

**WBE: JAVASCRIPT**

**GRUNDLAGEN**

# ÜBERSICHT

- JavaScript und Node.js
- Grundlegende Sprachelemente
- Objekte und Arrays
- JSON

# ÜBERSICHT

- JavaScript und Node.js
- Grundlegende Sprachelemente
- Objekte und Arrays
- JSON

# JAVASCRIPT

- Dynamisches Typenkonzept
- Objektorientierter und funktionaler Stil möglich
- Mächtige und moderne Sprachkonzepte
- Leistungsfähige Laufzeitumgebungen
- Aber: ein paar Design-Mängel aus den Anfangstagen
- Problem: grundlegende Änderungen nicht möglich

Aus den Slides von letzter Woche:

## JAVASCRIPT

- Veröffentlicht 1995 in Vorversion des Netscape Navigators 2.0
- Unter Zeitdruck entwickelt von Brendan Eich
- Ziel: Scripts um Webseiten dynamischer zu machen
- Zunächst: **LiveScript**, dann **JavaScript** (Marketing)
- **JavaScript und Java haben wenig gemeinsam (!)**

Brendan Eich und Allen Wirfs-Brock haben an der PLDI 2021 (bzw. HOPL IV: History of Programming Languages) einen Vortrag zur Entstehung von JavaScript gehalten:

JavaScript: The First 20 Years

[https://www.pldi21.org/prerecorded\\_hopl.12.html](https://www.pldi21.org/prerecorded_hopl.12.html)

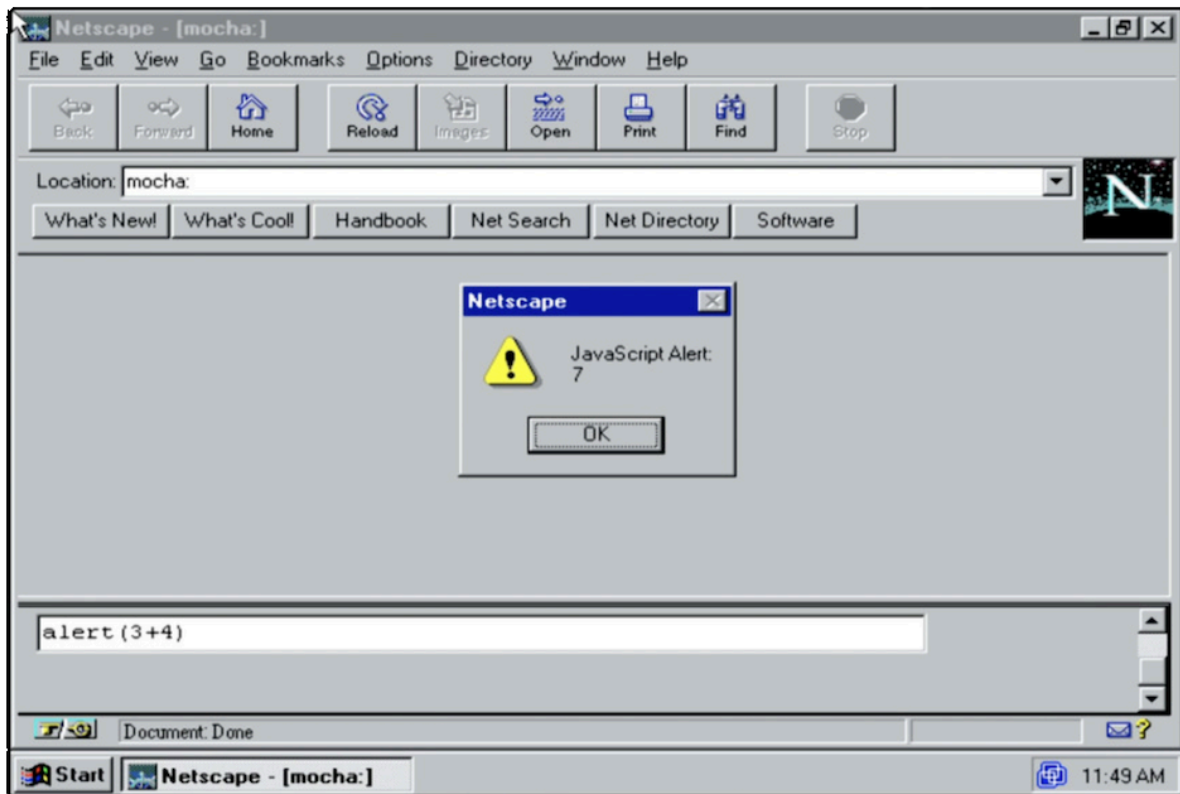
Anfang 1995 wurde Brendan Eich von Netscape eingestellt – ursprünglich war die Absicht, Scheme (ein Lisp-Dialekt) in den Browser einzubauen. Das erwies sich aber schnell als problematisch. Microsoft wollte bereits im Jahr zuvor Netscape übernehmen und war eine starke Konkurrenz. Netscape führte daher Gespräche mit Sun, welche gerade eine Marketing-Kampagne für Java starteten. Es entstand die Idee, Java in den Browser zu integrieren. Netscape kündigte also an, Java für den Browser zu lizenzieren.

Damit war für das Management von Netscape und Sun (oder besser: ein Teil davon) klar, dass allenfalls noch eine kleine Sprache als Alternative für Java im Browser in Frage kommt. Warum überhaupt eine zweite Sprache? Java schien für Anfänger oder Web-Entwickler für einfache Aufgaben als ungeeignet.

Brendan Eich sollte also eine einfache Sprache für den Browser entwickeln. Dazu gab es ein paar Vorgaben: Die Syntax der Sprache sollte ähnlich zu Java und für Anfänger geeignet sein. Und es sollte eine Sprache ohne Klassen sein, damit es keine Konkurrenz für Java ist. Die grosse Frage war, ob Netscape überhaupt in der Lage ist, eine solche Sprache zu entwickeln und bis zur geplanten Beta von Netscape 2 im September des gleichen Jahres in den Browser zu integrieren.

Brendan Eich machte sich also an die Arbeit und entwickelte die erste Version der Sprache unter dem Code-Namen **Mocha** innerhalb von 10 (!) Tagen im Mai 1995. Er orientierte sich an der Sprache Self. Nach den zehn Tagen konnte er einen ersten, rudimentär in eine pre-alpha-Version von Netscape 2 integrierten Prototyp demonstrieren.

