Einführung in JavaScript von Petra Treubel

Inhalt

1. Die Programmiersprache JavaScript ... 1
   1.1 Einführung ... 1
   1.2 Das Einbinden von JavaScript ... 1
   1.2 Ein kurzes Beispiel ... 3
2. Sprachumfang ... 4
   2.1 Variablen ... 4
   2.2 Datentypen ... 5
   2.3 Anweisungen und Operatoren ... 7
   2.4 Kontrollstrukturen ... 9
3. Funktionen ... 11
4. DOM ... 12
   4.1 Einführung in das DOM ... 12
   4.3 DOM-Manipulation ... 14
5. Ereignisse ... 20
   5.1 HTML-Event-Handler ... 20
   5.2 DOM-Event-Handler ... 21
   5.3 DOM-Event-Listener ... 23
6. Objekte in JavaScript ... 26
   6.1 Einführung ... 26
   6.2 Objektorientierung mit Prototypen ... 26
   6.3 Objektorientierung mit Pseudoklassen ... 28
   6.4 Objektorientierung mit Klassensyntax ... 29
7. Daten über XML und JSON ... 31
   7.1 Einführung ... 31
   7.2 XML ... 31
   7.3 JSON ... 32
8. Was ist AJAX? ... 33
   8.1 Einführung ... 33
   8.2 Einblick in die Arbeitsweise von http ... 34
   8.3 Erste Schritte ... 36
   8.4 XMLhttpRequest ... 36
   8.5 Unterschied zwischen get und post ... 40
   8.6 XmlHttpRequest und PHP ... 41
   8.7 Ajax mit JQuery ... 42

1. Die Programmiersprache JavaScript

1.1 Einführung

JavaScript ist eine an Java angelehnte, plattformunabhängige Scriptsprache, die 1995 von Netscape in Kooperation mit Sun Microsystem entwickelt worden ist. Diese Sprache hieß zunächst Mocha, dann LifeScript und später JavaScript. Microsoft integriert 1996 zum ersten Mal im Internet Explorer 3 einen JScript-Interpreter. 1997 veröffentlicht die European Computer Manufacturers Association (ECMA) ECMAScript als standardisierte Version von JavaScript und reicht den Standard bei der ISO ein. Im gleichen Jahr verknüpft Microsoft HTML, Scripting und Style Sheets zu DHTML.

Der derzeitige Standard ist ECMAScript 2017 vom Juni 2017 (ECMA-262). Dabei gibt es von Browser-Hersteller zu Hersteller verschiedene Implementationen. ECMAScript wird im allgemeinen Sprachgebrauch JavaScript genannt.

JavaScript-Code wird direkt (oder indirekt) in eine HTML-Seite eingefügt und vom in den Web-Browser integrierten Interpreter ausgeführt, wenn er an den entsprechenden Code kommt. JavaScript ist also im Gegensatz zu PHP eine clientseitige Sprache. Innerhalb des Browser wird JavaScript wie in einem "Sicherheitskäfig" ausgeführt. So kann ein Script im Allgemeinen weder auf das Dateisystem des Rechners, noch auf andere geöffneten Webseiten zugreifen. Allerdings gibt es in diesem Zusammenhang Ausnahmen (Windows Scripting Host, ActiveX etc.).

1.2 Das Einbinden von JavaScript

JavaScript kann direkt in HTML-Seiten eingebunden werden. Dafür wird der Script-Tag benutzt, in dem JavaScript-Code geschrieben wird.

```
<!DOCTYPE html>
<html lang="de">
<head>
  <meta charset="utf-8">
</head>
<body>

  <script>
    document.getElementById('demo').innerHTML = "JavaScript!";
  </script>

</body>
</html>
```

Wichtig: Am Sinnvollesten ist es jedoch den Java-Code in Extra-Dateien mit der Endung `.js` zu schreiben und diese einfach nur einzubinden.

```
<script src="./js/extern.js"></script>
```

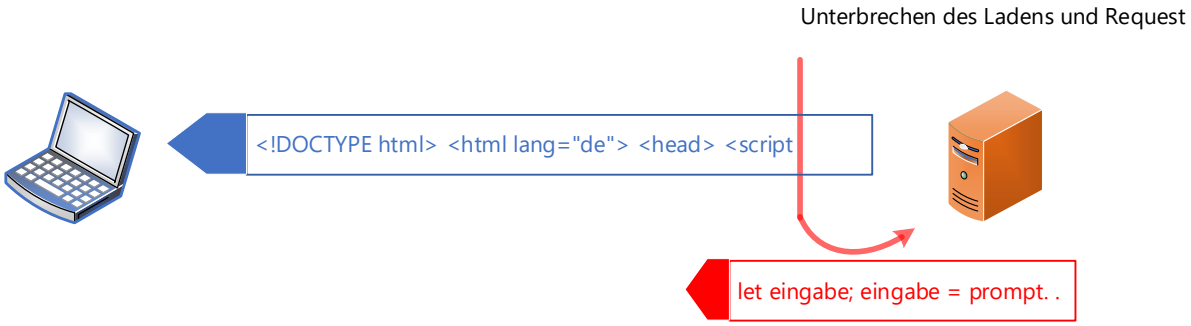
Manchmal findet man noch ältere Versionen der Script-Tags, die aber in HTML 5.x nicht mehr verwendet sollten.

```
<script type="text/javascript">
```
Version für HTML 4.x und XHTML 1.x

```
<script language="JavaScript">
```
Version für HTML 3.2 (veraltet)

Eine weitere Überlegung ist, wo da der Script-Tag platziert werden soll. Es gibt hier verschiedene Meinungen. Wichtig ist das Laden der HTML- und JavaScript- Dokumente zu verstehen.

Der Browser beginnt mit dem Laden eines HTML-Dokuments und parst die HTML-Tags (auch HTML-Markups genannt). Sobald der Browser auf einen Script-Tag stößt, wird entweder der Code ausgeführt oder der Browser beginnt mit dem Laden der externen JavaScript-Datei. Wenn der Browser ein Request zum Laden der externen js-Datei sendet, wird das Parsen des HTML-Codes eingestellt, bis der Browser die js-Datei heruntergeladen hat.



Erst danach setzt der Browser das Parsen des restlichen HTML fort.

Aus diesem Grund wird häufig empfohlen die JavaScript-Datei kurz vor dem schließenden Body-Tag einzubinden. Der Vorteil ist, dass alle HTML-Elemente fertig geladen wurden, bevor ein JavaScript-Code darauf zugreifen möchte. Nachteilig ist dies bei größeren Seiten, da der Benutzer unter Umständen zu lange warten muss.

Wenn JavaScript-Dateien innerhalb des head-Tags eingebunden werden und mit asynch angegeben wird, dass die Daten asynchron geladen werden, stoppt der Browser das Parsen nicht. Die Scriptdateien werden dann parallel heruntergeladen.

```
<script src="script1.js" async></script>
```

Bei zusätzlicher Angabe von **`defer`** wird der JavaScript-Code erst ausgeführt, wenn die HTML-Seite vollständig geladen und geparst ist.

Ein Entwickler kann nie sicher sein, ob JavaScript so wie erwartet vom Browser ausgeführt wird oder ob die Benutzung im Browser deaktiviert wurde. Mit Hilfe des noscript-Tags kann man überprüfen, ob das der Fall ist.

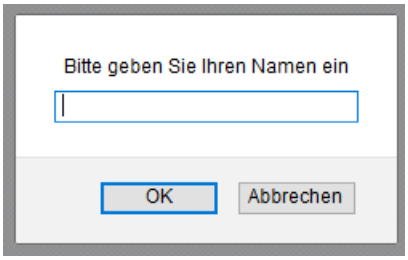
```
<noscript>
  JavaScript ist nicht verfügbar oder ist deaktiviert. <br />
  Bitte verwenden Sie einen Browser, der JavaScript unterstützt
  oder aktivieren Sie JavaScript.
</noscript>
```

1.2 Ein kurzes Beispiel

Die Anweisung

```
eingabe = prompt("Bitte geben Sie Ihren Namen ein", "");
```

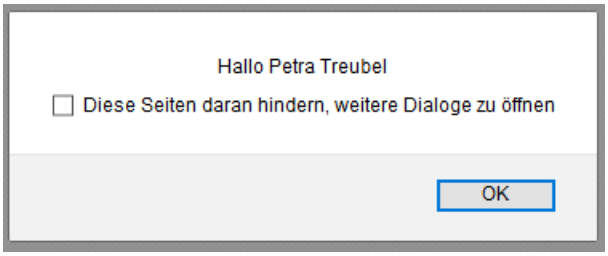
liefert das Dialogfeld



Nach Eingabe eines Namens wird der Text der Variablen `eingabe` zugewiesen. Dieser Text kann dann mit zusätzlichem Text '`Hallo `' einer Variablen `ausgabe` zugewiesen werden und ausgegeben werden. Wichtig ist das Leerzeichen rechts von `Hallo`, da sonst der eingegebene Name direkt an `Hallo` klebt!

```
alert(ausgabe)
```

liefert folgendes Dialogfeld.



Siehe im folgenden Beispiel:

```
<script>
   'use strict';
   let eingabe;
   let ausgabe;

   eingabe = prompt("Bitte geben Sie Ihren Namen ein", "");
   ausgabe = 'Hallo ' + eingabe;
   alert(ausgabe);
</script>
```

Der JavaScript-Code wird zur Laufzeit vom Interpreter des Webbrowsers interpretiert. Dabei muss beachtet werden, dass die JavaScript-Interpreter von Webbrowser zu Webbrowser sehr unterschiedlich agieren. Es gibt zwar einen Sprachkern, aber man sollte trotzdem das Programm in verschiedenen Webbrowser-Versionen testen.

2. Sprachumfang

2.1 Variablen

Variablen sind dafür da, um Werte zu speichern. Die Variable und der Wert der Variablen sind maximal so lange gültig, wie das Skript ausgeführt wird. Eine Variable kann immer nur einen Wert haben. Wird ihr ein neuer Wert zugewiesen, ist der alte Wert verloren.

Variablen werden mit dem reservierten Wort `let` oder `var` und einem folgenden Namen deklariert. Eine Variable muss einen eindeutigen Namen besitzen. Das erste Zeichen des Namens muss ein Buchstabe, ein Unterstrich oder ein Dollarzeichen sein. Folgende Zeichen können Buchstaben, Ziffern, Unterstriche oder Dollarzeichen sein. JavaScript unterscheidet zwischen Groß- und Kleinschreibung.

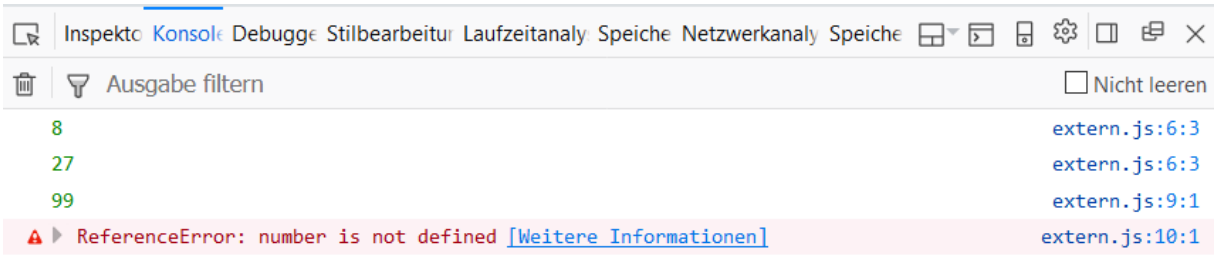
Beispiel:

```
let Name
let name;
var nettoPreis;
```

Dabei gibt es einen Unterschied zwischen `let` und `var`. Mit let deklarierte Variablen sind innerhalb der Blockklammern { } gültig und außerhalb nicht mehr sichtbar.

```
let globalNumber  = 99;

for (let index = 2; index <= 3; index ++) {
  let number = Math.pow(index, 3);
  console.log(number);                    // 8, 27
}

console.log(globalNumber );
console.log(number);
```

Ergebnis in der Web-Konsole:



Die Variable `globalNumber` ist überall sichtbar. Die Variable `number` ist nur innerhalb der Blockklammern sichtbar und außerhalb dieses Blocks gibt es einen `ReferenceError`. Genauso ist die Variable `index` nur innerhalb des Schleifenblocks sichtbar.

