

# JavaScript

## Clientseitige Web-Programmierung

# Code in Webseiten

- ▶ JavaScript ist gebräuchlichste Sprache für Webseiten
- ▶ **clientseitige** Programmiersprache für Browser
- ▶ Wird von praktisch allen Browsern unterstützt
- ▶ JavaScript läuft
  - im Browser
  - als Server mit Node.js
- ▶ Andere mögliche Sprachen:
  - JScript, VBScript, TypeScript, ...
- ▶ TypeScript: typisierte Version von Javascript, die in Javascript compiliert/transpiliert
- ▶ Aktuelle Version: ECMAScript 2021 (Juni 2021)

# Version-History

- ▶ Quelle: [https://www.w3schools.com/js/js\\_versions.asp](https://www.w3schools.com/js/js_versions.asp)
- ▶ Stand: April 2021 (ECMAScript fehlt leider in der Liste)

Ver	Official Name	Description
1	ECMAScript 1 (1997)	First Edition.
2	ECMAScript 2 (1998)	Editorial changes only.
3	ECMAScript 3 (1999)	Added Regular Expressions. Added try/catch.
4	ECMAScript 4	Never released.
5	ECMAScript 5 (2009) <a href="#">Read More: JS ES5</a>	Added "strict mode". Added JSON support. Added String.trim(). Added Array.isArray(). Added Array Iteration Methods.
5.1	ECMAScript 5.1 (2011)	Editorial changes.
6	ECMAScript 2015 <a href="#">Read More: JS ES6</a>	Added let and const. Added default parameter values. Added Array.find(). Added Array.findIndex().
7	ECMAScript 2016	Added exponential operator (**). Added Array.prototype.includes.
8	ECMAScript 2017	Added string padding. Added new Object properties. Added Async functions. Added Shared Memory.
9	ECMAScript 2018	Added rest / spread properties. Added Asynchronous iteration. Added Promise.finally(). Additions to RegExp.

ECMAScript 6 is also called ECMAScript 2015.

ECMAScript 7 is also called ECMAScript 2016.

Neuerungen 2019:

<https://alligator.io/js/es2019/>

Neuerungen 2020:

<https://alligator.io/js/es2020/>

Neuerungen 2021:

<https://en.wikipedia.org/wiki/ECMAScript#ES2021>

# Transpiler

- ▶ wandelt ESx-Code in ES5-Code um
- ▶ Babel:

- <https://babeljs.io/setup#installation>

- Installation:

```
C:\_PR\CSharp\PR4\300-399\330_JS_Tutorial>yarn add @babel/core @babel/cli @babel/preset-env --dev
yarn add v1.22.5
[1/4] Resolving packages...
[2/4] Fetching packages...
info fsevents@2.3.2: The platform "win32" is incompatible with this module
```

- babel.config.json / package.json (lokal)

```
{
  "presets": ["@babel/preset-env"]
}
```

```
{
  "devDependencies": {
    "@babel/preset-env": "^7.9.0",
    "@babel/plugin-proposal-class-properties": "^7.0.0",
    "@babel/core": "^7.9.0"
  }
}
```

- Ausführen: **babel -o es5.js es6.js**

# JavaScript in HTML – `<script>`

Code wird direkt in HTML-Seite geschrieben

3 Varianten

- ▶ Mit `<script>`-Tag irgendwo im `<body>`-Tag

```
<script>  
  //Hier JavaScript-Code  
</script>
```

- ▶ Mit `<script>`-Tag im `<head>`-Tag

```
<script>  
  //Hier JavaScript Code (Funktionen)  
</script>
```

- ▶ **js-Datei** mit `<script>`-Tag einbinden

```
<script src="xxx.js"></script>
```

- Variante 3 sollte verwendet werden
- Attribut `type="text/javascript">` seit HTML5 nicht mehr notwendig



# Gemeinsamkeiten JavaScript – Java

Die Syntax von Java und JavaScript ist **vielfach gleich**:

- ▶ Einzeilige Kommentare
- ▶ Geschwungene Klammern, Klammernsetzung
- ▶ Strichpunkt als Befehlsabschluss (stimmt nicht ganz)
- ▶ if – else
- ▶ (Bedingung) ? Ja-Teil : Sonst-Teil;
- ▶ while
- ▶ do – while
- ▶ switch – case – break
- ▶ for
- ▶ Sichtbarkeit von Variablen (stimmt nicht ganz)
- ▶ Operatoren: +, \*, &&, ||, +=, %, ...
- ▶ ==, != (stimmt nicht ganz)

# Unterschiede JavaScript – Java

# var

- ▶ JavaScript kennt keine Typ-Variablen!
- ▶ Einziger Typ: **var**
- ▶ Tatsächlicher Typ ergibt sich aus erster Zuweisung
  - `var x='abc'; var y=123;`
  - es gibt zwar unterschiedliche Typen, für diese jedoch keine Schlüsselwörter, sondern nur var
- ▶ Globale Variable: Variable ohne var
- ▶ **var** soll NIE verwendet werden!!