Die Sprache wurde später LiveScript und schliesslich JavaScript genannt. Zusammengefasst kann man JavaScript als Mischung von *Scheme* und *Self* in der Syntax von *C* oder *Java* beschreiben.



Leider sind in den Anfängen (wohl aufgrund des Zeitdrucks) verschiedene Design-Fehler gemacht worden, die sich später kaum noch korrigieren liessen. Das Problem ist, dass die Sprache nur erweitert werden kann, grundlegende Änderungen aber nicht möglich sind, da zahllose Websites sonst nicht mehr funktionieren würden.

Man kann sich aber gut behelfen, indem man sich auf die guten Teile von JavaScript beschränkt und problematische Sprachelemente meidet.

# JAVASCRIPT

„JavaScript is ridiculously liberal in what it allows.“

(Eloquent JavaScript)

- Sollte Anfängern den Einstieg erleichtern
- Führt aber leicht zu Problemen
- Aber auch: extrem mächtige Sprache
- Wichtig: Subset und Stil definieren und einhalten

„JavaScript: The Good Parts“ (Douglas Crockford, 2008, O'Reilly)

<https://www.oreilly.com/library/view/javascript-the-good/9780596517748/>

## Speaker notes

„JavaScript: The Good Parts“ ist ein sehr gutes Buch und nicht sehr umfangreich. Leider ist es heute nicht mehr auf dem aktuellen Stand.

Noch ein paar Zitate aus Vorträgen von Douglas Crockford:

*The World's Most Misunderstood Programming Language*

*[...] one of the world's most popular programming languages. Virtually every personal computer in the world has at least one JavaScript interpreter installed on it and in active use.*

*People don't feel a need to learn it before they start using it.*

*Hidden under a huge steaming pile of good intentions and blunders is an elegant, expressive programming language. JavaScript has good parts.*

*It is a language with enormous expressive power. [...] JavaScript is the first lambda language to go mainstream. Deep down, JavaScript has more common with Lisp and Scheme than with Java. It is Lisp in C's clothing.*

# JAVASCRIPT



# STANDARDS

- [ECMAScript](#)
- Versionen
  - ES3: 2000...2010 verbreitete Version
  - ES4: Übung 2008 abgebrochen
  - ES5: 2009, kleineres Update
  - ES6: 2015, umfangreiche Neuerungen
  - ES7, JavaScript 2016
  - dann jährliche Updates
- JavaScript-Alternativen: [TypeScript](#), [ReScript](#), [ClojureScript](#)
- Transpiler: [Babel](#)

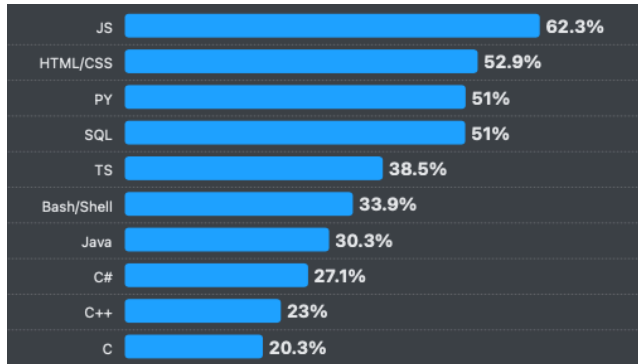
# TypeScript statt JavaScript?

- TypeScript ist eine Erweiterung von JavaScript
- Statisches Typenkonzept: typsichere Programmierung
- Verbesserte Editor-Unterstützung
- Kann schrittweise zu Projekt hinzugefügt werden
- Code etwas umfangreicher und komplizierter
- Compile-Schritt nötig

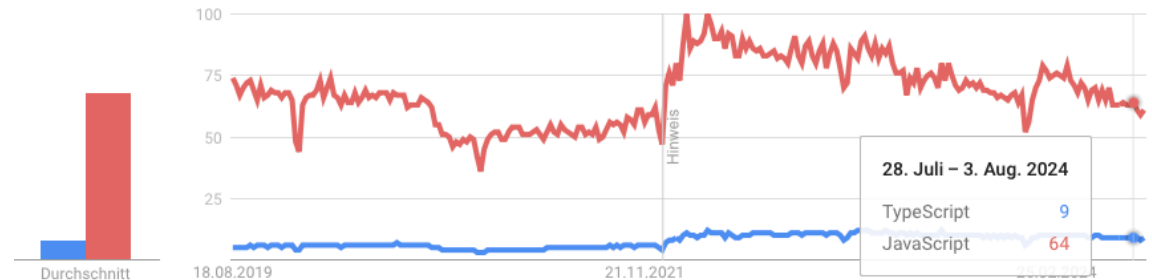
<https://www.typescriptlang.org>

# TypeScript statt JavaScript?

- Für grössere Projekte bietet **TypeScript** viele Vorteile
- Schwerpunkt in WBE: Browser-Sprache **JavaScript**
- JavaScript ist eine gute Basis zum Erlernen von TypeScript
- Doppellektion zu TypeScript später im Semester



StackOverflow Dev Survey 2024



Google Trends

# JAVASCRIPT IM BROWSER

- JavaScript Engines
  - Google Chrome: V8
  - Apple Safari: JavaScriptCore (Nitro)
  - Firefox: Spidermonkey
  - Edge: V8
- Plattformspezifische APIs
  - DOM, Document Object Model
  - Weitere: Cookies, Storage, ...

# JAVASCRIPT OHNE BROWSER: **NODE.JS**

- Asynchrone, ereignisbasierte JavaScript-Laufzeitumgebung
- Grundlage für skalierbare Netzwerk-Anwendungen
- Basiert auf Googles V8 Engine
- Open-Source und plattformübergreifend
- Ryan Dahl 2009

Node.js ist nicht die einzige JavaScript-Laufzeitumgebung ausserhalb von Browsern. Das neuere Deno (<https://deno.land>) basiert ebenfalls auf V8 und wurde vom Urheber von Node.js, Ryan Dahl, mitentwickelt.

Auch *Bun* ist eine solche JavaScript-Laufzeitumgebung, für die im September 2023 Version 1.0 erschienen ist. Im Gegensatz zu Node.js verwendet Bun Apple's WebKit engine. Es war zunächst nur für MacOS und Linux verfügbar, mittlerweile gibt es aber auch einen Build für Windows.

<https://bun.sh>

2014 gab es mit io.js einen Fork von Node.js, da es Unstimmigkeiten über die Abläufe und Verantwortlichkeiten im Node.js-Projekt gab. Nach Gründung einer neutralen Node.js Foundation wurden die Projekte unter dieser Organisation wieder zusammengeführt.