Würde alle Variablen mit `var` deklariert werden, gäbe es keinen Syntax-Fehler. Der Vorteil von `let` ist jedoch, dass durch die strengere Gültigkeit keine Seiteneffekte durch eine nicht erkannte Gültigkeit auftauchen können.

2.2 Datentypen

Es gibt in JavaScript keine explizite Typenfestlegung. Es wird zwar zwischen den primitiven Datentypen Zahlen, Text und boolesche Wahrheitswerte unterschieden, aber erst durch die Verwendung einer Variablen, wird ihr Datentyp festgelegt.

Beispiel:

```
let x, y, z;
x = 9.9;                  // eine Gleitkommazahl
y = 'Hallo Welt';         // eine Zeichenkette, bzw. String
z = true;          // eine boolesche Variable
```

JavaScript speichert **Gleitkommazahlen** immer als 64bit-Gleitkommazahl (Wertebereich +- 1.7976931349623157 * 10^{308} +-5 * 10^{-324}). Allerdings werden bestimmte Integer-Operationen auf 32-Bit-Integerwerten durchgeführt. Die größte darstellbare Zahl ist ± 2^{53} – 1. Diese Angaben können in unterschiedlichen Laufzeitumgebungen differieren.

Zeichenketten (Strings) werden als Unicode-Zeichensatz in einfachen oder doppelten Anführungszeichen angegeben. Um den Unterschied zu HTML-Attributen deutlich zu machen, werden für Strings im Skript überwiegend die einfachen Anführungszeichen verwendet. Will man ein Anführungszeichen in einer Zeichenkette unterbringen, muss dieses Zeichen durch einen Backslash außer Kraft (escaped) werden.

Beispiel:

```
buch = "\"Per Anhalter durch die Galaxis\" von D. Adams";
```

Mit dem Backslash könne aber auch bestimmte Escape-Sequenzen eingegeben werden:

\t	Tabulatorzeichen
\n	Zeilenumbruch
\xHH	Das Latin-1-Zeichen, welches durch den Hexadezimalcode HH angegeben wird
\uHHHH	Das Unicode-Zeichen, welches durch die vier Hexadezimalcodeziffern HHHH angegeben wird

Dabei ist zu beachten, dass Ausgaben mit der Funktion `alert("text")` oder `confirm("text")` zu unerwünschten Ergebnissen führen können. Das €- Zeichen muss über `unescape("%u20AC")` ersetzt werden. Die Funktion `unescape(text)` wandelt einen Wert %HH, wobei HH für einen hexadezimalen Wert steht, in das entsprechende ASCII-Zeichen um. Werte, die größer als 255 sein müssten, werden mit %uHHHH kodiert.

```
let text = "%50%65%74%72%61";
alert( unescape( text ));
```

JavaScript wandelt Zeichenketten automatisch in Zahlen um, wenn damit gerechnet wird:

```
let x = '80' ;
let y = '10' ;
let z = x / y ;       // Rechnung: 80 / 10 = 8
```

Allerdings ergibt

```
let z = x + y ;       // Rechnung: "80" + "10" = "8010"
```

(vergleiche Anweisungen und arithmetische Operatoren)

Neben den primitiven Datentypen unterstützt JavaScript noch Arrays und Objekte, wobei sich beide nur durch unterschiedliche Verhaltensweisen und Methoden unterscheiden.

Ein Objekt enthält ein oder mehrere Variablen, die meist als Eigenschaften bezeichnet werden. Außerdem gibt es Funktionen (Methoden), die im Zusammenhang mit dem Objekt aufgerufen werden können. Dabei wird das Objekt und die Eigenschaft, bzw. die Methode mit Punkt an den Namen des Objekts gehängt.

Ein Array enthält mehrere Variablen vom gleichen Typ und erinnert an eine Excel-Tabelle.

```
let arbeitsstundenProWoche = [34, 41, 25, 39];
```

Jeder einzelne Wert kann dann mit `arbeitsstundenProWoche[zahl]` angesprochen werden. Dabei wird der 1. Eintrag mit der Zahl 0 adressiert, der 2. mit 1 und der dritte mit 2 usw.:

```
arbeitsstundenProWoche[1]  // ergibt 41
```

Objekte sind zusammengesetzte Datentypen, die verschiedenartige Variable enthalten können.

```
let article = {
                name: 'MAKITA Schlagbohrmaschine HP2071';
                price: 279.99
                number: 661632
                }
```

Es gibt auch eine Reihe von Objekten, die zum Sprachumfang von JavaScript gehören. So findet sich im Objekt `Math` eine Anzahl von mathematischen Operationen oder Werte und das Objekt `Date` liefert Datums- und Zeitangaben.

```
//erzeugt ein Datumsobjekt mit heutigem Datum und Uhrzeit
let heute = new Date();
//Methode, die den Tag zurück gibt
heute.getDate();
```

Verwirrend dabei ist, dass auch die sogenannten primitiven Datentypen wie ein Objekt benutzt werden können.

Beispiel:

```
let zahl = 99;
```

Zahlen können in Zeichenketten umgewandelt werden:

```
let text = zahl.toString();              // enthält danach "99"
```

oder durch Angabe des Zahlencodes:

```
let binaerText = zahl.toString(2);       // enthält danach "1100011"
```

Auch Arrays können sich wie Objekte verhalten.

Beispiel:

```
let zahlenliste = new Array(2, 4, 6);
// Eigenschaft, die die Anzahl der Array-Element hier: 3 enthält
zahlenliste.length
```

Seit ECMA2015 gibt es auch den Ausdruck `const`, der für als eine Art Konstante fungiert. So erhält eine Variable einmalig einen Wert, der nicht verändert werden kann.

```
const PI = 3.14;
PI = PI * 2      // FEHLER!!!
```

Wichtig:

Für Objekte verhält sich `const` etwas anders, weil die Eigenschaften sich sehr wohl verändern dürfen, aber kein **neues Objekt** zugewiesen werden darf!

Natürlich gibt es auch Funktionen, die aus einem String eine Zahl erzeugen:
Mit `parseInt('99')` wird eine Zeichenkette bis zum ersten Auftreten eines nichtnumerischen Wertes in eine ganze Zahl umgewandelt. Mit `parseFloat('3.14')` in eine Gleitkommazahl.

Beispiel:
```
let zahl = parseInt('11');                // zahl enthält 11
let zahl2 = parseInt('ff', 16);           // zahl2 enthält 255,
                                          zur Basis 16
let zahl3 = parseInt('2 Chinesen')        // zahl3 enthält 2

let text = '';
isNaN(text)                          // true, wenn text keine Zahl ist
```

2.3 Anweisungen und Operatoren

Eine Anweisung ist eine Befehlszeile, die vom Interpreter ausgeführt wird. Anweisungen werden in JavaScript mit Semikolon getrennt. Eine einfache Anweisung ist die Zuweisung. Dabei wird der rechte Teil vom Gleichheitszeichen als Wert der Variablen links vom Gleichheitszeichen zugewiesen.

Beispiel:
```
let zahl = 3;
zahl = 13 + 55.99;
let ergebnis = zahl * 7 + ( 9 % zahl );
```

Eine Anweisung kann habe auch aus dem Aufruf einer Funktion bestehen:
```
window.close();
alert("Hallo Welt");
```

Falsch:
```
datum + 99 = 7;
```

Wenn innerhalb einer Anweisung gerechnet werden soll, können verschiedene Variablen und Operatoren verknüpft werden.

```
nettopreis = 120.95;
mwst = 19,
mwstEuro = nettopreis * mwst / 100 + nettopreis;
```

Arithmetische Operatoren (in der Reihenfolge der Bindung):

Negatives Vorzeichen	-	
Inkrement	++	zahl++ oder ++zahl
Dekrement	--	zahl-- oder --zahl
Multiplikation	*	
Division	/	
Modulo	%	
Addition	+	
Subtraktion	-	
mit der Zuweisung kombiniert		*=, /=, %=, +=, -=

Falsch:
```
let summe;
summe += 3;           // Achtung: FEHLER
```

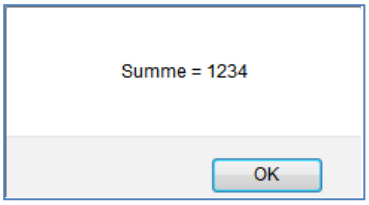
Wichtig: Beachten Sie, dass eine Variable `undefined` als Wert enthält, bevor ihr ausdrücklich ein Wert zugewiesen wurde. Dies kann immer dann zu einem unerwünschten Ergebnis führen, wenn mit dem Wert einer Variable gerechnet werden soll oder dieser ausgeben werden soll.

Das Pluszeichen hat noch die Fähigkeit Zeichenketten aneinander zu hängen. Dies sollte man beachten, wenn eine Variable nur Ziffern enthält und man vielleicht damit rechnen möchte.

```
let zahl1 = '123';
```

```
let zahl2 = '4';

alert('Summe = ' + zahl1 + zahl2);
```

Ergebnis:



Vergleichsoperatoren

Gleichheit	==
Identität	=== // Wert und Datentyp gleich
Kleiner als	<, bzw. kleiner gleich <=
Größer als	>, bzw. größer gleich >=
Ungleichheit	!=
Nichtidentität	!==

Logische Operatoren

Logisches Und	&&
Logisches Oder	||
Logisches Nicht	!

2.4 Kontrollstrukturen

Anweisung

Anweisungen werden in JavaScript mit Semikolon getrennt. Eine einfache Anweisung ist die Zuweisung. Dabei wird der rechte Teil vom Gleichheitszeichen als Wert der Variablen links vom Gleichheitszeichen zugewiesen.

Beispiel:

```
zahl = 13 + 55.99;
multi = zahl * 12;
```

Wichtig: Bei Berechnungen mit den arithmetischen Operatoren +, -, * und / gilt die Punkt-vor-Strich-Regel.

Eine Anweisung kann habe auch aus dem Aufruf einer Funktion bestehen:

```
window.close();
alert('Hallo Welt');
```

Verzweigung

Eine Verzweigung wertet die Bedingung aus und wenn die *bedingung* wahr ist, wird *anweisung1* ausgeführt. Ist die Bedingung falsch, wird mit der *anweisung2* im else-Block die Verarbeitung fortgesetzt.

```
if( bedingung ) {
   anweisung1 ;
}
else {
   anweisung2 ;
}
```

Ist die Variable divisor nicht 0 wird zahl durch divisor geteilt. Ist divisor gleich 0 wird eine Fehlermeldung ausgegeben:

```
if(divisor != 0) {
   quotient = zahl / divisor;
}
else {
   alert('nicht durch 0 teilbar!');
}
```

Fallunterscheidung

Wenn ein Ausdruck überprüft werden soll, ob er bestimmt Werte enthält, ist eine Fallunterscheidung effizienter als eine verschachtelte Verzweiung.

```
switch(ziffer){
   case 1:
   // führe hier Code aus, wenn ziffer 1 ist
   break;

   case 9
   // führe hier Code aus, wenn ziffer 9 ist
   break;
}
```

Schleifen

Soll eine Anweisung mehrfach hintereinander ausgeführt werden, bietet sich Schleifen an. Die Anweisungen im Block werden solange ausgeführt, wie die Bedingung im Kopf der Schleife wahr ist.

```
while( bedingung ) {
   anweisung1 ;
   anweisung2 ;
}
```

```
let zahl = 0;
while(zahl < 10){
   console.log(zahl);
   zahl = zahl + 1;
}
```

Es ist auch möglich die Bedingung am Ende der Schleife auszuwerten.

```
let zahl = 0;
do{
   console.log(zahl);
   zahl = zahl + 1;
}while(zahl < 10);
```

Eine Schleife, die automatisch eine Zählvariable benutzt, wir als for-Schleife bezeichnet.

```
for( initialisierung ; bedingung ; aktualisierung ) {
   anweisung1 ;
   anweisung2 ;
}
```

```
let zahl;
for(zahl = 0; zahl < 10; zahl++){
   console.log(zahl);
}
```

3. Funktionen

Es gibt Funktionen wie z. B. `parseInt(text)` oder `isNaN(text)`, die schon integriert sind. Man hat aber auch die Möglichkeit eigene Funktionen zu schreiben. Eigene Funktionen werden benutzt, um den Quellcode besser zu gliedern, in dem mehrere Anweisungen in einer Funktion zusammengefasst werden. Gleichzeitig entsteht damit die Möglichkeit, diese Anweisungen mehrfach von unterschiedlichen Stellen aus zu verwenden.

