

# Referenz Javascript/AJAX v1.129

**Dozent:**

Christian Heisch

Diese Doku dient der Zusammenfassung der besprochenen Themen als Referenz in einem übersichtlichen Format.

Sie finden diese auf dem FTP-Server. Bitte beachten Sie die Versionsnummer, die den Bearbeitungsstand darstellt. Sollten Sie Fehler oder Unzulänglichkeiten in der Doku entdecken, bin ich Ihnen für einen Hinweis dankbar.

# Einführung

Konfuzius sagt:

*„Liebe, was Du tust und Du wirst nie wieder arbeiten müssen.“*

Aufgrund der Funktionsweise von Javascript ist die Sprache um einiges langsamer als z.B. C++, dafür ist sie die einzige weit verbreitete Sprache, die direkt ohne Zeitverzögerung den Inhalt einer Webseite auslesen, mannigfaltig manipulieren und neu erstellen kann.

Im Rahmen dieses Lehrgangs kann aufgrund der Dauer von nur drei Wochen plus Projekt nur relativ oberflächlich auf einen Teil der Möglichkeiten eingegangen werden. Daher ist ein Eigenstudium während und nach dem Lehrgang eine gute Idee.

## Über diese Dokumentation

Diese Dokumentation versteht sich als Begleitwerk zum Lehrgang.

Sie ist als Referenz zu verstehen und bietet damit eine Möglichkeit, schnell nachzuschlagen, wie die eine oder andere Technik angewendet wird. Sie versteht sich *nicht* als vollständige Javascript-Dokumentation, das wäre zu viel verlangt.

Was diese Doku nicht abnehmen kann, ist die Ideenfindung.

Die ist immer noch mehr Kunst als Wissenschaft und das ist eigentlich auch ganz gut so 😊

## Lass Dich nicht beeindrucken

Es werden gern Begriffe wie Typecasting, InnerCaps, Klassen, Funktionen, Methoden, Parameter etc. verwendet. Dabei handelt es sich lediglich um den Versuch, die Kommunikation zu erleichtern, damit jeder sofort weiß, wovon der Sprecher gerade redet. Allerdings sind diese Begriffe nicht immer eindeutig und die Interpretation unterscheidet sich zum Teil sogar.

Daher mein Aufruf: **Lass Dich nicht beeindrucken**, alle anderen kochen auch nur mit Wasser.

Im Anhang findet sich ein Glossar mit den wichtigsten Begriffen.

Wenn Du erst ein wenig Erfahrung mit der Thematik hast, wirst Du schnell feststellen, dass Programmieren kein Hexenwerk ist, sondern vor allem eine Geduldsprobe. Große Programmierer sind nicht (nur) besonders genial, sondern vor allem besonders geduldig und werfen niemals die berühmte Flinte ins Korn.

## Nicht nur im Browser

Javascript lässt sich nicht nur im Webbrowser.

Mit Hilfe des Projektes Electron (<https://electronjs.org/>) können Apps auf Basis von HTML/CSS/Javascript erstellt und weitergegeben werden. Electron kümmert sich dabei um die Basisfunktionalitäten wie Updates, Native Menüs, Crashreporting, Debugging und Installationsroutinen. Auf die Weise haben schon hunderte von Apps das Licht der Welt erblickt.

Auch in InDesign kann man Javascript einbinden.

Mehr Info gibt es hier: <http://www.indesignscript.de/indesign-skripte-erstellen/>


## Wofür wird Javascript genutzt?

<https://molily.de/js/einsatz.html>

# Inhaltsverzeichnis

<b>EINFÜHRUNG .....</b>	<b>2</b>
<b>INHALTSVERZEICHNIS .....</b>	<b>3</b>
<b>ALLGEMEINES.....</b>	<b>6</b>
<b>EINBINDUNG .....</b>	<b>8</b>
STRICT-MODUS.....	9
<b>VARIABLEN.....</b>	<b>12</b>
VARIABLEN MIT LOKALEM SCOPE .....	13
KONSTANTEN.....	13
RECHNEN MIT VARIABLEN .....	14
STRING-METHODEN.....	16
TYPECASTING.....	20
HOISTING.....	21
LÖSCHEN VON VARIABLEN .....	22
<b>EIN- UND AUSGABE .....</b>	<b>23</b>
ALERT / CONFIRM / PROMPT .....	23
FORMULARFELDER .....	23
<b>VERZWEIGUNGEN .....</b>	<b>27</b>
IF.....	27
SWITCH .....	29
TERNÄRER OPERATOR .....	30
<b>SCHLEIFEN.....</b>	<b>31</b>
WHILE-SCHLEIFE .....	31
FOR-SCHLEIFE.....	31
DO-WHILE-SCHLEIFE.....	32
MANIPULATION DER SCHLEIFENAUSFÜHRUNG .....	32
<b>FUNKTIONEN.....</b>	<b>35</b>
RÜCKGABEWERTE .....	38
ZEITSTEUERUNG .....	38
HOISTING.....	39
ARROW-FUNKTIONEN.....	39
<b>ARRAYS.....</b>	<b>41</b>
ARRAY-METHODEN.....	43
MEHRDIMENSIONALE ARRAYS .....	48
<b>OBJEKTE .....</b>	<b>50</b>
METHODEN .....	51
KONSTRUKTOREN .....	53
KLASSEN .....	54
GETTER UND SETTER .....	56
DAS DATE-OBJEKT.....	57
DAS WINDOW-OBJEKT .....	60
PROTOTYPING .....	70

<b>DOM ( WEBSEITE MANIPULIEREN ) .....</b>	<b>62</b>
ELEMENTE ANSPRECHEN.....	74
ELEMENTE VERÄNDERN .....	76
ANLEGEN, KOPIEREN, VERSCHIEBEN, LÖSCHEN .....	78
<b>EVENTLISTENER .....</b>	<b>80</b>
<b>LOKALES SPEICHERN .....</b>	<b>85</b>
COOKIES .....	85
WEBSTORAGE (A.K.A. DOM STORAGE, SUPERCOOKIES) .....	87
<b>PROMISES .....</b>	<b>89</b>
CHAINING .....	90
THEN ... CATCH.....	91
IN DER PRAXIS ... ..	92
ASYNC, AWAIT .....	92
PROMISE UND AJAX .....	93
<b>AJAX.....</b>	<b>95</b>
JSON.....	99
REST .....	100
SOAP .....	101
FETCH .....	102
<b>WEBSOCKETS .....</b>	<b>105</b>
<b>XML .....</b>	<b>110</b>
<b>DATEIAUSWAHL .....</b>	<b>113</b>
NODEJS.....	113
UPLOAD ÜBER EIN FORMULAR.....	115
UPLOAD MIT AJAX .....	116
UPLOAD MIT FETCH.....	117
DRAG 'N' DROP .....	117
<b>CANVAS .....</b>	<b>118</b>
FARBFÜLLUNG.....	118
FORMEN .....	121
BILDER SETZEN .....	124
ANIMIEREN.....	126
TRANSFORMATION.....	126
PIXEL FÜR PIXEL.....	128
<b>FEHLERSUCHE.....</b>	<b>130</b>
<b>PERFORMANCE (A.K.A. SCHNELLERER CODE) .....</b>	<b>131</b>
<b>BIBLIOTHEKEN / FRAMEWORKS.....</b>	<b>135</b>
<b>JQUERY .....</b>	<b>136</b>
<b>ANGULARJS .....</b>	<b>141</b>
MODULE.....	144
FORMULARE .....	146
ANGULARJS UND AJAX .....	148
WARUM NICHT ANGULAR (V. 4)? .....	149

<b>NODEJS .....</b>	<b>150</b>
INSTALLATION UND ERSTE SCHRITTE .....	150
MODULE.....	151
WEBSERVER ERSTELLEN.....	154
DATEIZUGRIFF .....	157
DATENÜBERTRAGUNG (POST/JSON) .....	158
MYSQL.....	161
PAKETMANAGEMENT (PACKAGE.JSON).....	162
<b>REACTJS .....</b>	<b>164</b>
ERSTE SCHRITTE .....	164
VIRTUELLES DOM .....	164
KOMPONENTEN UND JSX .....	165
REACTJS UND AJAX .....	172
<b>VUE.....</b>	<b>174</b>
DARSTELLUNG IM VIRTUAL DOM .....	175
DIREKTIVEN .....	176
EIGENE DIREKTIVEN.....	182
METHODEN .....	184
COMPUTED PROPERTIES.....	184
EVENTLISTENER .....	185
KOMPONENTEN.....	187
PLUGINS .....	FEHLER! TEXTMARKE NICHT DEFINIERT.
<b>ANDERE BIBLIOTHEKEN .....</b>	<b>191</b>
<b>ANHANG: VERALTETE BROWSER.....</b>	<b>193</b>
SELBSTGESCHRIEBEN.....	193
<b>COMPILER .....</b>	<b>194</b>
<b>JS PREPROCESSORS / TRANSPILER .....</b>	<b>195</b>
<b>ANHANG: VORSICHT FALLE .....</b>	<b>FEHLER! TEXTMARKE NICHT DEFINIERT.</b>
HALTE DEINEN CODE SAUBER.....	196
HÄUFIG AUFTRETENDE FEHLER .....	196
<b>ANHANG: HILFSMITTEL.....</b>	<b>200</b>
<b>ANHANG: WEITERFÜHRENDE LITERATUR.....</b>	<b>203</b>
<b>ANHANG: GLOSSAR .....</b>	<b>204</b>
<b>ANHANG: JOBPLATTFORMEN .....</b>	<b>206</b>
 <b>NICHT IM IE .....</b>	<b>208</b>
<b>INDEX.....</b>	<b>210</b>

# Allgemeines

## Entstehung

Javascript wurde 1995 von Netscape entwickelt, um eine Möglichkeit zu schaffen, Webseiten dynamischer zu machen. Es hat nichts mit Java zu tun, das ist eine vollkommen andere Sprache mit ganz eigenen Regeln.

Als nach und nach andere Entwickler Javascript in Ihre Browser integrierten, wurde eine Standardisierung nötig, was durch die European Computer Manufacturers Association (ECMA) mit ECMAScript geschafft wurde. Heute ist ECMAScript (auch wenn man davon nur selten hört) die Basis und Javascript sind die jeweiligen Interpretationen der Browserhersteller. Die aktuelle Version von ECMAScript ist ECMAScript2016 (Stand Juni 2017), die letzten größeren Änderungen wurden allerdings in ECMAScript2015 (genannt ES6) eingeführt.

## Syntax

Grundsätzlich gibt es einige, wenige Regeln zu beachten:

- Jede Befehlszeile wird mit einem **Semikolon ( ; )** abgeschlossen.  
Das ist jedenfalls der übliche Weg. Es gibt auch einige Situationen, in denen das Semikolon weggelassen werden kann. Javascript erkennt dann trotzdem das Ende des Befehles. Man kann sogar ganz auf abschließende Semikola verzichten. Dazu muss man JS aber erstmal genauer kennen. Mehr hier: <https://mislav.net/2010/05/semicolons/>
- In Kontrollstrukturen (Schleifen, Verzweigungen, Funktionen) wird der auszuführende Code immer in **geschweiften Klammern** geschrieben.
- Bei Rechnungen gilt **Punkt- vor Strichrechnung**.

## Kommentare

Kommentare sind Zeilen im Programmcode, die vom Programm komplett ignoriert werden. Sie werden gern eingesetzt, zum

- Einfügen von Kommentaren und näheren Beschreibungen
- Entfernen von Codezeilen, ohne sie wirklich zu löschen. Meistens zur Fehlersuche.

In Javascript und den meisten anderen Programmiersprachen gibt es zwei Methoden, Kommentare einzufügen:

```
// Zwei Slashes (a.k.a. Querstriche) markieren nur diese eine Zeile als
Kommentar.

/*
Slash und Stern markieren alle Zeilen bis zum umgedrehten Zeichenpaar als
Kommentar.
*/
```

## Dateitypen

HTML- wie auch Javascript- und CSS-Dateien sollten im Zeichensatz **UTF-8 ohne BOM** gespeichert werden.

BOM steht für Byte Order Mark. Das ist eine Zeichenfolge, die bei umfangreicheren Zeichensätzen bestimmt, auf welche Weise die Bits interpretiert werden.

Bei UTF-8 ist dies nicht notwendig, der BOM kann aber eingebaut werden.

Das kann zu **Problemen** führen, wenn der Browser keinen BOM erwartet. Hier wird er dann in der Form: `ï»¿` dargestellt.

# Einbindung

Um Javascript (JS) in einer Webseite zu verwenden, muss der Code in den HTML-Code eingebunden werden.

Dazu gibt es verschiedene Herangehensweisen:

**Inline-Einbindung** bedeutet, der JS-Code wird innerhalb des script-Tag geschrieben. Dies ist vor allem dann sinnvoll, wenn der JS-Code relativ übersichtlich ist.

```
<html>
  <head>
    <script>
      //Javascript-Code
    </script>
  </head>
  <body>
  </body>
</html>
```

**Javascript-Links** sind a-Tags, die als Linkziel Javascript-Code enthalten. Aufgrund der empfohlenen Trennung von Programmlogik und Darstellung sollte nach Möglichkeit darauf verzichtet werden.

```
<a href="Javascript:funktionsaufruf();" >
  Klicktext
</a>
```

**Event-Listener** sind HTML-Attribute, deren Inhalt im Falle eines bestimmten Ereignisses ausgeführt wird. Aufgrund der empfohlenen Trennung von Programmlogik und Darstellung sollte nach Möglichkeit darauf verzichtet werden. Eventhandler lassen sich auch problemlos in reinem Javascript umsetzen.

```
<button onclick="funktionsaufruf();" >
</button>
```

**Externe Dateien** sollten immer dann eingebunden werden, wenn der Code umfangreicher wird oder von mehreren, verschiedenen Seiten darauf zugegriffen werden soll.

Einzig und allein die Reihenfolge, mit der Code auftaucht, entscheidet darüber, welche Zeilen zuerst ausgeführt werden. Wenn die externe Datei vor dem Inline-Code eingebunden wird, dann wird der Code in der externen Datei zuerst ausgeführt.

Code, der in dem Javascript-Tag eingebunden wird, welcher die Datei lädt, wird ignoriert.

```
<script src="meinScript.js">
  // Hier stehender Code wird ignoriert
</script>
```



```
<script>
  // Hier stehender Code ("inline") wird ausgeführt
</script>
```

### Ausführung verzögern

Das Attribut **defer** verzögert die Ausführung so lange, bis das gesamte Dokument geladen ist. Das Script wird dann noch vor dem DOMContentLoaded-Eventlistener und weit vor dem load-Eventlistener gestartet.

Javascript-Datei wird parallel zum HTML-Baum geladen -> Geschwindigkeitsvorteil.

Es handelt sich um ein boolesches Attribut, es kann also auf jede Weise geschrieben werden, die als *true* bzw. *false* interpretiert wird.

```
// Alternative 1
<script src="meinScript.js" defer></script>

// Alternative 2
<script src="meinScript.js" defer="defer"></script>

// Alternative 3
<script src="meinScript.js" defer="true"></script>
```

### Sofort ausführen

Das Attribut **async** sorgt dafür, dass das Skript sofort ausgeführt wird, wenn es geladen wurde. Die Reihenfolge, mit der Skripte aufgeführt sind, spielt dabei keine Rolle.

```
// Alternative Schreibweisen gelten wie bei defer
<script src="einScript.js" async></script>
<script src="anderesScript.js" async></script>
```

## Strict-Modus

Sinn des Strict-Modus ist es, den Benutzer zu sauberem Code zu zwingen.

Dinge, die ohne Strict-Modus einfach als schlechter Stil abgetan wurden, führen so zu einer Fehlermeldung. Das Ziel ist sauberer, sicherer Code.

Um den Strict-Modus zu aktivieren, schreibe an den **Anfang** des Programms oder der Funktion den folgenden Befehl:

```
"use strict";
// Weitere Befehle
```

Diese etwas ungewöhnliche Schreibweise sorgt dafür, dass alte Browser, die diesen Strict-Modus noch nicht kennen, diesen Befehl ignorieren (Besser gesagt: die Zeile wird ausgeführt, hat aber keine Wirkung).

Übersicht über Probleme und wie damit umgegangen wird:

	Ohne Strict	Mit Strict
Benutzen von nicht mit var angelegten Variablen	Variable wird global angelegt	Fehlermeldung
Erneutes Anlegen einer existierenden Variablen	Funktioniert	Fehlermeldung
Löschen einer Variablen mit delete	Funktioniert nicht	Fehlermeldung
Löschen einer Funktion mit delete	Funktioniert nicht	Fehlermeldung
Parameternamen in einer Funktion mehrfach verwenden	Zweiter Parameter überschreibt ersten	Fehlermeldung
Oktalzahlen mit prefix '0' (null)	Funktioniert, ist aber in der Schreibweise mehrdeutig.	Fehlermeldung Korrektur Prefix (0o)
Überschreiben einer Nur-Lese-Eigenschaft	Funktioniert nicht	Fehlermeldung
Löschen einer nicht löschbaren Eigenschaft (z.B. Konstruktor)	Funktioniert nicht	Fehlermeldung
Verwenden des with-Befehles	Funktioniert	Fehlermeldung
Verwenden als Variablenname: <ul style="list-style-type: none"> <li>• eval</li> <li>• arguments</li> <li>• implements</li> <li>• interface</li> <li>• let</li> <li>• package</li> <li>• private</li> <li>• protected</li> <li>• public</li> <li>• static</li> <li>• yield</li> </ul>	Überschreibt Schlüsselwörter	Fehlermeldung

Mehr dazu: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict\\_mode/Transitioning\\_to\\_strict\\_mode#Differences\\_from\\_non-strict\\_to\\_strict](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode/Transitioning_to_strict_mode#Differences_from_non-strict_to_strict) (Strg-Klick mich)

Und: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict\\_mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode) (Strg-Klick mich)

### Block einpacken (IIFE)

Erfahrene Entwickler **vermeiden globale Variablen**, wo es nur geht.

Darunter versteht man Variablen, die in allen Skripten zur Verfügung stehen, weil sie auf der Wurzelebene (außerhalb aller Funktionen, Schleifen oder if-Anweisungen) angelegt wurden.

Es gibt Situationen, in denen globale Variablen notwendig sind – z.B., wenn ein Wert auch in anderen Codeblöcken benötigt wird. Aber üblicherweise kapselt man die Skripte gegeneinander ab.

Die Technik dafür hört auf den schönen Namen IIFE (gesprochen iffi), was für *Immediately Invoked Function Expression* steht. Also eine Funktion, die sofort nach dem Anlegen ausgeführt wird.

In diese *IIFE* wird der gesamte Code geschrieben, der nicht global sichtbar sein soll.

```
<script>

  'use strict';

  // Diese Variable existiert in allen script-Tags
  var x = 42;

  (function(){

    // Diese Variable existiert nur innerhalb der geschweiften Klammern
    var y = 42;
    console.log ( y );

  }) ();

  console.log ( y );

</script>
```

# Variablen (Primitive Datentypen)

Der Dreh- und Angelpunkt in jeder Programmiersprache sind Variablen.

Dies sind Speicherbereiche im Computer, die sich durch Programmcode lesen und manipulieren lassen. Jede Variable kann von einem bestimmten Datentyp sein.

Es wird unterschieden zwischen *primitiven Datentypen* und *zusammengesetzten Datentypen* (*Collections*). In diesem Kapitel widmen wir uns ausschließlich primitiven Datentypen. Das bedeutet: solche Datentypen, die lediglich einen einzigen Wert speichern können.

Diese sind:

**Boolean:** Kann nur true oder false speichern. Wird vor allem verwendet, um sog. Flags zu setzen oder um das Ergebnis einer logischen Überprüfung zu speichern.

**Number:** Kann Zahlen mit bis zu 15 Nachkommastellen speichern. Javascript unterscheidet nicht zwischen ganzen Zahlen, Fließkommazahlen und ähnlichem.

**String:** Zeichenketten: Beliebige viele, aneinandergehängte Zahlen, Buchstaben und Zeichen. Mit anderen Worten: Text.

**Symbol:** Eindeutige und unveränderbare Variablen. ([Klick für mehr](#))

**Null:** Variable ohne Wert

**Undefined:** Sinngemäß für „Existiert nicht“.

Beim Anlegen der Variable muss der Datentyp nicht angegeben werden. Dieser wird automatisch aus dem Wert ermittelt, der in der Variable gespeichert wird. Das wird implizite Typisierung genannt.

Die Regeln zur Wahl des **Variablennamens** sind nicht besonders strikt.

- Er darf nicht mit einer **Zahl** beginnen.
- Bestehende **Schlüsselwörter** (name, index, etc.) dürfen nicht verwendet werden.
- Er darf zwar fast jedes Zeichen enthalten, man sollte sich aber auf Buchstaben (keine Umlaute), Zahlen und den Unterstrich '\_' beschränken.

Um sich selbst das Leben leichter zu machen, sollten noch andere Ideen berücksichtigt werden:

- Ein Bezeichner sollte **aussagekräftig** sein. Jemand anders, der den Code lesen will, sollte am Namen erkennen, was die Variable enthält. x oder y sind keine gute Wahl für einen Variablenamen.
- **Abkürzungen** können problematisch sein.
- Für die Variablenamen haben sich sogenannte **innerCaps** etabliert. Das bedeutet, der erste Buchstabe ist auf jeden Fall klein, eigenständige Begriffe im Variablenamen werden mit großem Buchstaben begonnen.
- Variablenamen sind **casesensitiv**. Das heißt, Groß- und Kleinschreibung werden sehr streng unterschieden.
- Ein Variablenname sollte aus Nomen bestehen. Also lieber *gewinn* als *gewonnen*
- Ein Variablenname sollte im Singular geschrieben sein. Also lieber *gewinn* als *gewinne*

```
// Anlegen einer Variable ohne Wert.  
var meineVariable;
```

```
// Anlegen einer Variable vom Typ Number mit Wert
var meineAndereVariable = 42;

// Anlegen einer Variable vom Typ String mit Wert.
// In der zweiten Zeile wird zwar eine Zahl gespeichert.
// Durch die Anführungszeichen wird aber String als Datentyp erzwungen.
var nochEineVariable = 'Zweiundvierzig';
var eineVierteVariable = '42';

// Anlegen einer Variable vom Typ Boolean mit Wert
var diesIstDieLetzteVariable = true;
```

Um in einer Variablen einen Wert zu speichern, weisen wir den Wert mit einem einfachen Gleichzeichen zu. Alles, was bis dahin in der Variable stand wird unwiederbringlich durch den neuen Wert überschrieben.

```
meineVariable = 43;
```

## Variablen mit lokalem Scope



[\(Strg-Klick für mehr\)](#)

Seit ECMA-Script 2016 gibt es das Schlüsselwort `let`.

Es funktioniert wie `var`, nur haben die Variablen einen *lokalen Scope*. Das bedeutet, dass Variablen, die innerhalb einer Schleife oder einer bedingten Anweisung erzeugt wurden, außerhalb derer nicht mehr zu existieren.

`Let` gilt als die modernere Variante, aber objektiv betrachtet haben beide Varianten ihre Vor- und Nachteile.

```
// Bedingte Anweisungen
if ( true ){
    let x = 5;
    console.log ( 'inner: ' + x );      // 5
}
console.log ( 'outer: ' + x );        // Reference Error

// Schleifen
for ( let i = 0; i < 10; i++ ){
    console.log ( 'innen: ' + i );      // 0 - 9
}
console.log ( 'aussen: ' + i );        // Reference Error
```

## Konstanten



[\(Strg-Klick für mehr\)](#)

Neben Variablen gibt es seit ECMAScript 2015 auch Konstanten. Das sind Speicher, die weder verändert noch neu angelegt werden können.

Konstanten benötigen einen Initialwert, d.h. sie können nicht ohne Wert angelegt werden.

Die Sichtbarkeit der Konstante ist auf den lokalen Scope begrenzt, sie verhält sich also wie eine let-Variable.

Der Bezeichner einer Konstante wird von einigen Entwicklern in sogenannter Screaming-Snake-Case geschrieben. Das heißt, alles wird großgeschrieben, Begriffe werden mit Unterstrich getrennt. Damit lassen sich im Code Konstanten und Variablen leicht auseinanderhalten.

```
const MEINE_KONSTANTE = 0.19;
const LEERE_KONSTANTE; // Fehlermeldung 'missing = in
// const declaration'
alert ( MEINE_KONSTANTE ); // 0.19
MEINE_KONSTANTE = 100; // 'Invalid Assignment'
const MEINE_KONSTANTE = 100; // 'Redeclaration of const x'
```

## Rechnen mit Variablen

Arithmetische Operationen (einfache Rechnungen) funktionieren immer auf dieselbe Weise: Die Rechnung auf der rechten Seite des Gleichzeichens wird ausgeführt und in der Variablen auf der linken Seite des Gleichzeichens übertragen. Man spricht von einer Wertzuweisung.

Aus dem Grund darf links vom Gleichzeichen immer nur ein Variablenname stehen. Alles, was vorher in der Variable links vom Gleichzeichen stand, wird überschrieben.

```
x = 3 * 4; // x wird mit einer 12 überschrieben
x = Math.sin( 45 / 180 * Math.PI ); // x wird mit eine 0.707... überschrieben
```

### Math-Objekt

In Javascript gibt es das Math-Objekt, das uns einige interessante Rechenoperationen zur Verfügung stellt.

```
// Sinus, Cosinus, Tangens: Beachte, dass der Ausgangswert (45) zunächst
// in Bogenmaß umgerechnet werden muss
x = Math.sin( 45 / 180 * Math.PI );

// Zufallszahlen (s.u.)
// Wurzel ziehen
x = Math.sqrt(121);

// Potenzierung, z.B. 12 ^ 2
x = Math.pow ( 12, 2 );

// Und viele mehr
```

## Eval-Funktion

Die Eval-Funktion nimmt einen String (Text) und führt die darin beschriebene Rechnung aus. Die Eval-Funktion kann noch weit mehr, sie kann nämlich beliebigen Javascript-Code als String entgegennehmen und ausführen. Allerdings wird die Verwendung von Eval unnötigerweise mit schlechtem Coding gleichgesetzt. Daher (und weil es häufig unnötig ist) sollte nach Möglichkeit von der Funktion Abstand genommen werden.

```
alert ( eval ( '3*4' ) );           // Gibt eine 12 aus
alert ( eval ( 'Math.sqrt(144)' ) ); // Gibt eine 12 aus
```

## Kurzformen

Es gibt einige Kurzformen häufig vorkommender Zahlenmanipulationen.

Sie werden auch *zusammengesetzte Zuweisungsoperatoren* genannt.

Sie haben nicht nur den Vorteil, dass es weniger zu schreiben ist. Viel wichtiger ist, dass der Code mit diesen Operatoren besser zu lesen und zu warten ist.

Beispiele:

```
// Variable um 1 erhöhen ('inkrement')
b++;
b += 1;
b = b + 1;

// Variable um 1 verringern ('dekrement')
b--;
b -= 1;
b = b - 1;

// Variable um beliebigen Wert erhöhen
b += x;
b = b + x;

// Variable mit beliebigem Wert multiplizieren
b *= x;
b = b * x;
```

## Zufall

Eine spezielle Zuwendung benötigen Zufallszahlen. Sie können verwendet werden, um testweise Variablen mit Inhalt zu befüllen, Wahrscheinlichkeiten (Würfel) umzusetzen und vieles mehr.

Das Erzeugen einer zufälligen Zahl ist eine Methode der Math-Klasse.

```
// Befüllen einer Variablen mit einer zufälligen Zahl
var z = Math.random();
```

Dieser Befehl erzeugt immer und ohne Ausnahme Zahlen von 0 bis (exklusive) 1. Um andere Zahlenbereiche zu erhalten, muss man die erzeugte Zahl umwandeln.

```
// Vorschlag für die Erzeugung einer zufälligen Zahl  
// innerhalb eines bestimmten Bereichs.  
var min = 50;  
var max = 100;  
var z = Math.random() * (max - min) + min;
```

Auch die Definition der Nachkommastellen ist möglich, allerdings muss ein wenig um die Ecke gedacht werden: zunächst wird das Komma nach rechts geschoben, dann gerundet (alle Nachkommastellen werden entfernt), dann wird das Komma wieder nach links geschoben.

```
z = Math.random() * (max - min) + min;  
z *= 100;  
z = Math.round(z);  
z /= 100;  
  
// Oder als Einzeiler  
var z = Math.round( (Math.random() * (max - min) + min) * 100) / 100;
```

## String-Methoden

Ein String lässt sich auf vielerlei Weisen zerlegen und analysieren.

```
var s = 'Hallo Welt';  
var s2;
```

### Mehrzeilige Schreibweise

Einen String auf mehrere Zeilen zu verteilen, kann insbesondere bei langen Strings die Übersicht erleichtern. Um das zu erreichen kann im String ein Backslash ( \ ) gesetzt und dann der String in der nächsten Zeile fortgesetzt werden.

Der \ und der dazugehörige Zeilenumbruch werden nicht ausgegeben.

Leerzeichen werden dagegen ausgegeben, auch wenn sie nur dem Einrücken dienen.

```
var s = 'Lorem ipsum dolor sit amet, consectetur \\  
adipiscing elit. Aenean commodo ligula eget dolor. \\  
Aenean massa.';
```

### Strings verketten

Um mehrere Strings zu verketten, gibt es mehrere Möglichkeiten.

Die einfachste ist, die Strings mit einem Plus-Zeichen aneinander zu hängen.

```
var userName = 'Max Mustermann';  
console.log ( 'Der Name lautet ' + userName + ', willkommen.' );
```



## Template-Literal

Eine alternative Möglichkeit ist die Verwendung von sogenannten **Template-Strings** oder **Template-Literals**. Hier wird der String mit einem `$`-Symbol und in geschweiften Klammern in einen anderen String geschrieben. Zu beachten ist dabei, dass der String nicht in Anführungszeichen steht, sondern in Backticks (Accént-Zeichen links neben dem Backspace).

```
var userName = 'Max Mustermann';
console.log ( `Der Name lautet ${userName}, willkommen.` );
```

## String wiederholen

Um einen gegebenen String mehrfach zu wiederholen, kann die `repeat`-Methode verwendet werden.

```
let s = "Hello world!";
let r = s.repeat(3);
console.log ( r );           // => Hello world! Hello world! Hello world!
```

## Definierte Nachkommastellen

In manchen Fällen soll eine Ausgabe eine bestimmte Zahl an Nachkommastellen haben. Da kann die `toFixed`-Funktion helfen.

```
var wert = 10;
console.log ( wert.toFixed(2) )      // Ausgabe: 10.00
```

## Länge

Länge des Strings ausgeben

```
s2 = s.length;
```

## Teilstrings

Teilstring von Index 3 bis Index 8 ausgeben

Wird der zweite Parameter weggelassen, dann wird von Index 3 aus der Rest des Strings genommen.

```
s2 = s.substring( 3, 8 );
```

Teilstring von 5 Zeichen Länge, beginnend bei Index 3, ausgeben.

Wird der zweite Parameter weggelassen, dann wird von Index 3 aus der Rest des Strings genommen.

```
s2 = s.substr( 3, 5 );
```

Mit *substr* lassen sich auch Indizes vom Ende aus abzählen.

Dazu wird als erster Wert einfach eine entsprechende negative Zahl benutzt.

```
var s = "Hallo";
console.log (s.substr(-1, 1));           // => o
console.log (s.substr(-3, 3));           // => llo
```

Wenn man bei substr den zweiten Parameter weglässt, werden alle Zeichen bis zum Ende zurückgegeben.

```
var s = "Hallo Welt";
console.log (s.substr( 6 ));              // => Welt
```

## Teilstring suchen

Prüfen, **ob** ein String in einem anderen vorkommt.

```
var s = "Hallo Welt";
console.log ( s.includes("We") );        // => true
console.log ( s.includes("we") );        // => false
```

Prüfen, **wo** der String **zum ersten Mal** vorkommt.

```
// Der zweite Parameter gibt an, ab welchem Index gesucht werden soll
// Wird der String nicht gefunden, ist das Ergebnis -1
console.log ( s.indexOf ('Wel', 0) );    // => 6
```

Prüfen, wo der String **zum letzten Mal** vorkommt.

```
// Wie indexOf, aber ohne Start-Index
console.log ( s.lastIndexOf ('Wel') );   // => 6
```

Prüfen, ob ein String mit einem bestimmten String anfängt

```
s = "Hello world, welcome to the universe.";
console.log ( s.startsWith ("Hello") );  // => true
console.log ( s.startsWith ("world") );  // => false
```

Prüfen, ob ein String mit einem bestimmten String endet

```
s = "Hello world, welcome to the universe.";
console.log ( s.endsWith ("universe.") ); // => true
console.log ( s.endsWith ("universe") );  // => false
```

## In Array umwandeln

String bei jedem Vorkommen der Zeichenkette **auftrennen** und das Ergebnis in einem Array speichern.

```
let meinArray = s.split(' ');
```

## Schreibweise

String in **Groß- oder Kleinbuchstaben** verwandeln. Z.B. um bei Gleichheitsprüfungen

```
// Groß-/Kleinschreibung ignorieren zu können  
s = s.toLowerCase();  
s = s.toUpperCase();
```

## Teilstring ersetzen

String nach einem Teilstring durchsuchen und dessen erstes Vorkommen **ersetzen**

```
s = s.replace('blau', 'rot');  
// Wenn alle Vorkommen des Teilstring ersetzt werden sollen,  
// dann muss auf reguläre Ausdrücke zurückgegriffen werden. Das /g steht  
// für global  
s = s.replace(/blau/g, 'rot');
```

## Optimierung

**Whitespaces** (überflüssige Leerzeichen vor und hinter dem String) entfernen:

```
s = s.trim();
```

## Vergleichen

Wir können Strings ebenso wie Zahlen prüfen, welcher "kleiner" ist. Wobei kleiner bedeutet, alphabetisch früher zu kommen. So gilt z.B.:

```
let s1 = 'a';  
let s2 = 'b';  
console.log ( s1 < s2 );           // -> true  
  
s1 = 'a';  
s2 = 'B';  
console.log ( s1 < s2 );           // -> false  
  
s1 = 'Hallo';  
s2 = 'Welt';  
console.log ( s1 < s2 );           // -> true
```

Großbuchstaben sind kleiner als Kleinbuchstaben, weil sie in der ASCII-Tabelle vor den Kleinbuchstaben kommen.

## Typecasting

In vielen Fällen ist Javascript in der Lage, automatisch zu erkennen, welcher Datentyp für eine Operation benötigt wird und die gegebenen Variablen automatisch umzuwandeln.

Gelegentlich aber kann es vorkommen, dass eine solche automatische Umwandlung nicht so funktioniert, wie erwartet.

In dem Fall muss der Wert von Hand umgewandelt werden. Man spricht dabei vom Typecasting.

Generell sollte immer, wenn ein solches Problem erkannt wird, manuelles Typecasting eingesetzt werden, statt sich auf die Automatismen zu verlassen – man weiß schließlich nie, wie zuverlässig die vielen Browser dieser Welt Typecasting einsetzen.

```
// Eine beliebige Variable in Number umwandeln
var x = Number(y);

// Eine beliebige Variable explizit in String umwandeln
var x = String(y);

// Eine beliebige Variable explizit in Boolean umwandeln
var x = Boolean(y);
```

### Umwandlungstabelle

Bei der automatischen Umwandlung von Datentypen ist nicht immer ganz klar, welcher Wert eines Datentyps in welchen Wert eines anderen Datentyps umgewandelt wird. Daher hier eine Übersicht.

x	Boolean(x)	Number(x)	String(x)
undefined	false	NaN	'undefined'
null	false	0	'null'
true (boolean)		1	'true'
false (boolean)		0	'false'
-1	true		'-1'
0	false		'0'
1	true		'1'
42	true		'42'
12.34	true		'12.34'
NaN	false	NaN	'NaN'
" (leerer String)	false	0	
'Hallo'	true	NaN	
'true' (string)	true	NaN	
'false' (string)	true	NaN	
'0'	true	0	
'42' (string)	true	42	

'12.34' (string)	true	12.34	
'12,34'	true	NaN	
Date() (object)	true	NaN	'Thu Aug 10 2017 13:53:57 GMT+0200'
{a:1,b:2}	true	NaN	'[object Object]'
[1,2,3,4]	true	NaN	1,2,3,4

## Typ erkennen

Unter Umständen ist es hilfreich, den Datentypen einer Variablen zu kennen. Dafür gibt es den Javascript-Befehl `typeof`.

```
var x;
console.log ( typeof ( x ) ); // undefined

x = 'Hallo Welt';
console.log ( typeof ( x ) ); // string

x = 42;
console.log ( typeof ( x ) ); // number

x = NaN;
console.log ( typeof ( x ) ); // number

x = true;
console.log ( typeof ( x ) ); // boolean

x = function(){ };
console.log ( typeof ( x ) ); // function

x = [1, 2, 3, 4 ];
console.log ( typeof ( x ) ); // object. Auch Arrays sind Objekte

x = { a: 1, b: 2 };
console.log ( typeof ( x ) ); // object

x = null;
console.log ( typeof ( x ) ); // object
```

## Hoisting

In Javascript sind Variablen und Funktionen 'hoisted' (dt. etwa: 'gehisst', wie eine Flagge hissen'). Dieses oft missverstandene Konzept bedeutet: Der Code wird vor Ausführung nach Variablen und Funktionen durchsucht. Jede Variable und Funktion wird automatisch am Anfang des Sichtbarkeitsbereichs (->Scope) angelegt.

**Aber:** Diese gehoistete Variable/Funktion wird nicht automatisch mit einem Wert belegt. Sie ist solange undefiniert, bis sie schließlich im Code angelegt wird. Es empfiehlt sich also nach wie vor, eine Variable / Funktion erst anzulegen und dann zu benutzen.

```
alert ( x );      // undefined  
var x = 100;  
alert ( x );      // hundert
```

## Löschen von Variablen

In Javascript ist das Löschen von Variablen nicht vorgesehen.

Einzig Attribute von Objekten können mit dem folgenden Code gelöscht werden:

```
delete meinObjekt.meinAttribut;
```

**Achtung:** Das Löschen eines Attributes gibt nicht den Speicher frei sondern entfernt nur die Referenzierung vom Objekt zum Speicherplatz.

# Ein- und Ausgabe

In Javascript gibt es mehrere Möglichkeiten, mit dem Benutzer zu interagieren.

## Alert / Confirm / Prompt

Die einfachste Möglichkeit der Interaktion sind sicherlich diese extra dafür erfundenen Fenster.

Der **Alert**-Befehl öffnet ein kleines Fenster, welches einfach einen String ausgibt. HTML-Tags werden in diesem Fenster als Quellcode ausgegeben. Ein kleines Manko beim Alert ist, dass dieser in manchen Browsern die Aktivität vollständig lahmlegt. Wenn also ein Tab dieses Browsers ein Alert-Fenster öffnet, dann muss das erstmal geschlossen werden, um den Browser weiter zu verwenden. Dadurch wurde schon so mancher Benutzer zur Verzweiflung getrieben.

```
// Der Alert gibt einfach einen String aus.  
alert('Dies ist eine Ausgabe');
```

Das **Confirm**-Fenster funktioniert ähnlich wie das Alert-Fenster. Es bietet aber zusätzlich einen Abbrechen-Button an. Je nachdem, welcher der Buttons gedrückt wurde, liefert das Fenster true (OK) oder false (Abbrechen) zurück. Dadurch kann das Programm auf die Eingabe reagieren.

```
var eingabe = confirm('Bitte bestätigen');  
if(eingabe){  
    alert('OK');  
}else{  
    alert('Abgebrochen');  
}
```

Ein **Prompt**-Fenster verlangt vom Benutzer eine Eingabe und liefert die Benutzereingabe wieder zurück.

```
// Die Eingabe des Benutzers wird hier in der Variablen Eingabe gespeichert  
var eingabe = prompt('Beschreibung', 'Vorbelegung');
```

## Formularfelder

### Textfelder / Textareas

Wir können auch die in HTML vorhandenen Formularfelder verwenden, um Benutzereingaben einzulesen oder Ergebnisse auszugeben.

```
<html><head>  
  <script type="text/Javascript">  
    function meineFunktion(){  
      var element1 = document.querySelectorAll('.eing')[0]; //Textarea  
      ansprechen
```

```

    var element2 = document.getElementsByClassName('eing')[1]; //Input
    ansprechen

    var x = element1.value;           // Inhalt des Textareas auslesen
    var y = element2.value;           // Inhalt des Inputfeldes auslesen

    element1.value = 'neuer Text';    // Inhalt des Textareas ändern
    element2.value = 'neuer Text';    // Inhalt des Inputfeldes ändern
}
</script>
</head><body onload="meineFunktion()">
    <textarea id="eingabel" class="eing" name="eingabel"></textarea>
    <input id="eingabe2" class="eing" name="eingabe2">
</body></html>

```

## Radiobuttons / Checkboxes

Neben Textfeldern können auch **Radiobuttons** und **Checkboxes** angesprochen werden. Hier kann die Eigenschaft „checked“ gelesen und geändert werden. Es handelt sich dabei um eine boolsche Eigenschaft, sie kann also nur true oder false sein.

```

<html><head>
<script type="text/Javascript">
    function meineFunktion(){
        var element = document.getElementsByName('geschl');

        // Prüfen, welcher Radiobutton aktiv ist
        for ( var i = 0; i < element.length; i++ ){
            if (element[i].checked == true){
                alert('Der ' + (i + 1) + 'te Radiobutton ist aktiv.');
            }
        }

        // Ersten Radiobutton aktivieren
        element[0].checked = true;
    }
</script>

</head><body onload="meineFunktion();">

    <label for="wbl">Weiblich: </label>
    <input id="wbl" name="geschl" type="radio">

    <label for="mnl">Männlich: </label>
    <input id="mnl" name="geschl" type="radio">

    <label for="na">Unentschlossen: </label>
    <input id="na" name="geschl" type="radio">

</body></html>

```

Ein kleiner Hack aus der CSS-Welt erlaubt uns, **ohne Schleife** auf den ausgewählten Radiobutton zuzugreifen.



```
// Ohne Schleife die ausgewählte Checkbox auswählen
var element =
document.querySelectorAll('input[name="auswahl"]:checked')[0];

// Für die Ausgabe des Value muss dieses Attribut natürlich vergeben sein.
alert( element.value );
```

## Auswahlboxen

Für **Auswahlboxen** lassen sich der Index und der Inhalt des ausgewählten Feldes ausgeben und verändern sowie neue Felder hinzufügen und bestehende entfernen.

Die folgenden Beispiele gehen von dem folgenden HTML-Aufbau aus:

```
<select id="auswahl" size="1">
  <option>Eins</option>
  <option>Zwei</option>
  <option>Drei</option>
</select>

// Vorbereitung für die folgenden Codes
var auswahl = document.getElementById ( 'auswahl' );
```

### Index der ausgewählten Option ausgeben:

```
alert ( auswahl.selectedIndex );
```

### Option mit dem Index 0 wählen:

```
auswahl.selectedIndex = 0;
```

### Wert der ausgewählten Option ausgeben:

```
alert ( auswahl.value );
```

### Option mit dem Wert 'eins' wählen

```
auswahl.value = 'eins';
```

### Neue Option einfügen:

```
var NeuerEintrag = new Option('Hallo ' + Math.floor(Math.random()*100));
auswahl.add ( NeuerEintrag );

// Alternativ kann auf die DOM-übliche Weise ein neues Element neu erzeugt
werden
var neuerEintrag2 = document.createElement ( 'option' );
neuerEintrag2.text = 'Hallo ' + Math.floor(Math.random()*100);
neuerEintrag2.selected = true;
auswahl.appendChild ( neuerEintrag2 );
```

### Option entfernen

```
if ( auswahl.length > 0 ){
    auswahl.remove(0);
}
```

# Verzweigungen

Anwendungsbeispiele:

- Entscheidung auf Basis einer logischen Aussage, welcher Code ausgeführt werden soll.
- Zu Allgemein, um gute Beispiele zu nennen.
- U.v.m.

## Logische Operatoren

Die Basis aller Verzweigungen (und Schleifen) sind logische Aussagen. Eine logische Aussage ist eine Aussage, die nur wahr (true) oder falsch (false) sein kann.

Das Ergebnis einer logischen Aussage ist also ein boolescher Wert.

Trivial gesprochen wäre eine logische Aussage: "Es ist jetzt nach 12Uhr". Das kann objektiv und eindeutig als wahr oder falsch beantwortet werden.

Beim Programmieren wird die Aussage folgendermaßen geschrieben:

```
new Date().getHours() > 12; // -> true oder false
```

Mögliche Operatoren, um eine solche logische Aussage zu treffen sind:

==	Ist gleich	42 == 42 // -> true
===	Ist gleich (prüft auch Datentyp)	42 === '42' // -> false
!=	Ist ungleich (prüft auch Datentyp)	42 != '42' // -> true
!==	Ist ungleich	42 !== 42 // -> false
<	Kleiner als	42 < 42 // -> false
<=	Kleiner oder gleich	42 <= 42 // true
>	Größer als	42 > 42 // false
>=	Größer oder gleich	42 >= 42 // true

## Logische Verknüpfungen

Wenn mehrere logische Aussagen gleichzeitig geprüft werden müssen, dann können diese mit *UND* (&&) bzw. *ODER* (||) verknüpft werden.

Für eine UND-Verknüpfung gilt, dass sie nur dann wahr ist, wenn beide logischen Aussagen wahr sind. Eine ODER-Verknüpfung ist dann wahr, wenn mindestens eine der beiden logischen Aussagen wahr sind.

true && true	true
true && false	false
true    true	true
true    false	true
false    false	false

### Beispiel:

```
let x = 42;
console.log ( x < 50 && x > 40);    // -> true
```

Sollten in einer logischen Verknüpfung sowohl UND als auch ODER vorkommen, so werden die UND-Operatoren als erste aufgelöst.

```
console.log ( false && true || true );    // -> true
console.log ( false && ( true || true) ); // -> false
```

### Logische Operatoren mit Nicht-Booleans

Javascript erwartet nicht unbedingt zwei Booleans. Typumwandlungen werden automatisch vollzogen, wobei das Ergebnis manchmal etwas irritieren kann.

Denn es wird immer der Operator zurückgegeben, der das Ergebnis bestimmt hat. Auch, wenn dieser kein Boolean ist.

