Orientation in Objects

) JavaScript)

Version 1.3.5 (01.10.2016, Autor: Frank Bongers, Webdimensions.de)

Weinheimer Straße 68
68309 Mannheim
http://www.oio.de

Inhaltsverzeichnis

1. Einführung

JavaScript kann als wichtigste Programmiersprache für den Webdesigner betrachtet werden, wenn es um Interaktivität auf der Clientseite, also beim User geht.

1.1. JavaScript – die Scriptsprache für den Browser

JavaScript ist eine objektorientierte (prototyp-basierte) Scriptsprache und wurde im Jahr 1995 von *Brendan Eich* für **Netscape** für die Erstellung von interaktiven Web-Seiten entwickelt. Trotz des Namens besteht keine enge Verwandschaft mit Java.

✔ Brendan Eich orientierte sich für die Syntax an **C**, für die funktionalen Aspekte an **Self** und lehnte die Prototyp-basierte Objektorientierung an die Sprache **Scheme** an.

Als Scriptsprache benötigt sie eine Laufzeitumgebung, die das Script interpretiert und ausführt. In unserem Fall ist diese Laufzeitumgebung der **Webbrowser**.

1.1.1. JavaScript - das Original von Netscape

Der **erste Browser**, der eine JavaScript-Laufzeitumgebung beinhaltete, war Netscape 2.0 (im Jahr 1995). In den darauf folgenden Jahren wurde die Sprache schnell verbessert und erweitert (Versionen JavaScript 1.1 bis 1.3).

Die Sprachversionen 1.4 bis 1.6 brachten nur punktuell Neues (erweitert wurden Fehlerbehandlung und damit verwandte Themen, was JavaScript in diesen Punkten näher an Java heranführte). Die letzte offiziell durch Mozilla eigenständig bennannte Version ist **JavaScript 1.8.6**. Seitdem folgt man den ECMA-Versionen.

Nach dem Niedergang von Netscape wurde die Weiterentwicklung von JavaScript von der **Mozilla-Foundation** übernommen.

1.1.2. JScript - der Gegenspieler von Microsoft

Microsoft benannte seine über *reverse engineering* erstellte JavaScript-Implementierungen von Anfang an mit der Bezeichnung **JScript**. Microsoft als „Verlierer“ des Scriptsprachen-Rennens glich sich allerdings früher an den **ECMA-Script-Standard** an.

Proprietäre Erweiterungen *erlaubt* ECMA-Script ausdrücklich (ohne sie natürlich zu Standards zu machen), und davon macht Microsoft seit Internet Explorer 4.0 (JScript 3.0) weidlich Gebrauch (Databinding, Editing, erweitertes DHTML usw.). Alle diese Erweiterungen (auch die gelungenen) sind allerdings in Nicht-Microsoft-Browsern nicht funktionsfähig!

✔ Der aktuelle Stand ist **JScript 10.0** (im Internet Explorer 10); JScript 9.0 (~ECMA5) im Internet Explorer 9 ist derzeit noch die verbreiteteste Version. Seit IE9 spricht Microsoft allerdings ebenfalls von „JavaScript“.

An dieser Stelle muss darauf hingewiesen werden, dass auch IE9 nicht vollständig kompatibel zu ECMA5 ist! Der ECMA5 **„strict mode“** wird erst durch IE10 unterstützt.

1.1.3. ECMAScript

Die „European Computer Manufacturers Association“ ECMA hat JavaScript im **Standard ECMA-262** als Standard spezifiziert. Die *erste Version* (1997) war annähernd kompatibel mit JavaScript 1.1. Sie wurde unter der Bezeichnung **ISO/IEC 16262** als Standard von der ISO übernommen und wurde seitdem mehrfach aktualisiert.

✔ **Meistimplementiert** ist *ECMAScript 5.1* (kurz: „ECMA5“, Juni 2011); siehe: www.ecma-international.org/publications/standards/

Seit JScript 2.0 (Microsoft) bzw. JavaScript 1.3 (Netscape) halten sich Microsoft und Netscape (bzw. Mozilla) an den ECMA-Standard. Alle aktuellen Browser (Chrome, Safari, Opera etc.) können ECMA5-kompatible Scripte ausführen.

✔ **Aktuelle Version:** Derzeit noch unvollständig implementiert, wurde die als *„Harmony“* oder *ECMAScript 6* bekannte Nachfolgeversion als

ECMAScript 2015 (im Sommer 2015) verabschiedet. Es soll jährliche Weiterentwicklungen des Standards als *ECMAScript 2016* etc. geben.

1.2. Plattformen für JavaScript

JavaScript Code kann in Web-Dokumente eingebunden werden und wird auf dem **Browser** beim Benutzer ausgeführt (clientseitig).

Alle aktuellen Webbrowser (auch deren mobile Versionen) unterstützen JavaScript (größtenteils in Form von ECMA5) – daher ist gewährleistet, dass die Mehrheit der Internet-Benutzer einen JavaScript-fähigen Browser benutzt (Ausnahmen sind verschiedene Screenreader).

✔ **Merke:** JavaScript kann jederzeit vom Client deaktiviert werden.

Es sind verschiedene JavaScript-Engines implementiert, deren Geschwindigkeit durch **„Just-In-Time"-Compiler** zunehmend optimiert werden.

Browser	**Aktuelle Script-Engine**
Firefox	SpiderMonkey (mit IonMonkey-Compiler)
Safari	Nitro (aka SquirrelFish)
Internet Explorer	Chakra (seit Vs. 9.0)
Opera	V8 (von Chrome) seit Vs. 15; Carakan (bis Vs. 14)
Chrome	V8

Der Haupteinsatz von JavaScript liegt auf der **Client-Seite**. Typische Anwendungen sind die Überprüfung von Benutzereingaben in Formularen, aber auch komplexe, interaktive User-Interfaces mit dem Nachladen von Daten, Animationen, interaktive Anpassungen der Darstellung (CSS) oder des Aufbaus (DOM) des HTML-Dokuments sind möglich und gängig.

✔ **Anmerkung:** Aktuell von Interesse ist auch eine *serverseitige* Anwendung von JavaScript – **node.js**.

Mehr hierzu: www.nodejs.org

1.3. Einbetten von JavaScript

Ein JavaScript ist in der Regel Teil einer HTML-Seite, oder ist mit ihr verknüpft. Für die Einbindung ist ein HTML-Tag zuständig, der `<script>`-Container.

1.3.1. Der `script`-Tag

Der Inhalt dieses Containers ist für den Browser als **Programmcode** gekennzeichnet, wird also gemäß der Syntax der in ihm enthaltenen Sprache ausgewertet.

Achtung: Im Inneren eines `<script>`-Containers gelten NICHT die Regeln von HTML!

- ✔ keine HTML-Kommentare wirken
- ✔ keine HTML-Entities werden expandiert
- ✔ keine HTML-Tags werden ausgeführt

1.3.2. Der `script`-Tag und seine Attribute

`type`-Attribut

- Das `type`-Attribut kennzeichnet den Mime-Typ des Blockinhalts, mit anderen Worten die Scriptsprache. In HTML 4 und XHTML gab es hierfür keinen Defaultwert, sodass das `type`-Attribut hier obligatorisch ist.

```
<script type="text/javascript">
   ...
</script>
```

- ✔ **Entfällt de facto:** In HTML5 *darf* das `type`-Attribut unterbleiben, da der MIME-Typ "`text/javascript`" zum Defaultwert erklärt wurde.

`src`-Attribut

Das `src`-Attribut nennt den URI einer externen Datei, die die in Zusammenhang mit dem Script-Container auszuführenden Programmanweisungen enthält. Der URL des Dokuments dient dabei als Basis-URL. Die externen Scriptdaten verhalten sich, als ob sie direkt im Container enthalten wären.

```
<script src="extern.js">
   // leer oder maximal JavaScript-Kommentare!
</script>
```

Der Scriptcontainer muss geschlossen werden und bleibt üblicherweise leer (Kommentare sind jedoch erlaubt).

Ein per `src`-Attribut zu ladendes Script *unterbricht* den Parsing-Vorgang, bis es **geladen und ausgeführt** ist. Vorher fährt der Browser nicht mit dem Rendering fort. Dieses Verhalten wird als **synchron** bezeichnet.

`defer`-Attribut

Das (seit HTML5 **boolesche**) `defer`-Attribut teilt dem Useragent mit, dass das Laden eines externen Scriptes bis nach Abschluss des Parsens des Dokuments aufgeschoben („deferred“) werden soll.

- *ohne* `defer`-Attribut (Default)
  Das Script soll (wie üblich) geladen und unmittelbar ausgeführt werden.

- `defer` oder `defer="true"`
  Das Script soll verzögert ausgewertet werden.

```
<script defer src="extern.js">
   // erst nach dem Ende des Parsens auszuführen.
</script>
```

Ein Useragent wird entsprechend den Scriptblock (vorläufig) überspringen und mit dem Parsing des Dokuments oder weiteren zu ladenden Scripten fortfahren, um so den Renderingvorgang zu beschleunigen. Das Script wird asynchron geladen. Der entsprechende Scriptblock wird **erst nach Benden des Parsings** des Quellcodes ausgeführt.

✔ Das `defer`-Attribut wird in HTML5 (wieder) propagiert und von allen aktuellen Browsern mit Ausnahme von Opera und einigen Mobilbrowsern unterstützt.

Achtung: Da `defer` in HTML5 als **boolesches Attribut** definiert ist, genügt seine *Anwesenheit* oder ein *beliebiger* Wert (anders gesagt: `defer="false"` aktiviert das Verhalten ebenfalls!!!)

async-Attribut

Neu in HTML5. Das boolesche `async`-Attribut (in Verbindung mit `src`) erlaubt das asynchrone Laden eines externen Scriptes. Per Default wird ein externes Script **synchron** geladen, was weitere Verarbeitung des Dokuments unterbricht, bis das Script geladen und ausgeführt ist.

- `async` oder `async="async"`
  Das Script soll *asynchron* geladen werden, was dem Browser erlaubt, unterdessen mit dem Rendering fortzufahren.

```
<script async src="extern.js">
   // asynchron zu laden.
</script>
```

Normalerweise blockiert ein Script während des Ladevorgangs das Laden und die Ausführung weiterer Scripte wie auch den Fortgang des Parsens des Dokuments. Mit `async="true"` wird das Script per Ajax geladen und erst nach (beliebig späterem) Eintreffen ausgeführt. Man beachte, dass **keine Abhängigkeiten** nachfolgend eingebundener Scripte (über Variablen, Klassen, Funktionsaufrufe etc.) bestehen sollten!

✔ Verwendet man gleichzeitig `async` und `defer`, so gilt für Browser, die asynchrones Laden nicht unterstützen, ein Fallback auf das (seit längerem existierende) `defer`-Verhalten.

Achtung: Da `async` in HTML5 als **boolesches Attribut** definiert ist, genügt seine *Anwesenheit* oder ein *beliebiger* Wert (anders gesagt: `async="false"` kann das Verhalten ungewollt *aktivieren*!!!)

2. Grundsyntax von JavaScript

2.1. Anweisungen (Statements)

Eine **Anweisung** (*statement*) kann im allereinfachsten Fall aus einem **Schlüsselwort** oder einem **Token** bestehen – die Regel ist aber der sogenannte "zusammengesetzte Ausdruck". Wir nehmen elementare Ausdrücke (*Token*) und verknüpfen diese mit Hilfe von **Operatoren**.

In JavaScript wird eine Anweisung mit einem **Semikolon** beendet.

```
erste Anweisung;
zweite Anweisung;
```

Achtung: Ein fehlendes Semikolon wird durch den JS-Parser ergänzt.

2.2. Blocks

Ein Anweisungsblock wird durch geschweifte Klammern begrenzt:

```
{
   erste Anweisung;
   zweite Anweisung;
}
```

Mehrere als Block zusammengefasste Stements werden wie ein einzelnes Statement betrachtet. Blöcke werden daher gewöhnlich in Zusammenhang mit **Kontrollstrukturen** verwendet:

```
if(bedingung) {
   ...
}

while(bedingung) {
   ...
}
```

Achtung: Blöcke besitzen **keinen eigenen Scope**, wie es in anderen Programmiersprachen der Fall ist. Scopes existieren in JavaScript ausschließlich in Zusammenhang mit Funktionslaufzeiten.

2.3. Operatoren

Die Operatoren der Sprache C hat man in JavaScript übernommen. Die Tabelle zeigt die Operatoren von JavaScript sortiert nach **Priorität** (hiermit ist die „Bindungskraft“ des Operators zwischen den Operanden gemeint), beginnend mit der höchsten Priorität.

Operator	**Funktion**
() []	Funktionsaufruf Array-Member
! ~ - ++ --	Logisch NOT Bitweise Negation Numerische Negation Increment, Decrement
* / %	Multiplikation, Division, Modulo
+ -	Addition, Subtraktion
<< >> >>>	Bitweise Verschiebung
< <= > >=	kleiner, kleiner gleich, größer, größer gleich
== !=	gleich, nicht gleich
&	Bitweises UND
^	Bitweises exklusives ODER
|	Bitweises ODER
&&	Logisch UND
||	Logisch ODER
?:	Bedingungsoperator
= += -= *= /= %= <<= >>= >>>= &= ^= |=	Zuweisung
,	Komma (Properties, Argumente)

2.4. JavaScript-Kommentare

Im Inneren des JavaScript-Blockes können die aus C++ bekannten Kommentare verwendet werden.

Beispiel:

Einzeiliger Kommentar mit // am Anfang oder irgendwo in einer Zeile. Der Kommentar gilt nicht mehr für die folgende Zeile (Zeilenumbruch beendet ihn).

```
var x = 3     // Rest der Zeile ist Kommentar

// diese Zeile ist auch auskommentiert
```

Beispiel:

Mehrzeiliger Kommentar mit /* ... */

```
/* Kommentare dieser
   Form können sich
   über mehrere Zeilen erstrecken. */
```

2.5. Variable

Variablen dienen in JavaScript, genau wie in anderen Sprachen, dem Speichern von Werten. Man braucht in JavaScript nicht vorher zu entscheiden, welchen **Typ** von Wert der Variablen zugewiesen werden soll.

Bei anderen Programmiersprachen (C, C++, Perl) ist dies erforderlich – man spricht dann von **"strong-typed"** (stark typisiert) Variablen.

JavaScript ist im Gegensatz dazu "**loose-typed**" (schwach typisiert). Das bedeutet jedoch nicht, dass *keine* Typisierung besteht - nur dass an einen Variablencontainer kein Typ fest („strong“) gebunden ist. Datentypen als solche existieren in JavaScript dennoch.

2.5.1. Variablen – Deklaration und Initialisierung

Eine Variable wird in JavaScript mit dem Schlüsselwort `var` deklariert:

```
var test;
```

Sie kann dann auch gleich mit einem Wert befüllt werden. Die Wertzuweisung erfolgt von rechts (*RHS*) nach links (*LHS*). Als Zuweisungeoperator dient das Gleichheitszeichen:

```
var test = "ein Wert";
```

Es können mehrere Variable innerhalb einer Anweisung deklariert und gleichzeitig oder später befüllt werden:

```
var strEins, strZwei, strDrei = " ";

strEins = "Hallo";
strZwei = "Welt";
```

Hierbei darf eine Anweisung auch über mehrere Zeilen ausgedehnt werden (am Komma-Operator ist stets ein Umbruch gestattet):

```
var num1 = 1,
    num2 = 2,
    num3 = 3;
```

Auf der RHS kann ein Wert oder ein Ausdruck stehen:

```
var ausgabe = strEins + strDrei + strZwei;
```

Achtung: Falls *RHS* eine Variable steht, muss diese deklariert sein.

2.5.2. Hoisting von Variablen

Variablendedeklarationen werden **„gehoisted“**, also virtuell an dan Anfang des Scriptblocks verschoben. Aus diesem Grund ist es zu überlegen, ob man die Deklaration gleich zu Beginn des Scriptes vornimmt. Der Übersicht halber ist dies ohnehin anzuraten. So sieht der geschriebene Scriptblock aus:

```
<script>

// viel Code...

// Deklaration mit Zuweisung:
var x = 42;

</script>
```

Dieser Zustand stellt sich nach dem **Hoisting** ein – beachten Sie, dass nur die Deklaration der Variable gehoistet wird, nicht die Wertzuweisung.:

```
<script>

// gehoistete Deklaration:
var x;

// viel Code...

// Zuweisung an gehoistete Variable:
x = 42;

</script>
```

✔ Der Sinn ist, dass eine irgendwo im Script deklarierte Variable stets zu Beginn des Scriptes bekanntgemacht wird, auch wenn ihr Wert erst später zur Laufzeit festgelegt wird.

In den meisten Fällen ist das Hoisting von Variablendeklarationen für den Developer irrelevant, da es sich nicht auf den Verlauf des Scripts auswirkt.

Achtung: Auch Funktionsdeklarationen werden gehoistet.

✔ Eine weitere Folge des Hoisting besteht in der **Nachdeklaration** von nichtdeklarierten Variablen, die LHS im Script auftreten (in RHS verursachen sie einen Fehler):

2.5.3. Variablen – Alternative Wertzuweisung

Die Wertzuweisung an eine Variable erfolgt normalerweise direkt. Es kann jedoch auch der Wert einer anderen Variable zugewiesen werden:

```
var b = a;  // a muss deklariert sein
```

Will man den Wert a nur unter bestimmten Umständen zuweisen (beispielsweise, wenn er nicht undefiniert oder leer ist), dann muss vor der Zuweisung geprüft werden. Dies kann mittels des ternären Operators geschehen:

```
var b = a? a : "Default";
```

Noch einfacher ist der Einsatz des Booleschen OR-Operators:

```
var b = a || "Default";
```

2.6. Datentypen und Typisierung

In JavaScript unterscheident man zwischen primitiven und komplexen Datentypen. All diese Typen sind als Inhalt von Variablen erlaubt und können diesen zugewiesen werden.

Primitive Typen:

- Zeichenketten (Typ *string*)
- Zahlen (Typ *number*)
- Spezialwerte für Zahlen: `+INF`, `-INF`, `NaN`
- Wahrheitswerte (Typ *boolean*)
- Undefinierter Wert (`undefined`)
- Nullwert (`null`)

Komplexe Typen:

- Objekte
- Arrays
- Funktionen

All diese Werte sind als Inhalt von Variablen erlaubt und können diesen zugewiesen werden.

2.6.1. Gültige Werte für Variablen (Typen)

Wert	**Typ**
4711	Ganzzahlen* (number)
3.99	Kommazahlen* (number)
The lazy dog jumps over the fox.	Zeichenketten (string)

true, false	Wahrheitswerte (boolean)

*JavaScript unterscheidet bei **Zahlen** nicht zwischen Ganzzahlen (Integer) und Kommazahlen (Float) sondern kennt nur den pauschalen Typ number. Es gibt somit in JavaScript drei "primitive" Datentypen.

2.6.2. Variablennamen

Der Bezeichner (volkstümlich "Name") der Variable ist in weiten Grenzen **frei wählbar**. Die erlaubte Länge des Namens ist im Prinzip unbegrenzt.

✔ Es sollten **"sprechende" Bezeichner** für die Variablen verwendet werden, die ihren Inhalt sinngemäß kennzeichnen.

Natürlich gibt es Einschränkungen und Regeln die zu beachten sind. Zum Einen dürfen keine der sogenannten reservierten Begriffe (wie Bezeichner von Programmbefehlen etc.) als Namen verwendet werden.

Ansonsten dürfen Variablennamen **beliebige Ziffern und Buchstaben** enthalten – die Buchstaben dürfen dabei aber **keine Umlaute** oder Ähnliches sein.

Weiterhin erlaubt sind Bindestrich und Unterstrich - aber keine der anderen Interpunktionszeichen, **keine Leerzeichen**, keine Steuerzeichen oder Anführungszeichen.

Variablenbezeichner dürfen *nicht*(!) mit einer Ziffer, sondern *müssen*(!) mit einem Buchstaben, dem Unterstrich oder dem $-Zeichen beginnen.