Funktionen müssen geladen werden, bevor sie aufgerufen werden können. Es ist deshalb gut, sie im `head`-Teil zu schreiben oder mit Hilfe einer externen Datei im `head`-Teil einzubinden.

Funktionen beginnen immer mit dem Schlüsselwort `function` einem gewählten Namen und den runden Klammern:

```
function tuWas(){
   . . .
}
```

Innerhalb der runden Klammern kann eine beliebige Anzahl von Parametern angegeben werden. Diese müssen auch beim Aufruf als Argumente zwischen die runden Klammern eingetragen werden.

Funktiondefinition:

```
function tuWas (zahl1, zahl2){

   return (zahl1 + zahl2) ;


}
```

Aufruf der Funktion:

```
let irgendeineZahl = 15;
alert ( 'Summe ' + tuWas (2, irgendeineZahl) );
```

Ausgabe des Programms:

Wichtig: Beim Aufruf der Funktion `tuWas` müssen genaue zwei Attribute übergeben werden. Diese Attributwerte werden den lokalen Variablen `zahl1` und `zahl2` übergeben. Die Konstante `2` wird an die lokale Variable `zahl1` übergeben und der Wert von `irgendeineZahl` wird an die Variable `zahl2` übergeben.

Durch die Anweisung `return` wird das Ergebnis der Berechnung `zahl1 + zahl2` an das aufrufende Skript zurückgeliefert und kann mittels `alert` ausgegeben werden.

4. DOM

4.1 Einführung in das DOM

Das DOM ist eine API (**A**pplication **P**rogramming **I**nterface Programmierschnittstelle), die definiert, wie auf Objekte des Dokumentes zugegriffen werden kann. Vom World Wide Web Consortium wurde ein Standard-DOM definiert, das von den meisten Webbrowser unterstützt wird.

Mittels DOM kann

- durch die Struktur und den Inhalt eines Dokuments navigiert werden,
- die Struktur durch Einfügen und Löschen von Elementen verändert werden,
- die Eigenschaften von Strukturelementen angepasst werden,
- Inhalte verändert werden und
- komplette Dokumente erstellt werden.

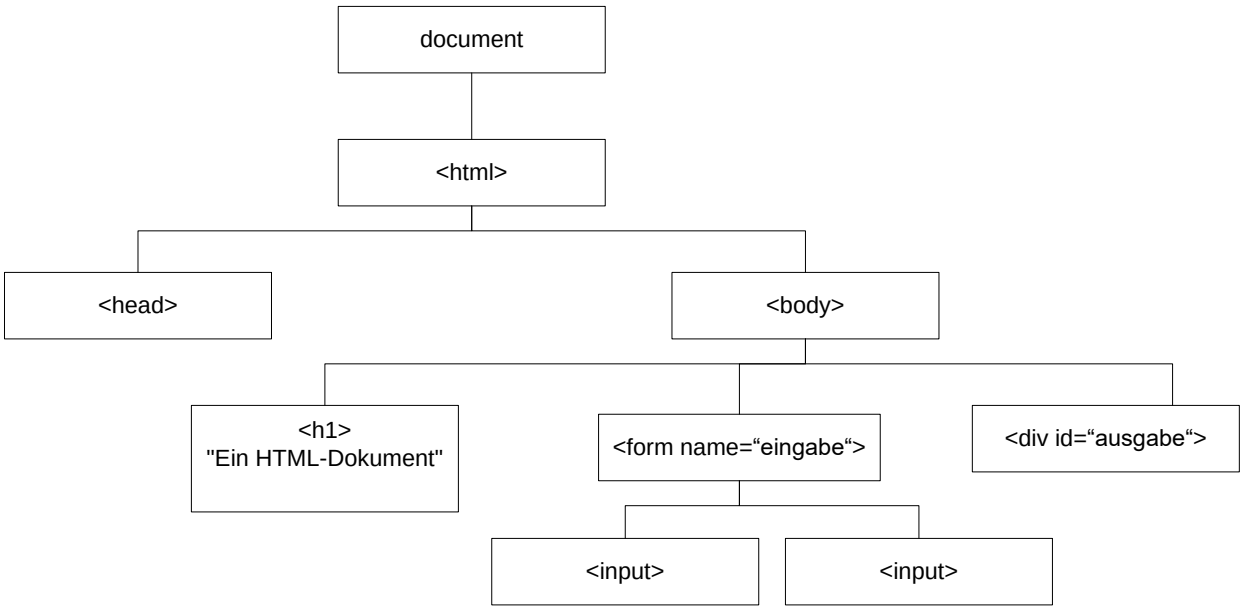


Abb.: Baumartige Struktur eines HTML-Dokuments

```
<!DOCTYPE html>
<html>
<head>
   <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
<body>

   <h1 id="ueberschrift"> Ein HTML-Dokument </h1>

   <form name="eingabe">
      Nettopreis:
      <input type="text" name="nettoName" id="nettoId">
      <input type="button" name="rechneButton" value="Berechnen">
   </form>

<div id="ausgabe">  </div>

</body>
</html>
```

Grundsätzlich kann man vier verschiedene Arten von Knotentypen in der DOM-Struktur identifizieren:

- der Dokumentenknoten, der als Wurzel am Anfang steht und in der Abbildung auf der vorherigen Seite ganz oben steht. Dieser Knoten repräsentiert das ganze Dokument und wird über das Objekt `document` angesprochen.
- die Elementknoten, die für die verschiedenen HTML-Elemente (z. B. head, h1, form) stehen
- die Attributknoten, welche den HTML-Elementen zugeordnet sind (z. B. id, name)
- die Textknoten, durch welchen die HTML-Elemente bestimmte Inhalte erhalten (z. B. "Ein HTML-Dokument" in <h1>)

Das `document`-Objekt hat verschiedene Eigenschaften und Methoden. So erhält man über die Eigenschaften `document.URI` die URL und über `document.images` ein Array mit allen images-Tags des aktuellen Dokuments. Über die Methode document.getElementById('ausgabe') erhält der Programmierer einen Zugriff auf das div-Tag mit der Id "ausgabe".

Es gibt folgende HTML-Dokument-Elemente, die Unterobjekte von `document` sind:

- document.anchors – Liste aller Verweisanker im Dokument
  `<a name="top">Anfang der Seite</a>`
- document.forms – Liste aller Formulare im Dokument
  `<form name="Testform" action="test.html"`
- document.images - Liste aller Bilder im Dokument
  `<img name="bild" src="./img/test.gif">`
- document.links – Liste aller Links im Dokument
  `<a href="http://www.google.de/">Google</a>`

Eine Übersicht der Eigenschaften und Methoden vom `document` findet sich z. B. auf https://wiki.selfhtml.org/wiki/JavaScript/DOM/Document .

Auch die Elementknoten haben verschiedene Eigenschaften und Methoden. Dabei muss man beachten, dass nicht alle Elemente die gleichen Eigenschaften und Methoden haben.

Mit der Methode `window.open()` wird ein neues Dokument geöffnet. Mit den Methoden von document `write()` und `writeln()` werden dem Dokument Inhalte hinzugefügt und mit `close()` sollte das Dokument wieder geschlossen werden, das heißt es ist fertig.

```
function myPopup(text){
     var fenster = window.open("", "", "width=400, height=300");
     var docImFenster = fenster.document;
     docImFenster.open();
     docImFenster.write ("<!DOCTYPE HTML>\n");
     docImFenster.write ("<html> \n <head> \n </head> ");
     docImFenster.write ("<body> \n");
     docImFenster.write ("<p>" + text + "</p>");
     docImFenster.write ("\n </body>");
     docImFenster.close();
  }
```

Wichtig: Über die verschiedenen Eigenschaften und Methoden kann die Struktur eines HTML-Dokuments ausgelesen und verändert (manipuliert) werden. Diese Veränderungen erfordern kein erneutes Laden des bereits geladenen HTML-Dokuments.

4.3 DOM-Manipulation

4.3.1 Elemente auswählen

Um im Aussehen des geladenen HTML-Dokuments Änderungen über JavaScript zu bewirken, muss das JavaScript-Programm erst Zugriff auf Elemente erhalten. Es gibt verschiedene Methoden, um HTML-Elementknoten auszuwählen.

Beispiel:
```
<input type="text" class="eingabe" name="nettoName"
                          id="nettoId">
      <p class="fettdruck" Hallo Welt </p>
```

Über das id-Attribut	`getElementById('nettoId')`	ein Element
Über den CSS-Selektor	`querySelector('input[type=text]')`	ein Element
Über den Namen	`getElementsByName('nettoName')`	Array von Elementen
Über den Tag-Namen	`getElementsByTagName('input')`	Array von Elementen
Über das `class`-Attribut	`document.getElementsByClassName("eingabe")`	Array von Elementen

Mit den Methoden `querySelector()` und `querySelectorAll()` kann man auf die CSS-Selektoren zugreifen und hat so vielfältige Möglichkeiten, um HTML-Elemente auszuwählen. Dabei wählt `querySelector()` das erste Element, welches auf den CSS-Selektor zutrifft, aus und liefert dies als Rückgabewert. Während `querySelectorAll()` alle Elemente auswählt und ein Array zurückgibt.

Beispiele:
```
// Auswählen des Elements mit id="nettoId"
let pElement = document.querySelector('#nettoId');

// Auswählen des 1. Elements mit class="nettoId"
let pElement = document.querySelector('.eingabe');

// Auswählen aller Element mit class="fettdruck"
let pElement = document.querySelectorAll('.eingabe');


// Auswählen des ersten Elements mit <p>-Tag
let pElement = document.querySelector('p');

//Auswählen des ersten Elements mit <p>-Tag und der class-Angabe
'fettdruck'
let pElement = document.querySelector('p.fettdruck');
```

4.3.2 Formulare in HTML

Formulare in HTML werden grundsätzlich zwischen dem <form> ... </form>-Tag platziert. Innerhalb des einleitenden form-Tags können noch zusätzliche Angaben gemacht werden.

```
<form name = "irgendeinName">
     <!-- hier folgen die Formularelemente -->
</form>
```

Innerhalb der form-Tags werden die Formularelemente angegeben. Formularelemente können:

- Eingabefelder (<input type="text" **...** >),
- Eingabebereiche (<input type="textarea" **...** >),
- Auswahllisten (<select **...** > <option> **...** </option> </select>),
- Radiobuttons (<input type="radio" **...** >),
- Checkboxen (<input type="checkbox" **...** >),
- Dateiuploadfelder (<input type="file" **...** >).
- und versteckte Element (<input type="hidden" **...** >)

Beispiel:

```
<form name="eingabe">
  Nettopreis:
  <input type="text" name="nettoName" id="nettoId">

  <select name="umsatzsteuer" size="5">
    <option value="7"> 7% </option>
    <option value="19"> 19% </option>
    <option value="0"> keine Umsatzsteuer </option>
  </select>

  <input type="radio" name="zahlmethode" value="MC"> Mastercard<br>
  <input type="radio" name="zahlmethode" value="Visa"> Visa<br>

  <input type="button" name="rechneButton" value="Berechnen">
</form>
```

Wie unter Punkt 4.3.1 schon beschrieben, gibt es verschiedene Methoden HTML-Elemente zu selektieren. Auch Formularelement können auf unterschiedliche Arten ausgelesen werden. Aus dem oben angegebenen Formular kann der Wert

des Textfeldes

1. über die DOM-Hierarchie ausgelesen werden

```
nettobetrag = document.forms[0].elements[0].value;
```

2. über die DOM-Hierarchie mittels der Namen ausgelesen werden

```
nettobetrag = document.eingabe.nettoName.value;
```

3. über das Attribut name ausgelesen werden

```
nettobetrag = document.eingabe.getElementByNames[0].value;
```

4. über das Attribut id ausgelesen werden

```
nettobetrag = document.getElementById('nettoId').value;
```

Für die DOM-Hierarchie gilt das 1. Formular wird über `forms[0]` angesprochen, das zweite über `forms[1]` usw. Die gilt genauso für andere DOM-Elemente wie z. B. die Input-Felder: `elements[0]`, `elements[1]`, ... oder Bilder `images[0]`, `images[1]`, ...