Nebenbei, hier noch ein Slide aus dem früheren Kurs WEB3 zu Node.js:

- Two big things are happening right now in the practice of writing software
- Node.js is at the forefront of both
- First, software is becoming increasingly asynchronous
  - For Big Data, interaction in a Web UI or responding to API requests
- Second, JS has quietly become the worlds standard Virtual Machine
  - Browsers, (NoSQL) databases and back-ends - JS unites them all
  - Using JS for every layer of the application stack is a viable option
  - And a great way to reduce software complexity

# NODE.JS – BEISPIEL

```
1  /* === hello-world.js === */
2  const http = require('http')
3
4  const hostname = '127.0.0.1'
5  const port = 3000
6
7  const server = http.createServer((req, res) => {
8    res.statusCode = 200
9    res.setHeader('Content-Type', 'text/plain')
10   res.end('Hello, World!\n')
11 })
12
13 server.listen(port, hostname, () => {
14   console.log(`Server running at http://${hostname}:${port}/`)
15 })
```

# NODE.JS – EINSATZ

- Script wird mit dem Kommando `node` gestartet
- `node` ohne Argument startet die interaktive **REPL** (REPL = Read Eval Print Loop)

```
$ node hello-world.js  
Server running at http://127.0.0.1:3000/  
# Abbruch mit CTRL-C
```

```
$ node  
Welcome to Node.js v22.7.0.  
Type ".help" for more information.  
>
```

# NODE.JS - REPL

- JavaScript interaktiv
- Auto-Vervollständigung von Funktions- und Objektnamen
- `_` liefert Resultat der letzten Operation
- `.help` gibt Hilfe zu weiteren Kommandos aus

```
> .load hello-world.js  
Server running at http://127.0.0.1:3000/
```

# CONSOLE.LOG

- Ausgabe von Werten auf der Konsole
- Browser: Konsole der Entwicklertools

```
1 let x = 30
2 console.log("the value of x is ", x)
3 // → the value of x is 30
4
5 console.log('my %s has %d ears', 'cat', 2)
6 // → my cat has 2 ears
```

<https://nodejs.org/api/console.html>

## Speaker notes

Das zweite Beispiel zeigt: `console.log` kann auch formatierte Ausgaben à la `printf` erzeugen: Wenn Argumente fehlen, wird einfach die %-Angabe in die Ausgabe übernommen. Zusätzliche Argumente werden an die Ausgabe angehängt.

Mit der Angabe `%o` kann eine Objektrepräsentation ausgegeben werden.

```
> console.log('%o', Number)
<ref *1> [Function: Number] {
  [length]: 1,
  [name]: 'Number',
  [prototype]: [Number: 0] {
    [constructor]: [Circular *1],
    [toExponential]: [Function: toExponential] { [length]: 1, [name]: 'toExponential' },
    ...
  }
}
```

Weitere Möglichkeiten:

| Anweisung   | Bedeutung            |
|---|----------------------|
| <code>console.clear()</code>                                  | Konsole löschen      |
| <code>console.trace()</code>                                  | Stack Trace ausgeben |
| <code>console.time()</code><br><code>console.timeEnd()</code> | Zeit messen          |
| <code>console.error()</code>                                  | auf stderr ausgeben  |

# FRAMEWORKS UND TOOLS

- Node.js ist eine low-level Plattform
- Zahlreiche Frameworks und Tools bauen darauf auf
- Beispiele:
  - [Express](#): Webserver, Nachfolger: [Koa](#)
  - [Socket.io](#): Echtzeitkommunikation
  - [Next.js](#): serverseitiges React Rendering
  - [Webpack](#): JavaScript Bundler
  - u.v.m.

# NPM

- Paketverwaltung für Node.js
- Repository mit > 1 Mio Paketen
- Werkzeuge zum Zugriff auf das Repository: `npm`, `yarn`
- Seit 2020: GitHub (und damit: Microsoft)

<https://www.npmjs.com>

# ÜBERSICHT

- JavaScript und Node.js
- Grundlegende Sprachelemente
- Objekte und Arrays
- JSON

# ZAHLEN

- Zahlentyp in JavaScript: **Number**
- **64 Bit Floating Point** entsprechend IEEE 754  
(wie *double* in Java)
- Enthält alle 32 Bit Ganzzahlen
- Konsequenz: alle Java *int* auch in JavaScript exakt dargestellt
- Weitere Konsequenz: oft Rechenungenauigkeit bei Zahlen mit Nachkommastellen

# ZAHLENLITERALE

```
17           // Ganzzahlliteral
3.14         // Dezimalstellen
2.998e8      // Dezimalpunktverschiebung mal 10 hoch 8
```

## Achtung:

Wie in Java werden Zahlen wie 0.1 nicht exakt dargestellt:

```
0.1          // hexadezimal 3FB999999999999A, entspricht nicht exakt 0.1
0.25         // hexadezimal 3FD0000000000000, entspricht exakt 0.25
```

## Speaker notes

0.25 lässt sich als Summe von Zweierpotenzen darstellen, 0.625 ebenso. Dies gilt aber nicht für 0.1, es kann mit Zweierpotenzen nur angenähert werden.

Nach IEEE 754 werden die 64 Bit folgendermassen verwendet:

- 52 Bit Mantisse
- 11 Bit Exponent (Verschiebung des Punktes in der Binärzahl)
- 1 Bit Vorzeichen

<https://bartaz.github.io/ieee754-visualization/>

# AUSDRÜCKE

- Rechenoperatoren wie in Java
- Spezielle “Zahlen”: `Infinity`, `-Infinity`, `NaN`

```
100 + 4 * 11          // 144
(100 + 4) * 11        // 1144
314 % 100              // 14

1/0                    // Infinity
Infinity + 1           // Infinity
0/0                    // NaN
```

## Speaker notes

NaN steht für “not a number”

# BIGINT

- Mit ES2020 eingeführt
- Literale mit anhängtem `n`
- Keine automatische Typumwandlung von/zu Number

```
1n + 2n           // 3n
2n ** 128n        // 340282366920938463463374607431768211456n

BigInt(1)         // 1n
Number(1n)        // 1

1n + 1            // TypeError: Cannot mix BigInt and ...
```

# typeof

- Operator, der Typ-String seines Operanden liefert
- Mit Klammern kein Abstand nötig

```
typeof 12           // 'number'  
typeof(12)          // 'number'  
typeof 2n           // 'bigint'  
typeof Infinity     // 'number'  
typeof NaN          // 'number'  !!  
typeof 'number'     // 'string'
```

[MDN Docs](#)

## Speaker notes

Operatoren sind also nicht generell einzelne Symbole. Sie können auch aus ganzen Wörtern bestehen.