In JavaScript wird bei Variablennamen zwischen **Groß- und Kleinschreibung unterschieden**. Gültige Variablennamen in JavaScript sind beispielsweise:

```
x

gehalt

Umsatz

_xpos
```

```
wert45

$meine_variable
```

2.6.3. Variablenbennenung

Das Prinzip der **"sprechenden" Variablennamen** kann noch erweitert werden, indem wir den Typ des zu speichenden Wertes im Namen der Variablen widerspiegeln. Dies ist Konvention in einigen Sprachen, jedoch kein "Muss".

Auf folgende Präfixe hat man sich (z.B. in Visual Basic) allgemein geeinigt:

```
str       String          strVorname / sVorname

bln       Boolean         blnGueltig / bGueltig

int       Integer         intZaehler / iZaehler

obj       Object          objMeinObj / oMeinObj
```

2.7. Zeichenketten (type „string“)

Zeichenketten als Literale werden durch einfache oder doppelte Anführungszeichen begrenzt:

```
"Ich bin ein String in JavaScript!"

'Ich auch!'
```

Eine Zeichenkette ist ein unveränderliches Gebilde (*immutable*), das aus keinem, einen oder vielen Zeichen (*characters*) besteht. Der Typ einer Zeichenkette ist „string“:

```
typeof "ein string"; // "string"
```

Für einen String sind **Methoden** verfügbar, die unmittelbar auf ihn angewandt werden können – streng genommen wird hierbei das

Stringliteral in ein Stringobjekt umgewandelt und wieder zurück. In den meisten Fällen ist das Ergebnis also wieder ein String (für `split()` ergibt sich ein Array):

```
"hello".charAt(0) // "h"

"hello".toUpperCase() // "HELLO"

"Hello".toLowerCase() // "hello"

"hello".replace(/e|o/g, "x") // "hxllx"

 1,2,3".split(",") // ["1", "2", "3"]
```

Strings besitzen darüber hinaus ein eine Eigenschaft `length`:

```
"Hallo".length // 5
"".length // 0
```

Ein leerer String ist im Booleschen Kontext *false*:

```
!"" // true

!"hello" // false

!"true" // false

!new Boolean(false) // false
```

Andere Typen können stets in Strings umgewandelt werden:

```
false.toString() // 'false'
[1, 2, 3].toString(); // '1,2,3'
```

Für Strings existiert ein Konstruktor, der direkt ein Stringobjekt erzeugt:

```
var meinString = new String("ein neuer String");
```

2.8. Zahlen (type number)

Beispiel für Zahlen in JavaScript:

```
12      // Ganzzahl

3.543   // Fließkommazahl
```

Zahlen sind in JavaScript stets Werte gemäß IEEE 754 (*double-precision 64-bit format*). Es wird nicht zwischen Ganzzahl und Fließkommazahl unterschieden. Auf Zahlen sind alle aus C hierfür bekannten Operatoren anwendbar:

```
+, -, *, /, %, =, +=, -=, *=, /=, ++, --
```

Der `type` einer Zahl ist "number". Dies kann über den typeof-Operator festgestellt werden:

```
typeof 12;     // "number"

typeof 3.543;  // "number"
```

Aus einem beliebigen Wert kann mittels der Methoden `parseInt()` und `parseFloat()` eine Zahl extrahiert werden. Dies ist jedoch nicht immer möglich. In diesem Fall lautet das Ergebnis `NaN`:

```
parseInt("hello", 10);        // NaN


isNaN(parseInt("hello", 10)); // true
```

Das Teilen durch 0 ergibt den Ausdruck `Infinity`. Ist der Operand negativ, so ist das Ergebnis `–Infinity`.

```
1 / 0;    // Infinity
```

Sowohl `NaN` als auch `Infinity` und `–Infinity` sind vom Typ “number”.

```
typeof NaN;      // "number"


typeof Infinity; // "number"
```

… wobei `NaN`-Werte nicht mit einander gleich sind:

```
NaN == NaN; // false (!)
```

Daher existiert eine Methode `isNaN()`, um entsprechende Werte im Rahmen eines Vergleichs erkennen zu können.

Zahlen können auch als Objekte über einen **Konstruktor** erstellt werden:

```
new Number(10);
```

2.9. Wahrheitswert (type „boolean“)

Ein boolescher Wert in JavaScript ist entweder *true* oder *false*:

```
if ( true ) console.log("immer!")

if ( false ) console.log("niemals!")
```

Ein Boolescher Wert kann ebenfalls über einen **Konstruktor** erzuegt werden:

```
new Boolean(false);
```

Achtung – es gilt:

```
typeof false;              // "boolean"
typeof new Boolean(false); // "object"
```

2.10. Typisierung –Typkonversion und Casting

In vielen Fällen nimmt JavaScript eine automatische Typkonvertierung vor. Meist möchte man dies etwas besser im Griff haben. Für diesen Zweck existieren ein paar Tricks sowie „legale“ Castingfunktionen.

Casting nach String

Zahlen und andere Werte können einfach in Strings umgewandelt werden, indem man sie mit dem leeren String verknüpft:

```
"" + 1 + 2; // "12"

"" + (1 + 2); // "3"
```

```
"" + 0.0000001; // "1e-7"

"" + false; // "false"
```

Casting nach Number

Mittels des unären Plus-Operators kann ein Wert in eine Zahl gecastet werden.

```
+'10' === 10; // true

+false === 0; // true

+true === 1;  // true
```

Casting nach Boolean

Durch den doppelt angewandten unären Negationsoperator kann jeder Typ nach Boolsch gecastet werden:

```
!!'foo';   // true

!!'';      // false

!!'0';     // true

!!'1';     // true

!!'-1'     // true

!!{};      // true

!!true;    // true
```

Casting mit Konstruktorfunktionen

Typecasting kann auch mittels der **Konstruktorfunktionen** für primitive Typen geschehen:

```
Number('10') === 10;

String(1) + String(2); //"12"

String(1 + 2); //"3"
```

2.10.1. Typisierung –Vergleichsoperatoren

In JavaScript existieren zwei grundsätzliche Möglichkeiten, die Gleichheit zweier Größen festzustellen:

- Gleichheit (*equality*) mit ==
- Gleichheit bei Typgleichheit (*strict equality*) mit ===

✔ **Achtung** – der Vergleich auf Gleichheit mit == bewirkt bei typverschiedenen Operanden eine implizite Typkonvertierung (*coercing*):

```
""          ==   "0"          // false
0           ==   ""           // true
0           ==   "0"          // true
false       ==   "false"      // false
false       ==   "0"          // true
false       ==   undefined    // false
false       ==   null         // false
null        ==   undefined    // true
" \t\r\n"   ==   0            // true
```

Anmerkung: Da sich beim Vergleich bestimmter Werte im Booleschen Kontext *false* ergibt, ohne dass tatsächlich *false* vorliegt (leere Strings, die Zahl 0, *null* und *undefined*), bezeichnet man diese Werte als „*falsy*“. Analog existiert der Ausdruck „*truthy*“.

Der Operator für *strict equality* wirkt ebensogut, jedoch **ohne Typkonvertierung**, und ist daher in der Regel vorzuziehen:

```
""          ===   "0"          // false
0           ===   ""           // false
0           ===   "0"          // false
false       ===   "false"      // false
false       ===   "0"          // false
false       ===   undefined    // false
false       ===   null         // false
null        ===   undefined    // false
" \t\r\n"   ===   0            // false
```

Zu beachten beim Vergleich zwischen Objekten ist, dass nicht auf Gleichartigkeit sondern auf **Identität** geprüft wird (d.h. gleiche Instanz):

```
{} === {};                    // false
new String('foo') === 'foo'; // false
new Number(10) === 10;       // false
var foo = {};
foo === foo;                  // true (identisch)
```

2.10.2. Typisierung – instanceof- und typeof-Operator

Der **typeof**-Operator gibt den Basistyp eines Wertes zurück, zeigt sich damit aber nicht sehr auskunftsfreudig. Gerade bei Objekten ist es wichtiger, den Konstruktor (die „Klasse“) des Objektes zu kennen. In der folgenden Tabelle sind Typ und Klasse verschiedener Werte aufgelistet:

Wert	**Typ (typeof)**	**Klasse**
`"foo"`	string	String
`new String("foo")`	object	String

`1.2`	number	Number
`new Number(1.2)`	object	Number
`true`	boolean	Boolean
`new Boolean(true)`	object	Boolean
`new Date()`	object	Date
`new Error()`	object	Error
`[1,2,3]`	object	Array
`new Array(1, 2, 3)`	object	Array
`new Function("")`	function	Function
`/abc/g`	object	RegExp
`new RegExp("meow")`	object	RegExp
`{}`	object	Object
`new Object()`	object	Object

Die Klasse lässt sich aus dem internen `[[Class]]`-Property eines Objektes auslesen. Das geschieht mit Hilfe der `toString`-Methode von `Object.prototype`.

Hier wird das zu prüfende Objekt als `obj` übergeben:

```
var cls = Object.prototype
                .toString
                .call(obj)
                .slice(8, -1);
```

Folgende Werte können sich ergeben: Arguments, Array, Boolean, Date, Error, Function, JSON, Math, Number, Object, RegExp, String.

Der **instanceof**-Operator vergleicht die **Konstruktoren** zweier Operanden.

```
new String('foo') instanceof String; // true
new String('foo') instanceof Object; // true

'foo' instanceof String; // false
'foo' instanceof Object; // false
```

Verwendbar ist **instanceof** im Wesentlichen nur beim Vergleich selbstdefinierter Objekte.

```
function Foo() {}
function Bar() {}

new Bar() instanceof Bar; // true
new Bar() instanceof Foo; // false

// Instanz von Foo an Bar.prototype zuweisen:
Bar.prototype = new Foo();

// daraus folgt dann:
new Bar() instanceof Bar; // true
new Bar() instanceof Foo; // true
```

3. Kontrollstrukturen

3.1. Bedingungen

Eine Entscheidung wird wie in eigentlich allen Programmiersprachen durch das Schlüsselwort `if` gekennzeichnet. Nach dem `if` hat die auszuwertende Bedingung zu stehen.

Anschließend folgt eine Anweisung (oder ein Block von Anweisungen), der ausgeführt wird, wenn die Bedingung erfüllt ist. Ist die Bedingung nicht erfüllt, so wird diese Anweisung/Block übersprungen, und das Programm fährt mit der nächstfolgenden Anweisung fort.

In JavaScript geht das so:

```
if (Bedingung) {
   statements
}
```

Eine Bedingung kann auch verwendet werden, um zwischen zwei Anweisungen (Anweisungsblöcken) zu wählen.

Der eine Zweig (`true`) wird gewählt, wenn die Bedingung erfüllt ist, und der bedingte Anweisungsblock durchlaufen – hier gibt es keinen Unterschied zu vorhin.

Man bietet jetzt aber für den Fall, dass die Bedingung **nicht** erfüllt ist, dem Programm einen **alternativen Anweisungsblock** an, der durch das Schlüsselwort `else` gekennzeichnet ist. Dieser wird dann und nur dann ausgeführt, wenn die Auswertung der Bedingung `false` ergibt.

In JavaScript geht das so:

```
if (Bedingung) {
   statements
} else {
   statements
}
```

3.2. Schleifen

Oft kommt man nicht ohne die **mehrfache Ausführung** einer Anweisung oder eines Anweisungsblockes aus. Das Konstrukt, das diese Wiederholung ermöglicht, ist die **Schleife**.

Natürlich wäre es eigentlich überhaupt kein Problem, eine Anweisung in einem Programm zu wiederholen – im Prinzip braucht man sie nur mehrmals hintereinander einzufügen. Es gibt einen dreifachen Nachteil, wenn man diesen Weg geht:

a) man legt sich fest auf die exakte Anzahl der Wiederholungen

b) die Menge des Quellcodes und damit die Dateigröße wächst

c) man hat mehr Gelegenheiten für Tippfehler ...

Da Programmierer immer Wert auf Eleganz gelegt haben (und auf Effizienz – was man auch als Faulheit bezeichnen kann), kürzt man solche Wiederholungen mit Hilfe einer Programmschleife ab.

3.2.1. while-Schleife

Der Anweisungsblock einer `while`-Schleife wird nur dann ausgeführt und nur dann wiederholt, wenn und solange die Schleifenbedingung erfüllt ist. Es ist also denkbar, dass die Anweisungen des Schleifenblocks überhaupt nicht ausgefürt werden, wenn die Anfangsbedingung nicht erfüllt ist.

```
while (Bedingung) {
   statements
}
```

3.2.2. do-while-Schleife

Anders als bei einer `while`-Schleife wird der Anweisungsblock einer do-while-Schleife mindestens einmal ausgeführt aber nur dann wiederholt, wenn die Schleifenbedingung erfüllt ist:

```
do {
   statements
} while (Bedingung)
```

3.2.3. for-Schleife

Die for-Schleife ist ein Schleifentyp, der das Bilden von Zählschleifen erleichtert. Sie sieht auf den ersten Blick komplex aus:

```
for (Initialisierung; Bedingung; Inkrement) {
   statements
}
```

Die Zählervariable wird traditionell mit `i` bezeichnet, kann aber auch einen beliebigen anderen Namen erhalten (falls z.B. der Bezeichner `i` schon in Gebrauch ist). Sie kann auf jeden beliebigen Startwert gesetzt werden – in der Regel wird es aber entweder 0 oder 1 sein.

✔ for-Schleifen sind besonders bei der Verarbeitung von **Arrays** von Bedeutung.

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

for(var i = 0, len = list.length; i < len; i++) {
    console.log(list[i]);
}
```

4. Arrays

In vielen Fällen ist es erforderlich, eine große Anzahl gleichartiger Variablen zu besitzen, um beispielsweise die Ergebnisse einer Messreihe

speichern zu können. Natürlich könnte man diese Variablen einzeln deklarieren – das wäre aber viel Arbeit.

4.1. Arrays und Arraymethoden

Statt einer "Schachtel", in der jeweils nur ein Wert Platz findet – was unserer "normalen" Variablen entspräche -, bräuchten wir ein Gebilde, das eher einem "Setzkasten" gleichkommt. Wir könnten dann die einzelnen "Fächer" dieses Setzkastens mit Werten füllen und hätten sie damit gleich in einer "Reihenfolge".

Aufgrund dieser bekannten Reihenfolge müssten wir, statt jeden Wert einzeln, nur dem Kasten einen Namen geben – und natürlich wissen, in welchem seiner Fächer der gesuchte Wert liegt.

Weil dieses Prinzip so sinnvoll und nützlich ist, ist er Bestandteil eigentlich aller Programmiersprachen - solche "Multivariablen" bezeichnet man als **Array**. Ein Array erhält einen Namen, genau wie eine gewöhnliche Variable. Die Fächer eines Arrays sind numeriert.

Um an den Wert eines Arrayfaches heranzukommen, muss man zwei Dinge wissen: den **Namen des Arrays**, und die **Indexziffer** des Faches. Es hat sich für fast alle Sprachen eingebürgert, die Indexziffer eines Arrayfaches in eckige Klammern hinter den Arraynamen zu setzen.

4.1.1. Ein Array deklarieren

In JavaScript müssen wir die **Konstruktorfunktion** für Arrays bemühen (so etwas wie ein "allgemeiner Bauplan für Arrays"), diese wird über die Schlüsselworte `new Array` aufgerufen (`Array` muss groß geschrieben werden!).

Ein Array wird wie folgt erzeugt und wird in eine normale Variable gespeichert:

```
// erzeugt ein leeres Array
var woche = new Array();


// leeres Array, 7 Fächer
var mein_array = new Array(7);
```

✔ Ein **Konstruktor** entspricht ungefähr einer Objektklasse, er legt fest, dass alle neugeschaffenen Arrayinstanzen bestimmte grundlegende Eigenschaften und Methoden besitzen.

Die „Multivariable“ speichert ihre Werte in „Fächern“, die über numerische Indizes erreicht werden. Die Nummerierung beginnt bei 0 (erstes Fach) und wird in eckigen Klammern an den Arraynamen angehängt:

```
woche[0] = "Montag";
```

Analog erfolgt das Auslesen:

```
alert(woche[0]); // -> Montag
```

Weitere Fächer können gefüllt werden, indem einfach der entsprechende Index angesprochen wird. Hierbei können auch Fächer übersprungen werden.

4.1.2. Modifizieren mit push(), pop(), shift() und unshift()

Oft benötigt wird das Feature, einen Wert an das Array anzuhängen. Hierfür wird die `push()`-Methode des Array eingesetzt:

```
woche.push("Dienstag");
```

Man braucht nicht zu wissen, das wievielte Fach hierbei befüllt wird. Die Zahl der Arrayfächer lässt sich jedoch mit der `length`-Eigenschaft auslesen:

```
alert(woche.length); // -> Alert: 2
```

Auch das Entfernen von Arrayfächern ist möglich. Mit `pop()` wird das *letzte* Fach, mit `shift()` das *erste* Fach entfernt:

```
woche.pop();   // Entfernen des letzten Faches

woche.shift(); // Entfernen des ersten Faches
```

Unser Array hat jetzt die Länge 0, da es keine Fächer mehr besitzt – es ist aber nach wie vor ein Array.

Statt wie mit `push()` am Ende können wir auch am **Anfang** des Arrays einen Wert hinzufügen. Dies geschieht mit `unshift()`:

```
woche.unshift("Montag");  // vor dem 1. Fach einfügen
```

Alle folgenden Fächer (zurzeit sind keine da) würden hierbei um einen Indexwert nach hinten geschoben.

Weitere Arraymethoden

Bekannte weitere Arraymethoden sind `reverse()`, `sort()`, `splice()`, `concat()`, `join()`, `slice()`, `toString()`.

In **ES5** kommen weitere Methoden dazu:

`indexOf()`, `lastIndexOf()`, `every()`, `filter()`, `forEach()`, `map()`, `reduce()`, `reduceRight()` und `some()`.

Ausführliche Erläuterungen finden Sie bei: https//developer.mozilla.org/

4.1.3. Arrayliterale

Ein leeres Array kann aber auch als **Literal** geschrieben werden. Das Wesentliche ist, dass ein Array nicht „erzeugt“ wird (beispielsweise über `new Array()`), sondern einfach „hingeschrieben“ werden kann:

```
var woche = []; // leeres Array wie mit new Array()
```

Hier ist es möglich, Inhalte gleich mit zu übergeben. Diese werden durch Komma getrennt:

```
var woche = ["Montag","Dienstag","Mittwoch", ... ];
```

Ebenso ist möglich (und üblich), ein Literal über mehrere Zeilen zu schreiben:

```
var woche = [
    "Montag",    // Komma nach dem Wert
    "Dienstag",  // Komma nach dem Wert
```

```
    "Mittwoch",  // Komma nach dem Wert
    // ...,
    "Sonntag"    // kein Komma nach letztem Wert
];
```

Wichtig – nach dem letzten Wert steht **kein Komma**!

4.2. Arrays und Schleifen

Um über die Fächer eines Arrays zu iterieren, bietet sich der Einsatz von Schleifen an. Hier existieren drei Möglichkeiten, nämlich die for-Schleife, die eigentlich für Objekte gedachte `for-in`-Schleife und die Arraymethode `Array.forEach()`.

4.2.1. Arrays und for-Schleifen

Nehmen wir ein Arrayliteral (zwei *leere* Fächer!):

```
var meinArray = ["Rosen","Tulpen", , ,"Nelken"];
```

…und lesen es mit einer for-Schleife aus:

```
// auslesen mit for-Schleife:
for(var i=0,len=meinArray.length; i<len; i++) {
   console.log(i," enthält: ", meinArray2[i]);
}
```

Dies gibt folgendes aus:

```
0 enthält: Rosen
1 enthält: Tulpen
2 enthält: undefined
3 enthält: undefined
4 enthält: Nelken
```

✔ Die `for`-Schleife liest auch die **leeren Fächer** aus.

4.2.2. Arrays und for-in-Schleifen

Anders verhält sich die `for-in`-Schleife für das gleiche Arrayliteral:

```
var meinArray = ["Rosen","Tulpen", , ,"Nelken"];

// for-in-Schleife über Objektproperties:
for(i in meinArray) {
   console.log(i, " enthält: ", meinArray2[i]);
}
```

Dies ergibt folgendes

```
0 enthält: Rosen
1 enthält: Tulpen
4 enthält: Nelken
```

Die `for-in`-Schleife überspringt die leeren Fächer.

4.2.3. Die Arraymethode forEach() als implizite Schleife

Ideal ist die implizite Schleife, die sich durch die Arraymethode `forEach` ermöglicht. Diese Methode ist eine **Higher-Order-Function**, die ein Funktionsobjekt entgegennimmt. Dieses erhält drei Argumente:

1. der Index des aktuellen Feldes
2. dessen Wert
3. Referenz auf das Array

```
var meinArray = ["Rosen","Tulpen", , ,"Nelken"];

// Arraymethode forEach:
meinArray.forEach(function(val, ind, arr){
     console.log(val," an Index ", ind);
});
```

Hier wird folgendes ausgegeben:

```
Rosen an Index 0
Tulpen an Index 1
Nelken an Index 4
```

```
// Arraymethode forEach:
meinArray.forEach(function(val,ind,arr){
     console.log(arr[ind]);
});
```

Hier wird folgendes ausgegeben:

```
Rosen an Index 0
Rosen
Tulpen an Index 1
Tulpen
Nelken an Index 4
Nelken
```

5. Funktionen und Scope

Ein Grundparadigma von JavaScript ist die Behandlung von Funktionen (also Funktionsobjekten) als reguläre Werte (*1st class citizens*).

Funktionen können daher als Wert einer Variable zugewiesen werden, als Parameter an eine andere Funktion weitergereicht und als Rückgabewert aus einer Funktion hinausgereicht werden.

✔ Dieses Prinzip ermöglicht es letztlich, in JavaScript auch funktional zu programmieren, es bildet die Grundlage für Closures und auch für die dynamische Veränderung von Objekten (durch Zuweisen oder Überschreiben von Methoden).

✔ Eine Folge davon ist, dass der Ausführungskontext einer Funktion nicht von vorherein feststeht, bzw. wechseln kann.

5.1. Funktionen – Funktions-Deklarationen

Anmerkung: Variablendeklarationen (nicht die Wertzuweisungen) werden „gehoistet“, d.h. an den Anfang des Scriptblocks verschoben.

Vor Hoisting (ursprünglich übergebener Code):

```
// irgendwo (weit unten) im Scriptblock:
var test = "Test";
```

Nach Hoisting:

```
// Gehoistet an den Beginn des Scriptblocks
var test;

// irgendwo (weit unten) im Scriptblock:
test = "Test";
```

Die Wertzuweisung verbleibt an der ursprünglichen Position im Script!

✔ Das **Hoisting** ist ein virtueller Vorgang, der im Rahmen eines ersten Durchgangs durch das Script geschieht. In diesem Zusammenhang werden alle LHS-Symbole, die im Block deklariert werden „gesammelt“ und katalogisiert.

5.1.1. Deklarieren einer Funktion

Eine Funktionsdeklaration ist im Prinzip ein benannter Anweisungsblock.