# Hoisting

- ▶ Egal, wo eine Variable mit **var** in einer Funktion deklariert wird, wird sie behandelt, als ob sie zu Beginn deklariert worden wäre

```
function hoistingExample2() {  
  //any code here  
  {  
    var xxx = 987;  
  }  
  console.log('xxx: ' + xxx);  
}  
hoistingExample2();
```

xxx: 987

```
function hoistingExample2() {  
  var xxx;  
  //any code here  
  {  
    xxx = 987;  
  }  
  console.log('xxx: ' + xxx);  
}
```

- ▶ Es gibt **keine Blockgültigkeit** für Variable mit **var**

```
function hoistingExample() {  
  var val = 123;  
  console.log('1) val: ' + val);  
  {  
    var val = 987;  
    console.log('2) val: ' + val);  
  }  
  console.log('3) val: ' + val);  
}
```

1)	val: 123
2)	val: 987
3)	val: 987

# Block-Scope

- ▶ In ES5 gibt es keinen Block-Scope
- ▶ Nur Funktionen definieren eigenen Scope
- ▶ Neues Schlüsselwort (seit ES6) **let**

```
function blockScopeES5() {  
  logHeading('blockScope ES5');  
  var x = 123;  
  console.log(x);  
  if (true) {  
    var x = 666;  
    console.log(x);  
  }  
  console.log(x);  
}
```

blockScope ES5
123
666
666

```
function blockScopeES6() {  
  logHeading('blockScope ES6');  
  var x = 123;  
  console.log(x);  
  if (true) {  
    let x = 666;  
    console.log(x);  
  }  
  console.log(x);  
}
```

blockScope ES6
123
666
123

# Konstanten

- ▶ In ES5 gibt es keine unveränderliche Variablen (außer mit großem Aufwand)
- ▶ Neues Schlüsselwort (seit ES6) **const**

```
var x = 123;  
const CONST = 666;  
var x = 789;  
//CONST = 777; //this is not allowed  
console.log(`x = ${x}`);  
console.log(`CONST = ${CONST}`);
```

# Zusammenfassung

- ▶ **var** niemals verwenden
- ▶ Variable mit **const** deklarieren
- ▶ Nur falls sich der Wert ändern kann, mit **let** notieren
  - das ist viel seltener der Fall, als vermutet

# Objekte / Klassen

## ▶ JavaScript

- ist eine objektorientierte Sprache
- Ist Prototyp-basiert
- kennt **keine Klassen!**
- Erlaubt das Erzeugen eigener Typen

## ▶ Es gibt nur folgende Typen

- Primitive Typen
  - **number**
  - **string**: single/double quotes erlaubt
  - **boolean**
  - **undefined**
- Objekt-Typen
  - **object**
  - **function**

```
> var n=1; typeof n;  
"number"
```

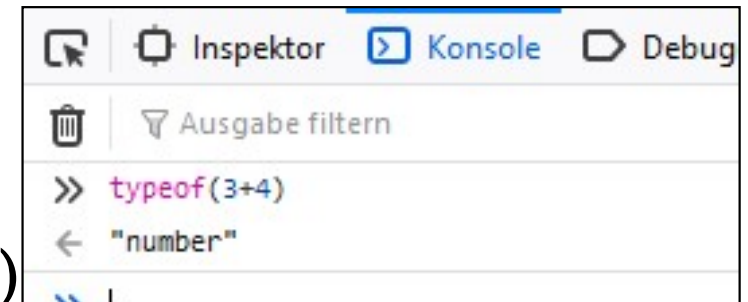
## ▶ Welchen Typ hat eine Variable → **typeof**

# Übungen Typen



# Übungen Typen

- ▶ Probiert folgendes aus
  - auf der Browser-Konsole (<F12>)
  - oder mit Node (node installieren, dann **node**)



```
C:\>node
Welcome to Node.js v12.10.0.
Type ".help" for more information.
> typeof(3+4)
'number'
> .exit
```

```
var n=1; typeof n;
var n2=1.234; typeof n2;
var s='abc'; typeof s;
var s2="xyz"; typeof s2;
typeof unkownVar;
var xxx=undefined; typeof xxx;
var yyy=null; typeof yyy;
function calc(x,y){ return x+y; }
typeof calc;
typeof calc(4,5);
typeof unkonwnFunction;
s=1234; typeof s;
n="xxx"; typeof n;
```

```
var n3=0377; n3;
var n4=0x11; n4;
var n5=2e4; n5;
var n6=Infinity; n6;
typeof n6;
var n7=123/0; n7;
var n8=NaN; n8;
n8=Math.sqrt(-1)
typeof n8;
var n9=123*"abc"; n9;
```