So ist das Ergebnis einer UND-Operation der erste Operator, wenn dieser false ist:

```
console.log ( false && "Hallo Welt" );    // -> false
console.log ( 0 && true );                  // -> 0
```

Wenn bei einer UND-Operation der erste Operator ein TRUE ist (auch, wenn sich dieses erst nach einer Typumwandlung ergibt), so bestimmt der zweite Operator automatisch das Ergebnis. Javascript gibt dann den zweiten Operator zurück.

```
console.log ( true && 'Hund' );            // -> Hund
console.log ( 84 && 12 );                   // -> 12
```

Wenn bei einer ODER-Operation der erste Operator ein true ist, so ist dieser automatisch das Ergebnis.

```
console.log ( false || 'Hund' );          // -> Hund
```

Wenn bei einer ODER-Operation der erste Operator ein false ist, so bestimmt der zweite Operator automatisch das Ergebnis.

```
console.log ( 0 || 12 );           // -> 12
```

## If

Eine eindeutige logische Bedingung, die nur mit 'wahr' oder 'falsch' beantwortet werden kann (boolean), bestimmt die Richtung.

```
if ( Bedingung ) {  
    // Wird ausgeführt, wenn die Bedingung erfüllt wird  
} else if (andere Bedingung){  
    // Wird ausgeführt, wenn nicht die erste, aber diese Bedingung  
    erfüllt wird.  
} else {  
    // Wird ausgeführt, wenn kein der vorigen Bedingungen erfüllt werden.  
}
```

Die Angabe von *else if* und *else* sind optional.

### Kurzschreibweise

Wenn der auszuführende Code nur aus einem einzigen Befehl besteht, können die geschweiften Klammern auch weggelassen werden.

```
if ( Bedingung ) // Ein einzelner Befehl  
else if (andere Bedingung) // Ein einzelner Befehl  
else // Ein einzelner Befehl
```

## Switch

Die Switch-Anweisung nimmt die Verzweigung, die dem Inhalt der Angabe entspricht. Der Vorteil gegenüber der If-Anweisung ist, dass viele mögliche Werte unterschiedliche Ergebnisse liefern können.

Die *break*;-Befehle sind notwendig, um die Case-Bedingungen voneinander abzugrenzen.

```
switch (meineVariable){  
    case 'fall1':  
        // Wird ausgeführt, wenn meineVariable den Inhalt 'fall1' hat  
        break;  
    case 'fall2':  
        // Wird ausgeführt, wenn meineVariable den Inhalt 'fall2' hat  
        break;  
    case 'fall3':  
    case 'fall4':  
    case 'fall5':  
        // Wird ausgeführt, wenn meineVariable  
        // den Inhalt 'fall3', 'fall4' ODER 'fall5' hat  
        break;
```

```
default:
    // Wird ausgeführt, wenn keine der oberen Case-Bedingungen zutreffen.
}
```

## Ternärer Operator

Es gibt eine interessante Möglichkeit, den Inhalt einer Variablen vom Ergebnis einer logischen Aussage zu machen.

Ist die Aussage (im Beispiel `Math.random() > .5`) wahr, so bekommt die Variable den ersten Wert hinter dem Fragezeichen ('a'). Ist die Aussage falsch, bekommt die Variable den zweiten Wert ('b').

```
let x = Math.random() > .5 ? 'a' : 'b';
console.log ( x );
```

# Schleifen

Anwendungsbeispiele:

- Beliebig oft wiederholte Ausführung bestimmten Codes.
- Hochzählen einer Variablen und Ausführung von Code mit dem jeweiligen Inhalt.
- U.v.m.

Eine Schleife ist eine Kontrollstruktur, die es uns ermöglicht, einen Teil unseres Codes mehrfach auszuführen.

## While-Schleife

Eine While-Schleife ist für bestimmte Zwecke leichter zu verwenden, für eine Zählschleife aber (die geschätzte 90% aller Schleifen ausmachen) ist sie etwas unhandlich.

In einer While-Schleife wird einfach nur eine Bedingung geprüft und im Falle, dass sie stimmt, der Codeblock ausgeführt.

```
var b = true;

while ( b ){

    // Beliebig viel Code
    if( irgendeine Bedingung ){
        b = false;
    }
}
```

### Kurzschreibweise

Wenn der auszuführende Code nur aus einem einzigen Befehl besteht, können die geschweiften Klammern auch weggelassen werden.

```
var zahl = 0;
while ( zahl < 42 )    document.write(zahl++ + ', ');
```

## For-Schleife.

```
for( Zähler anlegen; Bedingung; Zähler manipulieren ){
    // Dieser Code wird in jedem Schleifendurchlauf einmal ausgeführt
}
```

Zwischen die Angaben in der runden Klammer gehören Semikola.

Es müssen nicht alle Angaben in der runden Klammer gemacht werden, die trennenden Semikola müssen aber eingefügt werden.

```
// Eine einfache Schleife, die 100mal durchläuft
for ( var i = 0; i < 100; i++ ){
    // Code
}

// Eine Schleife, deren Bedingung anders als durch Zählen manipuliert wird
for (var b = true; b; ){
    if( irgendeine Bedingung ){
        b = false;
        // Zu diesem Zweck könnte man natürlich auch eine while-Schleife
        // verwenden, dies dient dem Aufzeigen der Möglichkeiten
    }
}
```

### Kurzschreibweise

Wenn der auszuführende Code nur aus einem einzigen Befehl besteht, können die geschweiften Klammern auch weggelassen werden.

```
for ( var zahl = 0; zahl < 42; zahl++ ) document.write(zahl + ', ');
```

## Do-While-Schleife

In seltenen Fällen (mir fällt kein Beispiel ein) kann es hilfreich sein, eine Schleife zu bauen, deren Codeblock mindestens einmal durchlaufen wird, bevor die Bedingung überhaupt geprüft wird. Dafür wurde die Do-While-Schleife erdacht.

```
var b = true;
do {
    // Beliebig viel Code
    if( irgendeine Bedingung ){
        b = false;
    }
} while( b );
```

Da die Bedingung am Ende des Codeblocks steht, spricht man hier auch von einer "fußgesteuerten Schleife". Im Gegensatz zu einer "kopfgesteuerten Schleife".

## Manipulation der Schleifenausführung

### Schleife abbrechen

Jede Schleife kann mit dem Befehl `break` abgebrochen werden.

Der Code im Schleifenkörper nach dem `break` wird dann nicht mehr ausgeführt und das Programm läuft hinter der Schleife weiter.

```
for ( var i = 0; i < 6; i++ ){
    document.write ( i );
    if ( i == 2 ){
```



```

    break;
}
document.write ( ' hinter dem <i>break</i><br>' );
}

/*
Ergebnis:
0 hinter dem break
1 hinter dem break
2
*/

```

## Eine Ausführung überspringen

Der Befehl *continue* bricht die Ausführung des Codes im Schleifenkörper sofort ab und springt zur nächsten Ausführung der Schleife weiter.

```

for ( var i = 0; i < 6; i++ ){
    document.write ( i );
    if ( i == 2 ){
        continue;
    }
    document.write ( ' hinter dem <i>break</i><br>' );
}

/*
Ergebnis:
0 hinter dem break
1 hinter dem break
23 hinter dem break
4 hinter dem break
5 hinter dem break
*/

```

## Abbrechen verschachtelter Schleifen

In einer verschachtelten Schleife wird durch *break* bzw. *continue* die innere Schleife abgebrochen.

```

for ( var i = 0; i < 10; i++ ){
    for ( var j = 0; j < 10; j++ ){        // Diese Schleife wird abgebrochen
        console.log ( i + ' * ' + j + ' = ' + (i*j) );
        if ( i == 3 ) break;
    }
}

```

Falls wir eine weiter außen liegende Schleife abbrechen wollen, müssen wir diese markieren und im *break* bzw. *continue*-Befehl darauf beziehen.

```

wombat:for ( var i = 0; i < 10; i++ ){    // Diese Schleife wird abgebrochen
    for ( var j = 0; j < 10; j++ ){
        console.log ( i + ' * ' + j + ' = ' + (i*j) );
        if ( i == 3 ) break wombat;
    }
}

```

```
}  
}
```

# Funktionen

Anwendungsbeispiele:

- Übersichtlichkeit des Codes erhöhen.
- Denselben Code von verschiedenen Stellen im Programm aufrufen.
- Denselben Code mit unterschiedlichen Vorgaben (Parametern) ausführen.
- Zeitverzögerte Ausführung von Code.
- In bestimmten Zeitabständen wiederholte Ausführung von Code.
- U.v.m.

Funktionen sind ein wesentlicher Bestandteil von Javascript.

Sie dienen dem Zusammenfassen und Zugänglichmachen von einzelnen Programmbestandteilen und Unterprogrammen.

Funktionen werden in Javascript als sog. First Class Funktionen behandelt. D.h., sie können :

- in Variablen abgelegt,
- an andere Funktionen übergeben,
- in einem Objekt gespeichert,
- zur Laufzeit erzeugt,
- und als anonyme Funktionen verwendet

werden.

## Anlegen einer Funktion

Zum Anlegen einer Funktion gibt es verschiedene Wege.

Grundsätzlich wird eine Funktion so angelegt:

```
function() {  
  // Auszuführender Code  
}
```

Eine solche Funktion nennt man **anonyme Funktion**. Diese kann nicht direkt aufgerufen werden, aber durchaus z.B. in einem [EventListener](#) verwendet werden.

Um eine Funktion ausführen zu können, kann man sie in einer Variablen speichern:

```
var meineFunktion = function() {  
  // Auszuführender Code  
}
```

Eine kürzere Schreibweise sieht so aus:

```
function meineFunktion() {  
  // Auszuführender Code  
}
```

Diese beiden Schreibweisen scheinen dasselbe zu machen (eine Funktion anlegen). Wenn man sich aber mit den Details auseinander setzt, dann gibt es ein paar Unterschiede.

### IIFE (Immediately invoked function expression)

IIFE sind Funktionen, die sofort nach dem Anlegen ausgeführt werden sollen.

```
(function ( meinParameter ) {  
    Console.log ( meinParameter );  
}) (42);
```

<http://javascriptissexy.com/12-simple-yet-powerful-javascript-tips/>

### Parameter

Parameter sind quasi Variablen, die beim Aufruf der Funktion angelegt werden. Sie können innerhalb der Funktion wie jede andere Variable auch verwendet werden.

```
function meineFunktion( parameter1, parameter2 ){  
    // Auszuführender Code z.B.:  
    console.log ( parameter1 * parameter2 );  
}
```

### Aufruf

Eine Funktion kann beliebig oft von einer anderen Stelle des Programms, auch aus einer anderen Funktion heraus, aufgerufen werden.

Zum Aufruf wird einfach der Funktionsname angegeben, in die Klammern kommen die zu übergebenden Werte. Die Werte werden der Reihenfolge nach an die Parameter übergeben.

```
meineFunktion ( 42, 23 );
```

### Wertvorgabe

Falls unklar ist, ob wirklich alle Parameter übergeben werden, haben wir die Möglichkeit, Werte vorzugeben. Falls beim Aufruf der Funktion ein Parameter nicht übergeben wird, so nimmt die Funktion dann die Vorgabe.

Zu beachten ist dabei, dass die übergebenen Werte der Reihenfolge nach den Parametern zugewiesen werden. Daher sind leere Parameter (also das Weglassen derselben) nur am Ende der Parameterliste sinnvoll.

```
function rechne(a = 1, b = 1, c = 1){  
    var erg = a * b + c;  
    alert( erg );  
}  
rechne(100);          // Gibt 101 aus, da die leeren Parameter  
// jeweils mit 1 besetzt werden. 100 * 1 + 1 = 101
```

## Arguments-Objekt

Alle Parameter, die an eine Funktion übergeben werden, stehen unabhängig von den Parameterbezeichnungen auch im Arguments-Objekt zur Verfügung. Auch Parameter, die in der Funktion nicht angegeben wurden, stehen im Arguments-Objekt bereit.

Dieses kann verwendet werden wie ein Array, besitzt aber nicht die üblichen Array-Methoden außer `length`.

Das Arguments-Objekt ist vor allem für Funktionen hilfreich, bei denen die Anzahl der übergebenen Parameter beim Programmieren unbekannt ist.

```
function summe () {  
    var summe = 0;  
    for ( var i = 0; i < arguments.length; i++ ) {  
        summe += arguments [ i ];  
    }  
    return summe;  
}  
  
alert ( summe ( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ) );
```

## Rest Parameter



[\(Strg-Klick für mehr\)](#)

Etwas moderner als das Arguments-Objekt ist die Schreibweise der Rest Parameter. Sie können eingesetzt werden, wenn unklar ist, wie viele Parameter an eine Funktion übergeben werden.

Wenn der letzte benannte Parameter einen Prefix von drei Punkten hat, dann wird dieser als Array angelegt, das alle Parameter enthält, die keinen eigenen Parameter abbekommen haben.

Der Rest halt.

Haben wir keinen weiteren Parameter neben dem mit dem Prefix, dann enthält dieser natürlich alle Parameter.

```
function meineFunktion ( p1, p2, ...args ) {  
    console.log ( p1, p2 );  
    console.log ( args );  
}  
  
meineFunktion ( 1, 2, 3, 4, 5 );
```

## Sichtbarkeit / Scope

Parameter und Variablen, die innerhalb einer Funktion angelegt werden, sind außerhalb der Funktion nicht vorhanden. Um außerhalb der Funktion auf Variablen in der Funktion zuzugreifen, sind diese vorher außerhalb der Funktion anzulegen. Dann können sie in der Funktion manipuliert und außerhalb der Funktion weiterverwendet werden.

```
var globaleVariable = 100;  
  
function funktionsName( parameter ) {
```

```

    var meineVariable;
    globaleVariable++;
}

funktionsName( 100 );
alert( parameter );           // funktioniert nicht
alert( meineVariable );       // funktioniert nicht
alert( globaleVariable );     // gibt 101 aus

```

## Rekursion

Eine Funktion kann auch sich selbst aufrufen. Hier ist aber Vorsicht geboten, da dies in einer unendlichen Rekursion enden kann. Die meisten sind dagegen gewappnet, aber darauf sollte man sich nicht verlassen.

```

function meineFunktion( parameter ){
    alert( parameter );
    meineFunktion( parameter );
}
meineFunktion( 1 );

```

## Rückgabewerte

```

function funktionsName(){
    // Irgendwelcher Code
    return irgendwas;
}
var ding = funktionsName();

```

Der Befehl `return` beendet die Ausführung des Codes an der Stelle. Alles, was danach kommt, wird nicht ausgeführt.

Der Wert bzw. die Variable, die bei dem Befehl `return` steht, wird an den Funktionsaufruf zurückgegeben. Hier wird kann der Rückgabe weiterverwendet werden. Im Beispiel wird der Rückgabewert in die Variable *ding* geschrieben.

Es kann nur ein einziger Wert zurückgegeben werden. Sollte es einmal unumgänglich sein, mehrere Werte zurückzugeben, so muss man auf ein Array oder ein Objekt ausweichen.

## Zeitsteuerung

Der Aufruf von Funktionen lässt sich auch zeitlich steuern.

### Timeout

```

function funktionsName( parameter1, parameter2 ){
    // irgendwelcher Code
}
var timerID = setTimeout( funktionsName, 1000, 'wert1', 'wert2' );
// Irgendwelcher Code
clearTimeout( timerID );

```

Mit dem Timeout wird die Funktion *funktionsName* nach 1000ms aufgerufen.  
setTimeout selbst ist ebenfalls eine Funktion und sie hat die ID des Timers als Rückgabewert. Dieser wird in einer Variablen zwischengespeichert, um diesen Timeout ggf. abbrechen zu können.  
Zu übergebende Parameter werden in der Klammer nach der Wartezeit eingetragen.

## Interval

```
function funktionsName( parameter1, parameter2 ){  
    // irgendwelcher Code  
}  
var timerID = setInterval( funktionsName, 1000, 'wert1', 'wert2');  
clearInterval( timerID );    // Achtung: Diese Zeile ist ziemlich dumm,  
s.u.
```

Ein Intervall ruft die Funktion nach definierten Zeitabständen immer wieder auf, bis sie per clearInterval beendet wird.

Die Syntax ist identisch mit dem Timeout.

**Achtung:** Die Angabe der Millisekunden ist nicht wirklich exakt. Es gibt zu viele Faktoren, die den Takt beeinflussen. Wenn die Zeit genau gemessen werden muss, sollte auf die Date-Objekte zurückgegriffen werden.

## Hoisting

Siehe

Hoisting

## Arrow-Funktionen



[\(Strg-Klick für mehr\)](#)

Seit ECMA-Script 2015 gibt es in Javascript sog. Arrow-Funktionen.

Dabei handelt es sich quasi um vereinfachte Funktionen, die dank der Vereinfachung ein paar angenehme Eigenschaften aufweisen:

- Pro: weniger Code zu schreiben
- Pro: kein eigenes this, daher bleibt das this aus dem aufrufenden Sichtbarkeitsbereich sichtbar
- Pro: kein eigenes Arguments-Objekt
- Pro: Sehr einfache Wertrückgabe in der Kurzschreibweise
- Kontra: Arrow-Funktionen sollten nicht für Methoden verwendet werden.
- Kontra: aus Arrow-Funktionen kann kein Konstruktor gebildet werden.
- Kontra: kein eigenes this, für Eventlistener muss mit dem event-Objekt gearbeitet werden

Eine Arrow-Funktion wird folgendermaßen geschrieben:

```
( parameter1, parameter2 ) => {  
  // Mach was cooles mit den Parametern  
}
```

Damit die Funktion auch später aufgerufen werden kann, weisen wir sie einer Variablen zu:

```
var meineFunktion = ( wert1, wert2 ) => {  
  wert1 *= wert2;  
  alert( wert1 ); // Gibt '36' aus  
}  
  
meineFunktion ( 12, 3 );
```

Wenn nur ein Parameter übergeben wird, kann die runde Klammer weggelassen werden.  
Wenn der Codeblock nur aus einer Zeile besteht, kann die geschweifte Klammer weggelassen werden:

```
var meineFunktion = wert1 => alert( wert1 * 2 );  
meineFunktion ( 12 );
```

Da Arrow-Funktionen das `this` aus dem übergeordneten Sichtbarkeitsbereich verwenden, kann der folgende Code funktionieren. Jede Sekunde wird das Alter um 1 hochgezählt.

```
function Person(){  
  this.age = 0;  
  
  setInterval(() => {  
    this.age++; // |this| bezieht sich weiterhin auf das Person-Objekt  
  }, 1000);  
}  
  
var p = new Person();
```



# Arrays (Collection-Datentyp)

Anwendungsbeispiele:

- Speichern von vielen, gleichartigen Daten. Z.B.
- Wetterdaten, Musiksammlung, Einwohner, Spielfiguren, Produkte, etc.
- Organisation von Spielfiguren auf einem Spielfeld
- Inhalte von Tabellen
- U.v.m.

Um große Datenmengen zu speichern, eignen sich normale Variablen unter Umständen nicht optimal. Werden aus einem Sensor bspw. Millionen von Daten geliefert, ist es langsam und unhandlich, jedes Datum in eine eigene Variable zu legen. Zu dem Zweck wurden Arrays erfunden. Ein Array ist eine nummerierte Liste von Variablen.

Die Nummerierung beginnt bei 0.

```
// Anlegen eines Arrays
var A_werte = new Array();
// Alternativ
var A_werte2 = [];

// Anlegen eines Array mit Werten
var A_werte2 = new Array(42, 815, 654);
```

Die Speicherplätze im Array werden mit der eckigen Klammer angesprochen.

```
// Auslesen des ersten Speicherplatzes
alert( A_werte2[0] );
// Überschreiben eines Speicherplatzes mit einem neuen Wert
A_werte[0] = 1000;
```

Um große Datenmengen in ein Array zu schieben, eignet sich eine for-Schleife hervorragend

```
// Array mit zufälligen Zahlen befüllen
for (var i = 0; i < 100; i++){
    A_werte[i] = Math.random() * 1000;
}
```

## Länge

Um die Länge eines Arrays zu bestimmen, kann auf die length-Eigenschaft zurückgegriffen werden.

```
// Länge des Array ausgeben
alert( A_werte.length );
```

```
// Die length-Eigenschaft kann auch "missbraucht" werden,  
// um z.B. die letzte Speicherstelle auszugeben  
alert( A_werte[ A_werte.length - 1] );
```

## Iterieren: Alle Speicherstellen ansprechen

Es gibt vier Möglichkeiten, um jede Speicherstelle eines Arrays anzusprechen.

Die Standard-Methode verwendet eine **ganz normale Schleife**:

```
var A_mein = new Array ( 'Januar', 'Februar', 'März', 'April', 'Mai' );  
for ( var i = 0; i < A_mein.length; i++ ) {  
    document.write( A_mein[i] );  
}
```

Mit der **for-of**-Schleife werden alle Speicherstellen des Arrays angesprochen und in der Schleife unter dem Variablennamen zur Verfügung gestellt.



[\(Strg-Klick für mehr\)](#)

**Achtung 1:** Speziell ältere Browser halten sich nicht unbedingt an die Reihenfolge der Indizes.

**Achtung 2:** Durch Prototyping veränderte Arrays haben eine andere Länge und können dadurch zu Problemen führen.

**Achtung 3:** Die Variable ist nur lesbar. Ein Überschreiben der Variable (im Beispiel "p") führt zu keiner Änderung des Arrays.

```
var A_mein = new Array ( 'Januar', 'Februar', 'März', 'April', 'Mai' );  
for ( var p of A_mein ) {  
    document.write ( p + '<br>' );  
}
```

Die **for-in**-Schleife funktioniert ähnlich der for-of-Schleife, allerdings liefert sie nicht die Inhalte der Speicherplätze sondern lediglich deren Indizes.

```
var A_mein = new Array ( 'Januar', 'Februar', 'März', 'April', 'Mai' );  
for ( var p in A_mein ) {  
    document.write ( A_mein [ p ] + '<br>' );  
}
```

Die **forEach-Methode** ist eine Methode, um für jede Speicherstelle eines Arrays eine Funktion aufzurufen. Die forEach-Methode bekommt als einzigen Parameter die Funktion, die aufgerufen wird. Die Funktion wird dann von der Methode für jede Speicherstelle einmal aufgerufen und bekommt bis zu drei Parameter übergeben (siehe Beispiel).

```
var A_mein = new Array ('Januar', 'Februar', 'März', 'April', 'Mai');
function f_foreach(wert, index, array){
    document.write(index, ': ', wert, '<br>');
}
A_mein.forEach( f_foreach );
```

Die **entries-Methode** erzeugt ein *Iterator*-Objekt, mit welchem die Einträge durchlaufen werden können.

```
var A_mein = ['a', 'b', 'c'];
var i = A_mein.entries();
while ( !(o = i.next()).done ) console.log ( o.value );
```

Die **values-Methode** erzeugt ein *Iterator*-Objekt, mit welchem die Einträge durchlaufen werden können.

**Achtung:** die values-Methode funktioniert nur in sehr aktuellen Browsern (Stand 6/18)

```
var A_mein = ['a', 'b', 'c'];
var i = A_mein.values();
while ( !(o = i.next()).done ) console.log ( o.value );
```

## Array-Methoden

### Array verlängern

Die Funktion push hängt an das Array beliebig viele neue Werte an.

```
A_mein.push( wert1, wert2, wert3);
```

### Elemente entfernen/einfügen

Es gibt eine spezielle Funktion, die dem Array an beliebiger Stelle Speicherstellen entfernen und einfügen kann. Diese Splice-Funktion benötigt mindestens zwei Parameter.

Der erste Parameter gibt an, ab welchem Index Inhalte entfernt/eingefügt werden sollen. Der zweite Parameter gibt an, wieviele Speicherstellen entfernt werden sollen. Ab dem dritten Parameter folgen Werte, die ab dem Index in Parameter1 eingefügt werden sollen.

```
A_mein.splice(beginn, anzahl, wert1, wert2, wert3);
// Beispiel: ab Index 5 ein Wert einfügen, ohne einen zu entfernen
A_mein.splice(5, 0, wert1);
// Beispiel: ab Index 5 nur einen Wert entfernen, nichts einfügen
A_mein.splice(5,1);
// Beispiel: Den Wert an Index 5 durch einen anderen ersetzen
A_mein(5, 1, Wert1);
```

## Inhalt suchen

Ähnlich wie in einem String können wir auch in einem Array prüfen, an welchem Index in bestimmter Inhalt das erste Mal vorkommt.

Dieser **indexOf** prüft nur, ob der Inhalt eines Speicherfeldes genau der Abfrage entspricht. Teilstrings werden nicht berücksichtigt.

Kommt der gesuchte Wert in dem String nicht vor, liefert die Abfrage das Ergebnis -1.

```
var A_meinArray = ['hallo', 'welt', 'wetter', 'gut'];
alert( A_meinArray.indexOf('wetter') ); // => 2
alert( A_meinArray.indexOf('wet') );    // => -1
```

Die Methode **lastIndexOf()** sucht das letzte Vorkommen eines Suchbegriffs.

```
var A_meinArray = ['hallo', 'welt', 'wetter', 'hallo', 'gut'];
alert( A_meinArray.lastIndexOf('hallo') ); // => 3
```

Die **includes()**-Methode prüft, ob in einem Array ein bestimmter Inhalt überhaupt vorkommt. Diese Methode liefert einen Boolean (true oder false).



[\(Strg-Klick für mehr\)](#)

```
var A_meinArray = ['hallo', 'welt', 'wetter', 'gut'];
alert( A_meinArray.includes('wetter') ); // => true
```

Um zu prüfen, ob **mindestens eine Speicherstelle einem bestimmten Kriterium entspricht**, wurde die *some*-Methode erdacht. Wenn die übergebene Funktion für mindestens eine der Speicherstellen ein *true* liefert, liefert die *some*-Methode ebenfalls ein *true*.

```
var ages = [4, 12, 16, 20];
var ueber = ages.some ( age => age >= 18 );
console.log ( ueber ); // => true
```

Um zu prüfen, ob **alle Speicherstellen einem bestimmten Kriterium entsprechen**, wurde die *every*-Methode erdacht. Nur, wenn die übergebene Funktion für alle Speicherstellen ein *true* liefert, liefert die *every*-Methode ebenfalls ein *true*.

```
var ages = [24, 32, 16, 20];
var ueber = ages.every ( age => age >= 18 );
console.log ( ueber ); // => false
```

## Inhalt austauschen

Für Arrays gibt es keine fertige Methode wie *replace*.

Aber mit ein wenig Geschick lässt sich eine Suchen-und-Ersetzen-Funktion schaffen:

```
var meinArray = [ 'Albert', 'Berta', 'Xanthippe', 'Erich'];
meinArray.splice ( meinArray.indexOf('Berta'), 1, 'Bärbel' );
alert ( meinArray );
```

## Array kopieren

Der intuitive Versuch, ein Array zu kopieren, sieht meist so aus:

```
var a = [1,2,3,4,5];
var b = a;           // Nur selten das, was man will
```

Auf diese Weise wird allerdings nur eine Verknüpfung erzeugt. Wenn sich also am kopierten Array etwas ändert, so ändert sich das Original gleich mit und umgekehrt.

```
a[1] = 100;
alert (b); // Ausgabe: 1,100,3,4,5
```

Um eine echte Kopie zu erzeugen, legt man mithilfe des Slice-Befehls (Nicht mit splice verwechseln!) ein neues Array an und speichert dieses in einer Variablen. Somit ist das neue Array vollkommen unabhängig vom alten.

Der erste Parameter bestimmt, ab welchem Index die Inhalte kopiert werden. Der zweite Parameter bestimmt, bis zu welchem Index (exklusive) die Elemente kopiert werden. Wenn kein zweiter Parameter angegeben ist, werden alle Elemente, beginnend beim ersten Parameter, kopiert.

```
var a = [1,2,3,4,5];

var b = a.slice(0);
a[0] = 100;
alert (b); // Ausgabe: 1,2,3,4,5

var c = a.slice(1,3);
alert(c); // Ausgabe 2,3
```

## Arrays kombinieren

Um aus zwei Arrays eines zu machen, können diese mit der Concat-Methode aneinandergehängt werden.

```
var a = [1,2,3,4,5];
var b = [65,88,13,35];
var c = a.concat(b);
alert ( c );           // -> 1,2,3,4,5,65,88,13,35
```

Die **Spreadsyntax** macht das sogar noch einfacher und flexibler:

```
var dasArray = [0, 1, 2];
var array2 = [3, 4, 5];

dasArray.push( ...array2 );
```

```
console.log ( dasArray );
```

### String zu Array wandeln

Um ein String zu einem Array umzuwandeln, gibt es die split-Methode.

Sie wird auf den String angewendet und trennt diesen dort, wo der übergebene Trennstring vorkommt. Der Trennstring wird entfernt.

```
var t = 'Hans und Peter und Sandra und Tanja';  
var a = t.split ( ' und ' );  
alert ( a ); // Hans,Peter,Sandra,Tanja
```

### Array zu String wandeln

Der join-Befehl hängt alle Elemente eines Arrays als String zusammen. Der übergebene Parameter wird als Trenn-Zeichenkette verwendet.

```
var a = [ 'Banana', 'Orange', 'Apple', 'Mango' ];  
var t = a.join ( ' und ' );  
alert ( t ); // Banana und Orange und Apple und Mango
```

### Array umdrehen

Falls ein Array umgekehrt werden muss, kann die Reverse-Methode gute Dienste

```
var a = [1,2,3,4,5];  
alert ( a ); // Ausgabe: 1,2,3,4,5  
a.reverse();  
alert ( a ); // Ausgabe: 5,4,3,2,1
```

### Sortieren

Mit der sort-Methode lässt sich ein Array sortieren. Grundsätzlich findet die Sortierung alphabetisch statt:

```
var meinArray = ['Mazda', 'VW', 'Volvo', 'Hyundai', 'Citroen'];  
meinArray.sort();  
alert(meinArray); // Ausgabe: Citroen,Hyundai,Mazda,VW,Volvo
```

Das ist schön und gut, hat aber ein Problem. Die Sortierung findet alphabetisch, also nach Position der Zeichen in der ASCII-Tabelle, statt. Während die Sortierung oben also noch gut funktioniert, gibt es Probleme bei Arrays, die nur aus Zahlen bestehen. Da kommt dann 10 vor 9, weil die Eins kleiner ist. Das Ergebnis einer solchen 'Sortierung' wäre z.B.: 104, 11, 12, 123, 13, 2, 230, 4, 415, 46

Um dem Problem zu begegnen, haben wir die Möglichkeit, eine eigene Sortierfunktion zu erstellen, die numerisch sortiert:

```
var meinArray = [104, 11, 12, 123, 13, 2, 230, 4, 415, 46];  
meinArray.sort( function(a, b){ return a-b } );
```

```
alert(meinArray); Ausgabe: 2,4,11,12,13,46,104,123,230,415
```

## Für jede Speicherstelle eine Funktion aufrufen

Mit der *forEach*-Methode können wir für jede Speicherstelle eines Arrays eine Funktion aufrufen. Die aufzurufende Funktion wird der *forEach*-Methode als Parameter gegeben.

Die *forEach*-Methode ruft die übergebene Funktion mit jedem Wert einmal auf und übergibt dieser den jeweiligen Wert aus dem Array und den Index.

**Achtung:** Die Funktion wird für leere Speicherstellen nicht aufgerufen.

Im Beispiel wird mit einer Arrow-Funktion gearbeitet, das macht den Code wesentlich kürzer.

```
let meinArray = [ 5, 8, 6, 7 ];
let ausgabe = (wert, index) => console.log( index + ': ' + wert );
meinArray.forEach ( ausgabe );
```

## Jede Speicherstelle manipulieren



[\(Strg-Klick für mehr\)](#)

Um jede Speicherstelle in einem Array mit einem einzigen Befehl zu manipulieren, wurde die Higher Order Methode *map* erdacht.

Als Parameter wird eine Funktion übergeben, die für jede Speicherstelle einmal ausgeführt wird. Diese Speicherstelle wird dann durch den Rückgabewert der Funktion ersetzt.

**Achtung:** Die Funktion wird für leere Speicherstellen nicht aufgerufen.

```
let numbers = [4, 9, 16, 25];
let doubles = numbers.map( function( element ){
  return element * 2;
});
console.log ( doubles );
```

## Speicherstellen filtern

Die *Filter*-Methode gibt ein Array zurück, dass nur die Speicherstellen des Arrays enthält, bei denen der Rückgabewert *true* war.

**Achtung:** Die Funktion wird für leere Speicherstellen nicht aufgerufen.

```
var ages = [32, 14, 33, 16, 40];
var erwachsen = ages.filter(function (age) {
  return age >= 18;
});
console.log ( erwachsen );
```

Die *Filter*-Methode wird gern mit anderen Methoden kombiniert, z.B. um eine Vorauswahl der Speicherstellen zu treffen, die bearbeitet werden sollen.

```
var ages = [32, 14, 33, 16, 40];
ages.filter( age => age >= 18 ).forEach( age => console.log ( age ) );
```

## Array zu einem Wert reduzieren

Die *Reduce*-Methode bekommt eine Funktion als Parameter.

Diese Funktion nimmt zunächst die ersten zwei Speicherstellen als Parameter.

Danach wird sie für jede weitere Speicherstelle mit dem Rückgabewert des vorherigen Funktionsaufrufs und der Speicherstelle erneut aufgerufen.

Wurden so alle Speicherstellen abgearbeitet, dann ist der Rückgabewert das Ergebnis der Methode.

**Achtung:** Die Funktion wird für leere Speicherstellen nicht aufgerufen.

```
let numbers = [2, 4, 8, 16];
let summe = numbers.reduce(function (total, num) {
    return total + num;
});
console.log ( summe );
/*
  2 + 4 = 6
  6 + 8 = 14
  14 + 16 = 30
  => 30
*/
```

## Initialwert

Manchmal unterscheidet sich der Datentyp des Rückgabewertes vom Datentyp der Speicherstellen im Array. Z.B., wenn sich in dem Array Objekte befinden. In dem Fall ist es hilfreich, der *reduce*-Methode einen Initialwert zu geben. Dieser wird nach der Funktion geschrieben.

```
let a = [
    {wert: 1}, {wert: 2}, {wert: 3}, {wert: 4}, {wert: 5}
];

a = a.reduce ( (x,y) => {
    return x + y.wert;
}, 0);

console.log ( a );
```

## Die Spread-Syntax



[\(Strg-Klick für mehr\)](#)

Seit ES6 bietet Javascript eine mächtige Technik zum Restrukturieren von Daten, die sog. Spread-Syntax. Auch Array profitieren davon.



Ein wesentliches Element der Spread-Syntax ist der Prefix ... (*drei Punkte*). Dieser steht für die Anweisung, ein Array oder ein Objekt zu "spreizen" (*spread*) und die Attribute zur Verfügung zu stellen.

Dieser Prefix erinnert an die Rest-Parameter ([Strg-Klick für mehr](#)) bei Funktionsdeklarationen, die beiden haben aber bestenfalls ähnliche Einsatzbereiche: Das Zusammenführen oder Extrahieren von Attributen.

So können wir z.B. ein Array **kopieren**, statt auf die slice-Methode zurückzugreifen:

```
let dasArray = [ 1, 2, 3 ];
let array2 = [ ...dasArray, 10, 11, 12 ];
dasArray[0] = 42;

console.log ( array2.join ( ' , ' ) ); // -> 1, 2, 3, 10, 11, 12
```

Oder wir können ein Array auf einfache Weise an ein anderes **anhängen**:

```
var dasArray = [0, 1, 2];
var array2 = [3, 4, 5];

dasArray.push( ...array2 );

console.log ( dasArray );
```

[https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Operators/Spread_operator)

[http://exploringjs.com/es6/ch\\_destructuring.html#sec\\_rest-operator](http://exploringjs.com/es6/ch_destructuring.html#sec_rest-operator)

## Mehrdimensionale Arrays

Ein Array ist eine (eindimensionale) Liste von Variablen.

Um Dinge wie z.B. Spielfelder oder Tabellen darzustellen, genügt solch eine einfache Liste nicht.

Dafür müssen wir auf sogenannte "mehrdimensionale Arrays" zurückgreifen. Dabei handelt es sich aus technischer Sicht um Arrays innerhalb anderer Arrays.

Wir legen also zunächst ein leeres Array an, in diesem speichern wir weitere Array, welche dann mit den Inhalten befüllt werden.

```
// Array anlegen und mit Werten befüllen
var a = [];
for ( var i = 0; i < 5; i++ ){
  a.push ( [] );
  for ( var j = 0; j < 5; j++ ){
    a[i].push ( Math.floor ( Math.random() * 100 ) );
  }
}
```

Um auf eine bestimmte Speicherstelle zuzugreifen, werden die x- und y-Koordinate der Speicherstelle einfach in zwei aufeinanderfolgenden [] eingetragen.

```
// Ausgabe
var ausgabe = '';
for ( var i = 0; i < a.length; i++ ){
    for ( var j = 0; j < a[i].length; j++ ){
        ausgabe += ( a[i][j] + ', ' );
    }
    ausgabe += ( '<br>' );
}

document.write ( ausgabe );
```

#### Anwendungsbeispiele:

- Speichern von mehreren Eigenschaften eines realen Objektes.
- Autohaus, das jedes Fahrzeug in seiner Software speichern will.
- Firma, die in der Verwaltungssoftware jeden Mitarbeiter mit allen relevanten Informationen speichern will.
- U.v.m.

## Objekte (Collection-Datentyp)

Objekte sind den Arrays in der Grundidee recht ähnlich: Es geht bei beiden darum, viele Variablen in eine leicht verständliche Form zu bringen. Im Gegensatz zu den Arrays, die lange Listen gleichartiger Werte darstellen, sollen Objekte verschiedenartige Attribute zusammenfassen.

Gerne wird das Beispiel einer Person herangenommen, dessen Vorname, Nachname, Geburtsjahr, etc. als Eigenschaften des Objektes *Person* gespeichert werden.

```
// Objekt anlegen
var meinObjekt = {vorname: 'Max', nachname: 'Mustermann', Alter: '44'};

// Glücklicherweise werden Zeilenumbrüche bei der Interpretation ignoriert
// Daher können wir diese recht lange Codezeile übersichtlicher gestalten
var meinObjekt = {
    vorname: 'Max',
    nachname: 'Mustermann',
    geburt: '1978'
};
```

### Eigenschaften ändern/einfügen

Um eine Eigenschaft zu ändern, greifen wir mit dem Punkt als Trennzeichen darauf zu.

Wenn diese Eigenschaft noch nicht existiert, dann wird sie neu angelegt.

```
meinObjekt.vorname = 'Hans';
```

## Dynamische Attributnamen

Es kann vorkommen, dass während des Programmierens nicht klar ist, wie ein Attribut heißen muss. Dann kann in einer eckigen Klammer ein String konstruiert werden, der dann als Attributname verwendet wird.

**Achtung:** Vor der eckigen Klammer steht kein Punkt.

```
var vornamen = ['Hans', 'Max', 'Jürgen', 'Matthias', 'Stefan'];
var person = {};
for(var i = 0; i < 3; i++){
  var zufall = Math.floor(Math.random()*vornamen.length);
  person['vname' + i] = vornamen[ zufall ];
}
alert( person.vname0 + ' ' + person.vname1 + ' ' + person.vname2 );
```

## Methoden

Eine Methode ist nichts anderes als eine Funktion, die in einem Objekt liegt. Sie wird gern verwendet, um ständig wiederkehrende Aktionen zu vereinfachen. Innerhalb einer Methode ist das Objekt mit dem Schlüsselwort *this* ansprechbar.

**Achtung:** Weil eine **Arrow-Funktion** kein *this* kennt, sollte sie nicht als Methode verwendet werden.

```
// Anlegen eines neuen Objektes mit einer Methode
var meinObjekt = {
  zahl: 42,
  ausgabe: function(){
    return this.zahl * 2;
  }
}

// Aufruf der Methode - bitte die Klammern beachten.
alert ( meinObjekt.ausgabe() );    // => 48
```

## Alle Elemente ansprechen

Um in einem Objekt alle Elemente anzusprechen, kann eine for-Schleife mit dem Schlüsselwort *in* in der Bedingung verwendet werden.

Es werden dann alle Speicherstellen des Objektes durchgelaufen.

Innerhalb dieser Schleife kann der Zähler verwendet werden, um auf eine Eigenschaft nach der anderen zuzugreifen.

```
var person = {vname:"John", nname:"Doe", age:25};
for (var i in person) {
    txt += i + ",: " + person[i] + " ";
}
```

Um **alle Schlüssel** in einem Rutsch auszugeben, wurde die *keys*-Methode erdacht. Eine Besonderheit ist, dass sie eine Methode des *Object*-Prototyps ist, aber nicht von dem *Object* selbst aufgerufen werden kann.

Der Rückgabewert der *keys*-Methode ist ein Array, das alle Schlüssel aus dem Objekt enthält

```
let meinObjekt = {
    a: 42,
    b: 23
}
console.log ( Object.keys( meinObjekt ) );    // => [a, b]
```

Um **alle Werte** in einem Rutsch auszugeben, wurde die *values*-Methode erdacht.

Der Rückgabewert der *values*-Methode ist ein Array, das alle Werte aus dem Objekt enthält



[\(Strg-Klick für mehr\)](#)

```
let meinObjekt = {
    a: 42,
    b: 23
}
console.log ( Object.values( meinObjekt ) );    // => [42, 23]
```

Um **alle Schlüssel-Wert-Paare** in einem Rutsch auszugeben, wurde die *entries*-Methode erdacht.

Der Rückgabewert der *entries*-Methode ist ein Array, das alle Schlüssel-Wert-Paare in Form von Arrays enthält.



[\(Strg-Klick für mehr\)](#)

```
let meinObjekt = {
    a: 42,
    b: 23
}
console.log ( Object.entries( meinObjekt ) );    // => [['a',42],['b', 23]]
```

## Objekte kopieren

Eine Besonderheit von Objekten ist, dass sie als **Referenz** arbeiten. Mit anderen Worten: Im folgenden Beispiel ist die Variable *x* nur eine Referenz auf / Verknüpfung mit dem Objekt *person*. Wenn man eines ändert, ändert sich das andere gleich mit.

```
var person = {vname:"John", nname:"Doe", age:25};
var x = person;
person.vname = 'Jim';
alert ( x.vname );
```

Um ein Objekt komplett zu kopieren gibt es keinen einfachen Weg.

Vielmehr musst Du Dir zum Kopieren eines Objektes eine *for in* Schleife (s.o.) verwenden, um eine Eigenschaft nach der anderen in ein neues Objekt zu kopieren.

## Objekte an einander hängen



[\(Strg-Klick für mehr\)](#)

Seit ECMAScript2015 gibt es eine Möglichkeit, Objekte zu konkatinieren.

Dazu wird die *assign*-Methode des Object-Prototyps angewiesen, zwei oder mehr Objekte aneinander zu hängen und der Rückgabewert des Objektes – das resultierende Objekt – wird in eine Variable geschrieben.

**Achtung:** Funktioniert nicht im IE

```
var o = { wert1: 42 };
var x = { wert2: 60, wert3: 'Hallo Welt' };
Object.assign(o, x);
console.log ( o ); // => Object{wert1: 42, wert2: 60, wert3: "Hallo Welt"}
```

## Konstruktoren

Objekte sind prima, um einzelne Objekte anzulegen, die viele Eigenschaften beinhalten. Wenn wir aber mehrere Objekte mit denselben Eigenschaften (sagen wir: Personen in einem Programm für Personalverwaltung) speichern wollten, müssten wir für jede Person ein eigenes Objekt anlegen. Um uns das Leben zu erleichtern, können wir mithilfe eines *Konstruktors* Objekte anlegen. Der stellt bereits alle Eigenschaften bereit, die dann nur noch gefüllt werden wollen.

Einen Konstruktor legen wir als Funktion an, die dann mit Parametern aufgerufen wird. Diese Parameter enthalten die Werte, die in den Attributen des Objektes gespeichert werden.

Üblicherweise werden Konstrukturen mit **großem ersten Buchstaben** geschrieben, um sie auf den ersten Blick von normalen Funktionen zu unterscheiden.

```
// Anlegen des Konstruktors
function Person( vorname, nachname, alter = 0 ){
  this.vorname = vorname;
  this.nachname = nachname;
  this.alter = alter;
}

// Anlegen von Objekten mit Hilfe des Konstruktors
var vater = new Person('Jürgen', 'Schulz', 50);
alert ( vater.vorname ); // Gibt 'Jürgen' aus
```

## Eigenschaften vererben

Es kann vorkommen, dass mehrere Objekttypen gebraucht werden, die einige gemeinsame und andere abweichende Eigenschaften haben. In dem Fall bietet sich die Vererbung von anderen Konstruktoren an.

Dabei wird mit Hilfe der *call*-Funktion ein zusätzlicher Konstruktor aufgerufen, der dem Objekt einige Eigenschaften gibt.

```
function Person ( vname, nname ){
  this.vname = vname;
  this.nname = nname;
}

function Student ( vname, nname, fach ){
  Person.call ( this, vname, nname );
  this.fach = fach;
  this.ausgabe = function(){
    return ( this.vname + ' studiert ' + this.fach );
  }
}

function Arbeiter ( vname, nname, firma ){
  Person.call ( this, vname, nname );
  this.firma = firma;
  this.ausgabe = function(){
    return ( this.vname + ' arbeitet bei ' + this.firma );
  }
}

var max = new Student ( 'Max', 'Mustermann', 'Mathematik' );
alert ( max.ausgabe() );

var thomas = new Arbeiter ( 'Thomas', 'Müller', 'SAP' );
alert ( thomas.ausgabe() );
```

## Klassen



[\(Strg-Klick für mehr\)](#)

Mit ECMAScript 2015 wurden sog. Klassen eingeführt. Die Bezeichnung ist etwas unglücklich, denn diese Javascript-Klassen haben einen anderen Ansatz als die Klassen, die man ggf. aus anderen Programmiersprachen kennt.

Klassen sind eine Alternative zu den oben beschriebenen Konstruktoren, um Objekte zu erstellen.

Eine Klasse legen wir folgendermaßen an:

```
class Person {
  constructor ( vname, nname, geburt ){
```

```

    this.vname = vname;
    this.nname = nname;
    this.geburt = geburt;
}

// Methode anlegen (in Kurzform)
alter() {
    return (new Date().getFullYear() - this.geburt);
}
}

let vater = new Person ( 'Max', 'Mustermann', 1980 );
alert ( vater.alter() );

```

## Klassen vererben

Wie normale Konstruktoren, können auch Klassen vererbt werden. Wir können also Klassen anlegen, die auf anderen Klassen basieren.

Syntaktisch hat sich Javascript an andere Sprachen angenähert und so werden Klassen mit dem Schlüsselwort `extends` vererbt.

