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

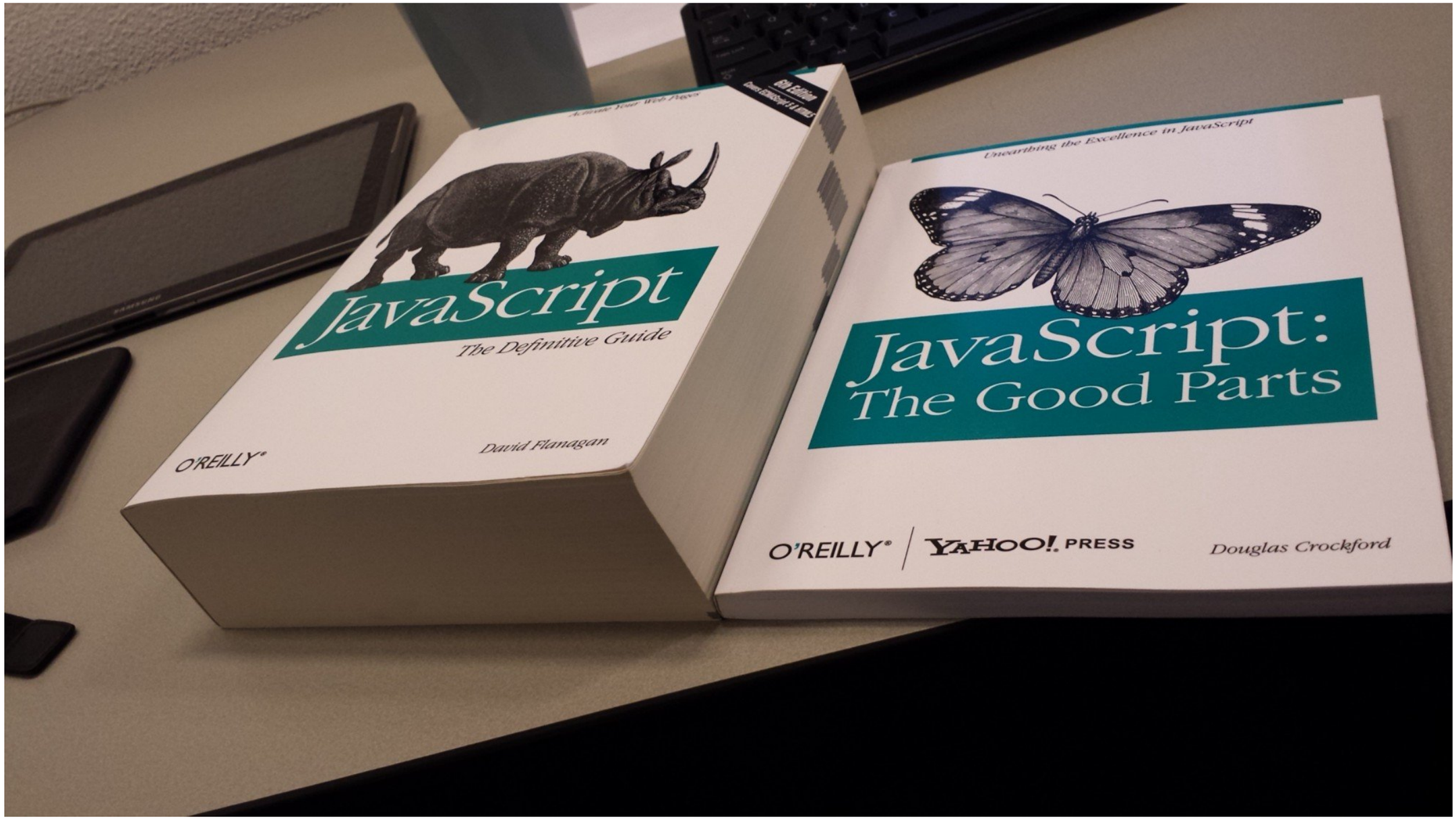
JAVASCRIPT

„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

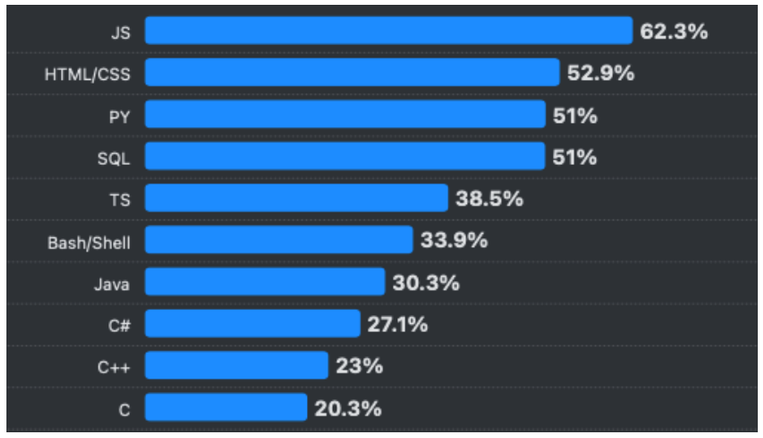
TypeScript statt JavaScript?

- 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

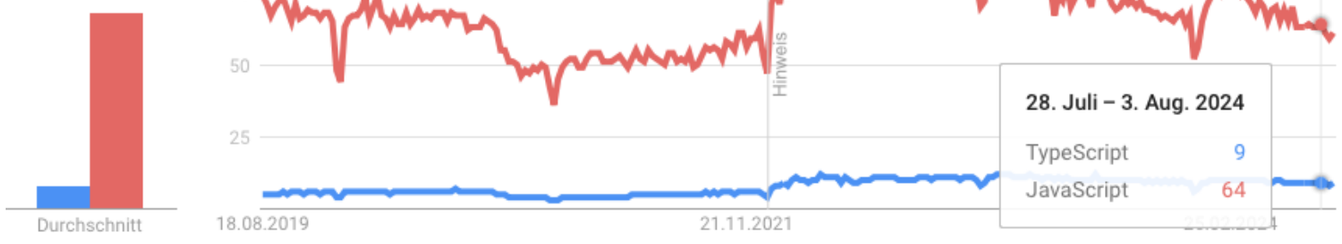
https://www.typescriptlang.org

TypeScript statt JavaScript?

- Für grössere Projekte bietet TypeScript viele Vorteile
- 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 – 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

```
let x = 30
console.log("the value of x is ", x)
// → the value of x is 30

console.log('my %s has %d ears', 'cat', 2)
// → my cat has 2 ears
```

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
- 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
```

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 anghängtem `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
```

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

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)

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.”

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

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]

OBJEKTLITERALE

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

- 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

ZUGRIFF AUF ATTRIBUTE

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

console.log(person.name)       /* → John Baker  */
console.log(person["age"])     /* → 23          */
console.log(person["exam"])    /* → undefined   */
```

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

OPTIONAL CHAINING

```
const adventurer = {
  name: 'Alice',
  cat: { name: 'Dinah' }
}

console.log(adventurer.dog.name)              /* → TypeError */
console.log(adventurer.dog && adventurer.dog.name)  /* → undefined */
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
```

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 }
```

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)

ARRAYS

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

```
let a = [1, 2, 3]
a[10] = 99

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

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 ]
```

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 ]
```

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 }
```

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

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

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

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