# undefined vs. null

- ▶ In JavaScript gibt es zwei Schlüsselwörter für fehlende Werte:
- ▶ **null** ist ein Objekt, dessen Wert null ist
  - Heißt: hat keinen Wert
- ▶ **undefined** ist kein Objekt, der Typ ist undefined
  - Heißt: gibt es nicht
  - Nicht initialisierte Werte sind undefined
- ▶ Bsp:
  - `var xxx; //xxx is undefined`
  - `var yyy=null; //yyy is null`
  - `var zzz=undefined; //don't do this!`

# type conversion

- ▶ JavaScript konvertiert bei Bedarf Typen automatisch
  - string ↔ number
  - string,number,... ↔ boolean
- ▶ Kann zu überraschenden Ergebnissen führen
  - Bsp: `var s='5'; var n=3;`
- ▶ Bei Vergleichen werden alle Werte auf **true** umgewandelt, außer (diese sind **false**):
  - Leerstring `""`
  - **null**
  - **undefined**
  - Zahl **0** u. **-0**
  - Zahl **NaN**
  - **false**

>	s+n;
	"53"
>	n+s;
	"35"
>	s*3;
	15

▶ Daher spricht von **truthy** u. **falsy** values

# Übungen type conversions

# Übungen type conversions

- Probiert folgendes aus (Browser oder Node)

```
var s='5', n=3, t=true, f=false, 1+'2'+3;
noDef, o=null;                    1+(2+'3');
typeof s;                        3+o;
typeof n;                        3+noDef;
typeof t;                        typeof noDef;
typeof f;                        typeof !!noDef;
typeof noDef;                    typeof !!o;
typeof o;                        if(s) console.log('yes'); else
n+t;                             console.log('no');
n+f;                             s='';
s+t;                             if(s) console.log('yes'); else
t+f;                             console.log('no');
s+calc;                          if(n) 'yes'; else 'no';
s+calc(1,2);                     n=0;
s+calc();                       n?'yes':'no';
1+2+3;                          if(3+null) 'yes'; else 'no';
1+2+"3";                        if(3+undefined) 'yes'; else
                                'no';
```

# Vergleiche: ==, !=, ===, !==

- ▶ Aufgrund der automatischen Typumwandlung kommt es auch bei Vergleichen teilweise zu unerwarteten Ergebnissen
- ▶ == u. != wandelt unterschiedliche Typen um
- ▶ === u. !== liefert nur **true**, wenn auch die Typen gleich sind

▶ Bsp:

> '66'==66
true
> '66'===66
false

- ▶ Überprüfen auf undefined:

- **xxx===undefined**
- **typeof xxx === 'undefined'**



# Equality Table

(<https://dorey.github.io/JavaScript-Equality-Table/>)

	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[[] ]	[0]	[1]	NaN
true	✓		✓					✓												✓	
false		✓		✓					✓		✓					✓		✓	✓		
1	✓		✓					✓												✓	
0		✓		✓					✓		✓					✓		✓	✓		
-1					✓					✓											
"true"						✓															
"false"							✓														
"1"	✓		✓					✓												✓	
"0"		✓		✓					✓										✓		
"-1"					✓					✓											
""		✓		✓							✓					✓		✓			
null												✓	✓								
undefined												✓	✓								
Infinity														✓							
-Infinity															✓						
[]		✓		✓							✓										
{}																					
[[]]		✓		✓							✓										
[0]		✓		✓					✓												
[1]	✓		✓					✓													
NaN																					

Ergebnisse, wenn man **==** verwendet

Moral of the story:  
**Always use 3 equals unless you have a good reason to use 2!**

# Übungen Vergleiche

# Übungen Vergleiche

- ▶ Probiert folgendes aus (Browser oder Node)

```
66=='66';  
66=== '66';  
var nodef;  
nodef==null;  
nodef==undefined;  
nodef===null;  
nodef===undefined;  
nodef==false;  
nodef==true;  
nodef==0;  
!!nodef==0;  
0=='';  
0=== '';
```