Im Beispiel unten wird eine Klasse (Prototyp) namens *Arbeiter* angelegt, die ihre Eigenschaften von *Person* erbt. Damit der Konstruktor in der Klasse *Arbeiter* funktioniert, muss zunächst mit dem Schlüsselwort *super* der Konstruktor der übergeordneten Klasse (*Person*) gestartet werden.

```

class Person {
    constructor ( vorname, nachname ){
        this.vname = vname;
        this.nname = nname;
    }
}

class Arbeiter extends Person {
    constructor(vn, nn, ag){
        super ( vn, nn);
        this.arbeitgeber = ag;
    }
}

var vater = new Arbeiter ( 'Max', 'Mustermann', 'Telekom' );
alert ( vater.arbeitgeber );

```

Mehr dazu: <https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Klassen>

## Geschützte Attribute

Variablen, die in einem Konstruktor (oder einer Klasse) mit dem Schlüsselwort *var* oder *let* angelegt werden, sind nicht als Attribut abrufbar.

```

// Anlegen des Konstruktors
function Person( vorname, nachname, alter ){

```

```

this.vorname = vorname;
this.nachname = nachname;
var alter = alter;    // kein 'this'!
}

// Anlegen von Objekten mit Hilfe des Konstruktors
var vater = new Person('Jürgen', 'Schulz', 50);
alert ( vater.alter ); // => undefined

```

## Konstruktoren erweitern

Unter Umständen muss ein Prototyp (Konstruktor) um neue Attribute oder Methoden erweitert werden. Normalerweise sollte das dadurch gemacht werden, dass einfach das Attribut bzw. die Methode in den Konstruktor eingetragen wird.

Falls das nicht geht, kann über die Prototype-Eigenschaft das neue Attribut bzw. die neue Methode eingetragen werden.

Das neue Attribut ist dann in allen Objekten vorhanden, die mithilfe des Konstruktors erzeugt wurden. Auch dann, wenn das Objekt vor der Erweiterung des Konstruktors erzeugt wurden.

**Vorsicht:** Auch, wenn es verführerisch ist. Dieser Weg sollte nur für eigene Prototypen verwendet werden. In Javascript vorgegebene Prototypen (Date, Window, Array, etc.) sollten nicht nachträglich verändert werden – auch nicht, wenn es geht.

```

// Anlegen des Konstruktors
function Person( vorname, nachname, alter ){
    this.vorname = vorname;
    this.nachname = nachname;
    var alter = alter;    // kein 'this'
}

// Anlegen von Objekten mit Hilfe des Konstruktors
var vater = new Person('Jürgen', 'Schulz', 50);

Person.prototype.status = 'berufstätig';
alert ( vater.status );    // Gibt 'berufstätig' aus
vater.status = 'Rentner';
alert ( vater.status );    // Gibt 'Rentner' aus

```

## Getter und Setter

### Getter

Manchmal wünscht man sich, auf bestimmte Informationen aus einem Objekt zuzugreifen, die nicht in einem Attribut gespeichert sind. Dies könnte die Summe eines Arrays sein, der letzte Index eines Array oder ein bestimmter Wert, der tief in einem verschachtelten Objekt vergraben ist.

Wenn dafür keine Methode aufgerufen werden soll (vielleicht um die Klammern zu sparen oder aus Performancegründen), bieten sich Getter dafür an.

Ein Getter scheinen Methoden recht ähnlich, unterscheiden sich aber in Details von ihnen.

Um in einem Objekt einen Getter anzulegen, wird das Schlüsselwort *get* verwendet.



Der Getter wird angesprochen wie ein normales Argument (ohne Klammern) und kann keine Parameter übergeben.

```
var o = {
  meinArray: [6,8,4,2,3,4,7],

  get durchschnitt(){
    let ds = this.meinArray.reduce ( (sum,e) => sum + e );
    ds /= this.meinArray.length;
    return ds;
  }

};

console.log ( o.durchschnitt );
```

## Setter

Setter sind das Gegenstück zu Getter.

Sie dienen dazu, beliebige Veränderung im Objekt vorzunehmen und den Aufruf wie das Setzen eines normalen Attributes aussehen zu lassen.

Zum Anlegen eines Setters dient das Schlüsselwort `set`. Der Setter ist ähnlich einer Methode aufgebaut, kann aber nur genau einen Parameter annehmen. Der Aufruf sieht aus, als würde einem normalen Attribut ein Wert vergeben werden.

Was der Setter dann mit diesem Wert anfängt, ist der Fantasie des Entwicklers überlassen.

```
var o = {
  meinArray: [6,8,4,2,3,4,7],

  set neu(p) {
    this.meinArray.push (p);
    console.log(this.meinArray);
  }

};

o.neu = 2;
o.neu = 7;
o.neu = 12;
```

## Die Spread-Syntax



[\(Strg-Klick für mehr\)](#)

Angenommen, wir möchten vorhandene Variablen und Attribute restrukturieren. Dann bietet Javascript seit ES6 einige interessante Möglichkeiten. Diese sind unter dem Begriff ***Spread*** oder ***Spread Properties*** zusammengefasst.

Im folgenden einige Beispiele, was mit diesen Techniken möglich ist.

**Achtung:** Diese Technik ist nur auf einigermaßen aktuellen Browsern möglich. Hier bitte testen

Objekt auf Basis vorhandener Variablen füllen.

Bitte beachte, dass in dem Fall, dass ein Attribut bereits existiert, der Wert des zuletzt zugewiesenen Attributes gewinnt.

```
let position = 'Poststelle';

let person = {
  vn: 'Max',
  nn: 'Mustermann',
  wohnort: 'Bielefeld',
  geboren: 1976
};

let mitarbeiter = {
  ...person,
  position,
  geboren: 1980
};

console.log ( mitarbeiter );
```

Es geht auch anders herum. Falls wir ein Objekt in einzelne Variablen umwandeln wollen.

Zu beachten ist hier, dass die Variablen dieselben Namen tragen müssen wie die Attribute im Objekt. Die Reihenfolge ist irrelevant.

```
let werte = {a:1,b:2,c:3,d:4,e:5};

let { a, b, x, ...r } = werte;

console.log ( a );
console.log ( b );
console.log ( r );      // Objekt
console.log ( x );      // -> x existiert nicht
```

Oder das **Kopieren eines Objektes**, ohne dass nur eine Verknüpfung entsteht.

```
let wert1 = { a:1, b:2, c:3 };
let wert2 = { ...wert1 };
wert1.a = 23;
console.log ( wert2 );      // -> {a: 1, b: 2, c: 3}
```

<https://derickbailey.com/2017/06/06/3-features-of-es7-and-beyond-that-you-should-be-using-now/>

# Das Date-Objekt

## Anwendungsbeispiele:

- Dauer bis zu einem definierten Ereignis ermitteln
- Stoppuhr
- Wecker, Erinnerung
- Kalender, Terminplaner
- Timestamp

Das Date-Objekt gibt Dir die Möglichkeit, mit Daten (Plural von Datum) zu hantieren.

Um mit dem Datum zu arbeiten, braucht es erstmal ein Objekt, in dem das Datum gespeichert ist.

```
// Legt ein Objekt mit dem aktuellen Datum an
var datum1 = new Date();

// Legt ein Date-Objekt an
// und füllt es mit der Anzahl Millisekunden seit dem 1.1.1970
var datum2 = new Date(34664148443135);

// Legt ein Date-Objekt an und füllt es mit dem 23.03.2016, 16:56:00
// Beachte: Monate beginnen bei 0, Tage nicht
var datum3 = new Date(2016,2,23,16,56,0);
```

Die wichtigsten Methoden für ein Date-Objekt sind die folgenden:

- |  |   |
|--|---|
| • <a href="#">getDate()</a> (Monatstag ermitteln)                                    | • <a href="#">setDate()</a> (Monatstag setzen)                                |
| • <a href="#">getDay()</a> (Wochentag ermitteln)                                     | • <a href="#">setFullYear()</a> (volles Jahr setzen)                          |
| • <a href="#">getFullYear()</a> (volles Jahr ermitteln)                              | • <a href="#">setHours()</a> (Stunden der Uhrzeit setzen)                     |
| • <a href="#">getHours()</a> (Stundenteil der Uhrzeit ermitteln)                     | • <a href="#">setUTCHours()</a> (Stunden der Uhrzeit am Null-Meridian setzen) |
| • <a href="#">getUTCHours()</a> (Stundenteil der Uhrzeit am Null-Meridian ermitteln) | • <a href="#">setMilliseconds()</a> (Millisekunden setzen)                    |
| • <a href="#">getMilliseconds()</a> (Millisekunden ermitteln)                        | • <a href="#">setMinutes()</a> (Minuten der Uhrzeit setzen)                   |
| • <a href="#">getMinutes()</a> (Minutenteil der Uhrzeit ermitteln)                   | • <a href="#">setMonth()</a> (Monat setzen)                                   |
| • <a href="#">getMonth()</a> (Monat ermitteln)                                       | • <a href="#">setSeconds()</a> (Sekunden der Uhrzeit setzen)                  |
| • <a href="#">getSeconds()</a> (Sekundenteil der Uhrzeit ermitteln)                  | • <a href="#">setTime()</a> (Datum und Uhrzeit setzen)                        |
| • <a href="#">getTime()</a> (Zeitpunkt in ms seit dem 1.1.1970, 00:00 ermitteln)     |   |

## Lokalisierung

Die Ausgabe des Date-objekts in der Form:

```
var datum = new Date();
```

```
alert( datum );
```

liefert uns zwar das Datum, aber in amerikanischer Schreibweise.

Um Ausgaben in der gewohnten, lokalisierten Schreibweise zu erhalten, benutzen wir die folgenden Methoden.

```
var datum = new Date();  
alert( datum.toLocaleString() );  
alert( datum.toLocaleDateString() );  
alert( datum.toLocaleTimeString() );
```

Die Methode kann als Parameter die gewünschte Lokale wie 'fr' oder 'de' bekommen. Standard ist die im System/auf der Webseite eingestellte Lokale.

### Ohne Variablen arbeiten

Das Date-Objekt (und andere durch Konstruktoren erzeugte Objekte) muss nicht unbedingt in einer Variablen liegen, um Informationen herauszurücken. Wenn wir nur mal eben schnell z.B. das aktuelle Jahr ermitteln wollen, können wir direkt den Konstruktor starten und die Information aus dem gelieferten Objekt herausziehen.

```
alert ( new Date().getFullYear() );
```

## Das Window-Objekt

Anwendungsbeispiele:

- Popup-Fenster, diese werden allerdings häufig unterdrückt und sind nicht mehr zeitgemäß. Als Alternative werden gern 'Modal-Fenster' verwendet.

So können wir mit Hilfe des Window-Objektes die folgenden Eventlistener feuern:

- Sobald das Dokument vollständig geladen ist  
window.onload = funktion;
- Scrollen des Fensters  
window.onscroll = funktion;
- Verändern der Fenstergröße  
window.onresize = funktion;

### Eigenschaften

Das Window-Objekt bietet außerdem eine Reihe Eigenschaften an, die wir auslesen und verwenden können.

Vor allem ist dies die Fenstergröße. Hierbei wird unterschieden zwischen der Größe des Dokumentenbereichs und der Gesamtgröße des Fensters.

```
//Ausgeben der Größe des sichtbaren Bereichs  
alert( window.innerWidth + ' * ' + window.innerHeight );
```

```
//Ausgeben der Größe des Fensters  
alert( window.outerWidth + ' * ' + window.outerHeight);
```

## Methoden

Viele Methoden des Window-Objektes ermöglichen uns tiefgreifende Möglichkeiten, Fenster zu erzeugen und zu manipulieren.

### Open

Öffnet ein neues Fenster. Kann drei Parameter annehmen: URL, Name des neuen Fensters, Zusätzliche Angaben, replace.

```
window.open('URLDerDatei.html', 'nameDesFensters', 'width=400,height=800');
```

Die URL kann auch leer sein, dann wird ein leeres Fenster geöffnet.

Wenn ein Fenster mit dem gewünschten Namen schon existiert, dann wird die gewünschte Seite in diesem Fenster

Der dritte Parameter erlaubt uns, die Größe und Positionierung des neuen Fensters zu bestimmen. Die meisten anderen Möglichkeiten sind browser- und umgebungsabhängig.

Der vierte Parameter bestimmt, ob die neue Seite den Platz der ursprünglichen Seite in der Browser-History übernehmen soll.

Um nachträglich auf das Fenster oder dessen Inhalt zuzugreifen, empfiehlt es sich, das Fenster in einer Variablen zu referenzieren:

```
var meinFenster = window.open('URLDerDatei.html', 'nameDesFensters');
```

**Achtung:** So gut wie alle aktuellen Browser unterdrücken das automatische Öffnen eines neuen Fensters. Um den Popup-Blockers zum Testen abzuschalten, gibt es reichlich Anleitungen im Netz.

### Inhalt verändern

In dem geöffneten Fenster können wir auch den Inhalt nachträglich verändern. Dabei ist zu unterscheiden, ob das neu geöffnete Fenster eine Datei lädt oder leer ist.

Bei einem leeren, neuen Fenster kann einfach direkt nach dem Öffnen das document-Objekt des neuen Fensters verwendet werden:

```
neuesFenster = window.open('', 'neu');  
var element = neuesFenster.document.createElement('div');  
element.innerHTML = 'Mein neues Fenster.';  
neuesFenster.document.body.appendChild(element);
```

Wenn ein HTML-Dokument in das neue Fenster geladen wird, dann muss zunächst der onload-Event abgewartet werden, bevor das DOM manipuliert werden kann.

**Achtung:** Das funktioniert nur, wenn sich die neu geöffnete Seite in derselben Domain befindet, wie die ursprüngliche:

```
neuesFenster = window.open('laden.html', 'neu');
neuesFenster.onload = function() {
    var element = this.document.createElement('div');
    element.innerHTML = 'Mein neues Fenster.';
    this.document.body.appendChild(element);
}
```

## Close

Schließt das Fenster.

```
// Schließt das Fenster in der Variable
meinFenster.close();
```

## History

Das history-Objekt ermöglicht uns, in der Liste der bereits besuchten Seiten zu springen.

```
// Geht einen schritt zurück
window.history.back()

// Geht einen Schritt vor
window.history.forward();

geht um die definierte Anzahl Schritte vor (zurück mit negative Werten)
window.history.go();
```

- Sichere Umwandlung von geladenen Dateien in eine Collection
- Verwenden von DOM-Nodes als Schlüssel
- U.m.

# Map (Collection-Datentyp) [\(Strg-Klick für mehr\)](#)

*Objekte* sind praktisch, wenn es darum geht, Datensammlungen zu speichern. Aber manchmal (selten) kommt es vor, dass der Schlüssel nicht als String dargestellt werden kann. Vielleicht, weil der Schlüssel dann einem geschützten Wort wie `'__prototype__'` oder `'constructor'` entspräche.

In dem Fall kann uns der Datentyp *Map* helfen.

Eine Map wirkt wie ein Zwischending aus Object und Array.

Es ermöglicht uns, ähnlich einem Objekt, Schlüssel-Wert-Paare zu speichern. Allerdings werden die einzelnen Attribute als 2-Element-Arrays angelegt.

Maps bieten neben einer anderen Syntax v.a. folgende Vorteile:

- Als Schlüssel eines *Map*-Attributes können neben Strings auch *Zahlen*, *Objekte* und sogar *Funktionen* erhalten. Wogegen Objekte nur Strings (und auch nicht jeden beliebigen String) als Key verwenden können.
- Die Größe einer *Map* lässt sich über die *size*-Eigenschaft auslesen.
- Eine *Map* ist einfach iterierbar, ähnlich einem Array.

Aber natürlich auch ein paar Nachteile:

- Maps sind viel langsamer als Objekte oder Arrays (Stand 05/18)
- Maps können nicht zu JSON umgewandelt werden

Um eine *Map* **anzulegen**, wurde der *Map*-Konstruktor erfunden:

```
let karte = new Map();

karte.set ( 'a', 1 );
karte.set ( 'b', 2 );
karte.set ( 'c', 3 );

// set kann auch verkettet werden
let karte2 = new Map()
.set ( 'x', 42 )
.set ( 'y', 23 )
.set ( 'z', 5 );

console.log ( karte );
```

Das **Auslesen** der Attribute ist ähnlich einfach:

```
console.log ( karte.get('a') );
console.log ( karte.get('b') );
```

Aufgrund des Aufbaus einer *Map* kann auch ein **2D-Array als Parameter** an den Konstruktor übergeben werden. Dieses *Array* wird dann in eine *Map* konvertiert.

```
let a = [ ['x',42], ['y',23], ['z',5] ];
let karte = new Map( a );

console.log ( karte );
```

## Map iterieren

Eine *Map* kann auf verschiedene Weisen iteriert werden.

Eine **for...of**-Schleife stellt uns alle Key-Value-Paare zur Verfügung:

```
let a = [['x',42],['y',23],['z',5]];
let karte = new Map( a );
```

```

for ( i of karte ){
  console.log ( i );
}
/* ->
Array [ "x", 42 ]
Array [ "y", 23 ]
Array [ "z", 5 ]
*/

```

Die **forEach**-Methode dagegen liefert nur die Werte:

```

let a = [['x',42],['y',23],['z',5]];
let karte = new Map( a );

karte.forEach ( el => console.log ( el ) );

/* ->
42
23
5
*/

```

Die **Entries()-Methode** liefert uns ein *Iterator*-Objekt([Strg-Klick für mehr](#)) mit allen Schlüssel-Wert-Paaren, welches wir einfach durchlaufen können.

```

let a = [['x',42],['y',23],['z',5]];
let karte = new Map( a );

let i = karte.entries();

while ( !(o = i.next()).done ) console.log ( o.value );

/* ->
Array [ "x", 42 ]
Array [ "y", 23 ]
Array [ "z", 5 ]
*/

```

Die **Keys()-Methode** liefert uns ein *Iterator*-Objekt([Strg-Klick für mehr](#)) mit allen Schlüsseln, welches wir einfach durchlaufen können.

```

let a = [['x',42],['y',23],['z',5]];
let karte = new Map( a );

let i = karte.keys();

while ( !(o = i.next()).done ) console.log ( o.value );

/* ->

```



```
x
y
z
*/
```

Die **Values()-Methode** liefert uns ein *Iterator*-Objekt([Strg-Klick für mehr](#)) mit allen Werten, welches wir einfach durchlaufen können.

```
let a = [['x',42],['y',23],['z',5]];
let karte = new Map( a );

let i = karte.values();

while ( !(o = i.next()).done ) console.log ( o.value );

/* ->
42
23
5
*/
```

## Anzahl von Elementen

Die Anzahl von Speicherstellen in einer Map lässt sich mit der *size*-Eigenschaft auslesen.

```
let a = [['x',42],['y',23],['z',5]];
let karte = new Map( a );

console.log ( karte.size ); // -> 3
```

## Elemente entfernen

Um **ein Element** aus der Map zu entfernen, gibt es die *delete* Methode.

```
let a = [['x',42],['y',23],['z',5]];
let karte = new Map( a );

karte.delete ( 'y' );

console.log ( karte ); // -> Map { x: 42, z: 5 }
```

Um **alle Elemente** mit einem Befehl zu löschen, gibt es die *clear*-Methode:

```
let a = [['x',42],['y',23],['z',5]];
let karte = new Map( a );

karte.clear ();

console.log ( karte ); // -> Map { }
```

## Existenz eines Elements prüfen

Um zu prüfen, ob eine Map ein Element mit einem bestimmten **Schlüssel enthält**, gibt es die *has()*-Methode.

```
let a = [['x',42],['y',23],['z',5]];
let karte = new Map( a );

console.log ( karte.has( 'x' ) ); // -> true
console.log ( karte.has( 'a' ) ); // -> false
```

## Anwendungsbeispiel/e

Speichern von Werten für eine Auswahl an DOM-Nodes, ohne auf *data*-Attribute zurückgreifen zu müssen.

```
let mappe = new Map();

let pTags = document.querySelectorAll('p');

for ( var i = 0; i < pTags.length; i++ ){

    mappe.set ( pTags[i], {
        content: pTags[i].innerHTML,
        value: 0
    });

    pTags[i].onclick = evt => {
        evt.currentTarget.innerHTML = mappe.get ( evt.currentTarget ).value++;
        console.log ( 'Inhalt: ' + mappe.get ( evt.currentTarget ).content );
    }
};

console.log ( mappe );
```

# Set (Collection-Datentyp)

- Sammeln von Informationen, die immer gemeinsam bearbeitet werden.  
Bspw. Agenten in einer Schwarmsimulation oder Zellen in einer Wettersimulation.

In einer Set-Variablen kann eine Sammlung an Daten gespeichert werden, wie in allen anderen Collections auch. Der Unterschied ist, dass die Daten hier keinen Schlüssel haben. Es kann also kein einzelnes Element explizit angesprochen werden.

Ein Eintrag in einem Set muss eindeutig sein, d.h., es gibt einen Wert immer nur einmal.

Sets sind in der Ausführung noch etwas langsamer als Maps.

Zum **Anlegen** eines Sets haben wir wie den Set-Konstruktor:

```
let s = new Set();
```

Wir können diesem Set von vornherein ein Array als Parameter mitgeben. Die Werte werden dann in das Set übertragen.

In der Ausgabe erscheinen die Werte in der Reihenfolge, in der sie an den Set angehängt wurden.

```
let meinArray = [ 'Apfel', 'Banane', 'Birne' ];  
let s = new Set( meinArray );  
console.log ( s );
```

Um Werte in das Set **hinzuzufügen**, gibt es die add-Methode.

Dabei gilt, dass alle Werte eindeutig sind. Ein Duplikat wird nicht angehängt.

Groß- und Kleinschreibung wird unterschieden.

```
let meinArray = [ 'Apfel', 'Banane', 'Birne' ];  
let s = new Set( meinArray );  
  
s.add ( 'Lauch' );  
s.add ( 'Kohl' );  
s.add ( 'Apfel' );  
s.add ( 'banane' );  
  
console.log ( s );
```

## Set iterieren

Ein Set kann auf verschiedene Weisen iteriert werden.

Eine **for...of**-Schleife stellt uns alle Werte zur Verfügung:

```
let satz = new Set( [42,23,5] );  
  
for ( i of satz ){
```

```
    console.log ( i );
  }
  /* ->
  42
  23
  5
  */
```

Die **forEach**-Methode liefert ebenfalls die Werte:

```
let satz = new Set( [42,23,5] );
satz.forEach ( el => console.log ( el ) );

/* ->
42
23
5
*/
```

Die **Values()-Methode** liefert uns ein *Iterator*-Objekt ( [Strg-Klick für mehr](#) ) mit allen Werten, welches wir einfach durchlaufen können.

```
let satz = new Set( [42,23,5] );

let i = satz.values();

while ( !(o = i.next()).done ) console.log ( o.value );

/* ->
42
23
5
*/
```

## Anzahl von Elementen

Die Anzahl von Speicherstellen im Set lässt sich mit der *size*-Eigenschaft auslesen.

```
let satz = new Set( [42,23,5] );
console.log ( satz.size );    // -> 3
```

## Elemente entfernen

Um **ein Element** aus der Map zu entfernen, gibt es die *delete* Methode. Diese nimmt als Parameter einfach den Wert an, der entfernt werden soll.

```
let satz = new Set( [42,23,5] );

console.log ( satz );
satz.delete ( 23 )
```

```
console.log ( satz );
```

Um **alle Elemente** mit einem Befehl zu löschen, gibt es die `clear`-Methode:

```
let satz = new Set( [42,23,5] );  
  
satz.clear()  
  
console.log ( satz );    // -> Set []
```

### Existenz eines Elements prüfen

Um zu prüfen, ob eine Map ein Element mit einem bestimmten **Schlüssel enthält**, gibt es die `has()`-Methode.

```
let satz = new Set( [42,23,5] );  
console.log ( satz.has( 42 ) );    // -> true
```

# Iteratoren / Generatoren



(Strg-Klick für mehr)

## Iteratoren

In Javascript gab es bis vor kurzem zwei verschiedene Collection-Datentypen: Arrays und Objekte. Diese wurden/werden auf verschiedene Weise iteriert, wenn alle Elemente angefasst werden sollen. Das ist inkonsequent und spätestens mit der Einführung der neuen Collection-Datentypen (Map und Set) wurde es Zeit, die Iteration zu standardisieren.

Daher wurden Iteratoren erfunden.

Ein Iterator ist grob gesagt ein Objekt, das sich die aktuelle Position in der Collection merken, den Inhalt ausgeben und automatisch weiterzählen kann.

In Javascript besitzen die Datentypen *String*, *Array*, *Map* und *Set* einen eingebauten Iterator. Dieser lässt sich immer auf dieselbe Weise benutzen:

```
let a = new Map ( [[21,32],[45,65],[87,98],[41,52]] );
let i = a.values();

console.log ( i.next().value );
console.log ( i.next().value );
console.log ( i.next().value );
console.log ( i.next().value );
```

Die *values*-Methode liefert als Rückgabewert den Iterator.

Dieser kann genau eine Sache: bei Aufruf der *next()*-Methode wird der aktuelle Inhalt eines Elementes zurückgegeben und der Iterator zeigt auf das nächste Element.

Jedes der Elemente ist ein Objekt, das zwei Attribute enthält:

**Value:** Wert in diesem Element

**Done:** Info, ob das letzte Element des Iterators erreicht ist.

Durch einen geschickten Aufbau lässt sich nun das Script oben deutlich verbessern:

```
let a = new Map ( [[21,32],[45,65],[87,98],[41,52]] );
let i = a.values();

while ( !(wert = i.next()).done ) {
  console.log ( wert.value );
}
```

Mehr lesen ( Strg-Klick)

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration\\_protocols](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols)

<https://developer.mozilla.org/en-US/docs/Web/API/NodeIterator>

# Generatoren (eigene Iteratoren)

Vorgefertigte Iteratoren sind in den meisten Fällen vollkommen ausreichend und sehr praktisch. Aber es gibt Situationen, in denen das einfach nicht ausreicht. Z.B., wenn ein Iterator alle oder nur bestimmte Attribute eines Objektes iterieren soll. Oder wenn wir eine bestimmte Anzahl zufälliger Zahlen brauchen, oder oder oder.

In den Fällen können wir mit Hilfe von *Generator*-Funktionen eigene Iteratoren definieren.

Eine *Generator*-Funktion wird aufgerufen und liefert den *Iterator* als Rückgabewert.

Die Funktion ist durch den *Stern* (\*) als *Generator* gekennzeichnet.

In der Funktion steht mindestens einmal der Befehl *yield*(...). Wenn die *next*()-Methode des *Iterators* aufgerufen wird, dann liefert diese den Wert des nächsten *yield*() zurück, das gefunden wird. An der Stelle bleibt die Funktion stehen und wartet auf den nächsten Aufruf von *next*(), um den Code weiter auszuführen und den Wert des nächsten *yield*() zu liefern.

```
const g = function*() {
  yield 42;
  yield 23;
  yield Math.sqrt(2);
}

let meinIterator = g();

console.log ( meinIterator.next().value );    // -> 42
console.log ( meinIterator.next().value );    // -> 23
console.log ( meinIterator.next().value );    // -> 1.4142135623730951
```

Ein solcher Generator kann durchaus komplexer aufgebaut sein und denselben *yield* mehrfach aufrufen:

```
const g = function* ( anzahl ) {
  for ( let i = 0; i < anzahl; i++ ) {
    yield Math.floor ( Math.random() * 100 );
  }
}

let meinIterator = g(10);

while ( !( i = meinIterator.next() ).done )
  console.log ( i.value );
```

## Iteratoren für Objekte

Mit dem Wissen ausgestattet, ist es ein leichtes, einen Generator zu entwickeln, der z.B. alle Attribute eines DOM-Objektes außer den Methoden und den Eventlistenern iteriert:

```
const meinGenerator = function* ( dasObjekt ) {
  for ( i in dasObjekt ) {
    if ( typeof dasObjekt[i] !== 'function' && !i.startsWith('on') )
      yield {key: i, value: dasObjekt[i], type: typeof dasObjekt[i]};
  }
}
```

```
}  
  
let derIterator = meinGenerator( document.body );  
  
while ( !( o = derIterator.next() ).done )  
    console.log ( o.value );
```



# Prototyping

In Javascript basiert die gesamte Datenstruktur auf sog. Prototypen.

Prototypen kann man sich wie Blaupausen vorstellen, auf deren Basis Objekte angelegt werden. Das klingt zunächst einmal nach Konstruktoren (s.o.), geht aber weiter.

Der grundlegende Prototyp ist `Object`.

Alle anderen Datentypen basieren darauf, d.h. sie erben die Eigenschaften des Prototyps und erschaffen noch eigene dazu.

So ist ein `Array` ein Kind des `Object`-Prototyps, `String` ebenso.

Schön zu wissen. Aber was nutzt uns das.

## Erweitern von Konstruktoren

Das Wichtigste, was wir mit diesem Wissen anstellen können ist, bestehende Konstruktoren zu erweitern.

Im Beispiel unten wird der `Date`-Konstruktor um eine neue Methode erweitert.

Bitte beachte, dass nun alle Objekte, die mithilfe des `Date`-Konstruktors angelegt wurden, die neue Methode tragen. Selbst jene Objekte, die vor dem Prototyping angelegt wurden.

```
var d = new Date()
var e = new Date(2000,11,24,15);

Date.prototype.gruss = function(){
  console.log ( 'Hallo Welt' );
}

d.gruss();
e.gruss();
```

**Achtung:** Um das Überschreiben existierender Methoden zu vermeiden, wird geraten, nur eigene Konstruktoren auf diese Weise zu verändern.

# DOM ( Webseite manipulieren )

Anwendungsbeispiele:

- Verändern des Webseiteninhaltes und auslesen vorhandener Informationen
- Optische Darstellung eines Warenkorbes.
- Darstellung von Berechnungsergebnissen.
- Darstellung von Inhalten, die nachträglich per Ajax geladen wurden.
- Webapps
- U.v.m.

Um auf die Inhalte einer Webseite in standardisierter Weise zuzugreifen, wurde das Document Object Model, kurz DOM, erdacht.

Es definiert, wie die einzelnen Elemente einer Webseite und deren Eigenschaften ausgelesen und manipuliert werden können.

Im Rahmen dieser Doku werde ich nicht das DOM in die Tiefe besprechen, sondern nur einige Methoden und Eigenschaften zeigen. Eine Liste aller Attribute eines DOM-Elementes findest Du hier: [http://www.w3schools.com/jsref/dom\\_obj\\_all.asp](http://www.w3schools.com/jsref/dom_obj_all.asp)

## Elemente ansprechen

Der erste Schritt, Eigenschaften von Elementen einer Webseite zu lesen oder zu schreiben ist der, diese Elemente anzusprechen. Der empfohlene Weg ist der, die Ergebnisse der Ansprachen in einer Variablen zu speichern, um später leichter darauf zugreifen zu können.

Alle Eigenschaften, die in einer dieser Variablen geändert werden, sind auf der Webseite sofort sichtbar. Ebenso sind alle Änderungen, die auf der Webseite durchgeführt werden, sofort in der Variable verfügbar.

Um Elemente auf einer Webseite anzusprechen, stehen uns im Wesentlichen zwei verschiedene Methoden zur Verfügung: *querySelector()* und *querySelectorAll()*. Diese nehmen als Parameter einen CSS-Selektor an und liefern die entsprechenden Elemente als Rückgabewerte.

```
// Speichert das erste Element, das dem Selektor entspricht.
var element = document.querySelector ( '#meineID' );

// Speichert alle Elemente, die dem Selektor entsprechen,
// in Form einer statischen NodeList (s.u.)
var elemente = document.querySelectorAll ( '#meins p:nth-of-type(even)' );

// Außerdem lässt sich der body gesondert ansprechen
var meinBody = document.body;
```

Von v.a. **historischem Interesse** sind weitere Möglichkeiten, Elemente anzusprechen:

```
// Ansprache per ID, liefert genau ein Element
```

```

var element1 = document.getElementById('IDname');

// Ansprache per Klasse. Achtung: Elements im Plural
// Liefert eine Liste von Elementen in Form einer dynamischen Nodelist
var elemente2 = document.getElementsByClassName('Klassenname');

// Ansprache per Tagbezeichnung. Achtung: Elements im Plural
// Liefert eine Liste von Elementen (sog. nodeList)
var elemente3 = document.getElementsByTagName('Tagname');

```

## Nodelisten

Ansprachen, die mehrere Elemente liefern können, geben eine sog. Nodelist zurück. Dabei handelt es sich um eine Art "Array mit eingeschränktem Funktionsumfang".

Man unterscheidet dabei zwischen statischen und dynamischen Nodelisten.

Eine dynamische Nodelist wird beim Entfernen eines vorhandenen oder dem Hinzufügen eines neuen Elementes automatisch aktualisiert.

In einer statischen Nodelist wechselt der Speicherplatz eines entfernten Elementes zu undefined. Ein neu erzeugtes Element wird in eine statische Nodelist nicht eingetragen.

Nodelisten lassen sich auch in ein Array umwandeln, um dann sehr komfortabel mit Array-Methoden alle Knoten der Nodelist zu bearbeiten.

```

let elemente = Array.from ( document.querySelectorAll ( '.hallo' ) );
elemente.forEach ( e => e.innerHTML = 'Die forEach-Methode ist cool.' );

```

## Arbeitserleichterung

Das ständige Eingeben dieser zugegebenermaßen recht langen Befehle kann anstrengend sein. Daran gewöhnt man sich zwar im Laufe der Zeit. Aber wenn man sich das Leben leichter machen kann, warum nicht.

Die erste Erleichterung ist es, ein DOM-Element mit einer ID einfach mit dessen ID anzusprechen. Mit anderen Worten: Auf das getElementById zu verzichten.

Diese Technik ist allerdings undokumentiert und es gibt keine Garantie, dass dies bei allen Browsern gleichermaßen gut funktioniert – obwohl es das bei allen aktuellen Browsern tut.

```

<script type="text/Javascript">
  meinElement.innerHTML = 'Hallo Welt';
</script>

<div id="meinElement"></div>

```

Wenn man die offiziellen Wege nicht verlassen will, dann kann man sich auch eine Funktion bauen, die das Selektieren übernimmt und damit langfristig Arbeit sparen. Diese Funktion könnte z.B. auch in eine eigene Bibliothek integriert werden.

```
function sID( bezeichnung ){
// IDs suchen
var element = document.getElementById ( bezeichnung );
return element;
}

function sC( bezeichnung ){
// Klassen suchen
var elemente = document.getElementsByClassName ( bezeichnung );
return elemente;
}

window.onload = function(){
var a = sID('meinElement');
a.innerHTML = 'Irgendwas anderes';
}
```

## Kind-Elemente ansprechen

Manchmal ist es notwendig, die Kind-Elemente anzusprechen.

Das Schlüsselwort *children* liefert eine Node-Liste, die alle Elemente enthält, die direkte Kinder des Ausgangselementes sind.

```
var element1 = document.getElementById('IDname');
for ( var i = 0; i < element.children.length; i++){
    element.children[i].innerHTML = 'Hallo Welt!';
}
```

## Elemente verändern

### Inhalt lesen/ändern

Um ein HTML-Element aus dem DOM-Baum auszulesen und zu manipulieren, gibt es reichlich Möglichkeiten. Eine ist, auf den Inhalt des Elementes zuzugreifen.

Dies wird mit der Eigenschaft `innerHTML` erledigt:

```
// Elementinhalt verändern
var element = document.getElementById('inhalt');
element.innerHTML = 'Dies ist der neue Inhalt';
element.innerHTML += 'Dies ist angehängter Inhalt';
```

Alternativ kann auch `innerText` oder `textContent` verwendet werden.

### CSS-Klassen vergeben/entfernen (`classList`)

Das Zuweisen von CSS-Klassen geschieht über sog. Classlists. Jedes Element im DOM-Baum besitzt eine solche Classlist, die aussagt, welche CSS-Klassen das Element besitzt.

```
var element = document.getElementById('inhalt');

// CSS-Klasse(n) hinzufügen
element.classList.add('rot', 'hervorheben');

// CSS-Klasse entfernen
```

```
if(element.classList.contains('rot')){
element[2].classList.remove('rot');
}
```

### Attribute verändern

Es können Attribute der HTML-Tags verändert werden. Besonders hilfreich, wenn der Link eines a-Tags oder in einem img-Tag ein anderes Bild angezeigt werden soll.

```
document.getElementById("myImage").src = 'landscape.jpg';
document.getElementById("myLink").href = 'http://www.google.com/';
```

### Style-Eigenschaften ändern

Manchmal ist es nicht ausreichend, eine CSS-Klasse zuzuweisen, um einen bestimmten Effekt zu erzeugen. Dann kann auch direkt auf die CSS-Eigenschaften Einfluss genommen werden.

**Vorsicht:** Die Eigenschaften unterscheiden sich teils in ihrer Schreibweise von der CSS-Version. Insbesondere Bindestriche werden durch innerCaps ersetzt. Übersicht über die Eigenschaften: [http://www.w3schools.com/jsref/dom\\_obj\\_style.asp](http://www.w3schools.com/jsref/dom_obj_style.asp)

```
var element = document.getElementById('inhalt');

// style-Eigenschaft ändern
element.style.color = '#f00'; //Elemente anlegen
```

### Style-Eigenschaften auslesen

Das Style-Attribut (siehe oben) kann nur Style-Eigenschaften ausgeben, wenn diese inline definiert wurden.

Um auf alle Style-Eigenschaften zugreifen zu können, wurde die getComputedStyle-Methode erdacht. Diese ist ein Kind des Window-Objekts und bekommt als Parameter jenes DOM-Element übergeben, dessen Styling-Eigenschaften wir wissen wollen.

Der Rückgabewert der Methode ist ein Objekt, welches alle Style-Eigenschaften enthält.

**Achtung:** Abgekürzte Eigenschaften (z.B. margin, background, etc.) werden so nicht ausgelesen. Es muss die ausführliche Schreibweise (z.B. margin-top, background-color, etc.) verwendet werden.

```
var element = document.querySelector ( '#meineID' );
var stil = window.getComputedStyle ( element );
for ( var i in stil ){
    console.log ( i + ': ' + stil.i );
}
```

### Alle Style-Eigenschaften

Um das gesamte Style-Attribut auszugeben oder zu verändern, kann das *cssText*-Attribut verwendet werden. Dieses enthält alle Styles, die im *style*-Attribut stehen. CSS-Angaben aus dem *style*-Tag bzw. der CSS-Datei bleiben unberührt.

```
var element = document.querySelector ( '#meineID' );

console.log ( element.style.cssText ); // Gibt das ganze style-Attribut aus

element.style.cssText = 'color: #f00'; // überschreibt das gesamte
// style-Attribut durch den neuen Inhalt
```

## Anlegen, kopieren, verschieben, löschen

### Neues Element anlegen

Das Anlegen eines neuen Elementes im DOM geschieht in vier Schritten:

1. Erzeugen eines neuen Elementes (wird noch nicht dargestellt)
2. Erzeugen des Quellcodes, der in dem Element angezeigt werden soll.
3. Anhängen des Quellcodes in das Element
4. Anhängen des Elementes in den Seitenbaum. Erst dann wird etwas angezeigt.

```
var neuesElement = document.createElement('p');
var neuerText = document.createTextNode('hallo Welt');
neuesElement.appendChild(neuerText);
document.getElementById('anderesElement').appendChild(neuesElement);
```

Das Element, das im obigen Beispiel in der Variablen *neuesElement* angelegt wurde, kann nur einmal auf der Seite platziert werden.

Wird es später an einer anderen Stelle nochmal positioniert, so verschiebt dies das Objekt an die neue Stelle. Um ein Element mehrfach anzulegen, muss es kopiert werden.

### Elemente kopieren

Um ein Element zu kopieren, wurde der Befehl *cloneNode* erdacht.

Er erzeugt einen Klon des aufrufenden Elements (DOM-Knotens) und speichert ihn ggf. in eine neue Variable.

Der Parameter gibt an, ob neben dem Element und seinen Attributen auch die Kind-Elemente (u.a. der Inhalt) mitkopiert werden.

```
var neuesElement = document.createElement('p');
var neuerText = document.createTextNode('hallo Welt');
neuesElement.appendChild(neuerText);
var neuesElement2 = neuesElement.cloneNode(true);

document.getElementById('anderesElement').appendChild(neuesElement);
document.getElementById('anderesElement').appendChild(neuesElement2);
```

### Element vor ein anderes einfügen

Bis auf die letzte Zeile funktioniert es genauso wie das Anlegen eines Elementes im vorigen Artikel:

```
var neuesElement = document.createElement('p');
var hiervor = document.getElementById('hiervor');
var neuerText = document.createTextNode('hallo Welt');
neuesElement.appendChild(neuerText);

hiervor.parentNode.insertBefore(neuesElement, hiervor);
```

## Elemente entfernen

Ein Element aus dem DOM zu entfernen ist relativ einfach.

```
var element = document.querySelector('irgendwas');
element.remove();
```

Leider funktioniert das im Internet Explorer nicht.

Im IE kann ein Kind-Elemente nur von seinem Eltern-Element entfernt werden.

```
var element = document.getElementById('irgendwas');
element.parentNode.removeChild(element);
```

## Eigenschaften von Formularelementen

Auch Formular-Element gehören zum DOM und deren Eigenschaften (Inhalt, ausgewähltes Element, etc.) lassen sich auslesen.

Das Thema ist unter Ein- / Ausgabe abgehandelt. [Strg-Klick hier](#).

# Eventlistener

Anwendungsbeispiele:

- Reagieren auf Benutzereingaben.
- Buttons, Eingabetextfelder, Radioknöpfe, Auswahlfelder
- Anpassen des Webseiteninhaltes auf eine Änderung der Fenstergröße (jenseits der CSS-mediaqueries)
- Anpassen des Inhalts / der Darstellung als Reaktion auf einen Scrollvorgang.
- Warten, dass die Webseite vollständig geladen ist

Die Begriffe Eventlistener und Eventhandler sind gleichbedeutend.

Ein Eventlistener wartet darauf, dass ein bestimmtes Ereignis eintritt. Sobald das Ereignis geschieht, wird eine festgelegte Funktion aufgerufen.

Um einen Eventhandler anzulegen gibt es verschiedene Möglichkeiten:

```
// Alternative 1
document.getElementById("myBtn").onclick = funktionsName;

// Mit anonymer Funktion
document.getElementById("myBtn").onclick = function() {
    // Code
};

// Alternative 2
document.getElementById("myBtn").addEventListener("click", funktionsName);

// Mit anonymer Funktion
document.getElementById("myBtn").addEventListener("click", function() {
    // Code
});
```

Im Beispiel ist zu beachten, dass in Alternative 2 die Art des Events ohne das *on* geschrieben werden.

Der Vorteil von Alternative 1 liegt in seiner relativen Einfachheit. Der Vorteil von Alternative 2 liegt darin, dass sich die Eventhandler nicht gegenseitig überschreiben. Ein Objekt kann dadurch für einen Event **mehrere Handler** besitzen:

```
document.getElementById("myBtn").addEventListener("click", myFunction);

document.getElementById("myBtn").addEventListener("click",
someOtherFunction);
```

Um eine Funktion mit Parametern aufzurufen, verwende einen gesonderten Funktionsaufruf innerhalb einer anonymen Funktion.



```
document.getElementById("myBtn").addEventListener("click", function() {
    meineFunktion(p1, p2);
});
```

Um einen Eventlistener wieder zu **entfernen**, kommt es darauf an, welche Alternative zum Anlegen des Eventlisteners gewählt wurde.

```
// Alternative 1: Überschreibe den Eventlistener einfach
// mit einer leeren Funktion
document.getElementById("myBtn").onclick = function(){};

// Alternative 2: Entferne die aufzurufende Funktion.
// Dies funktioniert nur, wenn eine benannte Funktion aufgerufen wird.
// Eine anonyme Funktion lässt sich nicht entfernen.
document.getElementById("myBtn").removeEventListener("click", myFunction);
```

### Liste der wichtigsten Events:

- onclick: Element wird angeklickt.
- onload: Laden der Seite ist vollständig abgeschlossen.
- onunload: Wenn die Seite aus dem Speicher genommen wird, z.B. durch Beendigung des Browsers oder wenn eine andere Seite geladen wird.
- onerror: Beim Laden externer Ressourcen ist ein Fehler aufgetreten.
- onchange: Der Inhalt des Elements wurde verändert. Gilt v.a. für Formularfelder. Onchange wird erst gefeuert, wenn das betreffende Element den Fokus verliert.
- oninput: Der Event wird sofort gefeuert, sobald sich im Eingabefeld etwas ändert, ohne dass das Element den Fokus verlieren muss.
- onscroll: Das Element wird gescrollt
- onresize: Die Größe des Fensters wird verändert.

Vollständige Liste: [https://www.w3schools.com/jsref/dom\\_obj\\_event.asp](https://www.w3schools.com/jsref/dom_obj_event.asp)

### Sichtbarkeit

Bitte beachte, dass Elementen nur dann Eventhandler zugewiesen werden können, wenn diese Elemente schon existieren.

Mit anderen Worten: Das Zuweisen von Eventhandlern an Elemente darf erst geschehen, wenn die Seite vollständig geladen wurde.

```
window.onload = function(){
    document.getElementById('auswahl').onclick = function(){
        // Irgendein Code
    }
}
```

this

Viele Eventlistener werden von einem DOM-Objekt ausgelöst.

Am Beispiel des onclick-Eventlisteners ist es das DOM-Element, das angeklickt wurde.

Am Beispiel des onmouseenter-Eventlisteners ist es das Element, über das die Maus bewegt wurde.

Am Beispiel des oninput-Eventlisteners ist es das Element, dessen Inhalt verändert wurde.

Innerhalb der ausgeführten Funktion steht genau dieses DOM-Element unter dem Begriff *this* zur Verfügung.

```
btn.onclick = function () {  
  console.log ( this );  
  this.innerHTML = 'Geklickt';  
}
```

### Arrow-Funktionen und this

Arrow-Funktionen haben bekanntermaßen kein eigenes this, im Falle von Eventlistenern also keinen Bezug zum aufrufenden Objekt. Aber mit einem kleinen Kniff lässt sich dieses Problem leicht umgehen.

Das Event-Objekt, das beim Aufruf durch den Eventlistener automatisch übergeben wird, enthält im Attribut `currentTarget` das aufrufende Element.

```
btn.onclick = event => {  
  console.log ( event.currentTarget );  
  event.currentTarget.innerHTML = 'Geklickt';  
}
```

Neben `currentTarget` gibt es noch das Attribut `target`. Es enthält nicht unbedingt das Element, das den Eventlistener trägt, sondern immer das (Kind-)Element, das tatsächlich angeklickt wurde.

### Mausposition

Die Mausposition lässt sich nur über einen mausbezogenen Eventlistener auslesen.

Der Parameter 'event' enthält ein Objekt, das unterem die Mausposition enthält.

Die Mausposition ist relativ zur gesamten Seite. Um die Mausposition relativ zum Objekt zu ermitteln, das den Event ausgelöst hat, muss dessen Position von der Mausposition abgezogen werden.

```
document.getElementById('element').onmousemove = function ( event ){  
  var posX = event.pageX - this.offsetLeft;  
  var posY = event.pageY - this.offsetTop;  
};
```

### Rechtsklick

Wenn ein Element angeklickt wird, ist es erstmal egal, mit welcher Taste dies geschieht. Die Information, welche Maustaste gedrückt wurde (und andere) sind in einem Parameter enthalten, der automatisch an die Funktion übertragen wird.

Im Beispiel heißt dieser Parameter *Event*. Mit Hilfe einer bedingten Anweisung kann dann die Eigenschaft *button* dazu dienen, die Maustasten voneinander zu unterscheiden.

