

JavaScript

Basics, Praxis und Neuerungen

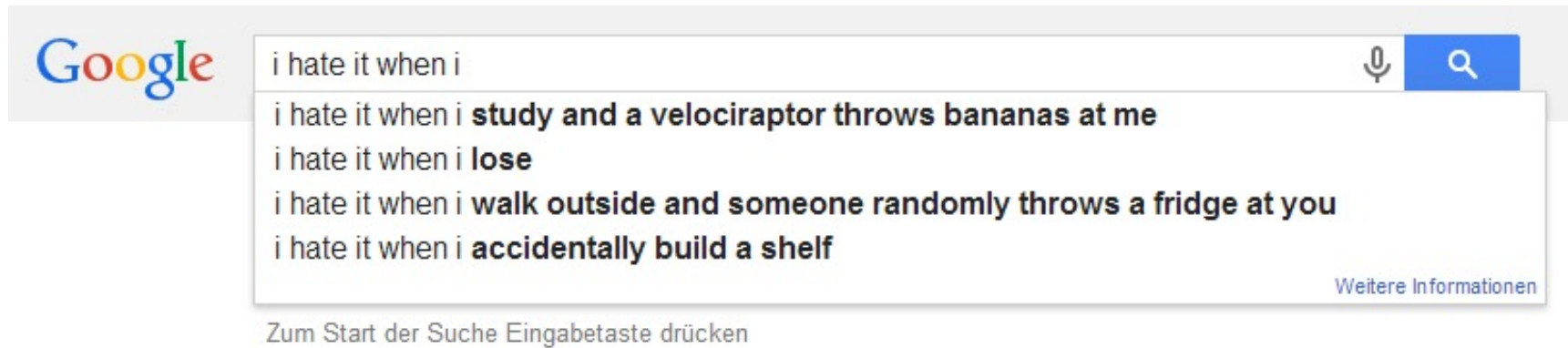
Basics

- Syntax
- Funktionen & Closures
- Klassen (ein wenig)
- Event Loop

JavaScript im Einsatz

- Node.js
- DOM
- Promises
- AJAX
- Events
- Typescript

JavaScript ist überall (im Browser)



- Beispiele
 - > Google-Suche
 - > Facebook-Timeline
 - > Twitter
 - > Browser-Plugins
 - > ...



Motivation

(Ehemals) schlechter Ruf der Sprache

- (Werbe-)Pop-ups, Quelltextverschleierung, Verschleiern von Internetadressen auf die ein Link verweist, ...



Node.js in 2009 (serverseitiges JavaScript)

- Event-basierter Ansatz eine Stärke der Sprache



Heutzutage

- Unterstützung von nahezu jedem Gerät (mit Browser)



APACHE
CORDOVA™

Motivation

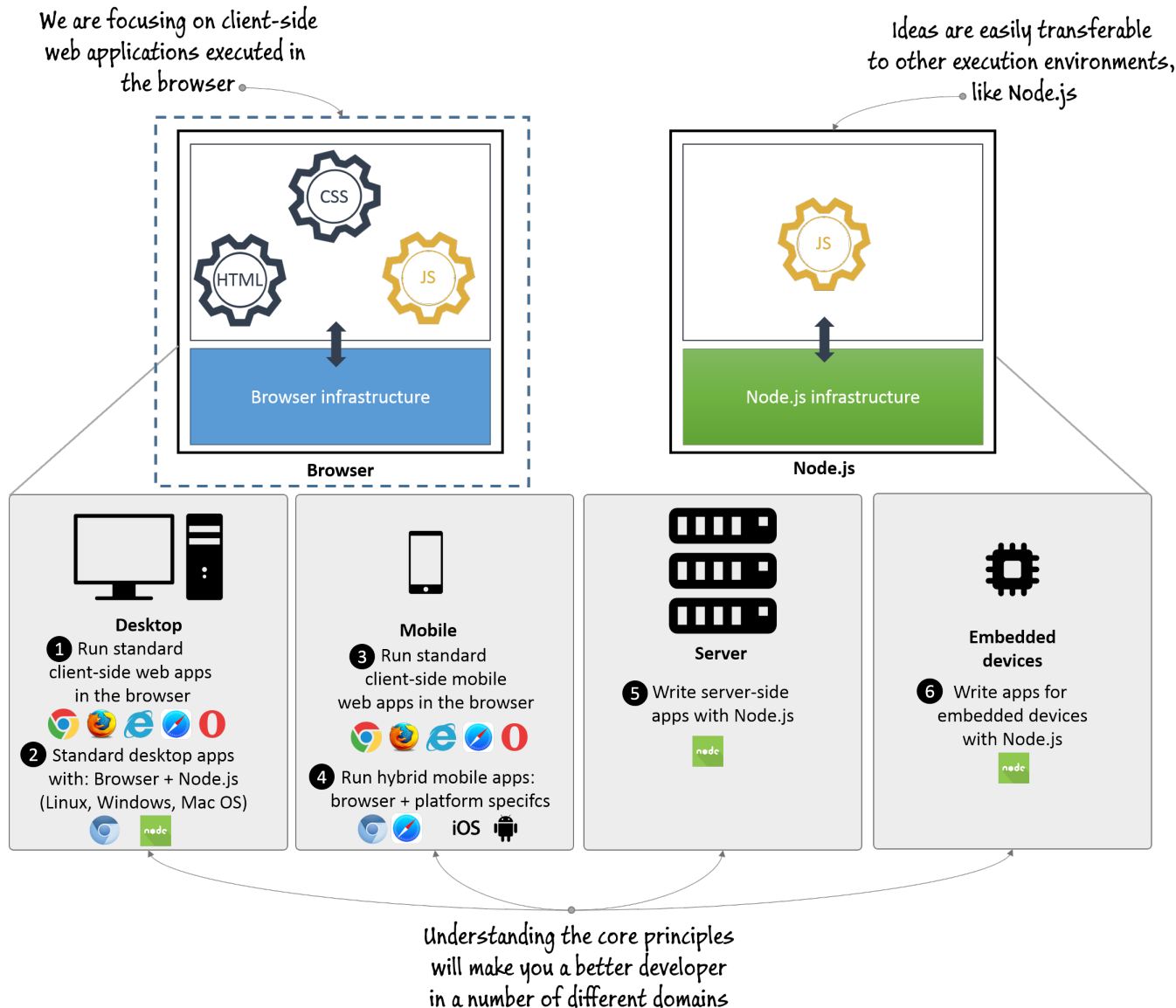
<https://dzone.com/articles/secrets-javascript-ninja>



RWTH AACHEN
UNIVERSITY

JÜLICH
FORSCHUNGSZENTRUM

RWTH AACHEN
UNIVERSITY OF APPLIED SCIENCES



HTML & CSS bereits kennengelernt

- Trennung wichtig

Trennung von Inhalt und Präsentation

- Semantisches Markup bzw. strukturelle Informationen in HTML
- Wird durch Inhalte ergänzt (z.B. „dieser Text ist wichtig“)
- Informationen zur Darstellung nur im Stylesheet

Vorteile

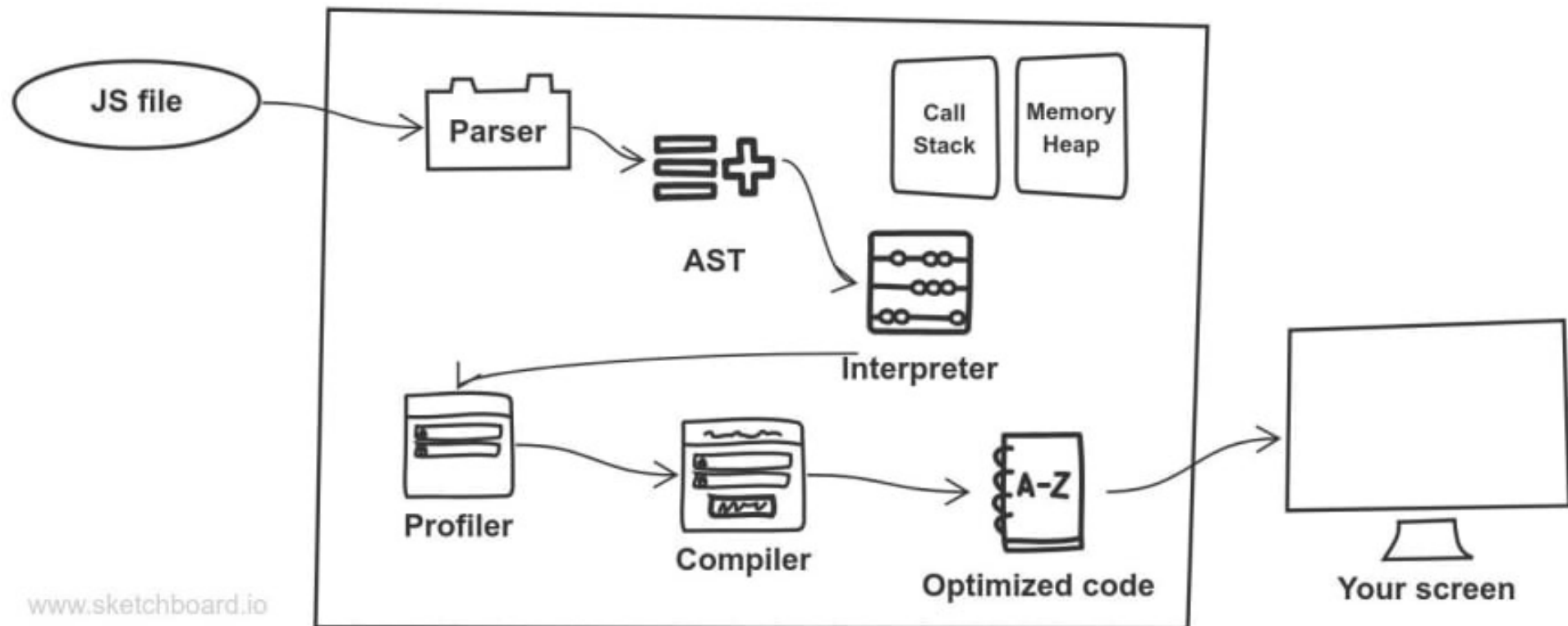
- Präsentation ist mit geringem Aufwand anpassbar, ohne Änderungen im HTML-Code vornehmen zu müssen
- Je „sauberer“ die Trennung von HTML/CSS desto einfacher das Zusammenspiel mit JavaScript

JavaScript-Engine im Browser

<https://dev.to/sanderdebr/a-brief-explanation-of-the-javascript-engine-and-runtime-2idg>