# STRINGS

```
`Down on the sea`  
"Lie on the ocean"  
'Float on the ocean'
```

- Sequenz von 16-Bit-Unicode-Zeichen
- Kein spezieller char-Typ
- Strings mit "\"" und "'" verhalten sich gleich
- Escape-Sequenzen: `\n` für LF, `\\` für ein `\`-Zeichen u.a.
- String-Verkettung mit dem `+`-Operator:

```
`con` + "cat" + 'enate'
```

## ESCAPE-SEQUENZEN

| Sequenz                 | Bedeutung            |
|-------------------------|----------------------|
| \\                      | backslash            |
| \n                      | new line             |
| \r                      | carriage return      |
| \v                      | vertical tab         |
| \t                      | tab                  |
| \b                      | backspace            |
| \f                      | form feed            |
| \uXXXX (4 Hexziffern)   | UTF-16 code unit     |
| \xXX (2 Hexziffern)     | ISO-8859-1 character |
| \XXX (1-3 Oktalziffern) | ISO-8859-1 character |

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

# TEMPLATE-STRINGS

- Strings mit `'...'` sind Template-Strings
- Ein `\` wird als `\` interpretiert  
(Ausnahme: vor `'`, `$` und Leerzeichen)
- Kann Zeilenwechsel enthalten
- **String-Interpolation**: Werte in String einfügen

```
`half of 100 is ${100 / 2}`      // 'half of 100 is 50'
`erste Zeile
zweite Zeile`                   // 'erste Zeile\nzweite Zeile'
```

## Speaker notes

Einige `-Zeichen auf diesen Slides sind ähnlich aussehende UTF8-Zeichen. Copy&Paste funktioniert in diesem Fall nicht. Grund: Die Slides sind in Markdown geschrieben und da hat das ` eine spezielle Bedeutung.

## Die Aussage

- Ein `\` wird als `\\` interpretiert

bedeutet, dass zum Beispiel ein `\n` nicht durch einen Zeilenwechsel ersetzt wird, sondern als `\n` im String bleibt. Das heisst die meisten Escape-Sequenzen werden nicht interpretiert sondern bleiben wie geschrieben im String. Ausgenommen die auf der Folie aufgeführten Ausnahmen.

```
> `\"`  
``'  
  
> `${2**10}`  
'1024'  
  
> `\"`${2**10}``  
'`${2**10}`'
```

# LOGISCHE AUSDRÜCKE

```
typeof true           // 'boolean'  
3 > 4                 // false  
1 < 2 && 2 < 3         // true  
4 >= 5 || !(1 == 2)   // true  
"ab" == "a" + "b"     // true
```

- Typ **boolean** mit den beiden Werten `true` und `false`
- Vergleiche liefern Ergebnis vom Typ `boolean`
- Logische Operatoren entsprechen denen in C und Java
- Strings sind Werte: Vergleich mit `==` kein Problem

## Speaker notes

String ist in JavaScript ein Wertetyp (value type) und kein Referenztyp. Während Strings in Java mit der equals-Methode verglichen werden, ist in JavaScript ein Vergleich mit `==` problemlos möglich.

Wichtig: Hier ist mit String der primitive String-Typ gemeint. Mit String-Objekten verhält es sich anders:

```
> var a = new String("ab")
> var b = new String("ab")
> a == b
false
> a.valueOf()
'ab'
> a.valueOf() == b.valueOf()
true
```

Ein Wert, der nicht gleich zu sich selbst ist, ist übrigens NaN:

```
> NaN == NaN
false
```

# SPEZIELLE WERTE

```
> null  
null
```

```
> undefined  
undefined
```

```
> let wert  
> wert  
undefined
```

- Zwei spezielle “Werte”: `null` und `undefined`
- Stehen für: Abwesenheit eines konkreten Werts
- Nicht vorhandene Objektreferenz eher `null`
- Eigentlich aber austauschbar

# DYNAMISCHES TYPENKONZEPT

- Typen werden bei Bedarf konvertiert
- Dies kann zu unerwarteten Ergebnissen führen
- Problematisch: Überladener Operator `+` kombiniert mit Typumwandlung

```
> 8 * null
0
```

```
> "5" - 1
```

```
4
```

```
> "5" + 1
```

```
"51"
```

```
> null == undefined
```

```
true
```

```
> [!0, !0n, !"", !false, !undefined, !null, !NaN ]
[ true, true, true, true, true, true, true ]
```

Falsy values in JavaScript

# VERGLEICH MIT `==` ODER `===`

- `==`: Vergleich mit automatischer Typkonvertierung
- `===`: Vergleich ohne Typkonvertierung (oft vorzuziehen)
- Ebenso: `!=` und `!==`

```
> 12 == "12"  
true  
> 12 === "12"  
false  
> 12 != "12"  
false  
> 12 !== "12"  
true
```

```
> undefined == null  
true  
> undefined === null  
false  
> "" == false  
true  
> "" === false  
false
```

# LOGISCHE OPERATOREN

- Bereits eingeführt: `&&`, `||`
- Wenn das Ergebnis feststeht, werden weitere Operanden nicht mehr ausgewertet (**short-circuiting**)
- Null coalescing operator `??`: liefert nur für `null` oder `undefined` den zweiten Operanden

```
null    || 'user'    // 'user'
'Agnes' || 'user'    // 'Agnes'
```

```
' '      || 'user'    // 'user'
' '      ?? 'user'    // ' '
```

```
1 > 2 ? 'A' : 'B'    // 'B'
```

// ausserdem gilt:

```
a && b    ≡    a ? b : a
a || b    ≡    a ? a : b
!a        ≡    a ? false : true
```

## Short-Circuiting

- `&&` liefert den ersten Operanden, der als *false* interpretiert werden kann, sonst den letzten Operanden
- `||` liefert den ersten Operanden, der als *true* interpretiert werden kann, sonst den letzten Operanden
- Der `||`-Operator wird häufig benutzt, um Defaults zuzuweisen, wenn der eigentliche Wert undefiniert ist:

```
"OK" || 'default string'    // → "OK"  
null || 'default string'    // → "default string"  
"" || 'default string'      // → "default string"
```

Problem: In der letzten Zeile sieht man, dass auch `""` durch den Default ersetzt wird, weil als *false* interpretierbar. `""` kann aber durchaus ein gewünschter Wert sein.