# Objekte

- ▶ Objekte werden wie JSON-Maps erzeugt
- ▶ **Erzeugen:**
  - `var empty = {};`
  - `var obj = {aa: 'Hansi', bb: 'Susi'};`
  - Bei den keys dürfen die Anführungszeichen fehlen (außer sie enthalten Sonderzeichen)
- ▶ **Zugriff:**
  - `var name = obj['aa'];`
  - `var name2 = obj.bb;`
- ▶ **Zuweisung** wie in Java:
  - `obj['aa'] = 'Franzi';`
  - `obj.aa = 'Franzi';`
- ▶ Nichtvorhandene Properties eines Objekts sind **undefined**

# Übungen Objekte

# Übungen Objekte

- ▶ Probiert folgendes aus (Browser oder Node)




```
var person={ firstname:'Hansi', lastname:'Huber', age:66};
typeof person;
person;
console.log(person);
console.dir(person);
console.table(person);
person.lastname
person['firstname']
person.firstname + ' ' + person.lastname
person.city='Wien'
person
console.dir(person)
if(person) 'yes'; else 'no';
var xxx={#&*: 'aaa'};
var xxx={'#&*': 'aaa'};
```



# Ausgabe

# Ausgabe 1 / 4

- ▶ Es gibt mehrere Möglichkeiten, auf die Console zu schreiben

>	<code>console.log('Hallo')</code>
	Hallo
>	<code>console.info('This is for your information')</code>
	This is for your information
>	<code>console.warn('Achtung')</code>
	Achtung
>	<code>console.error('Hoppala')</code>
	Hoppala

- ▶ Für Objekte: `console.dir`

>	<code>console.log(window.screen)</code>										
	Screen { availWidth=1920, availHeight=1050, width=1920, mehr... }										
>	<code>console.dir(window.screen)</code>										
	<table><tr><td>mozLockOrientation</td><td><code>mozLockOrientation()</code></td></tr><tr><td>mozUnlockOrientation</td><td><code>mozUnlockOrientation()</code></td></tr><tr><td>availWidth</td><td>1920</td></tr><tr><td>availHeight</td><td>1050</td></tr><tr><td>width</td><td>1920</td></tr></table>	mozLockOrientation	<code>mozLockOrientation()</code>	mozUnlockOrientation	<code>mozUnlockOrientation()</code>	availWidth	1920	availHeight	1050	width	1920
mozLockOrientation	<code>mozLockOrientation()</code>										
mozUnlockOrientation	<code>mozUnlockOrientation()</code>										
availWidth	1920										
availHeight	1050										
width	1920										

# Ausgabe 2/4

- ▶ Tabellarische Ausgabe: `console.table()`

```
>> var person={ firstname:'Hansi', lastname:'Huber', age:66};
```

```
>> console.log(person)
```

```
Object { firstname: "Hansi", lastname: "Huber", age: 66 }
```

```
>> console.dir(person)
```

```
{firstname: "Hansi", lastname: "Huber", age: 66}
```

```
  age: 66
```

```
  firstname: "Hansi"
```

```
  lastname: "Huber"
```

```
  ▶ __proto__: Object
```

```
>> console.table(person)
```

```
console.table():
```

(Position)	Wert
age	66
firstname	"Hansi"
lastname	"Huber"

# Ausgabe 3/4

## ▶ Zeitmessung

- Timer starten: `console.time('id')`
- Timer beenden: `console.timeEnd('id')`
- ➔ Ausgabe der verstrichenen Zeit in Millisekunden

```
>> console.time('timerA')
    timerA: Timer gestartet
← undefined
>> console.time('timerB')
    timerB: Timer gestartet
← undefined
>> console.timeEnd('timerB')
    timerB: 8411.03ms
← undefined
>> console.timeEnd('timerA')
    timerA: 15753.48ms
← undefined
>> console.timeEnd('timerA')
← undefined
```

# Ausgabe 4/4

- ▶ Oft hilft die Umwandlung in einen String:  
**JSON.stringify()**

```
>> var person={ firstname:'Hansi', lastname:'Huber', age:66};
```

```
>> JSON.stringify(person)
```

```
← '{"firstname":"Hansi","lastname":"Huber","age":66}'
```

- ▶ Farbige Ausgabe im Browser mit CSS:
- ▶ Prefix '**%c**' + Style

```
>> console.log('%cHello World', 'background-color: blue; color: white');
```

Hello World

```
>> console.log('%cHello World', 'color: red; font-size:2em; font-style:italic;');
```

*Hello World*

# Strings



# „Klasse“ String

- ▶ Praktisch wie in Java
- ▶ Anführungszeichen: " oder ' (sind gleichwertig)
- ▶ Erzeugen: `let s = 'abc';`
- ▶ **length** im Gegensatz zu Java ohne Klammern!
- ▶ Restlichen Methoden wie gewohnt:
  - `toUpperCase()`
  - `toLowerCase()`
  - `indexOf('_')`
  - `charAt(3)`
  - `substr(idxVon, anzahl)`
  - `substring(idxVon, idxBis)`
  - `split('.')`
  - `startsWith('x'), endsWith('x')`

...