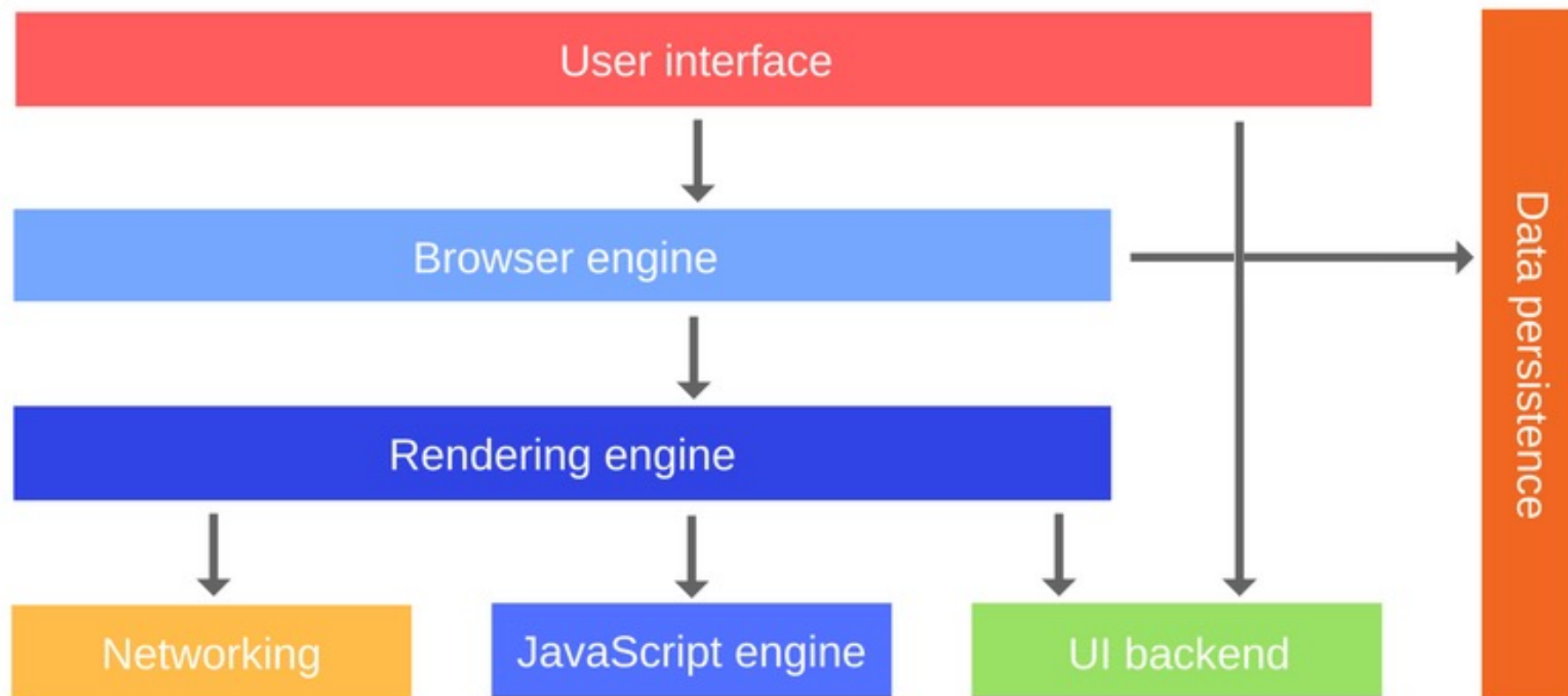


Javascript Engine



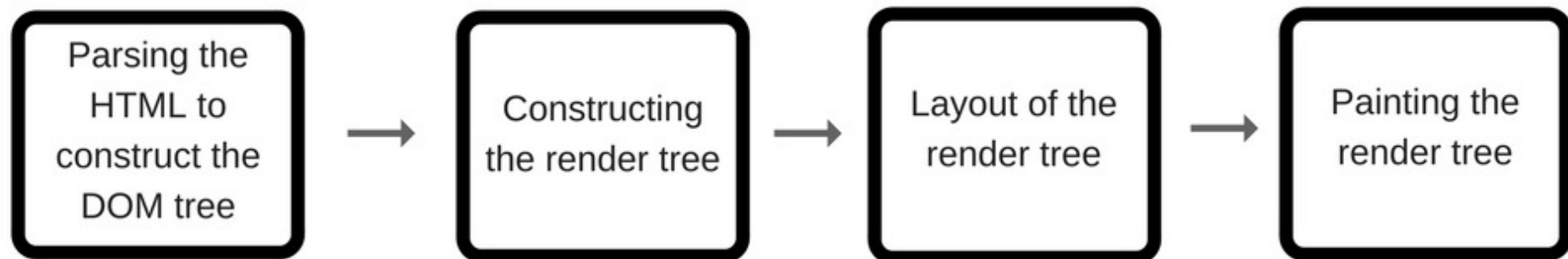
Im Browser

<https://blog.sessionstack.com/how-javascript-works-the-rendering-engine-and-tips-to-optimize-its-performance-7b95553baeda>



Im Browser

<https://blog.sessionstack.com/how-javascript-works-the-rendering-engine-and-tips-to-optimize-its-performance-7b95553baeda>

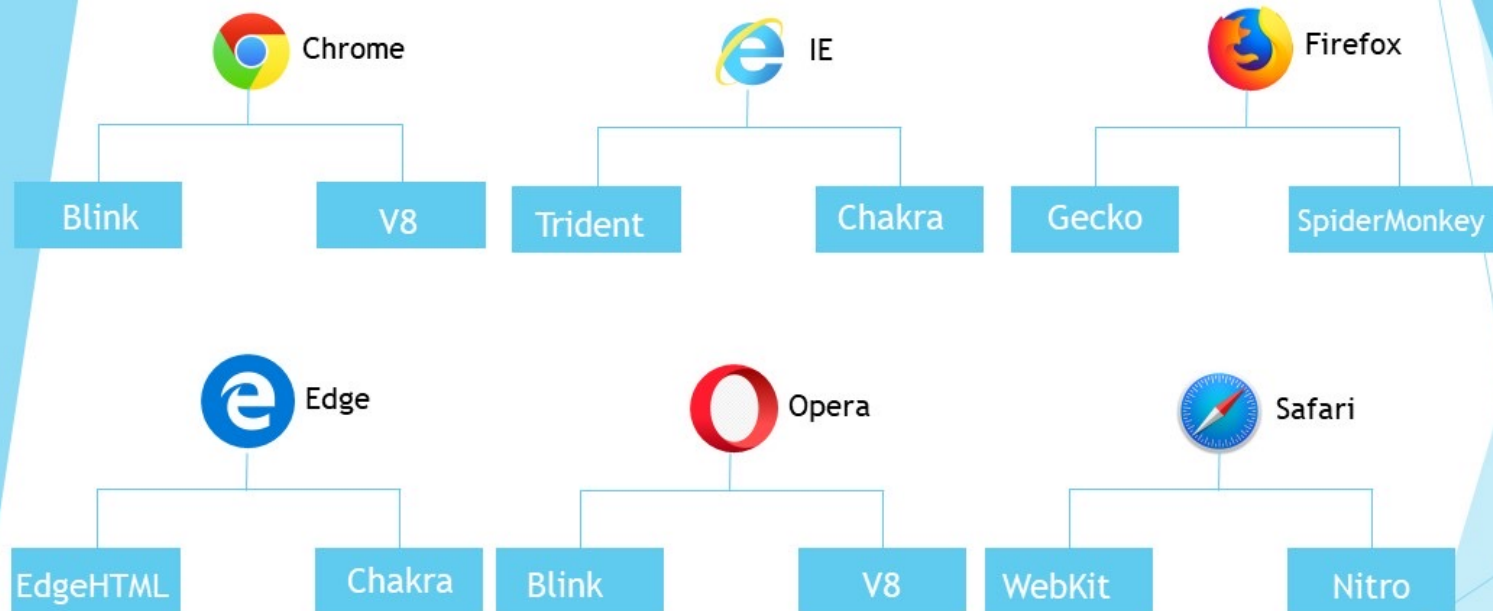


Rendering und JavaScript-Engines

<https://www.lambdatest.com/blog/browser-engines-the-crux-of-cross-browser-compatibility/>

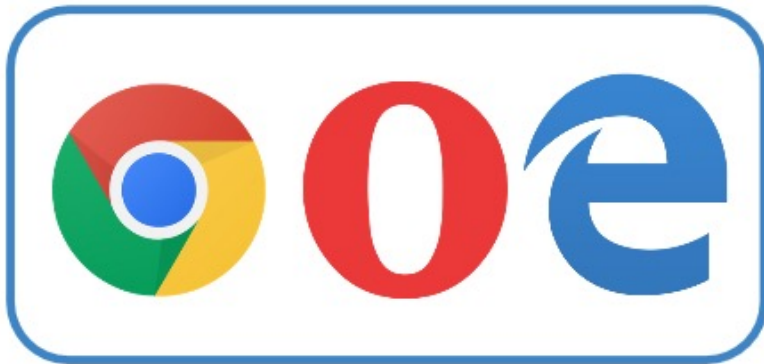


Rendering Engines & JavaScript Engines



Die Besonderheiten beim Ablauf von JavaScript im Browser werden wir noch behandeln

BROWSER ENGINES



**BLINK
ENGINE**



**WEBKIT
ENGINE**



**TRIDENT
ENGINE**



**GECKO
ENGINE**

Microsoft Edge nun auch Chromium-basiert (Legacy Edge ist ausgelaufen)

Rendering und JavaScript-Engines

<https://bloggeek.me/chrome-only-browser/>



RWTH AACHEN
UNIVERSITY

JÜLICH
FORSCHUNGSZENTRUM

RWTH AACHEN
UNIVERSITY OF APPLIED SCIENCES



Microsoft Edge nun auch Chromium-basiert (Legacy Edge ist ausgelaufen)

Grundlagen JavaScript

Entwickelt 1995 von Brendan Eich

ECMAScript (ES) liefert den Ursprung und regelt in den unterschiedlichen Versionen den Standard und damit den Funktionsumfang und die Syntax

- Hatte ursprünglich auch andere Sprachen ActionScript und Jscript
- Wichtiger Schritt: ECMAScript 6 in 2015

Objektorientiert, aber klassenlos (Klassen werden aber ab ES6 nachempfunden)

- OOP mittels Prototyping

Merkmale der Sprache

„It's a **single-threaded non-blocking**
asynchronous event-based language!“

Das Thema single-threaded wird uns noch beschäftigen.