Die Werte sind:

Linksklick: 0

Mittelklick: 1

Rechtsklick: 2

Die letzte Zeile im Beispiel dient dem Verhindern des Kontextmenüs.

```
klick.onmousedown = function(event){
  if ( event.button == 2 ){
    alert ( 'Rechtsklick' );
  }
  return false;
}
klick.addEventListener('contextmenu', event => event.preventDefault());
```

### onerror

Der Event onerror verhält sich etwas unterschiedlich zu den anderen Events.

Er wird gefeuert, wenn bei der Ausführung des Programmes ein Fehler auftritt.

Der **wichtigste** (aber nicht einzige) Anwendungsfall ist der, dass beim Laden einer Datei (Bild, etc) ein Fehler aufgetreten ist. Da der onload-EventListener erst gefeuert wird, wenn die Fehlerbehandlung vollständig abgeschlossen ist, kann der onerror-EventListener innerhalb des onload-Eventlisteners nicht auf Ladefehler reagieren. Daher muss das Script für den onerror-Event nach dem HTML-Code stehen.

```
<html><head></head>
<body>
  

  <script type="text/Javascript">
    document.getElementsByTagName('img')[0].onerror = function(){
      alert('Fehler');
    }
  </script>
</body>
```

Wenn der Code im Script vor dem HTML-Teil auftaucht, dann werden nur Fehler gemeldet, die beim nachträglichen Aufrufen eines Bildes/Dokumentes durch JS auftauchen.

Eine untergeordnete Bedeutung hat der Anwendungsfall, dass einige Fehlermeldungen außerhalb der Konsole angezeigt werden können. Der unten stehende Code gibt für jeden Fehler ein eigenes Alert-Fenster aus.

```
window.onerror = function(messageOrEvent, source, lineno, colno, error) {
  alert ( messageOrEvent + '\n ' +
source + '\n ' +
lineno + '\n ' +
colno + '\n ' +
error);
}
```

Dieser Weg ist allerdings Geschmackssache, weil einerseits nicht alle Fehler vom onerror-Eventlistener abgedeckt werden. Außerdem erscheinen diese Fehler (und mehr) ohnehin arrowin der Konsole.

## Scrolling

Um auf das Scrolling zu reagieren, hilft der folgende Code:

```
window.onscroll = function() {  
    ...  
}
```

Die Scrollposition kann so ausgelesen werden:

```
document.documentElement.scrollTop
```

Ältere Browser (insb. IE, Safari) verwenden jedoch noch eine ältere Schreibweise:

```
document.body.scrollTop
```

Bei Verwendung der falschen Methode liefert die Abfrage "0".

Um die Scrollposition in allen Browsern zuverlässig auszugeben, muss man beide Methoden lesen und dann die richtige aussuchen:

```
window.onscroll = function() {  
    Math.max(document.documentElement.scrollTop, document.body.scrollTop);  
}
```

# Lokales Speichern

## Cookies

### Anwendungsbeispiele:

- Benutzereingaben für mehrseitige Formulare (damit der Benutzer vor und zurück kann)
- Automatisches Einloggen
- Applikationseinstellungen speichern
- Speichern von zuletzt besuchten Seiten, um ein Tracking zu ermöglichen
- Etc.

Cookies sind eine Methode, um Daten lokal zu speichern.

Sie können von der geöffneten Webseite oder von einem PHP-Script gelesen werden (per http) und haben eine maximale Größe von 4095 Byte.

Gegenüber dem localStorage können Cookies auch vom Server mit PHP abgefragt werden.

Cookies haben ein Ablaufdatum, nach dem der Cookie automatisch gelöscht wird. Ist kein Ablaufdatum angegeben, so wird der Cookie beim Beenden des Browsers gelöscht. Man spricht dann von Session-Cookies.

```
// Schreiben eines Session-Cookies
document.cookie = 'meinCookie=Hallo Welt';

// Schreiben eines Cookies mit Ablaufdatum
// Beachte: Das Ablaufdatum im Beispiel ist UTC
document.cookie = 'meinCookie=Hallo Welt; expires=Wed Apr 06 2016 11:06:00 UTC';

// Schreiben eines Cookies, der in zwei Stunden entfernt wird
var bald = new Date();
bald.setHours( bald.getHours()+2 );
document.cookie = 'meinCookie=Hallo Welt; expires=' + bald;

// Mehrere Cookies werden einzeln geschrieben
document.cookie = 'meinCookie=Hallo Welt';
document.cookie = 'nochEiner=Hallo Mond';
document.cookie = 'dritterCookie=123';

// Existiert ein Cookie dieses Namens bereits, wird er überschrieben
document.cookie = 'meinCookie=Neuer Inhalt';

// Laden eines Cookies
var x = document.cookie;
```

Beim **Lesen** der Cookies werden immer alle Cookies dieser Domain als String in der Form zurückgegeben:

```
var x = document.cookie;
alert( x );

/*
ergibt: meinCookie=Hallo Welt;nochEiner=Hallo Mond;dritterCookie=123
*/
```

Um einzelne Cookies lesen zu können, empfiehlt sich eine Funktion wie die folgende. Alle Eintragungen des Cookies werden in ein eigenes Objekt geschrieben, aus dem alle Eintragungen leicht erreichbar sind.

```
function loadCookie(){
    var temp = document.cookie.split('; ');
    var cookies = {};
    for (var i = 0; i < temp.length; i++){
        var tempCookie = temp[i].split('=');
        cookies[ tempCookie[0] ] = tempCookie[1];
    }
    return cookies;
}
```

Zum **Löschen** wird der Cookie mit einem Ablaufdatum gespeichert, das in der Vergangenheit liegt.

```
var damals = new Date ( 0 );
document.cookie = 'meinCookie=; expires=' + damals;
```

**Achtung:** Der Debugger im Firefox (und mglw. in anderen Browsern) wird nicht immer zuverlässig aktualisiert. Wenn nach einem Löschen aller Cookies noch ein paar Cookies geblieben sind, dann reicht ggf. das Schließen und erneute Öffnen des Debuggers.

# Webstorage (a.k.a. DOM Storage, Supercookies)

## Anwendungsbeispiele:

- Benutzereingaben für mehrseitige Formulare (damit der Benutzer vor und zurück kann)
- Automatisches Einloggen
- Spielstände von Browsergames
- Applikationseinstellungen speichern
- Warenkorbfunktionalitäten
- Speichern von zuletzt besuchten Seiten, um ein Tracking zu ermöglichen
- Etc.

Webstorage ist ein Speicher, der ausschließlich vom Client beschreiben und gelesen werden kann. Dieser ist vom Dokument unabhängig und kann dazu benutzt werden, um Informationen von einer Webseite zu einer anderen zu übertragen.

Es können mehr als 5MB im Webstorage abgelegt werden.

Sollen mehr Daten in den Webstorage gespeichert werden, bekommen wir eine Fehlermeldung. Ein einfacher Test, wie viele Daten in Deinen Webstorage passt, ist hier:

<https://arty.name/localstorage.html>

Die Domains sind voneinander getrennt. D.h. eine Seite in einer Domain kann nicht auf den Webstorage einer anderen Domain zugreifen.

Es wird unterschieden zwischen **localStorage** und **sessionStorage**.

Der Unterschied ist, dass der sessionStorage beim Beenden des Browserfensters oder des Tabs gelöscht wird.

Es gibt keine Ablaufzeit.

Information werden in Key-Value-Paaren gespeichert und liegen grundsätzlich als String vor. Das bedeutet, dass Objekte oder Arrays vor dem Ablegen per JSON in einen String umgewandelt werden müssen.

Die Verwendung ist denkbar einfach:

```
// Speichern
localStorage.setItem('key', 'value');
sessionStorage.setItem('key', 'value');

// Lesen
localStorage.getItem('key');
sessionStorage.getItem('key');

// Entfernen
localStorage.removeItem('key');
sessionStorage.removeItem('key');

// Alles entfernen
localStorage.clear();
sessionStorage.clear();
```

### Alle Items ansprechen

Die obigen Techniken sind prima, eignen sich aber nicht, um den gesamten Webstorage (dieser Domain) auszulesen. Dafür brauchen wir eine Möglichkeit, um die Indizes anzusprechen.

Diese Möglichkeit haben wir in der key-Methode:

```
for (var i = 0; i < localStorage.length; i++){  
    alert(localStorage.getItem(localStorage.key(i)));  
}
```

Der Debugger im Firefox (und mglw. in anderen Browsern) wird nicht immer zuverlässig aktualisiert. Wenn nach einem Löschen aller Webstorage-Einträge noch ein paar Einträge geblieben sind, dann reicht ggf. das Schließen und erneute Öffnen des Debuggers.

### Sicherheitsfragen

Als ein Sicherheitsproblem könnte man die Möglichkeit des XSS ansehen. Das bedeutet Cross-site-scripting und ist der Versuch, Javascript-Code über Formularfelder zu injizieren. Der in die Formularfelder eingegebene Code würde dann auf der eigenen Domain ausgeführt und hätte dadurch Zugriff auf den Webstorage.



# Promises



[\(Strg-Klick für mehr\)](#)

Promises (Versprechen) sind der neue, hippe Weg (Stand 05/18), um mit asynchronen Operationen umzugehen.

Asynchrone Operationen können sein:

- Server-Abfragen per Ajax
- Timeouts
- Operationen im Webworker
- Warten auf Benutzer-Interaktion
- Etc.

Auf den ersten Blick sind Promises den Eventlistenern sehr ähnlich. Sie bieten aber verschiedene Vorteile:

- Der Entwickler kann selbst bestimmen, in welchem Fall die Operation erfolgreich ausgeführt wurde oder nicht.
- Ein (verketteter) Aufruf mehrerer Callback-Funktionen wird einfacher möglich.
- Ein Promise wird nur einmal erfüllt oder abgelehnt.

Der Promise selbst ist ein Objekt, das neben der auszuführenden Operation auch die Informationen enthält, was im Erfolgs- oder im Fehlerfall gemacht werden soll.

[Strg-Klick für mehr](#): Alexander Naumov

[Strg-Klick für mehr](#): Google Developers

Einen Promise legen wir mithilfe des *Promise*-Konstruktors an.

Der Konstruktor bekommt eine Funktion übergeben, die sofort ausgeführt wird.

Diese Funktion wiederum bekommt zwei Parameter: *resolve* und *reject* (Namen sind beliebig).

Im folgenden Beispiel wird eine Variable auf einen zufälligen booleschen Wert gesetzt.

```
let meinPromise = new Promise ( (resolve, reject) => {  
  
  let z = Math.random();  
  let b = z > .5;  
  console.log ( 'Funktionsausführung' );  
  
});
```

Im nächsten Schritte müssen wir noch bestimmen, wann wir die Aktion als erfolgreich (*resolve*) oder fehlgeschlagen (*reject*) definieren.

```
let meinPromise = new Promise ( (resolve, reject) => {  
  
  let z = Math.random();  
  let b = z > .5;  
  console.log ( 'Funktionsausführung' );  
  
});
```

```
if ( b ) resolve('Erfolg:' + z);
else reject ( 'Problem: ' + z );

});
```

Zu guter Letzt fehlen noch die Funktionen, die jeweils durch `resolve` bzw. `reject` aufgerufen werden. Diese Funktionen werden dem Promise-Objekt mithilfe der *then*-Methode übermittelt.

Diese bekommt als Parameter üblicherweise zwei Funktionen.

Die erste Funktion wird ausgeführt, falls der *resolve*-Fall eingetreten ist.

Die zweite Funktion wird ausgeführt, falls der *reject*-Fall eingetreten ist.

Beide Funktionen können nur **einen** Parameter bekommen, der üblicherweise das Ergebnis des Promise enthält.

**Achtung:** Wenn ein *Promise* sein *resolve* oder *reject* ausgeführt hat, gilt dieser als erledigt und wird kein weiteres *resolve* oder *reject* ausführen.

```
let meinPromise = new Promise ( (resolve, reject) => {

  let z = Math.random();
  let b = z > .5;
  console.log ( 'Funktionsausführung' );

  if ( b ) resolve('Erfolg:' + z);
  else reject ( 'Problem: ' + z );

});

meinPromise.then (
  data => console.log ( data ),
  msg => console.log ( msg )
);
```

## Chaining

Die *then*-Angabe kann auch direkt hinter den Konstruktor geschrieben werden:

```
let meinPromise = new Promise ( (resolve, reject) => {

  let z = Math.random();
  let b = z > .5;
  console.log ( 'Funktionsausführung' );

  if ( b ) resolve('Erfolg:' + z);
  else reject ( 'Problem: ' + z );

}).then (
  data => console.log ( data ),
  error => console.log ( error )
```

```
);
```

Auf diese Weise können wir mehrere Funktionen hintereinander ausführen lassen. Jede folgende Funktion arbeitet dann mit dem Rückgabewert der vorangegangenen Funktion weiter. In den folgenden Beispielen lasse ich zur Vereinfachung den *reject*-Event weg.

```
let meinPromise = new Promise ( (resolve, reject) => {  
  
  let count = 1;  
  console.log ( 'Ursprüngliche Funktion: ' + count );  
  
  resolve( count * 2 );  
}).then (   
  data => {  
    console.log ( data )  
    return data * 2;  
  }).then (   
    data => {  
      console.log ( data )  
      return data * 2;  
    }).then (   
      data => {  
        console.log ( data )  
        return data * 2;  
      });  
});
```

## Then ... catch

Neben der Schreibweise mit den zwei Funktionen im then-Event können wir auch den *resolve*- und den *reject*-Event komplett voneinander trennen:

```
let meinPromise = new Promise ( (resolve, reject) => {  
  
  let z = Math.random();  
  let b = z > .5;  
  console.log ( 'Funktionsausführung' );  
  
  if ( b ) resolve('Erfolg:' + z);  
  else reject ( 'Problem: ' + z );  
  
}).then (   
  data => console.log ( data )  
) .catch (   
  msg => console.log ( msg )  
) ;
```

## In der Praxis ...

... werden Promises vor allem dazu verwendet, um Funktionen die Möglichkeit zu geben, eine vom Ergebnis abhängige Callback-Funktion ausführen zu lassen.

Dazu wird der Promise als Rückgabewert der Funktion verwendet

```
function zufall() {  
  
  return new Promise((res, rej) => {  
    let z = Math.random();  
    console.log("Hier passiert etwas.");  
  
    if ( z > .5) {  
      res("Yeah")  
    } else {  
      rej("Nope")  
    }  
  })  
}  
  
const promise = zufall().then(  
  data => console.log ( 'Erfolg: ' + data),  
  error => console.log ( 'Fehler: ' + error)  
);
```

## async, await

Async und await sind erdacht worden, um die Arbeit mit Promises etwas zu vereinfachen.

### async

Eine mit async markierte Funktion wird immer als asynchrone Funktion mit einem Promise als Rückgabewert angelegt.

Ein einfaches Beispiel: Beim Aufruf der Funktion *quadrat* wird ein Parameter übergeben und die *resolve*-Funktion bestimmt.

```
async function quadrat(wert){  
  return ( wert ** 2 );  
}  
  
let ausgabe = data => console.log ( data )  
  
quadrat ( 4 ).then ( ausgabe );
```

Oder mit einer Arrow-Funktion:

```
let quadrat = async wert => wert ** 2;  
  
let ausgabe = data => console.log ( data )  
  
quadrat ( 4 ).then ( ausgabe );
```

## Await

Mit `await` kann der Code an einer Stelle pausiert werden, bis der Promise ein Ergebnis liefert. Da `await` ausschließlich mit asynchronen Funktionen arbeitet, gibt es kein Performance-Problem. Denn der Code außerhalb der asynchronen Funktion läuft ganz normal weiter.

```
async function quadrat(wert){

  let meinPromise = new Promise ( (res, rej) => {
    setTimeout (
      () => res ( wert ** 2 ),
      1000
    );
  });
  let erg = await meinPromise;      // Hier warten, bis der Timeout
                                    // ausgeführt wurde
  console.log ( erg );
}

quadrat ( 12 );
```

## Promise und Ajax

Im Folgenden eine einfache Ajax-Abfrage, bei der die Antwort per *promise* behandelt wird.

```
window.onload = function(){

  let laden = new Promise ( (res, rej) => {

    let xhr = new XMLHttpRequest();
    xhr.open ( 'GET', '_daten.txt' );
    xhr.onreadystatechange = () => {

      if ( xhr.readyState==4 && xhr.status==200 ) res ( xhr.responseText );
      else if ( xhr.readyState == 4 ) rej ( xhr.status );

    }
    xhr.send();

  }).then (

    data => log.innerHTML = data,
    msg => console.log ( 'Fehler: ' + msg )
  );
}
```

Weiterführende Lektüre:

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

<https://javascript.info/promise-basics>

<http://wiki.selfhtml.org/wiki/JavaScript/Promise>

# Ajax

## Anwendungsbeispiele:

- Autovervollständigung in einer Suchmaske (Google)
- praktisch alle Webprojekte, die umfangreiche Daten schnell im Frontend nachladen und verarbeiten müssen ( <http://explorer.worldwind.earth/>, <https://www.google.de/maps>, MS Office 365, u.v.a.m.)
- Interaktives Nachladen von Content ( <http://www.facebook.com>, <https://www.flickr.com>, u.v.a.m.)
- Web-Desktop ( <https://www.eyeos.com>, <https://www.aidroid.com/de/>, etc.)
- Online-Mailclients (Gmail, u.a.)
- Messenger und Chat-Programme

Ajax ist nur ein Marketing-Begriff. Es ist keine Neuerfindung sondern liefert einen Namen für die Möglichkeit, asynchron – also ohne die Notwendigkeit, eine Seite zu aktualisieren – Dateien zu laden. Ursprünglich ging es bei Ajax darum, XML-Dateien vom Server nachzuladen, XML wird allerdings nur selten tatsächlich eingesetzt.

## Was Ajax kann:

- Preload von Suchergebnissen in Suchmaschinen und Shops
- Dateibearbeitung (Office-webapps)
- Alles, wo aufgrund einer Benutzereingabe Daten nachgeladen werden

## Ajax ist ...

- ... das Senden einer http-Anfrage an einen Webserver und die Auswertung des Ergebnisses.
- ... eigentlich ganz einfach.

## Anwendung

### Schritt 1: Ajax-Objekt anlegen.

```
var meinRequest = new XMLHttpRequest();
```

### Schritt 2: Verbindung zur Zieldatei herstellen.

Diese muss sich innerhalb derselben Domain befinden.

```
meinRequest.open('Sendemethode', 'URL_der_Datei', Asynchronität);
```

**Sendemethode:** POST oder GET

GET ist meistens ausreichend.

Post muss verwendet werden, wenn nicht gecached werden darf, viele Daten gesendet werden sollen oder unbekannte Zeichen in den gesendeten Daten sein können.

**URL:** Zeigt auf die zu ladende Datei

**Asynchronität:** Boolean; bestimmt, ob synchron oder asynchron geladen wird. True (asynchron) ist Standard.

Wenn die Asynchronität auf True steht, dann wird das Script weiter ausgeführt, ohne auf das Laden der externen Ressource zu warten. Sobald die Daten fertig geladen wurden, wird ein Eventlistener gefeuert, der überprüft, ob die Daten erfolgreich geladen wurden und dann eine entsprechende Aktion auslöst.

Wenn die Asynchronität auf false steht, bleibt das Script stehen und arbeitet erst weiter, wenn die Daten geladen wurden.

**Parameter übergeben:**

Um mit der **GET**-Methode Parameter zu übergeben, hänge sie an die URL an.

```
meinRequest.open('GET', 'meinScript.php?var1=wert&var2=wert', true);
```

Um mit der **POST**-Methode Parameter zu übergeben, muss zusätzlich ein Requestheader angegeben werden. Dazu gibt es verschiedene Möglichkeiten, je nachdem welche Information übergeben werden sollen.

```
// Einfachster Header. Daten werden wie in der GET-Methode kodiert
meinRequest.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');

// Gut zum Übertragen von Formulardaten. Insbesondere, wenn Dateien
und/oder Binärdaten übertragen werden müssen.
meinRequest.setRequestHeader('Content-type', 'multipart/form-data');

// Daten werden als JSON-String übergeben
meinRequest.setRequestHeader('Content-Type', 'application/json');

// Daten werden als XML-String übergeben und als XML-Objekt übertragen
meinRequest.setRequestHeader('Content-Type', 'text/xml');

// Einfachster Header. Nur für Debugging-Zwecke nützlich
meinRequest.setRequestHeader('Content-Type', 'text/plain');
```

### Schritt 3: Verarbeitung / Ausgabe der geladenen Daten.

Das Attribut **readyState** enthält den Status der Dateiabfrage. Dieser Status wird geändert, wenn es beim Laden einen Fortschritt gibt und enthält einen von fünf Werten:

- 0: Anfrage noch nicht gestartet
- 1: Verbindung zum Server hergestellt
- 2: Anfrage wurde abgesendet
- 3: Anfrage wird vom Server verarbeitet
- 4: Ladevorgang abgeschlossen, Antwort steht bereit



Bei jeder Änderung des `readyState` wird der Eventlistener `onreadystatechange` gefeuert. Das nutzen wir, um auf die Fertigstellung des Ladevorganges zu reagieren (s.u.)

Sobald der Eventlistener gefeuert wurde, muss der Inhalt der Eigenschaft `readyState` überprüft werden. Denn erst, wenn dieser eine 4 enthält, ist die Dateiübertragung abgeschlossen.

```
meinRequest.onreadystatechange = function() {  
    if ( meinRequest.readyState == 4 ){  
        // Dieser Code wird ausgeführt, wenn der Ladevorgang abgeschlossen ist.  
    }  
}
```

Leider sagt der `readyState` nichts darüber aus, ob die Übertragung erfolgreich verlief. Daher enthält das `XMLHttpRequest`-Objekt auch die Eigenschaft `status`. Dieser sagt aus, welchen Status der Webserver hat und wir können diesen verwenden, um zu prüfen, ob die Übertragung erfolgreich war.

Ein paar der Statii sind:

- 200 – Übertragung erfolgreich abgeschlossen
- 401 – Nicht autorisierter Zugriff
- 403 – Zugriff wurde verweigert
- 404 – Datei nicht gefunden
- 408 – Timeout
- 410 – Datei wurde entfernt und steht nicht mehr zur Verfügung
- 451 – Datei ist aufgrund gesetzlicher Bestimmungen nicht verfügbar

Damit können wir den Eventlistener folgendermaßen erweitern:

```
if ( meinRequest.readyState == 4 && meinRequest.status == 200 ){  
    // Wird ausgeführt, wenn Ladevorgang erfolgreich abgeschlossen wurde  
    var daten = meinRequest.responseText;  
} else if ( meinRequest.readyState == 4 ){  
    // Wird ausgeführt, wenn der Serverstatus nach dem Ladevorgang ein  
    // anderer als 200 ist  
    alert ( 'Error ' + meinRequest.status );  
}
```

Sobald nun die Daten geladen wurden, müssen wir noch darauf zugreifen können. Dazu gibt es die Response:

`meinRequest.responseText` gibt den Inhalt als String zurück.

`meinRequest.responseXML` gibt den Inhalt als XML-Objekt zurück.

Eine XML-Datei kann dann durchsucht und analysiert werden.

#### Schritt 4: Senden

Zu guter Letzt muss der Request noch abgesendet werden. Dazu dient der `send`-Befehl.

```
meinRequest.send();
```

Wird die Datei über POST geladen, so müssen an die send-Methode die zu übertragenden Parameter übergeben werden.

```
// Bei Verwendung des urlencoded-Headers  
meinRequest.send ( 'p1=12&p2=42' );
```

Die Dateiverbindung braucht nicht **beendet** zu werden. Sobald der Request abgeschlossen ist, beendet sich die Verbindung zur Datei selbst.

**Ein vollständiges Listing für eine einfache Ajax-Abfrage kann also wie folgt aussehen:**

```
<!doctype html><html><meta charset="utf-8">  
  
<head>  
  <script type="text/Javascript">  
    var meinRequest;  
    var ausgabeelement  
  
    function init() {  
      ausgabeelement = document.getElementById('ausgabe');  
  
      function geladen() {  
        if (meinRequest.readyState == 4 && meinRequest.status == 200) {  
          ausgabeelement.innerHTML = meinRequest.responseText;  
          // Alternativ für XML  
          // var meinXML = meinRequest.responseXML;  
        } else if (meinRequest.readyState == 4) {  
          // Fehlermeldung  
        }  
      }  
    }  
  
    meinRequest = new XMLHttpRequest();  
    meinRequest.open('GET', 'laden.txt', true);  
    meinRequest.onreadystatechange = geladen;  
    meinRequest.send();  
  }  
</script>  
</head>  
  
<body onload="init();">  
  <div id="ausgabe"></div>  
</body>  
  
</html>
```

### XMLHttpRequest-Objekt mehrfach verwenden

Grundsätzlich funktioniert Ajax so, dass ein XMLHttpRequest-Objekt nur einmal benutzt werden kann. Nach der Benutzung ist der readyState auf vier geändert und das Objekt verweigert die Zusammenarbeit.

Um ein XMLHttpRequest-Objekt wiederzuverwenden, muss das Objekt resettet werden:

Dazu definiere die open-Methode jedes Mal neu, wenn die Ajax-Abfrage neu gestartet werden soll.

**Achtung:** Wenn (z.B. bei POST-Anfragen) ein RequestHeader definiert ist, dann muss der nach der Definition der open-Methode ebenfalls neu belegt werden.

```
var meinRequest = new XMLHttpRequest();
meinRequest.onreadystatechange = meineFunktion;

function aendern(){
    // Diese Funktion dient dem wiederholten Aufruf der Send-Methode
    meinRequest.open('POST','script.php',true);
    // Der Header ist nur bei POST-Anfragen notwendig
    meinRequest.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    meinRequest.send('wert1=123');
}

function meineFunktion(){
    if(meinRequest.readyState == 4){
        document.getElementById('ausgabe').innerHTML = meinRequest.responseText;
    }
}
```

## Cross Origin Policy

Um mit Ajax zu arbeiten, sollte grundsätzlich ein Webserver verwendet werden. Dabei ist unerheblich, welche Art von Server (Apache, NodeJS, IIS, etc.) Verwendung findet, solange dieser mit dem http-Protokoll arbeitet.

Ajax-Abfragen müssen dabei in einem Domainspace bleiben, man darf also von einer Domain keine Ajax-Anfrage an eine andere Domain stellen.

Die Möglichkeit, Chrome mit dem Parameter `--allow-file-access-from-files` oder gar mit `--disable-web-security` zu starten, empfiehlt sich nicht, da damit dieses Sicherheitsfeature abgeschaltet wird.

## JSON

Anwendungsbeispiele:

- Umwandeln von Arrays/Objekten, bevor sie per Ajax verschickt oder im Webstorage abgelegt werden.
- Umwandeln von Rückgabewerten eines Ajax-Aufrufes in Arrays/Objekte
- U.v.m.

JSON ist eine Schreibweise, die es ermöglicht, Objekte, Arrays und sonstige Variablen in für Menschen lesbarer Weise niederzuschreiben. Die Idee ist, mit Hilfe der JSON-Methoden komplette Arrays und Objekte von einer Programmiersprache in eine andere zu übertragen.

Im Falle von Ajax heißt dies z.B., dass ein PHP-Script angefragt wird. Dieses liefert möglicherweise ein Array als Rückgabewert. Dieser Rückgabewert muss an das Javascript der Webseite übergeben

werden. Daher wandelt PHP die Daten per JSON in einen String. Und Javascript wandelt diesen String dann wieder zurück in ein Array.

Um Variablen via JSON umwandeln zu können, müssen die Variablen bestimmte Vorgaben erfüllen. Es wird streng zwischen einfachen Hochkommata (') und doppelten (") unterschieden:

- Alle Eigenschaftsnamen in einem Objekt müssen in doppelten Anführungszeichen notiert sein.
- Führende Kommas in Objekten und Arrays sind verboten.
- Bei Zahlen sind führende Nullen verboten und einem Dezimalpunkt muss mindestens eine Ziffer folgen.
- Strings müssen durch doppelte Anführungszeichen begrenzt sein.
- Ein Objekt darf keine Funktionen enthalten

Die Benutzung von JSON in Javascript ist denkbar einfach, es gibt nur zwei Methoden:

```
// Eine vorhandene Variable (Array/Objekt/etc) in JSON-String umwandeln:  
var jsonString = JSON.stringify(meineVariable);  
  
// Einen vorhandenen JSON-String in eine Variable umwandeln:  
var meineVariable2 = JSON.parse(jsonString);
```

## REST

REST oder ReST (Representational State Transfer) ist ein Begriff, der von verschiedenen Branchen unterschiedlich aufgefasst wird. Für Server- und Netzwerkfachleute ist er ein Paradigma, das eine Architektur zur verteilten Verarbeitung von Daten beschreibt ([Strg-Klick für mehr](#)).

Für uns als Softwerker ist diese allgemeine Betrachtungsebene nicht besonders interessant. Spannender ist für uns die sog. Anwendungsschicht. Hier ist beschrieben, wie im Protokoll verschiedene Kommunikationswege funktionieren.

Das bei weitem am häufigsten verwendete Protokoll ist HTTP bzw. HTTPS.

Bei einer Anfrage an den Server wird immer bestimmt, mit welcher Methode die Abfrage erfolgen soll. Mögliche Methoden und ihre Einsatzzwecke sind:

- GET – Fordert eine Ressource (z.B. eine Datei oder eine Information) an.
- POST – Wird meist verwendet, um eine Ressource anzufordern und dabei komplexe Parameter (z.B. Dateien, Formulardaten, etc.) an den Server zu übertragen.
- PUT – Sendet eine Information an den Server.
- DELETE – Löscht eine Ressource vom Server.
- PATCH – Ändert eine Ressource auf dem Server.
- HEAD – Fordert Header-Daten einer Ressource an.
- OPTIONS – Prüft, welche Methoden für eine Ressource verwendet werden können.
- CONNECT – Stellt eine Verbindung her. Wird eigentlich nur in Zusammenhang mit Proxy-Servern gebraucht.
- TRACE – Gibt die Anfrage unverändert zurück. Ähnlich einem Ping.

Die bei Weitem am häufigsten eingesetzten Methoden sind GET und POST. Die anderen Methoden kommen nur selten vor.

Um zu bestimmen, welche Methode verwendet wird, gibt es im Umgang mit Ajax (Vanilla-JS) die open-Methode ([Strg-Klick für mehr](#)):

```
xhr.open ( 'POST', 'beispiel.php', true );
```

Auf Serverseite wird die Anfrage dann angenommen und der Methode entsprechend verarbeitet. Besonders *NodeJS* mit *Express*-Modul tut sich hier durch seinen flexiblen Umgang mit den Methoden hervor ([Strg-Klick für mehr](#)).

## SOAP

Soap (ursprünglich für *Simple Object Access Protocol*) ist ein Netzwerkprotokoll zur Datenübertragung zwischen einem Webservice und einem Client.

Ein Webservice ist eine Software (kann als Blackbox betrachtet werden), welche speziell für Maschine-zu-Maschine-Interaktionen bereitgestellt wird.

Die Idee hinter Soap lässt sich folgendermaßen beschreiben:

Eine vollständige Nachricht (sog. SOAP:Envelope, meist in Form der XML-Variante 'WSDL'), die auf beliebigem Weg an einen Webservice gesendet wird und das Ergebnis abwartet. Diese Nachricht kann Informationen oder auch Funktionsaufrufe enthalten.

Die Form der Nachricht ist vorgeschrieben. Nicht aber, auf welchem Wege sie übertragen wird.

Die Struktur einer Soap-Nachricht in WSDL sieht folgendermaßen aus:

```
<?xml version="1.0"?>
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
    <m:Optional>Daten</m:Optional>
  </s:Header>
  <s:Body>
    <m:Anweisung>Hier werden Anfragen definiert</m:Anweisung>
  </s:Body>
</s:Envelope>
```

Diese Nachricht wird **z.B. per Ajax** an den Webservice übertragen und eine entsprechende Antwort zurückgegeben.

Bei der Verwendung von Ajax muss beachtet werden, dass die Daten per POST übertragen werden und der Header für die Übertragung von XML-Daten eingestellt wird:

```
var envelope = 'SOAP-Envelope als XML-Baum wie oben';

var meinRequest = new XMLHttpRequest();
meinRequest.open('POST', 'script.php', true);

meinRequest.onreadystatechange = function(){
  if(meinRequest.readyState == 4 && meinRequest.status == 200){
    document.getElementById('ausgabe').innerHTML = meinRequest.responseText;
  }
};

// Der Header für SOAP-Abfragen
meinRequest.setRequestHeader('Content-Type', 'text/xml');
```

```
meinRequest.send ( envelope );
```

Die eigentliche Arbeit wird dann auf der Gegenseite erledigt, die diese Anfrage bekommt. Was dort geschieht, ist bei jedem Projekt unterschiedlich.

Die Arbeit mit Soap ist relativ aufwendig und der einfachste Weg, Soap zu implementieren ist die jQuery-Erweiterung: <https://github.com/doedje/jquery.soap>

## Fetch



(Strg-Klick für mehr)

Die *fetch*-API ist eine Alternative zu der oben beschriebenen *Ajax*-API. Sie basiert auf [Promises \(Strg-Klick für mehr\)](#) und wird als die modernere, mächtigere und flexiblere Variante eines Ajax-Aufrufes gesehen.

**Achtung:** Fetch wird nicht in allen Browsern vollständig unterstützt. Hier ist intensives Testing notwendig.

Die *fetch*-Methode bekommt als Pflicht-Parameter den zu ladenden Pfad.

Der Rückgabewert ist ein *Promise*, welcher die Information enthält, ob der Aufruf erfolgreich war oder nicht.

Die Ausgabe ist im Beispiel noch nicht ganz zufriedenstellend, das kommt noch:

```
fetch ( 'http://localhost:8080/daten' ).then(
  data => console.debug ( data ),
  error => console.log ( 'Fehler: ' + error )
)
```

Wenn wir den *Request* genauer definieren wollen, können wir ihn mithilfe des *Request*-Konstruktors als Objekt anlegen und dann verwenden:

```
let req = new Request ( 'http://localhost:8080/daten', {
  method: 'POST',
  headers: new Headers({
    'Content-Type': 'application/json'
  })
});

fetch ( req ).then(
  data => console.log ( data )
)
```

Nun ist die Ausgabe in der Konsole noch nicht besonders aussagekräftig.

Das liegt daran, dass der Rückgabewert des *Fetch*-Befehles selbst ein *Promise* ist.

Um dessen Inhalt auszulesen, müssen wir ihn zunächst umwandeln. Dazu haben wir verschiedene Methoden zur Auswahl:

- `.json()` wandelt die Rückgabe in ein JSON-Objekt um. Vorausgesetzt, die Daten liegen als JSON vor.
- `.text()` wandelt die Rückgabe zu einem String um
- `.blob()` wandelt zu einem Blob (Daten von in JS nicht näher definierter Struktur) um.
- `.formData()` wandelt zu Formulardaten um
- U.a. <https://developer.mozilla.org/en-US/docs/Web/API/Response>

Die Rückgabewerte der oben genannten Methoden sind ihrerseits auch *Promises*. D.h., sie müssen mit einem verketteten *Then* verarbeitet werden, falls sie erfolgreich umgewandelt wurden.

Im Beispiel gehen wir davon aus, dass der Rückgabewert ein Objekt ist.

```
let req = new Request ('http://localhost:8080/daten', {
  method: 'POST',
  headers: new Headers({
    'Content-Type': 'application/json'
  })
});

fetch ( req )
  .then(
    data => data.json()
  ).then(
    data => {
      console.log ( data );
      // Irgendwas cooles
    }
  );
```

## Parameter übergeben

Um Parameter an den Server zu übergeben, nutzen wir am besten die POST-Methode.

Im Grunde nehmen wir das obige Beispiel und ergänzen es durch die zu übergebenden Parameter:

```
let kopf = new Headers({
  'Content-Type': 'application/json'
});

let daten = {
  p1: 42,
  p2: 23
};

let req = new Request ('http://localhost:8080/daten', {
  method: 'POST',
  headers: kopf,
  body: JSON.stringify(daten)
});
```

```
fetch ( req )
  .then(
    data => data.json()
  ).then(
    data => {
      console.log ( data );
    }
  );
```

## Datei hochladen

Ausgehend von einem NodeJS-Server, der wie in der dazugehörigen Lektion aufgebaut ist, sieht der Fetch-Aufruf folgendermaßen aus:

```
dateien.onChange = () => {

  var formular = new FormData ( document.querySelector("form") );

  let req = new Request ( 'http://localhost:8080/fileupload', {
    method: 'POST',
    body: formular
  });

  fetch ( req )
    .then(
      data => data.text()
    ).then(
      data => {
        console.log ( data );
      }
    );
}
```

## Polyfill

<https://github.com/github/fetch>



# Websockets

Anwendungsbeispiele:

- Online-Games
- Chat-Anwendungen
- Live-Ticker
- Soziale Streams
- Etc.

Ajax ist praktisch und funktioniert super, wenn es darum geht, Daten zu übertragen und es nicht auf den einen oder anderen Sekundenbruchteil ankommt.

Das http-Protokoll, auf dem Ajax basiert, ist allerdings sehr aufwendig und damit langsam. Und es kann auch andere Wünsche nicht abdecken. Z.B. eine Möglichkeit, dem Client mitzuteilen, dass neue Daten zur Verfügung stehen u.ä.

Manchmal (z.B. bei Onlinegames) helfen Ajax und dessen Workarounds wie *lange Abfragen*, *Push*, *XHR-Multipart*, etc. nicht wirklich weiter.

Aus dem Grunde wurden **Websockets** ins Leben gerufen.

Bei Websockets (dt. etwa: Netz-Sockel) wird eine dauerhafte Verbindung zum Server aufrecht erhalten, über die sowohl Server als auch Client jederzeit Daten senden können.

Um die vielen gleichzeitigen Verbindungen verwalten zu können, brauchen wir einen grundsätzlich neuen serverseitigen Aufbau. Hierfür bietet sich – neben anderen - NodeJS mit der Erweiterung SocketIO an.

Ab hier wird davon ausgegangen, dass eine laufende NodeJS-Umgebung zur Verfügung steht und damit umgegangen werden kann. Mehr zu NodeJS findet sich hier ([Strg-Klick](#))

## Vorbereitung

Zunächst wird socket.io im Terminal installiert.

```
npm install socket.io
```

Socket.io setzt auf dem http-Protokoll auf, das wiederum aus dem Server herausgelöst wird. Daher richten wir zunächst einen Webserver ein (am einfachsten geht das mit Express).

Dann nehmen wir das http-Protokoll und geben es an das Objekt, das wir mit dem socket.io-Modul angelegt haben.

Damit haben wir dann die WebSocket-Funktionalität zur Verfügung und können z.B. die Eingaben im Terminal an alle Clients übertragen.

**Achtung:** In der letzten Zeile muss der richtige Server stehen, damit auch Websockets geht

```
'use strict';

// Webserver einrichten
let express = require ( 'express' );
```

```

let server = express();
server.use ( express.static ( 'public' ) );

// Socket.io einrichten
let http = require ( 'http' );
let webServer = http.Server ( server );
let socketIo = require ( 'socket.io' );
let io = socketIo ( webServer );

// Benutzer loggen, sobald sie sich verbinden
io.on ( 'connection', socket => {
    console.log ( 'neuer Client: ' +
        socket.id + ' von ' +
        socket.conn.remoteAddress );
});

// Jede Eingabe im Protokoll wird als 'nachricht' an alle Clients
übertragen
let stdin = process.openStdin();
stdin.addListener ( 'data', d => {
    io.emit ( 'nachricht', d.toString() );
});

// Auf Port 8080 horchen
webServer.listen ( 8080, 'localhost' );

```

Wenn das erledigt ist, müssen wir noch die Client-Seite vorbereiten. Also die Datei, die vom Browser angezeigt wird, mit dem Websocket verbinden.

Dazu wird zunächst die Datei `/socket.io/socket.io.js` als Javascript eingebunden. Diese Datei wird vom Modul `socket.io` automatisch zur Verfügung gestellt und braucht nirgends als Datei vorzuliegen. Dies ist die Bibliothek, die in der Webseite die Websockets-Funktionalität zur Verfügung stellt.

Dann wird mittels `io.connect()` eine Verbindung zum Websocket hergestellt und diese in einer Variablen gespeichert.

Schließlich kommt ein *EventListener*, der auf das Eintreffen einer Nachricht mit Namen `'nachricht'` reagiert: die Funktion des Eventlisteners gibt den Inhalt der Nachricht in einem DOM-Element aus.

```

<!doctype html>
<html>

<head>
  <title>Websocket</title>
  <meta charset="utf-8">

  <script src="/socket.io/socket.io.js"></script>
  <script>

    var socket = io.connect();
    socket.on ( 'nachricht', data => {
      var element = document.querySelector ( '#ausgabe' );
      element.innerHTML += data + '<br>';
    });
  </script>

```

```
</script>
</head>

<body>
  <h1>Websocket</h1>
  <div id="ausgabe"></div>
</body>

</html>
```

Nun kann auf dem Terminal des Servers beliebiger Text eingegeben werden. Nach einem Druck auf die Enter-Taste erscheint in allen angeschlossenen Browsern der eingegebene Text.

## Senden

Im nächsten Schritt wollen wir dem Client ermöglichen, selbst Nachrichten abzusenden.

Dazu muss gar nicht viel geändert werden. Die Logik ist genau wie vorher, nur mit umgedrehten Rollen.

Zunächst bekommt der Client (die HTML-Seite) einen Eventlistener, der auf eine Eingabe in einem Input-Element wartet, die dann mit der `emit`-Methode an den Server übergeben wird.

Dann erweitern wir den Server um einen *EventListener*, der auf eine bestimmte Nachricht eines jeden Clients wartet. Wenn diese Nachricht kommt, leitet der Server diese an die anderen verbundenen Clients weiter.

Fertig ist das Chat-Programm. Hier nochmal das vollständige Listing:

## Server

```
'use strict';

// Webserver einrichten
let express = require ( 'express' );
let server = express();
server.use ( express.static ( 'public' ) );

// Socket.io einrichten
let http = require ( 'http' );
let webServer = http.Server ( server );
let socketIo = require ( 'socket.io' );
let io = socketIo ( webServer );

// Benutzer loggen, sobald sie sich verbinden
io.on ( 'connection', socket => {
  console.log ( 'neuer Client: ' +
    socket.id + ' von ' +
    socket.conn.remoteAddress );

  socket.on ( 'data', daten => {
    console.log ( daten );
    io.emit ( 'nachricht', daten );
  });
});
```

```
// Jede Eingabe im Protokoll wird als 'nachricht' an alle Clients
übertragen
let stdin = process.openStdin();
stdin.addListener ( 'data', d => {
  io.emit ( 'nachricht', d.toString() );
});

// Auf Port 8080 horchen
webServer.listen ( 8080, 'localhost');
```

## Client

```
<!doctype html>
<html>

<head>
  <title>Websocket</title>
  <meta charset="utf-8">
  <script src="/socket.io/socket.io.js"></script>
  <script>
    window.onload = function(){
      var socket = io.connect();

      socket.on ( 'nachricht', data => {
        var element = document.querySelector ( '#ausgabe' );
        element.innerHTML += data + '<br>';
      });

      eingabe.onchange = function(){
        socket.emit ( 'data', this.value );
        this.value = '';
      }
    }
  </script>
</head>

<body>
  <h1>Websocket</h1>
  <input id="eingabe" />
  <div id="ausgabe"></div>
</body>
</html>
```

## PM (Personal Messages)

Man kann auch Nachrichten an einen spezifischen Client senden. Dazu müssen wir zunächst die ID des Users beim Anmelden in eine Variable ablegen.

```
let dieID = '';

io.on ( 'connection', socket => {
  dieID = socket.id;
});
```

Diese ID kann nun verwendet werden, um diesem einen User eine Nachricht zu senden.  
Zum Beispiel bei einer Eingabe in das Terminal:

```
let dieID = '';  
  
io.on ( 'connection', socket => {  
  dieID = socket.id;  
});  
  
let stdin = process.openStdin();  
stdin.addListener ( 'data', data => {  
  if (io.sockets.connected[dieID]) {  
    io.sockets.connected[dieID].emit ('nachricht', 'Nur für Dich');  
  }  
});
```

Bitte beachte, dass beim obigen Beispiel immer die ID des zuletzt angemeldeten Users als *dieID* gespeichert ist. IRL wäre sicher ein Array, aus dem dann die passende ID gepickt wird, die bessere Wahl.

# XML

Die ursprüngliche Idee hinter Ajax war – außer dass unbedingt ein X im Namen vorkommen sollte – dass XML-Dateien geladen und verarbeitet werden können.

XML – eXtensible Markup Language – ist ein universelles Dateiformat, um beliebige Dateien zu speichern.

Eine typische XML-Datei beginnt aus einer Einführungs-Zeile:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Ihr folgen die Zeilen, welche die Informationen enthalten. Diese sind ähnlich wie HTML in Anfangs- und Endtags organisiert, zwischen denen sich die Informationen befinden.

Es gibt keine vorgefertigten Tagnamen, diese überlegt sich der Autor der XML-Datei selbst.

Die Tags können beliebig verschachtelt werden, wobei immer ein Knoten (root) über allem stehen muss.

Eine beispielhafte XML-Datei könnte so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<familie>
  <mitglied rolle="Vater">
    <vname>
      Max
    </vname>
    <nname>
      Mustermann
    </nname>
    <alter>
      44
    </alter>
  </mitglied>
  <mitglied rolle="Kind">
    <vname>
      Kevin
    </vname>
    <nname>
      Müller-Mustermann
    </nname>
    <alter>
      16
    </alter>
  </mitglied>
</familie>
```

Es gibt in diesem Beispiel keinen besonderen Grund, warum die Rolle als Attribut angelegt ist außer den, dass es geht.

Angenommen, im Ajax-Aufruf wurde das XML-Objekt in die Variable `meinXML` geschrieben. Um ein bestimmtes Element **anzusprechen**, haben wir dieselben Befehle zur Verfügung wie beim Lesen des DOM.

