

# Javascript Sprachfeatures

## Funktionen und This



<https://iuk.one/1066-1123>

Clemens H. Cap  
ORCID: 0000-0003-3958-6136

Department of Computer Science  
University of **Rostock**  
Rostock, Germany  
[clemens.cap@uni-rostock.de](mailto:clemens.cap@uni-rostock.de)

Version 1



# 1. Funktionen

Ergänzende Bemerkungen zu Funktionen

# 1. Funktionen

2. This und lexical this

3. Iteratoren und Generatoren

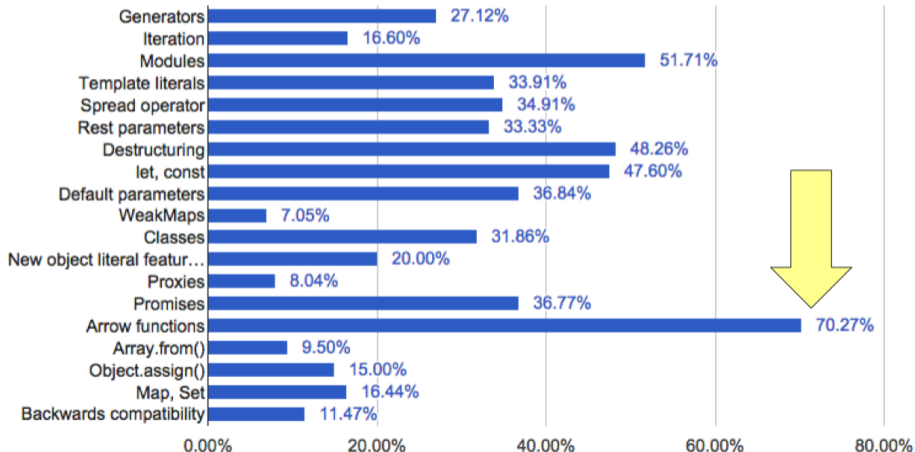
## Arrow Funktionen (1)

### Was sind Arrow Funktionen?

- Syntaktisch kompaktere Notation von Funktionen.
- Wegen funktionaler Orientierung von JS aber sehr hilfreich.
- Gut für filter, map/reduce usw. Anwendungen geeignet.
- Nicht als Konstruktor nutzbar.

## Arrow Funktionen (2)

What are your favorite ES6 features? <http://www.2ality.com/2015/07/favorite-es6-features.html>



## Arrow Funktionen (3)

```
1 odds = evens.map ( v => v + 1 );  
2 pairs = evens.map ( v => ({ even: v, odd: v + 1 }) );  
3 nums = evens.map ( (v, i) => v + i );
```

**Src. 1:** Neue Notation mit Arrow Funktionen.

```
1 function (v)    { return v + 1; }  
2 function (v)    { return { even: v, odd: v + 1 }; }  
3 function (v, i) { return v + i; }
```

**Src. 2:** Bisherige Funktionsdefinitionen.

## Arrow Funktionen (4)

```
1  /* Definitionen */
2  () => { ... }           // kein Parameter
3  x => { ... }           // ein Parameter
4  (x,y) => { ... }      // mehrere Parameter
5
6  /* Rückgaben */
7  x=> { return x*x; }    // Rückgabewert mit return
8  x => x*x;              // Rückgabewert direkt
```

Src. 3: Syntax von Arrow Funktionen.

## Default Werte funktionaler Parameter

```
1 function f (x, y, z) {  
2     if (y === undefined) {y = 7;}  
3     if (z === undefined) {z = 42;}  
4     return x + y + z;  
5 };  
6 f(1) === 50;
```

Src. 4: Bisherige Behandlung von default Werten in Parametern.

```
1 function f (x, y = 7, z = 42) {return x + y + z}  
2 f(1) === 50
```

Src. 5: Neue Behandlung von default Werten in Parametern.