Non-Blocking und Asynchronous werden eine Schlussfolgerung sein

Ausführung des Skriptes auf dem Rechner des „Websurfers“



Clientseitige Ausführung des Skriptes

- Clientseitige Programmiersprache
- Eingebettet in HTML-Dokumente
- Strikte Trennung von PHP und JavaScript-Dateien empfiehlt sich

Zugriff auf das Browserfenster mit dem darin enthaltenen Dokument (DOM)

- Möglichkeit auf Benutzereingaben zu reagieren ohne Server
- Manipulation des Dokuments (Single Page)
- Kommunikation mit dem Server möglich

Auf (so gut wie) jeder Plattform bzw. jedem Browser unterstützt

Wir beschäftigen uns auch mit der serverseitigen Ausführung!

Vorteile von JavaScript im Browser

- Veränderung der Webseite (DOM-Modell) ohne diese neu laden zu müssen
 - > Reduziert Netznutzung und Zeit
 - > Schnellere Interaktion mit dem Benutzer
 - > Auslagern von Funktionen auf den Client möglich (Rich Internet Client)
 - > Steigert die Nutzbarkeit und Benutzerfreundlichkeit

- Typische Anwendungsfälle:
 - > Verbesserte Interaktion mit dem Benutzer
 - Validierung von Formulareingaben vor der Übertragung zum Server
 - Anzeige von Dialogfenstern
 - Vorschlagen von Suchbegriffen während der Eingabe
 - > Entlastung des Servers und Nutzung von Rechenkapazitäten des Clients
 - > Werbung
 - > Single-Page-Anwendung

Beispiel 1: (Einbindung von Code)

```
<!DOCTYPE html>
<html>
<head>
  <title>Skriptprogrammierung</title>
  <link rel="stylesheet" href="style.css" />
  <script src="script.js" defer></script>
</head>
<body>
  <h1>Willkommen im Kurs!</h1>
  <p>...</p>
</body>
</html>
```

Defer lädt die Daten parallel und führt dazu, dass der DOM fertig ist, wenn auf DOMContentLoaded zugegriffen wird

Beispiel 1: script.js

```
window.onload = function () {  
    alert('Willkommen!');  
};
```

Ist das Dokument fertig geladen, geben wir eine Nachricht aus

Hinweise:

- Einfachere Entwicklung durch Verwendung von Frameworks
- Klassische Ausgabe gibt es nicht (JavaScript-Konsole: console.log)
- Verwenden Sie nicht window.onload am Anfang, das ist ein Anti-Pattern

Beispiel 2:

```
<!DOCTYPE html>
<html>
<head>
  <title>Skriptprogrammierung</title>
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <button onclick="doStuff();" >Klick mich!</button>
  <script>alert('Willkommen!');</script>
</body>
</html>
```

Problem: Keine Trennung von HTML und JavaScript!

- JavaScript besser in eigene Dateien auslagern

JavaScript

Einbinden über den Script-Tag



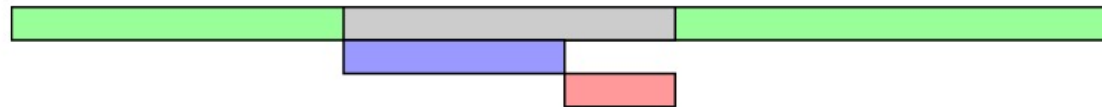
<https://www.growingwiththeweb.com/2014/02/async-vs-defer-attributes.html>

Legend

- HTML parsing
- HTML parsing paused
- Script download
- Script execution

`<script>`

Let's start by defining what `<script>` without any attributes does. The HTML file will be parsed until the script file is hit, at that point parsing will stop and a request will be made to fetch the file (if it's external). The script will then be executed before parsing is resumed.



`<script async>`

`async` downloads the file during HTML parsing and will pause the HTML parser to execute it when it has finished downloading.



`<script defer>`

`defer` downloads the file during HTML parsing and will only execute it after the parser has completed. `defer` scripts are also guaranteed to execute in the order that they appear in the document.



Nur geringe Ähnlichkeiten mit Java

Bekannt aus PHP:

- Variablen ohne festen Typen
- Funktionen ohne festen Rückgabetypen
- Keine Überladung von Funktionen

Viele Konzepte vollkommen neu

- Besonderheiten, die man aus anderen Sprachen nicht kennt
- Semikolons sind optional

Syntax

<https://opinionator.blogs.nytimes.com/2012/07/02/semicolons-a-love-story/>



Der Code kann mit Semicolons
aber lesbarer werden

Syntax

Variablen – Der ursprüngliche Weg



Beispiel:

```
var wert; // Instanziierung ohne Wert
wert = 5; // Variable wird auf 5 gesetzt
var timId = 'xy123456'; // Wert sofort setzen
```

Variablen-Deklaration mittels Schlüsselwort „var“

- Kein fester Typ!
- Nicht initialisierte Variablen haben den Wert *undefined*

Variablen, die außerhalb von Funktionen deklariert werden sind global!
Innerhalb von Funktionen sind neue Variablen lokal

Syntax

Variablen – Der ursprüngliche Weg

Variablen-Deklaration mittels Schlüsselwort „var“ ignorieren **Blöcke**, beachten **aber Funktionen!**

```
if (true) {  
  // "if" block scope  
  var count = 0;  
  console.log(count); // 0  
}  
console.log(count); // 0
```

```
function run() {  
  // "run" function scope  
  var message = 'Run, Forrest, Run!';  
  console.log(message); // 'Run, Forrest, Run!'  
}  
  
run();  
console.log(message); // throws ReferenceError
```

<https://dmitripavlutin.com/javascript-scope/>

Variablen-Deklaration mittels Schlüsselwort „let“ oder Konstante mittels „const“ beachten Blöcke

```
if (true) {  
  const message = 'Hello';  
}  
console.log(message); // ReferenceError: message is not defined
```

```
while (/* condition */) {  
  // "while" block scope  
  const message = 'Hi';  
  console.log(message); // 'Hi'  
}  
console.log(message); // => throws ReferenceError
```

```
{  
  // block scope  
  const message = 'Hello';  
  console.log(message); // 'Hello'  
}  
console.log(message); // throws ReferenceError
```

<https://dmitripavlutin.com/javascript-scope/>

Datentypen

- Typeof-Operator liefert den Typen zurück:
 - > Beispiel: `typeof 5 // "number"`

Typ	Beschreibung	Beispiel/Literal
boolean	Wahrheitswert	true, false
number	Ganz- oder Kommazahl	0, 123, -123, 3.14, 1.2e5
string	Zeichenkette	'foo', "bar"
function	Funktion	function() { }
undefined	Undefinierter Wert	undefined, var a;
object	Objekte, aber auch „alles andere“	{}, new Car()

Weitere Informationen: <https://developer.mozilla.org/de/docs/Web/JavaScript/Datenstrukturen>

Datentypen

- Object („die Mutter aller Objekte“)
 - > Alle Objekte stammen von diesem obersten Kernobjekt ab
 - > Sammlung mehrerer Datentypen (und Funktionen)
 - > Es gibt von JavaScript vorgegebene Objekte (Arrays, ...) und Browser-Objekte (Document, ...)
 - > Eigene Objekte möglich
- Primitive Datentypen (Boolean-, Number- und String-Werte)
 - > Übergabe als Kopie
- Objekte und Funktionen
 - > Übergabe als Referenz

Typumwandlung

- Implizit – Beispiel:

```
var num = '3' + 5; // 35
```

- Explizit – Beispiele:

```
var foo = parseInt('5.3'); // 5
var bar = parseFloat('5.3'); // 5.3
var b1 = !! (0); // false
var b2 = !! ('0'); // true
var str = (5.3).toString(); // "5.3"
```

Vergleich von Variablen

- „**falsy values**“ (diese Werte liefern bei Interpretation als Boolean false)
 - > False
 - > 0
 - > ""
 - > Null
 - > undefined
 - > NaN (Not-A-Number, ist aber eine Number!)
- Alle anderen Werte (z.B. auch leere Arrays) ergeben true
 - > „truthy values“

Weitere Informationen: <https://developer.mozilla.org/en-US/docs/Glossary/Falsy>

Vergleich von Variablen

```
var a = (false == ""); // true
var b = (false == 0);  // true
var c = (0 == "");     // true

var d = (null == null); // true
var e = (null == false); // false
var f = (undefined == undefined); // true
var g = (undefined == null); // true

var h = (NaN == null); // false
var i = (NaN == NaN);  // false !!!
```

Auch JavaScript bietet den **typstarken Vergleich**: **a === b**

Verkettung mittels +-Operator

Literale:

```
'Ich bin ein String'
```

```
"Ich auch!"
```

- Kein Unterschied (anders als bei PHP), trotzdem ist Einheitlichkeit sinnvoll

Steuerzeichen können mittels Backslash angegeben werden

- Beispiel: `'Zeilenumbruch folgt\n2. Zeile!'`

Benutzung des Anführungszeichens muss maskiert werden

```
document.write("Hallo, Sie befinden sich auf \"meiner Seite\");
```

Syntaktischer „Sugar“ um die Ersetzung von Variablen in Zeichenketten lesbar zu machen

`` ... ${ Ausdruck mit Variablen } ... ``

```
let a = 5;
let b = 10;
console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b) + '.');
// "Fifteen is 15 and
// not 20."
```

```
let a = 5;
let b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
// "Fifteen is 15 and
// not 20."
```


Grundsätzliches bekannt aus Java (jetzt auch aus PHP!)

- if
- else if
- else

- for
- while
- do-while
- break
- continue

for-in-Schleife

```
for (let key in obj) {  
    alert(key + ': ' + obj[key]);  
}
```

Durchläuft das Objekt (oder Array) und gibt die enthaltenen Eigenschaften aus

- Besser mit Bedacht zu verwenden
 - > Arrays möglichst mit `for ... of` verwenden
 - > Vorsicht bei der Änderung der Prototypen im Durchlauf
 - > Mehr dazu später

Weitere Informationen: <https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/for...in>

Eigenschaften

- Keine feste Größe
- Als Wert ist alles erlaubt
 - > Arrays in Arrays sind möglich
 - > Objects in Arrays sind möglich
- Arrays können keine Lücken haben (Lücken werden mit undefined aufgefüllt)

Assoziative Arrays

- Gibt es nicht!*
- > Dafür gibt es Objekte (die sich aber anders verhalten)

Beispiele

- Erstellung eines Arrays

```
var emptyArr1 = new Array(); // leeres Array erstellen
var emptyArr2 = [];          // Alternative (präferiert)
```

```
var arr = ['one', 2];
```

- Zugriff auf Werte

```
var one = arr[0];           // one
var two = arr[arr.length-1]; // 2
```

```
arr[4] = 4;
var len = arr.length;       // 5
```

for-of-Schleife

```
const array1 = ['a', 'b', 'c'];
```

```
for (const element of array1) {  
    console.log(element);  
}
```

```
// expected output: "a"
```

```
// expected output: "b"
```

```
// expected output: "c"
```

Iteriert über die Werte eines entsprechende Iterable-Objekts

For-in iteriert über die aufzählbaren Eigenschaften eines Objekts

Weitere Informationen: <https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/for...of>

Objekte und Funktionen unterscheiden sich stark von vielen anderen Programmiersprachen

- Typen und Vererbung werden in den meisten Programmiersprachen durch Klassen gelöst
- JavaScript besitzt keine (echten) Klassen!*
- > Stattdessen gibt es Konstruktorfunktionen und Prototypen für Objekte
- > Objekte kapseln aber auch hier Eigenschaften und Funktionen
- > Jede Funktion hat eine prototype-Eigenschaft, die für Vererbung genutzt werden können (müssen)

Wenn wir von Objekten reden, meinen wir „Object-Objekte“

- Eigentlich ist auch eine Zahl, ein String, und so ziemlich alles in JavaScript ein Object
- Mit Objekten meinen wir Werte, bei denen der typeof-Operator "object" zurückgibt
 - > Außer null...

