

WBE: JAVASCRIPT

GRUNDLAGEN

ÜBERSICHT

- JavaScript und Node.js
- Werte, Typen, und Operatoren
- Programmstruktur
- Funktionen

ÜBERSICHT

- JavaScript und Node.js
- Werte, Typen, und Operatoren
- Programmstruktur
- Funktionen

JAVASCRIPT (WH)

- 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 (!)

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

JAVASCRIPT

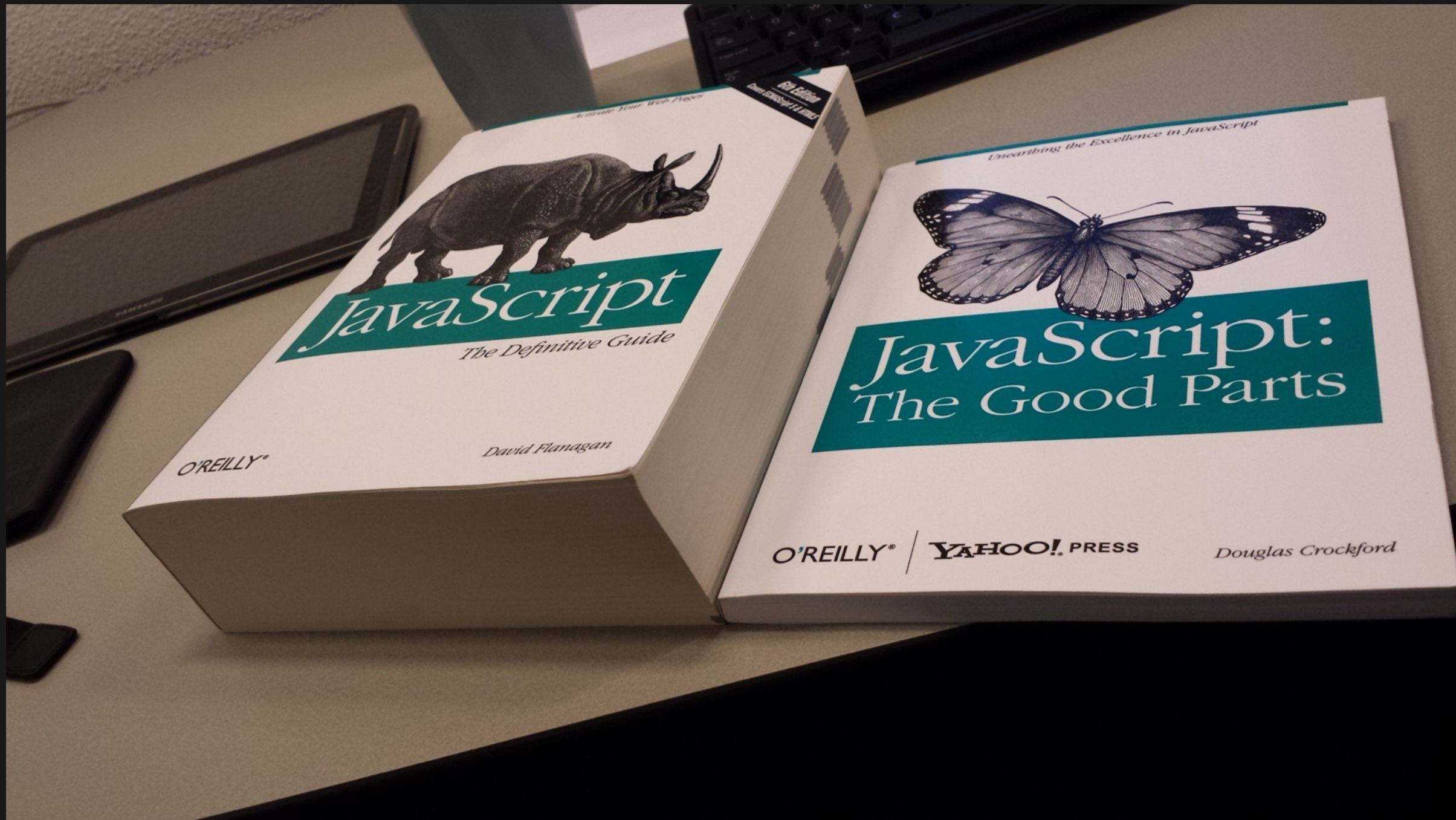
„JavaScript is ridiculously liberal in what it allows.“
(Eloquent JavaScript)