## Variadische Funktionen

```
1 function f (x, y) {  
2   var a = Array.prototype.slice.call(arguments, 2);  
3   return (x + y) * a.length;  
4 };  
5 f(1, 2, "hello", true, 7) === 9;
```

**Src. 6:** Bisherige Behandlung variadischer Funktionen.

```
1 function f (x, y, ...a) {return (x + y) * a.length}  
2 f(1, 2, "hello", true, 7) === 9
```

**Src. 7:** Neue Behandlung variadischer Funktionen



## 2. This und lexical this

1. Funktionen
2. This und lexical this
3. Iteratoren und Generatoren

# Mehrfachrolle von Funktionen

Funktionen in Javascript haben verschiedene **Rollen**

- ① **Konstruktoren** eines Objekts
- ② **Instanz-Funktionen** eines Objekts
- ③ **Objekt-freie Funktionen**

Das “this-Binding” sieht dort jeweils anders aus.

# Problemlage

```
1  function Counter() {this.i = 0; this.start();}
2
3  Counter.prototype.start = function() {
4    window.setInterval(this.add,500); };
5
6  Counter.prototype.add = function() {
7    console.log (this.i); this.i++;};
8
9  new Counter();
10
11  // Problem: In welchem Scope wird die Funktion this.add aufgerufen?
12  // In dem scope von setInterval
13  // Das ist der scope window.setInterval
```

Src. 8: Problemlage

## 2. This und lexical this

# Probleme des This

### "Normales" this

- |  |                                  |
|--|----------------------------------|
| ① Im Kontext eines Konstruktors (new): | Das neue, gerade erzeugte Objekt |
| ② Im Kontext einer Methode:            | Das Objekt selber                |
| ③ Sonst:                               | Globaler Kontext (window)        |

# Beispiel für Probleme dynamischer This Nutzung

### Praktische Probleme dynamischer This Nutzung

- Vermischung der beiden Situationen (was ist das this ?)
- Trennung der beiden Fälle nur im Aufruf erkenntlich, das bedeutet
  - ① nicht am Ort der Verwendung ersichtlich
  - ② nicht durch statische Code-Analyse ersichtlich

```
1  if (expression) {foo();}  
2  else           {bar.foo();}  
3  
4  function foo() { ... /* was bedeutet this-Nutzung hier? */ ... }
```

Src. 9: Beispiel für Probleme dynamischer This Nutzung

# Lösungsansätze für unklare this Anwendung

### Hintergrund:

- Sprache bereits normiert und verwendet.
  - Lösungen müssen Spracherweiterungen oder Pattern sein.
- ① Lösung 1: Self / That - Pattern
  - ② Lösung 2: this-binding
  - ③ Lösung 3: Strict mode
  - ④ Lösung 4: Lexical this
  - ⑤ Lösung 5: Funktionen mit "this" slot

# Self / That Pattern

- Verwende closure Mechanismus
- Nutze statt “this” eine selbstgeschaffene Variable (“self”, “that”)
- Befülle diese Variable rechtzeitig mit gewünschtem Wert.
- Typisch für: Event Handler, timed functions, higher order functions

```
1 function Counter() {this.i = 0; this.start();}
2 Counter.prototype.start = function() {
3   var that = this;
4   setInterval(function() {that.add();},500);
5 };
6 Counter.prototype.add = function() {
7   console.log (this.i); this.i++;};
8 new Counter();
```

Src. 10: Self / That Pattern

# This-Binding

Die zu benutzende Funktion explizit an ein "this" binden.

```
1 function Counter() {this.i = 0; this.start();}
2
3 Counter.prototype.start = function() {
4   var fct = (function() {this.add();}).bind(this);
5   setInterval(fct,500);
6 };
7
8 Counter.prototype.add = function() {
9   console.log (this.i); this.i++;};
10
11 new Counter();
```

Src. 11: This-binding



# Strict Mode (1)

### Strict Mode

Ausführungsmodus, der unklare Semantiken präzisiert und typische Fehler verhindert indem er viele Warnungs-Fälle als Fehler behandelt.

#### Beispiele:

- 1 Nicht durch Konstruktor erzeugtes `this` ist undefined.
- 2 Nicht deklarierte Variable im global scope führen zu Fehler.
- 3 Aliasnamen in Parameter-Listen wie etwa: `function x(p1, p1) {}`; liefern Fehler
- 4 `eval (" ... ")` exportiert keine Variablen-Namen in den aufrufenden scope  
`eval ("var x=2"); alert (x);`

# Strict Mode (2)

### Aktivierung:

- Text `"use strict";`
- Am Anfang eines Script-Files: Ganzes Script
- Am Anfang einer Funktion: Ganze Funktion & darin definierte Funktionen

### Konzept:

- Aktivierung ohne Änderung der Syntax
- `"use strict";` darf als Stringkonstante irgendwo im Text stehen
- Veränderungen der Semantik in einer Form, die weniger Garantien gibt.