Seit ES6 gibt es das Schlüsselwort „class“, aber die hiermit erstellen Klassen verhalten sich anders als traditionelle Klassen aus anderen Sprachen, es bleibt prinzipiell beim Prototyping-Modell; IE11 unterstützt dies nicht

Beispiel:

```
function fak(n) {  
    if (n <= 1) {  
        return 1;  
    }  
    return n * fak(n - 1);  
}
```

- Erstellt eine Funktion und legt diese unter dem Namen „fak“ im aktuellen Geltungsbereich der Variablen (Scope) ab
- Wenn innerhalb von Funktionen Variablen ohne das Schlüsselwort var oder let deklariert werden, dann sind diese Variablen global, wurden also extern bereits erstellt
 - > Für lokale Variablen sicherstellen, dass „var“ oder "let" verwendet wurde
 - > Parameter sind immer lokal

Beispiel (als anonyme Funktion):

```
var fak = function (n) {  
    if (n <= 1) {  
        return 1;  
    }  
    return n * fak(n - 1);  
};
```

- Erstellt eine anonyme Funktion und **speichert den Funktionszeiger** in der Variable „**fak**“
- „fak“ wird im aktuellen Scope abgelegt
- Äquivalent zum ersten Ausdruck
 - > **Fast**, denn die Funktion wird hier erst bekannt, wenn die Zuweisung `var fak =` erreicht wird. Bei benannten Funktionen sind diese auch schon vorher bekannt und können schon **vor Deklaration** genutzt werden

Anonyme Funktionen

- Schnelle Erzeugung ohne Speicherung im Geltungsbereich
- Häufig verwendet bei Eventhandlern, die oft eine feste Zuordnung haben:

```
element.onclick = function () {  
    alert('Geklickt!');  
}
```

- Typisch für Javascript: Kapselung des Skriptes

```
(function() {  
    var info = 'Hallo'; // Nur innerhalb sichtbar  
    alert(info);  
})();  
// Variable info ist in diesem Geltungsbereich unbekannt  
// Es gibt auch keinen etwaigen Konflikt mit dem Namen
```

Vergleich zu anderen Sprachen:

- C++-Code (!!!):

```
int x = 1;
if (true) {    // neuer Geltungsbereich!
    int x = 2; // neue "innere" Variable x
}
// x hat den Wert 1
```

JavaScript-Code:

```
var x = 0;
(function() {           // Neuer Geltungsbereich
    var x = 1;
    if (true) {
        var x = 2; // JavaScript ignoriert das "var", da
                    // es bereits eine Variable x
                    // im aktuellen Scope gibt
    }
    // x hat den Wert 2
}) ();
// x hat den Wert 0
```

Funktion erstellt neuen Scope, geschweifte Klammern nicht!

- Variablen am Anfang einer Funktion deklarieren
- Anders wird dies bei der Verwendung von **let**

JavaScript-Code mit let (seit ECMAScript 6 2015):

```
var x = 0;
(function() {           // Neuer Geltungsbereich
    let x = 1;
    if (true) {
        let x = 2; // Neue Variable x im inneren Bereich
    }
    // x hat den Wert 1
}) ();
// x hat den Wert 0
```

Syntax

let vs. var



```
function abc() {  
    //i *ist* auch hier sichtbar  
    for( var i = 0; i < 5; i++ ) {  
        // i ist in der ganzen Funktion sichtbar  
    }  
    // i *ist* auch hier sichtbar  
}
```

```
function abc() {  
    //i *ist* hier nicht sichtbar  
    for( let i = 0; i < 5; i++ ) {  
        // i ist nur in diesem Block sichtbar  
    }  
    // i *ist* hier nicht sichtbar  
}
```

Konstante sollten mit const angelegt werden, Sichtbarkeit wie let!

```
const dieLoesungallerFragen = 42;
```

```
dieLoesungallerFragen = 13; // ergibt einen Fehler
```

```
var dieLoesungallerFragen = 13 // ergibt einen Fehler
```

```
let dieLoesungallerFragen = 13 // ergibt einen Fehler
```

```
const wertNotwendig; // ergibt einen Fehler
```

Vor let gab es viele Probleme durch „fehlenden“ Geltungsbereich

- Beispiel:

- > Wir assoziieren den onclick-Event mit einigen DOM-Elementen

- ```
var domelements = document.getElementsByClassName('name');
```

- > Das erste Element soll beim Klick 0 ausgeben, das zweite 1, usw.

```
for (var i = 0; i < 10; i++) {
 domelements[i].onclick = function () {
 alert(i);
 };
}
```

- Nach dem Durchlaufen hat *i* den Wert 10

- Unabhängig davon welches Element geklickt wird, hat *i* den Wert 10

- > Wir kommen überhaupt nicht zum Drücken eines Elements während die Schleife läuft

- > Wenn wir dann später drücken, dann haben wir die Schleife bereits verlassen, die alert(i)-Funktion greift aber auf das *i* des Geltungsbereichs zu

**Mögliche Lösung: Kapseln in einer Funktion mit Parameter (der lokal ist)  
(Übergang zum Closure-Ansatz)**

```
for (var i = 0; i < 10; i++) {
 (function(innerI) { // innerI "hält" nun i
 domelement[innerI].onclick = function () {
 alert(innerI);
 };
 })(i);
}
```

- Neuer Geltungsbereich mit Variable *innerI*
- *Damit haben wir durch den Aufruf (i) in der Schleife jeweils den Wert gesichert*

Weitere Informationen: <https://wiki.selfhtml.org/wiki/JavaScript/Funktion>




# Syntax

## let bietet Vorteile bei Funktionen



### Mögliche Lösung: „let“-Schlüsselwort



```
for (let i = 0; i < 10; i++) {
 domelement[i].onclick = function () {
 alert(i);
 };
}
```

- Variablen sind jetzt an Block gebunden (statt an Funktion!)
  - > Faktisch ist mit der Deklaration im For-Statement für jeden Durchlauf ein eigenes i

```
var a = 5;
let(a = 6) console.log(a); // 6
console.log(a); // 5
```

Weitere Informationen: <https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/let>, <http://caniuse.com/#feat=let>

## Variable Anzahl an Parametern möglich

- „Magische“ Variable arguments ist in jeder Funktion verfügbar
  - > Array-ähnliches Objekt, das die übergebenen Parameter enthält
  - > Beispiel:

```
function sumAll() {
 var sum = 0;
 for (var i=0; i<arguments.length; i++) {
 sum += arguments[i];
 }
 return sum;
}
```

**Rückgabewert kann ein beliebiger Wert (auch Objekt sein)**

Weitere Informationen: <https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Functions/arguments>

### Aufpassen: Scope beachten

- Das Beispiel funktioniert nicht so gut...

```
function getId() {
 var counter = 0;
 counter++;
 return counter;
}
```

```
var id1 = getId(); // 1
var id2 = getId(); // 1
var id3 = getId(); // 1
```

Wir brauchen so etwas wie statische Variablen

## Closures

- Wir verpacken die Funktion zum Zählen in eine anonyme Funktion, die den Geltungsbereich für die Zählvariable erstellt

```
function idGetter() {
 var counter = 0;
 return function() {
 counter++;
 return counter;
 };
};

var getId = idGetter();
var id1 = getId(); // 1, setzt counter (in idGetter) auf 1
var id2 = getId(); // 2
var id3 = getId(); // 3
```

- Direkter Zugriff auf die Variable *counter* ist nicht möglich! Wir sprechen vom Lexical Scope (Zugriff auf Parent Scope möglich)

## Closures

- Vereinfachte Schreibweise

```
var getId = (function () {
 var counter = 0;
 return function () {
 counter++;
 return counter;
 };
})();
```

```
var id1 = getId(); // 1
var id2 = getId(); // 2
var id3 = getId(); // 3
```

## JSON: JavaScript Object Notation

- JSON-Dokumente sind gültiges JavaScript
- Einfache Struktur
- Typisierung eingebaut!
- Beispiel:

```
{
 "id" : 2648,
 "Name" : "Mustermann",
 "Vorname" : "Max",
 "adr" : {
 "Stadt" : "Aachen",
 "plz" : 52064
 },
 "tel" : ["0241 1234", "0160123456"],
 "partner" : null,
 "maennlich" : true
}
```

Weitere Informationen: <http://www.json.org/>

## Typen:

- Number (wie in JavaScript)
- String (nur in doppelten Anführungszeichen)
- Boolean (true/false)
- Array (nur in eckigen Klammern)
- Object (in geschweiften Klammern)
- Null

## Deckt nicht alle möglichen JavaScript-Werte ab

- NaN, Infinity werden zu null serialisiert
- Function- und RegExp-Objekte werden verworfen

## Instanziierung eines Objektes

```
var user = new Object(); // Nutzung des Keywords "new"
var user = {}; // JSON-like Schreibweise
```

> Identisches Ergebnis, unterschiedliche Schreibweise

## Anschließende Zuweisung der Attribute

```
user.id = 12; // Attribut id hinzufügen
user.name = 'Max Mustermann'; // Attribut name hinzufügen
user['name'] = 'Max Mustermann'; // Alternative
```

## Alternative (nur für die JSON-like Schreibweise)

- Attribute sofort bei der Instanziierung setzen

```
var user = {
 id : 12,
 name : "Max Mustermann"
};
```



## Auslesen über .-Operator oder eckige Klammern

- Analog zur Zuweisung
- Beispiele:

```
var id1 = user.id; // 12
var id2 = user['id']; // Alternative
```

**Als Wert eines Attributes ist alles möglich**

## Unser erstes „richtiges“ Objekt

```
var auto = {
 maxSpeed : 140,
 name : 'Corsa',
 distance : 0,
 go : function(times) {
 auto.distance += auto.maxSpeed * times;
 },
 getDistance : function() {
 return auto.distance;
 }
};

auto.go(2);
alert(auto.getDistance()); // 280
```

