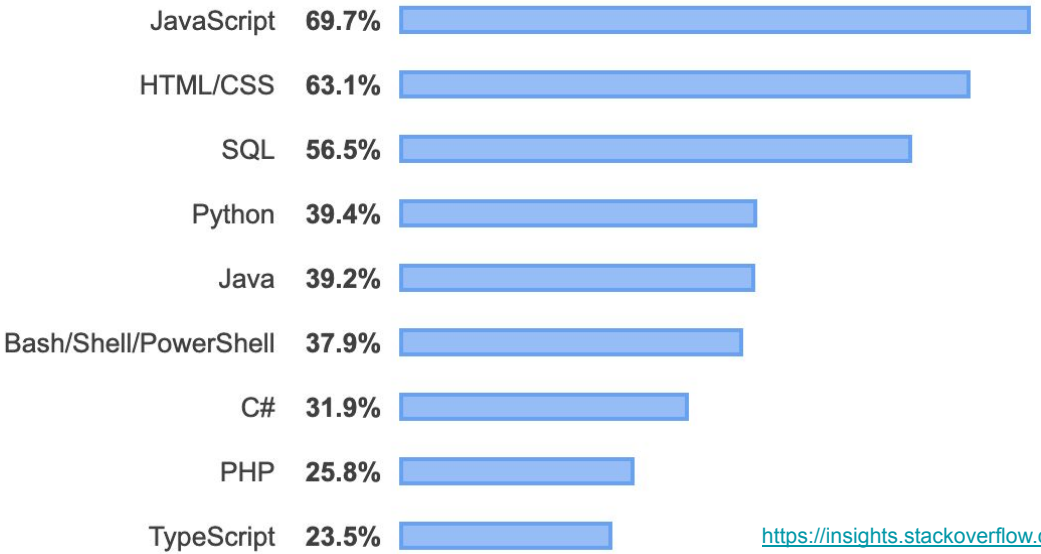
JavaScript ist die populärste Technologie

stackoverflow
Developer Survey Results
2019

90,000 developers

Programming, Scripting, and Markup Languages

All Respondents | Professional Developers



https://insights.stackoverflow.com/survey/2019

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.

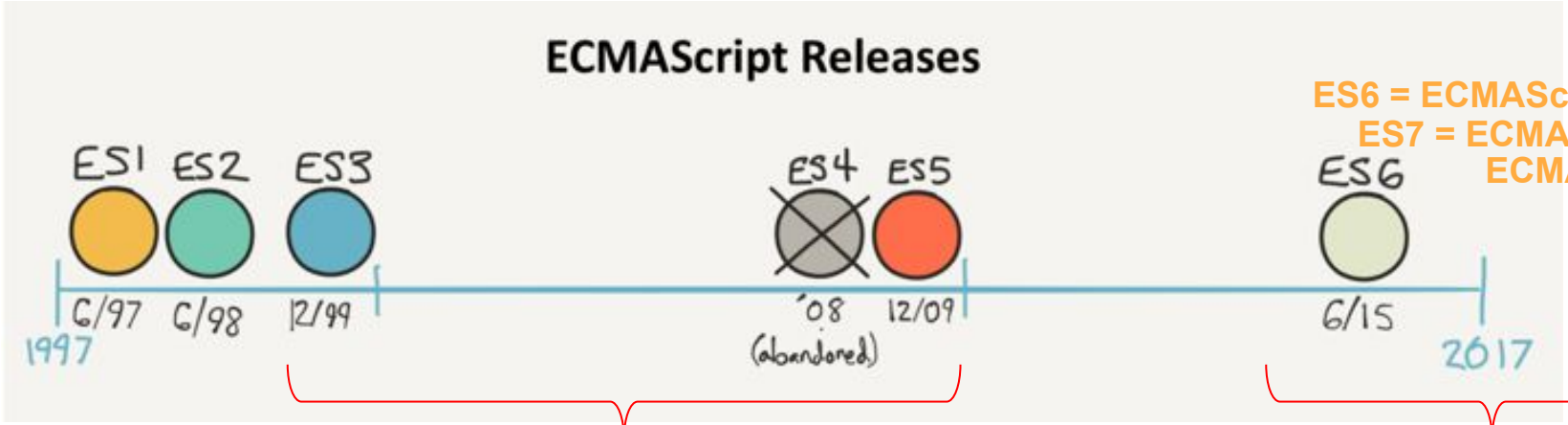
- **Alan Perlis: "*A language that doesn't affect the way you think about programming is not worth knowing*"**

aus "Epigrams on Programming"

Historie: JavaScript hat viele Namen:

~~LiveScript~~ = JavaScript = JS = ECMAScript2015,16,... = ES5,6,7,... = ~~JScript~~

es ist im Kern immer die gleiche Sprache gemeint



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

https://kangax.github.io/compat-table/es6/

Kompabilitätstabelle für ES6

Wie stellt man Browser-Kompatibilität her?

Shims Versus 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.

http://speakingjs.com/es5/ch30.html#id1267739

Die Programmiersprache JavaScript

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

1. Typen und der `typeof`-Operator

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

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

Der **Operator** `typeof` liefert einen `String`

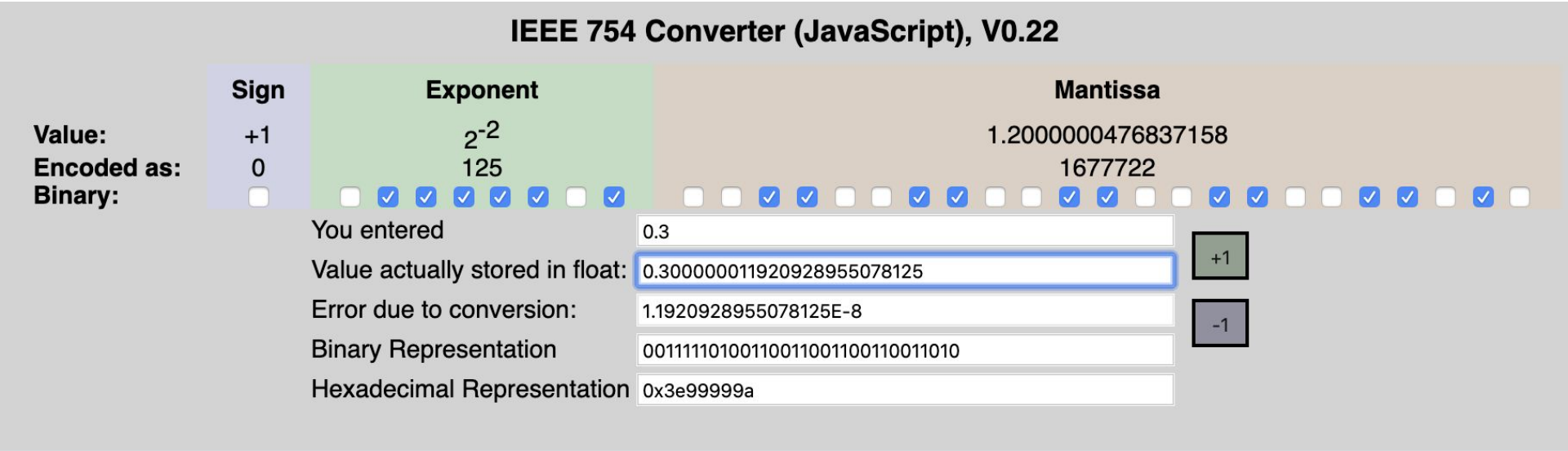
Typ	Rückgabewert
Undefined	`"undefined"`
Null	`"object"` (see below)
Boole'scher Wert	`"boolean"`
Zahl	`"number"`
Zeichenkette	`"string"`
Symbol (neu in ECMAScript 2015)	`"symbol"`
Host-Objekt (von der JS-Umgebung bereitgestellt)	*implementierungsabhängig*
Funktionsobjekt (implementiert [[Call]] nach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

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

1. Typen: 1.1. Der JavaScript-Typ Number

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

in Java "**double**"