# Weitere String-Methoden

- ▶ Seit ES6 gibt es einige neue Methoden

```
const s = 'Hansi';  
console.log(`${s}.startsWith("Ha") = ${s.startsWith('Ha')}`);  
console.log(`${s}.endsWith("I") = ${s.endsWith('I')}`);  
console.log(`${"x".repeat(9)} = ${'x'.repeat(9)}`);  
console.log(`${s}.includes("ans") = ${s.includes('ans')}`);
```

```
Hansi.startsWith("Ha") = true  
Hansi.endsWith("I") = false  
"x".repeat(9) = xxxxxxxxxxx  
Hansi.includes("ans") = true
```

- ▶ Neu ES2019: trimStart, trimEnd

# Template-Strings

- ▶ Strings zur Ausgabe mit Platzhaltern
- ▶ Werden mit `` markiert (**backtick**, 0x96)
- ▶ Platzhalter: **`${ }`**
- ▶ Man kann darin auch rechnen (oder Funktionen aufrufen)

```
var name = 'Hansi', age = 66;  
console.log(`Hallo ${name}, dein Alter ist ${age}`);  
name = 'Susi', age = 45;  
console.log(`Hallo ${name}, dein Alter ist ${age - 5}`);
```

```
Hallo Hansi, dein Alter ist 66
```

```
Hallo Susi, dein Alter ist 40
```

# Funktionen

# Funktionen 1 / 2

- ▶ Funktions-Definition **ohne Parameter- u. Return-Typ**
- ▶ Funktionsaufruf aber wie in Java

```
function calc(x,y)
{
    const sum = x + y;
    return sum;
}
```

```
const sum = calc(4,5);
```

```
const calc = function(x,y)
{
    const sum = x + y;
    return sum;
}
```


- ▶ JavaScript-Funktionen sollten bei HTML immer in einer externen Datei notiert werden!

# Funktionen 2/2

- ▶ „functions are first class objects“
- ▶ Funktionen verhalten sich wie Objekte
- ▶ Das hat zur Folge
  - Kann Funktionen als Wert einer Objekt-Property setzen
  - Funktionen können Attribute haben
  - Funktionen können weitere Funktionen haben

```
function calc(x,y)
{
    var sum = x + y;
    return sum;
}

var obj={
    s: 'abc',
    n: 123,
    f: calc
}
```



```
> obj.s
"abc"
> calc(4,5)
9
> obj.f(3,4)
7
```

```
> calc.aa=66
66
> calc.bb=77
77
> calc.aa+calc.bb
143
```

```
> typeof calc
"function"
> typeof obj.f
"function"
> typeof calc.g
"function"
```

```
> calc.g=function(x,y){return x*y;}
function(x, y)
> calc.g(8,9)
72
```

# Lambda-Expressions 1 / 2

- ▶ Javascript kann Lambda-Expressions
- ▶ Regeln wie in C#

```
var quadrat = function(x) {  
    return x * x;  
};  
var q = quadrat(3);  
console.log('Quadrat von 3 = ' + q);
```

```
var arr = [4, 2, 5, 3, 1, 6];  
console.log('arr', arr);  
arr.sort(function(a, b) {  
    return a - b;  
});  
console.log('sorted', arr);
```

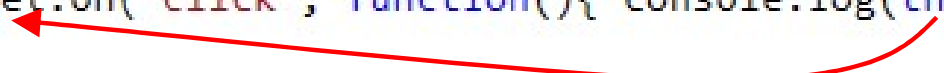
```
const quadrat = x => x * x;  
const q = quadrat(3);  
console.log(`Quadrat von 3 = ${q}`);
```

```
const arr = [4, 2, 5, 3, 1, 6];  
console.log('arr', arr);  
arr.sort((a, b) => a - b);  
console.log('sorted', arr);
```