Beispiel für das Lesen einer Eigenschaft:

```
// Text der Überschrift wird ausgelesen und der Variablen text zugewiesen
   let text = document.getElementById('ueberschrift').innerHTML;

// Inhalt des Eingabefeldes wird ausgelesen:
   let inhalt = document.getElementById('nettoId').value;
```

Beispiel für das Schreiben einer Eigenschaft:

```
// Der Inhalt der Variablen text wird dem Bereich div zugewiesen
   document.getElementById('ausgabe').innerHTML = text;

// Inhalt des Eingabefeldes wird gesetzt:
   document.getElementById('nettoId').value = 100;
```

Hierbei ist zu beachten, dass nicht alle Eigenschaft neu gesetzt werden dürfen.

Außerdem gibt es noch für die anderen Elemente eines Formulars Besonderheiten:

Beispiel für das Auslesen der Auswahlliste

```
   umsatzsteuer = document.eingabe.umsatzsteuer.
   options[document.eingabe.umsatzsteuer.selectedIndex].value;
```

Beispiel für das Auslesen der Radiobuttons

```
   if(document.eingabe.zahlmethode[0].checked){
      zahlmethode = 'MC';
   }
   else{
      zahlmethode = 'Visa';
   }
```

Wie eben beschrieben kann über die Objektart und die Position auf die einzelnen Elemente zugegriffen werden. Dies hat den Vorteil, dass man keine Namen verwenden muss, bzw. diese verändert werden können ohne den JavaScript-Code anpassen zu müssen.

```
   // Inhalt aus dem Textfeld wird der Variablen zugewiesen
   let inhalt = document.forms[0].elements[0].value;
```

Oder man verwendet die Namen oder ID's. Dies hat den Vorteil, dass auch nach Veränderung in der Reihenfolge korrekt auf das Textfeld zugegriffen werden kann.

```
   // Inhalt aus dem Textfeld wird der Variablen zugewiesen
   let inhalt = document.eingabe.nettoName.value;

   let inhalt = document.getElementById('nettoId').value;
   let inhalt = document.getElementsByName('nettoName')[0].value;
```

Es muss beachtet werden, dass die Methode getElementById() den Zugriff auf genau ein Element erhält, weil ID's auf einer HTML-Seite einmalig sind. Die Methode getElementsByName() liefert ein Array zurück, welches theoretisch alle Elemente mit dem Namen 'bestellwert' beinhaltet.

4.3.3 Elemente verändern

HTML-Elemente können verändert werden, nachdem sie geladen wurden. So kann, wie oben beschrieben wurde, häufig über die Eigenschaft `innerHTML` ein Inhalt verändert oder neu gesetzt werden.

Beispiel:

```
<div id="ausgabe">  </div>

ausgabe = document.getElementById('ausgabe')
ausgabe.innerHTML = 'Neuer Text…'
```

Dabei ist zu beachten, dass Elemente unterschiedliche Eigenschaften haben können. Ein Button erhält über die Eigenschaft `value` eine neue Beschriftung, der `title`-Tag der Website wird direkt gesetzt und ein Bild `image` bekommt über die Eigenschaft `src` eine neue Grafikdatei.

Beispiel:

```
test = document.querySelector('input[type=button]')
test.value = 'Hallo';

document.title = 'Umsatzrechner';

document.images[0].src = './img/chrysanthemum_thumb.jpg'
```

Wichtig: Ein einzelnes Element setzt ein vollkommen anderes Vorgehen voraus als eine Liste (HTMLCollection) von Elementen.

Auf eine Liste von Elementen kann man über einen Index zugreifen, der bei 0 beginnt und bei der Länge – 1 endet. Hat ein Dokument drei Bilder ist `document.images[0]` das erste Bild und `document.images[1]` das zweite Bild und `document.images[document.images.length-1]` das letzte Bild auf der Seite.

Anzahl der Bilder innerhalb des Dokuments:

```
document.images.length
```

Auslesen aller Bilddateien in einem Dokument:

```
let text = ''

for(let i=0; i < document.images.length; i++) {
   text += document.images[i].src + '\n'
}
alert(text)
```

Über *`element`*`.style.`*`eigenschaft`* `= 'wert'` kann Elementen ein anderes Aussehen gegeben werden.

```
<h1 id="headline"> Eine Überschrift </h1>
```

In JavaScript:

```
let ueberschrift = document.getElementById('headline')
ueberschrift.style.color = '#2B5184'
ueberschrift.style.border = '5px solid #F77B17'
```

Das Interface `Node` bietet noch viel weitreichende Möglichkeiten.

`Node` ist im DOM ein einzelner Knoten, der sich im `document`-Baum befindet und weitere Kind-Knoten haben kann oder einfach nur Text enthält.

Jeder Knoten hat folgende Eigenschaften:

Datentyp	Bezeichnung	Beschreibung
`DOMString`	`nodeName`	Name des Knotens
`DOMString`	`nodeValue`	Wert des Knotens
`unsigned short`	`nodeType`	Knotentyp (1 Element-, 2 Attribut-, 3 Text-Knoten)
`Node`	`parentNode`	Elternknoten
`NodeList`	`childNodes`	Liste aus Knoten
`Node`	`firstChild`	Erster Kind-Knoten
`Node`	`lastChild`	Letzter Kind-Knoten
`Node`	`previousSibling`	Vorheriger Geschwister-Knoten
`Node`	`nextSibling`	Nächster Geschwister-Knoten
`NamedNodeMap`	`attributes`	Liste der Attribute
`Document`	`ownerDocument`	Dokument, indem sich der Knoten befindet

`DOMString` ist eine UTF-16 kodierte Zeichenkette

Es ist jetzt möglich Knoten dynamisch zu erzeugen, zu verändern oder zu löschen. Dabei muss unterschieden werden, ob es sich um einen Knoten (`node`), ein Element (`child`), einen Text oder ein Attribut handelt.

Beispiel:

```
<div id="texte">
  <p>Text 11</p>
  <p>Text 12</p>
</div>
```

In JavaScript:

```
let textBereich =  document.getElementById('texte');

let neuesPElement = document.createElement("p");
let textKnoten = document.createTextNode("neuer Text");
neuesPElement.appendChild(textKnoten);
textBereich.appendChild(neuesPElement);
```

Danach:

```
<div id="texte">
  <p>Text 11</p>
  <p>Text 12</p>
  <p>neuer Text</p>
</div>
```

Elemente können auch entfernt werden. Dafür gibt es die Methode *`element`*`.removeChild(`*`element`*`)`. Allerdings sollte man sicher sein, dass das Element auch wirklich vorhanden ist, bevor versucht wird es zu entfernen.

Auch Attribute könnten ausgelesen oder verändert werden. Um alle Attribute auszulesen kann die Eigenschaft `attributes` verwendet werden.

```
let textBereich =  document.getElementById('texte');

if(textBereich.hasAttributes){
   for(let i = 0; i < textBereich.attributes.length; i++){
      alert( textBereich.attributes[i].name + '=' +
                textBereich.attributes[i].value );
   }
}
```

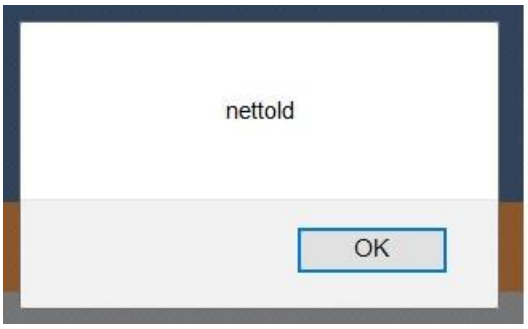
Über die Methoden *element*`.getAttribute(`*attributName*`)` kann der Wert eines Attributs ausgelesen werden.

```
Nettopreis: <input type="text" name="nettoName" id="nettoId" >
Bruttopreis: <input type="text" name="bruttoName" id="bruttoId" >
```

In JavaScript:

```
let eingabeElemente = document.getElementsByTagName('input');
alert(eingabeElemente[0].getAttribute('id'));
```

Ausgabe:



Mittels der Methode *element*`.setAttribut(`*attributName*`, `*wert*`)` kann ein neues Attribut für ein Element gesetzt werden oder ein bestehendes Attribut kann einen neuen Wert erhalten.

Allerdings können Attribute auch über die Objekteigenschaften gesetzt oder ausgelesen werden.

```
let eingabeElemente = document.getElementsByTagName('input');
alert(eingabeElemente[0].id));
```

Wichtig: Hierbei ist zu beachten, dass nur Standardattribute über die Objekteigenschaft auslesbar ist.

Beispiel:

```
<input type="text" name="bruttoName" id="bruttoId" xyz="nix" >

alert(eingabeElemente[1].getAttribute('xyz'));     //Ausgabe: nix
alert(eingabeElemente[1].xyz);                     //Ausgabe: undefined
```

5. Ereignisse

5.1 HTML-Event-Handler

Mit JavaScript ist es möglich, dass Skripte in Interaktion mit dem Benutzer treten. Das bedeutet, dass ein Skript nicht von Anfang bis Ende abgearbeitet wird, sondern dass in Abhängigkeit der Benutzereingaben Anweisungen ausgeführt werden. Das Skript reagiert auf bestimmte Ereignisse (Events).

Der Webbrowser erzeugt dabei Events, wenn der Benutzer zum Beispiel eine Taste drückt, und benachrichtigt das Programm. Man unterscheidet dabei Tastatur- oder Mausereignisse, aber auch Ereignisse, die durch das Laden des Dokuments oder eine Serverantwort hervorgerufen werden. Wenn ein Ereignis auftritt, versucht der Webbrowser eine entsprechende Funktion, einen sogenannten Event-Handler aufzurufen, um das Ereignis zu verarbeiten. Diese Event-Handler werden an entsprechenden Stellen im HTML-Teil positioniert.

Man unterscheidet dabei wiederum die verschiedenen Event-Modelle. Es gibt ein ursprüngliches Model, welches von allen JavaScript-fähigen Browsern unterstützt wird, das Standard-Event-Modell, auch HTML-Event-Handler genannt.

Bei dem ursprünglichen Event-Modell werden die Anweisungen zum Event-Handling mit Hilfe der Attribute der HTML-Elemente angegeben.

```
<body onload="alert('Hallo Welt!');">
<body onload="document.forms[0].elements[0].focus();mache();">
```

Nachdem das Dokument mit allen Bildern und JavaScript-Code vollständig geladen worden ist, erzeugt der Browser den onload-Event und führt den JavaScript-Code aus, der hinter dem entsprechenden Event-Handler-Attribut angegeben ist. Hier könnten eine einzelne oder mehrere Anweisungen stehen oder eine Funktion aufgerufen werden, die sich in einer externen Datei befindet. Wichtig ist, dass nach dem onload-Event keine Veränderungen des HTML-Dokuments mittels document.write() mehr gemacht werden dürfen.

Allerdings sollte man beachten, dass nicht jedes Event-Handler-Attribut bei jedem HTML-Element angewendet werden darf. So gibt es bei einer Schaltfläche (Button) kein onchange-Event, weil die Schaltfläche nicht vom Benutzer verändert werden kann.

```
<form name="form_1">
  <input name="cancel" type="reset" value="Abbrechen"
    onclick="bestaetige();">
```

In JavaScript:

```
function bestaetige(){
  return confirm('Wollen Sie das Formular wirklich zurücksetzen?');
}
```

Außerdem sollte man beim Abfangen von Ereignissen vorsichtig sein. Wenn falsche Eingaben eines Benutzers in ein Eingabefeld verhindert werden sollen, ist es besser, den Event-Handler `onchange` zu verwenden und nicht bei jedem Tastendruck (`onkeypress`) eine Überprüfung zu starten.