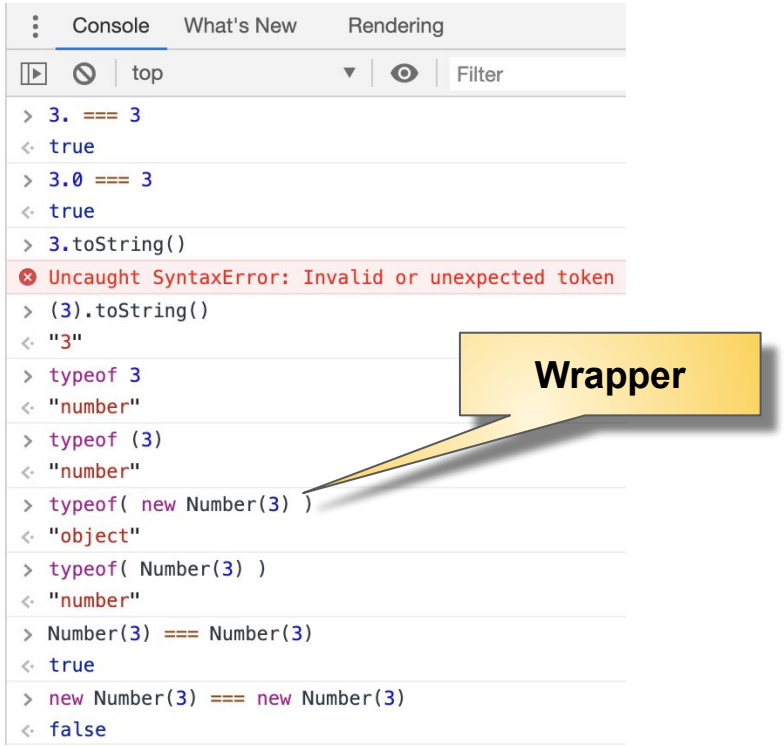
https://www.h-schmidt.net/FloatConverter/IEEE754.html

1. Typen: 1.1. Der JavaScript-Typ Number

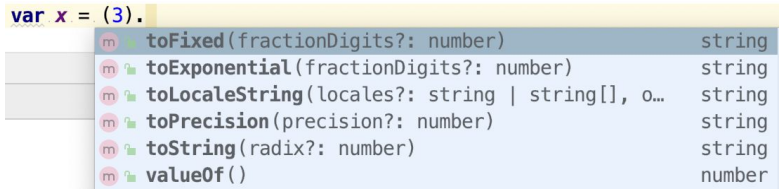
- doppeltpräzise 64-bit Werte im IEEE 754 Format:
  $\pm 2^{\text{Exponent}} * \text{Mantisse}$

```
// Falsche Modellierung von Euro und Cent
var einnahmen = 1.20, ausgaben = 1.10;
console.log( 'Gewinn = ', einnahmen - ausgaben ); // 0.09999999999999987
console.log( einnahmen - ausgaben == 0.10 ); // false

// Begrenzte Genauigkeit
console.log( 9007199254740992 + 1 ); // 9007199254740992

// Das Math-Objekt
var radius = 3;
var umfang = 2 * Math.PI * radius;
console.log( umfang );

// parseInt und parseFloat
console.log( parseInt("14") ); // 14
console.log( parseInt("abc") );  // NaN = Not a Number
console.log( parseFloat("1.23") ); // 1.23

// Infinity
console.log( 1 / 0 ); // Infinity
console.log( Infinity ); // Infinity
console.log( -1 / 0 ); // -Infinity
```

Number




```
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
```



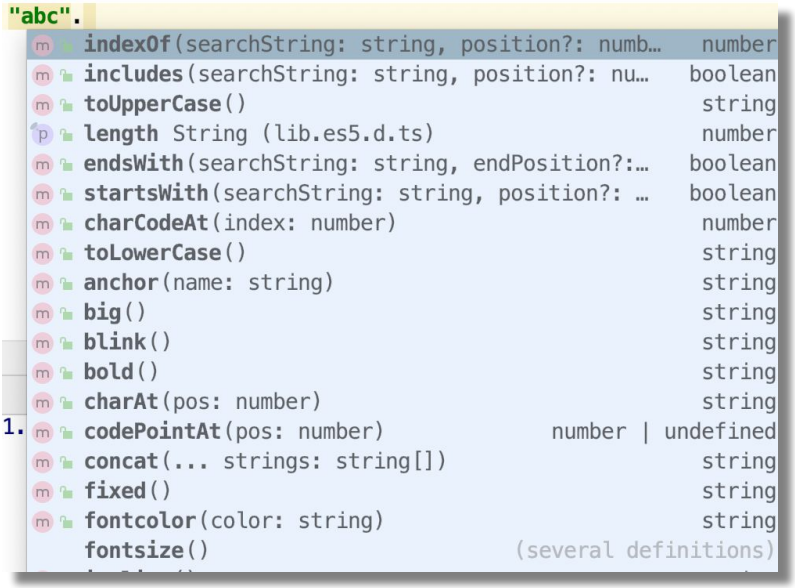
https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Number

1. Typen: 1.2. Der JavaScript-Typ String

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

"Template String"

```
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
```



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" );

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

not-Operator

1. Typen: 1.3. Gleichheitsoperatoren == und ===

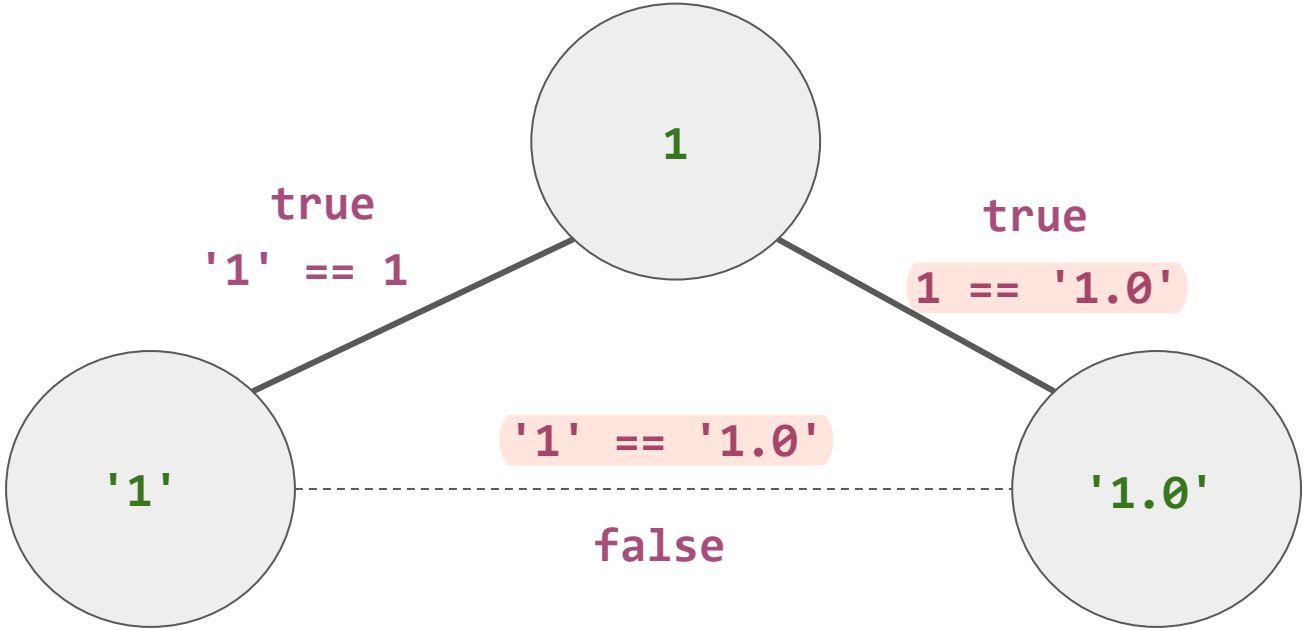
==	===
abstrakte Gleichheit	strikte Gleichheit
intransitiv	transitiv
vor dem Vergleich der Werte wird eine Typangleichung (type coercion) durchgeführt	gleiche Typen und gleiche Werte

`// true`
`console.log( 0 == '0' );`
`console.log( false == '0' );`
`console.log( null == undefined );`
`console.log( ' \t\r\n ' == 0 );`
`console.log( 0 == '' );`
	`// false`
	`console.log( 0 === '0' );`
	`console.log( false === '0' );`
	`console.log( null === undefined );`
	`console.log( ' \t\r\n ' === 0 );`
	`console.log( 0 === '' );`

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

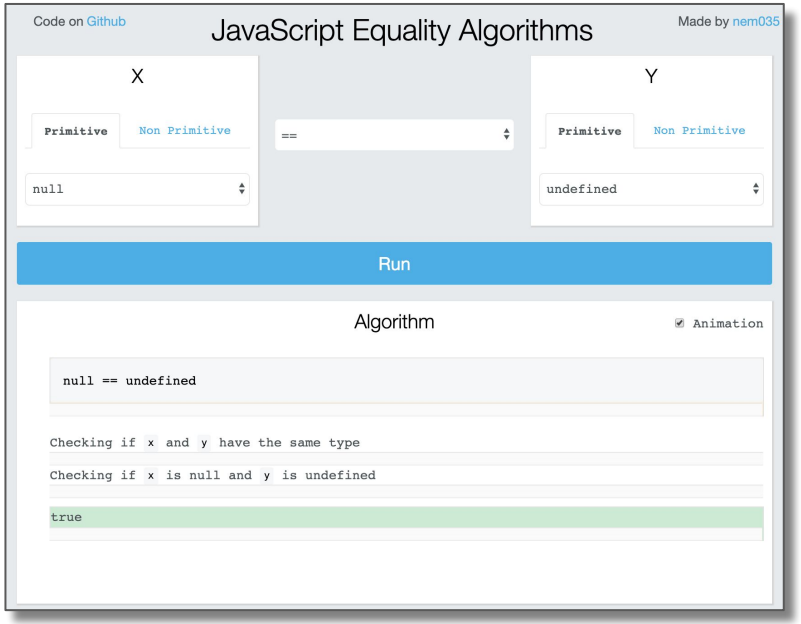
1. Typen: 1.4. Abstrakter Gleichheitsoperator ==