# Lexikalisches This

Eine Notation / Konstruktion einführen, die das this immer lexikalisch bindet.

### Lexikalisches this

Das “this” ist immer das “this” des Definitionszeitpunktes und niemals das “this” des Aufrufzeitpunktes.

**Hier:** Arrow-Funktionen nutzen immer ein lexikalisches this.

# Lexikalisches This: Beispiel

```
1  let obj = {
2    x: 27,
3    fct: function () {console.log (this.x);}    // this referenziert obj
4                                                // (Aufrufzeitpunkt)
5  }
6
7  obj.fct ();    // 27
8
9  let obj2 = {
10   x: 27,
11   fct: () => {console.log (this.x);}          // this referenziert window
12                                              // (Definitionszeitpunkt)
13 }
14
15 obj2.fct ();    // undefined
```

Src. 12: Beispiel für lexikalisches this.

## 2. This und lexical this

# Lexikalisches This bei call und apply

```
1  var adder = {
2    base: 1,
3    add : function(a) {var f = v => v + this.base; return f(a);},
4    addCall: function(a) {
5      var f = v => v + this.base;
6      var b = {base : 2};
7      return f.call(b, a); // Überschreibt das this in arrow fct nicht
8    },
9    addCallX: function(a) {
10     var f = adder.add;
11     var b = {base: 200};
12     return f.call (b, a); // Überschreibt das this in function
13   } };
14 console.log(adder.add(1));           // 2
15 console.log(adder.addCall(1));      // Immer noch 2
16 console.log(adder.addCallX(1));     // 201
```

**Src. 13: Beachte:** call und apply können lexical this nicht überschreiben.

# Funktionen mit this slot: Self/That Lösung

**Problem:** Welcher scope ruft die innere Funktion in map auf?

**1. Lösung:** Self/That

```
1 function Prefixer(prefix) {this.prefix = prefix;}
2 Prefixer.prototype.prefixArray = function (arr) {
3   return arr.map(function (x) {return this.prefix + x;}); }; // Problem
4
5 function Prefixer(prefix) {this.prefix = prefix;}
6 Prefixer.prototype.prefixArray = function (arr) {
7   var that = this;
8   return arr.map(function (x) {return that.prefix + x;}); }; // Self/That Closure
```

Src. 14: Self/That Lösung

# Funktionen mit this slot: Slot Lösung

**Problem:** Welcher scope ruft die innere Funktion in map auf?

**2. Lösung:** Funktion definiert einen this Slot

```
1 function Prefixer(prefix) {this.prefix = prefix;}
2 Prefixer.prototype.prefixArray = function (arr) {
3     return arr.map(function (x) {return this.prefix + x;}, this);
4 };
```

Src. 15: this-slot Lösung

**Häufiges Problem** bei higher-order Funktionen (map, forEach etc.)

## 2. This und lexical this

### Preisrätsel, Teil 1

Während der Arbeit an diesen Beispielen fand ich ein spannendes Problem, das in "Minimal Form" so aussieht:

```
1 <html><head></head><body>
2 <script>
3 this.name = "Bond";
4 console.log (this.name);
5 </script>
6 </body></html>
```

Src. 16: Preisrätsel, Teil 1

Frage: Was wird ausgegeben?



## 2. This und lexical this

### Preisrätsel, Teil 2

JS Datei editieren zu

```
1 <html><head></head><body>
2 <script>
3 console.log (this.name);
4 </script>
5 </body></html>
```

Src. 17: Preisrätsel, Teil 2

Reload. Was wird ausgegeben?

Erklärung?

## 2. This und lexical this

### Preisrätsel, Teil 3

JS Datei editieren zu

```
1 <html><head></head><body>
2 <script>
3 this.myname = "Bond";
4 console.log (this.myname);
5 </script>
6 </body></html>
```

Src. 18: Preisrätsel, Teil 3

Reload. Was wird ausgegeben?

Erklärung?