# Lambda-Expressions 2/2

- ▶ Für Lambda-Funktionen gilt:
  - **this** bezieht sich auf den Kontext der Definition und nicht auf den Aufrufer

```
function C1(target){  
  this.type='C1';  
  console.log(target.selector);  
  target.on('click', function(){ console.log(this.type);});  
}
```



```
function C2(target){  
  this.type='C2';  
  target.on('click', () => {console.log(this.type)});  
}
```



```
var btn = $('#btnLambda');  
var c1 = new C1(btn);  
var c2 = new C2(btn);
```

button
C2



# Variable Parameterlisten

- ▶ Variable Parameter werden mit ... angegeben
- ▶ Sie werden (wie in Java) als Array behandelt

```
function fullName(surName, ...givenNames) {  
  let s = surName;  
  givenNames.forEach(x => s += ` ${x}`);  
  return s;  
}  
  
const name = fullName('Huber', 'Hans', 'Franz', 'Sepp');  
console.log(name);
```

- ▶ Werden auch als **Rest-Parameter** bezeichnet, weil alle restlichen Parameter hier gesammelt werden

# Default-Parameter

- ▶ Auch Default-Parameter sind jetzt möglich
- ▶ funktionieren wie in C#

```
function fullName(surName = 'Huber', firstName = 'Hansi') {  
    return surName + ' ' + firstName;  
}  
  
console.log(fullName());  
console.log(fullName('Lehner'));  
console.log(fullName('Mair', 'Maxi'));
```

Huber Hansi
Lehner Hansi
Mair Maxi

# Monkey patch

- ▶ Man kann Funktionen einfach ersetzen
- ▶ Grund: Eine Funktion ist ja auch nur eine Property eines Objekts

```
var L = console.log
console.log = function (x) {
|   return 'I mog nimma';
}
L('x');
console.log('x');
```

```
console.log('aaa')
"I mog nimma"
```

# Übungen Funktionen

# Übungen Funktionen

## ► Probiert folgendes aus (Browser oder Node)

```
function sum(x,y) { return x + y; }
var mult = function(x,y) { return x * y; }
sum(3,4);
mult(3,4);
var doCalc = function(x,y,func) { return func(x,y); }
doCalc(3,4,sum);
doCalc(3,4,mult);
doCalc(3,4);
var doCalc = function(x,y,func) { return func===undefined?0:func(x,y); }
doCalc(3,4);
var doCalc = function(x,y,func) { func=func||mult; return func(x,y); }
doCalc(3,4,sum);
var L=console.log
console.log=function(x){return 'I mog nimma';}
L('x');
console.log('x');
function f(a,b,c){return `${a}/${b}/${c}`;}
f(1);
f(1,2);
```

# Weitere Begriffe

# Exceptions

- ▶ Bei Tippfehler bricht Ausführung einfach ab
- ▶ Es wird dabei aber Exception geworfen
- ▶ Am besten diese mit **catch** fangen u. auf die Konsole ausgeben
- ▶ Man kann mit **throw** eigene Exception werfen
- ▶ Dabei kann beliebiges Objekt verwendet werden

```
try{
  unknownFunc();
}
catch(exc) {
  console.log(exc);
}
```

ReferenceError: unknownFunc is not defined

index.js (Zeile 17)

unknownFunc();

```
try{
  //do something
  throw {message: 'My dummy message'};
}
catch(exc) {
  console.log(exc);
}
```

Object { message="My dummy message" }

index.js (Zeile 25)

```
try{
  //do something
  var e = {
    aaa: 'any strnig value',
    bbb: 1234,
    message: 'should always be included',
  };
  throw e;
}
catch(exc) {
  console.log(exc);
}
```

Object { aaa="any strnig value",  
bbb=1234, message="should always be  
included" }

index.js (Zeile 38)

# Destructuring 1 / 2

- ▶ Aus Array/Object Variable erzeugen
- ▶ Umkehrung der Erzeugung

```
const arr = ['aaa', 'bbb', 'ccc'];  
const [s1, s2, s3, s4] = arr;  
console.log(`${s1}/${s2}/${s3}/${s4}`);
```

aaa/bbb/ccc/undefined

```
const person = { lastname: 'Huber', firstname: 'Hansi' };  
const { lastname: x, firstname: y } = person;  
console.log(`${x} ${y}`);
```

Huber Hansi

```
const { lastname } = person;  
console.log(`lastname=${lastname}`);
```

lastname=Huber

- ▶ Nicht vorhandene Werte sind **undefined**
- ▶ Funktioniert auch mit **let** bzw. **var**



# Destructuring 2/2

- ▶ Mit **Destructuring** kann eine Funktion **mehr als einen Wert zurückgeben** bzw. Arrays als Parameter übernehmen

```
function getMinMax(values) {  
  const maxVal = Math.max(...values);  
  const minVal = Math.min(...values);  
  return [minVal, maxVal];  
}
```

```
const [min, max] = getMinMax([12, 6, 3, 74, 2, 45]);  
console.log(`min/max = ${min}/${max}`);
```