```
// Um den ersten 'mitglied'-Knoten anzusprechen:
meinXML.getElementsByTagName('mitglied')[0];

// Um das Attribut des ersten Knotens anzusprechen:
meinXML.getElementsByTagName('mitglied')[0].getAttribute('rolle');

// Um das Attribut des ersten Knotens zu ändern:
meinXML.getElementsByTagName('mitglied')[0].setAttribute('rolle',
'neuerWert');

// Um dessen erstes Kind (in diesem Beispiel 'vname') anzusprechen:
meinXML.getElementsByTagName('mitglied')[0].children[0];

// Um den Inhalt dieses ersten Kindes anzusprechen:
meinXML.getElementsByTagName('mitglied')[0].children[0].childNodes[0].nodeValue;
// Oder einfacher (geht nicht im IE):
meinXML.getElementsByTagName('mitglied')[0].children[0].innerHTML;

// Um den Inhalt dieses ersten Kindes zu ändern:
meinXML.getElementsByTagName('mitglied')[0].children[0].childNodes[0].nodeValue = 'neuer Wert';
// Oder einfacher (geht nicht im IE)
meinXML.getElementsByTagName('mitglied')[0].children[0].innerHTML = 'neuer Wert';
```

Ein **neues Element** im XML-Baum einzufügen, funktioniert ebenso wie im DOM:

```
// Das erste Mitglied der Familie erhält ein neues Kind namens 'arbeit'
var e = meinXML.createElement('arbeit');
var t = meinXML.createTextNode('irgendein Arbeitgeber');
e.appendChild(t);
meinXML.getElementsByTagName('mitglied')[0].appendChild(e);
```

## Whitespaces

XML-Dateien sind meistens so aufgebaut, dass zwischen den Tags sowie zwischen Tag und Inhalt zusätzliche Leerzeichen und Zeilenumbrüche sind. Das kann zu Problemen führen, wenn der Inhalt eines Knotens ausgegeben oder mit einer Vorgabe verglichen werden soll.

Besonders deutlich zeigt sich das Problem, wenn der Inhalt eines XML-Knotens in einem Textarea ausgegeben wird, weil dieser keine HTML-Interpretation vornimmt.

Um diese sog. Whitespaces zu entfernen, biete sich die String-Methode `trim()` an. Sie entfernt alle unnötigen Zeichen vor und hinter einem String und behebt damit das Problem:

```
// Angenommen, der XML-Baum hat an der entsprechenden Stelle 'Kevin'
gespeichert
var x =
meinXML.getElementsByTagName('vname')[2].childNodes[0].nodeValue.trim();

document.getElementById('ausgabe').innerHTML = '|' + x + '|';
document.getElementById('textarea').value = '|' + x + '|';

if( x == 'Kevin' ){
alert('Ich bin Kevin');
} else {
    alert('Ich weiß nicht, wer ich bin.');
```



# Dateiupload

Dateien auf einen Server zu laden ist eine wichtige Fähigkeit einer modernen Webseite. Dies geht auch komplett ohne Javascript mithilfe eines Formulars und eine PHP-Skriptes.

Hier wollen wir die Verwendung von JS beleuchten.

## NodeJS

Auf der Serverseite müssen wir die hochgeladenen Dateien empfangen und im Dateisystem speichern können. Im Rahmen dieses Lehrganges verwenden wir dafür NodeJS.

Im Folgenden erstellen wir einen Server, der vom Client Daten in Form von Formulardaten entgegennimmt. Diese haben sich als der einfachste Weg herausgestellt, Dateien zu übertragen.

Um vom Client über das *HTTP*-Protokoll angesprochen werden zu können, verwenden das Modul *Express*, das uns die Einrichtung eines Webserver erleichtert. Mehr zum Einrichten des Servers [hier \(Strg-Klick\)](#).

```
'use strict';

const express = require ( 'express' );
const server = express();

server.use ( express.static ( 'public' ) );

server.listen ( 80 );
```

Im nächsten Schritt müssen wir die vom Client gesendeten Daten im Server entgegennehmen und im Dateisystem ablegen.

Dazu gibt es eine reiche Auswahl an Modulen, ein recht beliebtes ist das Modul *Formidable*.

```
npm install formidable // Terminal-Befehl, um das Modul zu installieren
```

Dieses und das Modul *fs* binden wir mit *require* ein.

```
const formidable = require('formidable');
const fs = require('fs');
```

Dann teilen wir dem Server mit, dass er auf eine HTTP-POST-Anfrage an die Adresse */fileupload* reagieren soll.

```
server.post ( '/fileupload', (req, res) => {
});
```

Schließlich legen wir eine Variable an, die mit einem Objekt gefüllt wird, das vom `IncomingForm`-Konstruktor erzeugt wird.

Dieses Objekt erhält ein paar Einstellungen, um z.B. mehrere Dateien annehmen zu können.

```
var form = new formidable.IncomingForm();
form.uploadDir = 'public/upload/';
form.keepExtensions = true;
form.multiples = true;
```

Damit ist der Upload und das Ablegen der Dateien im Grunde erledigt.

Allerdings sind die Dateien nun mit einem temporären Dateinamen belegt, was in den wenigsten Fällen hilfreich ist. Daher müssen wir jede hochgeladene Datei umbenennen/verschieben. Das erledigt die `parse`-Methode für uns, die Teil des eben angelegten `form`-Objektes ist.

Wenn wir mehrere Dateien übergeben, ist `filetoupload` automatisch ein Array von `file`-Objekten. Sonst ist es ein `file`-Objekt. Daher unterscheiden wir in der `parse`-Methode zunächst, ob `filetoupload` ein Array ist oder nicht. Das geht am einfachsten durch eine Prüfung, ob es die `forEach`-Methode enthält.

Schließlich rufen wir für jede Datei eine Funktion auf, die das Verschieben erledigt.

```
form.parse(req, function (err, fields, files) {

  if ( files.filetoupload.forEach ){
    files.filetoupload.forEach ( file => renameMe(file) );
  } else {
    renameMe( files.filetoupload );
  }

  res.writeHead ( 200, { 'Content-Type': 'text/html' } ) ;
  res.end('File uploaded and moved!');

});
```

Jetzt fehlt nur noch die Funktion `renameMe`. Diese nimmt jedes `file`-Objekt und verwendet die Attribute `path` und `name`, um die Datei mit dem `fs`-Modul zu verschieben/umzubenennen.

Hier der gesamte Code:

```
'use strict';

const express = require ( 'express' );
const server = express();

const formidable = require('formidable');
const fs = require('fs');

server.use ( express.static ( 'public' ) );

server.post ( '/fileupload', (req, res) => {
  var form = new formidable.IncomingForm();
```

```

form.uploadDir = 'public/upload/';
form.keepExtensions = true;
form.multiples = true;

form.parse(req, function (err, fields, files) {
  if ( files.fileupload.forEach ){
    files.fileupload.forEach ( file => renameMe(file) );
  } else {
    renameMe( files.fileupload );
  }

  res.writeHead ( 200, {'Content-Type': 'text/html' } ) ;
  res.end('File uploaded and moved!');
});
});

server.listen ( 80 );

function renameMe(file){
  var oldpath = file.path;
  var newpath = 'public/upload/' + file.name;
  fs.rename(oldpath, newpath, function (err) {
    if (err) throw err;
  });
}

```

Die Server-Seite ist damit vorbereitet. Der obige Code funktioniert mit allen folgenden Upload-Varianten.

Nun folgt die Client-Seite.

## Upload über ein Formular

Der einfachste Weg, Datei-Uploads zu generieren funktioniert vollständig ohne Javascript.

Wir erzeugen uns ein Formular, das mit der *POST*-Methode seine Daten an den Pfad *fileupload* sendet.

Damit werden alle Daten aus dem Formular gesammelt und als *form-data* an die Adresse */fileupload* gesendet. Alles andere geschieht auf dem Server.

Diese Herangehensweise ist zwar schön simpel, hat aber den entscheidenden Nachteil, dass die aktuelle Seite durch die Antwort des Servers überschrieben wird. Würden wir also auf der Seite bleiben wollen, könnten wir hiermit nicht arbeiten

```

<form action="fileupload" method="post" enctype="multipart/form-data">
  <input type="file" name="fileupload" multiple><br>
  <input type="submit">
</form>

```

## Upload mit Ajax

Wenn wir den Upload mit Ajax realisieren, dann bleibt die Seite auf dem Bildschirm und wir können mit der Antwort des Servers anstellen, was wir wollen. Es kann eine Erfolgsmeldung erzeugt werden, die Bilder könnten als Thumbnails angezeigt werden, etc.

In dieser Herangehensweise reagieren wir auf eine Änderung des file-Inputfeldes.

Wenn wir also die hochzuladenden Dateien ausgewählt haben, wird der Event ausgelöst. Die Formulardaten werden in einem extra dafür gedachten *FormData*-Objekt gespeichert, das *XMLHttpRequest*-Objekt wird angelegt, die Datenübertragung wird vorbereitet und schließlich wird mit der *send*-Methode das *FormData*-Objekt an den Server übertragen.

War die Übertragung erfolgreich, bekommen wir das Ergebnis im *Div*-Container mit der ID *Ausgabe* angezeigt.

```
<script type="text/javascript">

window.onload = () => {
  dateien.onChange = event => {

    var fileData = new FormData(document.querySelector('#dateienForm'));
    var xhr = new XMLHttpRequest();
    xhr.open("POST", "/fileupload", true);

    xhr.onload = function() {
      if (xhr.status == 200) {
        document.querySelector('#ausgabe').innerHTML = xhr.responseText;
      } else {
        console.log("Fehler " + xhr.status, xhr);
      }
    };

    xhr.send(fileData);
  }
}

</script>

</head>
<body>
  <form id="dateienForm" enctype="multipart/form-data">
    <input type="file" name="filetoupload" id="dateien" multiple>
  </form>

  <div id="ausgabe">
  </div>

</body>
```

## Upload mit Fetch

Ausgehend vom NodeJS-Server von weiter oben, sieht ein möglicher [Fetch-Aufruf \(Strg-Klick für mehr\)](#) folgendermaßen aus:

```
<head>
  <script type="text/javascript">

    window.onload = () => {
      dateien.onChange = () => {
        var formular = new FormData ( document.querySelector("form") );
        let req = new Request ( 'http://localhost:8080/fileupload', {
          method: 'POST',
          body: formular
        });

        fetch ( req )
          .then(
            data => data.text()
          ).then(
            data => {
              console.log ( data );
            }
          );
      }
    };

  </script>
</head>

<body>
  <form>
    <input id="dateien" type="file" name="filetoupload" multiple>
  </form>
</body>
```

## Drag 'n' Drop

Moderne Browser behandeln das *input type="file"*-Element wie ein natives Drag'n'Drop-Element. Passe es einfach mit ein wenig CSS an und die Sache läuft in den meisten Browsern.

Falls nicht:

<https://www.html5rocks.com/de/tutorials/file/dndfiles/>

# Canvas

Anwendungsbeispiele:

- Grafische Darstellung von Daten (Diagramme)
- Animation von Elementen im Hintergrund
- Darstellung von Spielinhalten
- Zeichenprogramm
- U.v.m.

Der Canvas (dt: 'Leinwand') ist ein Bereich auf der Webseite, der einigermaßen frei bemalt werden kann. Dazu stehen z.B. Linien, Kästen und Kreise zur Verfügung, die beliebig eingefärbt und gezeichnet werden können.

Eingesetzt wird das Element beispielsweise für Bildbearbeitung (<https://www.picozu.com>), in Spielen (<https://playcanvas.com>), bei der Datenvisualisierung, für interaktive, grafische Benutzungskonzepte (<http://printio.ru/tees/new>) und vielem mehr.

Um einen Canvas in die Seite einzubinden, verwenden Sie den HTML-Tag `canvas`. Die Ausdehnung des Canvas muss im HTML-Code mit den `width`- und `height`-Attributen bestimmt werden, da CSS-Angaben die vorhandene Ausdehnung skaliert und damit die Darstellung verzerrt.

```
<!-- Eine ID wird empfohlen, um auf den Canvas zuzugreifen -->
<canvas id="meinCanvas" width="800" height="400">
</canvas>
```

Dann wird mit Javascript in den Canvas hineingezeichnet. Dazu empfiehlt es sich, den Canvas zunächst in einer Variablen zu referenzieren und zu bestimmen, auf welche Weise (Context) in den Canvas gezeichnet werden soll:

```
var c = document.getElementById('meinCanvas');
var ctx = c.getContext('2d');
```

Nun kann in den Canvas gezeichnet werden.

**Hinweis:** Der IE hat u.U. seine liebe Not mit der Methode `getContext`. Damit der richtig funktioniert, muss im HTML-Quellcode die Kompatibilität eingestellt sein:

```
<meta http-equiv="X-UA-Compatible" content="chrome=1, IE=edge" />
```

**Hinweis:** Diese Referenz ist nicht vollständig – einfach, weil das Canvas-Element so vielfältig ist. Für genauere Informationen ziehen Sie bitte weiterführende Literatur zu Rate.

## Farbfüllung

### Zeichen- und Füllstile

Die Zeichen- und Füllstile können vor jedem Zeichenschritt definiert werden. Wird der Stil nicht definiert, so wird der zuvor definierte Stil verwendet bzw. die Voreinstellung.

```
// Strichdicke einstellen
ctx.lineWidth = 5;

// Strichfärbung (eine der drei Alternativen):
// Angabe eines CSS-Farbwertes vergibt eine Farbe
ctx.strokeStyle = '#f40';

// Angabe eines Verlaufes zeichnet die Linie mit einem Verlauf.
// Mehr zu Verläufen weiter oben.
ctx.strokeStyle = meinVerlauf;

// Angabe eines Pattern zeichnet die Linie mit der Füllung
// Mehr zu Patterns weiter oben.
ctx.strokeStyle = meinPattern;

// Definieren, wie die Enden der Linien gezeichnet werden.
// Optionen: butt, round, square
ctx.lineCap = "round";

// Definieren, wie Ecken gezeichnet werden
// Optionen: bevel, round, miter
ctx.lineJoin = "round";

// Füllfarbe (eine der drei Alternativen):
// Angabe eines CSS-Farbwertes vergibt eine Farbe
ctx.fillStyle = '#f40';

// Angabe eines Verlaufes füllt die Form mit einem Verlauf.
// Mehr zu Verläufen weiter oben.
ctx.fillStyle = meinVerlauf;

// Angabe eines Pattern füllt die Form mit der Füllung
// Mehr zu Patterns weiter oben.
ctx.fillStyle = meinPattern;
```

## Verlauf

**Bevor wir zu zeichnen anfangen, zunächst ein paar Grundlagen über die Verwendung von Verläufen, Stilen, etc.**

Ein Gradient ist ein Verlauf, der sich sowohl für die Füllung von Flächen als auch von Linien verwenden lässt. Er muss als Variable angelegt werden, damit diese dann dem Attribut fillStyle bzw. strokeStyle zugewiesen werden kann.

### Linearer Verlauf :

Die Methode createLinearGradient erzeugt einen linearen(geraden) Verlauf. Dazu werden die folgenden vier Parameter übergeben:

X-Position des Startpunktes, Y-Position des Startpunktes, X-Position des Endpunktes, Y-Position des Endpunktes

Danach werden die Farben mittels Stopp-Punkten definiert. Die Position 0 ist der Startpunkt, Position 1 ist der Endpunkt.

```
// Linearen Verlauf anlegen
var meinVerlauf = ctx.createLinearGradient(0,0,0,170);
meinVerlauf.addColorStop(0,"#f00");
meinVerlauf.addColorStop(0.5,"#00f");
meinVerlauf.addColorStop(1,"#0f0");
```

### Radialer Verlauf:

Die Methode createRadialGradient erzeugt einen kreisförmigen Verlauf. Dazu werden die folgenden sechs Parameter übergeben:

X-Position des Startkreises, Y-Position des Startkreises, Radius des Startkreises in px, X-Position des Endkreises, Y-Position des Endkreises, Radius des Endkreises in px.

Danach werden die Farben mittels Stopp-Punkten definiert. Die Position 0 ist der Startradius, Position 1 ist der Endradius.

```
// Linearen Verlauf anlegen
var meinVerlauf = ctx.createRadialGradient (300, 300, 100, 200, 200, 300);
meinVerlauf.addColorStop(0,"#f00");
meinVerlauf.addColorStop(0.5,"#00f");
meinVerlauf.addColorStop(1,"#0f0");
```

### Muster, Pattern (Bildfüllung)

Ein Pattern ist eine auf einem Bild basierende Füllung. Dabei kann ein Bild-Objekt aus dem DOM, ein neues Bild-Objekt, ein Video-Element oder ein anderes Canvas-Element genommen werden.

**Achtung:** Wenn ein Bild das erste Mal geladen wird, dann liegt es noch nicht im Speicher und der Versuch, es zu zeichnen, wird ohne Bilddaten unternommen. Es wird also nichts gezeichnet. Zum Beheben des Problems wartet man, bis die Bilddaten im Speicher liegen:

```
// Neues Bild-Objekt laden
var imgObj = new Image();           // Anlegen des Image-Objektes
imgObj.src = "bild.jpg";           // Zuweisung einer URL
imgObj.onload = function() {
    meinPattern = ctx.createPattern(imgObj, 'repeat'); //Füllstil anlegen

    ctx.fillStyle = meinPattern;    // Füllstil zuweisen
    ctx.beginPath();
    ctx.moveTo(100,100);
    ctx.lineTo(400,100);
    ctx.lineTo(500,400);
    ctx.lineTo(10,400);
    ctx.closePath();
    ctx.fill();
}
```



```
// Bild aus dem DOM kopieren
// Natürlich muss das Element mit der ID auch vorliegen
var imgObj2 = document.getElementById('tier');

meinPattern2 = ctx.createPattern(imgObj2, 'repeat');

ctx.fillStyle = meinPattern2;
ctx.beginPath();
ctx.moveTo(200, 600);
ctx.lineTo(700, 600);
ctx.lineTo(700, 200);
ctx.lineTo(200, 200);
ctx.closePath();
ctx.fill();
```

## Formen

### beginPath

Um mit dem Zeichnen einer Form zu beginnen, sollte `beginPath()` eingetragen werden. Damit werden vorige Formen, die noch nicht mit `stroke()` oder `fill()` auf den Bildschirm gebracht wurden, aus dem Speicher genommen. Außerdem wird jede Verbindung zu vorherigen Zeichnungen beendet. Andernfalls bekäme man Verbindungslinien, die man nicht haben will.

```
ctx.beginPath();
```

### Stroke

Canvas-Befehle zeichnen die Form nicht automatisch auf den Bildschirm, da jeder Zeichenvorgang Rechenzeit beansprucht. Statt dessen wird das Zeichnen explizit dann befohlen, wenn es ausgeführt werden soll:

```
ctx.stroke();
```

### Fill

Die Methode `fill` arbeitet wie `Stroke`, zeichnet aber ausschließlich die Füllung auf den Bildschirm. Im den Rand und die Füllung zu sehen, verwende beide Methoden.

```
ctx.fill();
```

### Rechteck zeichnen

Zum Zeichnen eines Rechtecks, benutze die Methode `rect`. Die Parameter bezeichnen: X-Position, Y-Position, Breite, Höhe.

```
// ein gefülltes Rechteck zeichnen
ctx.beginPath();
ctx.lineWidth=5;
ctx.fillStyle="#FF0000";
ctx.strokeStyle = '#cba';
ctx.rect(300,300,100,50);
ctx.fill();
ctx.stroke();
```

## Linien zeichnen

Um eine Linie zu zeichnen, bestimme den Zeichenstil, den Startpunkt und Eckpunkte der Linie und führe am Ende mit Stroke die Zeichnung aus. Es ist empfehlenswert, das Zeichnen einer Linie mit `beginPath` zu beginnen, damit die Linie wieder gelöscht werden kann.

```
ctx.strokeStyle = '#f40';
ctx.beginPath();
ctx.moveTo(100,100);
ctx.lineTo(200,200);
ctx.lineTo(100,300);

// Um eine Linie zu beenden und an einem neuen Punkt eine neue Linie zu
beginnen
// wechsele die Position mit 'moveTo'
ctx.moveTo(300,100);
ctx.lineTo(500,200);
ctx.lineTo(300,300);

// closePath zeichnet eine Linie vom Endpunkt zum Startpunkt (dem letzten
moveTo)
ctx.closePath();
ctx.stroke();
ctx.fill();
```

## Kurven zeichnen

Wir haben drei verschiedene Methoden, um Kurven zu zeichnen. Der Startpunkt der Kurve ist immer die aktuelle Position auf der Zeichenfläche. Ggf. steuere den Startpunkt mit der `moveTo`-Methode an.

Der Begriff 'Stützpunkt' (SP) bezeichnet einen Punkt, der die Kurve in seine Richtung 'zieht'.

```
// Kurve mit einem Stützpunkt
ctx.beginPath();
ctx.lineWidth=5;
ctx.moveTo(200,200);
ctx.quadraticCurveTo(180,0,500,200); // Parameter:
    // X des SP, Y des SP,
    // X des Endpunktes, Y des Endpunktes,
ctx.stroke();
```

```
// Kurve mit zwei Stützpunkten
ctx.beginPath();
ctx.moveTo(20,20);
ctx.bezierCurveTo(20,100,200,100,200,20); // Parameter:
// X des 1. SP, Y des 1. SP,
// X des 2. SP, Y des 2. SP,
// X des Endpunktes, Y des Endpunktes,

ctx.stroke();

// Bogen schlagen
ctx.beginPath();
ctx.moveTo(40,40);
ctx.arcTo(200,40,200,200,160); // Parameter:
// X des SP, Y des SP,
// X des Endpunktes, Y des Endpunktes
// Radius

ctx.stroke();
```

## Kreis(-bogen) zeichnen

Kreise werden generell als Kreisbögen gezeichnet, auch wenn es volle Kreise werden sollen.

Die Parameter sind: X-Position, Y-Position, Radius, Startwinkel, Endwinkel, Richtung.

Die Winkel werden grundsätzlich in Radians angegeben, d.h., der volle Kreisumfang ist  $2\pi$ .

0° ist die rechte Kante des Kreises.

Der Parameter für die Richtung kann true (Gegen den UZS) oder false (im UZS, default) sein

```
// einen ganzen Kreis zeichnen
ctx.beginPath();
ctx.lineWidth=5;
ctx.strokeStyle = '#abc';
ctx.arc(95,50,40,0,2*Math.PI);
ctx.stroke();

// einen gefüllten Halbkreis gegen den UZS zeichnen
ctx.beginPath();
ctx.lineWidth=5;
ctx.fillStyle="#FF0000";
ctx.strokeStyle = '#cba';
ctx.arc(95,100,40,0,Math.PI,true);
ctx.fill();
ctx.stroke();
```

## Text

Die Methoden `fillText` und `strokeText` zeichnen Text auf den Bildschirm.

Sie benötigen keinen `stroke()` und auch keinen `fill()`.

Die Methode `font` verwendet dieselbe Syntax wie die im CSS. Dabei ist zu beachten, dass die `font-family` als letztes genannt wird.

**Achtung:** Der Text interpretiert keinen HTML-Quellcode – schade, eigentlich.

```
ctx.textAlign = "center";
ctx.font = '60px arial';
ctx.strokeText('Hallo Welt',200,50);
ctx.fillText('Alles Cool',200,90);
```

Ein Problem beim Zeichnen von Text ist die Verwendung von **Webfonts**, also Schriften, die aus dem Web nachgeladen werden. Wenn diese eingebunden werden, dann funktionieren sie zunächst nicht. Das liegt daran, dass der Schriftsatz nicht so schnell geladen werden kann, dass er *sofort* zur Verfügung steht.

Um dem Problem zu begegnen, eignet sich der folgende Weg:

Wenn auf der Webseite ein Element existiert, das diese Schrift verwendet (auch wenn es versteckt ist), dann wird die Schrift geladen, bevor der Event `window.onload` feuert.

Wenn also die Textdarstellung im `window-onload`-Event steht, sollte sie funktionieren.

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');

var link = document.createElement('link');
link.rel = 'stylesheet';
link.type = 'text/css';
link.href = 'http://fonts.googleapis.com/css?family=Vast+Shadow';
document.getElementsByTagName('head')[0].appendChild(link);

// Trick from http://stackoverflow.com/questions/2635814/
var image = new Image;
image.src = link.href;
image.onerror = function() {
    ctx.font = '50px "Vast Shadow"';
    ctx.textBaseline = 'top';
    ctx.fillText('Hello!', 20, 10);
};
```

## Bilder setzen

Bilder, die als Image-Objekt vorliegen (siehe Pattern) können auch im Ganzen an beliebige Positionen gesetzt werden.

```
// Neues Bild-Objekt laden
// Anlegen des Image-Objektes
var imgObj = new Image();

// Zuweisung einer URL
imgObj.src = "bild.jpg";
```

```
// Bild an definierte X- und Y-Position setzen
ctx.drawImage(imgObj, 30, 50);

// Bild an definierte Position setzen und auf 100*100px skalieren
ctx.drawImage(imgObj, 30, 50, 100, 100);
```

## Bildausschnitt

Die drawImage-Funktion lässt es auch zu, Bildausschnitte zu nehmen und darzustellen. Dazu benötigt sie die folgenden Parameter:

1. Bild-Objekt
2. X-Koordinate der linken Kante im Original
3. Y-Koordinate der oberen Kante im Original
4. Breite des Ausschnitts
5. Höhe des Ausschnitts
6. X-Koordinate des Canvas, an die der Ausschnitt gezeichnet wird
7. Y-Koordinate des Canvas, an die der Ausschnitt gezeichnet wird
8. Breite, mit der der Ausschnitt gezeichnet wird
9. Höhe, mit der der Ausschnitt gezeichnet wird

```
// Ausschnitt aus einem Bild skalieren und darstellen
var imgObj = new Image();
imgObj.src = "bild.jpg";
ctx.drawImage(imgObj, 50, 0, 20, 30, 200, 300, 40, 60);
```

Mit dieser Technik ist es z.B. auch möglich, aus einer Bilddatei mit vielen Animationsphasen einen Charakter zu animieren, indem nacheinander die richtigen Bildausschnitte dargestellt werden.

## Transparenz

Beim Zeichnen kann auch die Transparenz des jeweils gezeichneten Elements manipuliert werden. Dieser Wert (zwischen 0 und 1) wird beim fill-Befehl bzw. beim drawImage-Befehl verwendet.

```
ctx.beginPath();
ctx.rect(300,300,100,50);
ctx.globalAlpha = 0.5;
ctx.fill();
```

## Leeren

Wir können auch den Canvas oder einen Teil davon leeren. "Leeren" bedeutet in diesem Fall, einen Teil oder den ganzen Canvas mit dessen Hintergrund zu füllen.

```
//Leert den gesamten Canvas, falls der Canvas in der Variablen 'c'
referenziert ist
ctx.clearRect(0, 0, c.width, c.height);
```

## Animieren

Wenn wir nun diese Zeichenaktionen in eine Funktion kapseln und per Intervall die Funktion mit leicht veränderten Parametern immer wieder aufrufen, erhalten wir eine Animation.

**Achtung:** Vor jedem erneuten Zeichnen sollte der Canvas geleert werden, sonst kommt es zu Artefakten.

```
var c;
var ctx;
var meinVerlauf;

var x = 200;
var vx = 5;
var y = 200;
var vy = 5;

function bewege() {
    x+=vx;
    y+=vy;
    ctx.beginPath();
    ctx.clearRect(0, 0, c.width, c.height);
    ctx.arc(x,y,100,0,2*Math.PI);
    ctx.stroke();

    if(x > (c.width - 100)){
        vx *= -1;
    } else if(x < 100){
        vx *= -1;
    }
    if(y > (c.height - 100)){
        vy *= -1;
    } else if(y < 100){
        vy *= -1;
    }
}

window.onload = function() {
    c = document.getElementById('meinCanvas');
    ctx = c.getContext('2d');
    ctx.lineWidth=10;
    ctx.strokeStyle = '#f40';
    setInterval(bewege, 30);
}
```

## Transformation

Natürlich gibt es auch die Möglichkeit, Inhalte zu skalieren, zu verschieben und zu drehen. Diese Technik ist aber auf den ersten Blick etwas verwirrend – und auf den zweiten Blick auch.

Da der Canvas nur mit Pixeln gefüllt ist und nicht mehr auf die gezeichneten Objekte zugreifen kann, wurde das Problem anders gelöst. Auf dem Canvas liegt eine unsichtbare Transformationsmatrix, die

bestimmt, an welcher Position mit welcher Rotation und Skalierung die nächste Zeichnung ausgeführt wird.

**Achtung:** Die Transformation findet immer an der oberen, linken Ecke des Canvas statt. Falls also die Zeichenfläche gedreht oder skaliert wird, so muss ggf. auch die Positionierung mit angepasst werden.

## Verschieben

Die Methode `translate(x,y)` verschiebt die nächste Zeichnung um die angegebene Anzahl an Pixeln.

```
ctx.beginPath();
ctx.rect(100,100,300,200);
ctx.stroke();

ctx.translate(120,100);
ctx.strokeStyle = '#40f';
ctx.beginPath();
ctx.rect(100,100,300,200);
ctx.stroke();
```

## Drehen

Die Methode `rotate(winkel)` dreht die nächste Zeichnung um den angegebenen Winkel in Radians.

Um einen 'normalen' Winkel in die Radians-Angabe umzuwandeln, rechne die Angabe  $\cdot \pi / 180$ .

```
ctx.beginPath();
ctx.rect(100,100,300,200);
ctx.stroke();

ctx.rotate(10*Math.PI/180);
ctx.strokeStyle = '#40f';
ctx.beginPath();
ctx.rect(100,100,300,200);
ctx.stroke();
```

## Skalieren

Die Methode `scale(x,y)` skaliert die Zeichenfläche horizontal und vertikal um die angegebenen Werte.

```
ctx.beginPath();
ctx.rect(100,100,300,200);
ctx.stroke();

ctx.scale(2,2);
ctx.strokeStyle = '#40f';
ctx.beginPath();
ctx.rect(100,100,300,200);
```

```
ctx.stroke();
```

## Zurücksetzen

Die Methode `setTransform(scale X, neigen X, neigen Y, scale Y, bewege X, bewege Y)` setzt die Matrix-Transformation auf die gegebenen Werte. Diese Methode kann also zum Reset der Transformationen eingesetzt werden.

```
// Dieser Code setzt alle Transformationen wieder zurück  
ctx.setTransform(1,0,0,1,0,0);
```

## Pixel für Pixel

Um ein Schlaglicht darauf zu werfen, was mit Canvas noch alles möglich ist, soll hier kurz gezeigt werden, wie einzelne Pixel ausgelesen und manipuliert werden können.

Um an die einzelnen Pixel des Canvas zu kommen, legen wir ein `imageData`-Objekt mit Referenz auf den Canvas an.

```
c = document.getElementsByTagName('canvas')[0];  
ctx = c.getContext('2d');  
var daten = ctx.getImageData(0,0,c.width,c.height);
```

Das `imageData`-Objekt enthält drei Informationen. Dies sind die Breite des Bildbereichs (`width`), die Höhe des Bildbereichs (`height`) und die Pixeldaten (`data`). Die Pixeldaten sind der interessante Teil. Diese sind als Array organisiert, das für jeden Pixel die Farb- und Transparenzwerte in Form von ganzen Zahlen von 0 bis 255 enthält. Das Array ist folgendermaßen aufgebaut:

```
[ px1 rot, px1 grün, px1 blau, px1 alpha, px2 rot, px2 grün, px2 blau, px2  
alpha, px3 rot, px3 grün, px3 blau, px3 alpha, px4 rot, px4 grün, etc. ]
```

Dieses Array und damit die RGBA-Werte jedes Pixels kann nun manipuliert werden. Z.B. so:

```
for ( var i = 0; i < daten.data.length; i += 4 ){  
  var color = Math.round( Math.random() ) * 255;  
  daten.data[i] = color;  
  daten.data[i+1] = color;  
  daten.data[i+2] = color;  
  daten.data[i+3] = 255;  
}
```

Mit der `putImageData`-Methode können dann die manipulierten Pixeldaten in den Canvas geschrieben werden.

```
ctx.putImageData(daten,0,0);
```



### Hinweis:

Zum Thema Canvas kann man ganze Bücher füllen und hat das auch getan. Die hier vorgestellten Techniken sind nur die Basics. Wirklich spannend wird es, wenn es um Pixelmanipulation und andere fortgeschrittene Techniken geht.

Wenn Dich dieses Thema ebenso fasziniert wie mich, dann stöbere in der entsprechenden Literatur weiter.

# Fehlersuche

Wenn ein einfaches Starren auf den Code nicht mehr ausreicht, kann man durch verschiedene Techniken den Code genauer durchsuchen. Die wichtigste Technik ist die Ausgabe von Variablen oder Texten um zu sehen, ob das Programm an einer bestimmten Stelle ankommt und ob eine Variable den gewünschten Wert enthält.

Diese Ausgabe kann auf verschiedene Weisen erfolgen. Die gängigsten sind

## Alert

Benutze ein Alert-Fenster, um eine Ausgabe zu erzeugen. Die Alert-Box unterbricht den Programmablauf, was unter Umständen gewünscht ist.

```
var x = 42;
alert ( 'Die Variable x enthält ' + x);
```

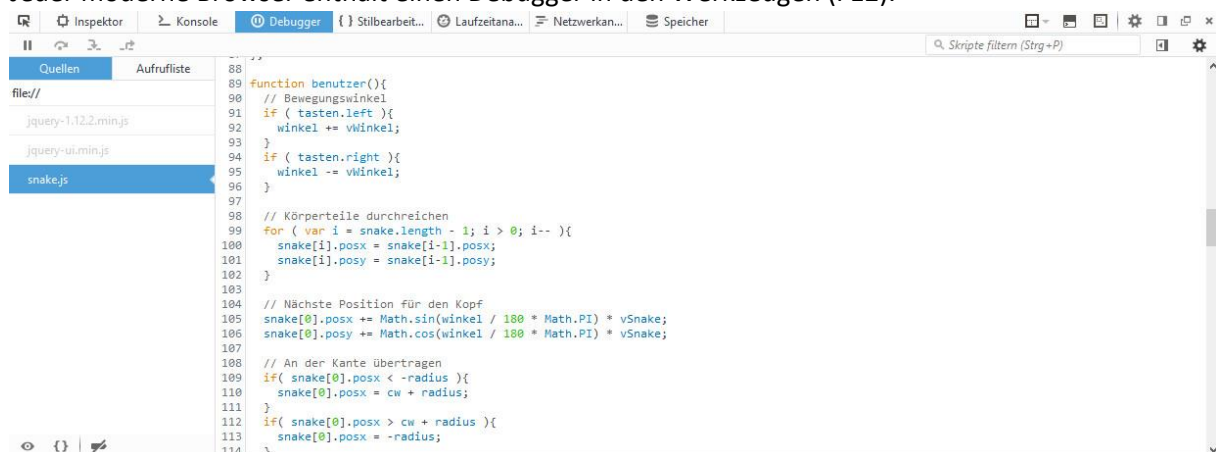
## Konsole

Diese Methode unterbricht den Programmablauf nicht. Stattdessen wird die Meldung in die Konsole des Browsers geschrieben, die in den Werkzeugen (F12) zu finden ist.

```
var x = 42;
console.log ( 'Die Variable x enthält ' + x);
```

## Debugger

Jeder moderne Browser enthält einen Debugger in den Werkzeugen (F12).



Im Debugger wird der Code der Seite angezeigt. Auf der linken Seite finden sich die Dateien, die zu dieser Seite gehören.

Klicken Sie auf eine Zeilennummer, dann wird dort ein Stoppunkt gesetzt. Dieser Stoppunkt sorgt dafür, dass die Programmausführung (ggf. nach dem nächsten Reload) an dieser Stelle gestoppt wird. Wenn der Programmablauf gestoppt ist, können Sie mit der Maus auf einen Variablennamen fahren und bekommen dann den Inhalt der Variablen angezeigt.

Oben links finden Sie Knöpfe, um das Programm weiterlaufen oder schrittweise ausführen zu lassen.

Dieser Debugger ist ein sehr mächtiges Werkzeug, dessen Möglichkeiten wir hier nur angekratzt haben.

## Performance (a.k.a. schnellerer Code)

Im normalen Alltag sind Javascripte von übersichtlicher Länge und innerhalb von Sekundenbruchteilen ausgeführt. Das Einfügen eines neuen Elementes zum Beispiel geht so schnell, dass mit bloßem Auge kein Zeitverlust zu erkennen ist.

Bei umfangreicheren Applikationen allerdings kann bei ungeschickter Programmierung unnötig Zeit verloren gehen. Daher möchte ich hier ein paar Dinge vorstellen, an die man denken sollte, um einen möglichst performanten Code zu erhalten.

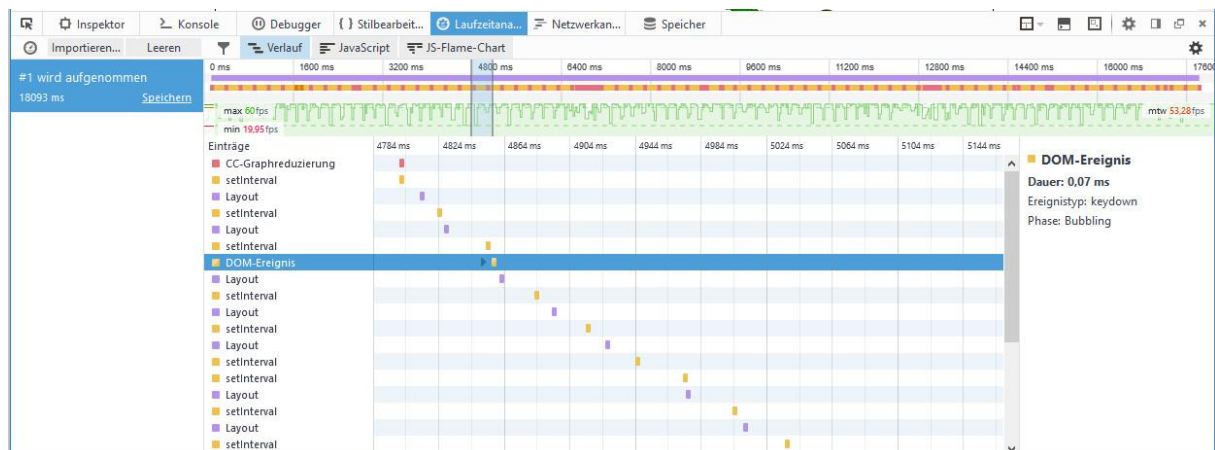
### Performance-Probleme

Wenn das Programm zu langsam läuft, muss das nicht daran liegen, dass Javascript zu langsam ist. Es kann auch sein, dass das Programm durch suboptimalen Aufbau Zeit verschwendet.

Um die Stellen zu finden, an denen das Programm stockt, bieten moderne Browser mächtige Hilfsmittel. Im Rahmen dieser Doku nehme ich auf den Firefox Bezug, andere Browser bieten ähnliche Hilfsmittel.

### Laufzeitanalyse

Mit einem Druck auf F12 öffnen sich die Werkzeuge. Hier findet sich u.a. die Laufzeitanalyse. Wird diese das erste Mal geöffnet, so bietet sich die Möglichkeit, die Aufnahme zu starten.



Sobald Sie die Aufnahme starten, wird jeder Befehl und dessen Laufzeit gemessen.

Wenn Sie dann die Aufnahme wieder beenden, wird das Ergebnis der Messung angezeigt. Besonders interessant ist dann die grüne Kurve. Diese zeigt die Frames pro Sekunde an. Wenn diese Kurve Ausreißer nach unten zeigt, so kann mit der Maus der Bereich markiert werden. Im Bereich darunter werden dann die Befehle angezeigt, die im markierten Zeitraum ausgeführt wurden und wie lange diese gebraucht haben.

Mit dieser Hilfe lassen sich relativ leicht Befehle finden, die immer wieder für Performance-Einbrüche sorgen. Dann gilt es, das Programm so umzuarbeiten, dass die problematischen Befehle nicht mehr oder nur noch möglichst selten aufgerufen werden.

### Zeitmessungen im Code

Um zu messen, wie lange das Programm für einen bestimmten Abschnitt gebraucht hat, bietet sich diese Möglichkeit an. Die Ausgabe erfolgt dann in der Konsole unter den Werkzeugen (F12)

```
console.time('messe');  
  
// Hier passiert irgendwas.  
  
console.timeEnd('messe');
```

### Wahl der Mittel

Es gibt einige Strukturen, die im Ruf stehen, viel Zeit zu verbrauchen. Das Begriff ‚viel Zeit‘ ist natürlich relativ und wird erst relevant, wenn diese Strukturen sehr häufig (Tausende oder Millionen mal) verwendet werden.

- So sind zwar z.B. Funktionsaufrufe an sich ziemlich schnell, Schleifen aber um einiges schneller. Daher kann es in Einzelfällen – wenn eine Funktion viele tausend Mal aufgerufen wird – sinnvoll sein, diese Funktion irgendwie einzusparen.
- Komplexe If - else if - else Strukturen gelten als langsamer als switch case-Anweisungen.

Natürlich gelten diese Aussagen in manchen Browsern mehr als in anderen. Sie teilen sich etwa Chrome und Firefox die Krone des schnellsten Javascript-Interpreters und 'langsame' Codeteile fallen kaum ins Gewicht. Im Internet Explorer / Edge dagegen fällt langsamer Code sehr viel stärker auf.

### DOM-Aufrufe

Ein einzelner Aufruf des DOM ist zwar in wenigen Millisekunden erledigt. Wenn aber viele solche Aufrufe getätigt werden sollen, dann kann der Seitenaufbau doch merklich verlangsamt werden.

Am längsten dauert es, in dem angezeigten Dokument ein Element per `appendChild` einzubinden. Daher empfiehlt es sich, zunächst alle Elemente inklusive ihrer Kind-Elemente und Textinhalte zu generieren und das gesamte Konstrukt in einem Rutsch auf der Seite zu platzieren.

Also statt jedes Element und Kind-Element mit einem eigenen DOM-Aufruf einzubinden, lieber so:

```
// Element anlegen  
var e = document.createElement('div');  
  
// Kindelement anlegen  
var v = document.createElement('p');  
var vt = document.createTextNode('Horst');  
v.appendChild(vt);  
e.appendChild( v );  
  
// Noch ein Kindelement anlegen  
var n = document.createElement('p');  
var nt = document.createTextNode('der Glücksschmetterling');  
n.appendChild(nt);  
e.appendChild( n );
```

```
// Alles in die Seite einbinden
document.getElementById('ausgabe').appendChild(e);
```

Eine alternative Möglichkeit wäre, den Quellcode in einer String-Variable zu erstellen und diese am Ende per innerHTML einzubinden. Auch hier wäre nur ein DOM-Aufruf nötig:

```
// Element anlegen
var e = '<div>';
e += '<p>Horst</p>';
e += '<p>der Glückschmetterling</p>';
e += '</div>';
document.getElementById('ausgabe').innerHTML += e;
```

## Codemenge

Eigentlich logisch, aber manchmal muss man draufgebracht werden.

Je mehr Code ein Programm beinhaltet, desto länger dauert das Laden dieses Codes und manchmal auch seine Ausführung.

Aus diesem Grund gibt es einige Codeminifier, die den Code entschlacken:

- Zeilenumbrüche und unnötige Leerzeilen werden entfernt
- Variablennamen können gekürzt werden
- Schreibweise von Zahlen kann angepasst werden
- Kommentare werden entfernt
- Etc.

Am Ende bleibt ein Code, der für den Menschen schwer zu lesen, dafür aber kurz ist.

Zwar haben diese Aktionen nur sehr beschränkte Auswirkungen auf die Ausführungsgeschwindigkeit, da moderne Browser den Javascript-Code ohne nach dem Laden Kompilieren, damit er schneller ausgeführt wird. Aber zumindest das Laden des Codes geht erheblich schneller, was dem Benutzer durchaus auffallen kann.

## Skripte zusammenfassen

Für die Ausführung des Codes nicht wirklich relevant, beim Laden desselben aber durchaus: Eine Seite, die auf viele externe Dateien zugreifen müssen (Skripte, Stylesheets, Bilder, etc.), müssen jede Datei mit einem eigenen http-Aufruf laden. Das benötigt jedes Mal etwas Zeit und wird außerdem von Google geahndet.

Deswegen ist es besser, Skripte ebenso wie CSS-Dateien zusammenzufassen. Dadurch werden diese Dateien zwar größer, benötigen aber nur eine Kontaktaufnahme zum Server. Was u.a. das Ranking in Suchmaschinen verbessert.

Ein vernünftiges CMS hat diese Funktion bereits an Bord.

## Webworker

Javascript arbeitet grundsätzlich linear. D.h. ein Befehl nach dem anderen wird abgearbeitet.

Abgesehen von Ajax und Intervallen gibt es kein Konzept von paralleler Tätigkeit.

Das kann ein Problem sein, wenn es Codeteile gibt, bei denen von vornherein klar ist, dass die

Ausführung recht lange dauern kann. In der Zeit ist der Browser nicht ansprechbar und vielleicht kommt sogar eine Fehlermeldung, dass ein Skript zu lange braucht. Um damit umzugehen, wurde das Konzept der Webworker erdacht.

Ein Webworker ist ein Stück Code, das parallel zum Rest des Javascript-Codes ausgeführt wird. Es handelt sich hier nicht um einen echten Thread.

Der Code für einen Webworker wird in eine externe Javascript-Datei ausgelagert. Diese Datei wird dann durch das Worker-Objekt geladen.

```
// Anlegen eines Worker-Objektes
var meinWorker = new Worker('worker.js');
```

Der Worker hat grundsätzlich keinen Zugriff auf das DOM. Das wird über den normalen JS-Code angesprochen, der über eine Schnittstelle mit dem Worker kommuniziert.

```
// JS-Code im HTML-Dokument:
var meinWorker = new Worker('worker.js');

// Nachricht an den Worker senden
meinWorker.postMessage('los');

// Wenn eine Nachricht vom Worker empfangen wird.
// Der übergebene Parameter ist ein Objekt, data ist der übergebene Wert
meinWorker.onmessage = function(rueckgabe){
    alert('Ergebnis: ' + rueckgabe.data);
}
```

Sobald der Webworker eine Nachricht (per *postMessage*) vom aufrufenden Code erhält, wird der Event *onmessage* gefeuert. Dieser überprüft den übergebenen String und führt die eine oder andere Aktion aus. Danach sollte ein Rückgabewert an das aufrufende Skript geliefert werden. Erst wenn der Rückgabewert beim aufrufenden Script angekommen ist, wird der dortige *onmessage*-EventListener gefeuert.

```
// JS-Code in der Datei worker.js
self.onmessage = function(uebergabe){
if ( uebergabe.data == 'los' ){
    // Irgendein Code, der wahnsinnig lange braucht
    self.postMessage(Rueckgabewert);
}
}
```

# Bibliotheken / Frameworks

Grundsätzlich wird gern unterschieden zwischen Bibliotheken (Libraries) und Frameworks. Die Terminologie schwankt zwar in Abhängigkeit davon, wen man fragt. Es lässt sich aber grob sagen:

- Eine Bibliothek ist eine Codesammlung, die Methoden, Variablen und Konstanten zur Verfügung stellt, um eine bestimmte Aufgabe zu erfüllen. Z.B. jQuery, Sweetalert, ThreeJS
- Ein Framework ist in der Bedeutung allgemeiner. Es ist mehr eine Umgebung, in der man arbeitet. Dem Entwickler werden Werkzeuge gegeben, z.T. Paradigmen vorgegeben. Z.B. React, Angular

Jeder Entwickler kann sich seine eigene Bibliothek für bestimmte, immer wiederkehrende Probleme erstellen. Dabei handelt es sich schlicht um eine Javascript-Datei, die in das Dokument geladen wird und die die benötigten Methoden bereitstellt.

