JavaScript

Basics, Praxis und Neuerungen

Themenübersicht

Basics

- Syntax
- Funktionen & Closures
- Prototyping

JavaScript im Einsatz

- DOM
- jQuery
- AJAX
- Events

Modernes JavaScript

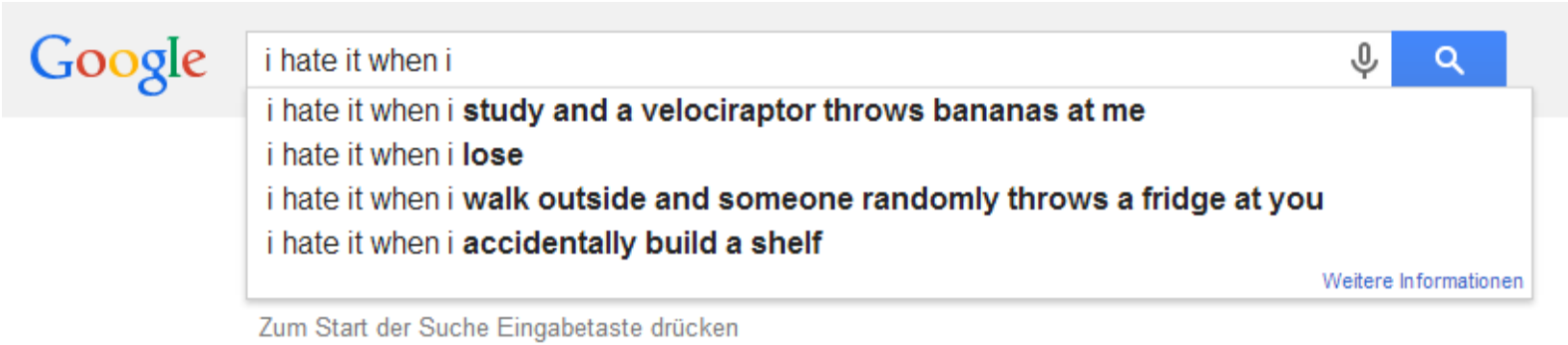
- ES6/ES7
- Event Loop
- Node.js

JavaScript

Grundlagen, Syntax, Funktionen, Closures, Prototyping, Module

Motivation

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, …

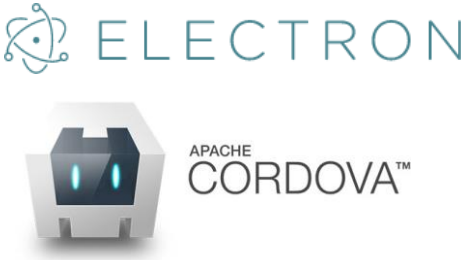
Node.js in 2009 (serverseitiges JavaScript)

- Event-basierter Ansatz eine Stärke der Sprache

Heutzutage

- Unterstützung von nahezu jedem Gerät (mit Browser)



Motivation

Trennung von Logik, Layout & Design

	Web	LaTeX	Java	Android	JavaFX	C++
Inhalt	HTML	TEX	Java	Java/XML	Java/FXML	C++
Design	CSS	CLS/STY	Java	XML	CSS	C++
Logik/Programmierung	JavaScript	TEX	Java	Java	Java	C++

- Einfache Trennung (im Vergleich zu anderen Sprachen)
- Auch für Anfänger einfach eine Anwendung zu erstellen
  - > Leider auch häufig von Anfängern programmiert
  - > Altes Wissen und alte Techniken weit verbreitet (auch auf Grund der schnellen Weiterentwicklung)

JavaScript

HTML & CSS bereits kennengelernt

- Trennung wichtig

Trennung von Inhalt und Präsentation

- Semantisches Markup
- Informationen zur Darstellung nur im Stylesheet
- Inhaltliche bzw. strukturelle Informationen (z.B. „dieser Text ist wichtig“) im HTML-Code

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

Grundlagen

Entwickelt 1995 von Brendan Eich

Basiert auf der Sprache ECMAScript

- wie z.B. auch ActionScript und JScript

Objektorientiert, aber klassenlos*

- OOP mittels Prototyping

Merkmale der Sprache

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

Ausführung des Skriptes auf dem Rechner des „Websurfers“

Grundlagen

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

- Möglichkeit auf Benutzereingaben zu reagieren ohne Server
- Manipulation des Dokuments
- Kommunikation mit dem Server möglich

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

Vorteile

- Veränderung der Webseite ohne neuladen der Seite
  - > Spart Traffic und Zeit
  - > Schnellere Interaktion mit dem Benutzer
  - > Auslagern von Berechnungen auf den Client möglich

- Typische Anwendungsfälle:
  - > Validierung von Formulareingaben vor der Übertragung zum Server
  - > Anzeige von Dialogfenstern
  - > Vorschlagen von Suchbegriffen während der Eingabe
  - > Werbung

Syntax

Beispiel 1: (Einbindung von Code)

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

Syntax

Beispiel 1: script.js

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

Ist das Dokument fertig geladen, geben wir eine Nachricht aus

- Einfachere Entwicklung mit Frameworks
- Klassische Ausgabe gibt es nicht (JavaScript-Konsole: console.log)

Syntax

Beispiel 2:

```
<!DOCTYPE html>
<html>
<head>
    <title>Skriptprogrammmierung</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

Syntax

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

Syntax
Variablen

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*

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 ebenfalls maskiert werden

Syntax
Sprachkonstrukte

Grundsätzliches bekannt aus Java (jetzt auch aus PHP!)

- if
- else if
- else

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

Syntax
Sprachkonstrukte

for-in-Schleife

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

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

- Nur mit Bedacht zu verwenden
  - > Konflikte mit Prototypen möglich sind
  - > 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)

Weitere Informationen: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Array

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
```

Syntax
Objekte & Funktionen

Objekte und Funktionen unterscheiden sich stark von vielen anderen Programmiersprachen

Wiederholung:

- Typen und Vererbung werden in den meisten Programmiersprachen durch Klassen gelöst
- JavaScript besitzt keine Klassen!*
  - > Stattdessen gibt es Konstruktorfunktionen und Prototypen für Objekte
  - > Jede Funktion hat eine prototype-Eigenschaft (genaueres folgt)

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

Syntax
Funktionen

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 deklariert werden, dann sind diese Variablen global, wurden also extern bereits erstellt
  - > Sicherstellen, dass in einem Geltungsbereich „var“ verwendet wurde
  - > Mit dem Schlüsselwort var sind diese lokal im Geltungsbereich der Funktion

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)

Anonyme Funktionen

- Schnelle Erzeugung ohne Speicherung im Geltungsbereich

- Häufig verwendet bei Eventhandlern:

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

- Und zur Kapselung des Skriptes:

```
(function() {
    var info = 'Hallo'; // Nur innerhalb sichtbar
    alert(info);
})();
// Variable info ist in diesem Geltungsbereich unbekannt
```

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
```

Syntax
Funktionen

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

Probleme durch „fehlenden" Geltungsbereich

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

```
for (var i = 0; i < 10; i++) {
    elements[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 (also i noch nicht 10 ist)!

Mögliche Lösung: Kapseln in einer Funktion

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

- Neuer Geltungsbereich mit Variable *innerI*
- Ähnliche Probleme häufig mit *this*

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

Mögliche Lösung: „let"-Schlüsselwort

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

- Variablen sind jetzt an Block gebunden (statt an Funktion!)
  - > So wie wir es auch aus anderen Sprachen kennen
- Error, wenn Variable bereits deklariert (wie man es erwarten würde)
- „let is the new var"

- Aber: Neu in ECMAScript 6 (2015)
  - > (Noch) Vorsicht was den Browser-Support angeht

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

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
```

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!

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 Strukutur
- 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/

Syntax
JSON

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

Syntax
Objekte

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
```

Syntax
Objekte - this

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 die Funktion von keinem Objekt aufgerufen, so enthält this das globale Objekt window
  > window ist das „globale" Browserobjekt

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

Syntax
Objekte - this

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
```

Syntax
Objekte - this

Unser erstes „richtiges“ Objekt (verbessert)

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

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

Syntax
Objekte - this

„Herausziehen“ von Funktionen kann zu Probleme führen

- Beispiel:

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



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

Syntax
Objekte - this

„Herausziehen“ von Funktionen kann zu Probleme führen

- Beispiel:

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

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

Syntax
Objekte - this

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

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 die Nutzung der Konstruktorfunktion ohne new zu unterbinden kann man z.B. einen Self-Invoking-Konstruktor verwenden
  - > Mit Bedacht verwenden, grundsätzlich von gutem Code ausgehen
- 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
```

Klassenlose Veerbung in JavaScript

- Über Prototypen (Objekte erben von Objekten)
  - > Beim Erstellen eines neuen Objektes mittels „new“ wird der Prototyp der Konstruktorfunktion genutzt
  - > Im Prototypobjekt können Attribute und Methoden gespeichert werden, auf die auch neu erzeugte Objekte zugreifen können, als ob sie unmittelbar Eigenschaften von ihnen selbst sind.
  - > Objekte können sich also Attribute und Funktionen teilen.
  - > Auch Prototypen haben einen Prototypen
    - – Es ergibt sich eine „Prototype-Chain“

Zugriff auf ein Attribut – Ablauf

- Beispiel: `alert(obj.a);`
  - > Besitzt obj ein Attribut a? Falls ja, nehmen wir den Wert
    - – Sonst: Besitzt der Prototype on obj ein Attribut a? Falls ja, nehmen wir den Wert
      - • Sonst: Besitzt der Prototype des Prototypes von obj ein …
  - > Solange durchführen bis ein Attribut a gefunden ist (sonst undefined)

Prototype und __proto__

- [Funktion].prototype
  - > „Magisches" Attribut prototype, das jede Funktion besitzt
  - > Grundlage für die Vererbung
  - > Enthält gemeinsam nutzbare Methoden und Attribute, auf die die neu erzeugten Objekte zugreifen können
- [Objekt].__proto__
  - > „Magisches" Attribut __proto__, das jedes Objekt besitzt
  - > Referenziert die Eigenschaft prototype der Funktion (nach Erzeugung mittels new)