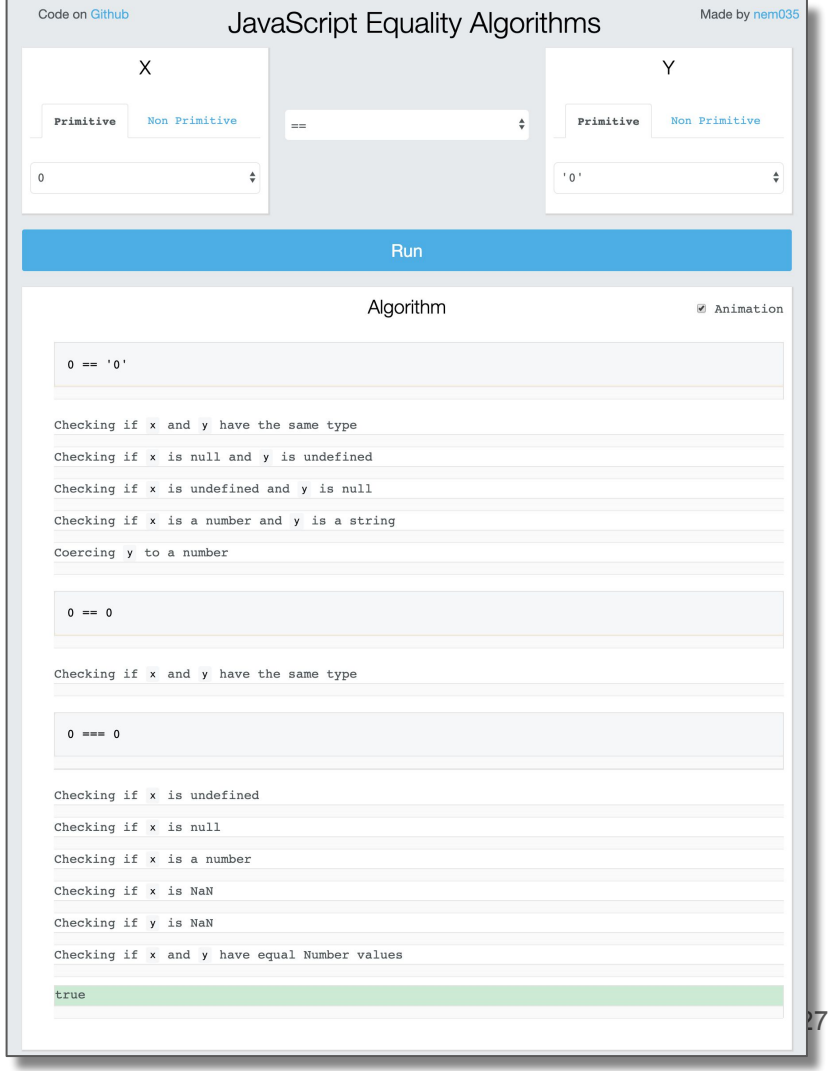
The comparison x == y, where x and y are values, produces **true** or **false**. Such a comparison is performed as follows:

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**.

https://tc39.es/ecma262/#sec-abstract-equality-comparison

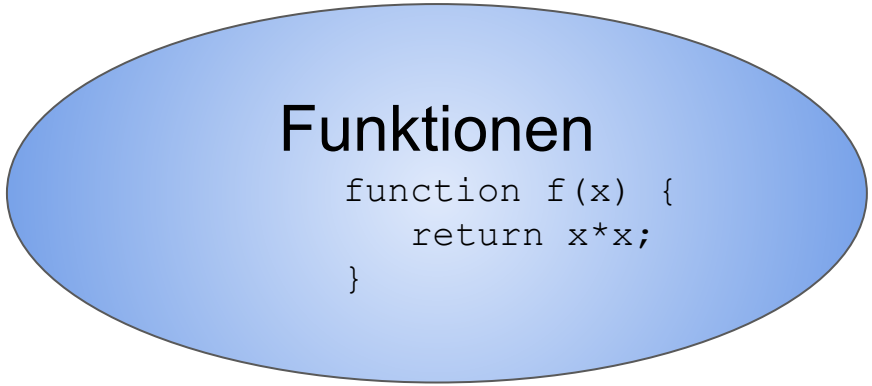
Animation zum abstrakten Gleichheitsoperator





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

2. Funktionen



Funktionen gehören zu den erfolgreichsten Abstraktionen der Mathematik

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

Wie würde eine einfachste
funktionale Programmiersprache aussehen?

2.1 Functions are first-class objects

```
function double1( x ){
   return 2 * x;
}

var double2 = function( x ){
   return 2 * x;
};

var double3 = x => 2 * x;

console.log( double1( 1 ) );
console.log( double2( 2 ) );
console.log( double3( 3 ) );
```

```
function inner_double( x ){
   function double( y ){
      return 2 * y;
   }
   return double( x );
}

function sum( ...restArgs ){
   sum = 0;
   restArgs.forEach( x => sum += x );
   return sum;
}

console.log( sum(1,2,3) );
```

innere Funktion

2.2 Funktionen als Parameter und als Rückgabewert

```
function funfun( fun, x ){
  return fun( x )
}

function double( x ){
  return 2 * x;
}

funfun( double , 3 ); // 6
```

Fabrikfunktionen: Funktionen, die Funktionen berechnen

```
function make_double(){
 return function( x ){
   return 2 * x;
 }
}

make_double()( 3 ); // 6

function make_multiplier( y ){
 return function( x ){
   return y * x;
 }
}

make_multiplier( 3 )( 4 ); // 12
```

2.3 Funktionsdeklarationen haben 2 Aufgaben:

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

```
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.

Zusammenfassung: Functions are First-Class

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; } }
  ```

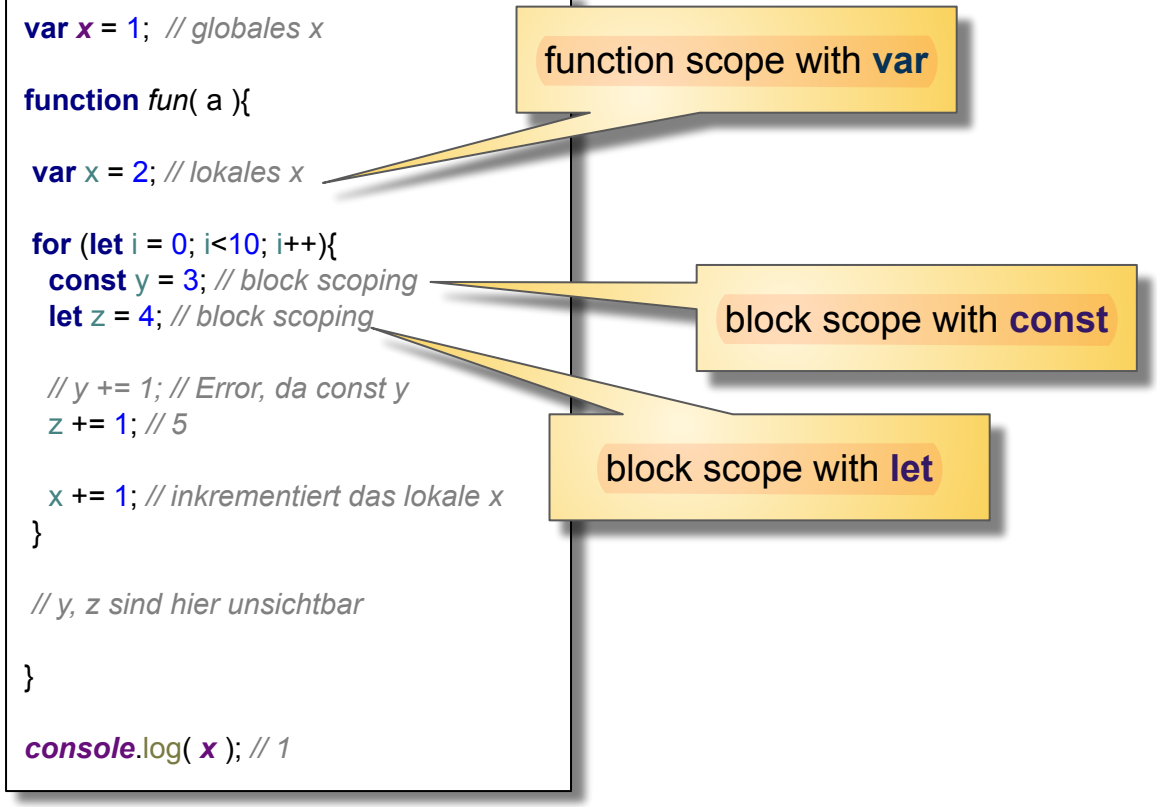
Übungen mit Funktionen in JavaScript -1-

```
// Identität in JavaScript

console.log( identity1( 1 ) );
console.log( identity2( 2 ) );

// Addition in JavaScript

console.log( add( 1, 2 ) );

// Multiplikation in JavaScript

console.log( mul( 2, 2 ) );

// Write a function that takes an argument and
// returns a function that returns that argument.

const idf = identityf( 5 ); // soll die Funktion idf erzeugen
console.log( idf() );  // soll 5 liefern

// Write a function that adds from two invocations.

console.log( addf(3)(3) ); // soll 6 liefern
```

```
// Write a function that takes a binary function,
// and makes it callable with two invocations.
addf = applyf(add);
console.log( addf(3)(4) );          // 7
console.log( applyf(mul)(2)(4) );   // 8

// Write a function that takes a function and an argument,
// and returns a function that can supply a second argument.
add3 = curry( add, 3 );
console.log( add3( 6 ) );           // 9
console.log( curry( mul, 5 )( 2 ) );  // 10
console.log( curry2( add, 7 )( 4 ) );

// Without writing any new functions,
// show three ways to create the inc function.

// Lösung 1
// Lösung 2
// Lösung 3

console.log( inc(11) );             // 12
console.log( inc(inc(11)) );        // 13
```