Event-Handler-Attribut	Ereignis	Unterstützt
onabort	Laden des Bildes wird abgebrochen	<img>
onblur	Feld verliert den Fokus	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Inhalt des Feldes wurde geändert, nachdem das Feld den Fokus erhalten hat und beim Verlassen des Feldes	<input>, <select>, <textarea>
onclick	beim Anklicken	Von den meisten Elemente
ondblclick	bei doppeltem Anklicken	Von den meisten Elementen
onerror	Wenn beim Laden des Bildes ein Fehler aufgetreten ist.	<img>
onfocus	beim Aktivieren des Elements	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	bei gedrückter Taste	Formularelementen und <body>
onkeypress	Nachdem eine Taste heruntergedrückt wurde, folgt keydown	Formularelementen und <body>
onkeyup	bei losgelassener Taste, folgt keypress	Formularelementen und <body>
onload	Das Dokument vollständig geladen wurde	<body>, <frameset>, <img>
onmousedown	bei gedrückter Maustaste	Von den meisten Elementen
onmousemove	bei weiterbewegter Maus	Von den meisten Elementen
onmouseout	beim Verlassen des Elements mit der Maus	Von den meisten Elementen
onmouseover	beim Überfahren des Elements mit der Maus	Von den meisten Elementen
onmouseup	bei losgelassener Maustaste	Von den meisten Elementen
onreset	Wenn das Formular zurückgesetzt werden soll. Wird false zurückgegeben, wird das Formular nicht zurückgesetzt.	<form>
onsubmit	Wenn das Formular abgeschickt werden soll. Wird false zurückgegeben, wird das Formular nicht abgeschickt.	<form>
onunload	Wenn das Dokument oder das Frameset aus dem Browser entfernt wird.	<body>, <frameset>

Innerhalb eines Formulars, welches durch Betätigen einen Buttons einen Request auslösen soll, kann beim Absenden das Ereignis `onsubmit` ausgelöst. So könnte man das Formular überprüfen und gegebenenfalls das Absenden verhindern.

```
<form action="folge.html" onsubmit="return pruefe(this)">
   . . .
   <input type="submit" name="rechneButton" value="Berechnen">
</form>
```

In JavaScript:

```
function pruefe(myForm){
   let eingabe = myForm.elements[0].value;
   let ok = false;

   if(eingabe !== '' && ! isNaN(eingabe)){
      ok = true;
   }
   return ok;
}
```

Wichtig: Das ursprüngliche, bzw. HTML-Event-Handler hat den Nachteil, dass HTML-Code und JavaScript vermischt werden. Dies sollte nach Möglichkeit vermieden werden. Deshalb sollte man eher das DOM-Event-Handling oder die DOM-Event-Listener verwenden.

5.2 DOM-Event-Handler

Hier werden die Event-Handler in JavaScript angegeben. Ein Event-Handler wird dem Event zugeordnet werden, indem der Funktionsname **OHNE KLAMMERN** dem Event-Handler zugewiesen wird.

```
<form name="form_1">
  <input name="cancel" type="reset" value="Abbrechen>
```

In JavaScript:

```
document.form_1.cancel.onclick=bestaetige;

function bestaetige(){
  return confirm('Wollen Sie das Formular wirklich zurücksetzen?');
}
```

Innerhalb der ausgelagerten JavaScript-Datei müssen dabei die Event-Handler-Registrierungen möglichst event-unabhängig realisiert warden oder an das Ereignis onload gebunden warden.

```
<form name="eingabe" id="myForm" action="folge.html">
  Nettopreis: <input type="text"> <br>
. . .
  <input type="submit" name="rechneButton" value="Berechnen">
</form>
```

In JavaScript

```
window.onload = init;

function init(){
  document.getElementById('myForm').onsubmit =
pruefe;
}

function pruefe(){
  let eingabe = document.getElementById('myForm').elements[0].value;
  let ok = false;

  if(eingabe !== '' && ! isNaN(eingabe)){
    ok = true;
  }
  return ok;
}
```

5.3 DOM-Event-Listener

5.3.1 Einführung in den Event-Listener

Mit den beiden vorangehenden Event-Handlern kann eine Funktion mit einem Ereignis verknüpft werden. Will man allerdings mehr als eine Funktion mit einem Ereignis verknüpfen, bieten sich die Event-Listener (oder auch Event-Handler) an. Hier ist der Vorteil, dass eine beliebige Anzahl mit dem Ereignis verknüpft werden kann und auch wieder "entbindet".

Funktion mit einem Ereignis verknüpfen:

```
element.addEventListener('click', funktion);
```

Bindung einer Funktion an ein Ereignis lösen:

```
element.removeEventListener('click', funktion);
```

Bei Versionen des Internet Explorers, die gleich oder älter als IE8 sind, funktionieren diese Funktionen nicht. Dann sollte man auf `attachEvent()` und `detachEvent()` zurückgreifen oder JQuery einsetzen.

Wichtig: Das Ereignis wird ohne Angabe von `on` angegeben. Also nicht `onclick`, sondern nur `click`!

Beispiel:

```
window.onload = init;

function init(){
   let formular = document.eingabe;

   formular.addEventListener('submit', pruefe);
}

function pruefe(){
   let eingabe = forms[0].elements[0].value;
   let ok = false;

   if(eingabe != '' && !isNaN(eingabe)){
      alert(forms[0].elements[0].value);
      ok = true;
   }
   return ok;
}
```

Hier gilt es aber zu beachten, dass bei oben verwendeter Methode, anders als bei den vorher gezeigten Beispielen, das **Absenden des Formulars nicht** verhindert wird.

5.3.2 Der Ereignisfluss

Wenn ein Ereignis ausgelöst wird, durchläuft das Ereignis verschiedene Phasen. Bei älteren Browsern unterscheiden sich diese Phase, sind aber mittlerweile vom W3C standardisiert.

Grundlegend verlaufen die Phasen so, dass zuerst in der Capturing-Phase ein Event vom obersten Knoten (dem Dokumentenknoten) zu dem Element, für welches das Ereignis ausgelöst wurde, "hinabsteigt". Beim Ziel (`target`) angekommen, für den dieses Ereignis ausgelöst wurde, beginnt die Target-Phase. Hier wird das Ereignis am Zielelement ausgelöst. Danach steigt das Ereignis in der Bubbling-Phase wieder auf zum Dokumenten-Knoten. Standardmäßig wird ein Event-Handler immer in der Bubbling-Phase registriert. Ist ein Event-Handler jedoch mit der Methode `element.addEventlistener( ereignis, funktion, true)` registriert, wird die Funktion in der Capturing-Phase aufgerufen. Dies ist aber nicht das Standardverhalten.

5.3.3 Die Eigenschaften und Methoden vom Event-Objekt

Das Event-Objekt verfügt über verschiedene Eigenschaften, die abrufbar sind. So kann über die Eigenschaft `type` der Name des Events ermittelt werden und über `target` erhält man das Zielelement, welches das Ereignis ausgelöst hat,

In Fällen, in denen mehrere gleichartige Eingabeelemente mit der gleichen Funktion mit einem Ereignis verknüpft werden sollen, bietet sich der Zugriff auf das Event-Objekt selber an.

Beispiel:

```
<form name="bestellform">
  <div class="form-group">
    <input type="radio" id="R1" name="bestellung" value="R1">
    <label for="R1">Pizza Margherita </label>
  </div>
  <div class="form-group">
    <input type="radio" id="R2" name="bestellung" value="R2">
    <label for="R2">Pizza Tonno </label>
  </div>
  <div class="form-group">
    <input type="radio" id="R3" name="bestellung" value="R3">
    <label for="R3">Pizza Spinaci </label>
  </div>
</form>

<div id="ausgabefeld"> </div>
```

In JavaScript:

```
let radioButtons = document.bestellform.bestellung;
for(let i = 0; i < radioButtons.length; i++){
  radioButtons[i].addEventListener('change', liesAus);
}

function liesAus(event){
  let ausgabe = document.getElementById('ausgabefeld');
  ausgabe.innerHTML = event.target.value;
}
```

Innerhalb es Event-Listeners kann auch auf mit dem Schlüsselwort `this` auf das jeweilige Element zugegriffen werden, an welchem der Event-Listener ausgelöst worden ist.

```
let inputFeld = document.getElementById("nachname")
inputFeld.addEventListener('change', function(event)
            {
              alert(this.value);
            });
```

Das Event-Objekt hat auch drei Methoden. So kann zum Beispiel das Abschicken eines Formulars man mit der Methode `event.preventDefault()` verhindert werden. Diese Methode bewirkt, dass die Standardaktion, die mit einem Element verbunden ist, nicht ausgeführt wird.

```
<form name="bestellform" action="bestellungsAnnahme.php">
  . . .
  <input type="text" name="anzahl">
  . . .
  <input type="submit" id="los" value="los">
</form>
```

In JavaScript:

```
document.bestellform.addEventListener('submit', pruefe);

function pruefe(event){
  let inhalt = bestellform.anzahl.value;
  if(inhalt == ''){
     event.preventDefault();
     console.log('abschicken verhindert');
  }
  else{
    console.log('abgeschickt');
}
```

Mit der Methode `event.stopImmediatePropagation()` wird das Abarbeiten weiterer Funktionen, die auch mit dem gleichen Ereignis verknüpft sind, gestoppt.

6. Objekte in JavaScript

6.1 Einführung

JavaScript ist keine objektorientierte Programmiersprache wie Java oder C++, sondern man spricht hier von einer objektbasierten oder prototypenbasierten Programmiersprache.

Mit dem Standard ECMAScript 6 (kurz ES6/ES2015), der 2015 verabschiedet wurde, wurde zwar eine Klassensyntax eingeführt, man kann aber nicht von echten Klassen reden.

In JavaScript gibt es drei verschiedene Arten der "objektorientierten Programmierung":

- Prototypische Objektorientierung
- Pseudoklassische Objektorientierung
- Objektorientierung mit Klassensyntax

6.2 Objektorientierung mit Prototypen

Bei dieser Art werden Objekte auf Basis von anderen Objekten gebildet. Das "Mutter"-Objekt ist dabei wie in Java das Objekt `Object`. Dafür wird mit dem Schlüsselwort `let` ein Objekt erzeugt, dessen Eigenschaften mit Werten vorbelegt werden. Methoden, die zu diesem Objekt gehören, können ebenfalls sofort oder später hinzugefügt werden.

Beispiel-Code 1:

```
let auto = {
   name: 'Golf',
   ps: 110,
   price: 35000.90,
   km: 0,
   drive: function(km){
      this.km += km;
      console.log("Ist " + km + " km gefahren");
   }
}

auto.drive(15);
console.log(auto.name);
console.log(auto.ps);
console.log(auto.price);

auto.km = 30;
console.log(auto.km);
```

Hier können jederzeit weitere Eigenschaften oder Methoden hinzugefügt werden. Dabei wird innerhalb von Methoden weiterhin das Schlüsselwort `this` verwendet.

Beispiel-Code 2:

```
auto.pimpMyCar = function(ps){
   this.ps += ps;
}

auto.pimpMyCar(30);
```

Weitere Objekte werden mit der Object.create()-Methode gebildet. Diese werden dann vom ursprünglichen Objekt abgeleitet.

Beispiel-Code 3:

```
let vw = Object.create(auto);
vw.name = "Golf";

let skoda = Object.create(auto);
skoda.name = "Oktavia";

console.log("VW " + vw.name);
console.log("Skoda " + skoda.name);
```

Für diese abgeleiteten Objekte können jetzt ebenfalls neue Eigenschaften oder Methoden gebildet werden.

Dabei hat jedes Objekt eine Prototyping-Eigenschaft, die auf das vorherige Objekt verweist. Mittels der Methode `Object.getPrototypeOf` kann auf das vorige Objekt (das Basis-Objekt) zugegriffen werden.

Beispiel-Code 4:

```
let proto1 = Object.getPrototypeOf(skoda); //Prototyp auto
console.log(proto1);

let proto2 = Object.getPrototypeOf(proto1);  //Prototyp Object
console.log(proto2);
//
let proto3 = Object.getPrototypeOf(proto2);  Prototyp null
console.log(proto3);
```

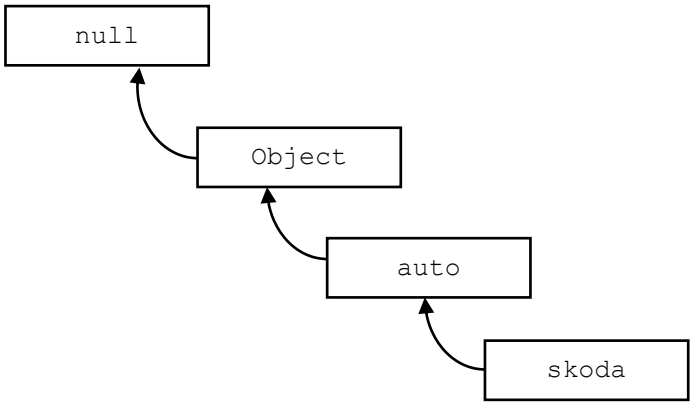


Abb.: *JavaScript prototype chain*

6.3 Objektorientierung mit Pseudoklassen

Anders als bei der literalen Schreibweise basieren hier die Objekte auf dem Einsatz von Konstruktorfunktionen. Konstruktorfunktionen sehen wie normale Funktionen in JavaScript aus, allerdings beginnen die Namen per Konvention mit einem Großbuchstaben und zu Pseudoklassen werden sie erst, in dem Objekt mit dem `new`-Operator erzeugt werden.