### Variable Verarbeitung eines JSON-Objekts:

```
var keys = Object.keys(o);
var values = Object.values(o);
for (j=0; j < keys.length; j++) {
 alert(keys[j] + " " + values[j]);
}
```

Hier würde dann aber auch die Funktionen als Key ausgegeben und als Value der Funktionstext

```
Object.keys(obj).forEach(e => console.log("key=${e} value=${obj[e]}"));
```

### Variable Verarbeitung eines JSON-Objekts:

```
function DisplayObjectProperties(obj) {
 var text = "";
 for (prop in obj) {
 if(typeof obj[prop] != "function" &&
 typeof obj[prop] != "object") {
 text = text + prop + ": " + obj[prop] + " " ;
 }
 else if (typeof obj[prop] === "object") {
 DisplayObjectProperties(obj[prop])
 }
 }
 alert(text) ;
}
DisplayObjectProperties(einObjekt) ;
```

### Variable this

- Innerhalb jeder Funktion verfügbar
- **Enthält das Objekt über das die Funktion aufgerufen wurde**
- Beispiel:

```
document.getElementById('btn').onclick = function() {
 this.innerHTML = 'GEKLICKT!'; // this enthält Button
};
```

- Wurde eine solche Funktion von keinem Objekt aufgerufen, so enthält this im Browser das globale Objekt window
  - > window ist das „globale“ Browserobjekt

Mehr zu window: <https://developer.mozilla.org/docs/Web/API/Window/window>

### Unser erstes „richtiges“ Objekt

```
var auto = {
 maxSpeed : 140,
 name : 'Corsa',
 distance : 0,
 go : function(times) {
 auto.distance += auto.maxSpeed * times;
 },
 getDistance : function() {
 return auto.distance;
 }
};
```

```
auto.go(2);
alert(auto.getDistance()); // 280
```

### Unser erstes „richtiges“ Objekt (verbessert)

```
var auto = {
 maxSpeed : 140,
 name : 'Corssa',
 distance : 0,
 go : function(times) {
 this.distance += this.maxSpeed * times; // this
 },
 getDistance : function() {
 return this.distance; // this
 }
};

auto.go(2);
alert(auto.getDistance()); // 280
```

### „Herausziehen“ von Funktionen kann zu Probleme führen

- Beispiel:

```
document.getElementById('btn').onclick = auto.go;
```

```
// ...
var auto = {
 // ...
 go : function () {
 // this zeigt auf den Button
 },
 // ...
};
```



### „Herausziehen“ von Funktionen kann zu Probleme führen

- Beispiel:

```
document.getElementById('btn').onclick = function () {
 auto.go();
};

// ...
var auto = {
 // ...
 go : function () {
 // this zeigt auf auto
 },
 // ...
};
```

### Manipulation des this-Zeigers, Beispiel:

```
var auto = { // ...
 go : function (times) {
 this.distance += this.maxSpeed * times;
 }, // ...
};
```

```
var auto2 = {
 maxSpeed : 100,
 distance : 0
};
auto.go.apply(auto2, [3]);
alert(auto2.distance); // 300
```

*auto2* bei Ausführung der Funktion *go* als *this*-Objekt nutzen

Parameter als Array

> Wendet Funktion *go* des Objektes *auto* auf das Objekt *auto2* an.

Weitere Informationen: [https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global\\_Objects/Function/apply](https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Function/apply)

## Manipulation des this-Zeigers

```
Function.apply(thisArg [, argsArray])
```

- Führt die Funktion aus und setzt den this-Zeiger auf thisArg
- Werte aus argsArray werden der Funktion als Parameter übergeben
  - > Sinnvoll, wenn unklar ist, wie viele Parameter übergeben werden

```
Function.call(thisArg [, arg1 [, arg2 [, ...]]])
```

- Führt die Funktion aus und setzt den this-Zeiger auf thisArg
- Parameter werden als normale Parameter übergeben

## Konstruktorfunktion

- Jede Funktion kann auch als Konstruktor genutzt werden
- Beispiel:

```
// Diese Funktion wollen wir als Konstruktor nutzen
function Fahrzeug(speed) {
 this.speed = speed;
 this.distance = 0;
};
```

```
// Erstellen eines neuen Objektes des Typs Fahrzeug
var fahrzeug1 = new Fahrzeug(120);
```

```
alert(fahrzeug1.speed); // 120
```

- Achtung: Man kann Fahrzeug auch ohne new aufrufen. This zeigt in diesem Fall auf window

## Konstruktorfunktion

- Um das this-Problem der Nutzung der Konstruktorfunktion ohne new zu unterbinden kann man z.B. einen Self-Invoking-Konstruktor verwenden
- Beispiel:

```
function Fahrzeug(speed) {
 if (!(this instanceof Fahrzeug)) {
 return new Fahrzeug(speed);
 }
 this.speed = speed;
 this.distance = 0;
};
```

```
new Fahrzeug(120);
Fahrzeug(120); // Funktioniert auch
```

- Seit ECMAScript 2015 können wir Objekte auch syntaktisch als Klasse darstellen (hierbei handelt es sich um kein echtes OOP-Modell)
- Beispiel:

```
class Rechteck {
 constructor (hoehe, breite) {
 this._hoehe = hoehe;
 this._breite = breite;
 }
}
```

- Es gibt die Konvention, dass private Attribute und Methoden mit \_ beginnen

# Syntax

## Hoisting und Inline-Definition



- **Im Gegensatz zu normalen Funktionsdeklarationen** kann eine Klasse erst dann verwendet werden, wenn sie vorher deklariert wurde

- Beispiel:

```
let k = new EineKlasse(); // Ergibt einen Reference-Fehler
```

```
class EineKlasse{...};
```

- Eine Klasse kann auch anonym und direkt zugewiesen werden

- Beispiel

```
let var = class {...};
```

```
let var = class EineKlasse{...};
```

- Wir können wie gewohnt Methoden in einer Klasse definieren (ohne `function`)
- Beispiel:

```
class Rechteck {
 constructor (hoehe, breite) {
 this._hoehe = hoehe;
 this._breite = breite;
 }
 get flaeche() { //getter mit schlüsselwort get
 return this.berechneFlaeche();
 }
 berechneFlaeche() { // klassische methode
 return this.hoehe * this.breite;
 }
}

const r = new Rechteck(5,10);
console.log(r.flaeche); // getter, über Attribute zugreifen
```



- Beispiel:

```
class Rechteck {
 // ... siehe vorherige Folie
 static compare(r1, r2) {
 if (r1.flaeche() > r2.flaeche) {
 return r1;
 }
 else
 return r2;
 }
}

const r1 = new Rechteck(5,10);
const r2 = new Rechteck(10,10);
console.log(Rchteck.compare(r1,r2).flaeche);
```

- Setter erlauben den Zugriff auf private Attribute mittels .
- Beispiel:

```
class Rechteck {
 constructor (hoehe, breite) {
 this._hoehe = hoehe;
 this._breite = breite;
 }
 get hoehe() {
 return this._hoehe
 }
 set hoehe(hoehe) {
 this._hoehe = hoehe
 }
 ...
}

const r = new Rechteck(5,10); r.hoehe = 42;
```

- Wir können Klassen ableiten
- Beispiel:

```
class Tier{
 constructor(name) { this._name = name; }
 sprich() {
 console.log(this._name + ' macht ein Geräusch'); } }

class Hund extends Tier{
 sprich() { console.log(this._name + ' wau'); }
}

var h = new Hund('Wuffii');
h.sprich();
```

## Arrow Funktion

- Gern genutzte Kurzschreibweise für Funktionen
- Zusätzlich **übernimmt** die aufgerufene Funktion den **this-Zeiger** und bindet diesen innerhalb der Funktion nicht neu
- Wird z.B. für anonyme Callbacks oder insbesondere für Transformationen (Lambdas) verwendet

```
[1,2,3].map((value) => value + 1);
```

```
function Person() {
 this.age = 0;

 setTimeout(() => {
 this.age++; // this zeigt immer noch auf Person
 console.log('Alter: ' + this.age);
 }, 1000);
}
var person = new Person();
```

Weitere Informationen: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow\\_functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

# Arrow-Funktion

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow\\_functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

```
const materials = [
 'Hydrogen',
 'Helium',
 'Lithium',
 'Beryllium'
];

console.log(materials.map(material => material.length));
// expected output: Array [8, 6, 7, 9]
```

```
// Parameterless arrow functions that are visually easier to parse
setTimeout(() => {
 console.log('I happen sooner');
 setTimeout(() => {
 // deeper code
 console.log('I happen later');
 }, 1);
}, 1);
```

|                              | Desktop |      |           |                   |       |        | Mobile          |                |                 |               |            |                  | Node.js    |
|------------------------------|---------|------|-----------|-------------------|-------|--------|-----------------|----------------|-----------------|---------------|------------|------------------|------------|
|                              | Chrome  | Edge | Firefox   | Internet Explorer | Opera | Safari | WebView Android | Chrome Android | Firefox Android | Opera Android | iOS Safari | Samsung Internet | Node.js    |
| Arrow functions              | 45      | 12   | 22 ★<br>▼ | No                | 32    | 10     | 45              | 45             | 22 ★<br>▼       | 32            | 10         | 5.0              | 4.0.0<br>▼ |
| Trailing comma in parameters | 58      | 12   | 52        | No                | 45    | 10     | 58              | 58             | 52              | 43            | 10         | 7.0              | 8.0.0      |

Weitere Informationen: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow\\_functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

# Single-Threaded Non-Blocking

---

```
setTimeout(() => { console.log('I happen sooner');
 setTimeout(() => {
 // deeper code
 console.log('I happen later');
 }, 1);
}, 1);

console.log('I am actually the first');
```

Weitere Informationen: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow\\_functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