2/74

- ▶ Nicht vorhandene Werte sind **undefined**

# Spread Operator 1 / 2

- ▶ Der Spread operator `...` teilt Arrays oder Objekt auf einzelne Werte auf

```
function dummy(a, b) {  
  return a + b;  
}  
const arr = [1, 2, 3, 4, 5];
```

```
const resA = dummy(arr);
```

```
1,2,3,4,5undefined
```

```
const resB = dummy(...arr);
```

```
3
```

- ▶ Restliche Werte werden ignoriert
- ▶ Nicht vorhandene Werte sind **undefined**

# Spread Operator 2/2

- ▶ Damit kann man einige Probleme elegant lösen:

```
const arr = [1, 2, 3, 4, 5];  
const arr2 = [6, 7];
```

```
const objA = { aa: 123, bb: 678 };  
const objB = { bb: 555, cc: 666, dd: 777 };
```

- ▶ Array kopieren: 

```
const arrCopy = [...arr];
```
- ▶ Array mergen: 

```
const arrMerged = [...arr, ...arr2];
```
- ▶ Objekt klonen: 

```
const clonedObject = { ...objA };
```
- ▶ Objekt mergen:

```
const fullObject = { ...objA, ...objB };
```

```
{"aa":123,"bb":555,"cc":666,"dd":777}
```

# Ausgewählte „Klassen“

# „Klasse“ Date

- ▶ Zugriff auf **Variablen** und **Methoden** wie in Java mit „.“
- ▶ Anlegen mit Schlüsselwort new
- ▶ Neues aktuelles Datum: **var d = new Date();**

- ▶ Methoden:

Funktion	Beschreibung
getYear()	Jahr 2-stellig
getFullYear()	Jahr 4-stellig
getMonth()	Monat (Werte 0-11 !)
getDate()	Monatstag
getDay()	Wochentag
getHours()	Stundenteil der Uhrzeit
getMinutes()	Minutenteil der Uhrzeit
getSeconds()	Sekundenteil der Uhrzeit
getMilliseconds()	Millisekunden
getTime()	Zeitpunkt
toLocaleString()	Zeitpunkt in lokales Format konvertieren



- ▶ Es gibt auch entsprechende setter-Methoden
- ▶ Mehrere Konstruktoren vorhanden `Date(year,month,day)`, `Date(year,month,day,hour,min,sec)`

# Übungen Date

# Übungen Date

- ▶ Probiert folgendes aus (Browser oder Node)

```
let d = new Date();  
typeof d;  
d.toString();
```

- Datum heute 0 Uhr anlegen
- Tag auf 10 setzen
- Uhrzeit auf 14:21:45 setzen
- Monat auf Februar setzen
- Millisekunden seit 1.1.1970
- Wochentag ausgeben

# „Klasse“ Array

- ▶ Array für beliebige Typen
- ▶ Index startet bei 0
- ▶ Erweitert sich automatisch!
- ▶ **Erzeugen:**
  - Leeres Array: `var arr = new Array();`  
besser: `const arr = [];`
  - Mit Werten: `const names = new Array('Hansi', 'Susi');`
  - Als JSON-Array: `const arr = ['aa', 'bb', 'cc'];`
- ▶ **Zugriff** wie in Java:
  - `const name = names[0];`
- ▶ **Zuweisung** wie in Java:
  - `names[2] = 'Franzi';`
- ▶ Es gibt jede Menge Array-Funktionen



# Array-Ausgabe

- Wieder mit `console.log()`, `console.dir()` oder `console.table()`:

```
>> var arr = ["aa", "bb", "cc"];
```

```
>> console.log(arr)  
Array [ "aa", "bb", "cc" ]
```

```
>> console.dir(arr)  
["aa", "bb", "cc"]  
  0: "aa"  
  1: "bb"  
  2: "cc"  
  length: 3  
  ▶ __proto__: Array[0]
```

```
>> console.table(arr)  
console.table():
```

(Position)	Wert
0	"aa"
1	"bb"
2	"cc"

# Array Funktionen I

Method	Notes
pop	Returns the last item in the Array and removes it from the Array.
push	Adds the item to the end of the Array.
shift	Returns the first item in the Array and removes it from the Array.
unshift	Inserts the item(s) to the beginning of the Array.
indexOf	Searches the Array for specific elements.
lastIndexOf	Returns the last item in the Array which matches the search criteria.
reverse	Reverses the Array so the last item becomes the first and vice-versa.
concat	Joins multiple Arrays
join	Joins all the Array elements together into a string.
slice	Returns a new array from the specified index and length.
sort	Sorts the array alphabetically or by the supplied function.
splice	Deletes the specified index(es) from the Array.
toSource	Returns the source code of the array.
toString	Returns the Array as a string.