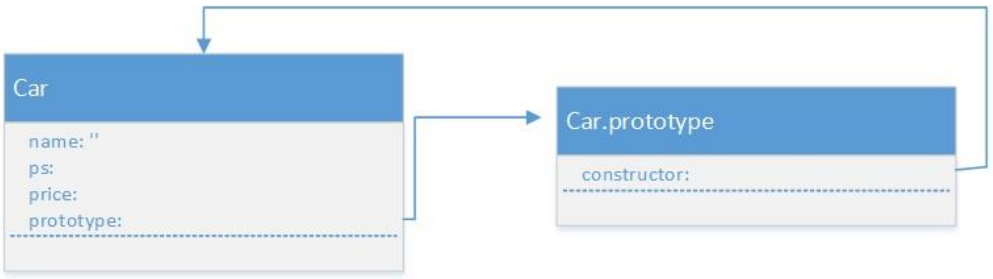
Beispiel-Code 5:

```
function Car(name, ps, price){
      this.name = name;
      this.ps = ps;
      this.price = price;
      this.km = 0;

      this.drive = function(km){
         this.km += km;
         console.log("Ist " + km + " km gefahren");
      }
   }

let auto = new Car("Audi A3", 110, 45000.00);

auto.drive(25);
```

Auch hier kann von der Basis-Pseudoklasse Klassen abgeleitet werden, um eine Vererbung zu implementieren. Allerdings ist dies etwas aufwendig.

Beispiel-Code 6:

```
/*
* Definition der Konstruktorfunktion Car
* Erstellung eines Objekts hinterlegt in der prototype-Eigenschaft
* über die Construktor-Eigenschaft von prototype wird auf Car verlinkt
*/
function Car(name, ps, price){
. . .
}
```




```
//Erstellen einer Unter-Klasse, die den Konstruktor von Car aufruft

function VW(name, ps, price, exhaust){
      Car.call(this, name, ps, price);
      this.exhaust = exhaust;

      this.changeExhaust = function(){
            --this.exhaust;
      }
}
```

Um den Konstruktor einer Basisklasse aufzurufen, wird die Methode `call()`, der das aufrufende Objekt (hier `VW`) und weitere Parameter übergegen wird, angegeben.

Eine Verbindung zwischen VW- und Car-Klasse wird über eine Verbindung der prototype-Eigenschaften realisiert.

```
//Setzen des VW.prototype
VW.prototype = new Car();

//Setzen des VW-Construktors, ume eine Verbindung zu Car zu bekommen
VW.prototype.constructor = VW;
```

6.4 Objektorientierung mit Klassensyntax

Wie schon oben beschrieben, gibt es seit ES6/ES2015 auch einen Klassensyntax, die stark an die Klassendefinitionen höherer Programmiersprachen angelehnt ist. Um eine Klasse zu definieren, wird das Schlüsselwort `class` verwendet. Hier gibt es auch einen Konstruktor `constructor`, der automatisch beim Erzeugen eines neuen Objekts aufgerufen wird. Der Name der Eigenschaft und der Name der Getter- oder Setter-Methode müssen sich unterscheiden

```
class Car{
   constructor(name, ps, price){
      this._name = name;
      this._ps = ps;
      this._price = price;
      this._km = 0;
   }

   drive(km){
      this._km += km;
      console.log('Ist ' + km + ' km gefahren');
   }

   get name(){
      return this._name;
   }

   set name(name){
      this._name = name;
   }

   get ps(){
      return this._ps;
   }

   set ps(ps){
      this._ps = ps;
   }

   get price(){
      return this._price;
   }

   set price(price){
      this._price = price;
   }
}
```

Die obenstehende Klasse kann benutzt werden, nachdem ein Objekt erstellt worden ist:

```
let auto = new Car('BMW', 150, 55000.75);
```

Über `new Car` wird der Konstruktor der Klasse `Car`, nämlich `constructor(name, ps, price)` indirekt aufgerufen und die Werte 'BMW' an `name`, 150 an `ps` und 55000.75 an `price` übergeben.

Mit dem Objekt `auto` kann dann auf die Eigenschaften der Klasse zugegriffen werden, die einen Getter, bzw. einen Setter haben.

Benutzung des Getters: `alert( auto.name )`
Benutzung des Setters: `auto.ps = 130`

Auf Methoden der Klasse kann normal zugegriffen werden, in dem der Objektname mit Punktoperator verwendet wird: `auto.drive(35)`

Ohne Getter-Methode muss der Originalname der Eigenschaft genommen werden.

```
console.log(auto._km);
auto._km = 30;
console.log(auto._km);
```

Auf diese Art und Weise ist es möglich eine Kapselung der Eigenschaften zu erreichen. Mit Kapselung ist gemeint, dass der direkte Zugriff auf die Eigenschaften nicht möglich ist. So soll gewährleistet werden, dass nur gültige Werte in den Eigenschaften eines Objekts abgelegt werden.

7. Daten über XML und JSON

7.1 Einführung

Um Daten, wie zum Beispiel Bilder oder längere Texte dynamisch nachladen zu können, gibt es Formate, in denen die Daten angeliefert werden. Normalerweise könnte man alle Daten als Zeichenketten anliefern, aber dann gingen Informationen über die Struktur der Daten verloren. Außerdem müsste die Formatierung durch HTML/CSS schon in der Zeichenkette vorhanden sein, was das Ganze sehr starr macht.

7.2 XML

E**x**tensible **M**arkup **L**anguage (XML) ist Auszeichnungssprache. Daten werden in Textdateien nach einem speziellen hierarchischen System strukturiert und zwischen Computersysteme ausgetauscht. XML wird auch als Datenaustauschformat bezeichnet.

Unabhängig von dem verwendeten Datenbanksystem, Tabellenkalkulations- oder Textverarbeitungsprogramm können Daten strukturiert werden und auf der Gegenseite richtig angezeigt oder verarbeitet werden.

Beispiel

E**x**tensible **M**arkup **L**anguage (XML) ist Auszeichnungssprache. Daten werden in Textdateien nach einem speziellen hierarchischen System strukturiert und zwischen Computersysteme ausgetauscht. XML wird auch als Datenaustauschformat bezeichnet.

Unabhängig von dem verwendeten Datenbanksystem, Tabellenkalkulations- oder Textverarbeitungsprogramm können Daten strukturiert werden und auf der Gegenseite richtig angezeigt oder verarbeitet werden.

Beispiel:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response>
  <books>
    <book>
      <title>
        Entwickeln von Web-Applikationen mit Ajax
      </title>
      <isbn>
        1-904811-82-5
      </isbn>
    </book>
    <book>
      <title>
        Freude im Blumengarten
      </title>
      <isbn>
        1-234567-82-5
      </isbn>
    </book>

  </books>
</response>
```

Jede XML-Struktur beginnt mit einer Kopfzeile, die Angaben über die folgende Inhalte enthält. Diese Deklarationen bestehen im einfachsten Fall aus der Version `<?xml version="1.0"?>`. Dann können die Kodierung angegeben werden und ein Ja/Nein-Attribut, ob die zugehörige DTD in der aktuellen Datei steht oder in einer separaten Datei. Die Reihenfolge muss immer `version=` (Version), `encoding=` (Zeichensatz) und `standalone=` (DTD ja/nein) sein

7.3 JSON

JavaScript Object Notation ist ein etwas schlankere Form um Daten strukturiert von einem System auf ein anderes zu übertragen.

Anders als XML erfordert JSON keine hierarchische Baumstruktur, sondern kann aus mehreren Name-Werte-Paare, bzw. arrayartigen Strukturen zusammengesetzt sein.

Name-Werte-Paare sind immer mit einem beliebigen Namen z. B. `"title":` und einem Wert z. B. `"Entwickeln von Web-Applikationen mit Ajax"` definiert. Mehrere Name-Werte-Paare, die zusammengehören, werden mit Komma getrennt. Ein Array, bzw. eine Liste beginnt mit einem Namen (hier: `"books":`).

Beispiel:

```
{
   "books": [
      {
         "title": "Entwickeln von Web-Applikationen mit Ajax",
         "isbn": "1-904811-82-5"
      },
      {
         "title": "Freude im Blumengarten",
         "isbn": "1-234567-82-5"
      }
   ]
}
```

Die Notation von JSON:

- 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.
- Es dürfen keine Funktionen enthalten sein

JSON-Strings müssen über JSON.parse() geparst werden, damit sie bequem über JavaScript verarbeitet werden können.

Beispiel:

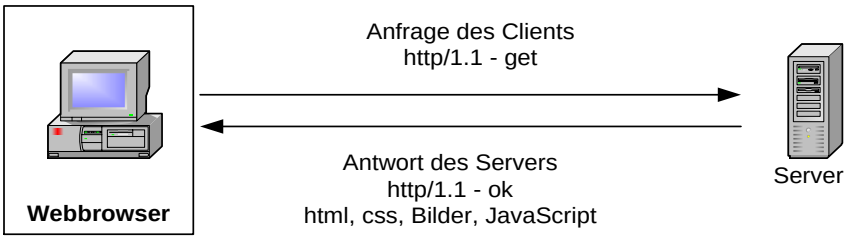
```
// Angabe des JSON-Objekts:
let jsonObjekt = '{"books": [ {"title": ". . .  }';

let jsonText = JSON.parse(jsonObjekt);
alert(jsonText.books[0].title);
```

8. Was ist AJAX?

8.1 Einführung

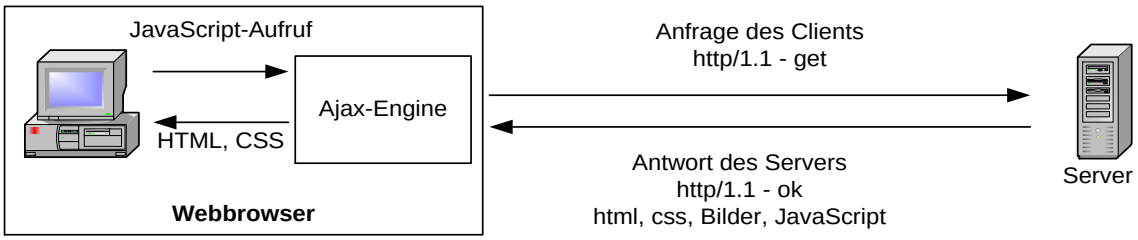
Das http-Protokoll dient in erster Linie dazu, um Webseiten im Webbrowser aus dem World Wide Web (WWW) anzeigen zu können. Diese Daten werden über das Netzwerk übertragen. Dafür müssen zuerst Daten von einem Server angefordert werden (request) und danach werden die Daten an den Client geliefert (response). Neben der reinen Datenübertragung gibt es noch diverse Kommunikationsmöglichkeiten zwischen dem Server und dem Client. So kann der Server den Client zum Beispiel darüber informieren, wenn das angeforderte Dokument nicht zur Verfügung steht.

Im traditionellen Webanwendungsmodell wird über den Webbrowser (Client) durch die Eingabe des Anwenders eine Anfrage an den Webserver (Server) gestellt.



Der Webserver antwortet, wenn die Anfrage erfolgreich war, mit einem ok und den angeforderten Daten.

Bei der Verwendung von Ajax wird durch eine meist durch eine Benutzeraktion, die eine JavaScript-Funktion aufruft, eine asynchrone Anfrage des Clients an den Server gesendet. Dabei wird eine Art Zwischenschicht, die Ajax-Engine zwischen Webbrowser und Webserver aktiv. Die Ajax-Engine ist im Grunde eine JavaScript-Funktion oder ein JavaScript-Objekt.



In traditionellen Webanwendungen arbeiten Webbrowser und Server synchron. Das heißt, der Webbrowser sendet eine Anfrage an den Server und wartet so lange, bis diese beantwortet wird. Asynchron bedeutet, dass die Ausführung nicht angehalten wird, um auf eine Antwort zu warten. So ist es möglich große Bilder oder lange Textabschnitte nachzuladen, ohne dass eine neue Webseite aufgerufen werden muss. Oder die Eingaben eines Benutzers können sofort bei der Eingabe kontrolliert werden.

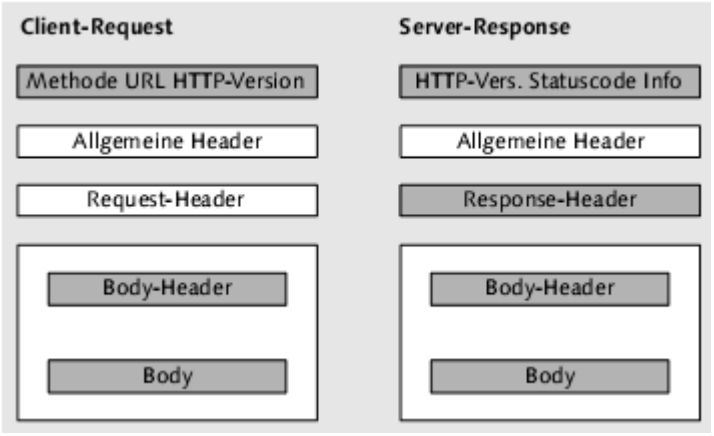
AJAX steht für **A**synchronous **J**avaScript **a**nd **X**ML.

Wichtig: Ab jetzt müssen Sie Ihre Programme über einen **Webserver** testen.

http://localhost/workspace/Ajax_Moodle/ajaxTest.html

8.2 Einblick in die Arbeitsweise von http

Unten ist der Aufbau eines HTTP-Datagramms aufgezeigt. Für den Benutzer wird im Browser nur der Body angezeigt. Dabei ist der Body im HTTP-Datagramm nicht mit dem HTML-Tag <body> zu verwechseln. Im HTTP-Body sind der <head>- und der <body>-Tag enthalten:



Gibt ein Benutzer in die Adresszeile seines Browsers zum Beispiel den Eintrag www.allianz.de/rechne.htm ein, beginnt der Client einen HTTP-Request zu senden.

Die erste Zeile des Headers enthält die Methode, die Zielressource und die verwendeten HTTP-Version. Die am meisten benutzten Methoden sind dabei GET und POST. Ein Schrägstrich (/) zeigt dabei an, dass die Anfrage für die Wurzel des Bereichs bestimmt ist.