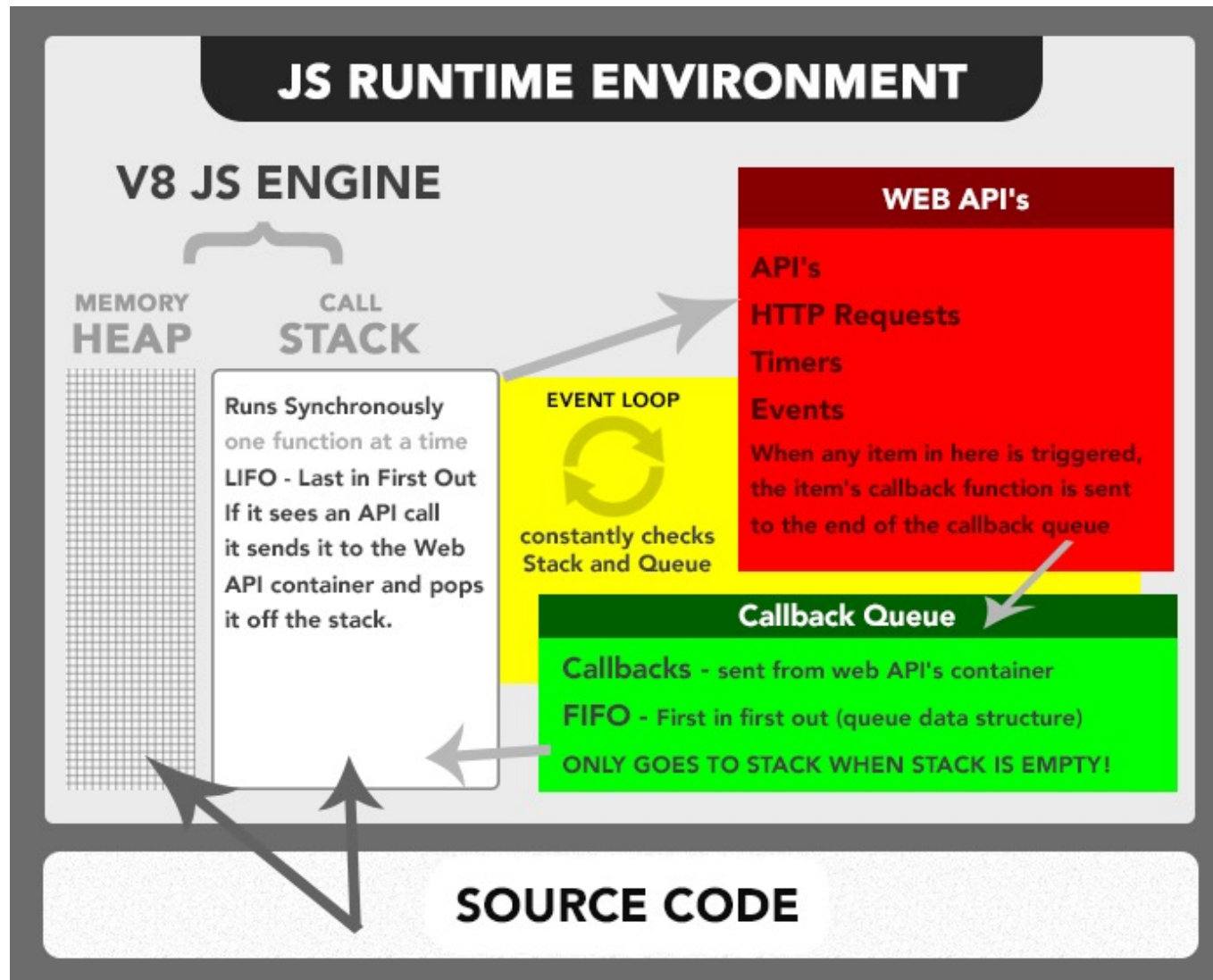
## Charakteristik einer JavaScript Runtime

„It's a single-threaded non-blocking asynchronous event-based language!“

- Eine Thread (nur einen Call-Stack)
- Nicht-blockierend (non-blocking)
  - > Techniken, die bei anderen Sprachen (z.B. Java) die Codeabarbeitung verhindern, passieren bei JavaScript im „Hintergrund“
    - Hintergrund? Ein Thread?
- Asynchron
  - > Benutzung von Callbacks um festzustellen wenn Code fertig ist
  - > Sehr passend für die Nutzung von Events

# Event Loop

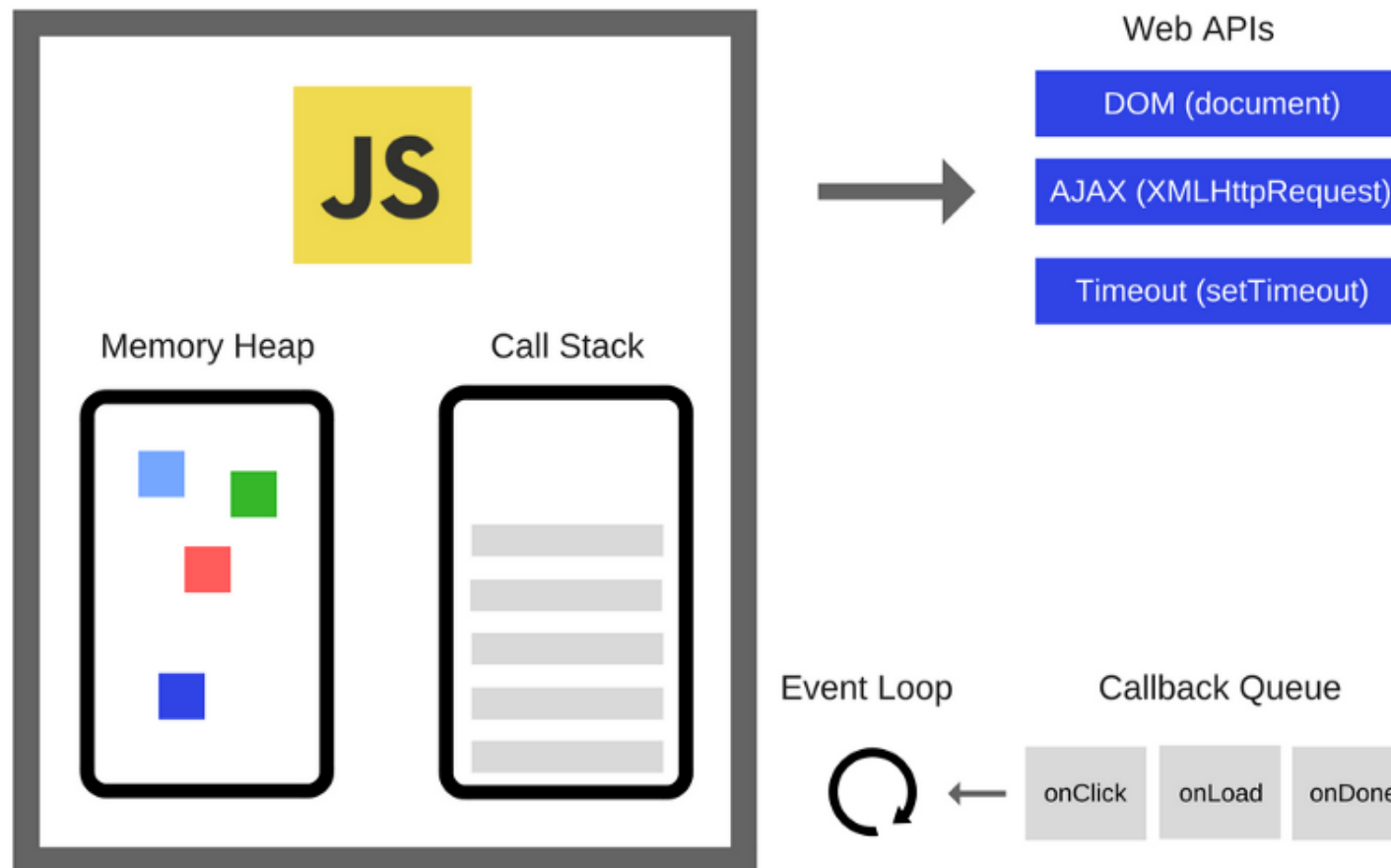
<https://medium.com/@monuchaudhary/single-threaded-non-blocking-asynchronous-and-concurrent-nature-of-javascript-a0d5483bcf4c>