- 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/>

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](#)

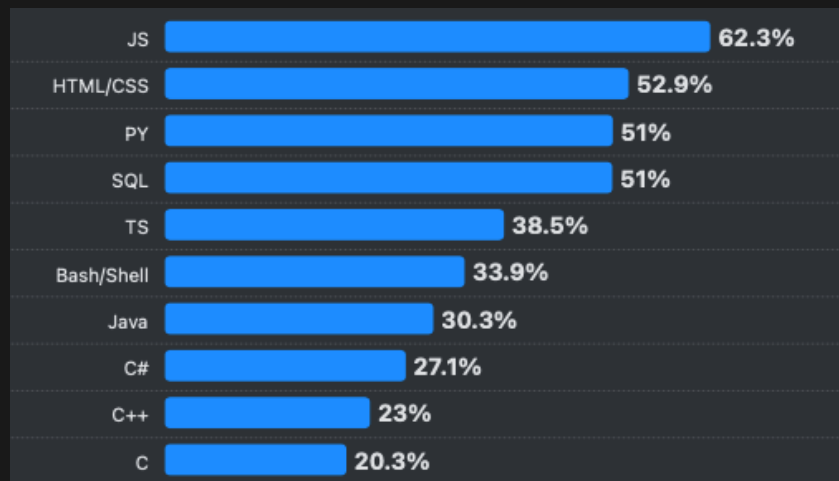
TypeScript statt JavaScript?