Zugriff auf ein Attribut – Ablauf

- Beispiel: `alert(obj.a);`
  - > Besitzt obj ein Attribut a? Falls ja, nehmen wir den Wert
    - – Sonst: Besitzt der obj.__proto__ ein Attribut a? Falls ja, nehmen wir den Wert
      - • Sonst: Besitzt obj.__proto__.__proto__ ein …
  - > Solange durchführen bis ein Attribut a gefunden ist (sonst undefined)

Weitere Informationen: https://developer.mozilla.org/de/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

Syntax
Vererbung

```
function Fahrzeug(speed) {
    this.speed = speed; // Eigenschaften werden an das
    this.distance = 0;  // Objekt selber gebunden
};

// Funktion für Objekte des Typs Fahrzeug
Fahrzeug.prototype.go = function(times) {
    this.distance += this.speed * times;
};

// porsche.__proto__ === Fahrzeug.prototype (auch für opel)
var porsche = new Fahrzeug(260);
var opel    = new Fahrzeug(140);
opel.go(2);

alert(opel.distance);                    // 280
```

Syntax
Vererbung

```
function Fahrzeug(speed) {
    this.speed = speed;
    this.distance = 0;
};
Fahrzeug.prototype.go = function(times) {
    this.distance += this.speed * times;
};
var fahrzeug1 = new Fahrzeug(120);
fahrzeug1.go(2);

function Auto(speed, fabrikat) {
    Fahrzeug.apply(this, arguments);
    this.fabrikat = fabrikat;
}

// Wir erzeugen ein Objekt, dass die prototype-Funktionen von Fahrzeug enthält
Auto.prototype = new Fahrzeug();
Auto.prototype.hupen = function() { // Wir hängen selber eine weitere Funktion an
    alert('Mööööp!');
}

var auto = new Auto(150, 'Corsa');
auto.hupen();          // Gibt aus: 'Mööööp!'
auto.go(1);
alert(auto.distance); // 150
```

Syntax
Vererbung

Auch mehrfache Vererbung ist mit JavaScript möglich

Meist werden stattdessen aber Frameworks verwendet, mit Hilfe derer die Nutzung einfacher ist

Auch klassenähnliche Konzepte sind möglich

- Beispiel: „Simple JavaScript Inheritance“ von John Resig
- http://ejohn.org/blog/simple-javascript-inheritance/

Syntax
Vererbung

```
var Person = Class.extend({ // Per new Person(boolean) nutzen
    init: function(isDancing){
        this.dancing = isDancing;
    },
    dance: function(){
        return this.dancing;
    }
});

var Ninja = Person.extend({ // Kann per new Ninja() erstellt werden
    init: function(){
        this._super( false );
    },
    dance: function(){
        return this._super();
    },
    swingSword: function(){
        return true;
    }
});
```

Das ist ein Beispiel für die Nutzung eines simplen Frameworks!

- Nicht mit den Standardmöglichkeiten verwechseln!

Object.create

- Einfache Möglichkeit Prototyping zu nutzen
- Beispiel 1

```
var foo = {
    eins : 1,
    zwei : 2,
    pi : 3.14159,
};

var bar = Object.create(foo); //Objekt mit foo als Prototyp
bar.drei = 3;

// Eigenschaften von bar:
bar.eins;   // 1 (vom Prototyp geerbt)
bar.zwei;   // 2 (vom Prototyp geerbt)
bar.drei;   // 3
bar.pi;     // 3.14159 (vom Prototyp geerbt)
```

Object.create

- Einfache Möglichkeit Prototyping zu nutzen
- Beispiel 2

```
var foo = {
    eins : 1,
    zwei : 2,
    pi : 3.14159,
};

var bar = Object.create(foo);//Objekt mit foo als Prototyp
bar.drei = 3;
bar.pi   = 4;
// Eigenschaften von bar:
bar.eins;   // 1 (vom Prototyp geerbt)
bar.zwei;   // 2 (vom Prototyp geerbt)
bar.drei;   // 3
bar.pi;     // 4 (Eigenschaft vom Prototypen überdeckt)
```

Object.create: Beispiel 2

```
var obj1 = {
    a : 1
};
var obj2 = Object.create(obj1);
obj2.b = 2;
var obj3 = Object.create(obj2);
obj3.c = 3;


obj1;   // {a: 1 }
obj2;   // {a: 1, b: 2 }
obj3;   // {a: 1, b: 2, c: 3 }
```

Object.create

```
Object.create(proto [, propertiesObject])
```

- Erstellt ein Objekt mit *proto* als __proto__
  - > Wird *null* übergeben, so wird ein Objekt ohne Prototyp erstellt
- Per *propertiesObject* können Eigenschaften des Objektes, sowie Getter- und Setter-Funktionen bestimmt werden
  - > Hier zunächst nicht weiter berücksichtigt

Welche Möglichkeiten kennen wir nun Objekte zu erstellen?

- Object.create

```
var obj = Object.create(null);
obj.foo = 1;
obj.bar = 2;
```

- JSON-like-Notation

```
var obj = {
    foo : 1,
    bar : 2
};
```

- Konstruktorfunktion

```
var obj = new Object();
obj.foo = 1;
obj.bar = 2;
```

Syntax
Module

```
var maxSpeed = 150;
function getTotalDistance() {
    return (corsa.distance + porsche.distance);
}
var corsa = {   // ...
};
var porsche = { // ...
};
function init() { // ...
}
init();
```

- Schlechte Kapselung
  - > Alle Objekte liegen im globalen Scope
  - > „private" Funktionen und Variablen fehlen

Syntax
Module

```
var myCars = (function() {     // Nur Objekt myCars liegt im globalen Scope
    var maxSpeed = 150;                    // private Variable
    function getTotalDistance() {
        return (corsa.distance + porsche.distance);
    }
    var corsa = {   // ...
    };
    var porsche = { // ...
    };
    function init() { // ...                // private Funktion
    }
    init();
    return {                               // Zugriff von außen:
        getTotalDistance : getTotalDistance,// myCars.getTotalDistance()
        car1 : corsa,                      // myCars.car1 zeigt auf corsa
        car2 : porsche                     // myCars.car2 zeigt auf porsche
    };
})();
```

Modularisierung mittels eigenem Scope

- Private Variablen und Funktionen möglich
- Kein Konflikt mit anderen Variablen im globalen Geltungsbereich
- Trennung von mehreren Skripten „sauberer“
- Öffentliche Funktionen und Objekte mittels return zurückgeben
- Meist wird das gesamte Skript auf eine einzige Variable „reduziert“
  - > Hier: myCars