## 2. This und lexical this

### Preisrätsel, Teil 4

JS Datei editieren zu

```
1 <html><head></head><body>
2 <script>
3 console.log (this.myname);
4 </script>
5 </body></html>
```

Src. 19: Preisrätsel, Teil 4

Reload. Was wird ausgegeben?

Erklärung?

## 3. Iteratoren und Generatoren

Funktionen mit Sondereigenschaften und  
Funktionen mit automagischem Status

1. Funktionen

2. This und lexical this

3. Iteratoren und Generatoren

# Enumerables: for ... in Konstruktion

Iteriert über alle enumerable Eigenschaften des Objekts und seiner Prototyp-Kette, die nicht Symbol Schlüssel sind.

```
1  const object = {course: "Web 20", docent: "Cap"};
2
3  for (const key in object) {console.log (key);}           // course \n docent
4  for (const key in object) {console.log (object[key]);}  // Web 20 \n Cap
```

Src. 20: for ... in Konstruktion

- `Object.keys` liefert enumerable Eigenschaften mit String Schlüssel
- `Object.getOwnPropertyNames` liefert Eigenschaften mit String Schlüssel
- `Object.hasOwn` testet auf eigene Eigenschaft (und nicht Prototyp)

# Iterables: Definition

### Iterables:

- sind iterierbare Objekte.
- implementieren die `@@iterator` Methode (selber oder in der Prototyp-Kette),
- die ein Objekt nach dem Iterator-Standard zurückgibt.

**Standard Iterables sind:** String, Array, Map, Set, NodeList

### Iterables als API Parameter

- Map, Set als Konstruktoren
- `Array.from` zur Umwandlung in eine echte Array
- `Promise.all`, `Promise.any` nutzbar als Parameter

# Iterables: Nutzung

### Nutzung in der Sprache selber:

- `for ... of`
- Spread Operator
- `yield*`
- Strukturelle Notation von Arrays

```
1  const arr = ['eins', 'zwei', 'drei'];  
2  
3  font (let ele of arr) { console.log (ele); }
```

Src. 21: for ... of

# Iterator: Definition

**Iterator-Objekte** sind Objekte, die den Iterator-Standard umsetzen.

- Diese implementieren die `next` Methode
- die ein `IteratorResult` zurück gibt.

`IteratorResult`

- Eigenschaft `done`:
  - `false`, wenn `next` noch einen weiteren Wert liefern kann.
  - `true`, wenn Iterationsvorgang beendet ist.
- Eigenschaft `value`
  - der nächste Wert, falls einer vorhanden ist
  - `undefined`, falls `done === true`



# Iterators: Nutzung

```
1  const object = {
2    course:  "Web 20",
3    docent:  "Cap",
4    idx:     -1,    // besser: durch ein weiteres Symbol umsetzen
5    next: () => {
6      object.idx++;
7      if (object.idx == 0) { return {value: object.course, done: false}; }
8      if (object.idx == 1) { return {value: object.docent, done: false}; }
9      return {value:undefined, done:true};
10   },
11   [Symbol.iterator]() {return this;}
12 };
13
14 for (const item of object) { console.log (item);} // Web 20 \n Cap
```

Src. 22:

## 3. Iteratoren und Generatoren

# Live und Static Iterables

Manche Iterables der DOM verändern sich durch Seiteneffekte

```
1 let parent      = document.getElementById("some-id");
2 let nodeList = parent.childNodes;           // ist eine live NodeList
3 console.log(nodeList.length);              // sei 5
4 parent.appendChild(document.createElement("span"));
5 console.log(nodeList.length);              // ist nun 6
```

Src. 23: Live NodeList verändert sich durch Seiteneffekte

```
1 let nodeList = document.querySelectorAll("span"); // ist eine live NodeList
2 console.log(nodeList.length);                  // sei 5
3 document.body.appendChild(document.createElement("span"));
4 console.log(nodeList.length);                  // ist noch immer 5
```

Src. 24: Static NodeList bleibt konstant

# Generator: Definition

### Was ist ein Generator?

- ① Funktion mit kontinuierlich fortgeschriebenem Status.
- ② Unterbrechung des Kontrollflusses bei `yield`
- ③ Rückgabewert ist dann ein spezieller Iterator.
- ④ Definition durch `function*`.
- ⑤ Zustand zum Wiedereinstieg muß nicht mehr separat gespeichert werden.
- ⑥ Auch für virtuell unendliche Objekte geeignet.