Variablen und Sichtbarkeit (Scoping)

```
var x = 1;  // globales x

function fun( a ){

 var x = 2; // lokales x

 for (let i = 0; i<10; i++){
   const y = 3; // block scoping
   let z = 4; // block scoping

   // y += 1; // Error, da const y
   z += 1; // 5

   x += 1; // inkrementiert das lokale x
 }

 // y, z sind hier unsichtbar

}

console.log( x ); // 1
```



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!

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

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.

http://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html

Übung zu Hoisting

Welche Ausgabe auf der Konsole?

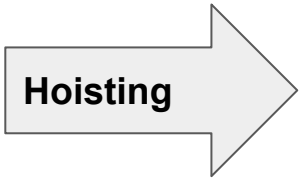
```
var foo = 1;

function bar() {

 var foo = foo ? 1 : 2;

 console.log( foo );

}

bar();
```

http://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html

Lösung zu Hoisting

```
var foo = 1;

function bar() {

 var foo = foo ? 1 : 2;

 console.log( foo );

}

bar();
```

Konsole: 2




```
var foo = 1;

function bar() {

 var foo = undefined;

 foo = foo ? 1 : 2;

 console.log( foo );

}

bar();
```

falsy

Konsole: 2

http://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html

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

4. Objekte

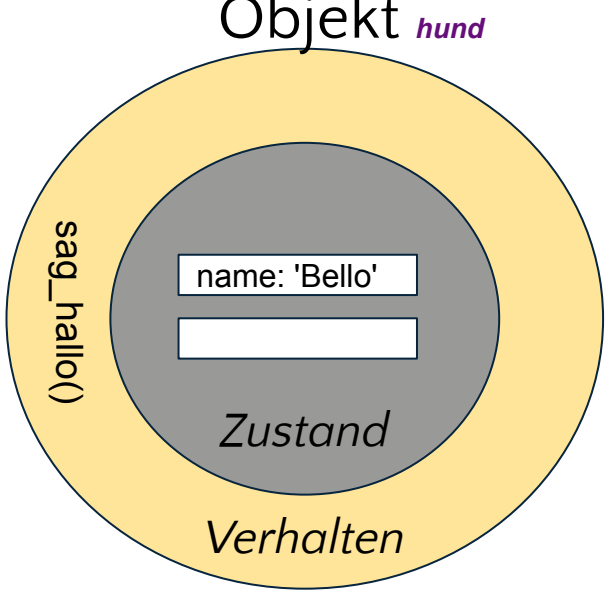
Objekte gehören zu den erfolgreichsten Abstraktionen in Programmiersprachen.

4.1. 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



4.2. Objekt-Literale versus JSON

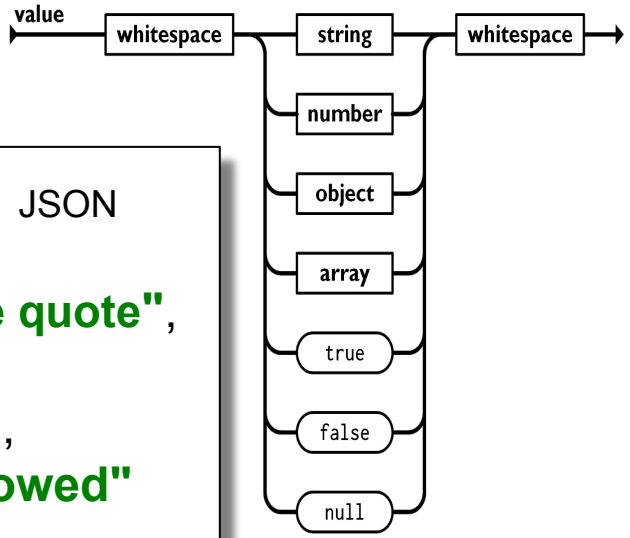
JavaScript

```
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
};
```

Datei-Format

JSON

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



https://www.json.org/

4.3. Fabrikfunktionen zur Erzeugung von Objekten

Statt Konstruktor eine Funktion, die ein Objekt berechnet

Methoden sind Funktionen des berechneten Objektes

Konstruktion eines neuen Hunde-Objekts

erzeugtes Hunde-Objekt

```
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() );
```

name kann von außen überschrieben werden.

Die Erzeugung von Objekten geht auch ohne Klassen.

4.4. 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

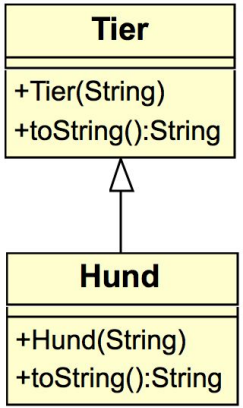
```
function makeDog( name ) {
 return {
   toString: () => "Hund " + name
 };
}

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

bello.name = "Bolle";
        // ohne Wirkung auf Ausgabe,
        // weil die Funktion toString() die Property "name" nicht benutzt

console.log( bello.toString() );  // Hund Bello
```


Douglas Crockford

4.5. Vererbung ohne Klassen

Objekt-Literal

```
function Tier(name) {
 return {
   name: name,
   toString: function () {
     return "Tier(" + this.name + ")";
   }
 };
}

function Hund(name) {
 var that = Tier(name);
 that.toString = function () {
   return "Hund(" + that.name + ")";
 };
 return that;
}

var bolle = Hund("Bolle");
bolle.name = "Bello";
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"

https://stackoverflow.com/questions/27257397/constructor-pattern-by-douglas-crockford

Übungen mit Objekten in JavaScript

```
// Erstellen Sie ein Objekt mit den beiden Eigenschaften a und b
// und den Werten 1 und 2
console.log( record );

// Greifen Sie auf die Eigenschaft a zu:


// Ergänzen Sie record um eine weitere Eigenschaft c mit dem Wert 3


// Schreiben Sie eine Funktion zur Generierung von beliebig großen Objekten
// mit den Eigenschaften und Werten 1,2,3,...
console.log( generate_record( 5 ) );
```

```
// Geben Sie alle Eigenschaftsnamen von record aus (Keys)
console.log( "keys" );

// Geben Sie alle Eigenschaftswerte von record aus
console.log( "values" );

// Geben Sie alle Eigenschaften von record als (Key,Value)-Paare aus
console.log( "entries" );

// Berechnen Sie die Summe aller Werte
console.log( sum( record ) );

// Machen Sie die Summenfunktion zu einer eigenen Eigenschaft von record
// Solche Funktionen nennt man auch Methoden
console.log( "sum = ", record.sum() );
```

4.6. this

- 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
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

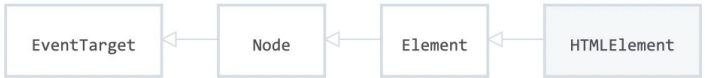
this in Event-Handlern

```
<!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>
```

this
vom Typ HTMLElement

Überschrift wird bei Click rot

Überschrift wird bei Click rot



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>
```