JavaScript im Einsatz

DOM, jQuery, AJAX, Events

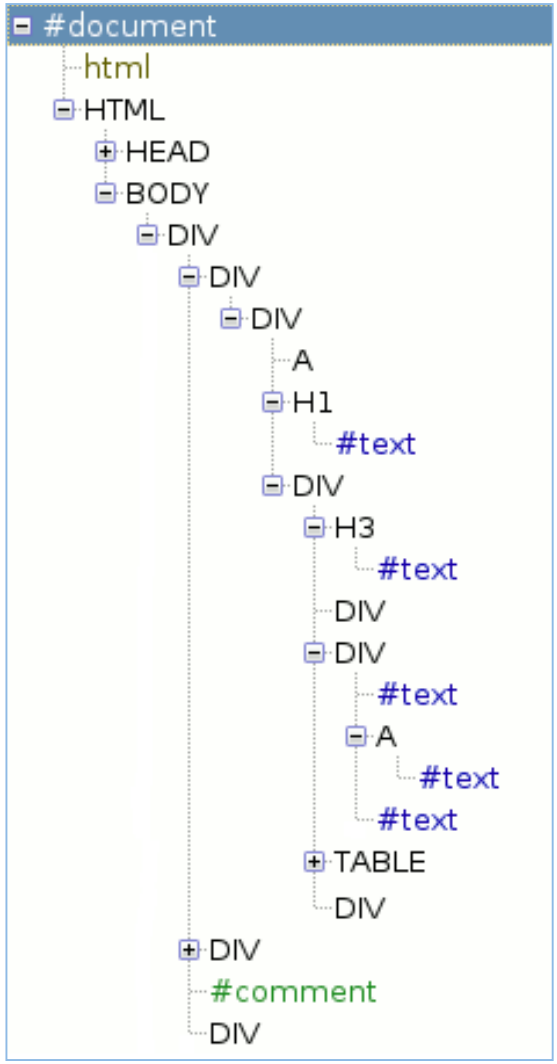
Document Object Model

Bietet Zugriff auf alle Tags und Attribute der Webseite

- Baumstruktur
- Erlaubt Manipulation des vorliegenden HTML-Dokuments

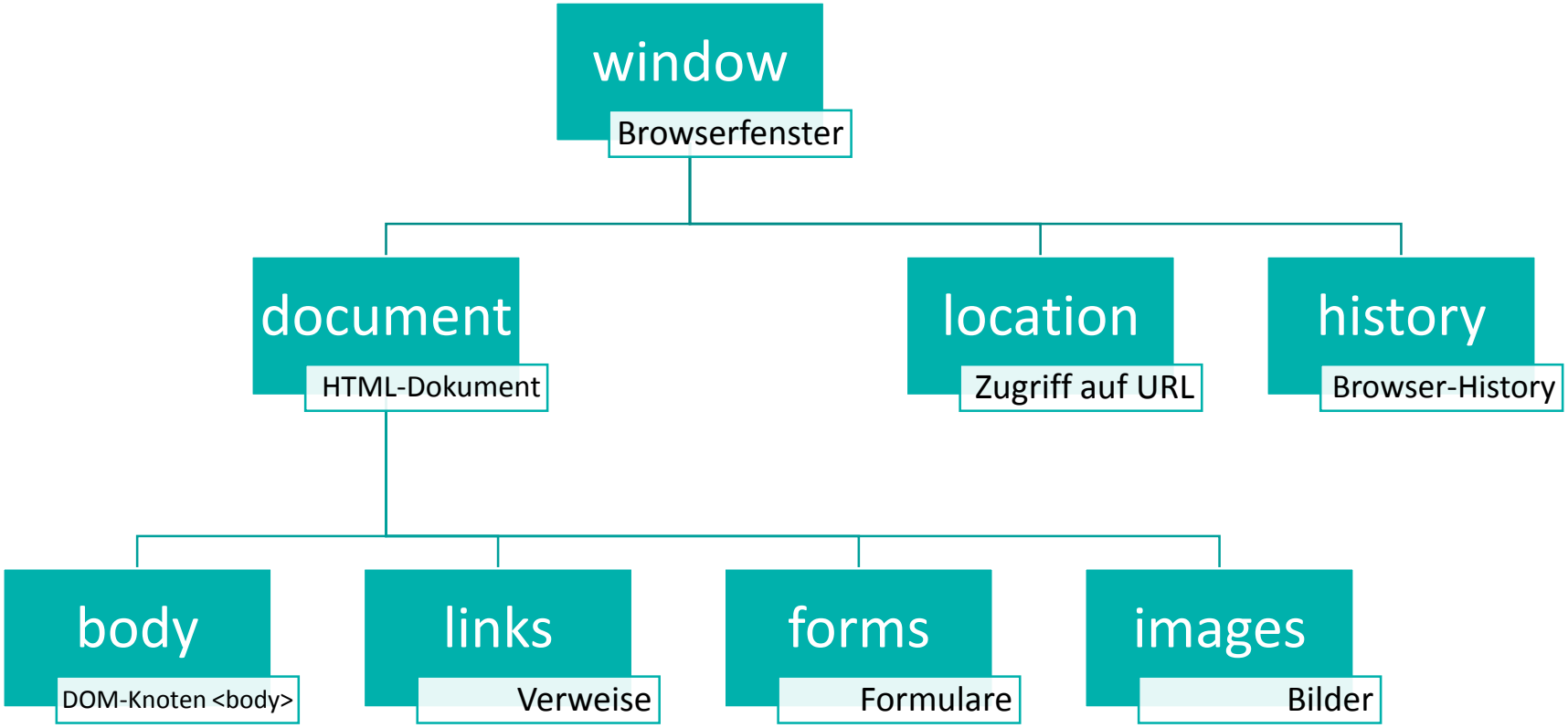
„Dunkle Vergangenheit“

- JavaScript war zwar standardisiert, nicht aber der Zugriff auf das HTML-Dokument
  - > Jeder Browser regelte den Zugriff anders
- Mittlerweile standardisiert
  - > Teilweise immer noch Unterschiede



Document Object Model

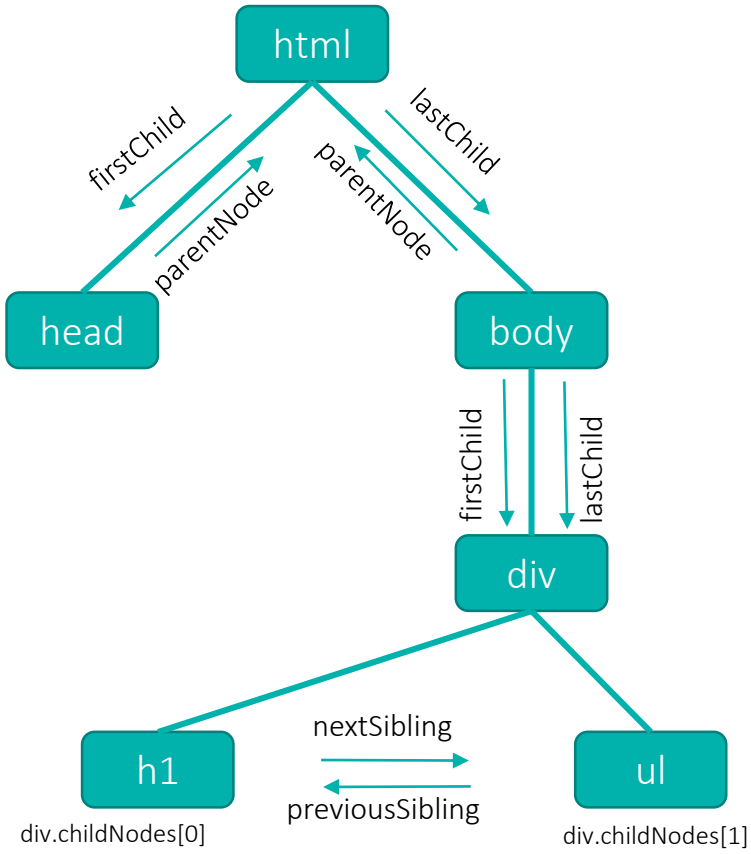
Ausschnitt:



Document Object Model

Zugriff auf DOM-Knoten

- über ihre id (einfachste Möglichkeit) oder per Klasse/Tag/Name
- über die numerische Ordnung in der Hierarchie durch Beschreiten des entsprechenden Feldes, das die Objekte beinhaltet
- über die Position im DOM-Baum und dem entsprechenden Navigieren (parentNode, previousSibling, nextSibling, firstChild, lastChild, childNodes)



Document Object Model

Beispiel

- Wir erkennen ein Element mit Namen div mit der id “meindiv”
- Es beinhaltet ein Textelement, welches über div.childNodes[0] referenziert werden kann (childNode ist ein Feld in dem alle Kindelemente gelistet werden)
- Der Text “Dies ist ein einfacher Text” ist somit kein Wert des div-Elements, vielmehr ist es der Wert des ersten und einzigen Kindelementes von div

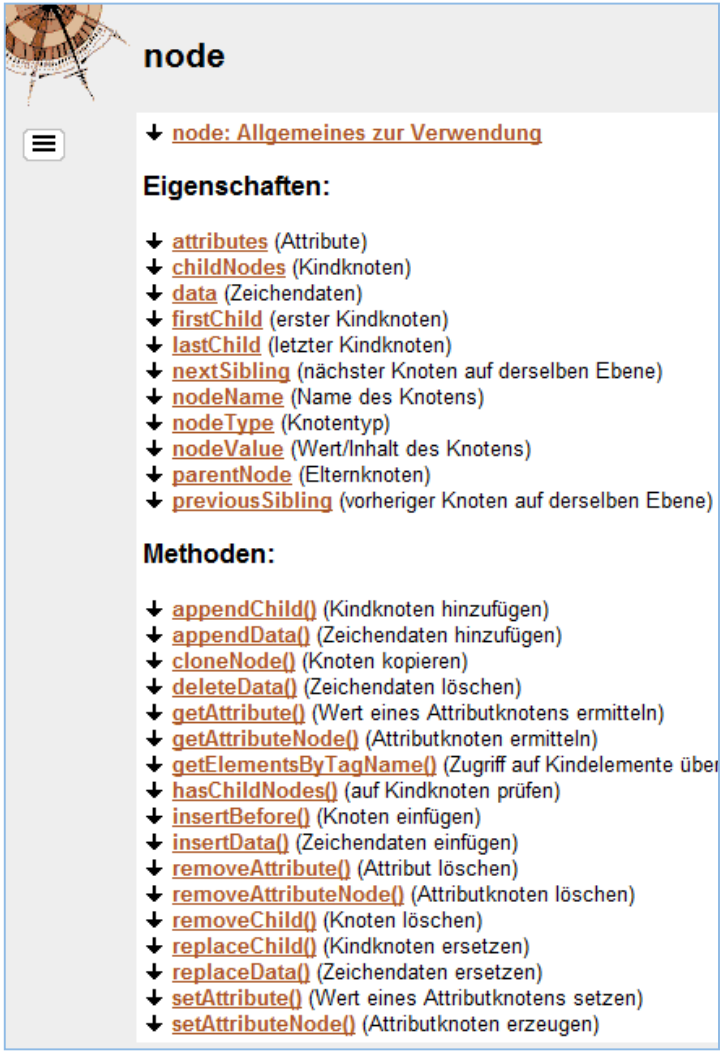
```
<div id="meindiv">
Dies ist ein einfacher Text
</div>
```

Document Object Model

Zugriff auf Elemente

```
document.getElementById(id)
```

- Liefert DOM-Knoten mit der ID id zurück (oder null)
- Viele andere Funktionen um Zugriff auf DOM-Knoten zu erhalten und um diese zu manipulieren



node

- node: Allgemeines zur Verwendung

Eigenschaften:

- attributes (Attribute)
- childNodes (Kindknoten)
- data (Zeichendaten)
- firstChild (erster Kindknoten)
- lastChild (letzter Kindknoten)
- nextSibling (nächster Knoten auf derselben Ebene)
- nodeName (Name des Knotens)
- nodeType (Knotentyp)
- nodeValue (Wert/Inhalt des Knotens)
- parentNode (Elternknoten)
- previousSibling (vorheriger Knoten auf derselben Ebene)

Methoden:

- appendChild() (Kindknoten hinzufügen)
- appendData() (Zeichendaten hinzufügen)
- cloneNode() (Knoten kopieren)
- deleteData() (Zeichendaten löschen)
- getAttribute() (Wert eines Attributknotens ermitteln)
- getAttributeNode() (Attributknoten ermitteln)
- getElementsByTagName() (Zugriff auf Kindelemente über
- hasChildNodes() (auf Kindknoten prüfen)
- insertBefore() (Knoten einfügen)
- insertData() (Zeichendaten einfügen)
- removeAttribute() (Attribut löschen)
- removeAttributeNode() (Attributknoten löschen)
- removeChild() (Knoten löschen)
- replaceChild() (Kindknoten ersetzen)
- replaceData() (Zeichendaten ersetzen)
- setAttribute() (Wert eines Attributknotens setzen)
- setAttributeNode() (Attributknoten erzeugen)

Quelle: http://de.selfhtml.org/javascript/objekte/node.htm

Document Object Model

Beispiel

```
window.onload = function() {
    document.getElementById('btn').onclick = function() {
        this.innerHTML = 'geklickt!';

        // Neues h1-Element erstellen mit Inhalt "Klick"
        var h1 = document.createElement('h1');
        var text = document.createTextNode('Klick!');
        h1.appendChild(text);

        // In den body einfügen
        document.body.appendChild(h1);
    }
};
```

Document Object Model

Browser-Engines

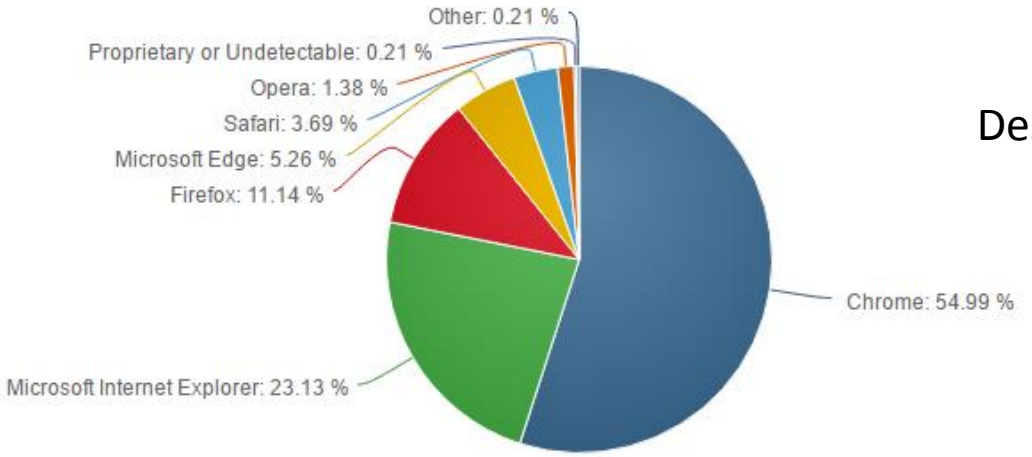
- WebKit   
  - Blink (Fork von WebKit)   
- Gecko  
- Trident  
  - EdgeHTML (Fork von Trident)  

- Jede Engine verhält sich ein wenig anders
  > Vor allem veraltete Browser!)