```
// function declaration
function func1(arg) {
   // globale Variable ausloggen:
   console.log("func1 liest: ", test);
   // arg ausloggen:
   console.log("func1 Argument arg: ", arg);
   // das Funktionsobjekt selbst:
   console.log("func1: ",func1);
}
```

Die Funktion wird aufgerufen

```
// Aufruf der Funktion func1:
func1("Hallo");
```

… und anschließend überschrieben:

```
function func1(arg) {
   // mach was anderes
}
```

… mit dem Hintergedanken, dass der nun folgende Aufruf etwas anderes tun soll:

```
// erneuter Aufruf der Funktion func1:
func1("Hallo");
```

Beide Aufrufe tun jedoch **dasselbe**.

✔ Der Grund ist, dass Funktionsdeklarationen, ebenso wie Variablendeklarationen **„gehoistet“** werden. Sie verhalten sich also *beide* so, als ob sie am Beginn des Scriptblocks stünden.

Deshalb überschreibt die zweite Deklaration die erste *unmittelbar*. **Auch der erste Aufruf verwendet die zweite Funktionsdeklaration.**

5.2. Funktionen in Variablen – Funktionsstatements

Ein Funtionsstatement entspricht der *Zuweisung* eines Funktionsobjekts an eine Variable.

✔ Auch hier geschieht **Hoisting**, allerdings ist davon nur die Variable, nicht die Zuweisung betroffen.

Hier ist die Variable am Anfang des Scriptblocks. Es spielt keine Rolle, ob sie eigentlich später deklariert wird. Es wird aber nicht ihr Wert verschoben; dessen Zuweisung geschieht an der ursprünglichen Stelle im Script.

```
// diese Variable wurde hierher gehoistet:
var func2;

// erstes function statement:
func2 = function(arg){
   // globale Variable ausloggen:
   console.log("func2 liest: ", test);
   // arg ausloggen:
   console.log("func2 Argument arg: ", arg);
   // das Funktionsobjekt selbst:
   console.log("func2: ",func2);
}

// Aufruf der Funktion func2:
func2("Welt");

// neue Funktion in func2 (zweites function statement):
func2 = function(arg){
      console.log("Yeah");
}
```

```
// Aufruf der Funktion func2:
func2(); // Yeah.
```

In diesem Fall ergibt der zweite Funktionsaufruf etwas anderes, da der Inhalt der Variable in der Zweischenzeit durch ein anderes Funktionsobjekt überschrieben wurde.

5.3. Funktionen – Globale vs. lokale Variable

Im „global scope“ deklarierte Variable sind stets als Eigenschaften des window-Objekts erreichbar.

```
var test; // global deklariert
alert(test==window.test); // ergibt true
```

Dies ist unabhängig davon, ob die Variable explizit mit var deklariert wurde, oder implizit durch Verwendung eingeführt wurde.

```
test2 = "Ich bin implizit deklariert!";
alert(test2==window.test2); // ergibt true
```

Desweiteren ebenfalls global sind in Funktionen implizit deklarierte Variablen (die dort also ohne var eingeführt werden):

```
function beispiel() {
    // implizit deklariert, daher global
    test3 = "Ich bin global!";
}


// Deklaration von test3 durch Aufruf
beispiel();

// ergibt true
alert(test3==window.test3);
```

In Funktionen explizit mit `var` deklarierte Variable gelten dagegen ausschließlich lokal innerhalb des Funktionsblocks. Sie sind außerhalb der Funktion also nicht verfügbar.

```
function beispiel2() {
    // explizit deklariert, daher lokal
    var test4 = "Ich bin lokal!";
```

```
}

// keine Deklaration von test4 durch Aufruf
beispiel2();

// test4 nicht existent; ergibt undefined:
alert(typeof test4);
```

Es ist allerdings durchaus möglich, außerhalb einer Funktion (d.h. scheinbar nach deren Laufzeit) auf eine ihrer lokalen Variablen zuzugreifen – allerdings kann dies nur über eine innere Funktion geschehen, mittels einer sogenannten „Closure“.

5.4. Funktionen – Scope-Chain, Closures

Über eine Closure kann eine innere Funktion auf Variablen oder übergebene Parameterwerte aus dem Scope ihrer übergeordneten Funktion zugreifen, selbst wenn diese mittlerweile beendet ist. Normalerweise werden alle lokalen Variablen einer Funktion nach deren Beenden gelöscht.

Im Falle einer Closure werden die Variablen der äußeren Funktion nach deren Ausführung nicht gelöscht, sondern verbleiben im Speicher.

```
function aussen() {
    // Definiere eine lokale Variable
    var lokVar = "Ich bin ein lokaler Wert";

    // Lege eine Funktion als lokale Variable an
    var innen = function () {

     // Variable des umgebenden Scopes ist verfügbar:
    alert("Variable der äußeren Funktion: " + lokVar);
    };

    // Führe die eben definierte Funktion aus
    innen();
}
// Funktion gibt lokalen Wert über innere Funktion aus:


aussen();
```

Die innere Funktion greift über eine **Closure** auf die Variable des umgebenden Scopes zu.

Erläuterung: Eine Closure kann als „Objekt“ betrachtet werden, das eine Funktion sowie deren Ausführungskontext beinhaltet. Wird eine Referenz auf ein Funktionsobjekt weitergegeben, behält die Funktion damit auch Zugriff auf die Variablen aus dem Kontext, in dem sie definiert wurde.

Nach Ablauf der äußeren Funktion behält die Closure den Zugriff auf deren Variablen – vorausgesetzt, sie wird gespeichert und kann dadurch später ausgeführt werden.

Eine Möglichkeit besteht darin, die Closure über Eventhandler an ein Element zu binden:

```
function aussen() {
    var lokVar = " Ich bin ein lokaler Wert";
    // Closure-Funktion in lokaler Variable ablegen:
    var meineClosure = function () {
    alert("Lokale Variable von aussen(): " + lokVar);
    };
    // Closure-Funktion an Event-Handler zuweisen:
    document.getElementById("meinId").onclick =
                                    meineClosure;
}

// Funktion ausführen
aussen();
```

5.5. Ausführungskontext – this

Funktionen werden stets in einem „Kontext“ ausgeführt, je nachdem, welchem Objekt sie „gehören“. Im Inneren der Funktion wird dieser Kontext mit dem Bezeichner `this` referenziert.

Außerhalb einer Funktion existiert `this` ebenfalls und zeigt auf das `window`-Objekt:

```
// das Kontextobjekt this
console.log(this); // -> window
```

Eine globale Funktion wird üblicherweise im globalen Kontext ausgeführt, also mit `window` als Kontext. Im Inneren der Funktion zeigt `this` daher ebenfalls auf `window`:

```
function test(){
   // das Kontextobjekt
   console.log(this); // -> window
}
```

Gehört eine Funktion (als Methode) zu einem **Objekt**, so wird das Objekt zu ihrem Ausführungskontext:

```
var meinObjekt = {
   meth : function(){
      console.log(this);  // -> meinObjekt
   }
};
```

Etwas eigenartig ist die Lage, wenn eine Objektmethode über eine **innere Funktion** verfügt. Für diese fällt `this` auf `window` zurück:

```
var meinObjekt = {
   a : "A",
   meth : function(){
      console.log(this);  // -> meinObjekt
      function innen(){
         console.log(this); // -> window
      }
      innen();
   }
};
```

Dies liegt daran, dass eine Funktion ihren `this`-Kontext stets **übergeben** bekommt. (Er wird also *nicht* aus dem Umfeld „geerbt“ – in diesem Fall könnte das innere `this` ebenfalls das Objekt sein.)

Das innere `this` **verdeckt** das auf das Objekt zeigende `this` daher unweigerlich. Ein Trick besteht darin, das „erwünschte“ `this` umzuspeichern und so (unter Alias) innen zugänglich zu machen.

```
var meinObjekt = {
   a : "A",
   meth : function(){
      console.log(this);  // -> meinObjekt
      // Kontext umspeichern:
      var that = this;
```

```
      function innen(){
         console.log(this); // -> window
         console.log(that.a); // Zugriff auf das Objekt
      }
      innen();
   }
};
```

5.6. Methoden von Funktionsobjekten – call() und apply()

Wenn wir uns damit abgefunden haben, dass eine Funktion ein Objekt darstellt, dann mag es weniger verblüffend sein, dass es auch Methoden von Funktionsobjekten gibt – wir betrachten zwei davon, nämlich `call()` und `apply()`.

Bevor wir damit herausrücken, wozu diese benötigt werden, betrachten wir noch einmal den **Kontext**, in dem eine Funktion gilt.

Wie oben ausgeführt, zeigt `this` meist auf `window`. Will man das *immer* so haben? Sicherlich nicht.

Erstellen wir drei Dinge – eine globale Variable, ein Objekt und eine Funktion im globalen Kontext:

```
var name = "das window-Objekt";

var person = { name: "Herr Müller" };


function hallo() {
      alert("Hi, ich bin " + this.name);
}
```

Setzen wir die Funktion einmal in Betrieb:

```
hallo();
// Alert: “Hi, ich bin das window-Objekt”
```

Das ist nicht verwunderlich, weil ja der Kontext während der Ausführung der Funktion (dies liegt ja im globalen Scope) auf dem `window`-Objekt liegt.

Wenn wir das aber nicht wollen? Dann nehmen wir `call()` zu Hilfe. Und das geht so:

```
hallo.call(person);
// Alert: „Hi, ich bin Herr Müller“
```

Die globale Funktion `hallo()` verhält sich mit Hilfe von `call()` so, als ob sie eine Methode des Objekts `person` wäre, die ihr als Parameter übergeben worden ist. Die „Funktionsmethode“ `call()` **verschiebt also den Kontext** hin zu dem ihr übergebenen Objekt.

Analog arbeitet `apply()`:

```
hallo.apply(person);
// Alert: „Hi, ich bin Herr Müller“
```

Der Unterschied zwischen `call()` und `apply()` kommt erst zum Tragen, wenn weitere Parameter übergeben werden. Während `call()` zusätzlich eine Liste aus Einzelargumenten akzeptiert, möchte `apply()` stattdessen ein Array, das diese weiteren Parameter enthält.

6. Objekte

Ein Objekt kann in JavaScript entweder über den allgemeinen Konstruktor `Object()` oder, wirkungsgleich, als Literal `{}` erzeugt werden. Letzteres hat sich im Rahmen der Frameworks wegen der kompakteren Schreibweise als Standard durchgesetzt.

6.1. Objekte – Literale, Konstruktor

Mit einem einzeiligen Statement erzugen wir ein neues, leeres Objekt und legen es in eine Variable ab. Die rechte Seite wirkt auf den ersten Blick wie ein Funktionsaufruf (was im Grunde gar nicht so verkehrt ist). Auffällig ist das vorangestellte Schlüsselwort `new`.

De facto ergibt ein Funktionsaufruf plus `new`-Schlüsselwort stets einen „Konstruktoraufruf“. Hierbei ist `Object` (achten Sie auf die Schreibweise) einer der vordefinierten Konstruktoren:

```
// ein neues leeres Objekt erstellen:
var meinContainer = new Object();
```

Wirkungsgleich zum Objektbau mit Konstruktor `Object()` ist das Schreiben eines **Objektliterals**. Mit anderen Worten, wir schreiben das Objekt einfach „wie es ist“ hin. Um so ein neues, leeres Objekt zu erzeugen, genügt dies:

```
var meinContainer = {}
```

6.2. Objekte – Properties und Methoden

Die Variable `meinContainer` ist keine „herkömmliche“ Variable, sondern eine Variable vom „Type Object“. Dadurch ist es uns gestattet, ihr Eigenschaften zu geben. Hierfür wird der Punktoperator eingesetzt. Die Namen der Eigenschaften können wir frei wählen, wie bei Variablen. Es handelt sich quasi auch um Variable, allerdings um solche, die diesem Objekt „gehören“:

```
// jetzt die Eigenschaften erzeugen:
meinContainer.meineEigenschaft = "wert";
```

Speichern wir in eine Objektvariable eine Funktion (eine anonyme Funktion tut hier gute Dienste), so nennen wir dies eine „Methode“ des Objekts.

```
meinContainer.meineMethode = function () {
   alert("Container-Eigenschaft: " +
          this.meineEigenschaft);
};
```

Diese Funktion ist anschließend nur im „Kontext“ des Objekts, also über die Objektnotation mit Punkt, aufrufbar:

```
meinContainer.meineMethode();
```

Das Hinzufügen weiterer Methoden (oder Eigenschaften) erfolgt analog. Hier wird dem Objekt eine zweite Methode `meinContainer.zweiteMethode` hinzugefügt, welche die erste Methode aufruft. Der Bezeichner `this` bezeichnet in der Methode ebenfalls das im Kontext befindliche Objekt, also meinContainer.

```
meinContainer.zweiteMethode = function () {
    this.meineMethode();
};
```

```
// Aufruf:
meinContainer.zweiteMethode();
```

Nachteil dieser Methode der Objektherstellung ist, dass wir anschließend genau *eine* Objektinstanz besitzen. Wollen wir ein zweites, gleichartiges Objekt erstellen, muss die ganze Prozedur wiederholt werden.

Auch durch ein Literal erzeugten Objekten können Eigenschaften und Methoden, wie eben, nachträglich hinzugefügt werden. Bei Literalen muss man dies jedoch nicht tun: Man kann dies gleich „vor Ort“ im Inneren der Literalklammern vornehmen. Achtung, zwischen den einzelnen Eigenschaftsdefinitionen steht ein **Komma**:

```
var meinContainer = {
   // Objektmembers werden im Inneren erzeugt:
   meineEigenschaft: "wert",
   meineMethode: function () {
      alert("Container-Eigenschaft: " +
       this. meineEigenschaft);
  }
}; // Ende des Literal-Statements
```

Der Aufruf der Objektmethode geht wie gehabt:

```
// Aufruf:
meinContainer.meineMethode();
```

Nachteil auch dieser Methode der Objektherstellung ist, dass wir anschließend nur genau *eine* Objektinstanz besitzen.

6.3. Objekte – Eigene Konstruktoren

Falls mehrere oder viele Instanzen eines Objekttyps benötigt werden, brauchen wir eine maßgeschneiderte Konstruktorfunktion.

Nehmen wir an, dass wir Objekte wie das vorhin vorgestellte „konkrete“ Objekt `meinContainer` erzeugen wollen (konkret bedeutet, dass die abstrakten Eigenschaften einen konkret für dieses eine Objekt festgelegt sind – man nennt dies eine „Instanz“ dieses Objekttyps). Hier der äquivalente Literalausdruck:

```
var meinContainer = {
  // Objektmembers werden im Inneren erzeugt:
  meineEigenschaft: "wert",
  meineMethode: function () {
  alert("Container-Eigenschaft: " +
         this. meineEigenschaft);
  }
}; // Ende des Literal-Statements
```

Wir brauchen nun eine Möglichkeit, analoge Objektinstanzen zu erzeugen. Von der oben gezeigten sollen sie sich nicht in der Struktur unterscheiden, sollen also über gleiche (und gleich benannte) Eigenschaften und Methoden verfügen. Nur die konkreten Werte der Eigenschaften und natürlich der Name der Objektinstanz sollen anders sein.

Der Gedanke liegt nahe, eine Funktion mit der Erstellung eines Objektes zu betrauen und dieser Funktion die benötigten konkreten Werte als Parameter zu übergeben. Im Inneren der Funktion wird der übergebene Wert der Eigenschaft zugeordnet.

Sie haben bemerkt, dass wir den Funktionsnamen des Konstruktors mit einem Großbuchstaben begonnen haben. Dies ist zwar nicht unbedingt nötig, aber es handelt sich um eine Konvention, die sich zur Bezeichnung von Klassen (und eben Konstruktoren) eingebürtert hat.

Kommen wir zum Hattrick der Angelegenheit – dem Aufruf des Konstruktors. Dies darf in der Tat nicht wie folgt geschehen (à la „normale“ Funktion):

```
MachContainer("neuer Wert");
```

Warum? Weil `this` im Inneren der Funktion in diesem Fall leider auf das globale Objekt zeigt. Was dann passiert, ist, dass wir im globalen Kontext (d. h. im `window`-Objekt) eine Variable namens `meineEigenschaft` erzeugen. Schlecht. Ein neues Objekt käme ohnehin nicht dabei heraus, weil die Funktion keinen definierten Rückgabewert hat:

```
// die Variable bleibt leer:
var leiderKeinObjekt = MachContainer("neuer Wert");
```

Aus diesem Grund muss der Aufruf eines Konstruktors mit vorangestelltem Schlüsselwort new erfolgen. Was bewirkt `new`? Ganz einfach – dies ist nun

die Anordnung, zunächst ein neues, leeres Objekt zu erzeugen *und* es als Kontextobjekt zu verwenden. Bingo:

```
var neuerContainer = new MachContainer("neuer Wert");
```

Fügen wir nun eine Methode hinzu. Hierfür gibt es zwei Möglichkeiten. Erstens könnten wir den Konstruktor erweitern. Kein Problem, solange die Funktion, die der Methode hinterlegt ist, knapp genug ist. Ansonsten „sprengen“ wir die Übersichtlichkeit unseres Konstruktors möglicherweise:

```
function MachContainer(wert) {
  this.meineEigenschaft = wert;
  this.meineMethode = function () {
    alert("Container-Eigenschaft: "+
           this. meineEigenschaft);
  }
}
var neuerContainer = new MachContainer("neuer Wert");
neuerContainer.meineMethode();
  // -> Alert: „neuer Wert“
```

Okay, das geht also. Beachten Sie, dass das `this` im Inneren der Methode tatsächlich nicht auf die Methode (als Funktionsobjekt), sondern auf das erstellte Objekt Bezug nimmt. So soll es auch sein.

Nehmen wir (als Hilfsvorstellung) an, dass wieder als erster Schritt ein leeres Objekt erstellt wird (obwohl dies in der Funktion nicht explizit angeordnet wird). Dieses Objekt ist dann automatisch Kontextobjekt der Operation. Wir können uns also mit `this` darauf beziehen, was wir verwenden, um Wert und Eigenschaft (die ja dem Objekt gehören soll, nicht der Funktion) zusammenzubringen. Dies ergibt etwa Folgendes (die Methode lassen wir vorerst weg):

```
function MachContainer(wert) {
  // übergebener Wert wert landet in meineEigenschaft:
  this.meineEigenschaft = wert;
}
```

6.3.1. Objekte – Erweiterung von Objekten

Gehen wir von einem Objektliteral und einer Funktion aus, die ein entsprechendes Objekt verarbeiten soll:

```
var r1 = { h: 100, b:200 };

function flaeche(rechteck) {
  return rechteck.b * rechteck.h;
}

console.log("Fläche r1: "+ flaeche(r1));
```

Dies geht, setzt aber voraus, dass der Funktion ein passendes Objekt beim Aufruf übergeben wird. Besser wäre es, wenn Rechteck-Objekte eine Methode zum Lesen ihrer Fläche besäßen.

Ein Konstruktor, der solche Objekte erzeugt, sähe so aus:

```
function Rechteck(h, b) {
   this.h = h;
   this.b = b;
   this.flaeche = function() {
      return this.h * this.b;
   }
}


var r2 = new Rechteck(200, 300);

console.log("Fläche r2 :" + r2.flaeche());
```

Nachteil ist, dass sowohl die Eigenschaften, als auch die Methode in jeder Instanz gespeichert sein müssen. Bei den Eigenschaften ist dies prinzipiell (meist) unumgänglich, sofern sie die Instanz individualisieren.

Ein weiterer Nachteil ist, dass beim Aufruf des Konstruktors die Eigenschaftswerte auch korrekt übergeben werden müssen. Ansosten sind eigenschaften undefiniert und die Methode kann nicht arbeiten. Hier kann man einfach abhelfen:

```
// Konstruktur mit Parametern (neu: Defaultwerte):
function Rechteck(a, b) {
   // sind a oder b "falsy", gilt der Defaultwert:
   this.a = a || 10;
   this.b = b || 20;
```

```
   this.flaeche = function() {
        return this.a * this.b;
   };
}
```

Um nicht an eine Reihenfolge bei der Parameterübergabe gebunden zu sein, kann auch ein Objekt `conf` (aka „Konfigurationsobjekt“) übergeben werden. Hier wird eine zweite Methode zur Berechnung des Umfangs hinzugefügt:

```
// Parameter werden in Objektform hineingereicht
function Rechteck(conf) {
    // ein nicht existentes Property in conf ist falsy:
    this.a = conf.a || 10;
    this.b = conf.b || 20;
    this.umfang = function() {
        return 2*this.a + 2*this.b;
    };
    this.flaeche = function() {
        return this.a * this.b;
    };
}

var rConf1 = new Rechteck( {} );
var rConf2 = new Rechteck( {b:17} );
var rConf3 = new Rechteck( {a:111} );
var rConf4 = new Rechteck( {b:7, a:36} );
```

Solange ein Objekt hineingegeben wird, ist der Konstruktor „zufrieden“. Die Reihenfolge der Eigenschaften im Objekt ist egal. Es können problemlos „falsche“ Eigenschaften übergeben werden:

```
var rConf5 = new Rechteck({hallo:"Welt"});
```

Methoden sind normalerweise über alle Instanzen hin gleich. Es gibt also gute Gründe, sie „auszulagern“. Ein erster Versuch könnte über das Referenzieren globaler Funktionen unternommen werden:

```
// Parameter werden in Objektform hineingereicht
function Rechteck(conf) {
     // nicht existentes Property in conf ist falsy:
   this.a = conf.a || 10;
   this.b = conf.b || 20;
   this.umfang = umfang;
   this.flaeche = flaeche;
}
```

```
// Methoden in globale Funktionen "ausgelagert"
// schlecht, weil der Aufrufkontext nicht feststeht
// gut, weil Methoden so nicht in der Instanz stehen
function umfang() {
      return 2*this.a + 2*this.b;
};

function flaeche() {
      return this.a * this.b;
};
```

Die nächste Idee bestünde darin, die Konstruktorfunktion zu erweitern:

```
// Jede Funktion ist ein Objekt. Diese hier auch.
function Rechteck(conf) {
   // ein nicht existentes Property in conf ist falsy:
   this.a = conf.a || 10;
   this.b = conf.b || 20;
}

// jedes Objekt lässt sich erweitern:
Rechteck.umfang = function() {
   return 2*this.a + 2*this.b;
};

Rechteck.flaeche = function() {
   return this.a * this.b;
};

var r1 = new Rechteck({});

// leider ist die Methode
// bei der Instanz nicht sichtbar (Fehler!):
console.log("r1.umfang(): ",r1.umfang());

// So geht's aufzurufen, macht aber keinen Sinn:
console.log("Rechteck.umfang(): ", Rechteck.umfang());
```

Es gibt jedoch eine „amtliche“ Möglichkeit, die Methoden an den Konstruktor zu binden, sodass er den Instanzen wirlich zur Verfügung steht. Hierzu dient das `prototype`-Objekt.

6.4. Objekte – Prototype-Objekt und Prototype-Chain

Wenn die Methoden umfangreicher sind, oder nachträglich hinzugefügt werden müssen, bietet sich ein anderer Weg (den viele Programmierer ohnehin bevorzugen – schon aus Prinzip sollten wir uns das mal ansehen). Dieser Weg wird als „Prototyping“ bezeichnet.

Erinnern Sie sich, dass am Anfang gesagt wurde, JavaScript sei eine „prototypisierte“ objektorientierte Sprache? Es handelt sich hier um ein grundlegendes Konzept der Sprache – die Erweiterbarkeit von Objekten und Objekttypen.

Gehen wir nochmals vom einfachen Konstruktor aus, den wir durch „Prototyping“ erweitern werden. Der Konstruktor sah so aus:

```
// Jede Funktion ist ein Objekt. Diese hier auch.
function Rechteck(conf) {
   // ein nicht existentes Property in conf ist falsy:
   this.a = conf.a || 10;
   this.b = conf.b || 20;
}
```

Jedes Objekt lässt sich erweitern. Lassen wir langwierige Erklärungen beiseite, erweitern wir es einfach:

```
// Die Lage ist: Rechteck.prototype = {};
Rechteck.prototype.umfang = function() {
   return 2*this.a + 2*this.b;
};

Rechteck.prototype.flaeche = function() {
   return this.a * this.b;
}
```

Der Aufruf des Konstruktors und anschließender Ruf der Methode klappen erneut:

```
var r1 = new Rechteck({});

//  die Methode ist bei der Instanz sichtbar:
console.log("r1.umfang(): ",r1.umfang());
console.log("umfang an r1? ",
            r1.hasOwnProperty('umfang'));
```

Warum ist dies nun besser? Und wie funktioniert es überhaupt? Besser ist es, weil wir nachträglich – und zwar jederzeit! – die Konstruktorfunktion um

Methoden erweitern können. Ob mit diesem Konstruktor bereits Objektinstanzen erzeugt wurden, ist dabei egal – die Erweiterung bezieht sich automatisch auch auf bereits existierende Objekte dieses Typs.

Schwieriger ist die Erklärung, **wie** es funktioniert. Nehmen wir an, jedes Objekt „erinnert“ sich daran, wer (oder was) es konstruiert hat. Nehmen wir weiter an, dass das Objekt sich „an der Quelle“ informiert, ob sich an den Konstruktionsvorschriften etwas geändert hat. Beides stimmt soweit auch – jedes Objekt besitzt eine Eigenschaft namens `constructor`, in dem eine Referenz auf „seine“ Konstruktorfunktion gespeichert ist.

Wenn wir den Namen des Kostruktors erfahren wollen, schreiben wir:

```
alert(r4.constructor.name);
```

Ähnlich mit der „Änderungsmeldung“ zwischen Instanz und Konstruktor. Dies geschieht über die sogenannte **Prototype-Chain**. Oben wurde eine Eigenschaft (eigentlich ein Unterobjekt) namens `prototype` eingesetzt, um an den Konstruktor eine Methode zu binden.

Diese Eigenschaft ist bereits im Stammobjekttyp `Object` enthalten. Weil sich von diesem alles ableitet, existiert `prototype` auch in jedem anderen Objekt, also auch in `Rechteck2()`.

Nun besitzt jedes Objekt, analog zur `constructor`-Eigenschaft, auch eine Referenz auf die `prototype`-Eigenschaft seines Konstruktors. Da sie eigentlich nicht dem Objekt gehört, sondern dem Konstruktor, erreichen wir sie von Objekt aus nur indirekt:

```
// die prototype-Eigenschaft ist ein Objekt
alert(typeof r4.constructor.prototype);
```

Gehen wir einen Schritt weiter, indem wir den Konstruktor vereinfachen:

```
function Rechteck3() { };
```

Schreiben wir anhand des neuen Konstruktors das Prototyping einmal um, damit der **Objektcharakter** der Eigenschaft `prototype` etwas deutlicher wird:

```
Rechteck3.prototype = {
    h = 100,
    b = 150,
    flaeche = function() {
```

```
        return this.h * this.b;
    },
    hallo = function() {
        alert('Ich bin ein Rechteck.')
    }
}
```

Das Manöver geschieht also in der `prototype`-Eigenschaft des Konstruktors, den dieser von `Object` geerbt hat. Was passiert dabei nun in der „betroffenen“ Objektinstanz? Die Antwort ist einfach: gar nichts.

Das ist auch nicht nötig, da zwischen der Objektinstanz und der `prototype`-Eigenschaft des Konstruktors sozusagen eine „stehende Verbindung“ vorliegt. Wird eine Objektinstanz um die Ausführung einer Methode „gebeten“, so nimmt sie über diese `prototype`-Verbindung Kontakt mit dem Konstruktor auf und ruft die benötigte Methode von dort ab. Man sagt, die Instanz „erbt“ die Methoden über die Prototypkette.

6.4.1. Objekte – Lokale und nicht-lokale Properties

Mit dem vorhin erstellten Konstruktor `Rechteck3` lassen sich nun Instanzen des Typs „Rechteck3“ erstellen:

```
var r5 = new Rechteck3();   // neue Instanz
r5.b = 300;  // ...und Eigenschaft b überschreiben

var r6 = new Rechteck3();
```

Beide Instanzen besitzen eine Fläche:

```
console.log("Fläche r5 :" + r5.flaeche());

console.log("Fläche r6 :" + r6.flaeche());
```

Hierbei werden b und h für `r6` aus den Prototype geholt. Sie sind also nicht *lokal* in der Instanz vorhanden. Die Eigenschaft b von `r5` hingegen ist tatsächlich am Objekt gespeichert.

6.4.2. Objekte – hasOwnProperty

Über die Methode `hasOwnProperty` lässt sich feststellen, ob eine Eigenschaft “lokal” an einem Objekt vorliegt, oder nicht (d.h. nur über die Prototype-Chain errreicht wird).

```
// Nur zur Demonstration: Object.prototype ändern:
Object.prototype.test = 1; // Object erweitern

var beispiel = {
      bla: undefined
};

beispiel.bla; // undefined (explizit!)
beispiel.blu; // undefined (implizit)
beispiel.test; // 1

// in Operator prüft nur, ob Property existiert:
'bla'  in  beispiel; // true (existiert)
'blu'  in  beispiel; // false (existiert nicht)
'test' in  beispiel; // true (existiert)

// hasOwnProperty prüft, ob Property lokal vorliegt:
beispiel.hasOwnProperty('test'); // false
beispiel.hasOwnProperty('bla'); // true
```

Anmerkung: Es lässt sich nicht einfach unterscheiden, ob ein Property nicht existiert (`beispiel.blu`), oder existiert und den Wert *undefined* besitzt (`beispiel.bla`) – es sei denn miitels der Methode `hasOwnProperty` (sofern die Eigenschaft lokal ist).

Die Rechtecke `r5` und `r6` aus dem vorigen Abschnitt können ebenfalls so untersucht werden:

```
console.log("r5 hat Breite: " +
r5.hasOwnProperty("b")); // true


console.log("r6 hat Breite: " +
r6.hasOwnProperty("b")); // false
```

6.4.3. Objekte – for-in-Schleife

Der in Operator gibt *true* zurück, sofern ein Property am Objekt oder in dessen Prototype-Chain existiert (siehe vorieger Abschnitt). Analog zur `for`-Schleife existiert eine `for-in`-Schleife, die diesen Operator ebenfalls einsetzt:

```
Object.prototype.test = 1; // Object erweitern

var beispiel = {
      pop: "pling"
};

for(var i in beispiel) {
    console.log(i); // gibt test und pop aus
}
```

Möchte man nur die lokalen Eigenschaften durchlaufen, so verwendet man zusätzlich `hasOwnProperty` als Filter:

```
// selbes Objekt wie eben:
for(var i in beispiel) {
    if (beispiel.hasOwnProperty(i)) {
        console.log(i); // gibt pop aus (nicht: test)
    }
}
```

6.4.4. Objekte – delete-Operator

Lokale Properties und Methoden an einem Objekt können mittels des `delete`-Operators gelöscht werden. Dies wirkt sich nicht auf Properties aus, die über die Prototype-Chain addressiert werden.

Globale Werte (Variablen und Funktionen) können hingegen nicht gelöscht werden, da sie über ein internes `DontDelete`-Flag verfügen, das auf *true* steht. Für eine globale Variable geschieht folgendes:

```
// globale Variable:

var a = 1; // DontDelete gesetzt

delete a; // false (d.h. es passiert nichts)

a; // 1
```

Analog für eine globale Funktion:

```
// normale Funktion:

function f() {} // DontDelete gesetzt

delete f; // false (d.h. es passiert nichts)

typeof f; // "function"
```

An einem Objekt können lokale Properties gelöscht werden:

```
// explizit esetztes Property:

var obj = {x: 1}; // bei Erstellung

obj.y = 2;        // nachträglich explizit

delete obj.x; // true
obj.x; // undefined

delete obj.y; // true
obj.y; // undefined
```

Auch hier ist das Rechteck `r5` wieder ein praktisches Anschauungsbeispiel:

```
var r5 = new Rechteck3();  // h ist 100, b ist 150

r5.b = 300;   // b lokal überschrieben

console.log("Fläche r5 :" + r5.flaeche());   // 30,000

// true:
console.log("r5 hat Breite: " +r5.hasOwnProperty("b"))

delete r5.b;

// false:
console.log("r5 hat Breite: " +r5.hasOwnProperty("b"))

console.log("Fläche r5 :" + r5.flaeche());   // 15,000
```

Mit `delete` kann man die lokale Eigenschaft löschen. Hierbei wird deutlich, dass diese eine Prototype-Eigenschaft „verdeckt“ hatte, die nun im Sinne eines Defaultwertes, wieder sichtbar wird.

6.5. Objekte – Private Properties

Normalerweise sind alle Eigenschaften eines Objekts „öffentlich“. Es ist jedoch möglich, Eigenschaften zu verstecken. Diese müssen einfach im Konstruktor als lokale Variable angelegt werden.

```
// Jede Funktion ist ein Objekt. Diese hier auch.
function Quader(name) {
   // "Private" Eigenschaft
   var name = name;
   // Getter-Methode über Closure:
   this.hallo = function() {
      console.log("Hallo, ich bin ", name);
   }
}

// diese Eigenschaften sind "public"
Quader.prototype.a = 10;
Quader.prototype.b = 10;
Quader.prototype.c = 10;

Quader.prototype.volumen = function() {
   return  this.a * this.b * this.c;
};

var q1 = new Quader("Block");
var q2 = new Quader("Klotz");

//  Volumen:
q1.a = 13;

console.log("Volumen von q1: ", q1.volumen()); // 1000
console.log("Gibt es a in q1? ",
q1.hasOwnProperty('a'));

// Enumeration beschränkt auf Instanzeigenschaften:
for(var i in q1) {
      // „name“ ist nicht zu sehen
      console.log(i,": ",q1[i]);
}

q1.hallo(); // Block
q2.hallo(); // Klotz
```

Ein Auslesen der privaten Eigenschaften ist nur über Methoden möglich, die im Konstruktor definiert sind:

```
// Getter-Methode extern KANN NICHT funktionieren:
Quader.prototype.hallo = function() {
      // wir greifen auf den globalen Scope zu:
      console.log("Hallo, ich bin ", name);
};
```

6.5.1. Objekte – Private und Priviligierte Methoden

Methoden, die auf private Eigenschaften zugreifen, werden als „priviligierte“ Methoden bezeichnet. Solche, die ebenfalls von außen nicht zugänglich sind, werden „private“ Methoden genannt. Sie können beispielsweise im Rahmen von Objektmethoden aufgerufen werden (die dann wiederum „privilegiert“ sind). Sowohl private Eigenschaften und Methoden als auch Methoden, die auf diese zugreifen, müssen im Rahmen des Konstruktors definiert werden.

```
function Quader(name) {
   // "Private" Eigenschaft
   var name = name;

   // "Private" Methode:
   function geldZaehlen(input) {
      // zählt Geld (insgeheim)
      return 100000 + input;
   };

   // "Privilegierte" Methode
   this.geldausgeben = function(geld) {
      console.log("Das kostet jetzt ",
                  geldZaehlen(geld));
   };

   // Getter-Methode über Closure:
   this.hallo = function() {
      console.log("Hallo, ich bin ", name);
   };

   // Setter-Methode über Closure
   this.taufen = function(input) {
      name = input ||  "Thomas Anders";
   }
}
```

6.6. Objekte – Vererbung und Simulation von Klassen

Manchmal möchte man von einem Objekt ein anderes ableiten. Dies kann zunächst einfach eine **Kopie** sein. Dies ist jedoch gar nicht so einfach.