5. Klassen (ab ES6, i.e. ECMAScript 2015)

private property(1)

```
class Dog{
 constructor( name, age ){
   this._name = name;
   this._age = age;
 }

 toString(){
   return `[Hund ${this._name}, Age ${this._age}]`
 }

 get name(){
   return this._name;
 }

 set age( newAge ){
   this._age = newAge;
 }

}
```

Getter & Setter

```
class ColoredDog extends Dog {
 constructor( name, age, color ){
   super( name, age );
   this._color = color;
 }
 toString(){
   return `[${super.toString()}, Color ${this._color}]`
 }
}
```

```
const bello1 = new Dog('Bello', 7);
const bello2 = new ColoredDog('Bolle', 8, 'black');
console.log( `Hunde: ${bello1}, ${bello2}` );
```

```
// Hunde: [Hund Bello, Age 7], [[Hund Bolle, Age 8], Color black]
```

(1) Schreibweise "_name" nur Konvention, keine echte Privatisierung

5.2. Setter und Getter in Klassen

```
class Dog{
 constructor( name, age ){
   this._name = name;
   this._age = age;
 }

 toString(){
   return `[Hund ${this._name}, Age ${this._age}]`
 }

 get name(){
   return this._name;
 }

 set age( newAge ){
   this._age = newAge;
 }

}
```

impliziter Aufruf des Getters

```
// Property mit Getter
console.log( `bello1.name === ${bello1.name}` );  // bello1.name === Bello

// Property ohne Getter
console.log( `bello1.age === ${bello1.age}` ); // bello1.age === undefined

// Property mit Setter
bello1.age = 10;
console.log( `bello1 === ${bello1}` ); // bello1 === [Hund Bello, Age 10]

// Property ohne Setter
bello1.name = 'Brendan'; // ohne Wirkung, da es keinen Setter für name gibt
console.log( `bello1 === ${bello1}` ); // bello1 === [Hund Bello, Age 10]
```

5.3 Privatisierung in Klassen

- Bisher gibt es **noch keine native Unterstützung zur Privatisierung** von Properties in Klassen.
- Aber es gibt schon ein Proposal des ECMA-TC39-Komitees
  - Stage 3 Draft / October 8, 2019 "Public and private instance fields proposal"
- Lösung heute: Nutze Function Scope des Konstruktors zur Privatisierung

```
class Person {
 constructor( name ) {
  let _name = name;
  this.setName = name =>_name = name;
  this.getName = () => _name;
 }
 toString(){
  return `Person(${this.getName()})`;
  // _name hier nicht sichtbar:
  // return `Person(${_name})`; // Fehler
 }
}
```

```
const p1 = new Person( 'Bolle' );
console.log( p1.getName() );
console.log( p1._name ); // undefined
p1.setName('neuer Name');
console.log( p1.toString() ); // Person(neuer Name)
```

https://stackoverflow.com/questions/22156326/private-properties-in-javascript-es6-classes

5.4. Stage 3 Draft / October 8, 2019 "Public and private instance fields proposal"

```
class Counter extends HTMLElement {
  #x = 0;

  clicked() {
    this.#x++;
    window.requestAnimationFrame(this.render.bind(this));
  }

  constructor() {
    super();
    this.onclick = this.clicked.bind(this);
  }

  connectedCallback() { this.render(); }

  render() {
    this.textContent = this.#x.toString();
  }
}
```

für private

https://github.com/tc39/proposal-class-fields

5.5. Klassenausdrücke & Anonyme Klassen

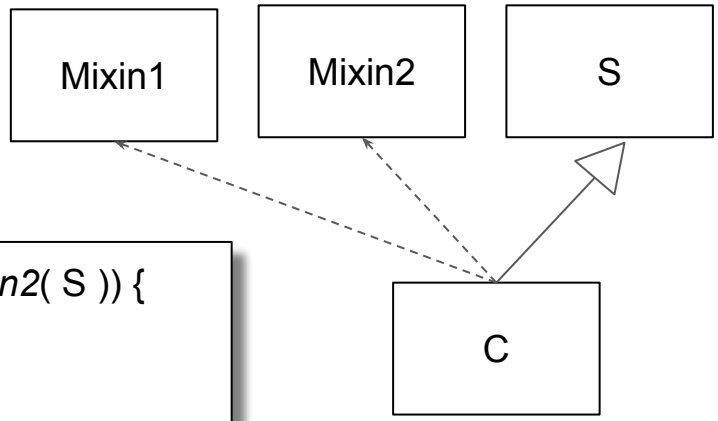
```
class Oberklasse {
 constructor(){
   console.log( 'Hallo aus Oberklasse' );
 }
}

const Unterklasse = class extends Oberklasse {
 constructor(){
   super();
   console.log( 'Hallo aus Unterklasse' );
 }
};

const ABC = Unterklasse;

const x = new ABC();
// Hallo aus Oberklasse
// Hallo aus Unterklasse
```

Klasse ohne Namen

```
// Fabrikfunktion für Klassen
function createClass( name ){
 return class {
   get name(){
     return name;
   }
 }
}

var Dog = createClass( 'Bello');
var bello = new Dog();
console.log( bello.name );
```

5.6. Mixins: Klassen als Parameter & Return Value

```
let Mixin1 = (superclass) => class extends superclass {
  foo() {
    console.log('foo from Mixin1');
    if (super.foo) super.foo();
  }
};

let Mixin2 = (superclass) => class extends superclass {
 foo() {
   console.log('foo from Mixin2');
   if (super.foo) super.foo();
 }
};

class S {
 foo() {
   console.log('foo from S');
 }
}
```




```
class C extends Mixin1( Mixin2( S )) {
 foo() {
   console.log('foo from C');
   super.foo();
 }
}

new C().foo();

// foo from C
// foo from Mixin1
// foo from Mixin2
// foo from S
```

http://justinfagnani.com/2015/12/21/real-mixins-with-javascript-classes/

Zusammenfassung: 3 Sorten von Fabrikfunktionen

1. Funktionen berechnen Funktionen
2. Funktionen berechnen Objekte
3. Funktionen berechnen Klassen

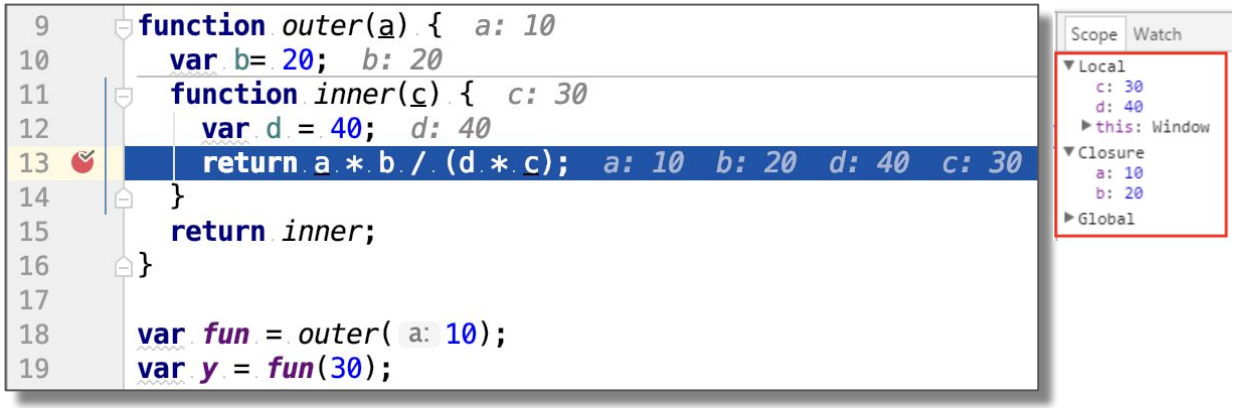
```
function makeMultiplier( multiplikator ){
 return function( multiplikand ){ return multiplikator * multiplikand }
}

function createObject( name ){
 return {
   name: name,
   toString: () => `Object(${name})`
 };
}

function createClass( name ){
 return class {
   get name(){
     return name;
   }
 }
}

var Dog = createClass( 'Bello');
var bello = new Dog();
console.log( bello.name );
```

Zusammenfassung: Mächtigkeit durch Ausdrücke

- Funktionen
  ⇒ Funktionsausdrücke
- Objekte
  ⇒ Objektausdrücke
- Klassen ab ES6
  ⇒ Klassenausdrücke (ab ES6)

- ⇒ Kombinationen aus Funktions-, Objekt- und Klassen-Ausdrücken

6. Closures



Zitat Wikipedia: "A function is first-class; a function is considered to be an object. As such, a function may have properties and methods, such as `.call()` and `.bind()`.[43] A *nested* function is a function defined within another function. It is created each time the outer function is invoked. In addition, each nested function forms a lexical closure: The lexical scope of the outer function (including any constant, local variable, or argument value) becomes part of the internal state of each inner function object, even after execution of the outer function concludes.[44] JavaScript also supports anonymous functions."