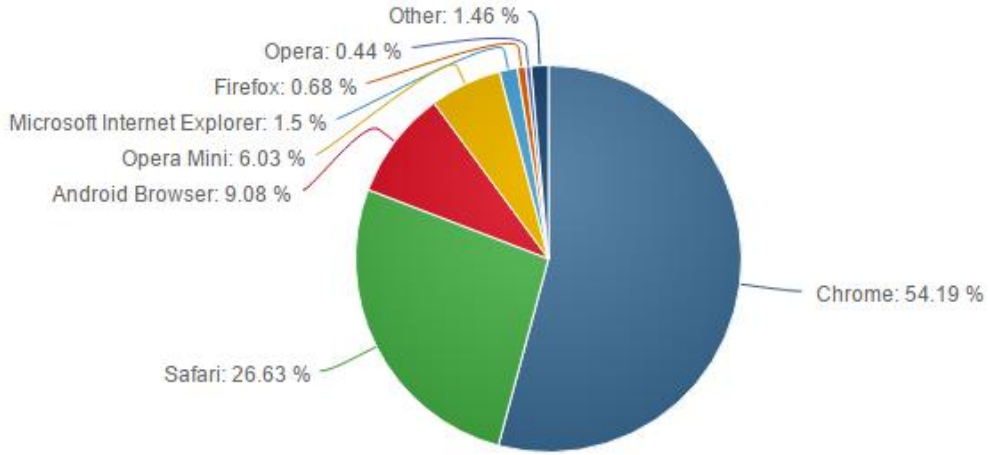
DOM-Manipulation ist kompliziert

- Daher nutzen wir jQuery!

Marktanteile Browser (weltweit)



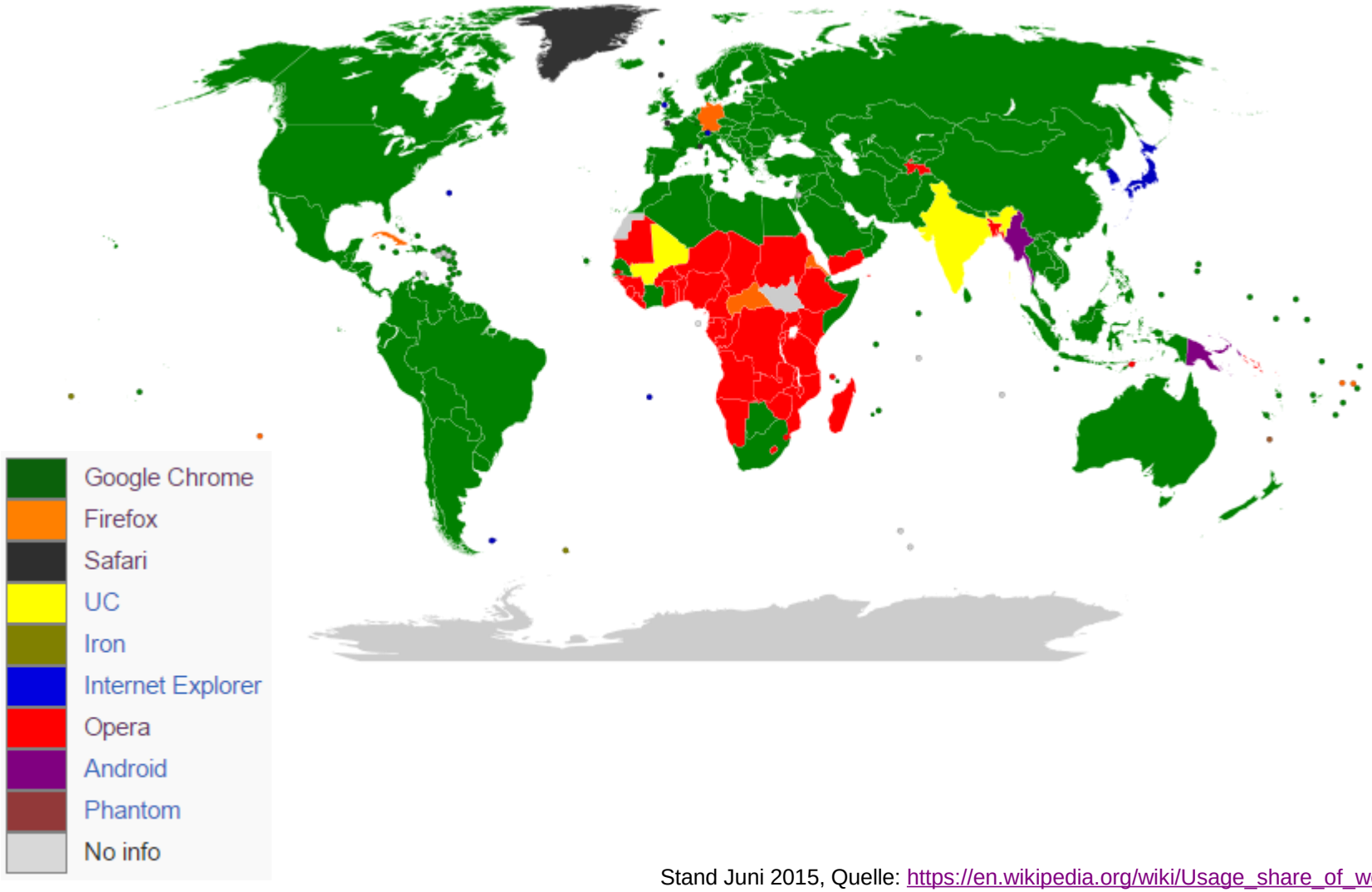
Desktop Market Share



Mobile Market Share

Stand November 2016, Quelle: https://netmarketshare.com/

Marktanteile Browser (weltweit)



Stand Juni 2015, Quelle: https://en.wikipedia.org/wiki/Usage_share_of_web_browsers

jQuery

JavaScript Framework

- Vereinfacht die clientseitige Entwicklung

Zahlreiche andere Frameworks

- Oft mit anderem Schwerpunkt
- Beispiele:
  - > Mootools
  - > Prototype
  - > Dojo
  - > Sencha Ext JS
  - > …

jQuery

Funktionalität: Vereinfachung

- Eventhandling
- DOM-Manipulation (browserübergreifend!)
- AJAX-Requests
- Hilfsfunktionen
- Animationen und Effekte
- Erweiterbarkeit durch zahlreiche Plugins (z.B. jQuery UI zur Darstellung von Oberflächen)

Download und Dokumentation:

- http://jquery.com/

jQuery

Arbeiten mit jQuery

- jQuery vor anderen Skripten einbinden (wie ein normales Skript)

```
<script type="text/javascript" src="jquery.js"></script>
```

- jQuery oder/und $ ist dann im globalen Scope verfügbar

- 2 grundsätzliche Arbeitsweisen
  - „$.function“ (führt jQuery-Funktion aus)
    - Beispiel:

```
$.ajax(/* ... */);
```

  - „$-Factory“ (erzeugt ein jQuery-Objekt auf dem gearbeitet wird)
    - Beispiel:

```
$('#btn').hide();
```

Besonderheiten

- Viel Funktionalität in einer Funktion
  - > Die meisten jQuery-Funktionen bieten eine Vielzahl von Aufrufmöglichkeiten an
  - > Funktionsweise ist oft abhängig davon was für ein Typ übergeben wird

- Chaining
  - > Funktionen geben (meistens) das Objekt zurück auf dem gearbeitet wird
  - > Beispiel:

```
$('#btn')
    .html('Neuer Text!')
    .addClass('important')
    .removeClass('hl')
    .show();
```

jQuery
DOM-Zugriff

Auslesen eines Elements:

```
var button = $('#btn');
```

- Liefert DOM-Knoten mit der ID „btn“ zurück, „verpackt“ als jQuery-Objekt
  - > Verwendung von jQuery-Objekten statt „rohen“ Daten aus dem DOM
- Syntax wie bei CSS-Selektoren
  - > Informationen zu Selektoren in der CSS-Einführung

Erstellen eines neuen DOM-Knotens

```
var h1 = $('<h1>Willkommen!</h1>');
```

- $-Funktion erstellt auch DOM-Knoten

Knoten in das DOM einfügen

```
$(document.body).append(h1);
```

- Erstellt jQuery-Objekt aus body-Knoten und fügt die Überschrift hinzu

Knoten manipulieren

- Beispiele:

```
$('#btn')
    .html('Neuer Text!')           // HTML des Knotens setzen
    .addClass('important')         // (CSS-)Klasse hinzufügen
    .removeClass('hl')             // (CSS-)Klasse entfernen
    .append('<p>Mehr Text!</p>')   // HTML-Knoten anhängen
    .prop('disabled', true)        // (DOM-)Property setzen
    .attr('id', 'btn2');           // Tag-Attribut setzen
```

jQuery
Event Handling

Bisher:

```
document.getElementById('#btn')
    .onclick = function() { /* ... */ }
```

jQuery-Syntax:

```
$('#btn').click(function() {
    // ...
});
```

„onload"-Handler:

```
$(document).ready(function() { /* ... */ });
// Kürzer:
$(function() { /* ... */ });
```

jQuery
Vergleich DOM-Zugriff

„Vanilla"-JavaScript

```
window.onload = function() {
    document.getElementById('btn').onclick = function() {
        this.innerHTML = 'geklickt!';

        // Neues h1-Element erstellen mit Inhalt "Klick"
        var h1 = document.createElement('h1');
        var text = document.createTextNode('Klick!');
        h1.appendChild(text);

        // In den body einfügen
        document.body.appendChild(h1);
    }
};
```

jQuery
Vergleich DOM-Zugriff

Beispiel

```
$(function() {
    $('#btn').click(function() {
        $(this).html('geklickt!');

        // Neues h1-Element erstellen mit Inhalt "Klick"
        $('<h1>Klick!</h1>')
            .appendTo(document.body); // In body einfügen
    });
});
```