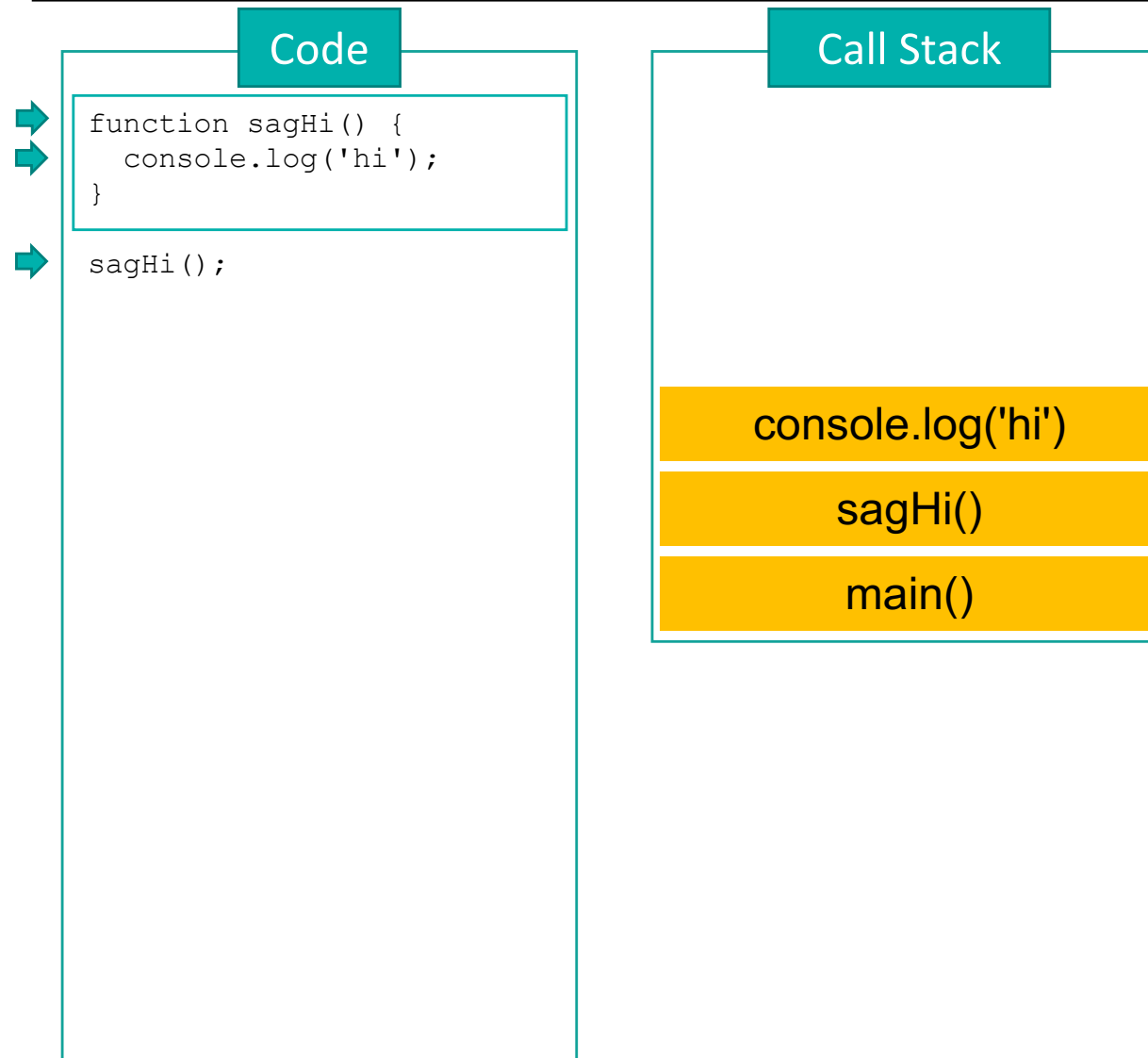


# Event Loop

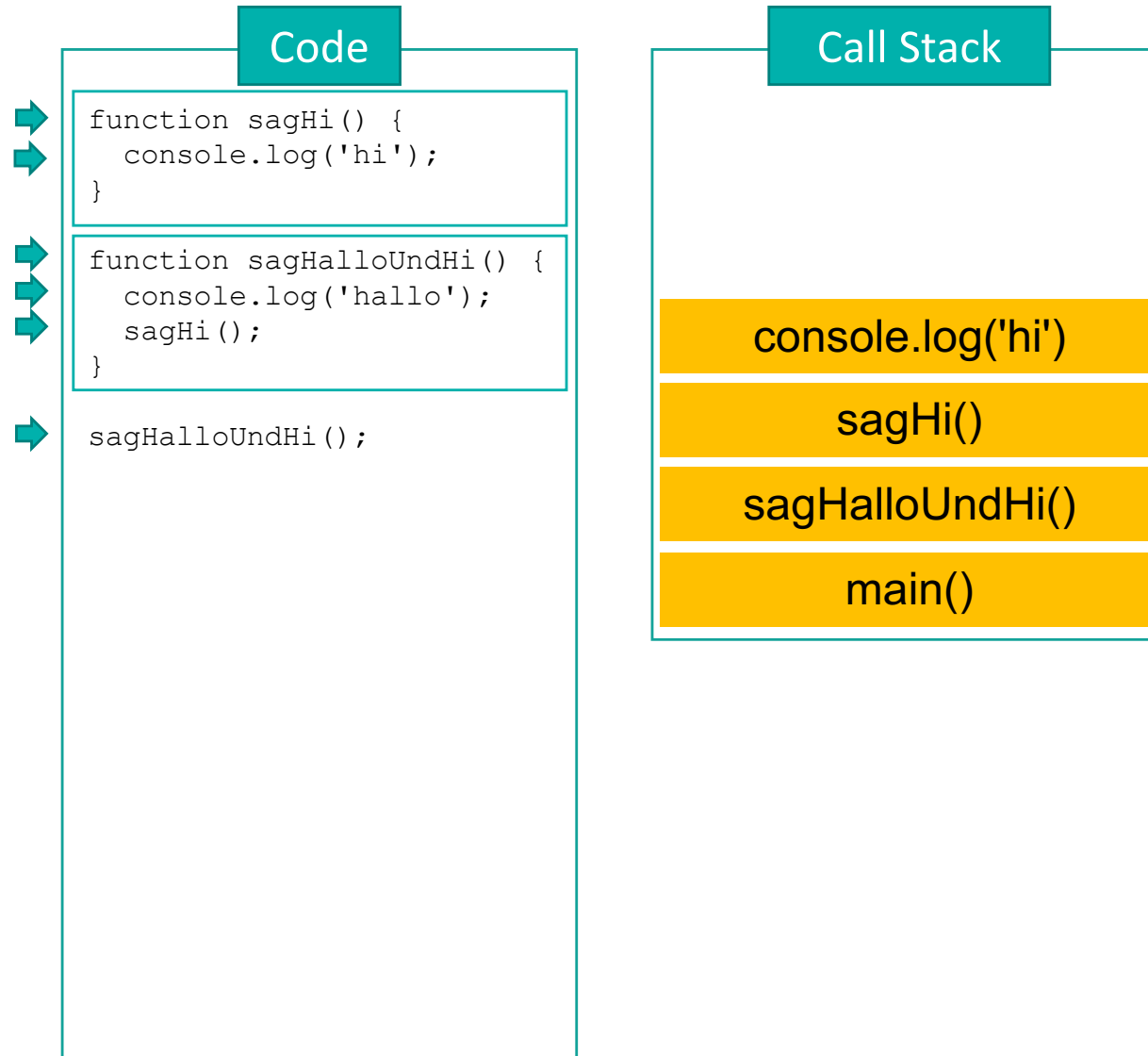
<https://medium.com/@monuchaudhary/single-threaded-non-blocking-asynchronous-and-concurrent-nature-of-javascript-a0d5483bcf4c>