```
GET   /rechne.htm    HTTP/1.1
```

In der zweiten Zeile ist der angesprochene Host enthalten.

```
Host: www.allianz.de
```

Darauf folgt der verwendete User-Agent. Diese Angabe ermöglicht es dem Server die benutzte Browserversion zu identifizieren.

```
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; de; rv:1.8.1.16)
Gecko/20080702 Firefox/2.0.0.16
```

Danach könnten Angaben über die Art des Cachings oder ob die TCP-Verbindung gehalten werden soll folgen. Außerdem könnte ein gespeichertes Cookie an den Server übertragen werden oder der Client könnte dem Server mitteilen, welche Sprache oder welche Codierungsverfahren er bevorzugt. Zuletzt muss eine Leerzeile das Ende des Headers anzeigen.

Im Header der HTTP-Antwort des Webservers sind die Antwort-Codes enthalten. Die relevanten Informationen im Antwort-Code sind die HTTP-Version des Servers gefolgt von dem Statuscode:

```
HTTP/1.1 200 OK
```

Diese Statuscodes lassen sich dabei in unterschiedliche Wertebereiche unterteilen (siehe Tabelle). Die Statuscodes 400 – 500 werden dem Benutzer angezeigt. Alle anderen sind für den Benutzer im Webbrowser unsichtbar.

Wertebereich	**Beschreibung**
100–199	Informationen während der Anfrage. Die Anfrage wird dabei vom Server noch bearbeitet.
200–299	Erfolgreiche Anfrage. Die eigentliche Aktion kann ausgeführt werden.
300–399	Umleitung der Anfrage. Eine weitere Bearbeitung der Anfrage wird notwendig.
400–499	Die Anfrage war unvollständig bzw. fehlerhaft und wurde daher abgebrochen. Es könnte aber auch die angefragte Datei nicht vorhanden sein oder die Zugriffsrechte fehlen.
500–599	Es ist ein Fehler auf dem Server aufgetreten.

Die bekanntesten Fehler sind der Statuscode 401 (nicht berechtigter Zugriff auf einen geschützten Bereich), Statuscode 403 (verbotener Bereich, bzw. unbekannter Benutzer/falsches Kennwort) oder der Statuscode 404 (nicht gefundenes Dokument).

Daneben überträgt der Server noch die von ihm verwendete Webserver-Version

```
Server: Apache/1.3.34 Ben-SSL/1.55
```

ob eventuell eine Authentifizierung erforderlich ist und zum Beispiel ein Cookie, welches beim Client gesetzt werden soll

```
Set-Cookie: PHPSESSID=ec6c378a2c69bfd8f1b39789fbb7e98b; path=/
```

Zu beachten ist, dass der Server keine Angaben darüber macht, mit welchem Typ der Client die Anfrage geschickt hat.

8.3 Erste Schritte

Es wäre möglich Ajax mit verborgenen Frames, bzw. iFrames zu realisieren. Allerdings hat Microsoft mit dem Internet Explorer 5.0 eine ActiveX-Komponente XMLHTTP zur Verfügung gestellt, die eine weit bessere Technik zur Verfügung stellt.

Mittlerweile unterstützen Safari (ab Vers. 1.2), Opera (ab Vers. 7.6), Firefox (Netscape 7/Mozilla 1.0) und IE die XMLHttp-Funktionalität. Jedoch unterscheiden sich die Funktionalitäten noch etwas. So unterstützen Safari und Opera nur GET und POST-Abfragen und der Internet Explorer benötigt je nach Version unterschiedliche Objekte.

Zuerst muss ein XMLHttpRequest-Objekt erzeugt werden. Dann muss das Objekt mittels der open-Methode initialisiert werden. Dabei gibt man die Methode (GET, POST, HEAD), die URL eventuell mit Name-Wert-Paaren und ein Flag für asynchrone oder synchrone Übertragung an.

Bei einer asynchronen Übertragung mit AJAX wird der aktuelle Status der Verbindung solange aktualisiert und als Information zurückgeliefert, bis die Anfrage vom Server vollständig ausgeführt wurde. Auf diese Weise kann der jeweilige Status einer Anfrage überwacht und gegebenenfalls bestimmte Aktionen durchgeführt werden.

Bei einer synchronen Anfrage mit AJAX erfolgt die Antwort erst nach dem vollständigen Übertragen der angeforderten Daten. Die Anfrage bleibt also solange aktiv, bis eine Antwort vom Server erzeugt wird. Das Ergebnis ist erst nach der Übertragung sofort und komplett verfügbar. Das größte Problem bei einer synchronen Datenübertragung liegt darin, dass der weitere Programmablauf solange blockiert bleibt, bis alle Daten vollständig übertragen wurden.

Die Schritte zum Ausführen einer AJAX-Anfrage:

1. Erzeugen eines AJAX-Objektes
2. Definition des passenden Event-Handlers (was passiert, wenn die Daten geladen wurden?)
3. Starten der Anfrage mit `open()`
4. (Eventuelles) Konfigurieren der Anfrage mit `setRequestHeader()`
5. Absenden der Abfrage mit `send()`

8.4 XMLhttpRequest

Die einzige Methode, um die richtige Version zu bestimmen, ist ein Objekt zu erzeugen.

Normalerweise ist eine try-and-error-Methodik in der Programmierung kein geeigneter Lösungsweg, aber hier geht es nicht anders. Am Besten ist die Erzeugung eines Objekts in einen try...catch-Block einzubinden. So führt die fehlerhafte Erstellung eines Objekts zwar zu einem Fehler, doch hat dieser keinen Stop des Programmablaufs zur Folge.

Für den Internet Explorer ab 7.0, Firefox, Opera und Safari würde es reichen, einfach ein neues Objekt zu erzeugen:

```
var xmlHttp = new XMLHttpRequest();
```

Bei allen anderen Versionen des Internet Explorers muss ein XMLHttpRequest-Objekt mittels eines ActiveX-Objekts erstellt werden. Hier gibt es folgende Möglichkeit:

```
var i;

var httpVersions = new Array("Microsoft.XMLHTTP",
          "MSXML2.XMLHTTP.6.0",
          "MSXML2.XMLHTTP.5.0",
          "MSXML2.XMLHTTP.4.0",
          "MSXML2.XMLHTTP.3.0",
          "MSXML2.XMLHTTP");


//Probier alle IE-Versionen durch
for(i = 0; i < httpVersions.length && !xmlHttp; i++)
{
   try
   {
      xmlHttp = new ActiveXObject(httpVersions[i]);
   }
   catch(e)
   {
      //nichts tun
   }
}
```

Alle Objekte unterstützen eine Reihe von Methoden und Eigenschaften. Die wichtigste Methoden und Eigenschaften, die von allen Objekten und Browsern unterstütz werden sollen, sind in den beiden folgenden Tabellen aufgeführt.

Methode	**Beschreibung**
`abort ()`	Mit dieser Methode können Sie den aktuell laufenden Request beenden.
`getAllResponseHeaders ()`	Gibt eine Liste aller vorhandener Header als key/value-Paare in einem String zurück.
`getResponseHeader (name)`	Gibt den Wert für den im Argument angegebenen Header zurück.
`open (method, url [,syncFlag, username, password])`	Hiermit wird der Request an die gewünschte Ziel adresse gestartet und die Art der Übertragung vereinbart.
`send (body|null)`	Sendet den Request an den Server mit einem optionalen Body.
`setRequestHeader (key, value)`	Mit dieser Methode können Sie einen optionalen Header für den Request als key/value-Paar setzen.

Eigenschaft	Beschreibung
`onabort`	Tritt auf, wenn die AJAX-Anfrage mit der Methode `abort()` abgebrochen wurde
`onerror`	Ereignis, welches im Fehlerfall auftritt, z. B. wenn die Netzwerkverbindung abbricht
`onload`	Event-Handler, der aufgerufen wird, wenn die AJAX-Anfrage erfolgreich ausgeführt wurde und eine Antwort enthält
`onreadystatechange`	Event-Handler, der bei jeder Änderung im Status des Requests aufgerufen wird. Wird in Verbindung mit der Eigenschaft readyState verwendet.
`readyState`	Diese Eigenschaft gibt den aktuellen Status des Requests zurück und ändert sich solange, bis das Ende der Anfrage erreicht ist.
`responseText`	Inhalt des Bodys als String aus der Antwort des Servers
`responseXML`	XML-Objekt für die Verarbeitung mit dem DOM, falls eine XML-Datei angefordert wurde
`status`	Numerischer Wert des Serverstatus am Ende der Übertragung, d.h., sobald die Eigenschaft readyState den Wert 4 hat
`statusText`	String mit einer Beschreibung des Serverstatus am Ende der Übertragung, d.h., sobald die Eigenschaft readyState den Wert 4 beinhaltet

Wie schon im vorgehenden Kapitel beschrieben, muss das XMLHttpRequest-Objekts initialisiert werden.

```
// synchrone Get-Anforderung
xmlHttp.open("GET", "test.txt", false);

// asynchrone Get-Anforderung
xmlHttp.open("GET", "beispiel.php?zahl1=33", true);
```

Dann muss ein Event-Handler registriert werden, der mit einer Funktion verknüpft ist, die aufgerufen wird, wenn die AJAX-Anfrage vollständig ausgeführt wurde.