Ablauf

- Benutzeraktion erzeugt JavaScript-Aufruf
- JavaScript erzeugt Daten
- Daten werden asynchron (im Hintergrund) versendet

Typische Anwendungsgebiete

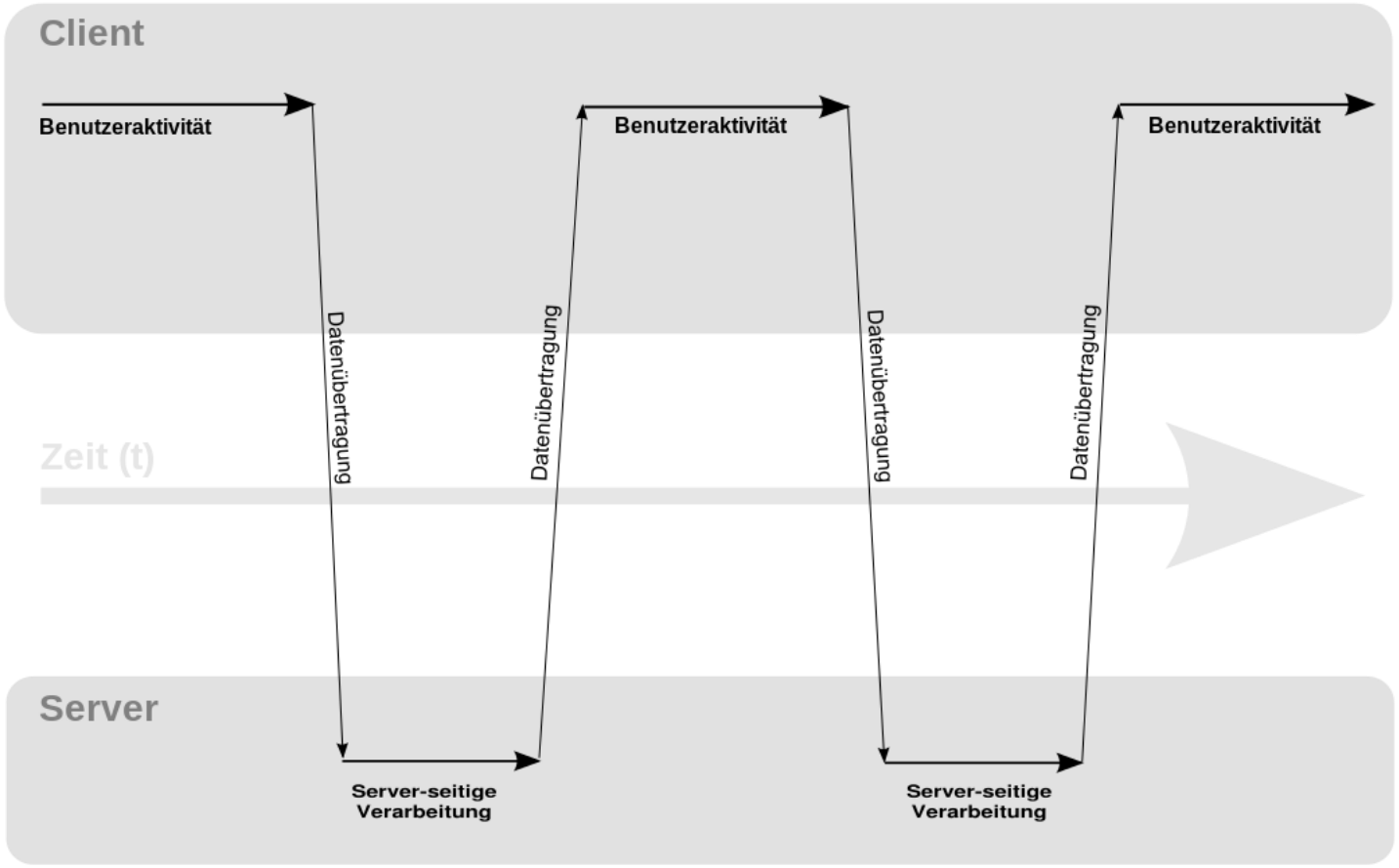
- Vorschläge für Suchbegriffe
- Webbasierte Anwendungen (Maps, Textverarbeitung, …)
- Cloud-Anwendungen (SaaS)

Auch ohne jQuery möglich!

- Mittels XMLHttpRequest (kein direkter Zugriff auf klassische Sockets)
- jQuery erleichtert das

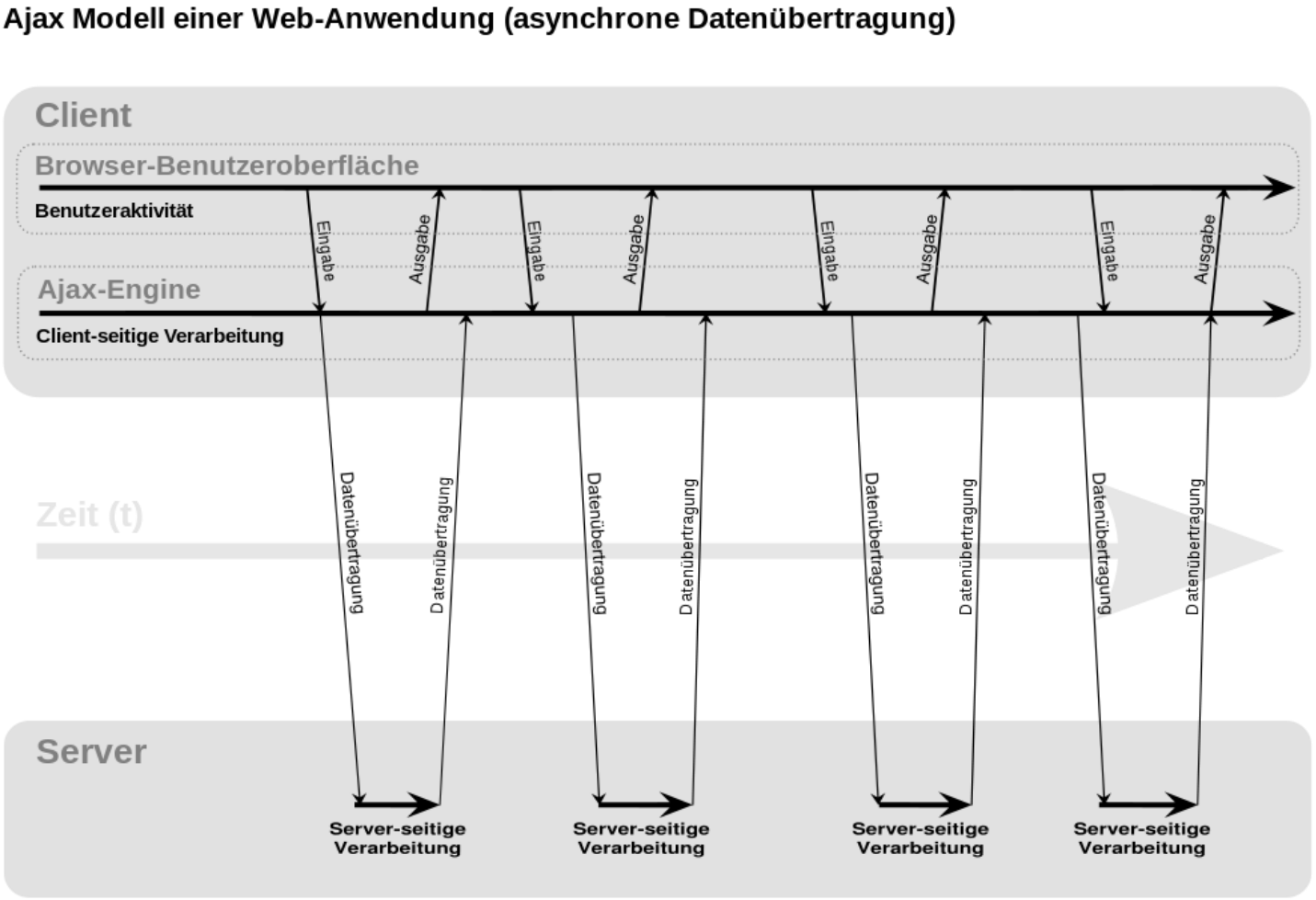
AJAX
Klassisches Abarbeitungsmodell

Klassisches Modell einer Web-Anwendung (synchrone Datenübertragung)



Quelle: Jesse James Garett, Ajax: A New Approach to Web Applications, adaptive path publications, February 2005

AJAX Abarbeitungsmodell



Quelle: Jesse James Garett, Ajax: A New Approach to Web Applications, adaptive path publications, February 2005

Vorteile

- Keine Auswirkungen auf die Darstellung der Seite
- Verringerte Serverlast
- Erhöhte Benutzerfreundlichkeit
  - > Beispiel: Anzeigen eines Ladeindikators möglich

Nachteile

- Bruch mit klassischen Technologien
  - > Zurück-Button des Browser funktioniert nicht
  - > Komplexität der URL-Ressource hoch, ggf. fehlende Eindeutigkeit (von daher auch Probleme mit Bookmarks)
  - > Benutzerempfinden bzgl. der Rückmeldungen hängt stark von der Programmierung ab
  - > „Suchmaschinenlesbarkeit“ bzw. Zugriff ohne JavaScript

Beispiel

```
$.ajax('time.php')
    .done(function(data) {
        alert(data);
    })
    .fail(function() {
        alert('error');
    });
```