Seit ECMAScript 2020 gibt es daher eine *offizielle* Lösung, die ohne den Trick mit dem logischen Oder auskommt, den Nullish coalescing operator (`??`):

```
"OK" ?? 'default string'    // → "OK"  
"" ?? 'default string'      // → ""  
null ?? 'default string'    // → "default string"
```

Nur im Falle von `null` oder `undefined` wird der Default-Wert verwendet.

<https://2ality.com/2019/08/nullish-coalescing.html>

Die logischen Operatoren können auch mit dem ternären Operator `?:` ausgedrückt werden (bzw. können als „syntactic sugar“ für die entsprechenden Ausdrücken mit dem ternären Operator angesehen werden):

```
a1 && a2      // entspricht: a1 ? a2 : a1
a1 || a2      // entspricht: a1 ? a1 : a2
!a1           // entspricht: a1 ? false : true
```

# PROGRAMMSTRUKTUR

Ähnlich wie in anderen Sprachen:

- Ausdrücke und Anweisungen
- Variablen, erlaubte Namen, Umgebung
- Bedingte Anweisungen: `if...else`, `switch...case`
- Schleifen: `while`, `do...while`, `for`
- Kommentare: `/*...*/`, `//...`

Mehr in den Lecture Notes oder Kapitel 1 und 2 von  
<https://eloquentjavascript.net>

## AUSDRUCK

- Stück Code, das einen Wert erzeugt
  - einzelner Wert (Literal) oder Variable
  - Funktionsaufruf, der Wert liefert
  - Kombination von Ausdrücken mit Operatoren und Klammern
- Beispiele:

12

m - 1

p \* (q + 10)

fertig ? 10 : 0

Math.sin(0.5)

# ANWEISUNG

- Aufforderung zu einer Aktivität
  - Zuweisung
  - Kontrollstruktur (Verzweigung, Wiederholung)
  - Funktionsaufruf mit Seiteneffekt
- Fakultativ mit Semikolon abgeschlossen
- Beispiele:

```
let a = 12
const square = (n) => n * n
console.log("fertig")
```

Theoretisch kann auch ein Ausdruck als Anweisung verwendet werden. Wenn dieser keine Seiteneffekte enthält, ist dies aber kaum sinnvoll:

```
15
!false
```

## ERLAUBTE NAMEN

- Buchstaben, Ziffern, \_ und \$
- Ziffer darf aber nicht am Anfang stehen
- Keine Schlüsselwörter wie case, class, if, while, ...

Namenskonventionen:

- Namen aus mehreren Wörtern im CamelCase-Stil
- Variablen und Funktionen mit Kleinbuchstaben beginnen
- Klassen und Konstruktorfunktionen mit grossem Anfangsbuchstaben

```
fuzzyLittleTurtle    // Variable oder Funktion  
FuzzyLittleTurtle    // Klasse oder Konstruktor
```

# UMGEBUNG

- Auch Funktionen und Objekte können an Variablen (oder Konstanten) gebunden werden
- **Umgebung**: Menge der Bindungen zu einem Zeitpunkt
- Beim Programmstart existieren bereits zahlreiche Bindungen
- Je nach eingesetztem Runtime-System (Browser, Node.js) sind unterschiedliche Bindungen vordefiniert
- Beispiel: `console`, ein Objekt mit einer Methode `log`

Das Objekt *console* existiert sowohl im Browser als auch unter Node.js. Das Objekt *window* ist dagegen nur im Browser vordefiniert.

# KONTROLLSTRUKTUREN

- Vergleichbar mit C/Java
- Verzweigungen: `if`, `switch`
- Schleifen: `while`, `do...while`, `for`
- Spezielle Varianten der `for`-Schleife

```
for (let i=1; i<50; i*=2) {  
  console.log(i)  
}  
// → 1, → 2, → 4, → 8, → 16, → 32
```

- In JavaScript funktionieren `if` und `switch` sowie der ternäre Operator praktisch gleich wie in C und Java
- Für Schleifen gilt wie in C und Java:
  - Mit `break` kann die Schleife direkt verlassen werden
  - Mit `continue` wird direkt die nächste Iteration begonnen

Bei Bedarf finden Sie hier weitere Informationen:

<https://www.w3schools.com/js/default.asp>

## KOMMENTARE UND CODE-NOTATION

- Kommentare mit `//...` und `/* ... */` wie in C/Java
- Konsistentes Einrücken von Code entsprechend der üblichen Konventionen ist zwingend für die Lesbarkeit

```
if (false != true) {  
    console.log("That makes sense.")  
    if (1 < 2) {  
        console.log("No surprise there.")  
    }  
}
```

# VARIABLENBINDUNG

```
1 let width = 10
2 console.log(width * width)           /* → 100 */
3
4 let answer = true, next = false
5 let novalue
6 console.log(novalue)                 /* → undefined */
```

- Keine Typangabe, dynamische Typzuordnung
- Im gleichen Gültigkeitsbereich (s. später) kann eine Variable nicht erneut definiert werden
- Alternativen zur Variablenbindung: `var` und `const` (s. später)

## Speaker notes

Mit `var` und `let` werden Variablen eingeführt, mit `const` Konstanten. Der Wert einer Variablen kann jederzeit durch eine Zuweisung geändert werden.

Auch wenn hier zunächst `let` eingeführt wird: Werte, die nicht geändert werden müssen, sollten immer als Konstanten mit `const` definiert werden.

JavaScript verwendet ein dynamisches Typenkonzept. Variablen sind nicht fest mit einem bestimmten Typ verbunden. Sie können zu einem Zeitpunkt eine Zahl und zu einem anderen Zeitpunkt einen String oder eine Referenz auf ein Objekt enthalten. Werte sind dagegen von bestimmten Typen:

```
> typeof 3  
'number'  
  
> typeof 'number'  
'string'
```

Für die Zuweisung stehen die üblichen Operatoren zur Verfügung:

```
let height = 10  
height = 20  
height += 5  
height++
```

Hinweis: Der Inkrement- und Dekrement-Operator führt leicht zu Programmierfehlern. Aus Swift wurde ++ und -- daher wieder entfernt. Also eher vermeiden. Ausnahme: zur Aktualisierung der Schleifenvariablen in einer for-Schleife.