https://en.wikipedia.org/wiki/JavaScript#Functional

Zitat Stackoverflow: "A closure is one way of supporting first-class functions; it is an expression that can reference variables within its scope (when it was first declared), be assigned to a variable, be passed as an argument to a function, or be returned as a function result."

https://stackoverflow.com/questions/111102/how-do-javascript-closures-work

Funktionen definieren den Scope

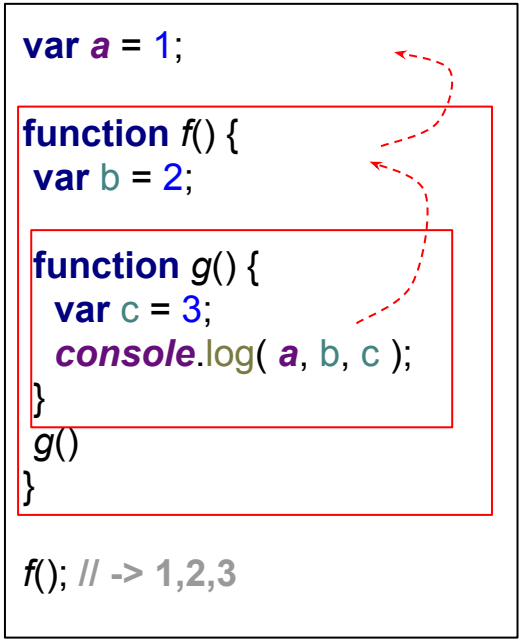
Scope = Sichtbarkeit der Variablen

Jede Funktion definiert einen Scope:

```
function f() {
  var a = 1;
  console.log(a); // works
}

console.log(a); // fails
```

(kein Block Scoping bis ES6)

Scope Chain



Funktionen dürfen geschachtelt werden.

Mit **var** Function Scope statt Block Scope

- Jede Funktion spannt einen neuen Namensraum auf. Die Variablen und Parameter, die in der Funktion deklariert werden, sind außerhalb der Funktion nicht sichtbar.

```
function fun() {
 var x = 1;
 if (x) {
   (function () {
     var x = 2;
   }());
 }
 console.log( x ); // x ist noch 1.
}
```

Closures

Alle JavaScript-Funktionen sind Closures, d.h. Objekte mit scope chain und darüber Zugriff auch auf die Variablen außerhalb der Funktion innerhalb der scope chain.

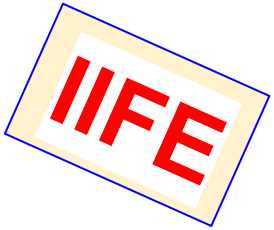
```
var outer = function() {
  var a = 1;
  var inner = function() {
    console.log(a);
  };
  return inner; // this returns a function
};

var fnc = outer(); // execute outer to get inner
fnc();     // call inner returned from outer  -> 1
```

fnc() ruft die Funktion *inner* auf, die auf die Variable a zugreift und deren Wert ausgibt, obwohl a nicht zur Funktion *inner* gehört, sondern zu outer und die Ausführung von outer bereits beendet war, bevor *inner* aufgerufen wurde.

Definition "Closure": A closure is a persistent scope which holds on to local variables even after the code execution has moved out of that block.

http://stackoverflow.com/questions/36636/what-is-a-closure

Closures eignen sich zum Privatisieren



- Verstecken von Variablen
- **⇒ IIFE pattern**

  Immediatley Invoked Function Expression

→ Entwurfsmuster

Es wird ein Objekt zurück gegeben mit einer Methode, die auf die innere Variable zugreift. Die innere Variable selbst gehört nicht zum zurück gegebenen Objekt, bleibt aber erhalten und kann weiter verwendet werden wg. Closure.

```
var counter = (function () {
 var inner = 0; // private

 return { // Objekt mit einer Methode
   increment: function () {
     return ++inner;
   }
 };
}());  // Die Funktion wird definiert und sogleich aufgerufen.

console.log( counter.increment() ); // -> 1
console.log( counter.increment() ); // -> 2
console.log( counter.inner );       // -> undefined
```

"**IIFE pattern**" = immediately-invoked function expression
auch self-executing anonymous function
siehe https://en.wikipedia.org/wiki/Immediately-invoked_function_expression

Auch Parameter gehören zur Scope Chain

Currying ist die Umwandlung einer Funktion mit mehreren Argumenten in eine Funktion mit einem Argument.

$add(a,b) \rightarrow add_a(b)$

Currying ist beliebt in Lisp und Scheme

```
function add (a) {
 return function (b) {
   return a + b;
 }
}

var add3 = add(3);

add3(4); // returns 7
```

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.

Generische Funktionen apply, call und eval

call, apply

Funktion binden, d.h. this definieren

```
const obj = { a: 2, b: 3 };

const f = function( start, factor ){
   return start + factor * ( this.a + this.b )
};

console.log( f.call( obj, 1, 2 ) );  // 11

console.log( f.apply( obj, [ 1, 2 ] ) );  // 11
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply

eval

Den JavaScript-Interpreter aufrufen