```
var basis = {
   a: "Wert 1",
   b: "Wert 2"
};

console.log(basis.a);

// Referenz, keine Kopie!
var keineKopie = basis;
```

Um ein Objekt *wirklich* zu kopieren, sind ein paar Ausweichmanöver nötig:

```
// Objekt kopieren:
var kopie = {};

for(var i in basis) {
   kopie[i] = basis[i];
}

console.log(kopie.a);
```

Um eine **Vererbung** zu ermöglichen, also alle Eigenschaften des Basisobjekts neuen Instanzen zur Verfügung zu stellen, kann man die Prototype-Chain einsetzen. Betrachten Sie dies hier für einen Augenblick:

```
function vererben(objInput) {
   function F() {   }
   F.prototype = objInput;
   return new F();
}

var obj = vererben(basis);
console.log(typeof obj);
console.log(obj.a);
console.log(obj.c);
```

Es können auch den Instanzen jeweils zusätzliche Eigenschaften hinzugegeben werden:

```
function vererben(objInput) {
   function F() {
```

```
      // Instanzeigenschaften
      this.c = "Wert C";
   }
   F.prototype = objInput;
   return new F();
}

var obj = vererben(basis);
console.log(typeof obj);
console.log(obj.a);
console.log(obj.c);
console.log(basis.c);

basis.a = "Neuer Wert";
console.log(obj.a);
```

Mit etwas mehr Aufwand ist ein **Verschmelzen** zweier Objekte darstellbar:

```
// stellt Instanzeigenschaften zur Verfügung:
var basis1 = {
   a: "Wert 1",
   b: "Wert 2",
   c: "Wert 3"
};

// dient als Prototype:
var basis2 = {
   c: "Wert C",
   d: "Wert D",
   e: "Wert E"
};

// diese Funktion verschmilzt zwei Objekte:
function merge(obj1, obj2) {
   function F() {
         for(var i in obj1) {
            this[i] = obj1[i];
         }
   }
   F.prototype = obj2;
   // ein neues Objekt mit Eigenschaften beider
Eingabeobjekte:
   return new F();
}

var merged = merge(basis1, basis2)
```

```
for (i in merged) {
   console.log(i, ": ",merged[i]);
}
```

Anstatt direkt Objektinstanzen zu erzeugen, kann die Vererbungsfunktion auch einen **Konstruktor** zurückgeben. Hierfür muss lediglich der new-Operator entfallen:

```
var basis = {
   a: "Wert 1",
   b: "Wert 2"
};

function vererben(objInput) {
   function F(input) {
      // Instanzeigenschaften
      this.c = input;
   }
   F.prototype = objInput;
   return F;
}

// Konstruktor erzeugen
var Test = vererben(basis);
var newObj = new Test(5);

console.log(newObj.c); // 5
console.log(newObj.b); // Wert 2
```

6.7. Verbessertes Objekt-Handling in ECMA-Script 5

Prinzipiell sind Objekte in JavaScript stets veränderlich („mutable“) und können zur Laufzeit modifiziert werden. Hierbei sind die Eigenschaften (Properties und Methoden) von außen in der Regel sichtbar („enumerable“), mithin öffentlich („public“).

6.7.1. Grundsätzliche Probleme mit JavaScript-Objekten

Eigenschaften können mit einiger Mühe immerhin versteckt werden (über Closures), quasi „private“ Eigenschaften können also simuliert werden. Ähnliches ist für Methoden möglich.

Nachteile der Arbeit mit **Closures**:

- Private Properties müssen in der **Instanz** definiert sein (im Rahmen des Konstruktors als *lokale Variablen*)
- Getter und Setter müssen ebenfalls in der Instanz existieren (als "privilegierte" Methoden; wiederum im Rahmen des Konstruktors)
- Prototype-Methoden können NIEMALS auf private Properties zugreifen (ebenfalls nicht auf private Methoden)

Fassen wir zusammen, was generell an JS-Objekten unangenehm auffällt:

- Private Properties umständlich zu definieren
- Werte öffentlicher Eigenschaften **überschreibbar**
- Öffentliche Eigenschaften können **gelöscht** werden
- Alle Eigenschaften "enumerable" (also sichtbar)
- Objektstruktur **veränderlich** (was aber Grundparadigma ist)

6.7.2. Objekteigenschaften in ECMA5

Diese Probleme werden in ECMA-Script 5.0 gelöst.

Derzeit sind *diese Aspekte* im Rahmen von JavaScript 1.8.5 spezifiziert (seitens der Mozilla Foundation). Implementiert in:
Firefox ab 4.0, IE ab 9, Safari ab 5.1, Opera ab 12, Chrome ab 7, Rhino 1.7

✔ Eine **Übersicht über Implementierungen** finden Sie hier:
http://kangax.github.com/es5-compat-table/

ES5 führt **neue Methoden** für das *Object*-Objekt ein.

6.7.3. Die Methode Object.defineProperty()

Hierunter ist die Methode `Object.defineProperty()`, die einem übergebenen Objekt ein Property zuweist und dabei Eigenschaften des Properties beschreibt. Der Bezeichner des Properties sei `name`:

```
var paul = {};

Object.defineProperty(paul, 'name', { value: 'paul' });
```

In der Tat besitzt besitzt das Objekt nun eine Eigenschaft `name`. Diese verhält sich jedoch "ungewohnt":

```
console.log(paul.name);   // lesen:         -> Paul
paul.name = "Klaus";      // überschreiben: geht nicht
console.log(paul.name);   // unverändert:   -> Paul
delete paul.name;         // löschen: geht nicht
console.log(paul.name);   // unverändert:   -> Paul
```

Warum ist das so? Beim Binden der Eigenschaft sind neben der explizit genannten **Descriptor-Komponente** `value` noch *drei weitere* mit ihren Defaultwerten beteiligt:

```
// Property name an paul binden:

Object.defineProperty( paul, 'name', {
      value: 'Paul',
      writable: false,       // Defaultwert
      configurable: false,   // Defaultwert
      enumerable: false      // Defaultwert
} );
```

6.7.4. Descriptoren für ES5-Objekt-Properties

Man unterscheidet, je nach der Art und Weise der Beschreibung des Properties *zwei Arten* von Descriptoren:

- **Data-Descriptor**
  Beschreibt *direkt* Wert und dessen Eigenschaften
- **Accessor-Descriptor**
  Beschreibt *Funktionen*, die den Wert des Properties lesen oder schreiben (Getter und Setter)

Im Rahmen des **Data-Descriptors** sind verfügbar:

- **value:** "Wert der Eigenschaft"
  Direkte Übergabe des Eigenschaftswertes

- **writable:** true | false (Default)
  Soll der Wert überschrieben werden können?

- **enumerable:** true | false (Default)
  Soll die Eigenschaft bei for-in erscheinen?

- **configurable:** true | false (Default)
  Soll die Description veränderlich sein?

Im Rahmen des **Accessor-Descriptors** sind verfügbar:

- **get:** function
  Wird beim Lesen gestartet

- **set:** function
  Wird beim Schreiben gestartet; Wert als Argument

- **enumerable:** true | false (Default)
  Soll die Eigenschaft bei for-in erscheinen?

- **configurable:** true | false (Default)
  Soll die Description veränderlich sein?

✔ Beide Accessor-Typen besitzen gleichermaßen die Meta-Properties `enumerable` und `configurable` und unterscheiden sich nur in der Art, wie der Wert in die Eigenschaft gelangt.

Beispiel für eine **Accessor-Description** (nur `get` und `set` explizit gesetzt):

```
var zahl = {};

Object.defineProperty( zahl, 'wert', {
      get: function() {
            console.log('Hole Wert');
            return wert;
      },

      set: function(x) {
            console.log('Setze Wert: ', x);
            wert = x;}
} );
```

Verwendung des Properties startet nun die Funktionen:

```
zahl.wert = 7;          // startet Setter
```

```
console.log(zahl.wert); // startet Getter
```

✔ **Achtung:** `value` oder `writable` können im Rahmen einer Accessor-Description nicht verwendet werden.

6.7.5. Die Meta-Eigenschaften writable und configurable

Mit `writable` und `configurable` kann eine Eigenschaft "festgesetzt" werden:

```
Object.defineProperty( paul, 'name', {
   value: 'Paul',
   writable: false,     // nicht schreibbar
   configurable: false, // nicht veränderlich/löschbar
   enumerable: true     // in for-in sichtbar
});
```

Hat configurable den Wert *true*, so kann der **Descriptor** nachträglich geändert werden:

```
Object.defineProperty( paul, 'name', {
   value: 'Paul',
   writable: false,     // nicht schreibbar
   configurable: true  // veränderlich(!)
});

// Neuzuweisung des Descriptors möglich:
Object.defineProperty( paul, 'name', {
   writable: true            // schreibbar(!)
});
```

6.7.6. Die Methode Object.defineProperties()

Um *mehrere* Properties *gleichzeitig* zu definieren, existiert die Methode `Object.defineProperties()`:

```
var myObj = {};

// Property-Liste wird als Objekt übergeben:

Object.defineProperties(myObj, {
```

```
   "value": {   value: true,    writable: false },
   "name": {  value: "Paul", writable: false }
});
```

6.7.7. Auslesen von Properties in ES5-Objekten

Um Properties auszulesen, existieren drei Verfahren:

- **for (i in obj)**
  erfasst alle "enumerable" Properties
  (inklusive jenen der Prototype-Chain)

- **Object.keys(obj)**
  erfasst nur lokale "enumerable" Properties
  (ohne Prototype-Chain)

- **Object.getOwnPropertyNames(obj)**
  erfasst alle lokalen Properties
  (enumerable und nicht-enumerable)

6.7.8. Mutability von ES5-Objekten

Wie eben gezeigt wurde, können einzelne Eigenschaften in ES5 als nicht schreibbar oder nicht löschbar definiert werden. Dies lässt sich auch auf Objekte als Ganzes ausdehnen.

Hierfür gibt es drei Möglichkeiten, die ebenfalls auf (neuen) Methoden des Object-Objekts beruhen:

```
Object.preventExtensions(obj);

Object.seal(obj);

Object.freeze(obj);
```

✔ **Achtung:**
Alle drei Methoden haben gemeinsam, dass sie nicht rückgängig gemacht werden können (es gibt kein *unfreeze*, *unseal* etc.)!

6.7.9. Die Methode Object.preventExtensions()

Die Methode `Object.preventExtensions()` verhindert, dass einem Objekt **neue Eigenschaften** zugewiesen werden können.

✔ **Vorhandene** Eigenschaften bleiben *zugänglich* (änderbar, löschbar).

```
var obj = {};
obj.p1 = "P1";   // Property zuweisen (geht)

Object.preventExtensions(obj);

obj.p2 = "P2";    // Property zuweisen schlägt fehl
console.log(obj.p1);  // P1
console.log(obj.p2);  // undefined (fehlgeschlagen!)
```

6.7.10. Die Methode Object.seal()

Die Methode `Object.seal()` verhindert, dass einem Objekt **neue Eigenschaften** zugewiesen werden können und dass vorhandene Eigenschaften **gelöscht** werden können.

✔ **Werte** existierender Properties bleiben schreibbar.

```
var obj = {};

obj.p1 = "P1";       // Property zuweisen

Object.seal(obj);

delete obj.p1;        // Löschen schlägt fehl
obj.p1 = "Neu!"       // Wert ändern erfolgreich
console.log(obj.p1);  // -> Neu!
```

6.7.11. Die Methode Object.freeze()

Die Methode `Object.freeze()` verhindert, dass einem Objekt neue Eigenschaften zugewiesen werden können und dass vorhandene Eigenschaften gelöscht oder ihre Werte verändert werden können.

✔ Das Objekt kann in keiner Form mehr verändert werden!

```
var obj = {};

obj.p1 = "P1";          // Property zuweisen

Object.freeze(obj);

delete obj.p1;          // Löschen schlägt fehl
obj.p1 = "Neu!"         // Wert ändern schlägt fehl
console.log(obj.p1);    // P1
```

6.7.12. Mutability-Zustand eines Objekts

Wie bereits erwähnt, ist die Anwendung dieser drei Methoden nicht umkehrbar. Der **Zustand** eines Objekts kann jedoch *ausgelesen* werden:

```
Object.isExtensible(obj);     // true | false

Object.isSealed(obj);         // true | false

Object.isFrozen(obj);         // true | false
```

7. Strict mode in ECMA5

Der sogenannte **Strict Mode** ist ein neues Feature in **ECMAScript 5** das gestattet, ein Programm (also einen *Scriptblock*) oder eine einzelne *Funktion* in einen “strikt” interpretierten Ausführungskontext zu bringen.

✔ Im strict mode sind bestimmte Operationen oder übliche Nachlässigkeiten („unsafe actions“) **verboten** (bzw. unterbunden) und es werden **Fehler** geworfen, die normalerweise vom Parser stillschweigend unterdrückt werden (*silent mode*). Desweiteren werden einige architektonische Probleme von Javascript umgangen.

Die Aktivierung des striuct mode erfolgt durch ein einfaches Statement („pragma“) in Stringform:

```
"use strict";
```

Der gesamte **folgende** Code unterliegt dem *strict mode*.

✔ Steht dieses Statement zu Beginn eines Scriptblocks, so wird der gesamte Block „strikt“ interpretiert.

Dies ist ebenfalls für einen Funktionskontext **separat** möglich:

```
function binStrict(){
      "use strict";
      // ...alles hier “strikt”...
}
```

7.1. Strict mode und Variablen

Im *strict mode* ist es nicht gestattet, eine Variable „implizit“ als global zu deklarieren, indem man sie ohne `var`-Statement einführt.

Folgendes führt zu einem Fehler:

```
"use strict";

// foo wurde nicht deklariert:
foo = "bar"; // -> Fehler
```

Üblicherweise wird der Versuch, ein globales Symbol zu **löschen** nicht geahndet, es geht einfach nicht („silent fail“). Nicht so im *strict mode*:

```
var foo = "bla";

function test(){}

delete foo; // -> Fehler

delete test; // -> Fehler
```

Dies gilt auch für **lokale Variablen** innerhalb von Funktionen:

```
function beispiel(arg) {
   delete arg; // -> Fehler
}
```

7.2. Strict mode und this

Erinnern wir uns an die Objektmethode, die über eine **innere Funktion** verfügt. Üblicherweise fällt für diese `this` auf `window` zurück:

```
var meinObjekt = {
   meth : function(){
      console.log(this);  // -> meinObjekt
      function innen(){
         console.log(this); // -> window
      }
      innen();
   }
};
```

Dies führt oft zu Verwirrung (und könnte Schaden im globalen Scope anrichten). Im *strict mode* wird der Ausführungskontext für solcherart aufgerufene Funktionen daher auf `undefined` gesetzt

```
var meinObjekt = {
   meth : function(){
      "use strict";
      console.log(this);  // -> meinObjekt
      function innen(){
         console.log(this); // -> undefined
      }
      innen();
   }
};
```

7.3. Strict mode und Objekte

Üblicherweise ist es JavaScript „egal“, ob ein Property innerhalb eines Objektliterals mehrfach auftritt. Nicht so im *strict mode*:

```
// -> Fehler
var meinObject = { foo: true, foo: false };
```

Operationen auf *sealed* oder *frozen* Objects, die im normalen Modus stillschweigend scheitern, werfen im *strict mode* einen Fehler:

```
var obj1 = { p1 : "Wert" };
Object.seal(obj1);
delete obj1.p1;        // Löschen: -> Fehler

var obj2 = { p1 : "Wert" };
Object.freeze(obj2);
delete obj2.p1;         // Löschen: -> Fehler
obj2.p1 = "Neu!"        // Wert ändern: -> Fehler
```

7.4. Strict Mode und eval()

Als dynamische Scriptsprache weist JavaScript einige prinzipielle Punkte auf, die als Schwachstellen geortet werden können, wenn es um Sicherheitsaspekte geht.

✔ Grundsätzlich läuft JavaScript im Browser in einer **Sandbox**, was direkten Zugriff auf das Dateisystem ohne Erlaubnis des Users unmöglich macht (es existiert jedoch in HTML5 eine File-API). Von Haus aus ist das recht sicher.

In jedem Fall muss jedoch **unsicherer Code** (als externen Quellen) vermieden werden, da es durchaus Möglichkeiten gibt, hier Schaden zuzufügen (meist durch Abgreifen oder Modifizieren von Information in der Laufzeit).

Ein Problem ist die globale Funktion `eval()`, die einen ihr übergebenen **String** als JavaScript interpretiert. Sie stellt ein potenzielles Einfallstor für Schadcode dar:

```
// Angenommen, folgende Werte stammen aus einem Ajax-
Request:
var a = req.body.a; // enthält die Zahl 1
var b = req.body.b; // enthält die Zahl 2

// Alle Variablen sind für eval() sichtbar, daher:
var sum = eval(a + "+" + b); // weist String '1 + 2' zu
```

Solange die Request-Daten die erwartete Struktur haben, geht alles gut. Wird dem Request jedoch ein **unerwarteter Wert** untergeschoben, geschieht folgendes:

```
var a = req.body.a; // enthält die Zahl 1
var b = req.body.b; // String '2; alert("Buh");'
var sum = eval(a + "+" + b); // '1 + 2'; alert("Buh");
```

✔ Der Variable `sum` wird (wie erwartet) der *String* zugewiesen. Anschließend wird jedoch ein *zweites* Statement, nämlich das **Alert**, ausgeführt!

Die Anwendung von `eval()` kann dazu führen, dass untergeschobener **Schadcode** ausgeführt wird. Die Funktion sollte daher vermieden werden!

Die Funktion `eval()` hat nicht nur *lesenden* Zugriff auf den globalen Scope, sondern kann diesen auch **modifizieren**:

```
// hier erzeugt eval eine Variable a im globalen Scope
eval("var a = true");
console.log(a); // gibt true aus
```

Im *strict mode* von ECMA5 hat `eval()` keinen Zugriff mehr auf den globalen Scope:

```
// Dieses Script läuft im strict mode:
"use strict";
eval("var a = true"); // gelingt nicht!
console.log(a); // Referenz-Fehler – a existiert nicht!
```

8. Namensräume in JavaScript

Im Zuge der Framework-Entwicklung geht man dazu über, zusammengehörende Komponenten in mehrdimensionale Objektstrukturen zu gruppieren und hierfür zu verschachteln – im Gegensatz zur früher vorherrschenden Herangehensweise, einzelne „lose“ Objekte einzusetzen.

Einzelne Komponenten im Inneren der Objekte werden über einen Objektpfad angesprochen, der beim Containerobjekt beginnt. Dieses wird als Namensraum bezeichnet.

Beispiel:

```
YAHOO.util.Event.onAvailable(...)
```

8.1.1. Ansätze für Namensräume

```
// leeres Objekt dient als Namensraum
var meinNamensraum = {};

// dieser kann Objekte enthalten:
meinNamensraum.meinContainer = {
    meineEigenschaft1 : wert,
    meineEigenschaft2 : wert
```

```
    ...
};

// oder Funktionen (also Methoden):
meinNamensraum.meineMethode = function (...) {
    ...
};

var beispiel = meinNamensraum.meineMethode(...);
```

Eine andere Herangehensweise verwendet ein Function-Statement:

```
var meinNamensraum = function() {
    return {
        meineMethode : function() {
            // ...
        },
        andereMethode: function() {
            // ...
        }
    };
}();
```

Interessant ist dies, weil so die Möglichkeit besteht, zusätzlich in der Funktion weitere Variablen und Funktionen zu deklarieren. Diese sind von außen nicht zugänglich, also „privat“. Ihre Werte können aber über die inneren Funktionen erreicht werden – richtig, hier liegt eine Closure vor…

```
var meinNamensraum = function() {

    var privaterWert = "geheim...";

    // hier beginnt das Namensraumobjekt
    return {
        wert1 : 1,
        wert2 : 2,
        methode1 : function() {
            alert("Ich komme aus dem Namensraum!");
        },
        getPrivaterWert : function() {
            return privaterWert;
        }
    };
}();
```

Auf diesem Weg ist es möglich, weitere Werte oder Methoden im Funktionsobjekt zu deponieren, die zwar nicht selbst Teil des Namensraums sind, über diesen aber (mittels Closures) zugeänglich gemacht werden können – wie hier einfach demonstriert über Getter- und Setter-Methoden.

Der Wert der lokalen Variable `privaterWert` des Funktionsobjekts ist von außen nicht erreichbar, kann aber über die Namensraummethode `getPrivaterWert()` ausgelesen werden. Analog ist eine Methode denkbar, die den Wert neu setzen könnte:

```
// dies und ähnliches geht nicht:
alert(meinNamensraum.privaterWert); // -> undefined


// geht:
meinNamensraum.getPrivaterWert();  // gibt Wert aus
```

Nun ist es auch einfach, beispielsweise selbstdefinierte **DOM-Helfer** in einen eigenen Namensraum einzugliedern:

```
var myNs = function() {
    return {
          id : function id(name) {
            return document.getElementById(name);
          },
          tag : function (name, elem) {
            return (elem || document)
               .getElementsByTagName(name);
          }
          // mehr Methoden...
    };
}();
```

… und somit die Grundlage eines eigenen JavaScript-Frameworks “myNs” zu legen, mit dem man beispielsweise folgendes tun kann:

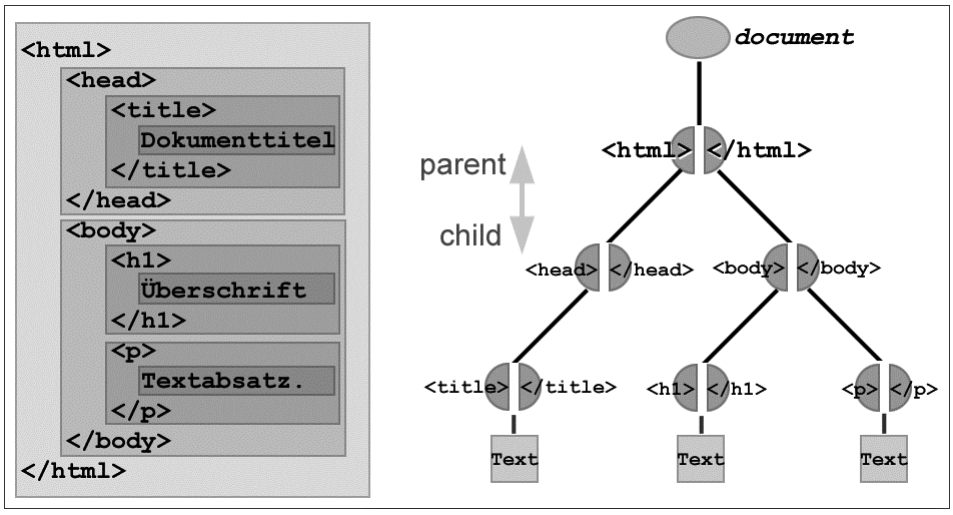
```
// Referenzen aller Divs des Dokuments:
var alleDivs = myNs.tag("div");