# SEMIKOLON

- Semikolon am Ende von Anweisungen weglassen??
- Trend, Semikolon wegzulassen, wo es nicht nötig ist
- Diverse Argumente für und gegen diesen Stil

## JavaScript Standard Style:

Zahlreiche JavaScript Stilregeln, diverse Tools, von vielen Projekten unterstützt, Sammlung von Regeln

u.a. zum Thema [Semicolons](#):

„No semicolons.”

## Speaker notes

- Entscheidung während WBE-Vorbereitung:  
**Keine Semikolons im WBE-JavaScript-Code**
- Zumindest dort, wo sie nicht nötig sind
- Nötig u.a. zwischen mehreren Anweisungen auf einer Zeile
- Die obigen Beispiele daher noch einmal:

In der ersten Version dieser Folien stand:

*Auch wenn das abschliessende Semikolon fakultativ ist, sollte es unbedingt am Ende jeder Anweisung verwendet werden.*

Nachdem ich auf immer mehr JavaScript-Projekte und -Pakete gestossen bin, welche neu auf Semikolons verzichten, habe ich meine Meinung geändert. Tatsächlich scheint mir JavaScript-Code ohne Semikolons etwas aufgeräumter auszusehen. Zumal Anweisungen mit Aufrufen von Callbacks und lange `then . . . catch`-Ketten oft über mehrere Zeilen gehen und vielen Entwicklern nicht ganz klar ist, wo genau Semikolons stehen müssen.

Der Stil ohne Semikolons entspricht übrigens dem Stil anderer Sprachen wie Swift, Kotlin oder Python.

Hier noch ein Beispiel, das etwas später im Semester angesehen wird:

```
promise.then(function(data) {  
    console.log('success: ' + data)  
}) // <== hier darf kein Semikolon stehen  
  
    .catch(function(data) {  
        console.log('fail: ' + data)  
    }) // <== hier würde ein Semikolon stehen
```

# FUNKTIONEN

```
1 const square = function (x) {  
2   return x * x  
3 }  
4  
5 console.log(square(12))      // → 144
```

- Funktion kann zugewiesen werden
- Schlüsselwort `function`
- Parameterliste, Rückgabewert

# FUNKTIONEN

```
1 // Parameterliste und Rückgabewert
2 // lokale Variablen (let nicht vergessen)
3
4 const power = function (base, exponent) {
5     let result = 1
6     for (let count = 0; count < exponent; count++) {
7         result *= base
8     }
9     return result
10 }
11
12 console.log(power(2, 10))      // → 1024
```

# FUNKTIONEN

```
1 const square1 = (x) => { return x * x }  
2 const square2 = x => x * x
```

- Weitere Möglichkeit, Funktionen einzuführen
- Genau ein Parameter: Parameterliste muss nicht geklammert werden
- Nur ein Ausdruck: `return` und Block-Klammern können entfallen
- Möglichkeit, einfache Funktionen kompakt zu schreiben

## Speaker notes

### Engl.: Arrow Functions

Arrow Functions wurden mit ES6 eingeführt.

Es gibt einen kleinen Unterschied zwischen Arrow Functions und mit `function` eingeführten Funktionen. Dieser betrifft u.a. das `this`-Schlüsselwort und wird in einer späteren Lektion angesehen.

# ÜBERSICHT

- JavaScript und Node.js
- Grundlegende Sprachelemente
- Objekte und Arrays
- JSON

# OBJEKTE UND ARRAYS

- **Objekte**: Werte zu Einheiten zusammenfassen
- **Arrays**: Objekte mit speziellen Eigenschaften

| Was             | Objekt                  | Array               |
|-----------------|-------------------------|---------------------|
| Art             | Attribut-Wert-Paare     | Sequenz von Werten  |
| Literalnotation | werte = { a: 1, b: 2 }  | liste = [ 1, 2, 3 ] |
| Ohne Inhalt     | werte = { }             | liste = [ ]         |
| Elementzugriff  | werte["a"] oder werte.a | liste[0]            |

# OBJEKTITERALE

```
1 let person = {  
2   name: "John Baker",  
3   age: 23,  
4   "exam results": [5.5, 5.0, 5.0, 6.0, 4.5]  
5 }
```

- Sammlung von Attributen und Werten
- Attributname und Wert durch Doppelpunkt getrennt
- Attribut-Wert-Paare durch Kommas getrennt
- Attributname als String, wenn es kein gültiger Name ist

## Speaker notes

Objektliterale werden mit geschweiften Klammern `{ . . . }` geschrieben. Geschweifte Klammern haben also zwei Bedeutungen in JavaScript: einerseits umfassen sie Anweisungsblöcke, andererseits bilden sie Objekte.

Attributnamen *können* als String geschrieben werden. Sie *müssen* als String geschrieben werden, wenn es keine gültigen JavaScript-Namen sind.

Als Attributwerte sind alle Datentypen zulässig, Strings, Zahlen, Wahrheitswerte, aber auch Arrays (wie im Beispiel) oder Objekte. Auch Funktionen können Attributwerte sein. In diesem Fall spricht man von *Methoden*.

Nach dem letzten Attribut-Wert-Paar muss kein Komma stehen. Man kann dieses aber einfügen, um zu vermeiden, dass beim Einfügen eines weiteren Attributs ins Objektliteral das Komma vor dem neuen Attribut vergessen geht:

```
let person = {  
  name: "John Baker",  
  age: 23,  
  "exam results": [5.5, 5.0, 5.0, 6.0, 4.5],  
  /* Objekt wird ggf. noch ergänzt */  
}
```

# ZUGRIFF AUF ATTRIBUTE

```
1 let person = {  
2   name: "John Baker",  
3   age: 23,  
4   "exam results": [5.5, 5.0, 5.0, 6.0, 4.5]  
5 }  
6  
7 console.log(person.name)      /* → John Baker */  
8 console.log(person["age"])    /* → 23          */  
9 console.log(person["exam"])   /* → undefined  */
```

- Punkt- oder Klammernotation zum Zugriff
- Punktnotation: Attribut muss gültiger Name sein
- Zugriff auf nicht vorhandenes Attribut liefert `undefined`

## Speaker notes

Wenn die Notation mit eckigen Klammern verwendet wird, kann ein beliebiger String oder eine Zahl als Attribut verwendet werden. In diesem Fall kann auch ein Ausdruck eingesetzt werden, der ausgewertet wird und das Attribut liefert.

`person.x` greift auf das Attribut "x" von `person` zu, `person[x]` wertet zunächst `x` aus und nimmt das Ergebnis als Attributname.