- Gibt den nachgeladenen Text zurück oder im Fehlerfall die Nachricht „error“
- Zahlreiche andere Methoden um Daten nachzuladen (oftmals fast equivalent)
  - $.get()
  - $.post()

Weitere AJAX-Funktionen: https://api.jquery.com/category/ajax/

Austausch zwischen Server und Client meist über festgelegte Datenformate

- XML (Extensible Markup Language)
  - > Rein textbasiert, keine echte Typisierung
  - > Teilweise unklar ob Sachen ein Attribut oder Tag sein sollten
  - > Selbstbeschreibend, von daher generell viel Overhead
    - – Schließende Tags
    - – Arrays

- JavaScript Object Notation (JSON)
  - > Typisierung (jedoch mit wenig Datentypen)
  - > Keine Unterscheidung in Attribut und Tag
  - > Einfach zu lesen
  - > Kompaktheit, von daher wenig Overhead
  - > Besser für den Austausch in AJAX geeignet

JavaScript Objekt zu JSON bzw. umgekehrt

- JSON.stringify(obj) erzeugt JSON-String
- JSON.parse(string) erzeugt aus JSON-String ein JavaScript-Objekt

Übergabe von PHP an JavaScript (PHP-Funktion):

```
string json_encode($value)
```

- Liefert JSON-Represantation des Wertes zurück
- Einfache Übergaben von komplizierten Objekten von PHP zu JavaScript

Übergabe von JavaScript an PHP:

```
mixed json_decode($json)
```

- Dekodiert JSON-String in eine PHP-Variable

Weitere Informationen: http://json.org/, http://php.net/manual/de/book.json.php

Modernes JavaScript

ES6/ES7, Event Loop, Node.js

ECMAScript

JavaScript basiert auf ECMAScript-Standard

- Ständige Weiterentwicklung durch Gruppe aus Browserherstellern
  - > Meist oberstes Gebot: Abwärtskompatibilität
- JavaScript erweitert ECMAScript um Features, die nicht im Standard vorgesehen sind

- Heute werden Features oft in Browsern integriert bevor der Standard diese überhaupt vorsieht
  - > Erlaubt einfachere Erkennung von Problemen bevor der Standard festgehalten wird
  - > Aber Entwickler müssen nicht standardisierte APIs benutzen

Entwicklung

Edition	Release	Name
1	1997	Zuerst „Mocha“, dann „LiveScript“, dann „JavaScript“ genannt
2	1998	Kleinere Änderungen
3	1999	Diverse Erweiterungen (z.B. Regular Expressions)
4	-	Nie veröffentlicht
5	2009 (!)	Einführung des strict modes, Fokus auf Kompatibilität
6	2015	Klassen, Arrow functions, Viele Erfahrungen aus Dialekten (z.B. CoffeeScript) übernommen
7	2016	Weiterentwicklung der Sprache
8	2017	Aktuell in Arbeit

Veröffentlicht im Dezember 2009

- Bedeutet nicht, dass Features ab diesem Datum genutzt werden können
- Implementierung in „alten" Browsern nicht vorhanden (z.B. IE9)
  - > Evtl. testen ob der jeweilige Browser die Funktion unterstützt
  - > Skript, das die API teilweise nachbildet: *es5-shim.js*

Interessante Neuerungen

- Strict Mode
- Getter und Setter
- Function.prototype.bind
- Array.prototype.(forEach|map|indexOf|filter|reduce|…)

caniuse.com

Schlechte Anweisungen werfen Fehler

- Einfacheres Schreiben von „sicheren“ Anwendungen
- Vergessenes „var“ vor einer Variablen wirft Fehler
- Syntaxfehler werden schneller erkannt
  - > Beispiel: Doppelte Belegung einer Property

Strict Mode aktivieren

```
"use strict";
noVarVariable = 42; // throws a ReferenceError
```

- Einfacher String mit „use strict“ zu Beginn des Skriptes führt zur Aktivierung
  - > Alternativ auch zu Anfang einer Funktion möglich
  - > Alte Browser ignorieren den Strict Mode

Getter und Setter für Objekte

- Beispiel:

```
var auto = {};

Object.defineProperty(auto, 'speed', {
    set: function(x){
        alert('Schneller!');
    }
});

auto.speed = 120;
```

ECMAScript 5
Function.prototype.bind

Rückblick:

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

- Ruft Die Funktion *go* des Objektes *auto* auf und setzt den this-Zeiger auf den Button
- Insgesamt sorgt das this-Objekt häufig für Verwirrung (und Fehler)

Neue Lösung per bind:

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

Sorgt dafür, dass Konflikte mit dem this-Zeiger einfacher verhindert werden können.

- Der this-Zeiger ist nun auf das Auto gesetzt (dem Argument)

Weitere Informationen: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

Neue Funktionen um einfacher mit Arrays zu arbeiten

- Beispiele:

```
[1,2,3].forEach(function(value, index) {
    console.log(value + ' (index: ' + index + ')');
});

var newArray = [1,2,3].map(function(value) {
    return value * 3;
}).filter(function(value) {
    return (value > 5);
});
// newArray = [6, 9]

if (newArray.indexOf(6) !== -1) {
    console.log('6 ist drin!');
}
```

Weitere Informationen: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

ECMAScript 6

Arrow function

- Kurzschreibweise für Funktionen
- Zusätzlich übernimmt die aufgerufene Funktion den this-Zeiger und bindet diesen nicht automatisch innerhalb der Funktion
- Wird zumeist für anonyme Callbacks 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

ECMAScript 6

Klassen

- Nicht wirklich...
- „suger over the prototype-based OO pattern“

```
class Polygon {

    constructor( height, width) {
        this.height = height;
        this.width = width;
        this.area = width * height;
    }

    updateWidth(width) {
        this.width = width;
        this.area = width * this.height;
    }
}
// Konstanten gibt es auch in ES6
const person = new Polygon(10, 20);
```

Weitere Informationen: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

ECMAScript 6

Und noch viel mehr…

- Template Strings
- Spread-Operator (…)
- let (bereits erwähnt)
- Iteratoren, Generatoren
- Module (!!!)
  - > Export und import von Funktionen

- …

Weitere Informationen und mehr Beispiele: https://hacks.mozilla.org/category/es6-in-depth/

ECMAScript 7

Veröffentlicht Juni 2016

- Viel geplant, am Ende wenig Neues
  - > Math.pow-Kurzschreibweise: x ** y
  - > Array.prototype.includes
- Für die nächste Version

→ **bisher nur stark eingeschränkter Browser Support**

(Stand Nov. 2016)

Feature name	Current browser (44%)	Traceur (28%)	Babel + core-js[1] (66%)	Closure (36%)	TypeScript + core-js (57%)	es7-shim (23%)	IE 11 (4%)	Edge 13[2] (11%)	Edge 14[2] (52%)	FF 45 ESR (17%)	FF 49 (44%)	CH 54[3] (55%)	SF 9 (6%)	SF 10 (44%)
2016 features														
exponentiation (**) operator	0/3	2/3	3/3	3/3	2/3	0/3	0/3	0/3	3/3	0/3	0/3	3/3	0/3	0/3
Array.prototype.includes	3/3	0/3	3/3	0/3	3/3	2/3	0/3	0/3	3/3	3/3	3/3	3/3	2/3	3/3
2016 misc														
generator functions can't be used with "new"[6]	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	No	Yes
generator throw() caught by inner generator[7]	Yes	No	No	Yes	No	No	No	No	Yes	Yes	Yes	Yes	No	Yes
strict fn w/ non-strict non-simple params is error[8]	No	No	No	No	No	No	No	Yes	Yes	No	No	Yes	No	Yes
nested rest destructuring, declarations[9]	Yes	No	Yes	Yes	Yes	No	No	Flag	Yes	No	Yes	Yes	No	No
nested rest destructuring, parameters[10]	Yes	No	Yes	Yes	Yes	No	No	Flag	Yes	No	Yes	Yes	No	No
Proxy, "enumerate" handler removed[11]	Yes	No	No	No	No	No	No	No	No	No	Yes	Yes	No	Yes
Proxy internal calls, Array.prototype.includes	Yes	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	No	Yes
2017 features														
Object.values	Yes	No	Yes	No	Yes[5]	Yes	No	No	Yes	No[4]	Yes	Yes	No	No
Object.entries	Yes	No	Yes	No	Yes[5]	Yes	No	No	Yes	No[4]	Yes	Yes	No	No
Object.getOwnPropertyDescriptors	0/2	0/2	1/2	0/2	1/2	1/2	0/2	0/2	0/2	0/2	0/2	2/2	0/2	1/2
String padding	2/2	0/2	2/2	0/2	2/2	2/2	0/2	0/2	0/2	0/2	2/2	0/2	0/2	2/2
trailing commas in function syntax	0/2	0/2	2/2	0/2	2/2	0/2	0/2	0/2	2/2	0/2	0/2	0/2	0/2	2/2
async functions	0/3	3/3	3/3	3/3	2/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3