```
console.log( eval( " 1 + 2 * 3 " ) );
var x = 7;
console.log( eval( " x === 1 + 2 * 3 " ) );
```

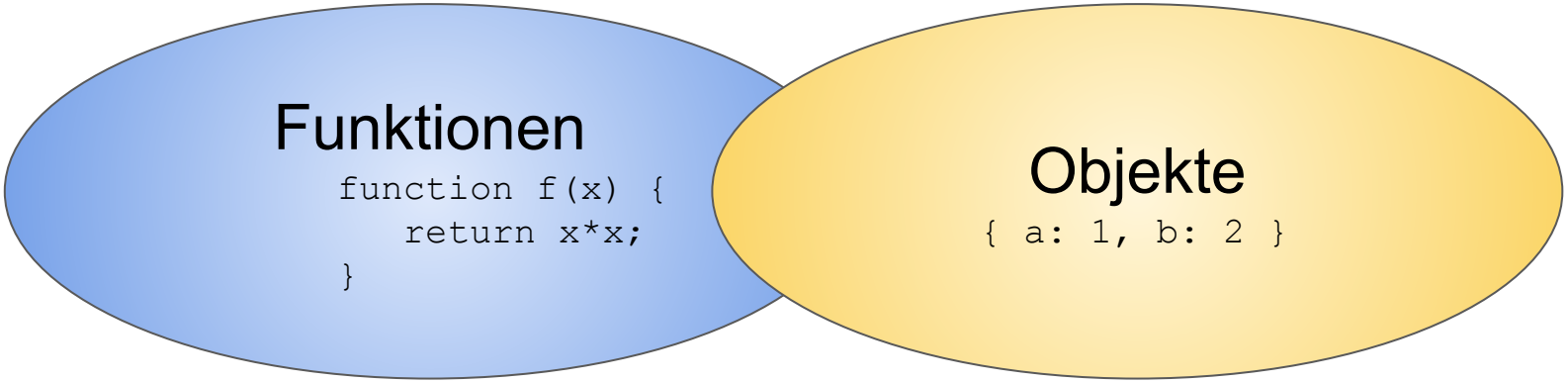
Don't use eval needlessly!

`eval()` is a dangerous function, which executes the code it's passed with the privileges of the caller. If you run `eval()` with a string that could be affected by a malicious party, you may end up running malicious code on the user's machine with the permissions of your webpage / extension. More importantly, third party code can see the scope in which `eval()` was invoked, which can lead to possible attacks in ways to which the similar `Function` is not susceptible.

`eval()` is also generally slower than the alternatives, since it has to invoke the JS interpreter, while many other constructs are optimized by modern JS engines.

There are safer (and faster!) alternatives to `eval()` for common use-cases.

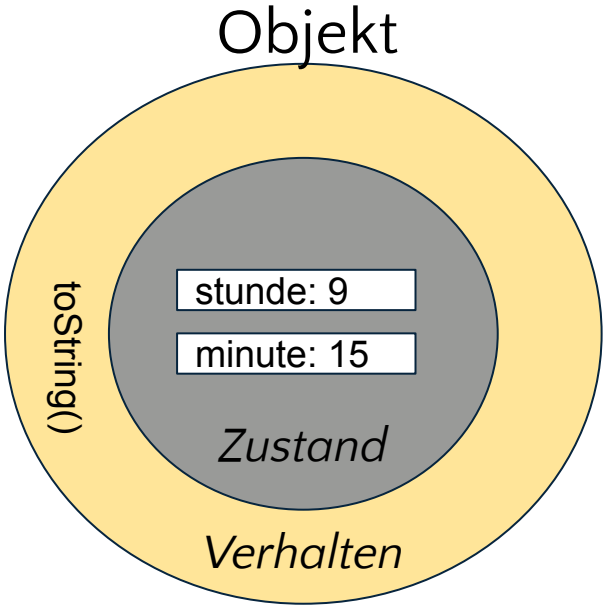
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval



Funktionale Programmierung mit Objekten

1. Objekte und Funktionen
2. Funktionale Dekomposition
3. Lego-Prinzip
4. Closures
5. filter-map-reduce

1. Objekte & Funktionen



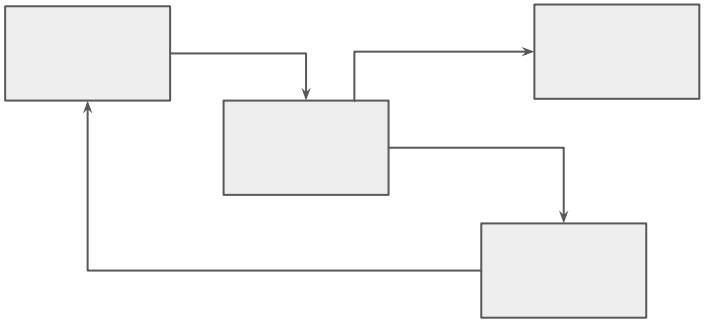
Kapselung ⇒ Function Scope

$$f(x)$$

- einfach
- universell
- seiteneffektfrei
- kombinierbar
- zusammen mit Mengen die erfolgreichste Abstraktion der Mathematik

2. Zerlegung (*Dekomposition*)

Zerlegung in Objekte



Das Paradigma der Organisation interagierender Objekte.

Zerlegung in Funktionen

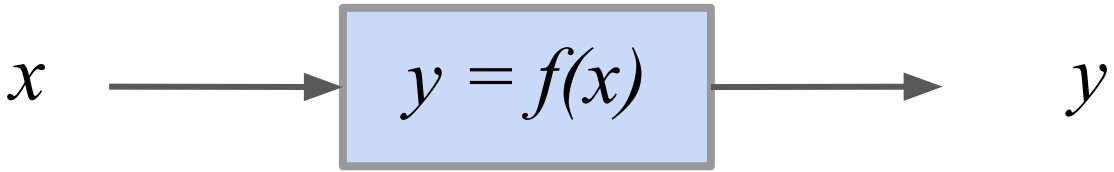
$$f(x) = g(x) * h(x)$$

Das mathematische Paradigma: Mengen und Funktionen sind die erfolgreichsten Abstraktionen der Mathematik.

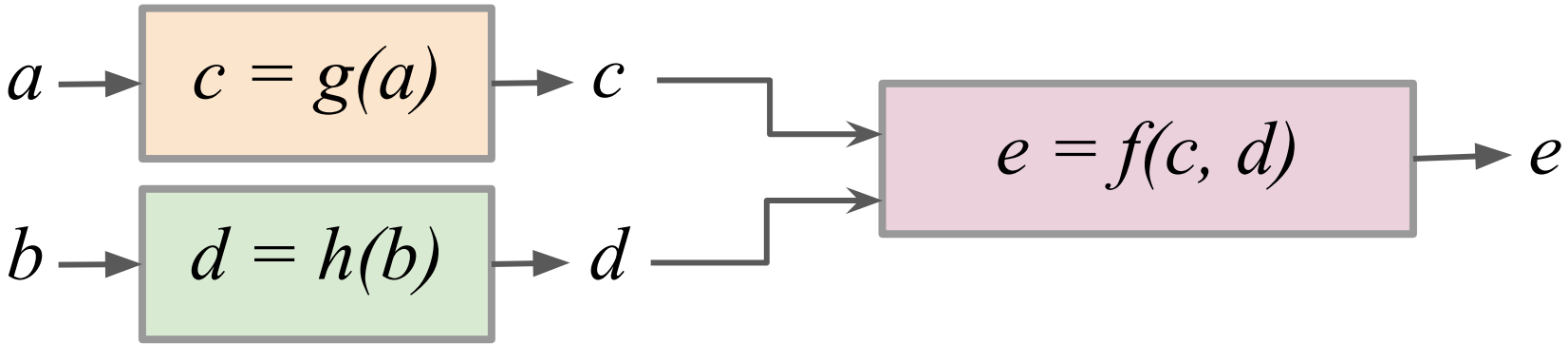
3. Lego-Prinzip für Funktionen

Funktionale Dekomposition

Output berechnen aus Input ohne Seiteneffekte



Baukasten für Funktionen



4. Lambdas und Closures

Lambda stammt aus der Mathematik

$$\lambda(x) \rightarrow x + 1$$

"anonyme Funktion"

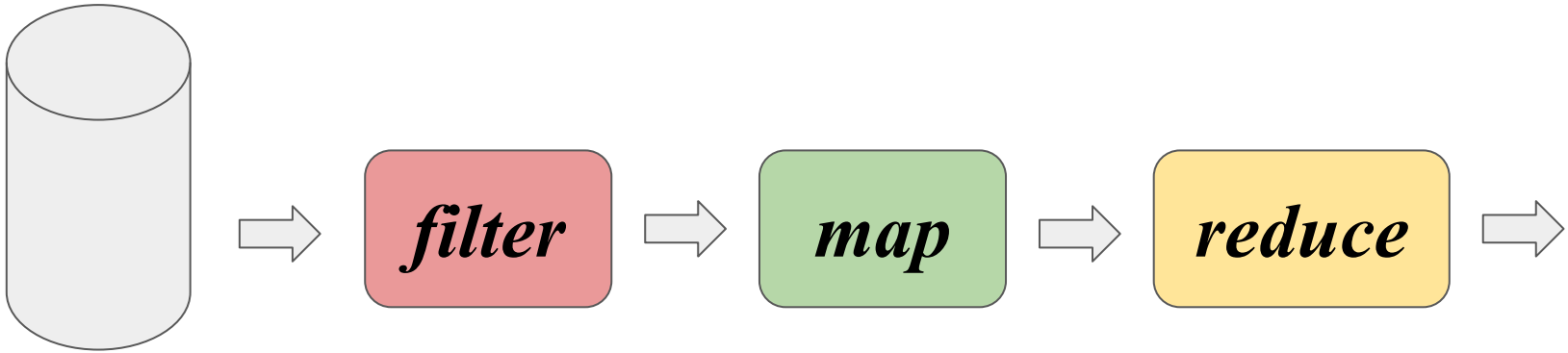
Closure stammt aus der Informatik

```
function add (a) {
  return function (b) {
    return a + b;
  }
}
```

b ist eine freie Variable

a ist gebunden

5. Array Processing



Datenquelle

Folge von Verarbeitungsschritten (Stream Pipeline) wie in Unix

filter

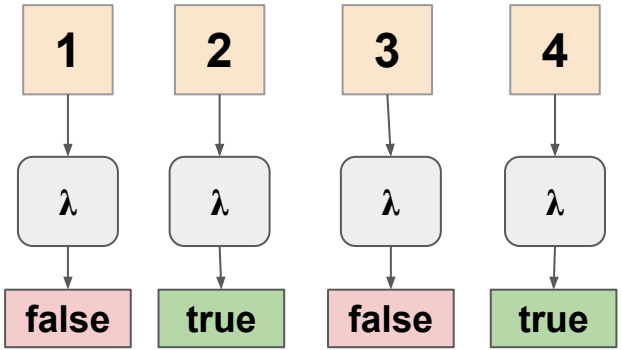
filtert Elemente aus einer Liste

```
[1,2,3,4].filter( x => x % 2 === 0 )
```
1)

identisch

λ

```
[1,2,3,4].filter(function(x){
  return x % 2 === 0;
})
```



Output

$$[\, x \mid x \bmod 2 == 0 \,]$$

1) Lambda-Schreibweise von ECMAScript 6

filter

Array.prototype.filter()

IN THIS ARTICLE

The **`filter()`** method creates a new array with all elements that pass the test implemented by the provided function.