Code, der auf die Bibliothek zugreift, darf natürlich erst nach der Einbindung der Bibliothek kommen.

```
<script type="text/Javascript" src="meineBibliothek.js"></script>
```

Wenn in der Bibliothek einfach Funktionen zusammengefasst würden, hätte der Benutzer/Entwickler schnell das Problem, dass die Benennung der Funktionen und Variablen in der Bibliothek mit den Namen im Hauptprogramm kollidieren können.

Daher ist der beste Weg, eine Bibliothek aufzubauen, das Anlegen eines Objektes, welches alle Variablen und Funktionen bereitstellt.

Im Folgenden ein Beispiel für eine Bibliothek zum Erzeugen einer Zufallszahl:

```
// Inhalt der Bibliotheks-Datei
var myLib = {
  zufall: function(min, max, dez){
    var z = Math.random() * (max + 1 - min) + min;
    z *= Math.pow(10,dez);
    z = Math.floor( z );
    z /= Math.pow(10,dez);
    return z;
  }
}

// Aufruf der Methode aus dem Hauptprogramm:
alert(myLib.zufall(1,6,0));
```

# jQuery

jQuery ist eine Bibliothek, die sich vor allem der Arbeitserleichterung verschrieben hat. Allerdings bietet sie auch eine Hand voll eigenständiger Funktionen, die ohne jQuery nicht so einfach zur Verfügung stünden.

Der allgemeine Umgang mit jQuery verläuft immer nach demselben Muster: DOM-Elemente werden in jQuery mit dem Dollar-Zeichen angesprochen. In der ersten Klammer erfolgt dann die Angabe per CSS-Selector, welche/s DOM-Element/e angesprochen werden sollen.

Mit einem Punkt getrennt folgt dann die Methode, die auf dieses Element angewendet werden soll.

Die meistbenutzten Methoden sind die folgenden. Eine genauere Auflistung und Syntax-Angaben finden sich hier: <http://api.jquery.com/>

## Inhalt

```
$('element').html();
```

Liest und ändert den Inhalt des Elementes

```
$('element').val();
```

Liest und ändert den Value des Formularelementes

```
$('element').wrap();
```

Umschließt das Element mit einem anderen Element

```
$('element').before();
```

Fügt den Parameter vor alle gefundenen Elemente ein

```
$('element').after();
```

Fügt den Parameter hinter alle gefundenen Elemente ein

```
$('element').prepend();
```

Fügt den Parameter an den Anfang aller gefundenen Elemente ein.

```
$('element').append();
```

Fügt den Parameter ans Ende aller gefundenen Elemente ein

```
$('element').attr();
```

Liest oder setzt das als Parameter übergebene Attribut

```
$('element').clone();
```

Legt eine Kopie des Elements inklusive aller Kind-Elemente an.

```
$('element').data();
```



Liest oder setzt beliebige Variablen zu einem Element

```
$('#element').empty();
```

Leert das Element

```
var neuesElement = $('<div></div>');
```

Legt einen neuen div-Container an. Hat das neue Objekt als Rückgabewert.

## Aussehen

```
$('#element').css({Objekt});
```

Ändert die CSS-Eigenschaften

```
$('#element').animate({Objekt}, Dauer);
```

Animiert CSS-Eigenschaften vom Typ Number, die im Objekt angegeben sind, innerhalb der Dauer. Farben lassen sich mit der Extension jquery-color animieren.

```
$('#element').fadeIn(Dauer);
```

Sichtbarkeit wird auf 1 erhöht

```
$('#element').fadeOut(Dauer);
```

Sichtbarkeit wird auf 0 verringert

```
$('#element').fadeTo(Dauer, Sichtbarkeit);
```

Sichtbarkeit wird auf den gegebenen Wert gebracht.

```
$('#element').fadeToggle(Dauer);
```

Sichtbarkeit wird umgeschaltet

```
$('#element').hide(Dauer);
```

Element wird unsichtbar gemacht

```
$('#element').show(Dauer);
```

Element wird sichtbar gemacht

```
$('#element').toggle(Dauer);
```

Element wird sichtbar oder unsichtbar gemacht

```
$('#element').slideUp(Dauer);
```

Element wird nach oben zusammengeschoben

```
$('#element').slideDown(Dauer);
```

Element wird nach unten auseinandergezogen

```
$('element').slideToggle(Dauer);
```

Element wird zusammengeschoben oder auseinandergezogen

```
$('element').addClass('klassenname');
```

Klasse wird hinzugefügt

```
$('element').removeClass(Dauer);
```

Klasse wird entfernt

```
$('element').toggleClass(Dauer);
```

Klasse wird hinzugefügt oder entfernt

## Navigation

Mit den unten stehenden Anweisungen können andere Elemente, ausgehend von der Angabe in `$ ( )` angesprochen werden.

```
$('element').children('kindelement')
```

Spricht alle direkten Kinder des Elementes an, auf das der Bezeichner zutrifft.

```
$('element').find('kindelement')
```

Spricht alle Nachfahren des Elementes an, auf das der Bezeichner zutrifft.

```
$('element').parent()
```

Spricht das Eltern-Element an.

```
$('element').siblings()
```

Spricht das Geschwister-Elemente an.

## Eventhandler

Auch die Zuweisung von Eventlistenern ist als Funktion umgesetzt. Diese Funktionen bekommen anonyme Funktionen als Parameter übergeben, die bei Auftreten des Events ausgeführt wird.

```
$('element').click(funktion);
```

Wird bei einem Klick auf das Element ausgeführt.

```
$('element').dblclick(Funktion);
```

Wird bei einem Doppelklick ausgeführt.

```
$('element').keyPress(Funktion);
```

Wird ausgeführt, wenn eine Taste gedrückt wird

```
$('#element').change(Funktion);
```

Wird bei einer inhaltlichen Änderung ausgeführt.

```
$('#element').error(Funktion);
```

Wird bei einem Fehler ausgeführt.

```
$('#element').hover(Funktion, Funktion);
```

Wird bei Mouseenter und Mouseleave ausgeführt.

```
$('#element').mouseenter(Funktion);
```

Wird bei Mouseenter ausgeführt.

```
$('#element').mouseleave(Funktion);
```

Wird bei Mouseleave ausgeführt.

Der input-Event wurde (noch) nicht implementiert. Soll dieser verwendet werden, so greife auf die lange Schreibweise zurück:

```
$('#element').on('input', Funktion);
```

## Codebeeinflussung

```
$('#element').delay(Dauer).andereAktion();
```

Wartet die Dauer ab, bevor die Aktionen hinter dem Delay ausgeführt werden. Funktioniert nur, wenn die zu verzögernde Funktion auf Zeit basiert (z.B. animate, etc.)

```
$('#element').each(function(index){});
```

Führt für jedes gefundene Element die Funktion aus. Innerhalb der Funktion stehen sowohl this als auch der Index zur Verfügung

## Variablen

```
$('#element').length;
```

Gibt die Anzahl der gefundenen Elemente zurück

## Überprüfungen / Filter

```
$('#element').hasClass('Klasse').Aktion
```

Filtert jene Objekte heraus, die die angegebene Klasse haben

## jQuery und AJAX

Mit Hilfe von jQuery können wir sogar komplette AJAX-Abfragen erledigen.

Mehr über Ajax: [Strg-Klick hier](#)

Mehr Möglichkeiten mit AJAX/jQuery: <http://api.jquery.com/jquery.ajax/>

Ein einfaches Beispiel für eine Ajax-Anwendung mit jQuery sieht wie folgt aus:

```
$.ajax('../daten.txt', {
  success: function ( data ){
    alert ( data );
  },
  error: function(){
    alert ( 'Problem' );
  }
});
```

Ein etwas anspruchsvolleres Beispiel für eine Ajax-Anfrage mit Parameterübergabe per POST:

```
$.ajax ({
  url: '../post.php',

  success: function( data ){
    $('body').append ( data );
  },

  error: function(){
    alert ( 'Fehler' );
  },

  method: 'POST',

  data: { p1: 42, p2: 23 }

});
```

# AngularJS

Die Idee hinter angularJS ist, HTML um neue Möglichkeiten zu erweitern. Wenn Du nur die Standard-Funktionalitäten von AngularJS nutzt, dann wirst Du keine Zeile Javascript-Code mehr schreiben müssen.

Im Umfeld von AngularJS werden häufig Begriffe wie Expressions, Direktiven oder Module verwendet. Hier gilt, wie immer beim Programmieren: Lass Dich nicht beeindrucken.

AngularJS lässt sich erst wirklich einsetzen, wenn man die Logik durchdrungen hat.

Diese Doku bietet bestenfalls eine Referenz, aber keine Anleitung. Um AngularJS wirklich zu erlernen, solltest Du Dir etwas Zeit geben und erstmal diese Video-Reihe:

<http://campus.codeschool.com/courses/shaping-up-with-angular-js>

Dieser Kurs kann natürlich nur einen kleinen Teil des Themas abdecken. Eine komplette Referenz gibt es auf der Projekt-Webseite: <https://docs.angularjs.org/api>

## App-Block

Um AngularJS zu verwenden, musst Du zunächst einen Teil Deines HTML-Codes für AngularJS freigeben. Das kann der gesamte HTML-Code sein, oder auch ein HTML-Element innerhalb der Webseite.

```
<body ng-app>
```

Oder

```
<div ng-app>
...
</div>
```

Damit ist definiert, in welchem Teil der Webseite AngularJS funktionieren soll.

## Ausdrücke (Expressions)

Anweisungen (z.B. Berechnungen) innerhalb des HTML-Codes lassen sich mit sog. **Expressions** ausführen und ausgeben. Eine Expression wird in einer doppelten geschweiften Klammer geschrieben.

Eine Expression wäre:

```
<p>
  {{ 1 + 1 }}
</p>
```

Die Expressions können auf die vorhandenen Variablen zugreifen.

```
<p>
  {{ 'Meine Lieblingsserie: ' + serien[1] }}
</p>
```

## Variablen

können im HTML-Tag mit der **ng-init**-Direktive angelegt werden. Die Variablen werden in demselben HTML-Tag angelegt, in dem auch die App (bzw. das Modul) definiert wurde.

```
<div ng-app ng-init="serien=['Tatort','Derrick','Urmel']; zahl=42;
text='Hallo'">
```

Die ng-init-Direktive kann auch direkt nach dem Ladevorgang eine Funktion aufrufen.

```
<body ng-app="laden" ng-controller="ladenCtrl" ng-init="meineFunktion();">
```

Die ng-init-Direktive wird nur selten verwendet. Normalerweise greift man eher auf Module und Controller zurück (s.u.).

## Anweisungen (Direktiven)

Die wirkliche Power von AngularJS sehen wir aber erst mit den **Direktiven**.

Diese werden einem HTML-Element als Attribut zugewiesen und sorgen dafür, dass dieses sich auf eine definierte Weise verhält.

Eine Zusammenfassung der wichtigsten Direktiven:

### ng-model

Stellt den Inhalt eines Eingabeelements (input, select, textarea) als Variable innerhalb von AngularJS bereit. Diese Variable lässt sich mit {{meinName}} ausgeben.

Der Inhalt der Variable und der Expression werden laufend aktuell gehalten.

```
<input ng-model="meineVariable">
```

### ng-show

Dieses Element wird nur dargestellt, wenn die genannte Variable einen Inhalt hat.

Es lassen sich auch mehrere Angaben logisch verknüpfen.

```
<div ng-show="meineVariable">Hallo Welt</div>
```

### ng-if

Das Dom-Element wird nur angezeigt, wenn die Bedingung ein true liefert.

Im Gegensatz zu ng-show bzw. ng-hide wird hier das Element komplett aus dem DOM entfernt bzw. neu angelegt.

Es lassen sich auch mehrere Angaben logisch verknüpfen.

```
<input type="checkbox" ng-model="marker1" />
<input type="checkbox" ng-model="marker2" />
<br>
<span ng-if="marker1 && marker2">Hallo Welt</span>
```

### ng-repeat

Eine **Schleife** (genauer: eine for...in-Schleife).

Das Element, in dem sich das ng-repeat-Attribut befindet, wird so oft wiederholt, wie Elemente in dem Array liegen. In jedem Schleifendurchlauf steht in einer Expression der Wert aus dem Array zur Verfügung.

```
<div ng-repeat="schlonz in ['Harry','Derrick','Urmel']">
  {{schlonz}}
</div>
```

**Achtung:** Falls das Array **Duplikate** enthält, bricht das Programm ab, weil dann die Speicherplätze nicht mehr eindeutig referenziert werden können. Eine Lösung ist die Verwendung der *track by*-Direktive. Sie sorgt dafür, dass die Arrayfelder nach einer anderen Eigenschaft als dem Inhalt referenziert werden. Meist wird dafür der Index genommen.

```
<div ng-repeat="schlonz in ['Harry','Derrick','Urmel'] track by $index">
  {{schlonz}}
</div>
```

## Filter

Um eine Ausgabe anzuhübschen oder bestimmte Ausgaben zu unterdrücken, bietet AngularJS eine große Auswahl an sog. Filtern. Diese werden mit einem Pipe-Symbol hinter die gewünschte Ausgabe gehängt.

```
<p ng-repeat="schlonz in ['Harry','Derrick','Urmel'] | filter:suche |
  limitTo:2 ">
  {{schlonz | limitTo:2 | uppercase }}
</p>
```

Ein paar mögliche Filter sind:

filter:expression - Ausgabe wird nach der Expression durchsucht  
currency - der (Zahlen-)Wert wird in eine Währung umgerechnet.  
Durch currency: '€ ' kommt das Ergebnis als Euro  
date - die auszugebende Zahl wird als Datum angezeigt (akzeptiert Optionen)  
uppercase / lowercase - Groß- oder Kleinschrift  
limitTo - Nur eine bestimmte Anzahl an Ausgaben machen.

Je nachdem, in welcher Reihenfolge und an welcher Stelle die Filter stehen, haben sie unterschiedliche Auswirkungen.

## ngStyle

Erlaubt es, CSS-Style-Angaben zu machen.

Die CSS-Angaben müssen dabei in Form eines Objektes erfolgen.

```
<div ng-style="{ 'background-color': '#0f0', color: '#a4a' }">Hallo Welt</div>
```

# Module

Eine umfangreichere App kann aus mehreren Bereichen bestehen.

Jeder dieser Bereiche wird *Modul* genannt. Auf einer HTML-Seite können ohne weiteres mehrere Module mit unterschiedlichen Zielsetzungen eingebunden werden.

Auch wenn mit Daten gearbeitet werden soll, empfiehlt sich das Anlegen eines Moduls.

Zur Implementierung eines Moduls wird dieses zunächst im JS-Code angelegt:

```
<script type="text/Javascript">
  // Dies ist das Modul namens autos, das in der Variable
  // autoApp zugänglich gemacht wird.
  var autoApp = angular.module('autos', []);
</script>
```

Und dann im ng-app-Attribut des HTML-Tags genannt:

```
<div ng-app="autos"> ... </div>
```

## Datenübertragung zwischen HTML und JS

Für die Datenübertragung sind die sog. *Controller* zuständig. Sie heißen so, weil sie die Daten *kontrollieren*. Ein Controller gehört immer zu einem Modul.

Ein Controller ist ein Javascript-Objekt und wird folgendermaßen angelegt:

```
<script type="text/Javascript">
  var appAutos = angular.module('autos', []);

  appAutos.controller('meinCtrl', function($scope) {
    $scope.marke = 'Fiat';
    $scope.modell = 'Stilo';
    $scope.baujahr = '2003';
  });
</script>
```

**\$scope** ist das Bindeglied zwischen der Ansicht (HTML) und der Funktion (Javascript). Es wird als Parameter an den Konstruktor übergeben und enthält alle Werte, auf die im HTML innerhalb des Moduls (des Programmbereichs) zugegriffen werden kann.

Dann muss dieser Controller noch im HTML-Teil deklariert werden, damit die App darauf zugreifen kann:

```
<div ng-app="autos" ng-controller="meinCtrl"> ... </div>
```

Um eine **Methode** im Controller anzulegen und sie anzusprechen, lege sie wie in jedem anderen Konstruktor an:

```
<script type="text/Javascript">
  var meinModul = angular.module('serien', []);
```



```

    meinModul.controller('meinCtrl', function($scope){
        $scope.tv = [
            'Game of Thrones',
            'Dr. Who',
            'Firefly',
            'Blacklist'
        ];
        $scope.addSerie = function() {
            // eingabe ist ein ng-model in einem input
            $scope.tv.push(this.eingabe);
        }
        $scope.removeSerie = function(index){
            $scope.tv.splice(index, 1);
        }
    });
</script>

<body>

    <form ng-submit="addSerie()">
        <input type="text" ng-model="eingabe">
        <button type="submit">Klick</button>
    </form>
    <br>

    <!--
in der nächsten Zeile ein HTML-Tag verwenden, das eine Verschachtelung
zulässt. Das p-Tag ist keine gute Wahl
-->

    <div ng-repeat="serie in tv">
        {{ serie }}
        <form ng-submit="removeSerie($index)">
            <button type="submit">X</button>
        </form>
    </div>

</body>

```

## Eventlistener

Auch in AngularJS gibt es Eventlistener.

Diese sind darauf zugeschnitten, die Methoden des Moduls zu starten.

```

// Ruft die Methode addSerie auf, wenn der Button geklickt wurde
<button ng-click="addSerie();">Serie hinzufügen</button>

```

Weitere Eventlistener, die auf dieselbe Weise verwendet werden, sind:

- ng-click – Reagiert auf das Anklicken des Elementes
- ng-change - Reagiert sofort auf eine Änderung des Inhaltes des Elements
- ng-dblclick - Doppelklick

- `ng-mousedown` - Wenn die Maustaste heruntergedrückt wird
- `ng-mouseup` - Wenn die Maustaste losgelassen wird
- `ng-mouseover` - Wenn sich die Maus auf das Objekt oder ein Kindelement bewegt
- `ng-mouseenter` - Wenn sich die Maus auf das Objekt bewegt
- `ng-mouseleave` - Wenn die Maus das Objekt verlässt
- `ng-mousemove` - Wenn die Maus auf dem Objekt bewegt wird
- `ng-focus` - Wenn das Objekt den Fokus bekommt
- `ng-blur` - Wenn das Objekt den Fokus verliert
- `ng-enter` – Wird gefeuert, wenn die Enter-Taste gedrückt wird. Dazu muss das Eingabefeld in einem *Form*-Tag stehen

## Formulare

AngularJS vereinfacht uns auch den Umgang mit und die Validierung von Formularen.

Um in AngularJS mit einem Formular zu arbeiten, muss dieses das *name*-Attribut tragen. Unter diesem Namen wird im \$scope automatisch ein Objekt angelegt, das einige Eigenschaften des Formulars bereitstellt:

- `$error` - Ggf. aufgetretene Fehlermeldungen (Objekt)
- `$name` - Name des Formulars
- `$dirty` - Ob im Formular seit dem letzten Laden etwas geändert wurde
- `$pristine` - Ob im Formular seit dem letzten Laden nichts geändert wurde (Gegenteil von `dirty`)
- `$valid` - Ob alle Eingaben den Vorgaben entsprechen
- `$invalid` - Ob nicht alle Eingaben den Vorgaben entsprechen (Gegenteil von `valid`)
- `$submitted` - Ob das Formular abgeschickt wurde. Ach, wenn es invalid ist.

### Validierung des Formulars

Um nun ein Formular zu validieren, müssen Vorgaben gemacht werden, die zu erfüllen sind. Felder, die ihre Vorgabe nicht erfüllen, gelten als invalid und bekommen die CSS-Klasse *ng-invalid*.

Jedes Formularelement muss mit einem *ng-model* bestückt sein, damit AngularJS es registriert.

Mögliche Vorgaben sind:

**`<input ng-model="input1" type="email">`**

Ziemlich unvollständige Prüfung, ob die Eingabe eine E-Mail-Adresse ist

**`<input ng-model="input2" type="number" ng-min="10">`**

Prüft, ob eine eingegebene Zahl mindestens dem Wert entspricht

**`<input ng-model="input3" type="number" ng-max="10">`**

Prüft, ob eine eingegebene Zahl maximal dem Wert entspricht

**`<input ng-model="input4" ng-maxlength="10">`**

Prüft, ob ein eingegebener String maximal die angegebene Anzahl Zeichen hat.

**<input ng-model="input5" ng-minlength="10">**

Prüft, ob ein eingegebener String mindestens die angegebene Anzahl Zeichen hat.

**<input ng-model="input6" type="number">**

Prüft, ob der eingegebene Wert eine Zahl ist.

**<input ng-model="input7" required>**

Prüft, ob überhaupt eine Eingabe erfolgt ist.

**<input ng-model="input8" type="url">**

Oberflächliche Prüfung, ob es sich bei einer Eingabe um eine URL handelt.

**<input ng-model="input9" ng-pattern="\d+">**

Ermöglicht die Eingabe einer Regular Expression, der die Eingabe entsprechen muss.

## CSS

Jedes Formularfeld, das seiner Vorgabe entspricht bzw. nicht entspricht, bekommt sowohl die CSS-Klasse `ng-valid/ng-invalid` als auch eine genauer spezifizierte CSS-Klasse bspw. `ng-valid-required/ng-invalid-required`.

Dazu kommen noch CSS-Klassen, die besagen, ob das Formularfeld verändert wurde (`ng-pristine/ng-dirty`) und ob es leer ist (`ng-empty/ng-not-empty`).

Außerdem bekommt das Formularelement eine Zusammenfassung aller seiner Kindelemente. Das Formularelement "weiß" also, ob eines seiner Pflichtfelder vielleicht nicht ausgefüllt ist oder ob ein E-Mail-Feld etwas anderes enthält als es darf.

Klassen, die im CSS-Code vorbereitet wurden, werden so angewandt und stellen schnell mögliche Fehler dar:

```
input.ng-invalid {
  outline: 2px dashed hsla(0, 100%, 50%, .5);
}
input.ng-empty {
  border: 1px solid #000;
}
input.ng-not-empty {
  border: 1px solid #0f0;
}
```

## Submit

Zum Absenden der Daten haben wir zwei Möglichkeiten:

Wir können einem Button angularJS's `click`-EventListener geben, der bei Klick eine Funktion ausführt:

```
<button ng-click="ausfuehren()">
  ausführen
</button>
```

Oder wir geben dem Formular einen *submit*-eventlistener, der nur auf den Button mit dem *type="submit"* reagiert:

```
<form name="angForm" ng-submit="ausfuehren()" novalidate>
  <!--Formularfelder -->
  <button type="submit">
    ausführen
  </button>
</form>
```

Schließlich bekommt der Form-Tag noch das *novalidate*-Attribute, das verhindert, dass der Browser seine eingebaute Validierung durchläuft. So stellen wir sicher, dass nur der von uns geschriebene Code ausgeführt wird.

Dieser kann dann auf Basis des eingangs erwähnten Objekts im Argument *error* auf die Einträge reagieren.

```
let ang = angular.module ( 'ang', [] );
ang.controller ( 'angCtrl', $scope => {

  $scope.ausfuehren = () => {

    if ( $scope.angForm.$error.email ){
      console.log ( 'E-Mail-Adresse ist falsch.' );
      console.log ( $scope.angForm.$error.email[0].$viewValue + ' ?' );
      break;
    }

    // Irgendwas mit den Daten anstellen.
  };

});
```

## AngularJS und Ajax

AngularJS bringt seine eigenen Methoden mit, um Daten von externen Quellen zu laden.

Um eine externe Ressource zu laden, muss dem Controller neben *\$scope* auch *\$http* als Parameter mitgegeben werden.

```
appExtern= angular.module ( 'extern', [] );
appExtern.controller('externCtrl', function($scope, $http){
```

Dann wird die *get*-Methode verwendet, um die Datei zu laden:

```
$http.get("pfadZurExternenDatei")
```

Und mit der Methode then, die zur Methode get gehört, wird schließlich die Aktion gestartet, die im Erfolgsfall ausgeführt werden soll. **Beachte**, dass das then direkt hinter die get-Methode geschrieben wird:

```
$http.get("pfadZurExternenDatei").then(function(response) {  
    $scope.variablenName = response.data;  
});
```

**Übrigens:** Sollten die geladenen Daten JSON-kompatibel sein, dann werden sie automatisch in Arrays/Objekte umgewandelt.

Im Ganzen sieht das Modul dann so aus:

```
var appExtern = angular.module('appExtern', []);  
appExtern.controller('externCtrl', function($scope, $http){  
    $http.get("pfadZurExternenDatei").then(  
        function(response) {  
            // Erfolg  
            $scope.variablenName = response.data;  
        }, function(){  
            // Fehler  
            alert ( "Problem" );  
        });  
    $scope.andereVariable = 42;  
    $scope.nochEineVariable = 'Hallo Welt';  
});
```

## Warum nicht Angular (v. 4)?

Als Nachfolger von AngularJS wird aktuell Angular vorangetrieben. AngularJS ist Version 1, Angular ist mittlerweile Version 4. Diese Version unterscheidet sich erheblich von Version 1.

Dass in diesem Lehrgang bis auf Weiteres AngularJS (v1) gelehrt wird, hat die folgenden Gründe:

1. Auf dem Stellenmarkt werden auf absehbare Zeit Skills in AngularJS (v1) erheblich häufiger abgefragt als Skills in Angular (v2). Da Angular nicht einfach eine Erweiterung von AngularJS ist, sondern vollkommen andere Wege geht, ist es sinnvoller, die auf dem Markt etablierte Methode zu erlernen.
2. Angular (v2) entfernt sich von den bekannten Methoden in Javascript und hilft nicht dabei, Javascript als Sprache besser zu verstehen. Es müssten vollkommen andere Programmierparadigmen erlernt werden.
3. Angular (v2) setzt voraus, dass das Projekt in einer komplexen Verzeichnisstruktur aufgebaut wird, welche Templates, Module, etc. enthalten. Das macht die Lernkurve – insbesondere für Javascript-Anfänger – viel zu steil.
4. Mit Abstand die meisten Anleitungen und Tutorials (v.a. die offiziellen) werden in Typescript – einer Sprache, die zu Javascript kompiliert werden kann – geschrieben. Das erschwert ein eigenständiges Weiterlernen erheblich.

# NodeJS

NodeJS ist ein JS-Framework, das es uns erlaubt, einen Server zu erstellen.

Bitte beachten: NodeJS läuft genau betrachtet nicht auf einem Server, denn es setzt keinen Webserver wie Apache oder IIS voraus. Vielmehr können wir NodeJS auf einem beliebigen Rechner installieren und diesen mit Hilfe von NodeJS zu einem Server machen.

Wenn dieser Rechner direkt mit dem Internet verbunden ist, haben wir einen vollwertigen Webserver erstellt.

Der große Vorteil ist, dass wir auf diesem Server direkt mit Javascript arbeiten können und auf zusätzliche Sprachen wie **PHP, Ruby o.ä. komplett verzichten** können. Wir beherrschen also nun mit unseren JS-Skills sowohl die Client- als auch die Serverseite. Wir können auf Serverseite Dateien lesen und schreiben, können auf verschiedene Arten von Datenbanken zugreifen, können mittels Websockets Browser miteinander kommunizieren lassen u.v.m.

## Installation und erste Schritte

Um NodeJS zu installieren, muss es zunächst von hier: <https://nodejs.org> heruntergeladen werden. Der Installationsprozess läuft ab wie jedes andere Programm auch.

Während der Installation wird automatisch die V8-Engine installiert. Das ist die in Chrome eingebaute Javascript-Engine, die in diesem Fall alleinsteht.

Von nun an müssen wir unterscheiden zwischen JS, das auf dem Server läuft und JS, das im Client ausgeführt wird.

- JS auf dem Server werden wir mithilfe des Terminals (bei Windows die Eingabeaufforderung) ausführen.
- JS im Client wird wie gewohnt vom Server an den Webseitenbesucher ausgeliefert und im Browser ausgeführt.

Wir legen uns zunächst an einem beliebigen Ort unseres Dateisystems einen Ordner an, in dem wir von nun an unsere Dateien halten. Dieser Ordner ist das, was bei einem Webserver *DocumentRoot* genannt wird.

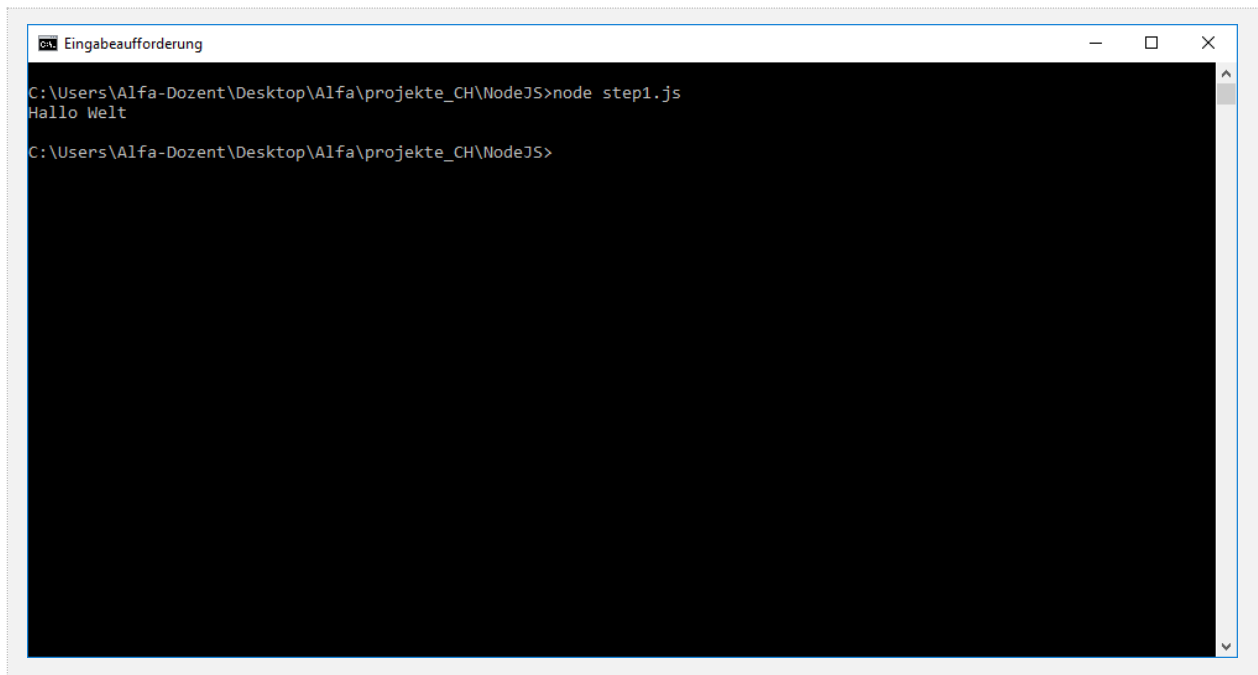
In diesem Ordner legen wir eine Datei namens `step1.js` an und schreiben in diese:

```
console.log ('Hallo Welt');
```

Nun öffnen wir das Terminal, wechseln in den Ordner und rufen die Datei folgendermaßen auf:

```
node step1.js
```

Das Ergebnis sollte so aussehen:



```
Eingabeaufforderung
C:\Users\Alfa-Dozent\Desktop\Alfa\projekte_CH\NodeJS>node step1.js
Hallo Welt
C:\Users\Alfa-Dozent\Desktop\Alfa\projekte_CH\NodeJS>
```

Glückwunsch, Du hast soeben erfolgreich Dein erstes Javascript-Programm auf dem Server erstellt und ausgeführt. Da wir ohne Browser arbeiten, werden console.log-Ausgabe im Terminal angezeigt.

Das Script kann nun natürlich beliebig kompliziert werden, der Aufruf geschieht immer auf dieselbe Weise.

```
let x = [];
while ( x.length < 500 ) {
    let z = Math.floor ( Math.random() * 10000 );
    x.push ( z );
};

x.forEach ( (element, index) => {
    console.log ( index + ': ' + element );
});
```

## Module

### Integriert

NodeJS hat einige integrierte Module, z.B. zur Kommunikation über http oder https oder auch, um per SSL verschlüsselte Daten zu handhaben und viele andere. Eine Liste gibt es hier:

[https://www.w3schools.com/nodejs/ref\\_modules.asp](https://www.w3schools.com/nodejs/ref_modules.asp)

Um ein solches Modul in das eigene Programm zu integrieren, muss es mithilfe des require-Befehls eingebunden werden.

```
var dns = require('dns');
var w3 = dns.lookup('alfatraining.de', function (err, addresses, family) {
    console.log(addresses);
});
```

```
}); // => gibt die IP der Alfa-Training-Webseite aus
```

## Nachladbar ( NPM )

Neben den integrierten Modulen gibt es eine wahre Flut an nachladbaren Modulen. Diese sind auf <https://www.npmjs.com/> zu finden. Zurzeit (Stand Februar 2018) sind dort fast 650.000 verschiedene Module registriert.

Um in diesem Wust an Modulen das richtige zu finden, muss man entweder wissen, wie das gesuchte Modul heißt. Oder sich vertrauensvoll an die Suche auf der Webseite wenden.

Als Beispiel wollen wir das Modul *Chalk* laden, das es uns erlaubt, im Terminal farbig zu schreiben.

Bevor wir das Modul installieren, müssen wir uns überlegen, wie es installiert werden soll:

- **Lokal:** Im aktuellen Ordner wird geschaut, ob sich hier ein Unterordner *node\_modules* befindet. Wenn nicht, werden schrittweise alle übergeordneten Ordner nach dem Unterordner *node\_modules* durchsucht. Wird irgendwo einer gefunden, wird dieser benutzt. Wenn nicht, wird im aktuellen Ordner ein neuer *node\_modules*-Ordner angelegt. Das Modul ist somit für dieses Projekt erreichbar und kann per *require* eingebunden werden.
- **Global:** In einem vordefinierten Ordner wird das Modul installiert. Dieser Weg ist dafür da, um Module als CLI-Befehle zu verwenden (z.B. Grunt o.ä.)

Zum Installieren des Moduls wechseln wir in der Konsole in den Projektordner.

Hier geben wir ein:

```
npm install chalk
```

Damit wird das Modul installiert und in NodeJS registriert.

Von nun an können wir dieses Modul verwenden.

```
const chalk = require ( 'chalk' );  
console.log ( chalk.red('Hallo') + ' ' + chalk.green('Welt'));
```

## Einige NPM-Befehle

Befehl	Beschreibung	Parameter
npm install Modulname	Installiert das Modul	-g global
npm uninstall Modulname	Deinstalliert das Modul	-g global
npm ls	Listet alle installierten Module und den Installationsordner auf	-g global
npm root	Zeigt den Installationsordner für Module an	-g global
npm config ls -l	Konfiguration anzeigen	
npm search Modulname	Modul suchen	
npm docs Modulname	Zeigt die Doku des Moduls im Webbrowser an (falls vorhanden)	



<code>npm update Modulname</code>	Aktualisiert ein bestehendes Modul	<code>-g global</code>
-----------------------------------	------------------------------------	------------------------

## Modul-Probleme

Wenn keine Module geladen werden können, dann ist mit hoher Wahrscheinlichkeit der Proxy nicht richtig gesetzt. Bei Alfa-Training helfen die folgenden Befehle:

```
npm config set proxy http://alfaproxy:3128
npm config set http-proxy http://alfaproxy:3128
npm config set https-proxy http://alfaproxy:3128
```

**Mehr zur Konfiguration gibt es hier:**

<https://docs.npmjs.com/misc/config>

**Mehr Info über die Ordnerstruktur**

<https://docs.npmjs.com/files/folders>

## Eigene Module

Wir können unseren eigenen Code auch in Modulen organisieren. Das ist besonders dann hilfreich, wenn Code in anderen Projekten wiederverwertet werden soll.

Ein zentraler Bestandteil eines Moduls ist das `exports`-Objekt. Es enthält alle Variablen, Funktionen und Klassen, die vom Modul an das Projekt übergeben werden sollen.

Der Code des einfachsten denkbaren Moduls könnte z.B. so aussehen:

```
exports.x = 42;
```

Dieses Modul können wir als `beispiel.js` in unserem Projektverzeichnis speichern oder im Unterordner `node_modules` im `DocumentRoot`. Um das Modul nun in unser Projekt zu integrieren, fügen wir den `require`-Befehl aus.

Wenn wir die Datei im `node_modules`-Verzeichnis liegen haben, brauchen wir nur den Namen der Datei schreiben. Die Endung `.js` kann dabei weggelassen werden.

Liegt die Datei irgendwo anders (z.B. im selben Verzeichnis wie unser aktuelles Projekt), muss der relative Dateipfad angegeben werden (z.B. `./beispiel`). Auch hier kann die Dateierweiterung weggelassen werden.

```
var md = require('beispiel');
console.log ( md.x );           // => 42
```

Im Folgenden ein Beispiel für ein Modul, das für einen Kreis mit einem gegebenen Radius den Umfang und den Flächeninhalt berechnen soll.

```
'use strict';
class kreis {
  constructor ( r ){
    this.radius = r;
  }
  umfang(){
    return this.radius * 2 * Math.PI;
  }
  flaeche(){
    return (this.radius ** 2 ) * Math.PI
  }
}
exports.kreis = kreis;
```

Das Modul gibt mit dem exports-Objekt nur die Klasse zurück, mit deren Hilfe ein Objekt erzeugt wird, das dann die Berechnungen für uns ausführen kann.

```
var c = require('./kreis');

let kreis = new c.kreis(42); // Hier wird das Objekt angelegt und
// der Radius festgelegt

console.log ( kreis.radius );      // => 42
console.log ( kreis.umfang() );    // => 263.89378290154264
console.log ( kreis.flaeche() );   // => 5541.769440932395
```

## Webserver erstellen

Einer der wichtigsten Einsatzzwecke, wenn nicht der wichtigste, ist der als Webserver.

Mit Hilfe von NodeJS lässt sich ein kompletter Webserver auf die Beine stellen, der sich hinter einem Apache nicht zu verstecken braucht.

Im Gegenteil, die Flexibilität und der eventgesteuerte Aufbau eines NodeJS-Servers machen ihn häufig zur besseren Wahl.

Einen Server aufzusetzen ist recht einfach, wenn man die notwendigen Schritte kennt.

Das Script wird erstellt und einfach über das Terminal gestartet. Dann wartet das Script auf Anfragen über den definierten Port. Das heißt, das Programm läuft weiter, bis es mit *Strg-C* abgebrochen wird.

**Achtung:** Ausgaben, die mit `response.write()` erzeugt werden sollen, müssen als String übergeben werden. Falls der erste Parameter kein String ist, muss er umgewandelt werden.

```
// Einbinden des HTTP-Moduls
const http = require ( 'http' );

// Server anlegen,
// request ist die Anfrage des Clients
// response ist die Antwort des Servers (die wir hier erzeugen)
http.createServer ( function ( request, response){
```

```
// Header definieren, dass HTML-Code erzeugt wird
response.writeHead ( 200, {'Content-Type': 'text/html' } ) ;

// Inhalt der Antwort definieren
response.write ( '<p>Hallo Welt</p>' );

// Antwort ist hier zu Ende
response.end();

}).listen(8080); // Port definieren, auf den NodeJS reagieren soll

// Meldung auf dem Terminal
console.log ( 'Server läuft ...' );
```

Wenn nun im Browser *localhost:8080* aufgerufen wird, dann wird ein P-Tag mit 'Hallo Welt' angezeigt.

## Request

Um Parameter aus einer Anfrage des Clients bearbeiten zu können, muss das Modul *url* geladen werden.

Danach können übergebene Parameter als Attribute eines Objektes gelesen werden.

```
const http = require ( 'http' );
const url = require ( 'url' );

http.createServer ( function ( request, response ){

    response.writeHead ( 200, {'Content-Type': 'text/html' } ) ;

    let query = url.parse ( request.url, true ).query;
    response.write ( '' + (query.p1 * query.p2) );

    response.end();

}).listen(8080);

console.log ( 'Server läuft ...' );
```

## Webserver mit Express-Modul

Im NPM existiert ein äußerst beliebtes Modul, das das Anlegen und Verwalten eines Webserver vereinfacht: Express.

Zunächst muss Express im Terminal installiert werden:

```
npm install express
```

Dann wird Express mit dem *request*-Befehl in das Script eingebunden und der Server aufgebaut. Das folgende Script startet den Server und definiert das Verzeichnis *public* als *documentRoot*:

```
'use strict';
```

```
let express = require ( 'express' );
let server = express();

server.use ( express.static ( 'public' ) );
server.listen ( 8080, 'localhost' );
```

Eine Stärke von Express ist, dass ohne großen Aufwand Pfade definiert werden können, die andere Daten liefern als einfache Webseiten. Z.B. dynamische Daten wie die Serverzeit.

Dazu wird mittels der *get*-Methode ein Pfad definiert und die Funktion bestimmt dann, was der Server bei Aufruf des Pfades ausliefert.

```
'use strict';

const express = require ( 'express' );
const server = express();

server.use ( express.static ( 'public' ) );

server.get ( '/time', (request, response) => {
    let zeit = new Date();
    response.send ( 'Zeit am Server: ' + zeit.toLocaleTimeString() );
});

server.listen ( 8080, 'localhost' );
```

## Logging

Um eine Abfrage mit Express zu loggen, hilft uns ein weiteres Modul, das Coremodul *path*. Coremodul bedeutet, dass dieses Modul bereits in Node integriert ist und nicht installiert werden muss.

Die Middleware, um das Logging vorzunehmen, besteht aus einer schlichten Funktion, die zum Schluss eine Callbackfunktion ausführt. Diese Callbackfunktion sorgt schlicht dafür, dass der Request weiter ausgeführt wird.

Wichtig ist dabei, dass diese Middleware vor dem *Static* eingebunden wird, da das *static* sonst den Request aufhält und nichts geloggt wird.

```
const express = require ( 'express' );
const url = require ( 'url' );

let server = express();

server.use ( (req, res, next) => {
    let abfrage = url.parse ( req.url, true );
    console.log ( abfrage.pathname );
    next();
});

server.use ( express.static ( 'public' ) );
```

```
server.listen ( 80 );
```

## Dateizugriff

Um auf das Dateisystem zugreifen zu können, binden wir das `fs`-Modul require ein.

Der einfachste Weg, Dateien zu lesen und zu schreiben ist über die `readFile`-, `writeFile`- und `appendFile`-Methode. Im Beispiel unten wird zunächst das Ergebnis der Multiplikation beider Parameter in eine Datei gespeichert. Dann wird dieselbe Datei wieder ausgelesen und an den Client ausgeliefert.

**Achtung:** Das asynchrone Laden und Speichern von Dateien benötigt eine Callback-Funktion, damit der Server an der Stelle nicht hängen bleibt.

```
const http = require ( 'http' );
const url = require ( 'url' );
const fs = require ( 'fs' );

http.createServer ( function ( request, response){

  let query = url.parse ( request.url, false ).query;
  let ergebnis = query.p1 * query.p2;

  fs.writeFile ( 'ergebnis.txt', ergebnis, error => {
    if (error) throw error;
    console.log('Gespeichert');
  });

  fs.appendFile ( 'ergebnis.txt', ' ist das Ergebnis', error => {
    if (error) throw error;
    console.log('String wurde angehängt');
  });

  fs.readFile ( 'ergebnis.txt', ( error, data ) => {
    response.writeHead ( 200, { 'Content-Type': 'text/html' } );
    response.write ( data );
    response.end();
  });
}).listen(8080);

console.log ( 'Server läuft ...' );
```

**Achtung:** `read` und `readSync` (wie in den WE-Unterlagen verwendet) sollten eigentlich nicht verwendet werden (<https://github.com/nodejs/node/issues/4530>).

## Dateien ausliefern

Bis jetzt liefern wir immer nur eine im Code festgelegte Datei aus. Das ist ggf. gut, wenn es sich um eine Single Page Application handelt.

Normalerweise wird aber über den Dateinamen bestimmt, welche Datei geladen werden soll. Um diese Standard-Situation zu erfüllen, müssen wir noch ein bisschen was umbauen.

Mit dem Code unten wird die Datei aus dem Dateisystem gelesen und deren Inhalt als HTML ausgeliefert.

```
const http = require ( 'http' );
const url = require ( 'url' );
const fs = require ( 'fs' );

http.createServer ( function ( request, response){

  let query = url.parse ( request.url, false );
  let datei = '.' + query.pathname ;
  console.log ( datei );

  fs.readFile ( datei, ( error, data ) => {
    response.writeHead(404, { 'Content-Type': 'text/html' });

    if ( error ) {
      console.log ( error );
      return response.end( 'error' );
    }

    response.write ( '' + data );
    response.end();

  });

}).listen(8081);

console.log ( 'Server läuft ...' );
```

## Datenübertragung (POST/JSON)

### Client

Um Daten vom Client zum Server zu übertragen, empfiehlt sich die POST-Methode in Zusammenhang mit JSON.

Wir schnüren auf Client-Seite die zu übertragenden Daten zu einem Objekt zusammen.

Dieses wandeln wir per JSON zu einem String und senden diesen per POST an den Server.

Die Rückgabe des Servers wird in diesem einfachen Beispiel in den Ausgabe-Container geschrieben.

```
<script>

window.onload = () => {
  document.querySelector ( '#btn' ).onclick = () => {

    let xhr = new XMLHttpRequest();
    xhr.open ( 'post', 'abfrage' );
    xhr.setRequestHeader ( 'Content-Type', 'application/json' );
    xhr.onload = ()=>{
      ergebnis.innerHTML = xhr.responseText;
    };
    xhr.send ( JSON.stringify ( {wert: eingabe.value } ) );
```

```

    }
  }

</script>
</head>

<body>

  <input id="eingabe" value="42">
  <button id="btn">Ausführung</button>

  <div id="ergebnis"></div>

</body>

```

## Server

Auf Serverseite bereiten wir einen *nodeJS*-Server vor, unsere Daten verarbeiten zu können. In unserem einfach Beispiel soll der übergebene Wert einfach mit 2 multipliziert und das Ergebnis zurück geliefert werden.

*Express* stellt uns den Webserver.

Interessant ist die Einbindung des Moduls *body-parser*, das zunächst installiert werden muss:

```
npm i body-parser
```

Das *body-parser*-Modul ermöglicht uns, die Parameter des *Requests* zu lesen.

Dazu müssen wir dem Server noch mitteilen, dass *JSON*-Daten zu verarbeiten sind.

In der *post*-Methode sind dann die Parameter dank des *body-parser*-Moduls im Objekt *body* zu finden.

Zu beachten ist noch, dass der zurückzusendende Text zwingend ein String sein muss.

```

'use strict';

const express = require ( 'express' );
const server = express();
const bodyParser = require ( 'body-parser' );

server.use ( express.static ( 'public' ) );
server.use ( bodyParser.json() );

server.post ( '/abfrage', (req, res) => {
  console.log ( req.body );
  let wert = req.body.wert;
  res.send ( String ( wert * 2 ) );
});

server.listen ( 80 );

```





# MySQL

NodeJS lässt sich recht simpel mit MySQL zusammenbringen.