Quelle: http://kangax.github.io/compat-table/es2016plus/

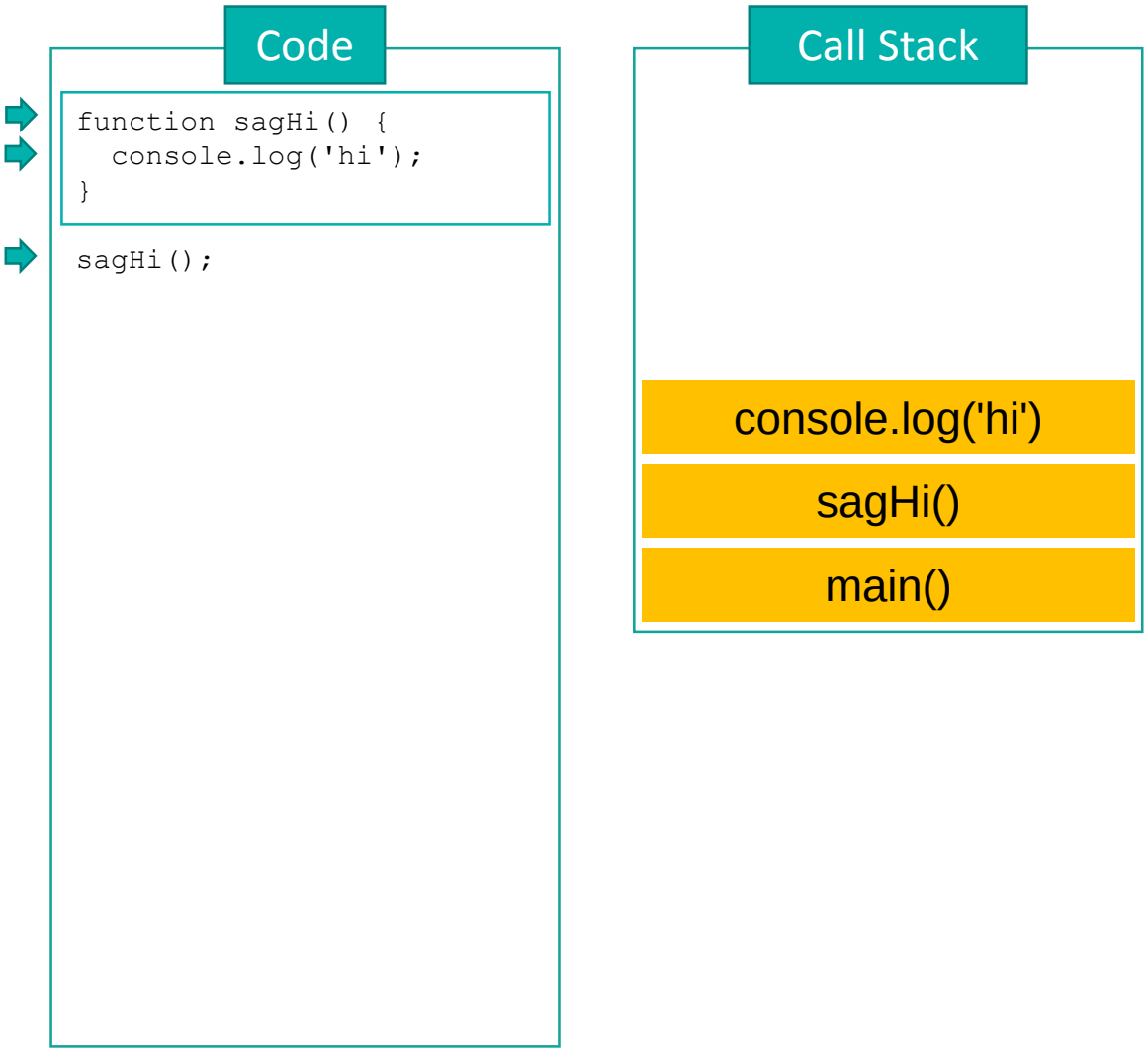
Event Loop

Charakteristik einer JavaScript Runtime

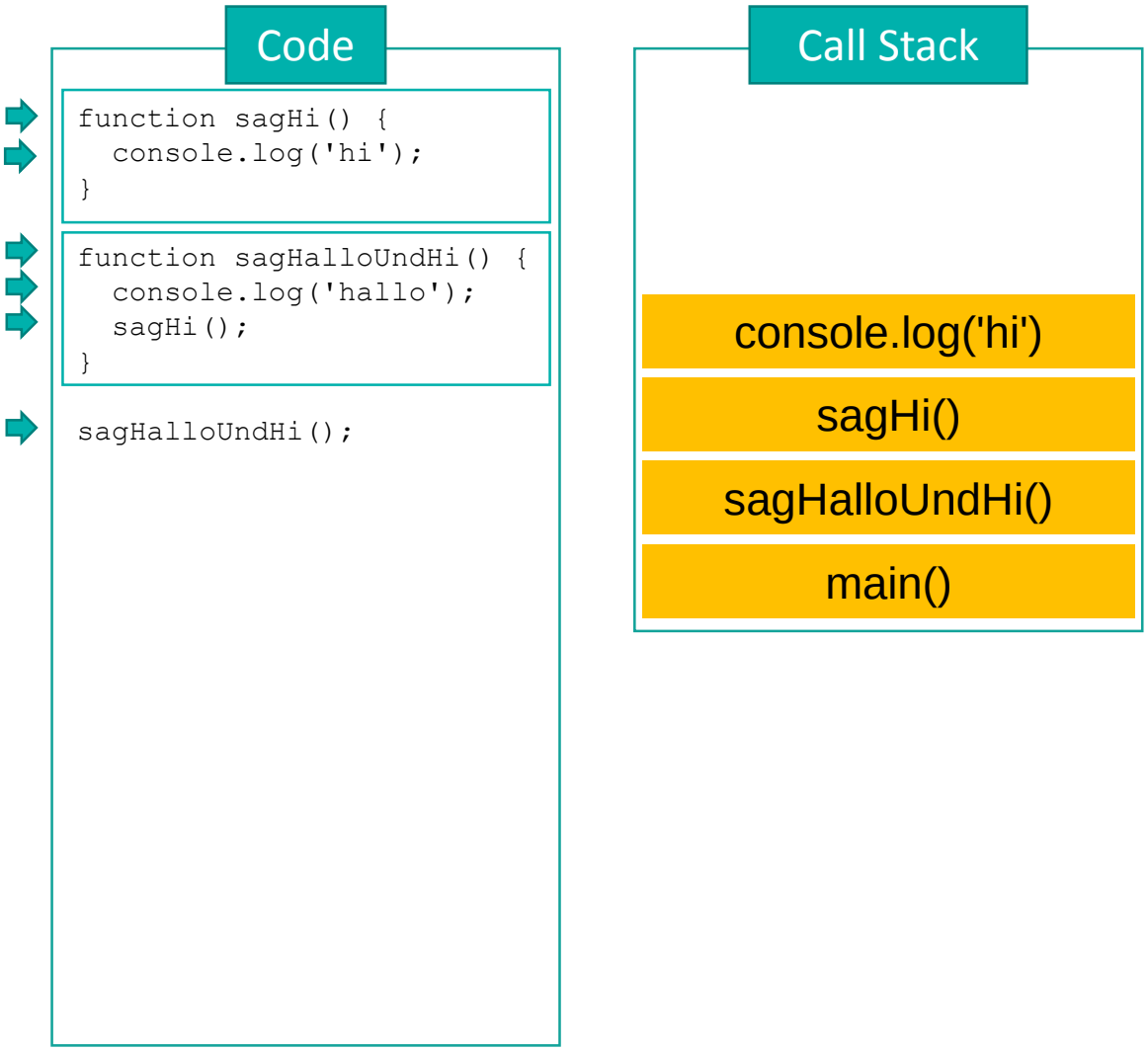
„It‘s a single-threaded non-blocking
asynchronous event-based language!“

- Ein Thread
- 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
  - > Sehr passend für die Nutzung von Events

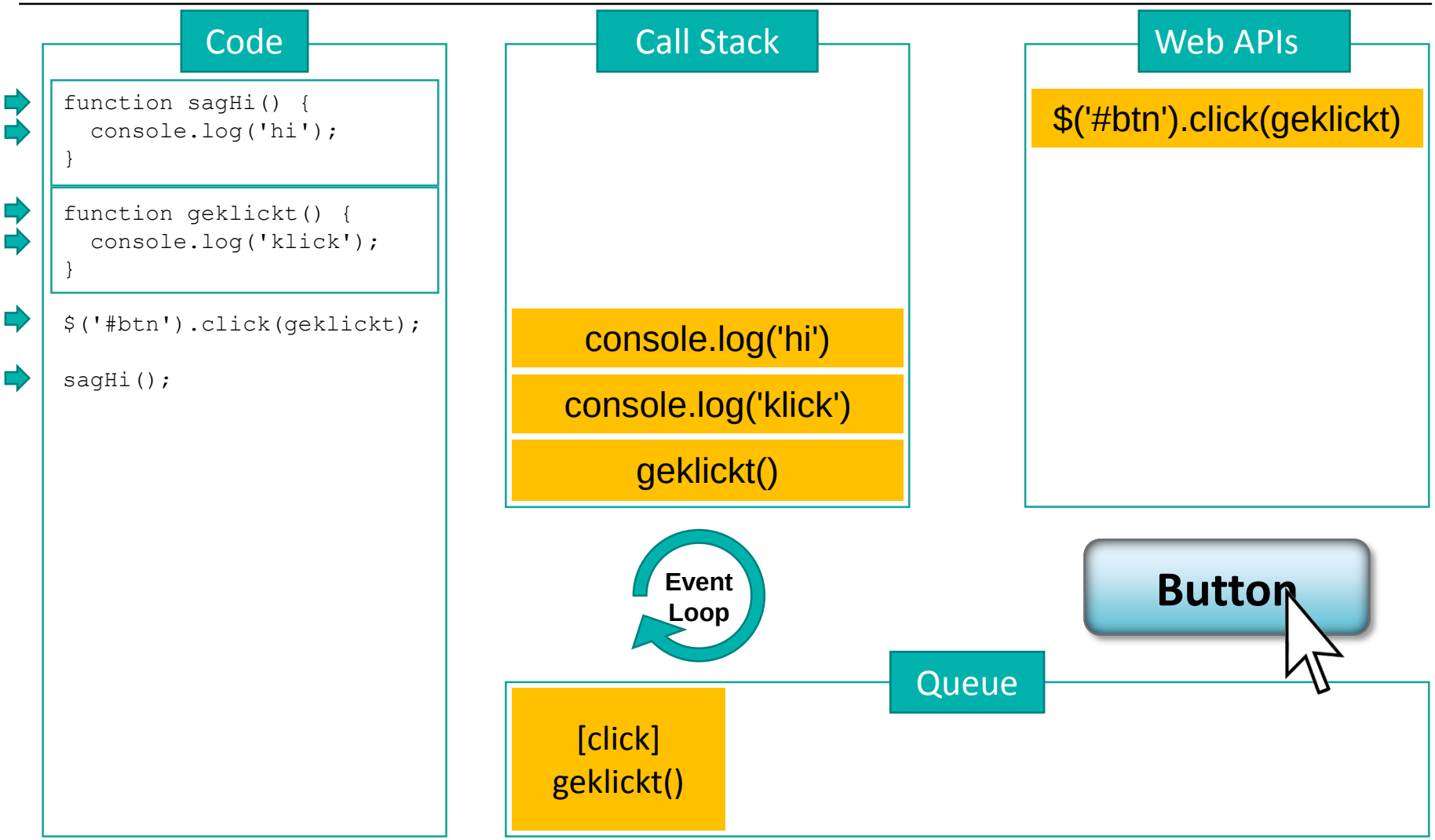
Event Loop


Code
function sagHi() {
console.log('hi');
}
sagHi();
Call Stack
console.log('hi')
sagHi()
main()