```
function isBigEnough(value) {
  return value >= 10;
}

var filtered = [12, 5, 8, 130, 44].filter(isBigEnough);
// filtered is [12, 130, 44]
```

Syntax

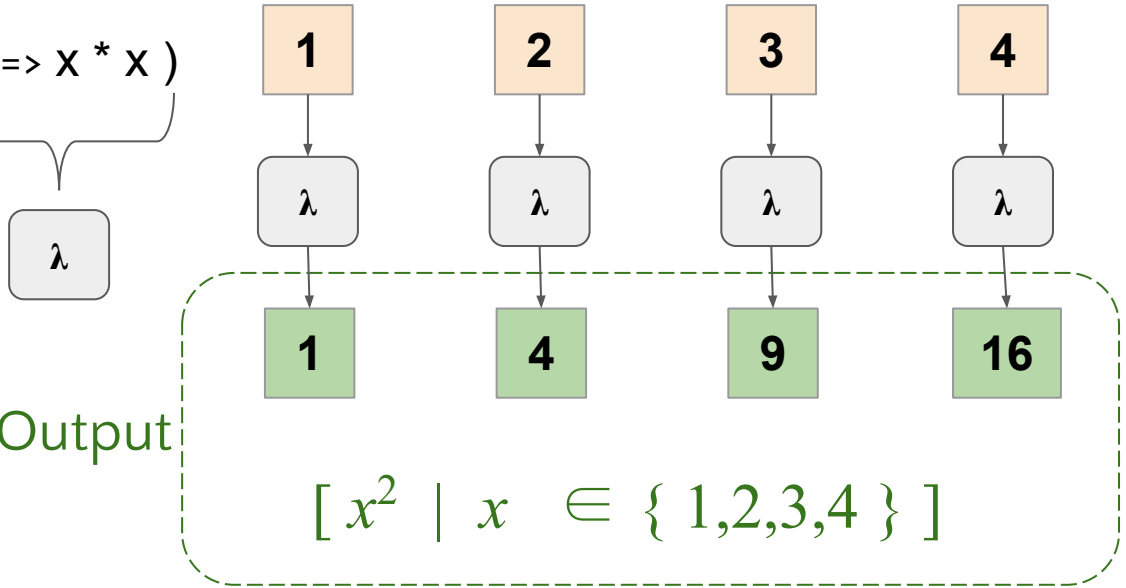
```
var newArray = arr.filter(callback[, thisArg])
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

map

"maps every value of a list"

```
[1,2,3,4].map( x => x * x )
```



map

Array.prototype.map()

IN THIS ARTICLE

The **`map()`** method creates a new array with the results of calling a provided function on every element in this array.

```
var numbers = [1, 5, 10, 15];
var roots = numbers.map(function(x) {
   return x * 2;
});
// roots is now [2, 10, 20, 30]
// numbers is still [1, 5, 10, 15]

var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]
```

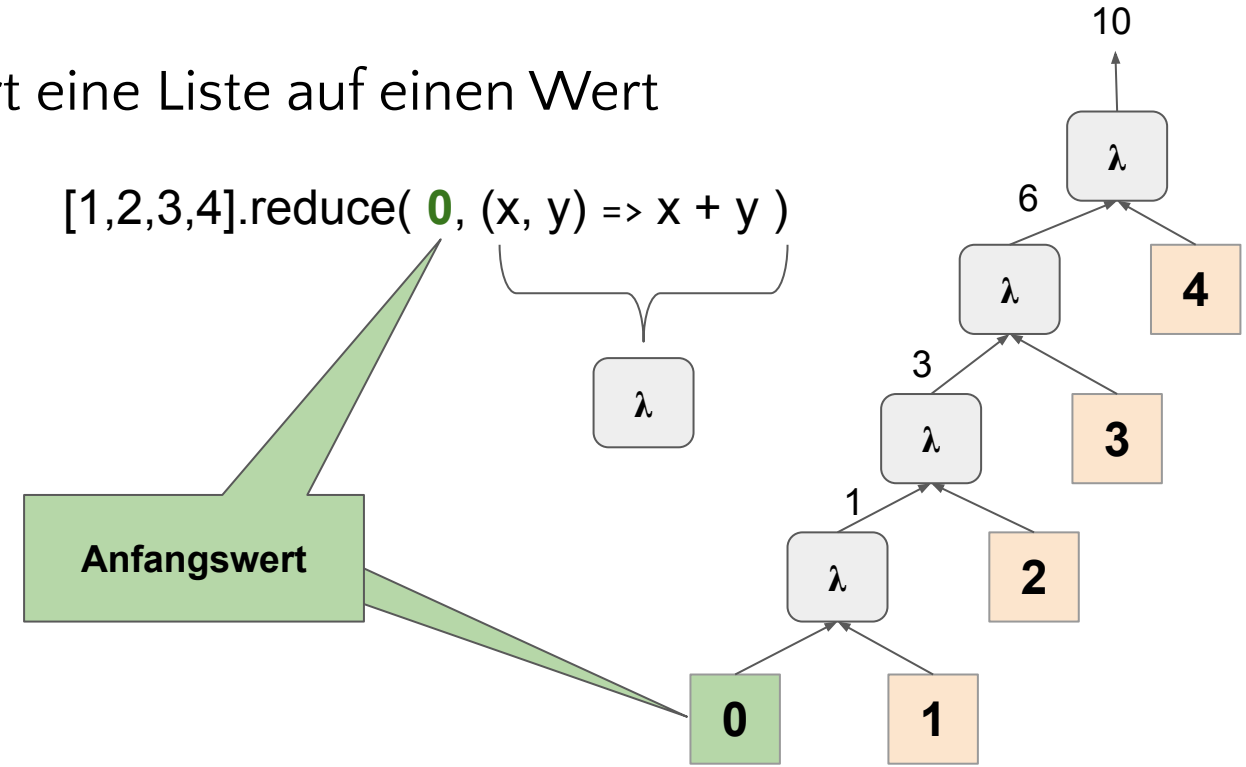
Syntax

```
var new_array = arr.map(callback[, thisArg])
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

reduce

reduziert eine Liste auf einen Wert



reduce

Array.prototype.reduce()

IN THIS ARTICLE

This article needs a technical review. How you can help.

This article needs an editorial review. How you can help.

The **reduce()** method applies a function against an accumulator and each value of the array (from left-to-right) to reduce it to a single value.

```
var sum = [0, 1, 2, 3].reduce(function(acc, val) {
  return acc + val;
}, 0);
// sum is 6

var list1 = [[0, 1], [2, 3], [4, 5]];
var list2 = [0, [1, [2, [3, [4, [5]]]]]];

const flatten = arr => arr.reduce(
  (acc, val) => acc.concat(
    Array.isArray(val) ? flatten(val) : val
  ),
  []
);
flatten(list1); // returns [0, 1, 2, 3, 4, 5]
flatten(list2); // returns [0, 1, 2, 3, 4, 5]
```

Syntax

```
arr.reduce(callback, [initialValue])
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce

forEach(next => do(next))

Array.prototype.forEach()

Languages

The `forEach()` method executes a provided function once for each array element.

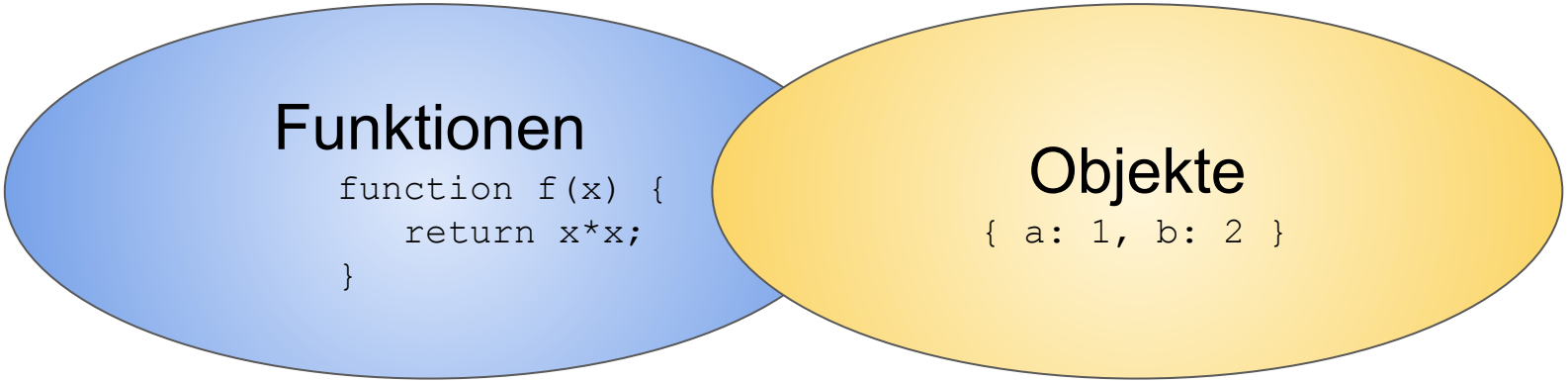
JavaScript Demo: Array.forEach()

```
1  var array1 = ['a', 'b', 'c'];
2
3  array1.forEach(function(element) {
4    console.log(element);
5  });
6
7  // expected output: "a"
8  // expected output: "b"
9  // expected output: "c"
10
```

Syntax

```
arr.forEach(function callback(currentValue[, index[, array]]) {
    //your iterator
}[, thisArg]);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach



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

1. Objekte und Funktionen
2. Funktionale Dekomposition
3. Lego-Prinzip bei Objekten, Funktionen und Klassen
4. Closures
5. filter-map-reduce