```
xmlHttp.addEventListener('load', loadHandler)

function loadHandler(){
  alert('habe fertig')
}
```

oder

```
xmlHttp.onload = function (event) {
     alert('habe fertig')
  }
```

Wichtig:
Allerdings ist `onload` NUR für die korrekte Abarbeitung von AJAX-Engine verantwortlich und nicht für Fehler, die bei der Verarbeitung auf dem Server passiert. So würde eine nicht vorhandene URI trotzdem ein onload-Ereignis auslösen und den http-Status 404 zurückgeben.

Bei älteren Browsern gibt es das load-Ereignis nicht. Hier muss der Event-Handler mit dem Event-Handler verbunden werden, der ausgeführt wird, wenn eine Statusänderung im Request-Objekt erfolgt ist. Dieser kann auf verschiedene Arten registriert werden. Entweder als anonyme Funktion

```
xmlHttp.onreadystatechange = function()
{
  if(xmlHttp.readyState == 4){
    if(xmlHttp.status == 200){
      alert(xmlHttp.responseText);
    }
  }
}
```

oder als eigenständige, benannte Funktion

```
xmlHttp.onreadystatechange = readyStateHandler;

...

function readyStateHandler()
{
  if(xmlHttp.readyState == 4){
    if(xmlHttp.status == 200){
      alert(xmlHttp.responseText);
    }
  }
}
```

Danach wird der Request mit der Methode send abgeschickt. Bei der GET-Methode muss null als Argument übergeben werden. (Vergleiche Kap. 4.3 Unterschied zwischen get und post)

```
xmlHttp.send(null);
```

Der Request kann verschiedene Status durchlaufen, die mittels der readyState-Eigenschaft abgefragt werden können.

Wert	Bedeutung	Beschreibung
0	uninitialized	Der Request wurde noch nicht durch die Methode open() ausgelöst.
1	loading	Der Request wird gestartet, wurde bisher aber noch nicht abgeschickt.
2	loaded	Der Request wurde durch die Methode send() ausgeführt. Eine Antwort des Servers steht noch aus.
3	interactive	Die Übertragung der Antwort des Servers läuft. Teile davon sind bereits im Buffer und mittels der Eigenschaften responseText oder responseXML verfügbar.
4	complete	Der Request wurde vollständig ausgeführt und beendet, bzw. durch den Anwender abgebrochen.

Wenn der readyState den Wert 4 hat und das HTTP-Statuskennzeichen aus der Antwort 200 ist, können mittels der Eigenschaften responseText und responseXML auf eine eventuelle Antwort zugegriffen werden. Sollte das HTTP-Statuskennzeichen nicht 200 sein, könnte man über die Eigenschaft statusText den entsprechenden Fehler ausgeben lassen:

```
if(xmlHttp.status == 200){
  alert('Die zurückgegeben Daten sind: ' + xmlHttp.responseText);
}
else{
  alert('Es ist ein Fehler aufgetreten: ' + xmlHttp.statusText);
}
```

8.5 Unterschied zwischen get und post

Es gibt bei den Methoden get und post den Unterschied, dass die Parameter bei get an die URL angehängt werden und bei der Methode post im Body des HTTP-Datagramms. Dafür müssen die Name-Werte-Paare der Methode send als Argument übergeben werden.

```
var method = "GET";
var url = "rechner.php";
var params = "zahl1=99&zahl2=12";
var async = true;

// Methode GET
if(method == "GET"){
   xmlHttp.open(method, url+"?"+ params, sync);
   xmlHttp.onreadystatechange = readyStateHandler;
   xmlHttp.send(null);
}


// Methode POST
else{
   xmlHttp.open(method, url, sync);
   xmlHttp.setRequestHeader("Content-Type",
                            "application/x-www-form-urlencoded");
   xmlHttp.onload = readyHandler;
   xmlHttp.send(params);
}
```

Zusätzlich muss mit `setRequestHeader()` der Mime-Typ also der "`Content-Type`" gesetzt werden. Die Angabe des Mime-Typs "`application/x-www-form-urlencoded`" ist für die Übertragung der Formulardaten als Name-Werte-Paar ähnlich des URL-Encodings verantwortlich. So sind alle Name-Werte-Paare mit & verbunden und z. B. Leerzeichen durch + ersetzt.

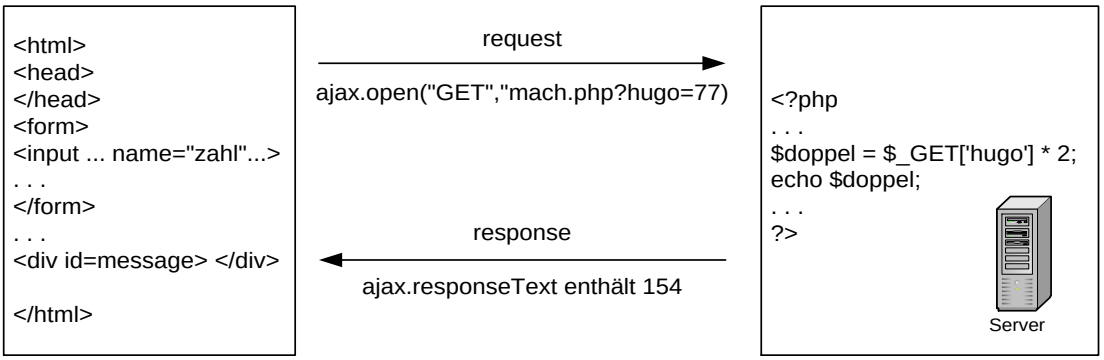
Die Methode `setRequestHeader()` kann aufgerufen werden, wenn der `readyState`=1 ist, das heißt nachdem die Methode `open()` aufgerufen wurde und bevor mit `send()` das Abschicken des Requests ausgelöst wurde.

8.6 XmlHttpRequest und PHP

Ein Einsatzgebiet für die Verwendung von Ajax ist, wenn Daten nachgeladen oder während der Eingabe Datenabfragen getätigt werden sollen. Hierfür bieten sich PHP-Skripte an. Diese können während der Initialisierung des XmlHttpRequest-Objekts als URL übergeben werden.

```
xmlHttp.open("GET", "mach.php", true);
```

Innerhalb des PHP-Skriptes kann man auf alle Angaben verzichten, die normalerweise in einem HTML-Dokument nicht fehlen sollten. Es wird über die Anweisung echo beliebiger Text an das aufrufende JavaScript-Programm zurückgegeben, der dann in der Eigenschaft responseText des XmlHttpRequest-Objekts zu finden ist.



Dabei ist der Programmierer für die Übergabe der Argumentenliste an das aufzurufende PHP-Skript selbst verantwortlich. Das PHP-Skript hat keine Möglichkeit auf die Formularfelder automatisch zugreifen zu können, wie es sonst der Fall war, wenn ein HTML-Dokument abgeschickt wird.

Es ist also notwendig die Inhalte der Formularfelder an die JavaScript-Funktion, die den Ajax-Request initialisiert, zu übergeben. Diese Name-Werte-Paare werden mit dem ?-Zeichen an die URL angehängt und weitere mit dem &-Zeichen gekoppelt. In dem unten angegebenen Beispiel sind varName und varZahl JavaScript-Variablen.

```
xmlHttp.open("GET", "mach.php?name="+varName+"&zahl="+varZahl);
```

Beachtet werden muss dabei, dass alle Name-Werte-Paare ohne Leerzeichen übergeben werden müssen. Normalerweise ersetzt der Browser diese automatisch durch %-Zeichen.

Allerdings kann man falls erforderlich verhindern, dass leere Name-Werte-Paare als Argumente an das PHP-Skript übergeben werden.

Genauso muss der Programmierer sicherstellen, dass die Zeichenkette, die die Eigenschaft responseText enthält, vernünftig in das Ursprungsskript eingebunden wird.

Die echo-Anweisung im PHP-Skript kann einmal oder mehrmals erfolgen. Der Text kommt immer als Ganzes in der Eigenschaft responseText an.

```
//die URI
echo "<p>angefordete URI: " . $_SERVER['REQUEST_URI'] . "</p>";
//die Methode
echo "<p>method: " . $_SERVER['REQUEST_METHOD'] . "</p>" ;
```

Außerdem können beliebige HTML-Tags übergeben und auf der Clientseite im bestehenden HTML-Dokument eingebettet werden.

8.7 Ajax mit JQuery

JQuery ist ein leistungsfähiges Framework, mit dem umfangreiche JavaScript-Funktionalität leicht erreicht werden kann. Dafür muss zuerst der JavaScript-Code des Frameworks eingebunden werden. Entweder durch Herunterladen der JavaScript-Dateien (http://jquery.com/download/ -> STRG A alles markieren und STRG C kopieren) oder durch Verweis auf externe Seiten:

```
<script
   src="https://ajax.googleapis.com/ajax/libs/jquery/x.x.x/jquery.min.js">
</script>

// x.x.x steht für die jeweilige Version
```

Es gibt die JQuery-Standardfunktionalitäten, in denen auch AJAX eingebunden ist. Daneben existieren Entwicklungen von JQuery, die sich auf das User Interface, auf mobile oder testing spezialisiert haben. Für jeden Bereich müssen dann die entsprechenden JavaScript-Bibliotheken in die Website eingebunden werden.

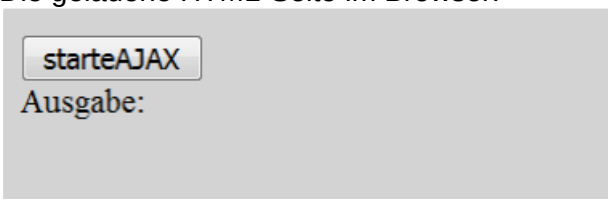
Die Bibliothek, sprich der JavaScript-Code kann auch im eigenen Ordner eingebunden werden:

```
<script src="./js/jquery-3.0.0.js"> </script>
```

Im Body-Tag sind ein Button und ein Ausgabebereich enthalten:

```
<form>
  <button type="button" id="starteAJAX"> starteAJAX </button>
</form>

Ausgabe: <div id="ausgabe"> </div>
```

Die geladene HTML-Seite im Browser:



In den JavaScript-Tag oder eine externe Datei wird der JQuery-Code für die Ausführung eines AJAX-Aufrufs eingebunden. Dafür wird immer das Ereignis document.ready (HTML-Seite ist fertig geladen) genutzt, um den JQuery-Code einzutragen.

```
<script>  jQuery(document).ready(function(){
     // weiterer JQuery-Code
  });
</script>
```

Üblicherweise wird JQuery durch das Dollarzeichen ersetzt:

```
$(document).ready(function(){
  // weiterer JQuery-Code
});
```

In JQuery wird eine id mit dem #-Zeichen markiert. Ein HTML-Tag wird einfach angegeben `'h1'` und eine Klasse mit Punkt bezeichnet `'.ueberschrift'`:

```
<button type="button" id="starteAJAX">
<h1 class="ueberschrift"> Titel </h1>
```

Wenn der Button angeklickt wird, soll der AJAX-Aufruf gestartet werden.

```
$('#starteAJAX').click(function(){
   // weiterer Code
});
```

Der eigentliche AJAX-Request wird gestartet mit

```
$.ajax({
   type: "GET",
   url: "server.php",
   data: {
      test: "Hallo Welt"
   }
});
```

`type` ist die verwendete Methode (get oder post), `url` ist das angeforderte PHP-Skript und `data` sind die übergebenen Name-Werte-Paare.

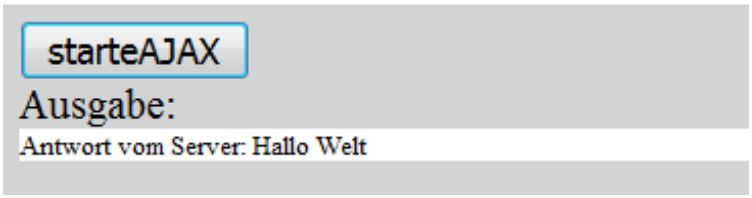
Der vollständige Code des Beispiels:

```
$(document).ready(function(){
   $('#starteAJAX').click(function(){
      let xmlRequest = $.ajax({
         type: "GET",
         url: "server.php",
         data: {
            test: "Hallo Welt"
         }
      });
      xmlRequest.done(function( responseText ) {
         $( "#ausgabe" ).html( responseText );
      });

      xmlRequest.fail(function( xmlRequest, textStatus ) {
         alert( "Request fehlgeschlagen: " + textStatus );
      });
   });
});
```

Inhalt der server.php:

```
<?php
   if(isset($_GET['test'])){
      echo "Antwort vom Server: " . $_GET['test'];
   }
?>
```

Nach dem Drücken des Buttons ergibt sich folgende Anzeige:



Die `done`- und `fail`-Methoden des XMLRequest-Objekts sind eine alternative Möglichkeit, um auf einen erfolgreichen (done) oder einen fehlerhaften (fail) Request zu reagieren.

Weitere Infos unter: http://api.jquery.com/jquery.ajax/