# Array Funktionen II

Method	Notes
every	Calls a function for every element of the array until false is returned.
filter	Creates an array with each element which evaluates true in the function provided.
forEach	Executes a specified function on each element of an Array
map	Creates a new array with the result of calling the specified function on each element of the Array.
some	Passes each element through the supplied function until true is returned.

Damit kann man sehr ähnlich wie mit **LINQ** arbeiten

► Neu seit ES2019: **Object.fromEntries**

```
const obj = Object.fromEntries([['at', 'Austria'], ['de', 'Germany']]);
```

```
obj
```

```
► Object { at: "Austria", de: "Germany" }
```

# for-Schleife

```
const arr = ['a', 'b', 'c', 'd', 'e', 'f'];  
const obj = { lastname: 'Huber', firstname: 'Fritzi', age: 66 };
```

Vorsicht bei for-Schleifen!!!

for bei Array:

```
for (let i = 0; i < arr.length; i++) {  
  console.log(`${i} --> ${arr[i]}`);  
}
```

0	-->	a
1	-->	b
2	-->	c
3	-->	d
4	-->	e
5	-->	f

for-in bei Array:

```
for (let i in arr) {  
  console.log(i);  
}
```

0
1
2
3
4
5

0	-->	a
1	-->	b
2	-->	c
3	-->	d
4	-->	e
5	-->	f

```
for (let i in arr) {  
  console.log(`${i} --> ${arr[i]}`);  
}
```

for-in bei Objekt:

```
for (let key in obj) {  
  console.log(`${key} --> ${obj[key]}`);  
}
```

lastname	-->	Huber
firstname	-->	Fritzi
age	-->	66

forEach

```
arr.forEach( function(item) {  
  console.log(item);  
});
```

a
b
c
d
e
f

# for-in vs for-of

Bei einer for-Schleife meint man fast immer for-of  
Ausgangspunkt:

```
const arr = ['a', 'b', 'c'];  
arr.abc = 123;
```

for-in:

```
for (let i in arr) {  
  console.log(`for-in: ${i}`);  
}
```

```
for-in: 0  
for-in: 1  
for-in: 2  
for-in: abc
```

for-of:

```
for (let i of arr) {  
  console.log(`for-of: ${i}`);  
}
```

```
for-of: a  
for-of: b  
for-of: c
```

# Übungen Arrays

# Übungen Arrays

- Probiert folgendes aus (Browser oder Node)

```
var a = [1,2,3,4,5,6];  
a  
a.toString()  
JSON.stringify(a)  
a.toSource()  
a.pop()  
a  
a.unshift(66)  
a  
a.shift()  
a  
a.push(66)  
a  
a.unshift(11)  
a
```

```
a.indexOf(4)  
a.indexOf(999)  
a.reverse()  
a  
a.join(';')  
var b = a.concat([77,88,99])  
b  
b.sort()  
b.sort(function(a,b){return a-b;})  
b.sort((a,b)=>a-b)  
b.slice(2,4)  
b  
b.splice(2,4)  
b
```

# Klassen

- ▶ Umweg über Konstruktor-Funktion nicht mehr notwendig
- ▶ Syntax wie erwartet
- ▶ Konstruktor heißt **constructor()**
- ▶ Statische Methoden: Schlüsselwort **static**

```
class Person {  
  constructor(last, first) {  
    this.last = last;  
    this.first = first;  
  }  
  toString() {  
    return `${this.first} ${this.last}`;  
  }  
}  
  
const hansi = new Person('Huber', 'Hansi');  
console.log(hansi.toString());
```

*It is essentially a syntactic sugar on top of prototype-based programming model in JavaScript. It gives us cleaner syntax compared to complex Function Constructors.*

(<https://medium.com/@naveenkarippai/debunking-fake-classes-in-javascript-78f67a6b5c96>)

**Hansi Huber**



# Vererbung

- ▶ Vererbung ist angelehnt an Java
- ▶ Überschreiben von Methoden wie erwartet

```
class Worker extends Person {  
  constructor(last, first, id) {  
    super(last, first);  
    this.id = id;  
  }  
  toString() {  
    return `${super.toString()} [${this.id}]`;  
  }  
}  
  
const pepi = new Worker('Lehner', 'Pepi', 666);  
console.log(pepi.toString());
```

Pepi Lehner [666]