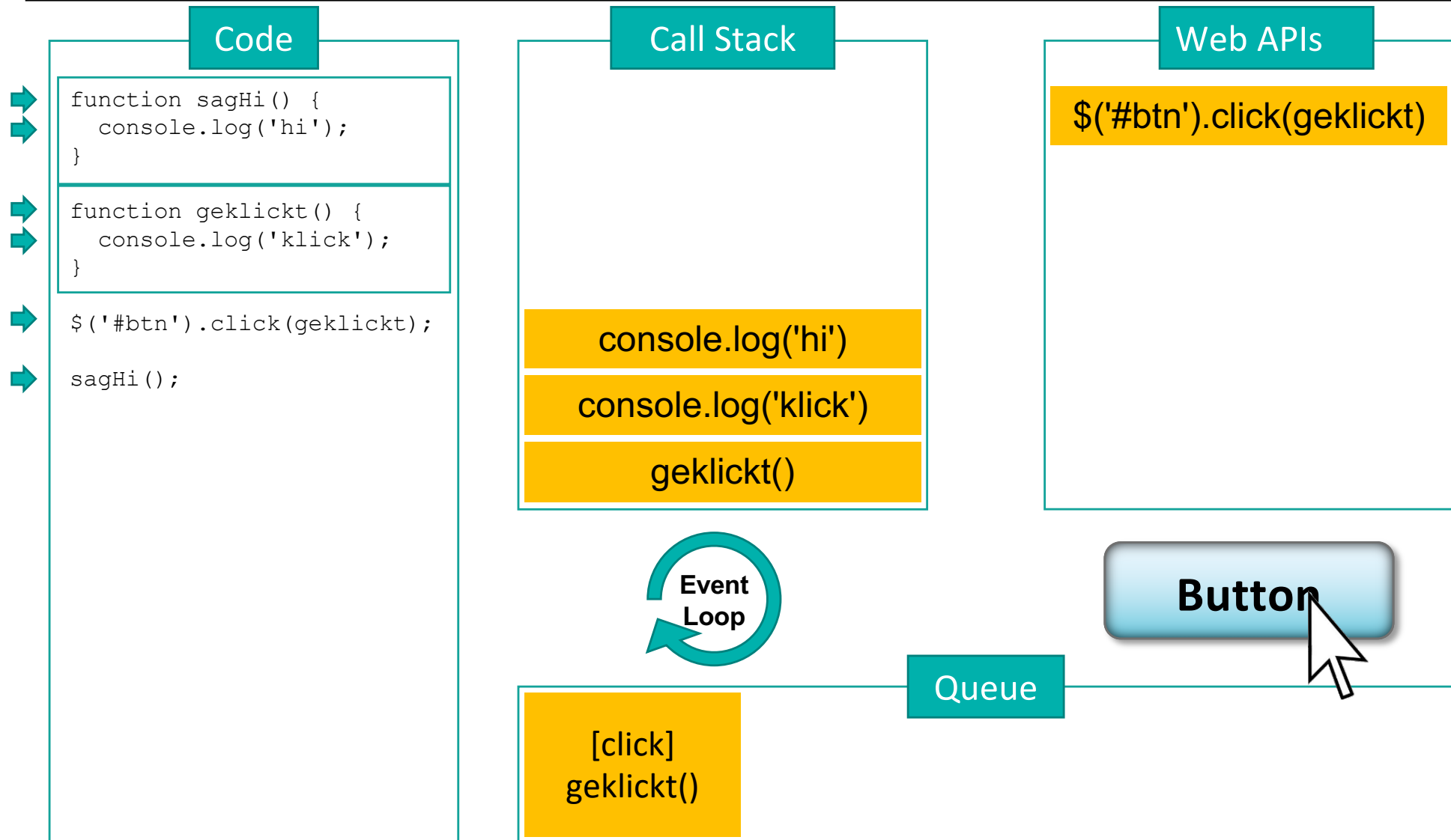
# Event Loop



# Event Loop

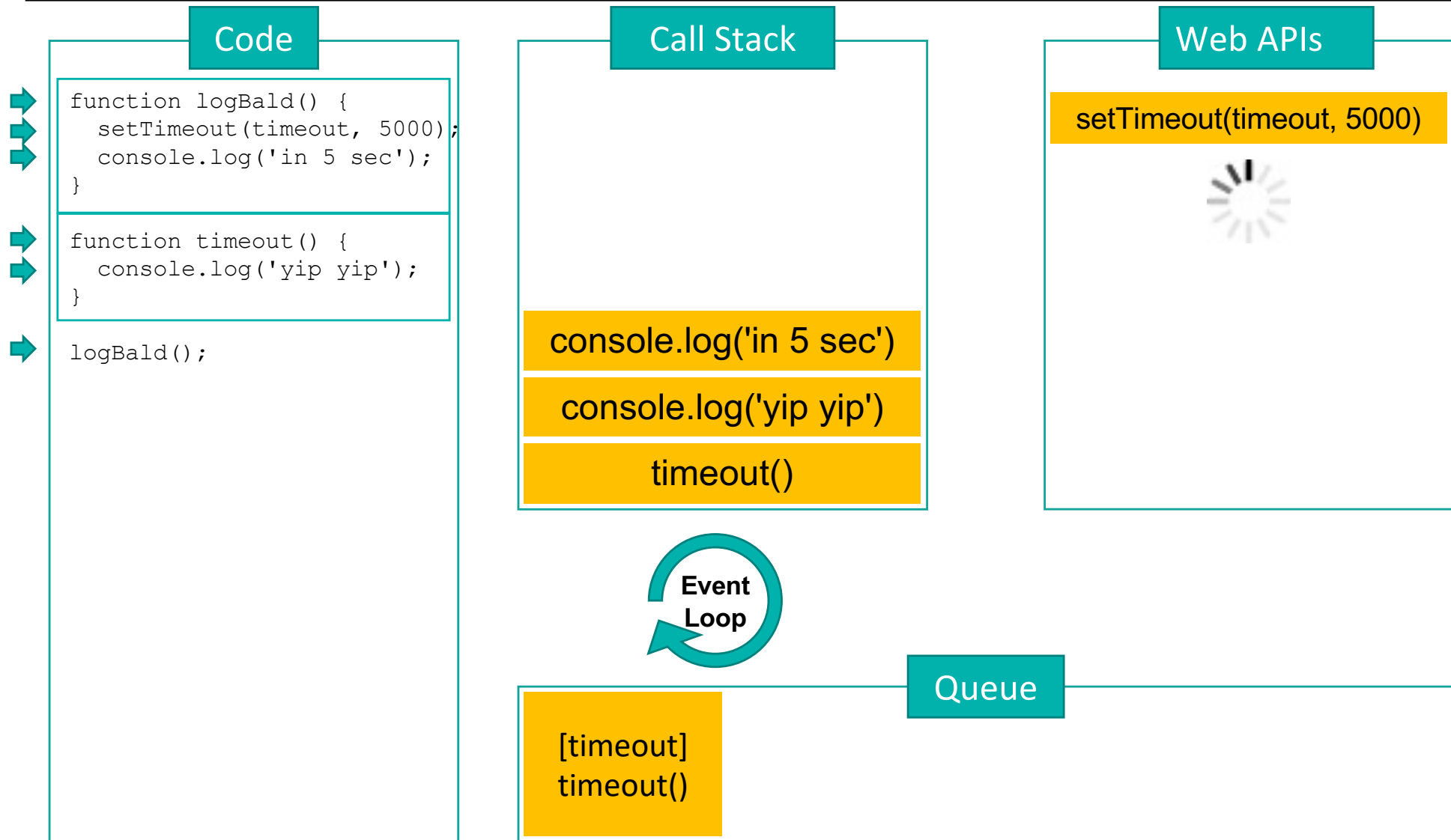


# Event Loop



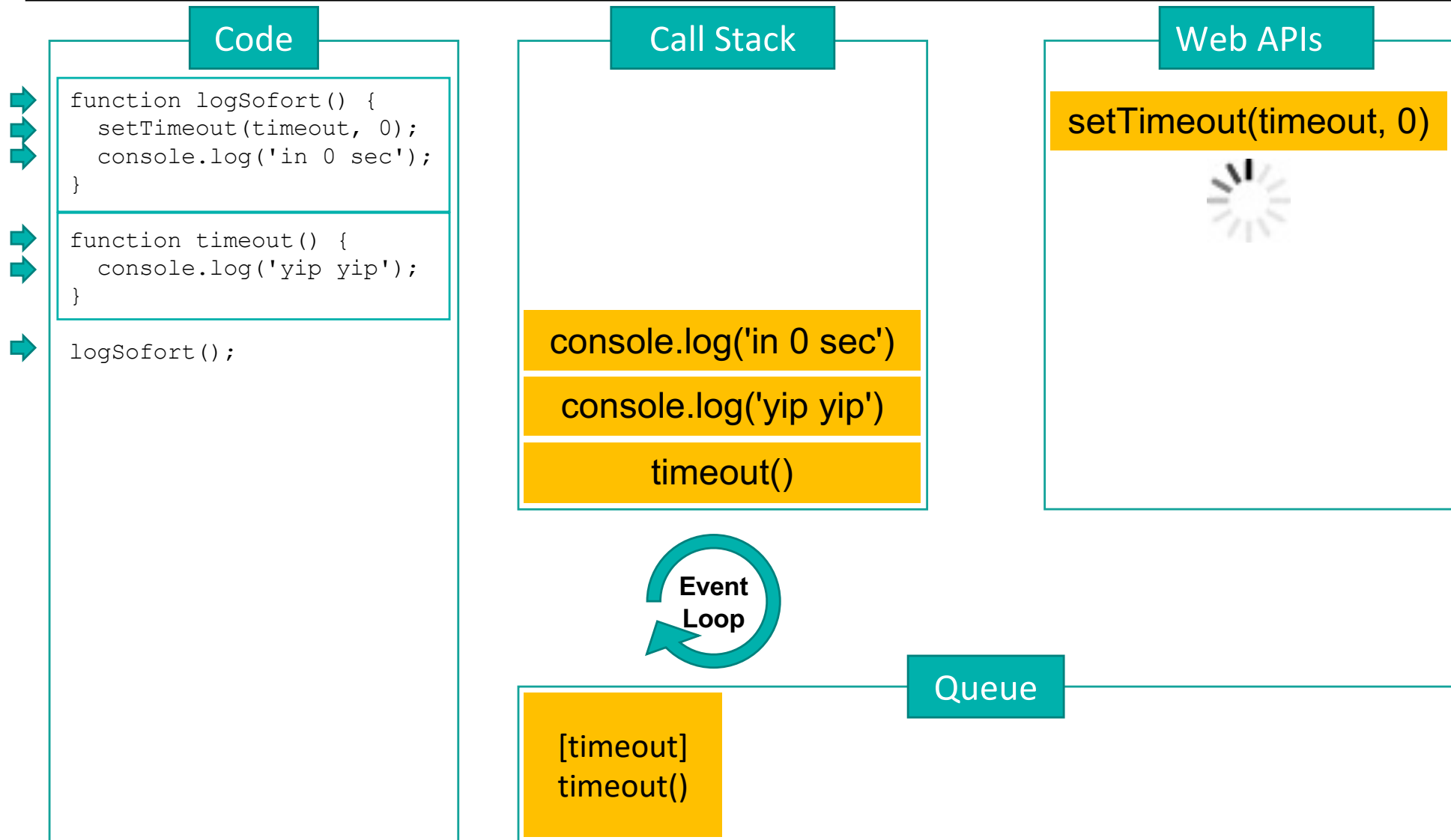
Selber ausprobieren: <http://latentflip.com/loupe/>

# Event Loop



Selber ausprobieren: <http://latentflip.com/loupe/>

# Event Loop



Selber ausprobieren: <http://latentflip.com/loupe/>

## Funktion der Event Loop

- Programm:

```
console.log("i am first");
console.log("i am second");
console.log("i am third");
```

- Ausgabe:

```
i am first
i am second
i am third
```

## Funktion der Event Loop

Programm:

```
console.log("i am first");

setTimeout(function timeout() {
 console.log("i am second");
}, 5000);

console.log("i am third");
```

Ausgabe:

```
i am first
i am third

i am second
```



## Funktion der Event Loop

Programm:

```
console.log("i am first");

setTimeout(function timeout() {
 console.log("i am second");
}, 0);

console.log("i am third");
```

Ausgabe:

```
i am first
i am third
i am second
```

setTimeout wird zwar sofort ausgeführt, aber die 3. Konsolenausgabe ist schon vorher da

## Funktion der Event Loop

- Arbeitet die Event Queue ab
  - > Wenn der Stack leer ist (also gerade nichts ausgeführt wird)
  - > Führe ersten Eintrag der Queue aus
- Erlaubt Asynchronität ohne Nebenläufigkeit
  - > Nur ein Thread für die JavaScript-Engine
  - > Implementierung des Systems um die JS-Engine herum, kann mit mehreren Threads erfolgen
  - > Probleme von parallelen Programmen werden vermieden
    - (z.B. paralleler Zugriff auf Variablen)

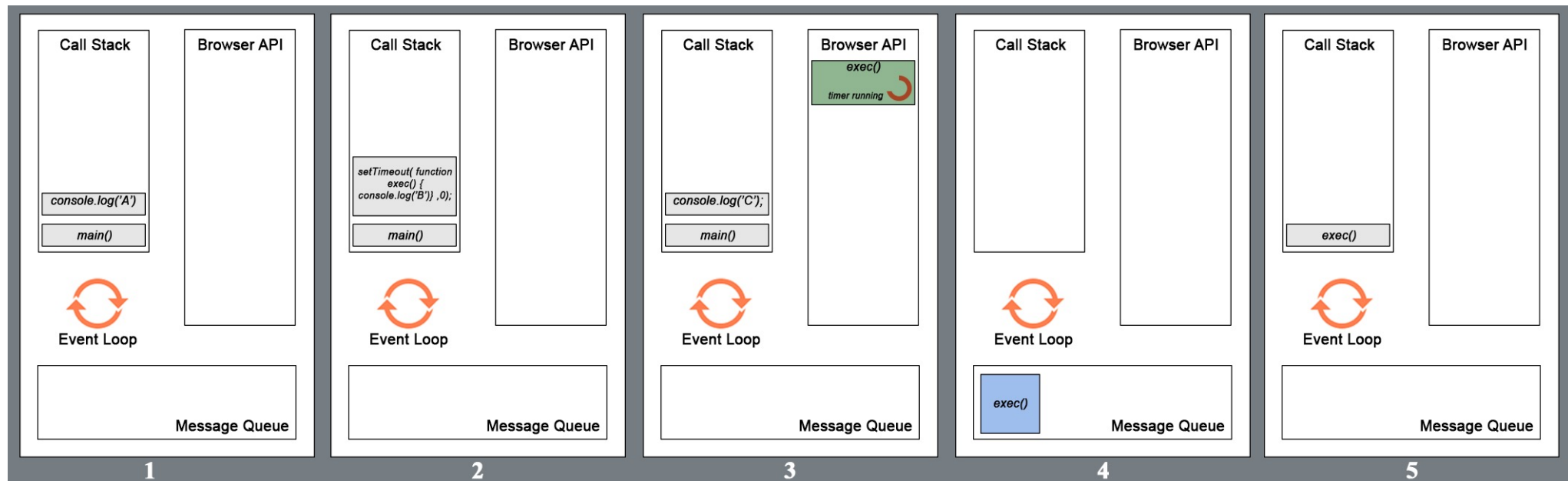
## Never block the Event Loop!

Endlosschleifen bedingen, dass keine Interaktion mit der WEB-API im Browser mehr möglich ist

# Event Loop

<https://medium.com/front-end-weekly/javascript-event-loop-explained-4cd26af121d4>

```
1 function main(){
2 console.log('A');
3 setTimeout(
4 function display(){ console.log('B'); }
5 ,0);
6 console.log('C');
7 }
8 main();
9 // Output
10 // A
11 // C
12 // B
```



# Event Loop

<https://medium.com/front-end-weekly/javascript-event-loop-explained-4cd26af121d4>

```
1 function main(){
2 console.log('A');
3 setTimeout(
4 function exec(){ console.log('B'); }
5 , 0);
6 runWhileLoopForNSeconds(3);
7 console.log('C');
8 }
9 main();
10 function runWhileLoopForNSeconds(sec){
11 let start = Date.now(), now = start;
12 while (now - start < (sec*1000)) {
13 now = Date.now();
14 }
15 }
16 // Output
17 // A
18 // C
19 // B
```