Im Wesentlichen geht jede Datenbankabfrage in drei Schritten vor:

- Eine Verbindung zur Datenbank aufbauen
- Einen oder mehrere Queries absetzen
- Die Verbindung beenden

Zunächst muss das Modul `mysql` ggf. **installiert** und importiert werden:

```
var mysql = require('mysql');
```

Dann wird eine Verbindung zur Datenbank aufgebaut. Dazu wird zunächst die Verbindung definiert und mithilfe der `connect`-Methode die Verbindung hergestellt.

```
var con = mysql.createConnection({
  host: "localhost",
  user: "mustermann",
  password: "abc",
  database: "test",
});
```

Dann wird ein SQL-Query definiert und abgesendet.

```
let query = 'SELECT * FROM tabellenname';
con.query ( query, (error, result) => {
  if ( error ) throw error
  else {
    console.log ( result );
  }
});
```

Hier der gesamte Code:

```
'use strict';

const mysql = require ( 'mysql' );

let con = mysql.createConnection ( {
  host: 'localhost',
  user: 'maxMustermann',
  password: 'sicheresPasswort',
  database: 'meineDB'
});

/* Obwohl es in der Doku so beschrieben wird,
führen sowohl "connect" als auch "end" zu Problemen.
Dieser Weg funktioniert, könnte sich aber in Zukunft ändern.
*/
```

```
let query = 'SELECT * FROM tabelle';

con.query ( query, (error, result) => {
  if ( error ) throw error
  else {
    console.log ( 'RESULT: ' );
    console.log ( result );
  }
});
```

## Paketmanagement (package.json)

Um ein neues Projekt in NodeJS zu starten, empfiehlt es sich, eine sog. *package.json*-Datei anzulegen. In dieser Datei wird das Projekt beschrieben, sein Name, Version, Autor, Beschreibung, Keywords, Abhängigkeiten, etc.

Diese Datei ist grundsätzlich nicht unbedingt notwendig. Später, wenn das Projekt z.B. zu *npm* hochgeladen werden soll, braucht man es aber. Besser, man legt es gleich mit an.

Um diese Datei anzulegen, gibt es den Befehl

```
npm init
```

Dieser führt den Benutzer durch eine Reihe von Abfragen, in denen die Informationen zusammengetragen werden. Am Ende wird im aktuellen Ordner eine *package.json*-Datei erzeugt, die z.B. so aussehen kann:

```
{
  "name": "krasses_projekt",
  "version": "0.1.0",
  "description": "Mein erstes Projekt.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "C. Heisch",
  "license": "ISC"
}
```

Diese Datei lässt sich natürlich manuell nach Belieben verändern. Zu beachten ist dabei nur, dass es sich um richtiges [JSON \(Strg-Klick\)](#) handeln muss.

### Module

Falls für das Projekt ein Modul benötigt wird, dann muss dieses Modul auch in die *package.json*-Datei eingetragen werden. Das kann manuell erfolgen. Einfacher ist es aber, bei der Installation des Moduls die Option *save* einzutragen:

```
npm i chalk -save
```

Damit wird das Modul bzw. die Abhängigkeit in der aktuellen Version in die *package.json*-Datei eingetragen.

## Veröffentlichen

Das Projekt – meistens ist es ein Modul, das in andere Projekte integriert werden kann – kann dann auf npm veröffentlicht werden. Dazu sind die folgenden Schritte notwendig:

1. Eröffne ggf. ein eigenes Konto.
2. Melde Dich mit dem Befehl `npm login` auf dem Terminal ein.
3. Aktualisiere ggf. die *package.json* mit einem wirklich einmaligen Namen.
4. Stelle sicher, dass Du Dich im Projektordner befindest und gib ein: `npm publish` .  
(beachte den Punkt)

Damit wird Dein Projekt zu npm hochgeladen und ist quasi sofort für jeden verfügbar.

Bitte spamme nun aber nicht npm mit Testprojekten voll. Eine Suche nach 'test' auf npmjs.com ergab schon fast 30000 Projekte (Stand 05/18).

# ReactJS

React ist eine Javascript-Bibliothek aus dem Hause Facebook, die sich zunehmender Beliebtheit erfreut.

Sie dient dem Aufbau des Frontends mit reinem Javascript. Ziel ist das Erzeugen sog. Web-Apps - das sind Webseiten, die aussehen und sich verhalten wie Apps bzw. Programme - und wird z.B. für Facebook, Instagram, WhatsApp, Yahoo, AirBnB, oder den Atom-Editor verwendet.

Der Antrieb hinter React ist, Software möglichst einfach schreiben zu können. Wobei *einfach* nicht mit *leicht* verwechselt werden sollte. Mit *einfach* ist v.a. gemeint, dass der Code weniger verwoben und komplex ist.

React ist darauf ausgelegt, mit ECMAScript 2015 (a.k.a. ES6) zu arbeiten. Das heißt u.a., wir können auf die kompakten Schreibweisen und aktuellen Techniken (Konstanten, Klassen, etc.) zurückgreifen.

## Erste Schritte

Ein wesentlicher Unterschied zu den meisten anderen Bibliotheken ist, dass React-Code kompiliert werden muss, um seine volle Geschwindigkeit und gleichzeitige Entwicklungsfreundlichkeit zu entfalten. Aus dem Grunde macht eine produktive Entwicklung nur Sinn in einer IDE (Entwicklungsumgebung), die uns das Kompilieren und nebenbei auch Hilfsfunktionen wie Lint, Transpiling oder Codevervollständigung abnimmt.

Eine Diskussion über die Entwicklungsumgebungen ist hier zu finden:

<https://discuss.reactjs.org/t/what-is-the-best-ide-for-react/>

Für die ersten Schritte sind Projekte wie <http://stackblitz.com>, <https://codepen.io/>, <http://jsbin.com> oder <https://jsfiddle.net> hilfreich. Diese können zwar eine IDE nicht ersetzen, ermöglichen aber schnelle erste Einblicke.

Wird React wirklich produktiv verwendet, dann ist z.B. das Projekt *create-react-app* (<https://github.com/facebook/create-react-app>) hilfreich sein. Die Software fußt auf NodeJS und wird über *npm* ([Strg-Klick für mehr](#)) installiert. Sie enthält den Compiler, einen Server und einige nützliche Befehle.

Zur Installation wird das Tool *npx* verwendet. Dies ist ein Programm, das gegenüber *npm* den Vorteil hat, dass es auch im Paket enthaltene Skripte einfach ausführen kann. Dadurch wird der Installationsprozess erheblich vereinfacht.

*Npx* ist allerdings erst ab der *npm*-Version 5.2 (`npm --version`) integriert.

## Virtuelles DOM

Der Kern von React ist das sog. *Virtuelle DOM*. Es handelt sich hierbei v.a. um eine Technik zur Beschleunigung der Darstellung.

Jeder Zugriff und jede Änderung des DOMs ist sehr zeitintensiv, er ist fast immer der Flaschenhals. Um die langsamen DOM-Zugriffe zu minimieren und trotzdem keine Datenänderung zu "übersehen" haben sich die React-Entwickler etwas Interessantes einfallen lassen:

Im Inneren von React befinden sich mind. Zwei vereinfachte Kopien des DOM. Eine, die anzeigt, wie der Dom eben noch war und eine mit den aktuellen Änderungen. Da diese Kopien relativ

übersichtliche Objekte sind, können sie wesentlich schneller gehandhabt werden als das Original-DOM.

React vergleicht diese beiden Kopien miteinander, sucht die veränderten Stellen heraus und ändert diese ganz gezielt im Original-DOM.

Dadurch ergibt sich eine bis dahin ungeahnte Performance.

## Komponenten und JSX

Eine Komponente ist der kleinste und einzige Baustein, aus dem wir eine React-App bauen. Sie beschreibt in Javascript-Code, wie ein Bauteil der App aussehen und was er machen soll.

Jede App besteht aus vielen Komponenten, die kombiniert und verschachtelt werden. Eine Komponente kann z.B. ein Button sein, eine andere Komponente kann eine Überschrift sein, eine dritte kann ein Eingabefeld oder ein Inhaltstext sein. Eine weitere Komponente kann ein sein, die Überschrift, Inhaltstext und Eingabefeld(er) zu einem Kontaktformular kombiniert.

Die Möglichkeiten sind kaum begrenzt.

Grundsätzlich sollten wir ReactJS in ES6 programmieren. Denn es gibt keinen Grund, auf die modernen Möglichkeiten und Techniken zu verzichten.

Um eine einfache Komponente zu erzeugen, benutzen wir eine Klasse, die auf dem Komponentenklasse basiert.

**Achtung:** der Variablenname muss mit einem Großbuchstaben beginnen.

```
class App extends Component {  
}
```

### Erzeugung von Inhalt (JSX)

Eine Komponente kann viele Methoden enthalten, die wichtigste davon ist **render**.

Diese Methode (oder vielmehr ihr Rückgabewert) bestimmt, wie diese Komponente im Frontend angezeigt wird (Im Beispiel als H1-Überschrift).

Der Rückgabewert der render-Funktion sieht in diesem Beispiel aus wie HTML, ist es in Wirklichkeit aber nicht. Es handelt sich dabei um **JSX**, eine Syntaxerweiterung für Javascript, welche die Erzeugung von HTML-Code erleichtert.

Das scheinbare HTML wird in Javascript umgewandelt. Dieses Javascript enthält dann Methoden, die zur Laufzeit die gewünschten Elemente erzeugen.

**Achtung:** Der JSX-Code muss immer eine umgebende Komponente haben (Im Beispiel H1), da JSX in Javascript umgewandelt wird und Javascript nur einen Wert zurückgeben kann.

```
class MeineKomponente extends Component {  
  
  render() {  
    return (  
      <p>Hallo Welt</p>  
    );  
  }  
}
```

### Darstellung einer Komponente auf einer Webseite

Um die Komponenten in der Seite darzustellen, werden sie mit dem Render-Befehl an ein HTML-Element übergeben.

```
class MeineKomponente extends Component {  
  
  render() {  
    return (  
      <p>Hallo Welt</p>  
    );  
  }  
  
}  
  
render(<MeineKomponente />, document.querySelector('#root'));
```

### Vereinfachung mit Stateless Functional Components

So manche Komponente dient einfach nur dazu, Inhalte darzustellen. Ganz ohne lokale Variablen, Funktionen und dergleichen. In solch einem Fall (und in anderen Fällen auch) spart eine *Stateless Functional Component* viel Zeit und Programmzeilen

Daneben ist eine bessere Performance gegenüber klassenbasierten Komponenten ein weiterer Vorteil.

Grundsätzlich ist eine *Stateless Functional Komponent* einfach eine Funktion, die eine Ausgabe erzeugt. In der einfachsten Form sieht sie so aus:

```
let Footer = () =>  
  <footer>Ich bin der Fuss.</footer>
```

*Stateless Functional Components* können natürlich ganz normal verschachtelt werden.

Und sie können auch Parameter bekommen. Diese Parameter sind wegen des anderen Aufbaus allerdings anders zu schreiben:

```
let Footer = () =>  
  <footer>  
    Ich bin der Fuß.<br />  
    <Link url="http://www.irgendwas.de" inhalt="Klick mich" />  
  </footer>  
  
let Link = props =>  
  <a href={props.url}> { props.inhalt } </a>
```

In eine *Stateless Functional Component* lassen sich auch ohne weiteres beliebige Funktionen einbinden:

```
let Footer = () => {  
  
  const klick = evt => console.log ( evt.currentTarget );  
  
}
```

```

return (
  <footer>
    Ich bin der Fuß.<br />
    <button onClick={klick}>Klick mich </button>
  </footer>
)

```

## Verschachtelung

Ein zentrale Funktionsweise von React ist, dass wir Komponenten miteinander verschachteln. Dazu rufen wir in einer Komponente eine andere Komponente mithilfe der spitzen Klammern auf. Die aufgerufene Komponente erzeugt dann das anzuzeigende Element und gibt es als Rückgabewert an den Aufruf zurück.

```

class Wurzel extends Component {
  render() {
    return (
      <div>
        <Header />
        <Inhalt />
      </div>
    )
  }
}

class Header extends Component {
  render() {
    return (
      <h1>Hallo Welt</h1>
    );
  }
}

class Inhalt extends Component {
  render() {
    return (
      <p>Ich bin ein Text</p>
    )
  }
}

render ( <Wurzel />, document.querySelector('#root'));

```

## Lokale Variablen

Ein sehr häufig verwendetes Attribut von React ist **state**. Dieses Objekt enthält Daten, die nur von dieser Komponente verwaltet werden können.

Das *state*-Objekt wird wie jedes andere Attribut mit der *constructor*-Methode angelegt. Der Vorteil das *state*-Objektes ist, dass *React* einige Methoden bietet, die das *state*-Objekt verwenden und zusätzlichen Luxus bieten. So zum Beispiel *setState*, das nicht nur die Daten ändert, sondern gleich auch alle Komponenten neu rendert, in denen die Daten aus dem *state* verwendet werden.

```

class Wurzel extends Component {

```

```

constructor() {
  super();
  this.state = {
    meineZahl: 42,
  }
}

render() {
  return(
    <h1>
      {this.state.meineZahl}
    </h1>
  )
}
}

```

### Parameter an eine Komponente übergeben

Parameter, die an eine Komponente übergeben werden, heißen *props*. Diese Parameter können in der Komponente nicht verändert werden.

Um einen prop an eine Komponente zu übergeben, wird dieser ähnlich einem Tag-Attribut in den Komponentenaufruf integriert.

In der aufgerufenen Komponente kann dann das props-Objekt gelesen werden, das den prop enthält.

Um mehrere Props zu übergeben, schreibe einfach beide hintereinander in den Aufruf.

```

class Wurzel extends Component {
  render() {
    return(
      <div>
        <Inhalt text1="ABC" text2="XYZ" />
      </div>
    )
  }
}

class Inhalt extends Component {
  render() {
    return (
      <div>
        <p>{this.props.text1}</p>
        <p>{this.props.text2}</p>
      </div>
    )
  }
}

```

### In einer Komponente rechnen

Der return-Befehl lässt keine Rechenoperationen zu. Um in einer Komponente zu rechnen, existieren zwei Möglichkeiten.



Führe die Rechnung vor dem return aus, speichere das Ergebnis in einer Variablen und benutze dann im return das Ergebnis.

```
class Inhalt extends Component {
  render() {
    let ergebnis = 42 * 23;
    return (
      <p>{ ergebnis }</p>
    )
  }
}
```

Oder lege eine Funktion an, die diese Rechnung ausführt und das Ergebnis per return zurückliefert. Rufe dann in der Renderfunktion einfach diese Funktion auf.

```
class Inhalt extends Component {
  rechne() {
    return 42 * 23;
  }
  render() {
    return (
      <p>{ this.rechne() }</p>
    )
  }
}
```

## Klassen vergeben

Um einem Element eine Klasse zu geben, brauchen wir das Schlüsselwort `className`. Im Beispiel unten werden dem `p`-Tag zwei Klassen gegeben: `feld` und `element`.

```
class Inhalt extends Component {

  render() {
    return (
      <p className="feld element">Hallo Welt</p>
    )
  }
}
```

## Eventlistener

Eventlistener werden in React scheinbar inline angelegt (im Beispiel als *onclick*-Attribut des HTML-Tags).

Dies war und ist einer der Hauptkritikpunkte an React: Die Idee der Auftrennung von Logik und Inhalt wird nicht verfolgt. Stattdessen wird gesagt, dass die Logik (der Javascript-Code) in React so eng mit dem Inhalt zusammenhängt, dass eine Trennung unsinnig ist.

Davon abgesehen ist im resultierenden HTML-Code im Client der *onclick*-EventListener nicht mehr im Tag eingetragen.

Der folgende Code lässt den Text *'geklickt'* ausgeben, wenn der Button angeklickt wird. Dazu wurde eine neue Methode namens *ausgabe* angelegt. Diese wird durch den `onClick`-EventListener aufgerufen.

**Achtung:** `onClick` muss mit innerCaps geschrieben werden, da der Begriff ohne innerCaps (`onclick`) bereits ein Javascript-Schlüsselwort ist und daher nicht verwendet werden darf.

Die auszuführende Funktion (*this.ausgabe*) wird in geschweiften Klammern geschrieben, damit diese Funktion als JS-Code erkannt wird.

```
class Btn extends Component {  
  
  ausgabe() {  
    alert ( 'geklickt' );  
  }  
  
  render() {  
    return (  
      <button onClick={this.ausgabe}>  
        Klick mich  
      </button>  
    )  
  }  
}
```

### Komponente geladen

Die `componentDidMount`-Methode bestimmt eine Funktion, die ausgeführt wird, sobald diese Komponente erfolgreich in die Seite eingebunden wurde.

Für jede eingebundene Komponente wird diese Methode jeweils einmal ausgeführt.

```
class Inhalt extends Component {  
  componentDidMount() {  
    console.log ( 'Element angelegt' )  
  }  
  render() {  
    return (  
      <p>Hallo Welt</p>  
    )  
  }  
}
```

### Mehrere Elemente anlegen

Um mehrere Elemente gleichzeitig anzulegen, müssen diese Elemente zunächst in einem Array zusammengefasst werden. Im `return` wird dieses vorher konstruierte Array ausgegeben.

```
class Wurzel extends Component {  
  constructor() {  
    super();  
    this.state = {  
      inhalte: [ 'Hallo', 'Welt', 'Klasse', 'Schnee', 'Kaffee' ]  
    }  
  }  
  render() {  
    let komponenten = [];  
    for ( let i of this.state.inhalte ) {  
      komponenten.push ( <Inhalt wert={i} /> )  
    }  
    return (  
      <div>  
        {komponenten}  
      </div>  
    )  
  }  
}
```

```

        </div>
    )}}

class Inhalt extends Component {
  render() {
    return (
      <p>{this.props.wert}</p>
    )}}

```

## Immer aktuell

Im Beispiel wird mit Hilfe eines Intervalls ein *state-Objekt* immer wieder geändert. Hier nutzen wir das Verhalten, dass nach dem Ausführen der *setState*-Methode die Komponente neu gerendert wird.

Besonderes Augenmerk sollte darauf gerichtet werden, dass die *update*-Methode kein *this* kennt. Daher übergeben wir das *this* aus der *componentDidMount*-Methode als Parameter an jene Methode.

```

class Counter extends Component {
  constructor() {
    super();
    this.state = {
      zahl: 0
    }
  }

  update( that ) {
    // that.state.index++;
    that.setState ({
      zahl: that.state.zahl + 1
    })
  }

  componentDidMount() {
    setInterval ( this.update, 1000, this );
  }

  render() {
    return (
      <h1>{this.state.zahl}</h1>
    )}}

```

## Kommentare

Ein Kommentar in JSX muss als mehrzeiliger Kommentar angelegt werden, damit das Kommentar-Ende genau definiert werden kann. Dieser Kommentar muss dann noch zusätzlich in eine *{}* geschachtelt werden, damit der Kommentar als zu interpretierender Code erkannt wird.

```

class Inhalt extends Component {
  render() {
    return (
      { /* Dies ist die Ausgabe */ }
      <p>Hallo Welt</p>
    )}}

```

## reactJS und AJAX

Anders als bei z.B. *AngularJS* oder *jQuery* hat *reactJS* keine Ajax-Funktionen mitgeliefert.

Tatsächlich können Ajax-Abfragen in reactJS erledigt werden, wie in normalem Javascript auch. Allerdings innerhalb der reactJS-typischen Strukturen:

```
class App extends Component {
  constructor() {
    super();
    this.state = {
      data: 0
    };
  }

  componentDidMount() {

    // Ajax-Funktionen
    let xhr = new XMLHttpRequest();
    xhr.open('GET', './daten.json', true);
    xhr.onload = () => {
      this.setState ( {data: xhr.responseText } );
    }
    xhr.send();

  }

  render() {
    return (
      <div>
        {this.state.data}
      </div>
    );
  }
}

render(<App />, document.getElementById('root'));
```

Oder einfacher mit der *fetch*-Methode:

```
class App extends Component {
  constructor() {
    super();
    this.state = {
      data: 0
    };
  }

  componentDidMount() {
    fetch('./daten.json')
      .then(res => {
        console.log ( res )
      })
  }

  render() {
```

```
    return (  
      <div>  
        {this.state.data}  
      </div>  
    );  
  }  
  
  render(<App />, document.getElementById('root'));
```

**Achtung:** in der Stackblitz-Umgebung funktioniert dieser Weg so nicht, weil der Server nicht die erwarteten Ergebnisse liefert.

# Vue

Vue ist ein JS-Framework, dass sich im Kern damit auseinandersetzt, Daten auf dem Bildschirm darzustellen (view).

Dazu kann eine Seite in verschiedene Teile untergliedert werden, die alle von verschiedenen Skripten unabhängig voneinander angesteuert werden können.

Wann immer sich die zugrundeliegenden Daten ändern, wird die Ansicht aktualisiert.

Vue hat einige Parallelen zu *AngularJS*, aber auch zu *ReactJS*. Wer sich in jenen Frameworks also wohlfühlt, wird sich auch schnell mit *Vue* anfreunden.

**Achtung:** Vue unterstützt nicht den Internet Explorer Version 8 und darunter. In meinen Augen eine vernünftige Entscheidung.

## Installation

### CDN

Der einfachste Weg, Vue zu integrieren ist über CDN.

Hier bieten sich diese beiden Code-Zeilen an:

**Entwicklungsversion** mit vielen hilfreichen Warnungen in der Konsole:

```
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

**Produktionsversion** für die fertige Webseite. Optimiert auf Geschwindigkeit und Dateigröße:

```
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

### Download

Oder man lädt Vue als *Datei* herunter und bindet diese ein.

Hier findet sich die jeweils aktuellste Version:

<https://vuejs.org/v2/guide/installation.html>

Mit einem einfachen `<script>`-Befehl bindet man dann die Datei ein.

### NPM

Vue kann auch mittels NPM installiert werden.

```
npm i vue
```

Das ist aber nur sinnvoll bei größeren Projekten, weil dann auch ein *Module Bundler* wie *Webpack* oder *Browserify* benötigt wird.

In dem Fall kann mit der offiziellen Kommandozeilenebene (<https://github.com/vuejs/vue-cli>) mit ein paar einfachen Befehlen ganze Projekte aus dem Boden gestampft werden. Das wird hier aber nicht thematisiert.

# Darstellung im Virtual DOM

Der HTML-Bereich eines Dokumentes wird von Vue als Template betrachtet.

Die Inhalte werden nach Direktiven und Expressions durchsucht, die dann durch Vue zu gültigem HTML umgewandelt werden.

Bei Änderungen an den Daten ermittelt Vue automatisch die zu verändernden Inhaltselemente und aktualisiert diese.

## DOM-Inhalt

Vue macht seine Ausgaben mittels Expressions. Diese werden mit einer Doppelten geschweiften Klammer markiert und lesen das Attribut des Objektes aus, das sich auf den Anzeigebereich bezieht:

Der umschließende DIV-Container dient dem leichten Zugriff über CSS-Selektoren.

```
<div id="ausgabe">
  {{ nachricht }}
</div>
```

Das Objekt, aus dem die Daten bezogen werden, legen wir mithilfe des Vue-Konstruktors an. Dieser wird mit dem DOM-Element verknüpft, auf das sich das *el*-Attribut bezieht.

Die zwei benötigten Parameter sind also:

- *el* – für Element. Bestimmt mit einem CSS-Selector, in welchem Element der Inhalt verwendet wird. Wenn es mehr als einen Hit gibt, dann wird das erste genommen.  
Das Attribut **muss** *el* heißen.
- *data* – bestimmt ein Objekt mit den Daten, die im DOM-Objekt dargestellt werden sollen.  
Das Attribut **muss** *data* heißen.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    nachricht: 'Hallo Welt'
  })
```

Um später die Nachricht zu verändern, greifen wir auf das *app*-Objekt zu.

Auch wenn der Konstruktor dies vermuten lässt, sind die Attribute nicht in einem *data*-Attribut verschachtelt. Viel mehr sind Attribute und Methoden direkte Kinder des Objektes.

Anders ausgedrückt: Um die Nachricht zu ändern, verzichten wir auf '*data*':

```
app.nachricht = 'Vue ist einfach';
```

## JS-Code in Expressions

Innerhalb der Ausdrücke können wir gültiges Javascript eintragen. Dies wird zuverlässig ausgeführt.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
```

```
werte: [54,20,89,20,16,60]
}});
```

```
<div id="ausgabe">
  <p> {{ 42 * 23 }} </p>
  <p> Summe: {{ werte.reduce ( (s,e) => s+e ) }} </p>
  <p> alle Volljährig: {{ werte.some ( e => e<18 ) ? 'Nein' : 'Ja' }}
</div>
```

## Direktiven

Für interessantere Effekte, z.B. Wiederholungen oder die Verknüpfung von Attributen, gibt es Direktiven. Diese werden wie HTML-Attribute eingetragen.

### Attribute verknüpfen mit v-bind

Wir können Attribute eines DOM-Elementes mit Daten füttern. Dazu bietet sich die *v-bind*-Direktive an. An diese mit einem Doppelpunkt angehängt steht die Information, welches Attribut verknüpft werden soll. Die Information wird hier **ohne** die geschweiften Klammern eingetragen.

```
var bild = new Vue({
  el: '#meinBild',
  data: {
    bild: 'http://via.placeholder.com/350x150'
  })
```

```

```

Natürlich lassen sich *Expressions* und *Bindings* auch kombinieren:

```
<script>
  var app = new Vue({
    el: '#ausgabe',
    data: {
      nachricht: 'Hallo Welt',
      bild: 'http://via.placeholder.com/350x150'
    })
  });
</script>
```

```
<div id="ausgabe">
  {{ nachricht }}<br>
  
</div>
```



Für *v-bind* können wir auch die **Kurzschreibweise** verwenden, bei der schlicht der Begriff *v-bind* weggelassen wird. Diese sollte aber nur verwendet werden, wenn keine zusätzliche Bibliothek diese Schreibweise für sich in Anspruch nimmt.

```
<div id="ausgabe">

  <!-- ausgeschrieben -->
  <a v-bind:href="url">Google</a><br>

  <!-- Kurz -->
  <a :href="url">Google</a>

</div>
```

Die **Zuweisung von Klassen** ist ein häufiger Wunsch.

Mithilfe von Vue können wir sehr einfach Klassen zuweisen und deren Benutzung von einem Boolean abhängig machen.

Mehrere Klassen werden mit Komma getrennt.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    werte: [ 21, 51, 42, 23, 95 ],
  },
  computed: {
    minderjaehrig: function(){
      return this.werte.some ( e => e < 18 );
    },
    USadult: function(){
      return this.werte.every ( e => e >= 21 );
    }
  }
});
```

```
<div id="ausgabe">
  <p v-bind:class="{ 'rot':minderjaehrig, 'fett': USadult }">
    {{ werte }}
  </p>
</div>
```

## Wiederholungen mit v-for

Um ein Array auf der Seite darzustellen, ist eine Wiederholung hilfreich. Die *v-for*-Direktive leistet genau das: im Parameter-Wert wird angegeben, welches Array verwendet wird und unter welchem Namen die einzelnen Speicherstellen des Arrays im DOM-Element ansprechbar sind.

Genauso, wie in der *ng-repeat*-Direktive von AngularJS.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    werte: [ 12, 51, 42, 23, 95 ]
  }
});
```

```
});
```

```
<div id="ausgabe">
  <ul>
    <li v-for="i in werte">
      {{ i }}
    </li>
  </ul>
</div>
```

Wir können auch eine einfache Schleife anlegen, die mehrere Elemente erzeugt.

```
<div id="ausgabe">
  <ol>
    <li v-for="num in 42">
      {{ num }}
    </li>
  </ol>
</div>
```

### Von Update ausschließen mit v-once

Wird eine Expression mit der v-once-Direktive ausgestattet, so wird sie nur bei Start der Seite einmal gefüllt und danach nicht mehr aktualisiert.

```
<div id="ausgabe" v-once>
  {{ werte }}
</div>
```

### HTML interpretieren mit v-html

Normalerweise werden Daten als reiner Text gesehen und entsprechend ohne HTML-Interpretation angezeigt. Wenn wir Inhalte doch als HTML-Code interpretieren lassen wollen, dann müssen wir die Daten der v-html-Direktive geben. Die anzuzeigenden Daten werden dann im Tag angezeigt.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    werte: '<b>Fetter Text</b>',
  }
});
```

```
<div id="ausgabe">
  <p>{{ werte }}</p>           // -> <b>Fetter Text</b>
  <p v-html="werte"></p>      // -> Fetter Text
</div>
```

## Sichtbarkeit steuern mit v-if/v-else

In der `v-if`-Direktive können wir eine Angabe machen, die zu einem Boolean aufgelöst wird. Dieser Boolean bestimmt dann, ob das Element angezeigt werden soll oder nicht.

Wir können auch einen Else-Fall einfügen.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    werte: [54,20,89,20,16,60]
  }});
```

```
<div id="ausgabe">
  <p v-if="werte.every ( e => e >= 21 )">Alle Volljährig (USA)</p>
  <p v-else-if="werte.every ( e => e >= 18 )">Alle Volljährig (EU)</p>
  <p v-else>Mind. eine(r) zu jung.</p>
</div>
```

## Formularfelder lesen und füllen mit v-model

Die `v-model`-Direktive wird in Formular-Eingabefeldern verwendet. Es ermöglicht uns, den Inhalt des Eingabefeldes auszulesen oder zu überschreiben.

Das Attribut muss vor der Verwendung im `Vue`-Objekt angelegt worden sein.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    meinText: ''
  },
});
```

```
<div id="ausgabe">
  <input v-model="meinText"><br>
  <p v-html="meinText"></p>
</div>
```

Einfache **Checkboxes** liefern ein Boolean bzw. nehmen ein solches an.

```
<div id="ausgabe">
  <input type="checkbox" v-model="meinHaken"><br>
  <p v-html="meinHaken"></p>
</div>
```

Wenn mehrere Checkboxes dasselbe `v-model`-Attribut tragen, können alle zu einem Array zusammengeführt werden. Dazu müssen wir jeder Checkbox einen individuellen *Value* geben und das *Attribut* im `Vue`-Objekt muss einen Array als Initial-Inhalt bekommen.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    cb: ['h2']
  },
});
```

```
<div id="ausgabe">
  <input type="checkbox" value="h1" v-model="cb"><br>
  <input type="checkbox" value="h2" v-model="cb"><br>
  <input type="checkbox" value="h3" v-model="cb"><br>
  <p>
    {{ cb }}
  </p>
</div>
```

**Radiobuttons** arbeiten ganz ähnlich wie Checkboxes. Das Attribut im *Vue*-Objekt enthält einen String, der mit der *v-model*-Direktive verknüpft ist. Jede Änderung der Auswahl ändert das Attribut und jede Änderung des Attributes ändert die Auswahl.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    auswahl: 'drei'
  }});
```

```
<div id="ausgabe">
  <input type="radio" name="rb" v-model="auswahl" value="eins"><br>
  <input type="radio" name="rb" v-model="auswahl" value="zwei"><br>
  <input type="radio" name="rb" v-model="auswahl" value="drei"><br>
  <p>{{auswahl}}</p>
</div>
```

**Auswahllisten** verhalten sich etwas komplexer.

Wenn nur eine Option ausgewählt werden kann, dann enthält das Attribut im *Vue*-Objekt einen String. Dieser ist mit den *Value*-Attributen der Options-Elemente verknüpft. Falls das Element kein Value-Attribut besitzt, so wird das Attribut mit dem Textinhalt verknüpft.

**Achtung:** Wenn der Initialwert des Attributes mit keinem *Option*-Element übereinstimmt, dann rendert das Element als *'unselected'*. In dem Fall wird ggf. kein Event gefeuert, falls die erste Option ausgewählt wird.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
```

```
    auswahl: ''
  }
});
```

```
<div id="ausgabe">
  <select v-model="auswahl">
    <option value="">Bitte auswählen</option>
    <option>Hallo Welt</option>
    <option>Vue ist einfach</option>
    <option>Qual der Wahl</option>
  </select>
  <p>{{auswahl}}</p>
</div>
```

Bei einem Auswahlfeld mit **multipler Auswahl** enthält das verknüpfte Attribut ein Array.

```
<div id="ausgabe">
  <select v-model="auswahl" multiple>
    <option value="">Bitte auswählen</option>
    <option>Hallo Welt</option>
    <option>Vue ist einfach</option>
    <option>Qual der Wahl</option>
  </select>
  <p>{{auswahl}}</p>
</div>
```

*v-model* kann die folgenden **Modifizier** verstehen:

- `.lazy` – feuert den event erst bei enter oder verlorenem Fokus statt bei jedem Tastenanschlag.
- `.number` – Die Eingabe wird in den Datentyp Number konvertiert.
- `.trim` – Whitespaces werden entfernt.

### Elemente überspringen mit *v-pre*

Elemente mit der *v-pre*-Direktive werden von *Vue* nicht berücksichtigt.

Das kann sinnvoll sein, wenn wir die doppelten Klammern angezeigt haben wollen oder auch, um den Code zu beschleunigen.

```
<div id="ausgabe">
  <div v-pre> {{ 42 * 23 }} </div>
</div>
```

### Anzeige verzögern mit *v-cloak*

Die *v-cloak*-Direktive hat keine wirkliche Wirkung. Sie bleibt einfach im DOM-Element stehen und wird nach der vollständigen Kompilierung der Webseite entfernt.

Der Sinn ist, dass mithilfe einer einfachen CSS-Angabe die Sichtbarkeit des Elements gesteuert werden kann.

```
<style type="text/css">
  [v-cloak]{
    display:none;
  }
</style>
```

```
<div id="ausgabe">
  <!-- Element wird erst nach dem Compiling angezeigt -->
  <div v-cloak> {{ 42 * 23 }} </div>
</div>
```

## Modifiers

Ein Modifier gibt zusätzliche Anweisungen, was geschehen soll und wie mit den Daten umgegangen werden soll. Er wird mit einem Punkt an den Event gehängt.

Direktiven verstehen die folgenden Modifier:

- `.prop` bindet eine Property statt eines Attributes ein. [Strg-Klick für mehr](#).
- `.camel` nimmt Bindestriche aus Attributnamen heraus.
- `.sync` ermöglicht eine Synchronisation

Modifiers können auch **verkettet** werden.

```
<div id="ausgabe">
  <div :meine-id.camel="auswahl">Hallo Welt</div>
</div>
```

## Eigene Direktiven

Neben den eingebauten Direktiven bietet Vue auch die Möglichkeit, eigene Direktiven anzulegen. Die Bedeutung dieser Möglichkeit hat zwar seit *Version 2* zugunsten von *Komponenten* abgenommen, in einigen Fällen kann es aber durchaus sinnvoll sein.

Um eine eigene Direktive anzulegen, bietet das Vue-Objekt den `directive`-Konstruktor. Dieser bekommt zwei Parameter:

- Den Namen der Direktive. Hier sind nur Kleinbuchstaben erlaubt.
- Ein Objekt, welches das Verhalten der Direktive steuert.

Beispielhaft legen wir eine Direktive an, welche die Anzahl der Zeichen in einem Element zählt und als Kindelement ausgibt.

Die Direktive besteht aus einer Funktion, die zu einem bestimmten Zeitpunkt (dem sog. *Hook*) ausgeführt werden kann. Mögliche Zeitpunkte sind:

- `bind`: Wird nur einmal ausgeführt in dem Augenblick, da die Direktive an das Element vergeben wird (normalerweise direkt nach dem Laden der Seite).
- `inserted`: Wird ausgeführt, wenn das DOM-Element in sein Elternelement eingehängt wurde. Dies bedeutet nicht automatisch, dass das *Elternelement* auf der Seite eingebunden ist.
- `update`: Wenn sich eine Information ändert und das DOM-Element aktualisiert wird. Das bedeutet nicht unbedingt, dass alle KindElement schon aktualisiert sind.
- `componentUpdated`: Wenn das Element und alle Kinder aktualisiert wurden.
- `unbind`: Wenn die Direktive von dem Element entfernt wird.

Der *Hook* bekommt dann eine Funktion zugewiesen, die zu dem definierten Zeitpunkt ausgeführt wird.

Diese Funktion kann mehrere Parameter annehmen, die Namen dieser Parameter sind festgelegt:

- `el` – Das DOM-Element mit der Direktive
- Das `Binding`-Objekt mit den folgenden Kind-Attributen:
  - `name`: Der Name der Direktive ohne das 'v-'
  - `value`: Ggf. der Wert, der als Parameter an die Direktive übergeben wurde.
  - `oldValue`: Der vorherige Wert. Funktioniert bei `update`.
  - `expression`: Der nicht interpretierte Ausdruck als String. Z.B.:
  - `arg`: Der Parameter, der dieser Direktive gegeben wurde.
  - `modifiers`: Die Modifier, der dieser Direktive gegeben wurden, als Objekt.

```
Vue.directive ( 'addlaenge', {
  inserted: function(el, binding){
    let neu = document.createElement ( 'div' );
    neu.classList.add ( 'menge' );
    neu.innerHTML = el.innerHTML.length;
    el.appendChild ( neu );

    console.log ( 'name: ', el.innerHTML.substr(0,10) );
    console.log ( 'val: ', binding.value );
    console.log ( 'arg: ', binding.arg );
    console.log ( 'mod: ', binding.modifiers );
    console.log ( ' ' );
  }
});

var app = new Vue({
  el: '#ausgabe',
  data: {
    meinParameter: 'Hallo',
  },
});
```

```
<div id="ausgabe">
  <div v-addlaenge>Hallo Welt</div>
  <div v-addlaenge="meinParameter">Consectetuer adipiscing.</div>
  <div v-addlaenge:meinArgument>Integer tincidunt.</div>
  <div v-addlaenge:meineMethode.a>Cras dapibus.</div>
```

```
</div>
```

## Methoden

Wir können einem Vue-Objekt auch Methoden geben, die z.B. durch einen Eventlistener gestartet werden. Dazu geben wir unserem Konstruktor das *methods*-Objekt mit.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    werte: [ 51, 42, 23, 12, 95 ],
  },
  methods: {
    sortieren: function(){
      this.werte.sort ( (a,b) => a-b );
    },
  },
});

app.sortieren();
```

```
<div id="ausgabe">
  {{ werte }}
</div>
```

## Computed Properties

Die *Expressions* von *Vue* verstehen zwar Javascript, allerdings wird eine solche Expression schnell unübersichtlich.

Auch, wenn mehrere Zeilen Code notwendig sind, um den gewünschten Wert zu erhalten, gehen den Javascript-Fähigkeiten der *Vue-Expressions* schnell die Luft aus.

Hierfür wurden *Computed Properties* (CP) erdacht, hinter denen sich nichts anderes als Getter ([Strg-Klick für mehr](#)) verbergen.

Eine *CP* ist also im Grunde nichts anderes als eine Methode ohne Parameter, die einen Rückgabewert liefert. Dieser Rückgabewert wird in der Expression angezeigt.

Der wesentliche Unterschied einer *Computed Property* gegenüber einer Methode, die dasselbe Ergebnis hätte, ist das Caching. Die Ergebnisse einer *CP* werden gecacht, bis sich die Daten ändern. Dadurch muss nicht jedes Mal die Methode ausgeführt werden und unser Code wird etwas schneller. Allerdings heißt das auch, dass von dem Objekt sich unabhängig ändernde Daten (z.B. Uhrzeit) nicht berücksichtigt werden.

Zum Anlegen einer *CP* muss in dem *Vue*-Objekt der *computed*-Bereich eingetragen werden:

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    werte: [ 12, 51, 42, 23, 95 ],
  },
```



```

    computed: {
      groesster: function(){
        let temp = this.werte.sort((a,b)=>a-b);
        return temp[temp.length-1]
      }
    }
  });

```

```

<div id="ausgabe">
  {{ groesster }}
</div>

```

## Eventlistener

Um einen Eventlistener anzulegen, benutzen wir die *v-on*-Direktive. Diese bestimmt den Event und die Methode, die dadurch gefeuert werden soll.

```

var app = new Vue({
  el: '#ausgabe',
  data: {
    werte: [ 51, 42, 23, 12, 95 ],
  },
  methods: {
    sortieren: function(){
      this.werte.sort ( (a,b) => a-b );
    },
  },
});

```

```

<div id="ausgabe" v-on:click="sortieren">
  {{ werte }}
</div>

```

Um an die Methode Parameter zu übertragen, hängen wir diese einfach in Klammern dahinter.

```

var app = new Vue({
  el: '#ausgabe',
  data: {
    werte: [ 51, 42, 23, 12, 95 ],
  },
  methods: {
    mehr: function(wieviele){
      for ( var i = 0; i < wieviele; i++){
        this.werte.push ( Math.floor(Math.random()*100));
      },
    },
  },
});

```

```

<div id="ausgabe">

```

```
<p v-on:click="mehr(12)">{{ werte }}</p>
</div>
```

Es muss nicht immer zwangsläufig eine Methode gefeuert werden. Manchmal will man einfach nur einen Wert ändern oder sonst einen einfachen JS-Code ausführen. Dazu tragen wir den gewünschten Code einfach als Wert in den *v-on:event* ein.

```
var app = new Vue({
  el: '#ausgabe',
  data: {
    wieOft: 0,
  },
});
```

```
<div id="ausgabe">
  <button v-on:click="wieOft++">Klick mich</button>
  <p>{{ wieOft }} mal geklickt</p>
</div>
```

## Kurzschreibweise

Um Code zu sparen, kann das *v-on:* durch ein *@* ersetzt werden.

```
<div id="ausgabe">

  <!-- ausgeschrieben -->
  <p v-on:click="sortieren">{{ werte }}</p>

  <!-- Kurzschreibweise -->
  <p @click="sortieren">{{ werte }}</p>
</div>
```

## Modifiers

Ein Modifier gibt zusätzliche Anweisungen, was geschehen soll und wie mit den Daten umgegangen werden soll. Er wird mit einem Punkt an den Event gehängt.

Events verstehen die folgenden Modifiers:

- *.stop* – Ausführung des Events wird gestoppt.
- *.prevent* – unterdrückt die Standard-Aktion.
- *.capture* – Events von Kindelementen feuern diesen Event vor ihrem eigenen.
- *.self* – Event wird nur gefeuert, wenn das Element selbst geklickt wurde und kein Kind
- *.once* – Event wird nur einmal gefeuert
- *.42* u.a. – Im Falle eines Tastatur-Events wird dieser Event nur dann gefeuert, wenn die Taste mit dem ASCII-Code 42 gedrückt wurde. Gilt für alle Tasten/ASCII-Codes
- *.enter* u.a. – Im Falle eines Tastatur-Events wird dieser Event nur dann gefeuert, wenn die Enter-Taste gedrückt wurde. Geht auch für *tab*, *delete* (feuert auch bei Backspace), *esc*, *space*, *up*, *down*, *left* und *right*.

- `.ctrl` u.a. – Event (Tastatur oder Maus) feuert nur, wenn auch die *Strg*-Taste gehalten wurde. Geht auch für *alt*, *shift* und *meta* (Windows- bzw. Command-Taste).
- `.left` – Linke Maustaste
- `.right` – Rechte Maustaste
- `.middle` – Mittlere Maustaste

Modifiers können auch **verkettet** werden.

```
// Submit-Funktion wird unterdrückt
<form v-on:submit.prevent="onSubmit"></form>

// Wenn Alt-Entf gedrückt wurde
<input v-on:keyup.alt.delete="leeren">

// Rechtsklick
<button v-on:click.right="rechtsklick">BTN</a>
```

## Komponenten

Eine Komponente ist eine Vorlage bzw. ein Template für Elemente, die wir auf der Seite einbinden wollen.

Der Vorteil von Komponenten liegt darin, dass wir äußerst komplexe Elemente als Komponenten anlegen können. Und diese können wir beliebig oft in der Webseite einbinden. Das spart Arbeit beim Anlegen und bei der Pflege.

Komponenten sind also, ähnlich wie Funktionen, mehrfach verwendbar.

Das Anlegen einer Komponente erfolgt in drei Schritten.

Zunächst **definieren wir die Komponente** mithilfe des *component*-Konstruktors.

Im Beispiel definieren wir die einfachste vorstellbare Komponente. Sie besteht nur aus dem Template-Attribut, welches den Aufbau im HTML-Teil bestimmt.

Das Template muss einen einzigen Root-Tag besitzen.

```
Vue.component ( 'ausgabe', {
  template: '<p>Hallo Welt</p>'
});
```

Soll die Komponente auch mit Daten haushalten, so müssen diese als Rückgabewert einer Funktion erzeugt werden. Damit wird sichergestellt, dass jede Komponente ihren eigenen Datenatz hat.

```
Vue.component ( 'ausgabe', {
  data: function(){
    return {
      zaehler: 0
    }
  },
  template: '<p>Klick mich: {{ zaehler }}</p>'
});
```

Soll die Komponente auch interaktive Elemente haben, so benötigt sie Methoden, die aufgerufen werden können. Diese werden im *methods*-Objekt gehalten. Eine Funktion mit Rückgabewert ist hier nicht nötig.

Diese Methoden können wie gewohnt mit *Direktiven* aufgerufen werden.

```
Vue.component ( 'ausgabe', {
  data: function(){
    return {
      zaehler: 0
    }
  },
  methods: {
    geklickt: function(){ this.zaehler++; }
  },
  template: '<p v-on:click="geklickt">Klick mich: {{ zaehler }}</p>'
});
```

Am Beispiel oben sieht man schon, dass der *template*-String schnell sehr lang wird. Glücklicherweise gibt es einige Alternativen, die [hier \(Strg-Klick für mehr\)](#) einzeln vorgestellt werden. Im Rahmen dieser Referenz werden wir Templateliterale [\(Strg-Klick für mehr\)](#) verwenden.