## 3. Iteratoren und Generatoren

### Generator: Beispiel (1)

```
1 function* generator() {
2   yield 1;
3   yield "Zwei";
4   yield {Name: "Beispiel"};
5 }
6
7 const gen = generator();
8
9 console.log(gen.next().value); // 1
10 console.log(gen.next());      // Iterator! Daher: {value: "Zwei", done: false}
11 console.log(gen.next().value); // {Name: 'Beispiel'}
12 console.log(gen.next().value); // undefined
```

Src. 25: Beispiel für einen Generator

### Generator: Beispiel (2)

```
1 function* generator() {
2   yield 1;
3   yield "Zwei";
4   yield {Name: "Beispiel"};
5 }
6
7 const gen = generator();
8
9 for (const item of gen) {console.log (item);} // 1 \n "Zwei" \n {Name: 'Beispiel'}
```

Src. 26: Beispiel für einen Generator

## Generator: Beispiel virtuell unendlicher Objekte

```
1  let fibonacci = function* (numbers) {
2      let pre = 0, cur = 1
3      while (numbers-- > 0) {
4          [ pre, cur ] = [ cur, pre + cur ]
5          yield cur
6      }
7  }
8
9  for (let n of fibonacci(1000)) {console.log(n); }
10
11 let numbers = [ ...fibonacci(1000) ];
12
13 let [ n1, n2, n3, ...others ] = fibonacci(1000);
```

Src. 27: Generatoren und Iteratoren

## 3. Iteratoren und Generatoren

# Generator: Beispiel für Range Iterator

```
1 function* range (start, end, step) {  
2     while (start < end) {  
3         yield start  
4         start += step  
5     }  
6 }  
7  
8 for (let i of range(0, 10, 2)) {  
9     console.log(i) // 0, 2, 4, 6, 8  
10 }
```

Src. 28: Generatoren und Iteratoren

# Beispiel: Kombination

### Gemeinsames Beispiel für

- 1 Iterable
- 2 Generator-Funktion
- 3 Spread-Operator

```
1  const iterierbar = {};  
2  
3  iterierbar [Symbol.iterator] = function* () {  
4    yield 1;  
5    yield 2;  
6    yield 3;  
7  };  
8  
9  console.log([...iterierbar]); // Ergibt Array [1, 2, 3]
```

Src. 29: Iterierbar, Generator und Spread5



# Async Iteration

Benötigt, wenn iterierte Objekte erst im Lauf der Zeit entstehen.

Kapselt alle auftretenden Objekte als Promise

Statt `Symbol.iterator` wird `Symbol.asyncIterator` genutzt.

### for await ... of Beispiel (1)

```
1  const delayedResponses = {
2    delays: [500, 1300, 3500],
3
4    wait(delay) {
5      return new Promise(resolve => {setTimeout(resolve, delay); });
6    },
7
8    // implementiere einen asynchronen Iterator im Objekt
9    async *[Symbol.asyncIterator]() {
10     for (const delay of this.delays) {
11       await this.wait(delay);
12       yield `Delayed response for ${delay} milliseconds`;
13     }
14   },
15 };
```

Src. 30: for await ... of Beispiel, Teil 1

### for await ...of Beispiel (2)

```
1  (async() => { // asynchrone lokale Funktion
2    for await (const response of delayedResponses) {
3      console.log(response);
4    }
5  })();
6
7  // Ausgabe:
8  // "Delayed response for 500 milliseconds"
9  // "Delayed response for 1300 milliseconds"
10 // "Delayed response for 3500 milliseconds"
```

Src. 31: for await ... of Beispiel, Teil 2

# Anhang

Übersicht

Programmquellenverzeichnis

Prog

Verzeichnis aller Abbildungen

Abb

Rechtliche Hinweise

§

Zitierweise dieses Dokuments

→

Verzeichnis aller Folien



1	Neue Notation mit Arrow Funktionen.....	5
2	Bisherige Funktionsdefinitionen.....	5
3	Syntax von Arrow Funktionen.....	6
4	Bisherige Behandlung von default Werten in Parametern.....	7
5	Neue Behandlung von default Werten in Parametern.....	7
6	Bisherige Behandlung variadischer Funktionen.....	8
7	Neue Behandlung variadischer Funktionen.....	8
8	Problemlage.....	11
9	Beispiel für Probleme dynamischer This Nutzung.....	13
10	Self / That Pattern.....	15