# OPTIONAL CHAINING

```
1 const adventurer = {  
2   name: 'Alice',  
3   cat: { name: 'Dinah' }  
4 }  
5  
6 console.log(adventurer.dog.name)           /* → TypeError */  
7 console.log(adventurer.dog && adventurer.dog.name) /* → undefined */  
8 console.log(adventurer.dog?.name)          /* → undefined */
```

- Eingeführt mit ECMAScript 2020
- Verschiedene weitere Möglichkeiten  
(s. [Optional Chaining](#))

# ATTRIBUTE HINZUFÜGEN / ENTFERNEN

- Objekte sind dynamische Datenstrukturen
- Sie können jederzeit erweitert werden
- Mit `delete` kann ein Attribut entfernt werden
- Mit `in` kann überprüft werden, ob ein Attribut existiert

```
> let obj = { message: "not yet implemented", tasks: 3 }
> obj.ready = false

> obj
{ message: 'not yet implemented', tasks: 3, ready: false }

> delete obj.message
> "message" in obj
false
```

## Speaker notes

```
> let obj = { message: "ready", ready: true, tasks: 3 }
> delete obj.message
> obj.tasks = undefined
> obj
{ ready: true, tasks: undefined }
> "message" in obj
false
> "tasks" in obj
true
```

Es ist also ein Unterschied, ob ein Attribut auf `undefined` gesetzt oder gelöscht wird. Im ersten Fall ist es immer noch vorhanden, im zweiten Fall nicht mehr.

Noch ein kleiner Ausblick: `attr in obj` liefert auch `true`, wenn `attr` geerbt wird, das heisst irgendwo in der Prototypenkette von `obj` vorkommt. Das wird zwar erst beim Thema *Prototypen von Objekten* behandelt, aber wenn Sie bereits einen Blick darauf werfen wollen, sehen Sie sich dieses Beispiel an:

```
> let protoObj = {attr:15}
> let obj = Object.create(protoObj)
> obj
{}
> obj.attr
15
> "attr" in obj
true
```

# METHODEN

- Attribute, deren Werte Funktionen sind, werden **Methoden** genannt
- Sie werden über das Objekt aufgerufen

```
> let cat = { type: "cat", sayHello: () => "Meow" }
```

```
> cat.sayHello  
[Function: sayHello]
```

```
> cat.sayHello()  
'Meow'
```

# OBJEKT ANALYSIEREN

- Methode `keys` von `Object`
- Liefert Array aller Attributnamen
- Analog liefert `values` alle Werte

```
> let obj = {a: 1, b: 2}
```

```
> Object.keys(obj)  
[ 'a', 'b' ]
```

```
> Object.values(obj)  
[ 1, 2 ]
```

# OBJEKTE ZUSAMMENFÜHREN

- Methode `assign` von `Object`
- Erstes Argument ist das Zielobjekt
- Attribute der weiteren Argumente ins Zielobjekt kopiert
- Referenz auf Ergebnis (erstes Arg.) zurückgegeben

```
> let objectA = {a: 1, b: 2}
```

```
> Object.assign(objectA, {b: 3, c: 4})  
{ a: 1, b: 3, c: 4 }
```

```
> Object.assign(objectA, {m: 10}, {n: 11})  
{ a: 1, b: 3, c: 4, m: 10, n: 11 }
```

## Speaker notes

Durch Kopieren der Attribute in ein leeres Objekt kann eine Top-Level-Kopie (s. später Thema Referenzen) eines Objekts erstellt werden:

```
> let obj = {a: 1, b: 2};  
> let objCopy = Object.assign({}, obj);  
> objCopy  
{ a: 1, b: 2 }  
> objCopy == obj  
false
```

In diesem Fall muss das Ergebnis von `Object.assign` aber zugewiesen werden, da wir noch keine Referenz auf das Zielobjekt haben. `Object.assign` modifiziert also das Objekt, welches als erstes Argument angegeben wird und gibt ausserdem eine Referenz auf dieses Objekt zurück.

Die Methoden `assign`, `keys` und `values` sind Methoden von `Object`, aber nicht von einzelnen Objekten:

```
> let obj = {a:1, b:2}
> obj.values()
Uncaught TypeError: obj.values is not a function
> Object.values(obj)
[ 1, 2 ]
```

Dagegen ist `toString` eine Methode einzelner Objekte (auch wenn die Ausgabe hier nicht besonders aufschlussreich ist):

```
obj.toString()
'[object Object]'
```

Diese Unterscheidung entspricht etwa der Unterscheidung von Klassen- und Instanzenmethoden in Java. In JavaScript wird das mit Prototypen umgesetzt, was wir in einer späteren Lektion genauer ansehen werden.

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object)

# SPREAD-SYNTAX

```
> let objectA = { a: 1, b: 2 }  
> let objectB = { c: 100, d: 200 }  
  
> {...objectA, ...objectB, c: 3}  
{ a: 1, b: 2, c: 3, d: 200 }  
  
> {...objectA}  
{ a: 1, b: 2 }  
  
> {...objectA} == objectA  
false
```

- Inhalte eines Objekts in ein anderes Objekt einfügen
- Spread-Operator ...

# OBJEKTE DESTRUKTURIEREN

```
1 let bar = 87
2 let obj = { foo: 12, bar, baz: 43 }
3
4 let {foo, baz} = obj
5 console.log(foo)                /* → 12 */
```

- Teile aus (möglicherweise grossen) Objekten extrahieren
- Auch in Funktionsparametern möglich (spätere Lektion)

## Speaker notes

Im Beispiel sieht man, dass man in einem Objektliteral anstelle von `bar:` auch abgekürzt `bar` schreiben kann. Es wird dann ein Attribut mit diesem Namen und dem Wert der Variablen eingefügt.

# ARRAYS

- Sequenzen von Werten
- Zugriff über Index (erstes Element hat Index 0)
- Nicht jede Position muss besetzt sein
- Nicht besetzte Positionen liefern `undefined`

```
1 let a = [1, 2, 3]
2 a[10] = 99
3
4 console.log( a )           /* → [ 1, 2, 3, <7 empty items>, 99 ] */
5 console.log( a.length )    /* → 11 */
6 console.log( a[1000] )     /* → undefined */
```

## Speaker notes

Wie in Java und C ist `a[1]` also nicht das erste sondern das zweite Element des Arrays. Oder allgemein: beim Zugriff auf `a[n]` werden vom Beginn des Arrays `n` Elemente übersprungen.