**Achtung:** Die *`Backticks`* der *Templateliterale* bringen so manchen Editor durcheinander und scheinen das Layout zu zerstören. Das ist nur eine Illusion, die aber ggf. irritiert.

```
Vue.component ( 'ausgabe', {
  data: function(){
    return {
      zaehler: 0
    }
  },
  methods: {
    geklickt: function(){ this.zaehler++; }
  },
  template: `

Die folgenden Schritte zum Anlegen einer Komponente sind recht einfach.



Als zweiten Schritt legen wir eine Verknüpfung zum HTML-Teil an, in dem die Komponente eingebunden wird. Z.B.:



```
new Vue({ el: '#ausgabe' })
```



Und im dritten Schritt binden wir die Komponente beliebig oft auf der Webseite ein:



Seite 188


```

```
<div id="ausgabe">
  <ausgabe></ausgabe>
  <ausgabe></ausgabe>
  <ausgabe></ausgabe>
</div>
```

## Komponenten kombinieren / verschachteln

Seine Muskeln kann Vue dann ausspielen, wenn wir das Konzept der Komponenten weiterdenken. Wenn wir nämlich im Beispiel oben die Überschrift auch als Komponente ausführen wollen, können wir das tun. Das hat den Vorteil, dass die Überschrift-Komponente beliebig oft wiederverwendet werden kann.

```
Vue.component ( 'ausgabe', {
  props: [
    'beschriftung',
    'header'
  ],
  template: `<div>
    <kopf :inhalt=header />
    {{ beschriftung }}
  </div>`
});

Vue.component ( 'kopf', {
  props: [
    'inhalt'
  ],
  template: `<h2>{{ inhalt }}</h2>`
});

new Vue({ el: '#ausgabe' });
```

```
<div id="ausgabe">
  <ausgabe></ausgabe>
  <ausgabe></ausgabe>
  <ausgabe></ausgabe>
</div>
```

## Datenübergabe mit Props

Wirklich hilfreich sind Komponenten vor allem, wenn wir auch Daten an die einzubindende Komponente übertragen können. Z.B., um den Inhalt oder eine Überschrift zu bestimmen, die in der Komponente angezeigt werden soll.

Dafür sind die Props da. Sie werden als Array angelegt, das mehrere Props bzw. deren Namen enthält.

```
Vue.component ( 'ausgabe', {
  props: [
```

```
    'beschriftung',  
    'header'  
  ],  
  template: `

<h1>{{ header }}</h1>  
    {{ beschriftung }}  
  </div>`  
});  
  
new Vue({ el: '#ausgabe' })


```

Beim Aufruf der Komponente werden die Props einfach als Attribute eingetragen:

```
<div id="ausgabe">  
  <ausgabe beschriftung="Der Inhalt" header="Überschrift"></ausgabe>  
  <ausgabe beschriftung="Hallo Welt" header="Header"></ausgabe>  
  <ausgabe beschriftung="abcdef" header="Oben"></ausgabe>  
</div>
```

# Andere Bibliotheken

Strg-Klick für mehr: Wichtigste Frameworks 2017

<p><b>Allrounder</b></p> <p><b>jQuery</b> Umfassende Manipulation des DOM <a href="http://jquery.org">http://jquery.org</a></p> <p><b>Prototype</b> Allrounder, hat eine treue Fangemeinde <a href="http://prototypejs.org">http://prototypejs.org</a></p> <p><b>Serverseitiges Scripting</b></p> <p><b>Node.js</b> – Serverseitiges Framework für Netzerkanbindungen <a href="http://nodejs.org">http://nodejs.org</a></p> <p><b>Optimierung</b></p> <p><b>Turbo</b> Einbindung der GPU zur Beschleunigung <a href="https://turbo.github.io/">https://turbo.github.io/</a></p> <p><b>Less</b> – Framework zum effizienteren Arbeiten mit CSS <a href="http://lesscss.org">http://lesscss.org</a></p> <p><b>Textbearbeitung</b></p> <p><b>Create.js</b> Interaktive Textbearbeitung im Frontend <a href="http://createjs.org">http://createjs.org</a></p> <p><b>Canvas</b></p> <p><b>Create.js</b> Umfangreicher, interaktiver Inhalt <a href="http://createjs.com">http://createjs.com</a></p> <p><b>Konva</b> Vereinfachung des Umgangs mit dem Canvas <a href="http://konvajs.github.io">http://konvajs.github.io</a></p> <p><b>Kute</b></p>	<p><b>WebApp-UI</b></p> <p><b>Dojo Toolkit</b> Webapp-Entwicklung <a href="http://dojotoolkit.org">http://dojotoolkit.org</a></p> <p><b>Angular</b> Interaktive Oberfläche ohne viel Javascript <a href="https://angularjs.org/">https://angularjs.org/</a></p> <p><b>React</b> Aufbau einer leistungsfähigen Oberfläche, v.a. für Single-Page-Anwendungen <a href="http://reactjs.de/">http://reactjs.de/</a></p> <p><b>Backbone</b> Strukturierung von Daten <a href="http://backbonejs.org/">http://backbonejs.org/</a></p> <p><b>Svelte</b> Erzeugt besonders wenig JS-Code <a href="https://svelte.technology/">https://svelte.technology/</a></p> <p><b>Vue.js</b> Angeblich leicht erlernbares Tool mit ähnlicher Funktionalität wie Angular oder React <a href="https://vuejs.org">https://vuejs.org</a></p> <p><b>Onsen UI</b> Speziell für Mobilgeräte ausgelegtes UI-Framework zum Erstellen von WebApps <a href="https://onsen.io/">https://onsen.io/</a></p> <p><b>Polymer</b> Components für Webapps <a href="https://www.polymer-project.org/">https://www.polymer-project.org/</a></p>
	<p><b>Darstellung</b></p> <p><b>PDF.JS</b> Bibliothek, um PDF zu parsen und zu rendern <a href="http://mozilla.github.io/pdf.js/">http://mozilla.github.io/pdf.js/</a></p> <p><b>Three.js</b> Bibliothek, um 3D-Objekte darzustellen <a href="http://threejs.org">http://threejs.org</a></p>

<p>Animations-Engine <a href="http://thednp.github.io/kute.js/">http://thednp.github.io/kute.js/</a></p> <p><b>Charts</b></p> <p><b>D3</b> Umfangreiches, aber aufwendiges Tool zur Datenvisualisierung <a href="https://d3js.org/">https://d3js.org/</a></p> <p><b>Plotly</b> Vereinfacht den Umgang mit D3 <a href="https://plot.ly/javascript/">https://plot.ly/javascript/</a></p> <p><b>Go JS</b> Diagramme in Javascript, kostenpflichtig <a href="http://Gojs.net">http://Gojs.net</a></p> <p><b>Inhalte/Widgets</b></p> <p><b>fullCalendar</b> Vollständiger Kalender mit Drag'n'Drop <a href="https://fullcalendar.io/">https://fullcalendar.io/</a></p> <p><b>MathJS</b> Umfangreiche Bibliothek mit Parser, großen, komplexen Zahlen, Brüchen, Einheiten und mehr. <a href="http://mathjs.org/">http://mathjs.org/</a></p> <p><b>Mobil</b></p> <p><b>Hammer</b> Touch-Gesten für Webapps: Pinch, Pan, Rotate, Swipe, Tap, Press <a href="http://hammerjs.github.io/">http://hammerjs.github.io/</a></p>	<p><b>Masonry</b> Grid Layouts <a href="http://masonry.desandro.com">http://masonry.desandro.com</a></p> <p><b>Push.JS</b> Erzeugen von Push-Notifications <a href="https://pushjs.org/">https://pushjs.org/</a></p> <p><b>P5</b> Erlaubt das Zeichnen auf dem Bildschirm. <a href="https://p5js.org/">https://p5js.org/</a></p> <p><b>Intense Images</b> Sehr schöne Vollbildfunktion, um Bilder genauer anzusehen <a href="http://tholman.com/intense-images/">http://tholman.com/intense-images/</a></p> <p><b>Embed.js</b> Bibliothek, um Inhalte einfach einzubinden <a href="http://riteshkr.com/embed.js/">http://riteshkr.com/embed.js/</a></p> <p><b>Sweetalert</b> Schöne Alertboxen <a href="https://limonte.github.io/sweetalert2/">https://limonte.github.io/sweetalert2/</a></p> <p><b>Turntable</b> Erzeugen von Pseudo-3D durch Aneinanderreihung von Bildern. <a href="http://polarnotion.github.io/turntable/">http://polarnotion.github.io/turntable/</a></p> <p><b>fullPage</b> Erleichtert des Scrollens von einem Seiteninhalt zum nächsten <a href="https://alvarotrigo.com/fullPage/">https://alvarotrigo.com/fullPage/</a></p> <p><b>MathJax</b> Darstellen von mathematischen Formeln und Schreibweisen auf Basis von MathML <a href="https://www.mathjax.org/">https://www.mathjax.org/</a></p>
	<p><b>Engines</b></p> <p><b>Phaser</b> Vollständige Gameengine <a href="http://phaser.io">http://phaser.io</a></p> <p><b>Crafty</b> Javascript Game Framework <a href="http://craftyjs.com/">http://craftyjs.com/</a></p>

und Hunderte, wenn nicht Tausende mehr ...



# Anhang: Veraltete Browser

Die Frage nach den Browsern ist auch immer die Frage nach den Programmierrichtlinien.

Jede Firma und häufig sogar jedes Projekt hat Programmierrichtlinien, in diesen festgehalten wird,

- welche Sprache zu verwenden ist,
- welche Browser zu unterstützen sind,
- wie Variablen benannt werden sollen,
- welche Schreibweise für Funktionen zu verwenden ist,
- wie die Dateistruktur zu sein hat,
- etc., pp.

In diesen Programmierrichtlinien sollte auch festgeschrieben sein, wie mit veralteten Browsern, namentlich Internet Explorer, umzugehen ist.

Anhand dieser Angabe entscheidet sich, welche Polyfills verwendet werden und ob vielleicht sogar eine Preprocessor-Sprache verwendet wird.

## Polyfills

Mit Polyfill ist ein Tool gemeint, dass moderne Features, die möglicherweise in älteren Browsern nicht unterstützt werden, in die älteren Schreibweisen ummünzt. Dabei ist natürlich zu beachten, dass sich die älteren Schreibweisen ggf. auch im Verhalten von den modernen Varianten unterscheiden.

Z.B. verhält sich eine `function x() {}` anders als `(x) => {}`.

## Selbstgeschrieben

### Polyfill-Datei

Um selbst einen Polyfill zu schreiben, gibt es mehrere Möglichkeiten.

Zunächst empfiehlt es sich, für die Polyfills eine eigene Javascript-Datei zu erzeugen und diese einzubinden. Da eigentlich nur der Internet Explorer Probleme mit ES6 hat (immer noch ~5% aller Besucher), kann das Einbinden der Polyfill-Bibliothek mithilfe von Conditional Comments auf den Internet Explorer beschränkt werden:

```
<!--[if IE ]>
  <script type="text/javascript" src="meinPolyfill.js"></script>
<![endif]-->
```

Bitte denk daran, die Polyfill-Datei als erstes einzubinden.

In dieser Datei kann nun für jede ES6-Funktion, die im Programm verwendet wird, ein Ersatz definiert werden. Als Quelle für diese Workarounds leistet z.B. das Mozilla Developer Network (MDN) gute Dienste. Ein Beispiel:

[https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global\\_Objects/Array/from#Polyfill](https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Array/from#Polyfill)

Der Code, der unter dem Reiter Polyfill angezeigt wird, kann einfach in die Polyfill-JS-Datei eingebunden werden und es funktioniert auch im IE.

Andere Polyfills finden sich hier:

<https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills>

**Vorsicht:** Bitte teste die Funktionen ausgiebig, ob sie sich wirklich so verhalten wie sie sollen.

### Arrow-Funktionen, Const, etc.

Die obigen Polyfills funktionieren leider nur mit Methoden, die zu existierenden Objekten/Prototypen gehören. Befehle oder Konstrukte wie `const`, `let` oder Arrow-Funktionen lassen sich mit solchen einfachen Polyfills nicht ersetzen. Diese sollten von vornherein vermieden werden, wenn die Kompatibilität mit dem IE wichtig ist.

### Alternative Javascripte

Sollte die Polyfill-Methode aus irgendeinem Grund nicht funktionieren, dann besteht natürlich auch die Möglichkeit, im Internet Explorer (der Edge-Browser funktioniert tadellos) auf eine alternative Javascript-Datei zurückzugreifen, in der die fraglichen Funktionen durch IE-kompatible Versionen überschrieben werden.

### Vergiss den IE

Es gibt sogar eine wachsende Anzahl Entwickler, die Benutzer des IE (ggf. unter einer bestimmten Version) komplett von der Benutzung der Webseite / WebApp ausschließen. Wer die Webseite verwenden will, soll auf einen neuen Browser wie z.B. Edge, Firefox oder Chrome ausweichen. Das würde ich natürlich auf einer Kundenseite nicht machen, aber für private Projekte ist das eine Alternative, die viele Nerven spart.

### Fertige Polyfills

Es gibt eine Handvoll fertiger Polyfill-Bibliotheken. Diese sind sicher auch einen Blick wert:

<https://polyfill.io/v2/docs/features/>

<https://www.npmjs.com/package/js-polyfills>

## Compiler

Ja, es gibt die Möglichkeit, Javascript zu kompilieren.

Hier tut sich insbesondere Babel (<https://babeljs.io/>) oder Traceur (<https://github.com/google/traceur-compiler>) hervor.

Dieser Compiler wird auf dem Server oder in der Entwicklungs-Pipeline installiert und sorgt dafür, dass der eingegebene Code zu ES5 umgewandelt wird und damit in allen Browsern funktioniert.

Damit kauft man sich natürlich das Problem ein, dass sich die Funktionen im kompilierten Skript ggf. anders verhalten als man es geplant hatte. Denn z.B. eine klassische Funktion verhält sich halt anders als eine Arrow-Funktion.

Daher sollte man sich den Einsatz eines Compilers gut überlegen und den erzeugten Code intensiv testen.

# JS Preprocessors / Transpiler

Es gibt eine Handvoll Sprachen, die erfunden wurden, um neue Programmier Techniken zu verwenden und deren Code in reines Javascript übersetzt (Transpiling) wird.

Einige Beispiele wären:

- Coffeescript
- Typescript
- JSX
- Livescript
- U.v.m.

Für diese Sprachen wird in der Entwicklungs-Pipeline oder auf dem Server ein Transpiler installiert, der dem Browser reines Javascript ausliefert. Der Vorteil dieser Lösungen ist, dass der Entwickler sich auf moderne Programmier Techniken konzentrieren kann und der Transpiler den Rest macht.

Der Nachteil ist natürlich, dass sich der Entwickler darauf verlassen muss, dass der Code aus dem Transpiler genau das macht, was vorgesehen war. Auch hier gilt, wie beim Compiler, dass sich die eine oder andere Programmier Technik (z.B. Arrow-Funktionen) anders verhalten als ES5-Javascript

JS-Preprocessors bei Codepen:

<https://blog.codepen.io/documentation/editor/using-js-preprocessors/>

# Umgang mit Fehlern und Problemen

Fehler passieren, das ist ganz normal. Daher gibt es auch ganze Bücher darüber, wie Fehler gefunden werden können. Es gibt aber einen ganz einfachen Trick, um Fehler zu vermeiden:

## Halte Deinen Code sauber.

"Sauber" bedeutet:

Benutze **Kommentare**, um die Codeabschnitte zu unterteilen.

**Rücke** den Inhalt von Codeblöcken **ein**:

```
function meineFunktion() {  
    var x = prompt('Ihre Eingabe');  
    if ( x == 'Hallo Welt' ) {  
        alert ( 'irgendwas' );  
    }  
}
```

## Häufig auftretende Fehler

Bei der Fehlerbehandlung (in so ziemlich jeder Programmiersprache) dreht es sich im Wesentlichen darum, Zustände zu vermeiden, in denen das Programm einen Fehler produziert. Dazu muss man erst einmal die möglichen Fehlerursachen kennen und wissen, wie man sie umgehen kann:

### DOM überschreiben

Es ist verführerisch, sollte aber mit Vorsicht genossen werden: Das Anhängen von DOM-Inhalten auf diese Weise:

```
var element = document.querySelector ('#element');  
element.innerHTML += 'Hallo Welt';
```

Der += Operator nimmt den ursprünglichen Inhalt, hängt den neuen Inhalt an und schreibt das Ergebnis in das Element. Wenn das Element aber bereits DOM-Knoten enthält, dann werden dummerweise weder Eventlistener noch Values mitkopiert.

D.h. Kindelemente in dem Element verlieren ggf. vorhandene Eventlistener und Values.

### Endlosschleifen

Zwar bieten die meisten Interpreter einen automatischen Abbruch, wenn eine Schleife ungewöhnlich lange brauchen (viele Sekunden bis Minuten). Unter Umständen ist es aber auch eine gute Idee, in eine Schleife einen zusätzlichen Mechanismus zu integrieren, der die Schleife nach einer bestimmten Anzahl von Durchläufen (oder auf Basis anderer Bedingungen) mit Gewalt abbricht.

```
var z = 0;  
for (var i = 0; i < 100; i++) {  
    // Hier wird i irgendwie wieder auf 0 zurückgesetzt  
    z++;  
    if (z > 5000) {
```

```
        // Fehlerbehandlung (z.B. Fehlermeldung)
        break;
    }
}
```

### Leere Variable

Es kann vorkommen, dass an einer bestimmten Stelle eine Variable, die verwendet werden soll, keinen Wert enthält. Z.B. wenn in einem Prompt-Fenster der Abbrechen-Knopf gedrückt wurde oder eine Variable angelegt aber nicht mit einem Wert belegt wurde.

Der einfachste Weg, dies abzufangen, ist dieser:

```
var x;
if( !x ){
    // Fehlerbehandlung (z.B. Fehlermeldung)
} else {
    // Operation
}
```

Alternativ kann auch auf undefined geprüft werden

```
var x;
if(x == undefined){
    // Fehlerbehandlung (z.B. Fehlermeldung)
} else {
    // Operation
}
```

Man spricht bei einer leeren Variablen auch von NULL. Gemeint ist nicht die Zahl 0, sondern ein vorgefertigtes Objekt, das von Javascript bereitgestellt wird. Es sagt aus, dass hier "nichts" ist.

```
var x;

if ( x == null ){
    // Fehlerbehandlung (z.B. Fehlermeldung)
} else {
    // Operation
}
```

### Nichtexistente Variable

Wenn eine Variable nicht nur leer ist, sondern noch nicht einmal angelegt wurde, dann bricht der Programmablauf ab. Um dies zu vermeiden, kann man auch die Existenz der Variable überprüfen

```
if(typeof x == 'undefined') {
    // Fehlerbehandlung (z.B. Fehlermeldung)
} else {
    // Operation
}
```

```
}
```

## Not a Number

JavaScript versucht, Variablen in den richtigen Datentypen zu übertragen, falls dies notwendig ist. Das gelingt aber nicht immer und wenn ein Script bei einer Rechnung oder Typumwandlung keine Zahl vorfindet, dann gibt es das Ergebnis Not a Number (*NaN*). Dieses wird auch ggf. in die Variable eingetragen. Um dies zu umgehen, kann eine Variable auf verschiedenen Weisen daraufhin überprüft werden, ob es sich um eine Zahl handelt.

```
var x = 'Mein String';
if( isNaN(x) ) {
  // Fehlerbehandlung (z.B. Fehlermeldung)
} else {
  // Operation
}
```

## Rundungsfehler

Für den Computer sind die Zahlen 1 und 1.0000000000000001 vollkommen unterschiedlich. Manchmal – besonders bei aufwendigen Rechnungen mit Sinus, Wurzel, etc. - können sich solche Rundungsfehler einschleichen. Die Probleme tauchen dann meist bei Vergleichsoperationen oder Schleifen auf. Hier gibt es einen einfachen Weg, um Fehler zu umgehen.

```
// Gerundete Zahlen vergleichen
if( Math.round( x ) == 0 ){
  // Operation
}
```

## Zu weit gezählt

In Schleifen kann es vorkommen, dass zu weit gezählt wird. Sei es, dass ein Rundungsfehler auftaucht, oder dass ein Wert übersprungen wird. Daher vermeiden erfahrene Programmierer bei einer Schleife den `==`-Operator.

```
for ( var i = 0; i >= 100; i += 3 ){
  // Operationen
}
```

## Try ... catch

Dies ist ein recht einfacher und übersichtlicher Weg, mit möglichen Fehlern umzugehen. Ein *try-catch*-Konstrukt reagiert selbständig darauf, ob eine Operation in einen Fehler läuft.

```
'use strict';

try {
  console.log ( x * y );           // Variablen wurden nicht angelegt
}
```

```
} catch (err) {  
  console.log ( 'problem: ', err);  
  // -> problem : ReferenceError: x is not defined ...  
}
```

Der Befehl im Try wird ausgeführt. Nach Ausführung wird geprüft, ob es einen Fehler gegeben hat. Falls ja, wird der Catch-Teil ausgeführt

Eine beliebige Anwendung ist auch das Prüfen der Gültigkeit von Daten:

```
function isValidJSON(text) {  
  try {  
    JSON.parse(text);  
    return true;  
  } catch {  
    return false;  
  }  
}
```

Finally  
Verschachtelt

# Anhang: Hilfsmittel

Die vielfältige Welt der Frontend-Entwicklung wird durch einige Hilfsmittel erleichtert. Dies sind z.B. Webseiten mit der einen oder anderen Hilfestellung oder auch Funktionen in den verwendeten Programmen.

## Editoren

<http://brackets.io/>

Ein quelloffener Texteditor, der auf die Entwicklung von HTML/CSS/JS spezialisiert ist.  
Meine Empfehlung für die Entwicklung von Webinhalten (HTML, CSS, JS, PHP, SQL, XML, etc.)

<https://krita.org/en/features/highlights/>

Ein quelloffenes Grafikprogramm mit Fokus auf künstlerische Arbeit.

## Inhalte generieren

<http://blindtextgenerator.de>

Erzeugen verschiedenartiger Blindtexte.

<http://lorempixel.com>

Erzeugen von Füllbildern aus einem Pool verschiedener Themen.

<http://placeholder.it/>

Erzeugen von generischen Platzhalterbildern mit benutzerdefinierter Größe.

<http://bgpatterns.com>

Erzeugen von nahtlos kachelbaren Hintergrundbildern.

<http://css3generator.com>

Erzeugen des Quellcodes für CSS3-Effekte inklusive Vendor-Specific-Prefixes, wenn nötig.

<http://www.favicon-generator.org/>

FavIcon (Icon für Tab und Bookmarks) erzeugen.

<http://keycode.info/>

Ein kleines Tool, um den ASCII-Code einer gedrückten Taste auszugeben.

<http://myjson.com>

Hier können eigene JSON-Daten abgelegt und unter einer URL abgefragt werden. Ein wunderbares Tool, um AJAX-Abfragen zu üben.

**Bonus:** Die Daten aller anderen Benutzer sind lesbar. Durch URLs mit zufälligem, mindestens vierstelligen Ende (z.B. <https://api.myjson.com/bins/v5196>) können also quasi-zufällige JSON-Daten geladen werden.

## Bildquellen

<http://www.textures.com>

Portal zum Download von Texturen/Patterns. Grundsätzlich kostenlos, benötigt Anmeldung.

<https://pixabay.com/de/>

Portal zum Download von Bildern, die keiner Lizenz unterliegen ( CC 0 )

## Bildbearbeitung



<https://pixlr.com/editor/>

Überraschend leistungsfähiges Bildverarbeitungsprogramm (auf Basis von Flash)

<https://www.picozu.com/>

Bildbearbeitungsprogramm auf Basis von HTML5

## Farben

<http://color.adobe.com>

Tool von Adobe, um aus Basis von Algorithmen Farbklänge zu erzeugen.

<http://hslpicker.com>

Colorpicker für das HSL(A)-Farbsystem.

<http://www.colorzilla.com/gradient-editor/>

CSS-Farbverläufe generieren

<http://html-color.org/names>

Liste aller Farbnamen und deren Entsprechung in RGB und HSB

## Schriften

<https://www.fontsquirrel.com/>

Sammlung kostenfreier Webfonts und Umwandlung vorhandener Schriften zu Webfonts

<https://fonts.google.com/>

Sammlung kostenfreier Webfonts und Hosting der Schriften. Können mit einem einfachen link-Tag in die Seite integriert werden.

## SEO

<http://spritegen.website-performance.org/>

Tool, um aus Einzelbildern Sitemap zu generieren. Hilfreich für CSS-Sprites und Canvas-Animationen.

<http://fontawesome.io/>

Sammlung von hunderten Icons in Form einer Schrift-Datei.

## Validation

<http://esprima.org/demo/validate.html>

Javascript Syntax Validator

## Foren, Communities, Referenz

<http://stackoverflow.com>

DAS Forum für Entwickler, wenn der Code mal nicht so will wie er soll.

<http://caniuse.com>

Prüfung, in welchem Browser ein bestimmter Code interpretiert wird und in welchem nicht.

<http://youtube.com>

Anleitungen, Tutorials, etc.

## Marktübersicht

<https://www.similartech.com/>

Übersicht über die Anzahl der Webseiten, die eine bestimmte Technologie (Node.JS, React, etc.) verwenden.

Inspiration

<https://js.libhunt.com/>

Hosting

<https://www.heroku.com/home>

Hosting-Anbieter mit kostenlosem Anfänger-Account und vielen interessanten Optionen.

# Anhang: Weiterführende Literatur

<https://pandoc.org/>

## You don't know JS

kostenloses, digitales "Buch" auf Github, das mit einigen Missverständnissen zu Javascript aufräumt.

<https://github.com/getify/You-Dont-Know-JS>

## Domenic Denicolas Blog

Ein interessanter Blog, der zwar seit 01/16 nicht mehr gepflegt wird. Die Themen dort sind aber trotzdem einen Blick wert.

<https://blog.domenic.me/>

## Studyguide zum Google mobile-web-specialist

[https://developers.google.com/training/certification/mobile-web-specialist/StudyGuide\\_MobileWebSpecialist.pdf](https://developers.google.com/training/certification/mobile-web-specialist/StudyGuide_MobileWebSpecialist.pdf)

## ExploringJS

Freies "Buch" zum Thema Javascript.

<http://exploringjs.com/>

## JavaScript: The Good Parts

<https://www.pluralsight.com/courses/javascript-good-parts>

## Youtube-Playlisten

<https://www.youtube.com/watch?v=UeZi8a99iS0&list=PLNmsVeXQZj7qOfMI2ZNk-LXUaiXKrwDli>

<https://www.youtube.com/watch?v=nVMIDghNcHg&list=PLNmsVeXQZj7onyAB57T0xqV2ZVSZOo79a>

<https://www.youtube.com/watch?v=IL8ZMMjc3P0&list=PLNmsVeXQZj7rkk6Urp1zdIqOVWanXsVtk>

[https://www.youtube.com/watch?v=r4u\\_6OPHiMo&list=PLNmsVeXQZj7qNMn6zimfu4JPuklG-4Uu4](https://www.youtube.com/watch?v=r4u_6OPHiMo&list=PLNmsVeXQZj7qNMn6zimfu4JPuklG-4Uu4)

[https://www.youtube.com/watch?v=q\\_4OrvVjWhA&list=PLNmsVeXQZj7rNukSNOVkUnNdhKNZ4sNfA](https://www.youtube.com/watch?v=q_4OrvVjWhA&list=PLNmsVeXQZj7rNukSNOVkUnNdhKNZ4sNfA)

# Anhang: Glossar

Hier eine Zusammenfassung einiger, wichtiger Begriffe.

Diese Liste ist nicht vollständig:

- **Arrays** – Nummerierte Liste von Variablen.
- **Artefakte** – Ungewollte sichtbare Überreste eine Bildbearbeitung (z.B.: JPG-Kompression, Bilddrehung, etc.)
- **ASCII, ASCII-Tabelle:** Allgemein gültige Tabelle, in der allen Zeichen eine bestimmte Nummer zugewiesen ist. Dient z.B. dazu, Zeichen anzusprechen, die sich nicht auf der Tastatur befinden, weil sie in der Landessprache nicht existieren.
- **Bibliothek** – Sammlung von Code, der die Arbeit erleichtert und/oder bestimmte Methoden fertig zur Verfügung stellt.
- **Boolscher Wert** – Eine Wahrheitsaussage, kann nur wahr (true) oder falsch (false) sein
- **By reference** – siehe *Referenzieren*
- **By value** – siehe *Referenzieren*
- **CamelCaps** - Besondere Schreibweise von Bezeichnern, in alle Buchstaben, die einen alleinstehenden Begriff beginnen, groß geschrieben werden. Z.B.: `MeineVariable`
- **Deklarieren** – von engl. 'to declare'. Einen Wert bestimmen und (meist) in einer Variablen speichern.
- **Event** – Ereignis wie z.B. Mausklick, das Überfahren eines Elementes mit der Maus, Beendigung des Ladevorganges, etc..
- **EventListener / Eventhandler** – Prozess, der auf das Auftreten eines bestimmten Ereignisses wartet. Tritt das Ereignis auf, wird eine dem Eventhandler zugewiesene Funktion gestartet.
- **Float** – Fließkommazahlen: Zahlen mit bis zu 15 Nachkommastellen und ggf. einem Vorzeichen. In JS sind Integer und Float als Number zusammengefasst.
- **Funktionen** – Programmteile, die von beliebiger Stelle des Programmes aus aufgerufen werden können
- **Index** – (Mz.: Indizes oder Indexe) Fortlaufende Nummer, um ein Element eindeutig zu benennen.
- **Indizieren** – Elemente fortlaufend nummerieren.
- **InnerCaps** – Besondere Schreibweise von Bezeichnern, in der der erste Buchstabe klein und in der Folge alle Buchstaben, die einen alleinstehenden Begriff beginnen, groß geschrieben werden. Z.B.: `meineVariable`
- **Integer** – Ganze Zahlen ohne Vorzeichen. Ist der einfachste Zahlen-Typ. In JS sind Integer und Float als Number zusammengefasst.
- **Intervall / Interval** – Wiederholter Aufruf einer Funktion nach einer definierten Zeit.
- **Iterieren / Iteration:** Die Speicherstellen eines Arrays/Objekts/Map nacheinander auslesen und zur Verfügung stellen. Z.B. über `Array.forEach`.
- **Klassen** – Konstruktoren mit
- **Konstruktoren** – Spezielle Funktion, die ein schnelles Anlegen vieler Objekte gleicher Art ermöglicht
- **Literal** – Eine Zeichenfolge, die in einer Variablen gespeichert wird.
- **Methoden** – Funktionen, die innerhalb von Objekten angelegt werden
- **NaN** – Abkürzung für Not a Number
- **NodeJS** – Framework, dass es erlaubt, auf dem Server mit Javascript zu arbeiten

- **Nodelist** – Nummerierte Liste von DOM-Elementen. Im Aufbau ähnlich einem Array, aber mit geringerem Funktionsumfang.
- **Objekte** – Variablentyp, der es erlaubt, viele Werte mit dazugehörigem Namen zu speichern
- **Paradigma** – Genauer: Programmierparadigma ist eine grundsätzliche Herangehensweise. Beispiele: Modulare Programmierung, Funktionale P., Objektorientierte P., etc.
- **Parameter** – Werte, die an eine Funktion übergeben und dort wie Variablen verwendet werden können.
- **Prototype** – Vorlage oder Blaupause für ein Objekt. Siehe Konstruktor.
- **Prozedur** – Eine Funktion ohne Rückgabewert wird manchmal Prozedur genannt. Die Definition variiert aber zwischen verschiedenen Programmiersprachen.
- **React** – Javascript-Bibliothek zur Darstellung einer UI
- **Referenzieren** – Verknüpfen. Wenn eine Variable *byReference* gesetzt wird, dann bedeutet das, dass bei einer Änderung der Variable das Original verändert wird, das in der Variablen referenziert ist. Im Gegensatz dazu wird bei *byValue* eine Kopie erzeugt und das Original bleibt unangetastet.
- **String** – siehe Zeichenkette.
- **Stylesheet** – Zusammengefasste Beschreibung der Darstellungsstile.
- **Syntax** – Die Art, Dinge/Befehle zu beschreiben.
- **Typecasting** – Umwandlung von einem Datentypen in einen anderen.  
Z.B. `Number('1000');`
- **Variable** – Veränderbarer Speicherbereich. Siehe entsprechender Bereich dieser Doku.
- **XML** – Dateiformat, das Node(Knoten)-basiert Daten zur Verfügung stellt.
- **Zeichenkette** – Eine Menge von hintereinanderstehenden Zeichen (Zahlen, Buchstaben, Satzzeichen).

# Anhang: Jobplattformen

Neben den Plattformen in der Liste vom FTP-Server gibt es noch einige weitere.

Da jene Liste sehr aufwändig ist und die spannenden Plattformen nicht besonders hervorgehoben sind, hier ein paar Vorschläge abseits der großen Plattformen (Jobagentur, Stepstone, Monster).

## Honeypot

Eine europaweit agierende Jobplattform, die vor allem auf Entwickler – also Dich - fokussiert ist.

Das Besondere daran ist, dass Du Dein Profil hochlädst und die Unternehmen sich bei Dir bewerben.

<https://www.honeypot.io>

## Xing / linkedIn

Für uns als Webworker mit dem Internet als natürlichen Lebensraum sind Social Media Plattformen das Tool der Wahl, um einen Job / Kontakte / Auftraggeber zu finden.

Lege unbedingt ein Profil auf den genannten Plattformen an oder, falls bereits geschehen, halte dieses aktuell und möglichst vollständig. Beide Plattformen werden regelmäßig von Talentscouts und Headhuntern durchsucht und so kann relativ schnell ein Kontakt entstehen.

Xing zumindest hat dazu noch eine sehr hilfreiche Jobsuche, die sich aus den Daten Deines Profils füttert und entsprechen gute Ergebnisse liefert.

Die Möglichkeit, von Events zu erfahren und dort Kontakte zu knüpfen, darf auch nicht vernachlässigt werden.

Der Unterschied zwischen beiden Anbietern ist im Wesentlichen, dass Xing v.a. im deutschsprachigen Raum etabliert ist. linkedIn dagegen ist eher global ausgerichtet.

<https://www.xing.com>

<https://www.linkedin.com/>

## T3N

Eine der spannendsten Zeitschriften im Bereich der Webentwicklung hat eine eigene, im Markt fest etablierte Jobplattform. Hier finden sich Jobs aus den Bereichen Entwicklung, Technik, Design, Projektmanagement, Marketing und PR.

Der Fokus liegt dabei klar auf Entwicklung.

<https://t3n.de/jobs/>

## Jobmessen

Auf einer Jobmesse hat man die Möglichkeit, direkt mit Jobanbietern in Kontakt zu treten. Ganz ohne den sonst üblichen Aufwand kann man hier sofort einen guten Eindruck hinterlassen.

**Pro-Tipp:** Unbedingt reichlich Kurzbewerbungen (Max. eine A4 Seite) und Visitenkarten in der Tasche haben.

Z.B.: <http://www.jobfair.de/index.php>

## Regional

Viele Unternehmen suchen auch regional nach Mitarbeitern. Hintergrund ist, dass ein Mitarbeiter, der nicht erst umziehen muss, schneller eingestellt werden kann und sich nicht erst eingewöhnen muss.

z.B.: <https://jobs.meinestadt.de/>



Nicht im IE bedeutet, dass diese Funktion/Technologie im Internet Explorer nicht zuverlässig unterstützt wird.

Dabei wird nicht nur die "aktuellste" Version des IE berücksichtigt, sondern auch ältere bis zurück zur Version 9. Wenn der IE9 diese Technik also nicht beherrscht, dann bekommt das Thema den Marker. Der Grund ist, dass Behörden, Krankenkassen u.ä. immer noch auf diese Steinzeittechnik setzen, da es als aufwendig gilt, einen neuen Browser auf seine Sicherheit zu prüfen – nach über sieben Jahren!

Generell verbreitet sich in der Entwicklergemeinde immer mehr die Ansicht, dass der Internet Explorer ruhigen Gewissens ignoriert werden kann. Die Gründe:

- Der Browser wird nicht mehr aktualisiert (letzte offizielle Version ist von 2013)
- Deswegen ist er ein offenes Scheunentor für Hackerattacken jeder Art
- Die Art der Darstellung entfernt sich immer mehr von aktuellen Browsern
- Er ist laaangsaaam
- Er unterstützt keine modernen Programmier Techniken
- Wenn mehr und mehr Seiten einfach nicht richtig funktionieren, wird ein Leidensdruck aufgebaut, der hoffentlich mehr und mehr Benutzer zu richtigen Browsern bewegt.

Selbst Microsoft versucht mit einer Multimillionendollar-Kampagne, die Benutzer auf den aktuellen Edge-Browser zu bewegen. Immerhin.

Trotzdem hat der Internet Explorer immer noch eine Marktdeckung zwischen 2% und 8%, je nachdem, wer gerade misst. Tendenz fallend.

## In der Praxis

Üblicherweise wird für ein Projekt geschaut, welche Art von Benutzer vermutlich angezogen wird. Sollten Liebhaber veralteter Software (Banken, Versicherungen, Behörden, Senioren, etc.) erwartet werden, so muss mit dem Problem *IE* umgegangen werden.

Man kann entweder so programmieren, dass die Seite im IE funktioniert.

Oder man zeigt dem IE-Benutzer einen Hinweis, dass er doch bitte zeitgemäße Technik einsetzt.

Oder beides.

Einen IE erkennt man recht leicht mithilfe von Conditional Comments.

```
<!--[if 'IE']>

<div class="modal">
  Bitte aktualisieren Sie Ihre Software.<br>
  <a href="https://www.microsoft.com/de-de/windows/microsoft-
edge">Edge</a><br>
  <a href="https://www.mozilla.org/de/firefox/">Firefox</a><br>
  <a href="https://www.google.de/chrome">Chrome</a>
</div>
```



```
<![endif]-->
```

Diese können auch nach Versionen auflösen. So kann z.B. ein IE erkannt werden, der älter ist als Version 9:

```
<!--[if lt IE 9]>  
    // ...  
<![endif]-->
```

# Index

3D .....	170	Browserfenster .....	73	<i>else if</i> .....	27
4095 Byte.....	71	<b>By reference</b> .....	182	<b>Embed</b> .....	171
Abgekürzte Eigenschaften	63	<b>By value</b> .....	182	empfangen .....	99
Ablaufdatum .....	71, 72	call .....	50	Endwinkel .....	109
Adobe .....	179	<b>CamelCaps</b> .....	182	entfernen .....	65
Ajax .....	80	Canvas .....	104	Envelope.....	86
AJAX .....	156	Case .....	27	<b>Event</b> .....	182
<b>Ajax-Objekt</b> .....	80	Chaining .....	76	<b>EventHandler</b> .....	182
Alert .....	23	Charts .....	170	<b>EventListener</b> .....	182
allow-file-access-from-files		Checkbox .....	24	<b>Event-Listener</b> .....	8
.....	84	Colorpicker .....	179	<i>Express</i> .....	145
Angular .....	135, 170	Conditional Comments..	186	extends .....	51
AngularJS .....	134	Confirm .....	23	eXtensible Markup	
AngularJS und Ajax .....	134	<i>continue</i> .....	31	Language .....	96
Animieren .....	112	Cookies .....	71	<b>Externe Dateien</b> .....	8
Architektur.....	85	<b>Create</b> .....	170	Farbfüllung .....	104
Arguments .....	35	<b>Create.js</b> .....	170	Farbverläufe .....	179
Arguments-Objekt .....	35	createClass .....	150	Favicon .....	178
Array .....	36	Cross Origin Policy .....	84	fetch .....	156
Arrays .....	85, 182	<b>D3</b> .....	170	Fill .....	107
Arrow .....	37	Darstellung .....	170	fillText.....	109
<b>Artefakte</b> .....	182	Dateibearbeitung .....	80	Filter .....	129
<b>ASCII</b> .....	182	Dateiformat .....	96	<b>Float</b> .....	182
<b>ASCII-Tabelle</b> .....	182	Dateizugriff .....	142	font .....	109
<b>async</b> .....	9	Datentyp .....	12, 19	For .....	29
asynchron .....	81	Date-Objekte .....	37	<b>forEach</b> .....	40
asynchrone .....	75	Debugger .....	72	<b>for-in</b> .....	40
Asynchrone Operationen	75	<b>defer</b> .....	9	Formen .....	107
Asynchronität .....	81	<b>Deklarieren</b> .....	182	Formularfelder .....	23
außerhalb der Funktion...	35	der Reihenfolge nach .....	34	Forum .....	179
<b>Backbone</b> .....	170	Diagramme .....	170	Frontend.....	149, 150
Bedingung.....	29	disable-web-security .....	84	Füllbilder .....	178
beginPath .....	107	Do .....	30	<b>fullPage</b> .....	171
Benutzereingaben .....	23	<b>Dojo</b> .....	170	Füllstil .....	105
<b>Bibliothek</b> .....	182	Do-While .....	30	Funktion .....	37
Bibliotheken .....	170	Download .....	178	Funktionen .....	33, 182
Bildausschnitt .....	111	Eckpunkt.....	108	Funktionsaufruf .....	34
Bildbearbeitung .....	178	ECMA .....	6	Funktionsname.....	34
Bilder .....	110	ECMAScript.....	6	fußgesteuert.....	30
Bildverarbeitungsprogramm		ECMA-Script 2015 .....	37	Geschützte Attribute.....	52
.....	179	ECMAScript2016.....	6	GET .....	52, 81
Blindtexte .....	178	Editoren .....	178	getComputedStyle.....	63
<i>body-parser</i> .....	145	Eigene Module .....	139	Getter .....	53
BOM.....	7	Eine Ausführung		Gettern .....	52
<b>Boolean</b> .....	12	überspringen .....	31	Glossar.....	182
<b>Boolscher Wert</b> .....	182	Element .....	97	<b>Go JS</b> .....	170
<i>break</i> .....	27, 30	<i>else</i> .....	27	GPU .....	170

Grid Layouts.....	170	Math .....	14	<b>readyState</b> .....	81
Hack .....	24	Math-Objekt .....	14	Rechteck .....	107
Hilfsmittel .....	178	<b>mehrfach</b> .....	83	rect .....	107
Hoisting.....	21	mehrfach ausführen.....	29	<b>Referenzieren</b> .....	183
<i>Hook</i> .....	165	<b>Methoden</b> .....	182	<i>reject</i> .....	75
HTML .....	7	Millisekunden .....	37	Rekursion .....	36
http-Anfrage .....	80	Module .....	130, 137	render.....	150
http-Protokoll .....	91	Muster .....	106	Requestheader .....	81
<i>HTTP</i> -Protokoll .....	99	nachladen .....	80	RequestHeader.....	84
IE 186		<b>NaN</b> .....	182	<i>resolve</i> .....	75
If 27		Neue Option .....	26	Response .....	82
Image-Objekt.....	110	Nicht im IE .....	186	responseText.....	82
Index .....	25, 182	<b>Node</b> .....	170	responseXML.....	82
<b>Indizieren</b> .....	182	<i>nodeJS</i> .....	145	REST.....	85
Inhalt .....	25	NodeJS.....	91, 99, 136	return .....	36
<b>Inline</b> .....	8	<b>NodeList</b> .....	182	Richtung .....	109
<b>InnerCaps</b> .....	182	npm .....	138	<b>Ruby</b> .....	136
<b>Integer</b> .....	182	<b>Number</b> .....	12	Rückgabewert .....	183
<b>Intense Images</b> .....	171	Objekt.....	36	Rückgabewerte .....	36
Internet Explorer .....	186	Objekte .....	85, 182	Schleife abbrechen.....	30
Interval .....	37, 182	<b>onreadystatechange</b> .....	82	Schleifen.....	29
<b>Javascript-Links</b> .....	8	Optimierung .....	170	Scope .....	21
JS-Framework .....	158	Option.....	25	Senden .....	82
JSON.....	85	Option entfernen .....	26	Server .....	136
Key-Value-Paare .....	73	optional .....	27	Session-Cookies.....	71
Klassen.....	51	Paradigma .....	85, 183	<b>sessionStorage</b> .....	73
Klassen vererben .....	51	Parameter.....	34, 81, 183	SET .....	52
Kommentare.....	6, 155	parse.....	85	Setter .....	53
Komponenten.....	150	Pattern.....	106	Settern.....	52
Konstruktoren .....	50, 182	Patterns .....	178	Setters .....	54
<i>Konstruktors</i> .....	50	<b>PDF.JS</b> .....	170	Sichtbarkeit .....	35
Kontrollstruktur .....	29	<b>PHP</b> .....	136	<i>Simple Object Access</i>	
<b>Konva</b> .....	170	Platzhalterbilder .....	178	<i>Protocol</i> .....	86
Kreis.....	109	<b>Plotly</b> .....	170	Single-Page-Anwendungen	
Kreisbogen.....	109	<b>POST</b> .....	81	.....	170
Kurven .....	108	Preload .....	80	SOAP.....	86
<b>Kute</b> .....	170	<i>Promise</i> .....	75	SocketIO .....	91
Ladevorgang .....	82	Promises .....	75	Spritemap.....	179
Leeren.....	111	Prompt.....	23	Startpunkt .....	108
'Leinwand' .....	104	Prototyp .....	52	Startwinkel .....	109
<b>Less</b> .....	170	<b>Prototype</b> .....	170, 183	state .....	151
let.....	13	<b>Prozedur</b> .....	183	Strict .....	9
<b>Linearer Verlauf</b> .....	105	Pseudo-3D .....	171	<b>String</b> .....	12, 183
Linien .....	108	<b>Push.JS</b> .....	170	stringify .....	85
<b>localStorage</b> .....	73	Push-Notifications .....	170	Stroke .....	107
logische Bedingung.....	27	Quellcode .....	23	strokeText .....	109
<i>lokalen Scope</i> .....	13	<b>Radialer Verlauf</b> .....	106	Style-Attribut.....	63
Lokales Speichern .....	71	Radians .....	109	<b>Stylesheet</b> .....	183
Maschine-zu-Maschine-		Radiobuttons .....	24	<i>super</i> .....	52
Interaktionen.....	86	Radius .....	109	<b>Svelte</b> .....	170
<b>Masonry</b> .....	170	React.....	149, 170	<b>Sweetalert</b> .....	171

Switch .....	27	Typecasting.....	19, 183	<b>Webfonts</b> .....	110, 179
synchron .....	81	undefiniert.....	21	Webserver .....	80
<b>Syntax</b> .....	183	Unterprogramm .....	33	Webserver erstellen.....	140
Tagnamen .....	96	<b>UTF-8 ohne BOM</b> .....	7	Webservice.....	86
Takt.....	37	V8 .....	136	Websockets .....	91, 136
Template-Literal .....	16	Validator .....	179	Webstorage.....	73
<b>Template-Strings</b> .....	16	<b>Variable</b> .....	183	Webworker .....	75
Text.....	109	Variablen .....	12	Wertvorgabe .....	34
Textareas .....	23	Vergangenheit.....	72	While .....	29
Textfelder .....	23	Verlauf .....	105	Whitespaces.....	97
then .....	76	Verschachtelung.....	151	XML .....	86, 96, 183
this .....	67	Verzweigung.....	27	<b>XMLHttpRequest</b> .....	83
<b>Three</b> .....	170	Verzweigungen.....	27	XML-Objekt .....	82
Timeout .....	36	Virtuelles DOM .....	149	Zahlen.....	12
Transformation.....	112	Vollbild .....	171	Zählschleife .....	29
Transparenz.....	111	Vue .....	158	<b>Zeichenkette</b> .....	183
trim .....	97	<b>Vue.js</b> .....	170	Zeichenketten .....	12
<b>Turbo</b> .....	170	Wartezeit.....	37	Zeitsteuerung .....	36
<b>Turntable</b> .....	171	WebApp.....	170	Zusammenfassen .....	33