11	This-binding .....	16
12	Beispiel für lexikalisches this. ....	20
13	<b>Beachte:</b> call und apply können lexical this nicht überschreiben.....	21
14	Self/That Lösung .....	22
15	this-slot Lösung .....	23
16	Preisrätsel, Teil 1 .....	24
17	Preisrätsel, Teil 2 .....	25
18	Preisrätsel, Teil 3 .....	26
19	Preisrätsel, Teil 4 .....	27
20	for ... in Konstruktion .....	29
21	for ... of .....	31

22	.....	33
23	Live NodeList verändert sich durch Seiteneffekte .....	34
24	Static NodeList bleibt konstant .....	34
25	Beispiel für einen Generator .....	36
26	Beispiel für einen Generator .....	37
27	Generatoren und Iteratoren .....	38
28	Generatoren und Iteratoren .....	39
29	Iterierbar, Generator und Spread5 .....	40
30	for await ... of Beispiel, Teil 1 .....	42
31	for await ... of Beispiel, Teil 2 .....	43



1 Bedeutung der Arrow Funktionen.....	4
---------------------------------------	---

# Rechtliche Hinweise (1)

Die hier angebotenen Inhalte unterliegen deutschem Urheberrecht. Inhalte Dritter werden unter Nennung der Rechtsgrundlage ihrer Nutzung und der geltenden Lizenzbestimmungen hier angeführt. Auf das Literaturverzeichnis wird verwiesen. Das **Zitatrecht** in dem für wissenschaftliche Werke üblichen Ausmaß wird beansprucht. Wenn Sie eine Urheberrechtsverletzung erkennen, so bitten wir um Hinweis an den auf der Titelseite genannten Autor und werden entsprechende Inhalte sofort entfernen oder fehlende Rechtsnennungen nachholen. Bei Produkt- und Firmennamen können Markenrechte Dritter bestehen. Verweise und Verlinkungen wurden zum Zeitpunkt des Setzens der Verweise überprüft; sie dienen der Information des Lesers. Der Autor macht sich die Inhalte, auch in der Form, wie sie zum Zeitpunkt des Setzens des Verweises vorlagen, nicht zu eigen und kann diese nicht laufend auf Veränderungen überprüfen.

Alle sonstigen, hier nicht angeführten Inhalte unterliegen dem Copyright des Autors, Prof. Dr. Clemens Cap, ©2020. Wenn Sie diese Inhalte nützlich finden, können Sie darauf verlinken oder sie zitieren. Jede weitere Verbreitung, Speicherung, Vervielfältigung oder sonstige Verwertung außerhalb der Grenzen des Urheberrechts bedarf der schriftlichen Zustimmung des Rechteinhabers. Dieses dient der Sicherung der Aktualität der Inhalte und soll dem Autor auch die Einhaltung urheberrechtlicher Einschränkungen wie beispielsweise **Par 60a UrhG** ermöglichen.

Die Bereitstellung der Inhalte erfolgt hier zur persönlichen Information des Lesers. Eine Haftung für mittelbare oder unmittelbare Schäden wird im maximal rechtlich zulässigen Ausmaß ausgeschlossen, mit Ausnahme von Vorsatz und grober Fahrlässigkeit. Eine Garantie für den Fortbestand dieses Informationsangebots wird nicht gegeben.

Die Anfertigung einer persönlichen Sicherungskopie für die private, nicht gewerbliche und nicht öffentliche Nutzung ist zulässig, sofern sie nicht von einer offensichtlich rechtswidrig hergestellten oder zugänglich gemachten Vorlage stammt.

**Use of Logos and Trademark Symbols:** The logos and trademark symbols used here are the property of their respective owners. The YouTube logo is used according to brand request 2-9753000030769 granted on November 30, 2020. The GitHub logo is property of GitHub Inc. and is used in accordance to the GitHub logo usage conditions <https://github.com/logos> to link to a GitHub account. The Tweedback logo is property of Tweedback GmbH and here is used in accordance to a cooperation contract.