Event Loop

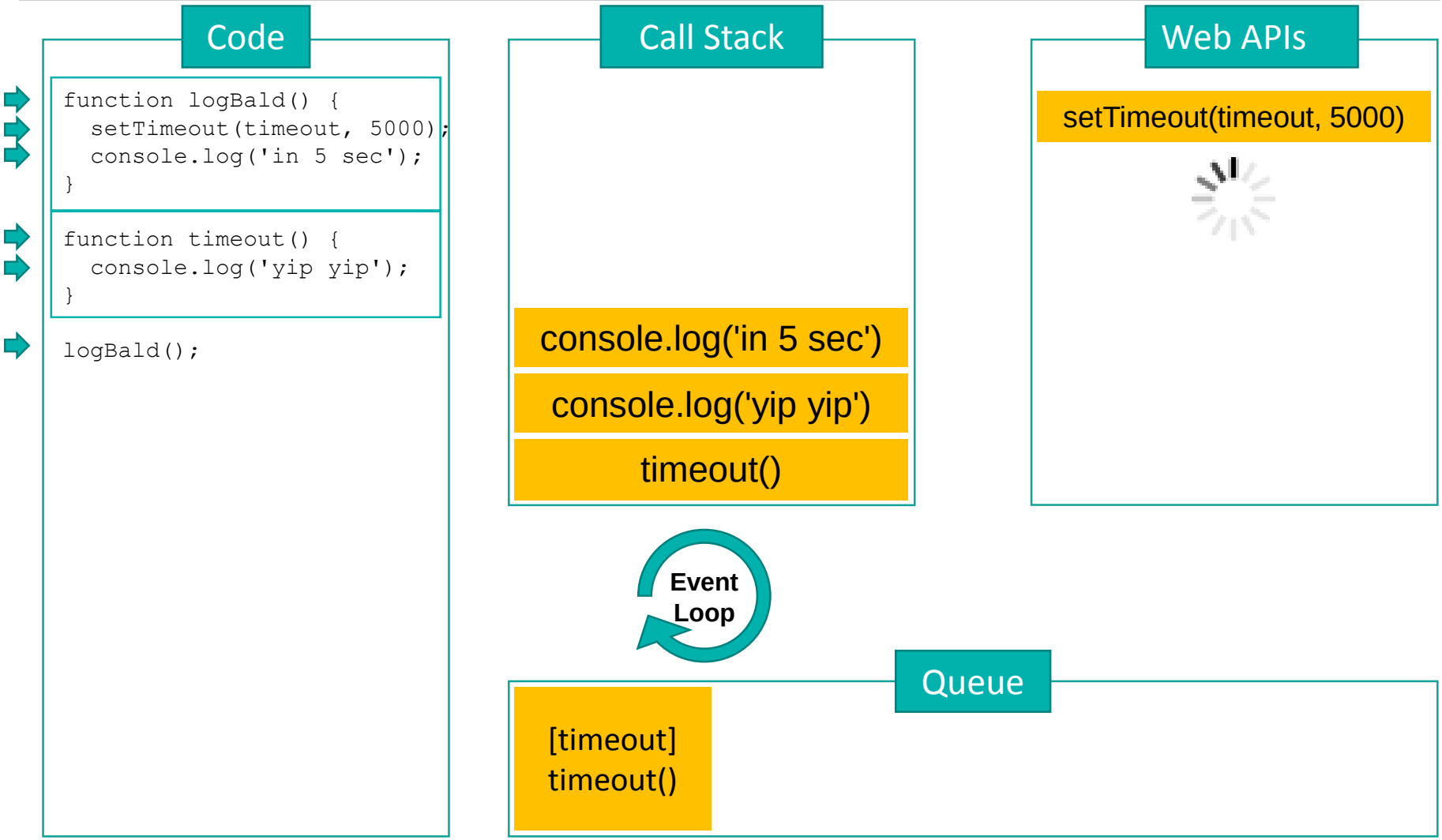

Code
function sagHi() {
console.log('hi');
}
function sagHalloUndHi() {
console.log('hallo');
sagHi();
}
sagHalloUndHi();
Call Stack
console.log('hi')
sagHi()
sagHalloUndHi()
main()

Event Loop



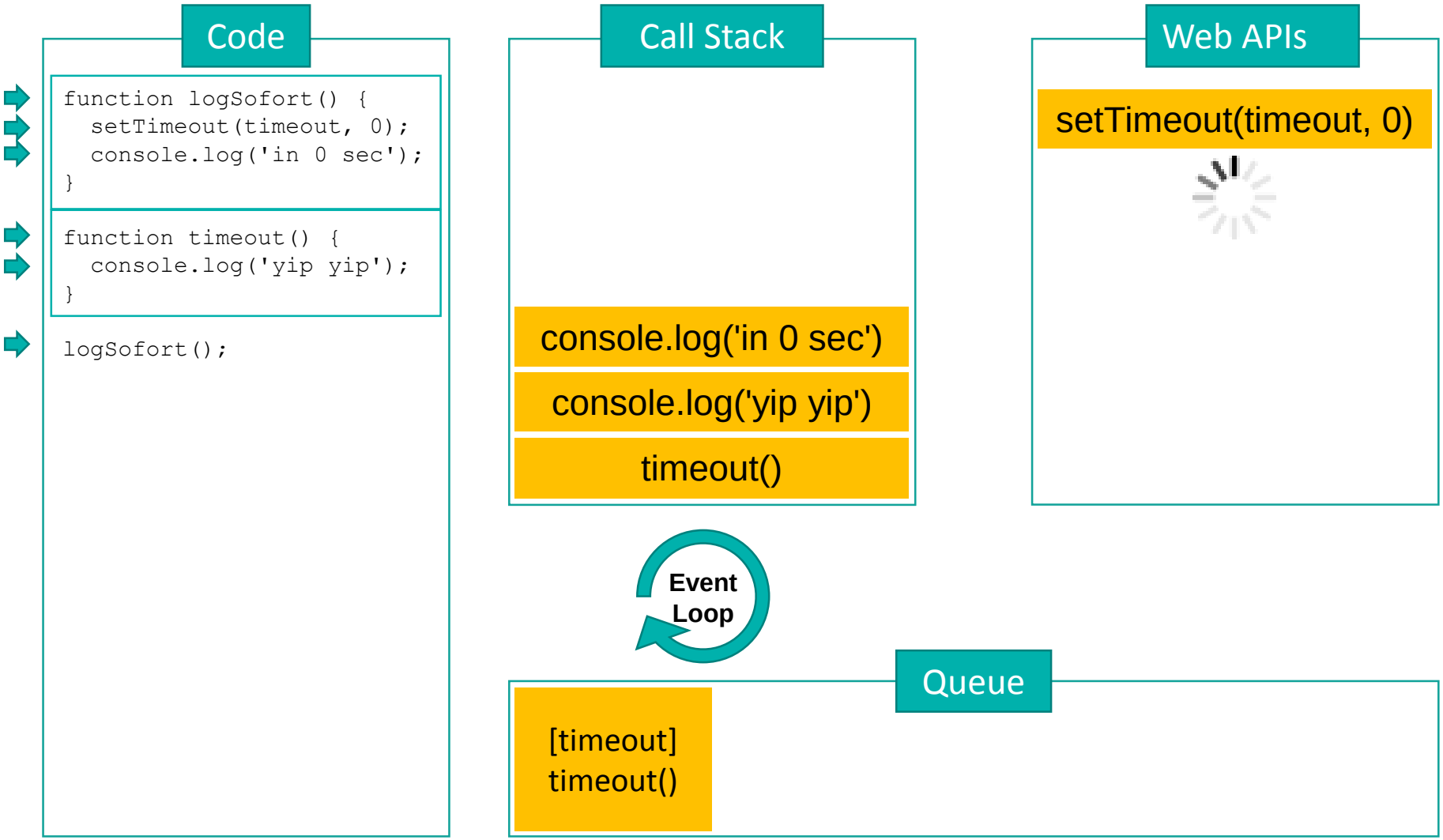
Selber ausprobieren: http://latentflip.com/loupe/

Event Loop



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

Event Loop



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

Event Loop

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)

Node.js

JavaScript Serverseitig

- Gleiche Bibliotheken für Server und Client
- Nur eine Sprache zur Entwicklung notwendig

Node.js

- Basierend auf (Chrome's) JavaScript Runtime V8
- Event-basierte, nicht-blockierende Ein- und Ausgabe
- hochperformante Web-Applikationen möglich
  - > Streaming
  - > JSON-basierte REST-Dienste
  - > Single-Page-Anwendungen

- Paketmanager `npm`
  - > Vergleichbar mit Composer für PHP

Weitere Informationen: https://nodejs.org/

Node.js

Beispiel: HTTP-Server in Node.js

```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

Server starten:

```
node hello.js
```

Exkurs: Serverseitige Verarbeitung von Anfragen

- „Traditionelle“ Möglichkeiten:
  - > Synchrones Handling von Anfragen (ein Request nach dem nächsten)
    - Langsam
    - Wird in der Regel nicht durchgeführt
  - > Parallele Ausführung von Code
    - Mit Hilfe von Threads (oder Prozessen)
    - Weniger Overhead (als bei Prozessen)
    - Anwendung: Apache
    - Kann zu Problemen führen
      - Beispiel: Dirty Reads
      - Synchronisation der Prozesse nötig

Node.js

Event Loop

- Code läuft in nur einem Thread
  - > IO-Zugriff (Dateien/Datenbanken) findet parallel statt
    - Thread-Pool zum Zugriff auf Daten
    - Asynchrone Verarbeitung von Datenzugriffen
- Kommunikation mit Datenbanken (als Beispiel) über Callbacks
  - > Datenbank „sagt Bescheid" wenn Daten bereit stehen
    - Callback wird aufgerufen

Stärken

- viele gleichzeitige Anfragen möglich
- Schnell
  - > Skaliert oft besser und benötigt weniger Speicher

Schwächen

- nicht geeignet für rechenintensive Anwendungen

Fazit

- Sinnvoller Einsatz von Node.js ist abhängig vom Anwendungsfall!