- 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

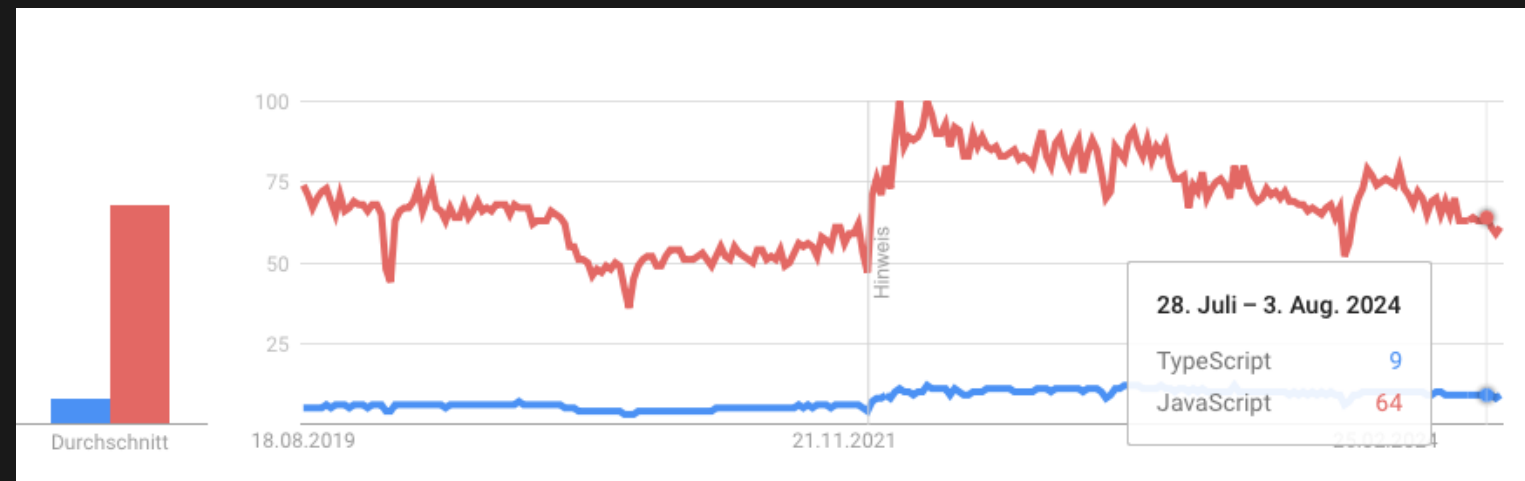
<https://www.typescriptlang.org>

TypeScript statt JavaScript?

- Für grössere Projekte bietet TypeScript viele Vorteile
- Wir werden uns in JavaScript mit der Sprache beschäftigen, welche direkt im Browser läuft: **JavaScript**
- JavaScript ist eine gute Basis zum Erlernen von TypeScript



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 – 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
```


CONSOLE

Weitere Möglichkeiten:

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

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

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
- Werte, Typen, und Operatoren
- Programmstruktur
- Funktionen

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
```

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
```

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

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'
```

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'
```

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

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
```


ÜBERSICHT

- JavaScript und Node.js
- Werte, Typen, und Operatoren
- Programmstruktur
- Funktionen

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");
```

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.”

SEMIKOLON

- 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:

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

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)

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`

KONTROLLSTRUKTUREN

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

```
1 for (let i=1; i<50; i*=2) {  
2   console.log(i)  
3 }  
4 // → 1, → 2, → 4, → 8, → 16, → 32
```

KOMMENTARE UND CODE-NOTATION

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

```
1  if (false != true) {  
2    console.log("That makes sense.")  
3    if (1 < 2) {  
4      console.log("No surprise there.")  
5    }  
6  }
```

ÜBERSICHT

- JavaScript und Node.js
- Werte, Typen, und Operatoren
- Programmstruktur
- Funktionen

FUNKTIONEN

- Strukturierung von Programmen
- Vermeiden von Redundanz
- Namen für Code-Stücke

FUNKTIONSDEFINITION

```
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

FUNKTIONSDEFINITION

- Die Parameterliste kann leer sein
- Die `return`-Anweisung kann fehlen:
es wird `undefined` zurückgegeben
- Beispiel:

```
1 const makeNoise = function () {  
2   console.log("Pling!")  
3 }  
4  
5 makeNoise()           // → Pling!
```

FUNKTIONSDEFINITION

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

- Variablenbindung, lokale Variablen

FUNKTIONEN SIND WERTE

```
1 let launchMissiles = function () {  
2   missileSystem.launch("now")  
3 }  
4 if (safeMode) {  
5   launchMissiles = function () { /* do nothing */ }  
6 }
```

- Funktionen können an Variablen gebunden werden
- Diese Bindung kann jederzeit neu zugewiesen werden (ausser es ist eine Konstante)
- Funktionen auch als Parameter und Rückgabewert möglich:
First Class Citizens

FUNKTIONSDEKLARATION

```
1 console.log(square(5))  
2  
3 function square (x) {  
4     return x * x  
5 }
```

- Alternative Möglichkeit, Funktionen einzuführen
- Funktionsdeklarationen werden zuerst ausgewertet
- Aufruf kann daher vor Deklaration stehen

PFEILNOTATION

```
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

REKURSIVE FUNKTIONEN

```
1  const factorial = function (n) {  
2    if (n <= 1) {  
3      return 1  
4    } else {  
5      return n * factorial(n-1)  
6    }  
7  }  
8  
9  /* oder kurz: */  
10 const factorial = n => (n<=1) ? 1 : n * factorial(n-1)  
11 console.log(factorial(10))    // → 3628800
```

- Funktionen können sich selbst aufrufen
- Abwägen zwischen eleganter und performanter Lösung



Fragen?

QUELLEN

- Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
<https://eloquentjavascript.net/>

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

- Kapitel 1 bis 3 von:
Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
<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>