**Disclaimer:** Die sich immer wieder ändernde Rechtslage für digitale Urheberrechte erzeugt ein nicht unerhebliches Risiko bei der Einbindung von Materialien, deren Status nicht oder nur mit unverhältnismäßig hohem Aufwand abzuklären ist. Ebenso kann den Rechteinhabern nicht auf sinnvolle oder einfache Weise ein Honorar zukommen, obwohl deren Leistungen genutzt werden.

Daher binde ich gelegentlich Inhalte nur als Link und nicht durch Framing ein. Lt EuGH Urteil 13.02.2014, C-466/12 ([Pressemitteilung](#), [Blog-Beitrag](#), [Urteilstext](#)). ist das unbedenklich, da die benutzten Links ohne Umgehung technischer Sperren auf im Internet frei verfügbare Inhalte verweisen.

Wenn Sie diese Rechtslage stört, dann setzen Sie sich für eine Modernisierung des völlig veralteten Vergütungs- und Anreizsystems für urheberrechtliche Leistungen ein. Bis dahin klicken Sie bitte auf die angegebenen Links und denken Sie darüber nach, warum wir keine für das digitale Zeitalter sinnvoll angepaßte Vergütungs- und Anreizsysteme digital erbrachter Leistungen haben.

Zu Risiken und Nebenwirkungen fragen Sie Ihren Rechtsanwalt oder Gesetzgeber.

Weitere Hinweise finden Sie im Netz [hier](#) und [hier](#) oder [hier](#).

# Zitierweise dieses Dokuments

Wenn Sie Inhalte aus diesem Werk nutzen oder darauf verweisen wollen, zitieren Sie es bitte wie folgt:

Clemens H. Cap: Javascript Sprachfeatures  
Funktionen und This. Electronic document. <https://iuk.one/1066-1123> 2. 7. 2023.

**Bibtex Information:** <https://iuk.one/1066-1123.bib>

```
@misc{doc:1066-1123,  
  author      = {Clemens H. Cap},  
  title       = {Javascript Sprachfeatures  
Funktionen und This},  
  year        = {2023},  
  month       = {7},  
  howpublished = {Electronic document},  
  url         = {https://iuk.one/1066-1123}  
}
```

## Typographic Information:

Typeset on ?today?

This is pdfTeX, Version 3.14159265-2.6-1.40.21 (TeX Live 2020) kpathsea version 6.3.2




This is pgf in version 3.1.5b

This is preamble-slides.tex myFormat©C.H.Cap

Titelseite .....	1
<b>1. Funktionen</b>	
Arrow Funktionen (1) .....	3
Arrow Funktionen (2) .....	4
Arrow Funktionen (3) .....	5
Arrow Funktionen (4) .....	6
Default Werte funktionaler Parameter .....	7
Variadische Funktionen .....	8
<b>2. This und lexical this</b>	
Mehrfachrolle von Funktionen .....	10
Problemlage .....	11
Probleme des This .....	12
Beispiel für Probleme dynamischer This Nutzung .....	13
Lösungsansätze für unklare this Anwendung .....	14
Self / That Pattern .....	15
This-Binding .....	16
Strict Mode (1) .....	17
Strict Mode (2) .....	18
Lexikalisches This .....	19
Lexikalisches This: Beispiel .....	20
Lexikalisches This bei call und apply .....	21
Funktionen mit this slot: Self/That Lösung .....	22
Funktionen mit this slot: Slot Lösung .....	23
Preisrätsel, Teil 1 .....	24
Preisrätsel, Teil 2 .....	25

Preisrätsel, Teil 3 .....	26
Preisrätsel, Teil 4 .....	27
<b>3. Iteratoren und Generatoren</b>	
Enumerables: for ... in Konstruktion .....	29
Iterables: Definition .....	30
Iterables: Nutzung .....	31
Iterator: Definition .....	32
Iterators: Nutzung .....	33
Live und Static Iterables .....	34
Generator: Definition .....	35
Generator: Beispiel (1) .....	36
Generator: Beispiel (2) .....	37
Generator: Beispiel virtuell unendlicher Objekte .....	38
Generator: Beispiel für Range Iterator .....	39
Beispiel: Kombination .....	40
Async Iteration .....	41
for await ... of Beispiel (1) .....	42
for await ...of Beispiel (2) .....	43

## Legende:

-  Fortsetzungsseite
-  Seite ohne Überschrift
-  Bildseite