Es kann problemlos auch auf eine Position ausserhalb des bestehenden Arrays zugewiesen werden. Die Länge des Arrays ist um 1 grösser als der grösste Index.

# ARRAYS

- Array-Elemente können von beliebigem Typ sein
- Typen können problemlos gemischt werden
- Hier ist das letzte Element des Arrays eine Funktion:

```
> let data = [41, 3.14, "pi", [1, 2, 3], n => 2*n]  
undefined
```

```
> data[4](3)
```

```
6
```

# ARRAYS

- Arrays sind Objekte mit speziellen Eigenschaften
- Sie haben Attribute und Methoden
- Test auf Array: `Array.isArray()`

```
> let data = [1, 2, 3]
```

```
> typeof(data)  
'object'
```

```
> Array.isArray(data)  
true
```

```
> data.length  
3
```

# ARRAY-METHODEN

- Für Arrays stehen zahlreiche Methoden zur Verfügung
- Zum Beispiel `push` und `pop`

```
> let data = [1, 2, 3]
```

```
> data.push(10)
```

```
4
```

```
> data.push(11, 12)
```

```
6
```

```
> data.pop()
```

```
12
```

```
> data
```

```
[ 1, 2, 3, 10, 11 ]
```

## Speaker notes

push hängt ein oder mehrere Elemente ans Ende eines Arrays an und gibt die Anzahl der Elemente im Array zurück.

pop entfernt das letzte Element aus dem Array und gibt es zurück.

# ARRAY-METHODEN

- `shift`, `unshift`: Einfügen und Entfernen am Array-Anfang
- `indexOf`, `lastIndexOf`: Element im Array finden
- `slice`: Bereich eines Arrays ausschneiden
- `concat`: Arrays zusammenhängen
- `at`: Zugriff auf Index (ECMAScript 2022)

```
> let data = [ 1, 2, 3, 10, 11 ]
```

```
> data.slice(1, 3)  
[ 2, 3 ]
```

```
> data.concat([100, 101])  
[ 1, 2, 3, 10, 11, 100, 101 ]
```

## Speaker notes

Die Methoden `slice` und `concat` erzeugen neue Arrays (s. gleich: Referenz-Datentypen):

```
> data.slice(0)
[ 1, 2, 3, 10, 11 ]
> data.slice(0) == data
false
```

Die neue Methode `at` (ECMAScript 2022) wird von allen Indextypen (Strings, Arrays, typisierte Arrays wie `Uint8Array`) unterstützt und erlaubt den Zugriff über einen bestimmten Index, der auch negativ sein kann (-1 greift auf das letzte, -2 auf das vorletzte Element zu usw.):

```
> [1, 2, 3, 4].at(1)
2
> [1, 2, 3, 4].at(-1)
4
```

Weitere Methoden:

- [https://www.w3schools.com/js/js\\_array\\_methods.asp](https://www.w3schools.com/js/js_array_methods.asp)
- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

# SCHLEIFEN ÜBER ARRAYS

```
1 /* Standard for-Schleife */
2 for (let i = 0; i < myArray.length; i++) {
3     doSomethingWith(myArray[i])
4 }
5
6 /* einfachere Variante für Arrays */
7 for (let entry of myArray) {
8     doSomethingWith(entry)
9 }
```

## Speaker notes

Zum Verarbeiten von Arrays gibt es auch eine Reihe vordefinierter Funktionen höherer Ordnung (s. später).

Übrigens kann man auch über Strings mit einer `for..of`-Schleife iterieren.

# SPREAD-SYNTAX

```
> let parts = ['shoulders', 'knees']  
> ['head', ...parts, 'and', 'toes']  
["head", "shoulders", "knees", "and", "toes"]  
  
> [...parts]  
['shoulders', 'knees']  
  
> [...parts] == parts  
false
```

- Inhalte eines Arrays in ein anderes Array einfügen
- Spread-Operator ...

# ARRAYS DESTRUKTURIEREN

- Mehrere Parameter oder Variablen aus einem Array zuweisen
- Vermeidet das spätere Zugreifen über den Array-Index

```
1 let numbers = [1, 2, 3]
2 let [a, b, c] = numbers
3 console.log(c)
```

/\* → 3 \*/

## Speaker notes

Wenn die Anzahl der Variablen/Werte bei der Zuweisung oder Initialisierung nicht übereinstimmt, werden überzählige Werte ignoriert oder überzählige Variablen auf `undefined` gesetzt.

Das letzte Element auf der linken Seite kann auch ein Rest-Element sein:

```
> let [a, ...b] = [1,2,3,4,5,6,7]
undefined
> a
1
> b
[ 2, 3, 4, 5, 6, 7 ]
```

Das Destrukturieren von Arrays ist oft nützlich. So können auf einfache Weise Variableninhalte vertauscht oder mehrere Werte aus Funktionen zurückgegeben werden:

```
> [a, b] = [1, 2]
> [a, b] = [b, a]
[2, 1]
> const web = () => [404, 'not found']
> web()
[404, 'not found']
> let [code, msg] = web()
```

# ÜBERSICHT

- JavaScript und Node.js
- Grundlegende Sprachelemente
- Objekte und Arrays
- JSON

# JSON

- JavaScript Object Notation
- Daten-Austauschformat, nicht nur für JavaScript
- Orientiert an Notation für JavaScript-Objektliterale

```
> JSON.stringify({ type: "cat", name: "Mimi", age: 3 })  
'{"type":"cat","name":"Mimi","age":3}'
```

```
> JSON.parse('{"type":"cat","name":"Mimi","age":3}')
```

```
{ type: 'cat', name: 'Mimi', age: 3 }
```

<https://www.json.org/json-en.html>

Fragen?



# QUELLEN

- Marijn Haverbeke: Eloquent JavaScript  
<https://eloquentjavascript.net/>

# LESESTOFF

Geeignet zur Ergänzung und Vertiefung

- Kapitel 1 bis 4 von:  
Marijn Haverbeke: Eloquent JavaScript  
<https://eloquentjavascript.net/>

# MEHR ZU JAVASCRIPT

- Axel Rauschmayer: Deep JavaScript  
<https://exploringjs.com/deep-js/toc.html>
- Axel Rauschmayer: JavaScript for impatient programmers  
<https://exploringjs.com/impatient-js/index.html>
- Sandro Turriate: Modern Javascript: Everything you missed...  
<https://turriate.com/articles/modern-javascript-everything-you-missed-over-10-years>