// alle P-Container in div#inhalt:
var alleP = myNs.tag("p", myNs.id("inhalt"));
```

9. DOM und Eventhandling

Das **DOM** stellt eine Abstraktion des Dokuments dar, die der Browser beim Einlesen des HTML-Quelltexts im Arbeitsspeicher erzeugt. Eigentlich existiert es also rein virtuell. Stellen Sie es sich als Baumstruktur vor, und zwar in Form eines nach unten hängenden Baums, der an einem Punkt aufgehängt ist und sich ab dort verzweigt. Bezeichnen wir den Punkt, an dem der Baum ansetzt, als *Dokumentknoten* (*documentNode*). Dieser Dokumentknoten besitzt keine Entsprechung im Dokument, sondern stellt das »Dokument an sich« dar.

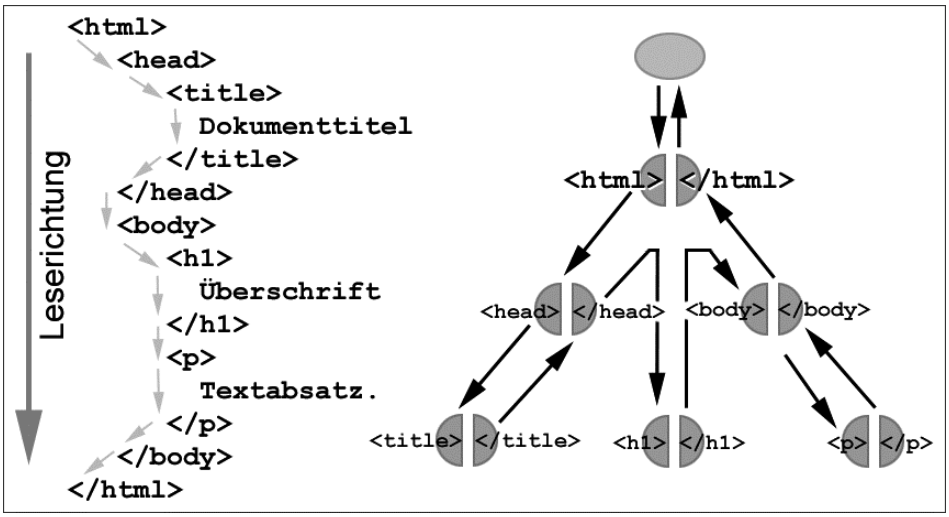
9.1. DOM –Browser als Laufzeitumgebung

Einen solchen Dokumentknoten erzeugt der Browser beim Einlesen (*Parsing*) des Dokuments. Er hängt dann für jedes Element, das er in Quelltextreihenfolge antrifft, einen weiteren Knoten unten an diesen Dokumentknoten an. Das erste Element, das er antrifft, ist stets das Root-Element HTML (allgemein als *documentElement* bezeichnet). Ab hier spaltet sich der Baum in zwei Zweige, die den `<head>` und den `<body>` repräsentieren (siehe Abbildung).



Der Baum besteht also aus Verbindungen und Knoten. Im Falle eines *Elementknotens* fallen in diesem Start- und Endmarke des Quelltexts zusammen. Textknoten bilden stets das Ende eines Zweiges (»leaf nodes«). Enthält ein Element einen Inhalt (gehen wir also hierarchisch in dessen Inneres), fächert sich der Baum weiter nach unten auf. Hierdurch entsteht für jeden Bestandteil dieses Inhalts ein weiterer Ast mit daran hängenden Knoten.

Hierbei gilt der oben liegende Knoten als »Elternknoten« (»parent«), die von ihm unmittelbar abstammenden Knoten als »Kindknoten« (»children«). Sowohl Elemente als auch Textknoten stehen stets in einer Eltern-Kind-Beziehung, wobei jeder Knoten genau einen Elternknoten besitzt (niemals mehrere). Ein Elementknoten besitzt darüber hinaus weitere Eigenschaften, zu denen (salopp gesprochen) auch seine Attribute gehören. Der Browser baut auf diese Weise sukzessive ein komplettes Abbild der hierarchischen Struktur des Dokuments auf.



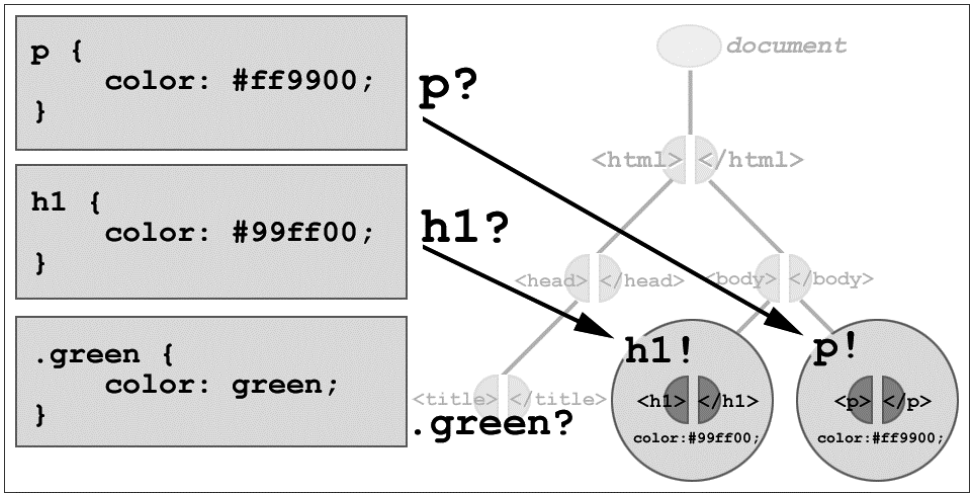
Das DOM ist also eine Abstraktion folgender Information:

- Wie ist der hierarchische Zusammenhang in der Dokumentstruktur?
- Welches Element ist an welcher Position der Hierarchie?
- Welche Eigenschaften hat es (z. B. Attribute, Inhalte, Nachbarelemente)?

9.1.1. Das »Schmücken« des DOM-Baums

Dieses Modell, in dem alle Elemente des Dokuments repräsentiert sind, ist noch eine Abbildung der »nackten« Struktur – durchaus wörtlich zu nehmen, denn bis jetzt hat der Browser die CSS-Informationen noch nicht eingebracht. Die nun folgende Phase wird als »decorating the tree« bezeichnet: Der Browser liest alle CSS-Informationen ein und löst dabei auftretende Konflikte und Unstimmigkeiten auf. Anschließend liegt das sogenannte *Stylesheet* vor: die vollständigen Präsentationsvorschriften, die auf das Dokument angewendet werden sollen.

Nun werden die Selektoren aller CSS-Regeln registriert und der Dokumentbaum anhand dieser Vergleichsmuster durchsucht. Jeder Elementknoten, auf den das Muster zutrifft, bekommt den Regelsatz zugewiesen. Auf diesem Weg werden alle Regeln durchgearbeitet. Regeln ohne Übereinstimmung mit dem Dokument liegen brach (siehe Abbildung).



Bekommt ein Element mehrfach Regeln zugewiesen, addieren sich diese. Ein auftretender Konflikt wird nach Rang der Regel beigelegt. Sobald der Vorgang beendet ist, ist der Baum »dekoriert«, und der Browser geht daran, ihn im Viewport darzustellen (*Rendering-Phase*).

Im dekorierten Baum sind folgende Informationen zusätzlich in den einzelnen Elementknoten gespeichert:

- Welche Präsentationsvorschriften existieren für dieses Element?

Die Stylevorschriften können wir uns als »Eigenschaften« des Elements vorstellen, die beim Elementknoten gelagert werden.

9.2. DOM – API für das HTML-Dokument

Der dekorierte Baum stellt ein Abbild des Dokuments im Arbeitsspeicher des Browsers dar, das der Darstellung des Dokuments im Viewport entspricht. Da es sich um eine rein virtuelle Sache handelt, ist dieses Abbild per Programmierung beliebig manipulierbar. Genau dies ist die Aufgabe des DOM – eine Schnittstelle (API) zu bieten, die es ermöglicht, mittels einer Programmiersprache auf das Dokument einzuwirken.

In Zusammenhang mit JavaScript bietet das DOM eine Reihe von Schnittstellenfunktionen, die eine Brücke schlagen zwischen der Scripting-Umgebung und dem Dokumentbaum. Einige dieser Methoden sind nur

dem Dokumentknoten zugeordnet. Andere Methoden stehen auch direkt den Elementknoten zur Verfügung.

Diese DOM-Methoden sind nicht ausgesprochen zahlreich und zum Teil umständlich anzuwenden – ihre Anzahl hängt zudem von der Implementierung des DOM ab (»DOM-Level«), die der ausführende Browser jeweils unterstützt.

Dies sind ihre Aufgaben:

- Selektieren von Knoten
- Traversieren (Bewegung) innerhalb des Baums
- Wert (Inhalt) eines Knotens auslesen oder schreiben
- Attribute eines Elementknotens lesen oder schreiben
- Erzeugen von Element- und Attributknoten
- Einhängen von Knoten in den Baum
- Löschen von Knoten im Baum

DOM-Methode	Erläuterung
`addEventListener()`	Bindet Event Handler an DOM-Element.
`removeEventListener()`	Löst Event Handler an DOM-Element.
`createAttribute()`	Erzeugt einen Attributknoten.
`createTextNode()`	Erzeugt einen Textknoten.
`createElement()`	Erzeugt einen Elementknoten.
`getAttribute()`	Liest einen Attributwert.
`setAttribute()`	Schreibt einen Attributwert.
`removeAttribute()`	Entfernt einen Attributknoten.

Tab.: Die wichtigsten DOM-Methoden zur DOM-Manipulation (I)

DOM-Methode	Erläuterung
`appendChild()`	Hängt Elementknoten am Ende des Inhalts des aktuellen Knotens ein.
`insertBefore()`	Hängt Elementknoten vor dem aktuellen Knoten ein.
`removeChild()`	Entfernt Kindknoten des aktuellen Knotens.
`replaceChild()`	Ersetzt Kindknoten des aktuellen Knotens.

Tab.: Die wichtigsten DOM-Methoden zur DOM-Manipulation (II)

9.2.1. DOM – Selektieren und Traversieren

Das *Selektieren* eines Knotens im Dokumentbaum geschieht in JavaScript über eine Methode, die diesen Knoten als Objekt zurückliefert. Die direkteste Art geschieht über einen *Identifizierer*, der dem Knoten als `id`-Attribut beigegeben ist. Die `document`-Methode `getElementById()` nimmt einen ID-Namen als Zeichenkette entgegen (es wird also nicht wie beim CSS-Selektor die Raute vorangestellt). Sie liefert einen *Einzelknoten* als DOM-Element zurück:

```
// liefert einen Elementknoten als Objekt zurück:
var meinKnoten = document.getElementById("meinID");
```

DOM-Methode	Erläuterung
`getElementById()`	Referenziert einen Elementknoten per ID.
`getElementsByClassName()`	Erstellt NodeList aus Elementknoten nach CSS-Klasse.
`getElementsByTagName()`	Erstellt NodeList aus Elementknoten nach Tag-Bezeichner.

Tab.: Die wichtigsten DOM-Methoden zur DOM-Manipulation (III)

Eine weitere Methode steht in Form `getElementsByTagName()` zur Verfügung, die einen *Elementnamen* als String akzeptiert. Elementnamen treten mehrfach im Dokument auf.

Demzufolge gibt die Methode auch keinen Einzelknoten zurück, sondern mehrere, die in einer sogenannten *Collection* zusammengefasst sind. Für unsere Begriffe kann eine Collection mit einem Array gleichgesetzt werden (technisch betrachtet ist sie jedoch keines).

9.3. DOM – Events binden und lösen

Ereignisse in einer HTML-Seite treten auf, wenn etwas mit beliebigen Elementen der Seite geschieht. Dies kann mit Nutzeraktionen zusammenhängen, wie dem Anklicken, Selektieren oder Ziehen von Objekten, aber auch mit Vorgängen, wie dem Laden oder Entladen des Dokuments. Damit ein Event sinnvoll behandelt werden kann, sind, salopp gesagt, drei Dinge von Bedeutung:

- Das Ereignis muss überhaupt bemerkt werden.
- Es muss bekannt sein, um was für ein Ereignis es sich handelt, an welchem Element und bei welchen Koordinaten es auftritt.

- Eine Funktion muss mit dem Auftreten des Events verknüpft sein.

Für die Erfüllung von Punkt eins sorgt der Event Listener, der mit dem Elementknoten verknüpft sein muss, an dem das Ereignis auftritt – vereinfacht gesagt (Sie werden gleich sehen, dass dies nicht ganz stimmt).

Der Event Listener kann über ein Attribut in den HTML-Code geschrieben oder per JavaScript nachträglich (also »unobtrusive«) hinzugefügt worden sein. Aufgabe des Listeners ist es, den sogenannten Event Handler zu starten, ein Funktionsobjekt, das die JavaScript-Anweisungen enthält, die beim Eintritt des Ereignisses, auf das der Handler wartet, ausgeführt werden. Dies erfüllt Punkt drei:

```
<p id="p1" onclick="alert('Ein Ereignis trat auf.')">
   Bitte hier klicken!</p>
```

Würde das Ereignis per Script gebunden, sähe dies etwa so aus (dies ist wirkungsgleich zur eben gezeigten Inline-Variante):

```
var meinP = document.getElementById('p1');

meinP.addEventListener('click', function() {
        alert('Ein Ereignis trat auf.');
}, false);
```

9.3.1. Das Eventobjekt

Welche Informationen sind über das Ereignis bekannt, sprich, was ist mit Punkt zwei der eben angeführten Liste? Klar ist, dass es sich um ein Ereignis vom Typ »Click« handelte, das an einem <p>-Container auftrat, den man deshalb als Target bezeichnet. Der Klick fand auch an bestimmten Koordinaten (genannt »pageX« und »pageY«) innerhalb des Textabsatzes statt, die vielleicht interessant sein könnten.

Praktischerweise fasst JavaScript eben diese Informationen (und noch mehr) zusammen und stellt sie als Objekt zur Verfügung, das deshalb als Eventobjekt bezeichnet wird. Ein solches **Eventobjekt** wird bei jedem auftretenden Ereignis gebildet. Bevor Sie zu überlegen beginnen, wie Sie an dieses Objekt herankommen, um es zu verwenden – hierfür ist bereits gesorgt: Das Eventobjekt wird stets dem Event Handler übergeben.

An dieser Stelle ist es nötig, zu präzisieren, *was genau* der Event Handler eigentlich ist – und zwar ist dies die an das Ereignis gebundene *Funktion*.

Einfacher zu zeigen ist dies bei der Scriptbindung – die Funktion ist fett markiert:

```
meinP.addEventListener('click', function() {
      alert('Ein Ereignis trat auf.');
      }, false);
```

Ein Wert, der einer Funktion übergeben wird, muss auch in einem Übergabeparameter aufgefangen werden, sonst landet er im Datennirvana (bislang geschieht das noch mit dem Eventobjekt). Nennen Sie die Variable e, und nehmen Sie weiter an, dass das Eventobjekt tatsächlich dort landet. Sie können die Art des Ereignisses aus dem übergebenen Objekt dann wie folgt auslesen:

```
meinP.addEventListener('click', function(e) {
      alert('Ein ' + e.type + '-Ereignis trat auf.');
}, false);
```

Eine analoge Methode ist für das Event-Handler-Attribut nicht möglich. Zwar wird *implizit* ebenfalls ein anonymes Funktionsobjekt um die Anweisungen im Attributwert gehüllt (Achtung – das Beispiel ist eine rein illustrative Verdeutlichung des Geschehens):

```
<!-- So kann man sich das vorstellen: -->
<p id="p1"
   onclick="function(){ alert('Ein Ereignis trat auf.')
 }">
   Bitte hier klicken!</p>
```

Da aber keine Möglichkeit besteht, den Wrap in ein Funktionsobjekt explizit vorzunehmen, kann kein Übergabeparameter bestimmt werden, der das Eventobjekt in den Anweisungsblock hineinreicht. Das ist aber nicht weiter schlimm, da Inline-Event-Handler sowieso möglichst wenig eingesetzt werden sollten.

Das Eventobjekt ist im Prinzip eine gute Sache. Leider ist auch dieser Aspekt nicht plattformübergreifend implementiert und wie immer folgt Internet Explorer (bis *Version 9*) nicht dem Standard und definiert das Eventobjekt als *globales Objekt* – immerhin mit »weitestgehend« ähnlichen Eigenschaften. (Da die erste Implementierung durch Microsoft allerdings vor der Standardisierung stattfand, sei dies verziehen. Tatsache ist, dass so vom Programmierer einiges an Verrenkungen verlangt würde, um plattformübergreifenden Code zu schreiben.)

9.3.2. Phasen des Eventhandlings

Tritt irgendwo im DOM-Baum ein Ereignis auf, beginnt seine (mögliche) Erfassung stets an der DOM-Wurzel. Von dort aus wandert das Ereignis ins Innere des Baums bis zum sogenannten *Target*, dem eigentlichen Ort, an dem das Ereignis stattgefunden hat. An jedem Knoten, der dabei passiert wird (der als *Observer* bezeichnet wird), kann das Ereignis über einen *Listener* erfasst werden. Diese Phase der Ereignisbehandlung ist die **Capture-Phase**.

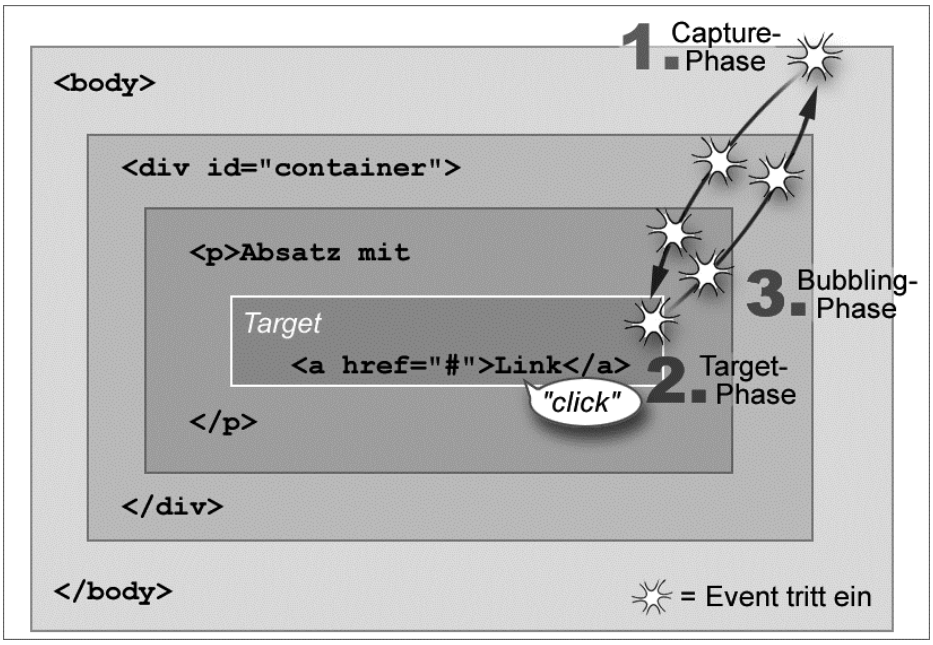


Abb.: Die drei Phasen des Eventflusses

In der Abbildung tritt am Link (dem *Target*) ein Klick auf, der während der *Capture*-Phase jedoch zuerst am Body, dann an einem Div und schließlich am `<p>`-Container, der den Link umgibt, erkannt werden kann.

Anschließend feuert das Event am *Target* selbst (Phase 2, die **Target-Phase**) und wandert dann von dort den Baum wieder nach oben (daher Bubbling) zum Elternknoten, dessen Elternknoten und schließlich, durch alle Instanzen, bis zur Dokumentwurzel (dem HTML-Tag). Dies ist die dritte Phase, die **Bubbling-Phase**. Auch hier kann ein Ereignis an jedem Punkt des Weges über einen Listener erkannt und mit einem Handler verarbeitet werden.

In welcher der Phasen ein Ereignis erkannt wird, hängt davon ab, wie der Listener definiert worden ist. Am Target selbst tritt das Ereignis nur einmal auf, vor Beginn der Bubbling-Phase.

Soll ein Event Listener für die Capture-Phase definiert werden, bekommt der dritte Parameter der `addEventListener()`-Methode den Wert `true`:

```
// Capture-Listener
document.body.addEventListener('click', function(){
      alert('Klick am Body erfasst (Capture)!');
   }, true);
```

Der Clou ist, dass das Ereignis in der Capture-Phase am Body erfasst wird, *bevor* es das eigentliche Target erreicht hat. Dort kann es natürlich ebenfalls aufgefangen werden. Anschließend beginnt die Bubbling-Phase. Ist ein zweiter (Bubble-)Listener für den Body definiert, kann das Ereignis auf dem Rückweg ein weiteres Mal aufgefangen werden. Der dritte Parameter ist `false`:

```
// Bubble-Listener
document.body.addEventListener('click', function(){
      alert('Klick am Body! (Bubbling)');
   }, false);
```

Es ist wichtig, zu verstehen, dass das Standardverhalten die *Weitergabe* des Ereignisses (auch **Propagation** genannt) ist. Es ist jedoch möglich, die Weitergabe zu unterbinden.

Unterbinden der Propagation in der Capture-Phase

Auf folgende Weise können Sie dem Event Listener im Body für die Capture-Phase befehlen, das Event *nicht* weiterzuleiten:

```
// Fängt das Event für confirm=true ab:
document.body.addEventListener( 'click', function(e){
      if(confirm('Event nicht weiterleiten?')) {
         e.stopPropagation();
      }
   }, true );
```

Halten Sie das Event durch `stopPropagation()` in der Capture-Phase auf, kann es an weiter innen im Baum liegenden Elementen nicht mehr erfasst werden. Eine Weitergabe kann aber auf gleiche Art auch erst auf dem Rückweg, vor oder während der Bubbling-Phase unterbrochen werden

9.3.3. Binden von Events

Wichtig ist, dass eine Manipulation des DOM (und dazu gehört auch das Binden von Events) erst geschieht, sobald das Dokument fertig geladen ist. Hierfür bindet man in der Regel eine anonyme Funktion an den `window.onload`-Eventhandler:

```
// mach was, sobald das Dokument fertig ist:
window.onload = function() {

   var meinP = document.getElementById("p3");
   console.log(meinP);

   meinP.addEventListener("click",function(){
      //alert("Geklickt!");
      this.style.backgroundColor = "#ffaaaa";
      this.setAttribute ("class", "green");
   }, false);
}
```

Holt folgenden Textabsatz aus dem Dokument und bindet einen Click-Handler:

```
<p id="p3">Ein Textabsatz it ID #p3</p>
```

Sollen mehrere Handler gebunden werden, so geschieht dies in einer Schleife:

```
window.onload = function() {

   // Element "Collection"
   var meineP = document.getElementsByTagName("p");
   console.log(meineP.length);

    for(var i=0, len = meineP.length; i < len; i++) {
        meineP[i].addEventListener("click",function(){
        this.style.backgroundColor = "#ffaaaa";
        this.setAttribute ("class", "green");
    }, false);
  }
}
```

10. Ajax & JSON

Aus dem aktuellen Webdesign ist Ajax nicht mehr wegzudenken. Zu sehr hat man sich daran gewöhnt, dass Webseiten nach einer Interaktion Daten nachladen und dass diese dann flüssig in die Seite eingebaut werden. Im früheren, »statischen« Seitenmodell war dies anders – der Server lieferte HTML und CSS, und der Client stellte dies dar.

Eine gewisse Dynamik war möglich, aber nur, wenn die dafür erforderlichen Daten (Scripte, Informationen, Bilder) bereits clientseitig vorhanden waren. Eine erneute Anfrage des Browsers bedeutete stets, eine komplette Seite anzufordern und diese dann neu im Viewport aufzubauen.

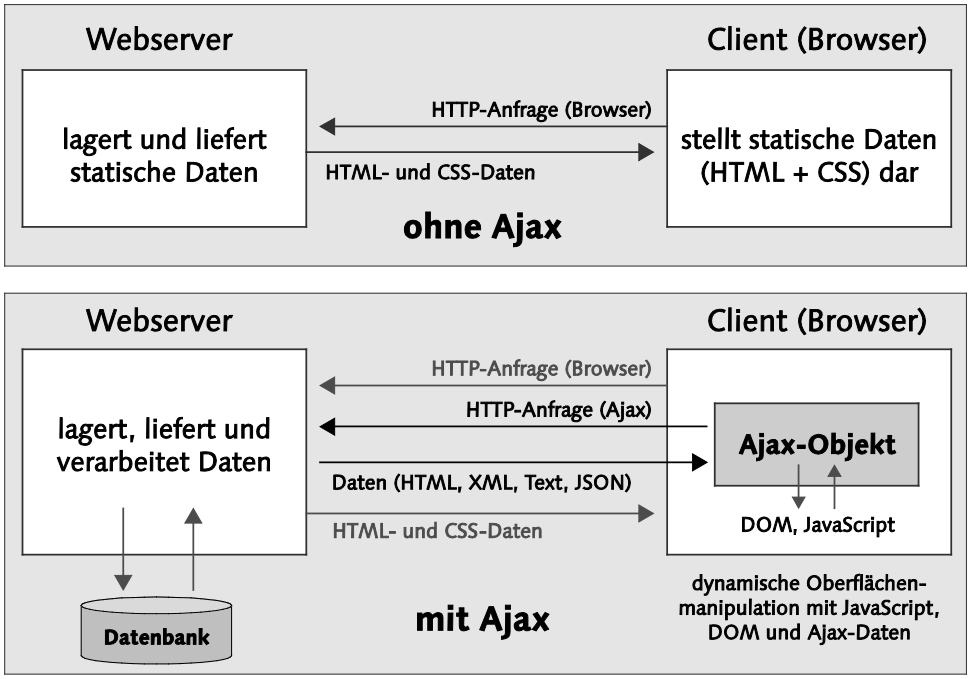


Abb.: Statisches Web (oben) und Ajax-Web (unten) im Vergleich

Mit dem in JavaScript implementierten Ajax-Objekt `XMLHttpRequest` wurde es möglich, aus einer dargestellten Seite heraus, gewissermaßen an den üblichen Kanälen des Browsers vorbei, separate HTTP-Anfragen zu starten, mittels derer auch nachträglich Informationen geholt werden konnten (die auch durch eine Datenbankabfrage etc. generierbar waren). Dies geschieht obendrein »asynchron«, d. h., noch während eine Anfrage läuft, können Scripte weiterverarbeitet werden.

Das Ajax-Objekt sendet und empfängt Daten, bemerkt und meldet also, wenn die gewünschten Informationen eingetroffen sind. Diese werden dann durch JavaScript in das DOM des Dokuments eingebaut und erscheinen somit in der Seite (sieheAbbildung).

10.1. Grundlagen zu Ajax

Prinzipiell ist eine Ajax-Anfrage keine besonders komplexe Sache – der Teufel steckt (wie üblich) im Detail, sprich, es gibt eine aktuelle standardisierte Implementierung und mehrere ältere von Microsoft.

Eine Ajax-Anfrage kann in vier Schritte unterteilt werden:

1. Ajax-Objekt erzeugen
2. Verbindung zum Server definieren
3. Daten abfragen (d. h. die Anfrage schicken, gegebenenfalls mit Parametern)
4. Antwort (oder Fehlermeldung) des Servers entgegennehmen

Schritt 1

Per JavaScript muss ein Objekt generiert werden, das die HTTP-Anfrage vornimmt und die Daten entgegennimmt. Das Objekt wird traditionell `xhr` genannt. Die standardisierte Methode (und für Internet Explorer ab 7.0), ein XHR-Objekt zu erzeugen, funktioniert so:

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

Schritt 2

Das erzeugte Objekt muss eine Verbindung zum Server aufnehmen. Hierfür dient die Methode `open()` des `XMLHttpRequest`-Objekts. Die Methode erwartet drei Parameter:

```
xhr.open(HTTP-Methode, Zieladresse, Requestmethode);
```

Parameter 1 – die HTTP-Methode:

Die Methode wird als String übergeben. Wie bei Formularen, muss hier entweder »GET« oder »POST« angegeben werden. Groß- und Kleinschreibung sind irrelevant – eingebürgert hat sich Großschreibung:

```
xhr.open("POST", ..., ...); // nehmen wir mal POST
```

Parameter 2 – die Zieladresse:

Die Adresse wird als String übergeben. Dieser bezeichnet (eventuell mit Pfad) den URL der Datei, die angefordert wird, bzw. die Ressource, die die Daten erzeugt, die verwendet werden sollen:

```
xhr.open("POST", "beispiel.html", ...);
```

Parameter 3 – die Request-Methode:

Der dritte Parameter ist ein Boolescher Wert. Er wird daher ohne Anführungszeichen übergeben und lautet entweder `true` oder `false`.

- `false`: Scriptausführung wird angehalten, bis der Server die Daten zurückliefert (synchron).
- `true`: Scriptausführung läuft weiter; HTTP-Anfrage wird im Hintergrund ausgeführt (asynchron).

Eine synchrone Abfrage wählt man nur, wenn die angeforderten Daten für das korrekte Weiterverarbeiten der Seiten unerlässlich sind – dies kann (eher selten) bei Formularverarbeitung erforderlich sein. Für eine asynchrone Abfrage (die Regel) steht daher *true*:

```
xhr.open("POST", "beispiel.html", true);
```

Das Problem, auf das man nun stößt, liegt auf der Hand: Das Eintreffen der Daten nach erfolgreicher Anfrage muss natürlich erkannt werden – und dies, obwohl das Script gleichzeitig weiterläuft. Glücklicherweise besitzt das XHR-Objekt eine Eigenschaft, die hierfür abgefragt werden kann: `xhr.readyState`.

readyState	Bedeutung	Erläuterung
0	Nicht initialisiert (*unsent*)	Das XHR-Objekt existiert, aber noch war nichts los.
1	Initialisiert (*opened*)	`open()`-Methode wurde erfolgreich aufgerufen.
2	Kontaktiert (*headers received*)	Server hat mit allen erforderlichen HTTP-Headern geantwortet.
3	Daten kommen (*loading*)	Antwort wird geladen.

Die vier Zustände von xhr.readyState

Den Zustandswechsel von `readystate` müssen Sie erkennen und behandeln (d. h., eine Funktion wird gestartet). Hierzu können Sie den Event Handler `onreadystatechange` einsetzen – und zwar ebenfalls am XHR-Objekt (als Event Listener):

```
xhr.onreadystatechange = handler;  //->  referenziert
```

Damit die Funktion `handler()` gestartet wird, müssen Sie diese an den Zustandswechsel binden. In diesem Fall als Referenz, weswegen *keine* Funktionsklammern verwendet werden dürfen.

Schritt 3

Die `send()`-Methode des Objekts sendet die Daten an den Server. Sie erwartet bei `POST` einen Parameter, nämlich die `name-value`-Pärchen der Daten (als Querystring; beispielsweise: `"wert1=xxx&wert1=yyy"`):

```
xhr.send(); // keine Parameter übergeben (leerer String
)
```

Achtung – wenn Daten per GET übertragen werden, **muss** der `send()`-Methode der Wert `null` übergeben werden!

```
var xhr = null;        // erstmal Variable erzeugen
if (window.XMLHttpRequest) {
   xhr = new XMLHttpRequest();
} else if (window.ActiveXObject) {
   xhr = new ActiveXObject("MSXML2.XmlHttp.3.0");
}
// ist auch nichts schiefgegangen?
if (xhr != null) {
   xhr.open("POST", "beispiel.html", true);
// Funktion binden
   xhr.onreadystatechange = auswerten;
// und los (keine Daten diesmal)
   xhr.send();
}
```

Die Funktion `auswerten()` überprüft bei *jedem* Statuswechsel, ob der erreichte Zustand den Wert 4 hat:

```
function auswerten() {
   // es interessiert jedoch nur Zustand 4:
```

```
   if (xhr.readyState == 4) {
      // wenn die Daten da sind, was machen...
      alert("Hey, es sind frische Daten da!");
   }
}
```

Schritt 4

Die empfangenen HTML-Daten befinden sich ebenfalls im XHR-Objekt, und zwar in den meisten Fällen als Textstring in das Property `responseText` gespeichert. Sie können sie »provisorisch« einfach per `alert()` oder über die Firebug-Konsole ausgeben:

```
function auswerten() {
   // sind wir "fertig"?
   if (xhr.readyState == 4) {
       // zusätzlich wird der Response-Status geprüft:
       if( xhr.status >= 200 && xhr.status < 300) {
          // wenn  brauchbare Daten da sind, reagieren:
          alert(xhr.responseText);
       }
       else {
          // es ist was schiefgegangen
       }
   }
}
```

Vom Server bekommt das XHR-Objekt auch eine Statusmeldung, die in dessen Eigenschaft `status` landet und ausdrückt, ob die Anfrage letztendlich erfolgreich war oder nicht: Es hilft wenig, wenn das `responseText`-Property lediglich eine Fehlermeldung enthält (deshalb sagt es nichts aus, wenn es »nicht leer« ist).

Statusmeldung des Servers – Erfolg oder Fehler?

Eine *HTTP-Statusmeldung* wird vom Server in Form einer dreistelligen Ziffer gemeldet. Generell lässt sich sagen, dass eine Zahl zwischen 200 und 299 als *Erfolg* sowie alle Zahlen ab einschließlich 300 als *Fehler* zu werten sind. Am bekanntesten sind die Werte 200 für »Erfolg« und 404 für »Resource nicht gefunden«. Jedoch ist es sicherer, anstatt auf exakte Werte auf Bereiche zu prüfen (es sei denn, Sie kennen Ihren Server persönlich).

Per `alert()` geben Sie die Daten natürlich nicht aus. Sie sollen ja in die HTML-Seite geschrieben werden. Hierfür bereiten Sie z. B. einen Container vor, der einen ID erhält:

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

Dieser kann nun über DOM angesprochen werden

```
document.getElementById("ausgabe") ...
```

und seinen Inhalt verändern

```
document.getElementById("ausgabe").innerHTML ...
```

indem der Dateninhalt des XHR-Objekts zugewiesen wird

```
document.getElementById("ausgabe").innerHTML =  xhr.res
ponseText;
```

Das Ergebnis sieht so aus:

```
<div id="ausgabe">
<h2>Dies ist ein AJAX-Test</h2>
<p>Dieser Text steht in der angeforderten Datei.</p>
<p>Es k&ouml;nnte nat&uuml;rlich auch eine
   Datenbankabfrage stattfinden...</p>
</div>
```

10.2. Daten und Datentypen für Ajax

Grundsätzlich geht es bei einer Ajax-Anfrage darum, dass der Server den Client mit Daten versorgt. Der Datentyp kann intern explizit festgelegt werden, was über die anschließende Verarbeitung entscheidet.

Textdaten

Der einfachste Fall ist, dass die Daten einfach in Form einer Zeichenkette übertragen werden. Diese kann beispielsweise anschließend als Inhalt in irgendeinen Tag-Container im Dokument geschrieben werden. Die innere Struktur der Daten ist entweder nicht relevant, oder es existiert gar keine. Solche Daten landen generell in der `responseText`-Eigenschaft des XHR-Objekts.

HTML

Eine Variante von Textdaten sind HTML-Daten. Eigentlich wird auch hier nur eine Zeichenkette übertragen, die allerdings aus HTML-Quellcode besteht. Auch HTML-Daten landen in `responseText` (es wird also im Grunde nicht zwischen HTML und Text unterschieden) und können über die Eigenschaft `innerHTML` in ein Element geschrieben werden.

Anschließend sorgt der Browser dann für die Interpretation des Codes. Es ist wichtig, zu wissen, dass in den HTML-String eingebettete JavaScript-Befehle (in Form von Scriptblöcken) ausgeführt werden, *bevor* der HTML-Inhalt eingefügt wird.

JSON

Das JSON-Format erfreut sich aufgrund seiner Kompaktheit in Zusammenhang mit Ajax inzwischen großer Beliebtheit. JSON (für »JavaScript Object Notation«) verpackt strukturierte Datenbeschreibungen in Objektliterale, die als Zeichenketten übertragen werden.

Diese Zeichenketten (es muss natürlich bekannt sein, dass es sich um JSON handelt) müssen nach Empfang noch geparst werden, um in echte JavaScript-Objekte umgewandelt werden zu können. Hierfür existieren als JSON-Parser bezeichnete Hilfsanwendungen. Auch JSON-Daten werden in das `responseText`-Property abgelegt.

Beispiel für einen JSON-String:

```
'{ "blumen1":"Rosen",
   "blumen2":"Tulpen",
   "blumen3":"Nelken" }'
```

Entgegen den sonst in JavaScript üblichen Gepflogenheiten müssen Strings in JSON-Daten stets mit *doppelten Anführungszeichen* umgeben werden. Vorsicht also mit »von Hand geschriebenem« JSON.

XML

Das Datenformat, dem Ajax zum Teil seinen Namen verdankt, ist XML. Grundsätzlich ist XML konzipiert, um strukturierte Daten zu beschreiben. Damit ist es vergleichbar mit JSON – allerdings besitzen beide Formate Stärken und Schwächen. Die Stärke von XML besteht in der beliebig komplexen Natur der Daten. XML muss allerdings erst in einen DOM-Baum umgewandelt werden, während geparstes JSON unmittelbar als zugängliches JavaScript-Objekt vorliegt.

11. JavaScript-Frameworks

Unter **Frameworks** versteht man grundlegende Bibliotheken, die dazu dienen, dem Programmierer stets wiederkehrende Aufgaben abzunehmen und somit die Entwicklung von Scripts zu beschleunigen. Ein gewünschter Nebeneffekt ist die Berücksichtigung und Umgehung browserspezifischer Bugs, was in einer browserübergreifen einheitlichen Funktionalität der erzeugten Scripte resultiert.

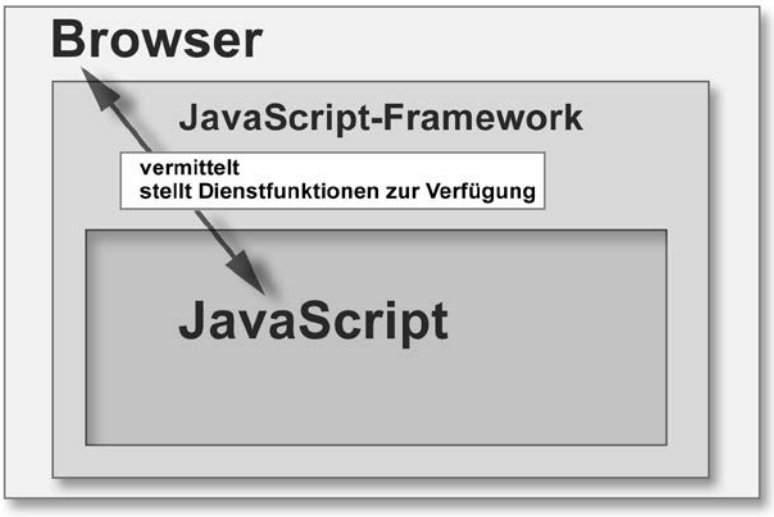


Abb.: Das Framework bildet eine Zwischenschicht

✔ Wörtlich übersetzt bedeutet „Framework“ (Programm-)Gerüst, Rahmen oder Skelett. Darin wird ausgedrückt, dass ein Framework in der Regel eine Anwendungsarchitektur vorgibt. (Wikipedia)

Übersetzen ließe sich der Begriff „Framework“ mit „Rahmenwerk“ oder schlicht „Rahmen“. Man kann es sich als ein um ein zu erstellendes Programm errichtetes Gerüst vorstellen, das als zusätzliche Schicht zwischen jenem Programm und der Außenwelt mit diesen zusammen in der gleichen Laufzeitumgebung betrieben wird.

Ein **JavaScript-Framework** ist daher auch „nur“ ein in JavaScript geschriebenes Programm – allerdings ein recht komplexes.

11.1. Unterscheidung

Frameworks, gerade solche für JavaScript, gibt es heutzutage quasi „im Dutzend“. Unter diesen kristallisieren sich derzeit einige wenige heraus, die häufiger zum Einsatz kommen.

- **jQuery**
  http://www.jquery.com/

  Frei verfügbares Javascript-Framework, aktuell in den **Version 1.12.4** (abwärtskompatibel bis IE6) und **Version 2.2.4** (ab IE9) verfügt über umfangreiche Funktionen zur Navigation und Manipulation der DOM-Syntax. jQuery ermöglicht einfache, „spaßorientierte“ Entwicklung von JavaScript und vereinfacht hierbei den Einsatz von Code. Umfang in der gepackten Version lediglich 32 kB.

Die Website www.jquery.com

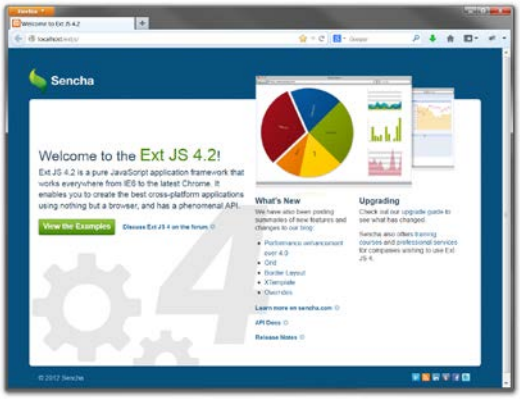
- **Dojo Toolkit**
  http://dojotoolkit.org/

  Dojo ist ein AMD-orientiertes, modulares Open Source Framework mit (wortwörtlich!) umfassenden Features, das eine Low-Level und eine High-Level Api verbindet und auf Business-Applikationen ausgerichtet ist. Aktuell ist die Version **Dojo 1.11**.

- **ExtJS**
  http://www.sencha.com/

  Der Funktionsumfang von ExtJS ist vergleichbar mir der von Dojo und umfasst Oberflächenwidgets für Formulare, Grids, Layouts und mehr. Darüberhinaus stellt Ext ebenfalls eine Sammlung aus Basisfunktionen

als Low-Level-API (Ext Core) zur Verfügung. Ext ist nur für nicht-kommerzielle Projekte frei verfügbar. Aktuell ist **ExtJS 6.2**.



Die Website www.sencha.com

- **AngularJS**
  http://www.angularjs.org/

  Von Google entworfenes, frei verfügbares Framework, das als neuen Ansatz 2-Way-Databinding und MVC propagiert und speziell auf die erstellung von Single-Page-Webapplications (SPA) ausgerichtet ist.

Die Website www.angularjs.com

11.2. Beispiel jQuery

Um zu verdeutlichen, wie **jQuery** dem User tagtäglich viel Arbeit abnimmt und ihm so das Webleben erleichtert, möchten wir hier zunächst mit einem Beispiel anfangen, das – noch ohne die Hilfe von jQuery – die Manipulation eines HTML-Elements vornimmt.

11.2.1. Vergleich: JavaScript mit und ohne jQuery

Hier ein Alltags-Beispiel, das die Manipulation eines HTML-Elements vornimmt. Angenommen, Sie wollen, abhängig von seinem Textinhalt, einem von mehreren <p>-Absätzen die Klasse `green` zuweisen und ihm damit eine neue Schriftfarbe geben.

Variante A: Ohne jQuery

Hier zunächst die Variante ohne die Hilfe von jQuery. Selbstverständlich lässt sich alles mit den gewohnten DOM-Methoden und herkömmlichem JavaScript lösen.

```
<html>
<head>
<title>Listing 1</title>
   <style type="text/css">
      .green { color:#009933; }
   </style>
   <script type="text/javascript">
      window.onload = function() {
         var elements = document.getElementById("box")
             .getElementsByTagName("p");
         for (var i = 0; i < elements.length; i++) {
            if (elements[i]
            .firstChild.data == "Zweiter Absatz") {
               elements[i].className = "green";
            }
         }
      }
   </script>
</head>
<body>
   <div id="box">
      <p>Erster Absatz</p>
      <p>Zweiter Absatz</p>
      <p>Dritter Absatz</p>
   </div>
</body>
</html>
```

Listing: Schriftfarbe ändern ohne jQuery

Sobald die komplette HTML-Seite geladen ist (`window.onload`), wird eine Kollektion mit allen Absätzen ("p") erstellt, die sich innerhalb des

Containers mit dem ID ("box") befinden. Diese Kollektion wird in einer Variablen `elements` gespeichert. Anschließend werden über eine `for`-Schleife und eine if-Anweisung alle gefundenen <p>-Container dahingehend geprüft, ob sie einen Textknoten mit Inhalt »Zweiter Absatz« enthalten.

Wenn dem so ist, wird dem betreffenden Absatz die Klasse `green` hinzugefügt und damit dessen Schriftfarbe auf »Grün« gesetzt. (Sie brauchen an dieser Stelle nicht zu sehr in die Tiefe zu gehen, um sich klarzumachen, dass Sie hier genau überlegen müssen, wie Sie in Ihrem Script mit Bedingungen und Schleifen Ihre Suchergebnisse filtern, bis am Ende das gewünschte Ergebnis im Browser erscheint.)

Variante B: Mit jQuery

Die Datei von vorhin muss ein wenig umgebaut werden. Legen Sie ein weiteres `<script>`-Element an, und fügen Sie die Framework-Datei ein. Das gesamte Beispiel sieht jetzt folgendermaßen aus (der jQuery-Code ist fett hervorgehoben):

```
<html>
<head>
   <title>Listing 2</title>
   <!-- Zunächst die Styleangaben: -->
   <style type="text/css">
      .green { color:#009933; }
   </style>

   <!-- Nun das Framework einbinden: -->
   <script type="text/javascript"
           src="../jquery/jquery.js"></script>

   <!-- ...und jetzt unseren eigenen Code: -->
   <script type="text/javascript">
      $(document).ready(function() {
         $("#box p:contains('Zweiter Absatz')")
          .addClass("green");
      });
   </script>
</head>
<body>
   <div id="box">
      <p>Erster Absatz</p>
      <p>Zweiter Absatz</p>
      <p>Dritter Absatz</p>
```

```
   </div>
</body>
</html>
```

Listing: Schriftfarbe ändern mit jQuery

Sofort ins Auge springt, dass gerade einmal drei Zeilen Code benötigt werden, um das gesteckte Ziel zu erreichen.

Die im vorigen Listing eingesetzten DOM-Funktionen `getElementById()` und `getElementsByTagName()` holen »nur« existierende Bestandteile des Dokuments, verändern diese aber nicht. Anders bei der `$()`-Funktion. Diese begnügt sich nicht damit, sich das bezeichnete Objekt zu beschaffen, es stattet dieses mit weiteren Fähigkeiten aus, die das »alte« Objekt wie ein Mantel umgeben – man spricht daher von »Wrapping«.

Das so erzeugte »neue« Objekt wird (unabhängig davon, welche Art von Objekt als Grundlage fungiert) als **jQuery-Objekt** bezeichnet.

11.2.2. Das jQuery Objekt

Mit dem Aufruf von `$()` wird also stets ein neues jQuery-Objekt erzeugt und zurückgegeben. Danach steht es mit seinen ganzen Methoden und Eigenschaften für das weitere Scripting zur Verfügung.

Die ready()-Methode

Eine dieser Methoden nennt sich `ready()`. Sie wird allerdings speziell dann verwendet, wenn das jQuery-Objekt aus einem `document`-Objekt gebildet wurde (zugegeben, ein Sonderfall). Bei `ready()` handelt es sich um eine grundlegende Methode innerhalb des Eventmoduls von jQuery: Eine Funktion *fn*, die an `ready()` übergeben wird, führt jQuery aus, sobald das Dokument geladen wurde. In unserem Fall ist es eine anonyme Funktion, die die Dokumentabfrage enthält.

✔ Der Grund für dieses Manöver ist folgender: Wenn Sie die Anweisung `$("#box p").addClass("green")` unmittelbar auswerteten, erzielten Sie keine Treffer – in diesem Moment gäbe es schlicht noch kein Dokument, aus dem das Script den <p>-Container holen könnte.

Analysieren wir nun den hervorgehobenen Code aus dem vorigen Listing Zeile für Zeile:

Sobald das Dokument fertig geladen wurde …

```
$(document).ready( ... );
```

… führen Sie die in `ready()` befindliche anonyme Funktion aus …

```
$(document).ready( function() {
       ...
});
```

… die besagt: Bilden Sie aus allen <p>-Elementen mit Inhalt *'Zweiter Absatz'* innerhalb des `<div>`-Containers mit dem ID `box`, ein jQuery-Objekt:

```
$(document).ready( function() {
     $("#box p:contains('Zweiter Absatz')")
       ...
});
```

… und wenden Sie auf dieses die Funktion `addClass()` an, um den <p>-Containern die Klasse `green` hinzuzufügen (wir machen wieder einen Zeilenumbruch vor dem Punktoperator – das ist für das Listing praktisch):

```
$(document).ready( function() {
     $("#box p:contains('Zweiter Absatz')")
       .addClass("green");
});
```

Das war's. Sie brauchen sich nicht mit Schleifen und Bedingungen herumschlagen. Stattdessen navigieren Sie mittels des Selektors `"#box p:contains('Zweiter Absatz')"` **direkt** zum gewünschten Element, verkapseln es als jQuery-Objekt und verändern es.

Die gewünschte Veränderung geschieht durch den Aufruf von `addClass()`. Mit dieser Methode können Sie beliebige CSS-Klassen an beliebige HTML-Elemente binden.

Das jQuery-Objekt als Knotenliste

Das jQuery-Objekt wurde nun als **Wrapper** für ein Einzelobjekt eingeführt. Für dieses Objekt stehen alle jQuery-Methoden zur Verfügung. Ein *Einzelobjekt* war es aber schlicht deshalb, weil der an `$()` übergebene Ausdruck lediglich *ein Objekt* als Ergebnis hatte.

Wenn dies aber nicht so ist? Ganz einfach – jQuery macht aus *allen übergebenen Objekten* gemeinsam ein jQuery-Objekt, das die Eigenschaften einer »Liste« annimmt. In JavaScript spricht man von einer Knotenliste (*node-list*). In jQuery läuft dies darauf hinaus, dass für jedes Element der Liste alle jQuery-Optionen zur Verfügung stehen.

Selektieren wir jetzt alle Absätze innerhalb des Containers `#box`. Hierfür genügt es, den Filter `:contains('Zweiter Absatz')` wegzulassen:

```
$(document).ready(function() {
    $("#box p").addClass("green");
});
```

Das Ergebnis ist wenig spektakulär, vielleicht sogar als »intuitiv vorhersehbar« zu bezeichnen: Alle Textabsätze erhalten grüne Schrift und Hintergrundfarbe.

✔ jQuery arbeitet eine Knotenliste Item für Item ab, und zwar vollautomatisch mit einer impliziten Schleife.

11.2.3. Beispiel: Bindung eines Click-Events

Sie wollen als Nächstes erreichen, dass ein Absatz seine Schriftfarbe erst ändert, sobald Sie ihn anklicken. Gehen wir kurz durch, wie dies ohne die Hilfe von jQuery erfolgen könnte.

Variante A: Ohne jQuery

Um einem Absatz »von außen« den Click-Event zuzuweisen, müssen zunächst alle <p>-Container innerhalb des Bereichs `#box` gesammelt werden. Dies geschieht wie zuvor:

```
var elements = document.getElementById("box")
              .document.getElementsByTagName("p");
```

Über das Array mit den Textabsatzknoten läuft eine Schleife, die den Click-Event bindet:

```
for(var i =0, len=elements.length; i<len; i++) {
    // Fein. Jetzt den Click-Event binden. Aber wie?
   }
```

Um den Click-Event zu binden, gibt es verschiedene, untereinander unverträgliche Varianten: die offizielle und die für den Internet Explorer. Eine Fallunterscheidung ist möglich, aber umständlich. Der Rettungsanker findet sich beim guten alten **DOM Level 0** und der dort definierten `onclick`-Eigenschaft. Diese Vorgehensweise ist zwar nicht zeitgemäß, hat aber den Vorteil, dass sie browserübergreifend funktioniert:

```
window.onload = function() {
    var elements = document.getElementsByTagName("p");
    for(var i=0, len=meineP.length; i<len; i++) {
        elements[i].onclick = function() {
                this.className = "green";
                }
    }
}
```

Dies ist in seiner Kompaktheit durchaus vertretbar. Sollte es noch schöner gehen? Oder zumindest – kürzer?

Variante B: Mit jQuery

In jQuery verwenden Sie nicht eine DOM-Eigenschaft namens `onclick`, sondern eine jQuery-Methode die den (treffenden) Namen `click()` trägt. Mit ihrer Hilfe wird jedem Absatz *innerhalb* des Elements mit dem ID `#box` ein Click-Event-Handler hinzugefügt. Wieder reichen drei Zeilen:

```
$(document).ready( function() {
    $('#box p').click( function() {
        $(this).addClass("green");
    });
});
```

Sobald ein <p>-Element tatsächlich geklickt wird, tritt die anonyme Funktion im Inneren von `click()` in dessen Namen in Aktion.

Das *this* in ihrem Inneren bezieht sich, wie gehabt, auf den geklickten <p>-Container. Da dieses this (als herkömmlicher <p>-Container) mit `addClass()` nicht anfangen könnte, wrappen Sie es über die `$()`-Funktion wieder in ein jQuery-Objekt.

```
// die anonyme Funktion in click():
$('#box p').click( function() {
   // das Kontextobjekt this wird jQueryfiziert:
   $(this).addClass("green");
});
```

Hier zusammenfassend der Quellcode der gesamten Datei:

```
<html>
<head>
   <title>Listing 3</title>

  <!-- Zunächst die Styleangaben: -->
   <style type="text/css">
      p {
         cursor:pointer;
      }
      .green {
         color:#009933;
         background-color:#E2FFEC;
         cursor:default;
      }
   </style>

   <!-- Nun das Framework einbinden: -->
   <script type="text/javascript"
           src="../jquery/jquery.js"></script>

   <!-- ...und jetzt unseren eigenen Code: -->
   <script type="text/javascript">
      $(document).ready(function() {
        $("#box p").click(function() {
             $(this).addClass("green");
        });
      });
   </script>
</head>
<body>
   <div id="box">
      <p>Erster Absatz</p>
      <p>Zweiter Absatz</p>
      <p>Dritter Absatz</p>
   </div>
</body>
</html>
```

Schriftfarbe ändern mit Click-Event (mit jQuery)

12. Tools und Hilfsmittel

12.1. Zen-Coding

Das **Zen-Coding**-Plugin ermöglicht ein beschleunigtes Coding von Hand, indem es eine, an die CSS-Syntax angelehnte Kurzschreibweise expandiert.

✔ **Zen-Coding steht als Plugin zur Verfügung für:** SlickEdit, Sublime, TextMate, TopStyle, UltraEdit, Coda, Dreamweaver, Espresso, Notepad++, PSPad, SciTE, Aptana, Komodo Edit und andere.

Im einfachsten Fall wird ein **Elementname** zum **Elementcontainer** expandiert:

```
p
```

wird zu

```
<p></p>
```

Hierbei kann ein ID festgelegt werden:

```
p#beispiel
```

wird zu

```
<p id="beispiel"></p>
```

Dasselbe funktioniert für CSS-Klassen:

```
p.eineKlasse
```

wird zu

```
<p class="eineKlasse"></p>
```

Zwei aufeinanderfolgende Elemente werden durch + abgekürzt:

```
h1+p  wird zu
```

```
<h1></h1>
<p></p>
```

Auch Hierarchien können erstellt werden:

```
ul>li  wird zu
```

```
<ul>
   <li></li>
</ul>
```

Zusätzlich stehen Multiplikatoren zur Verfügung:

```
ul>li*3>a  wird zu
```

```
<ul>
   <li><a href=""></a></li>
   <li><a href=""></a></li>
   <li><a href=""></a></li>
</ul>
```

Mit vordefinierten Kombinatoren wie html:4t, html:4s, html:xt, html:xs oder html:5 können komplette Dokumentrümpfe geschrieben werden:

```
html:5   wird zu
```

```
<!DOCTYPE HTML>
<html lang="en-US">
<head>
   <meta charset="UTF-8">
   <title></title>
</head>
<body>

</body>
</html>
```

Download: http://code.google.com/p/zen-coding/

Online-Demo: http://zen-coding.ru/demo/

12.2. JavaScript-Debugging mit Firebug

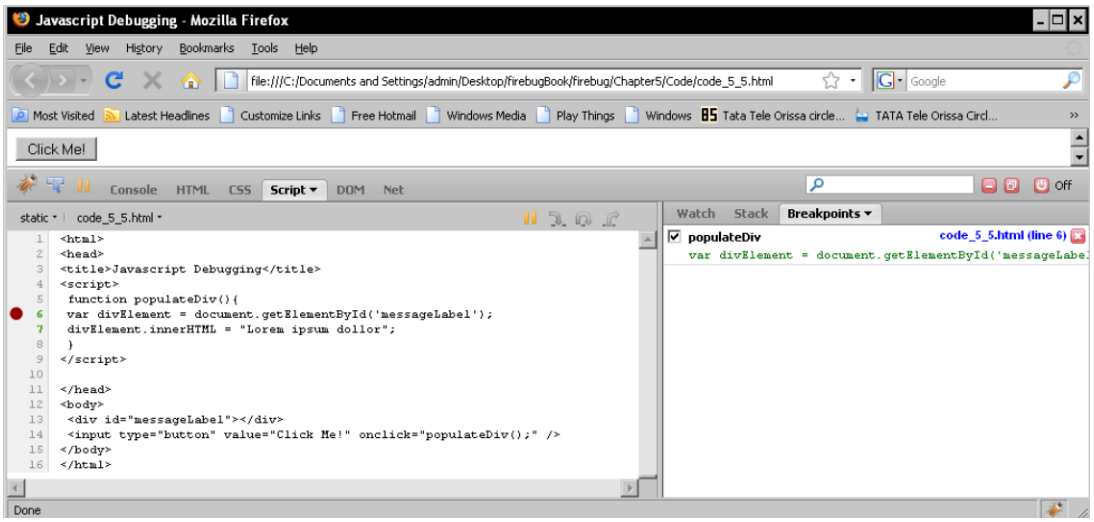
Die Browsererweiterung **Firebug** von *Joe Hewitt* ist das Instrument der Wahl, um recht umstandslos JavaScript zu debuggen. Dies soll nun an ein paar einfachen Beispielen gezeigt werden. Öffnen Sie das Dokument *debug01.html* in Firefox. Aktivieren Sie Firebug (Tastenkürzen **F12**).

12.2.1. Einen Breakpoint setzen

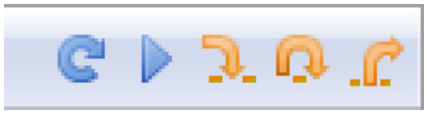
Wählen Sie die **Script-Konsole** von Firebug und setzen Sie in Zeile 6 einen Breakpunkt (im Code fett markiert).

```
<html>
<head>
<title>Javascript Debugging</title>
<script>
   function populateDiv(){
      var divElement =
       document.getElementById('messageLabel');
      divElement.innerHTML = "Lorem ipsum dolor...";
   }
</script>
</head>
<body>
   <div id="messageLabel"></div>
   <input type="button" value="Klick mich!"
onclick="populateDiv();" />
</body>
</html>
```

Klicken Sie hierfür in die Leiste links von der Codeansicht. Der Breakpoint ist im Quellcode des Panels mit einem roten Punkt am Zeilenanfang gekennzeichnet. Einen Breakpoint können Sie auch in der **Breakpointliste** im rechten Unterpanel des Script-Panels betrachten.



Starten Sie nun die Scriptausführung durch Klick auf den Button. Die Ausführung des Scriptes wird am Breakpoint angehalten. Das JavaScript kann nun debugged werden. Hierfür stehen fünf Optionen zur Verfügung, die über Buttons oberhalb des Codefeldes ausgewählt werden können.



- **Wiederholen (Rerun)**
  Wiederholt den aktuellen Stack und stoppt wieder am Breakpoint.

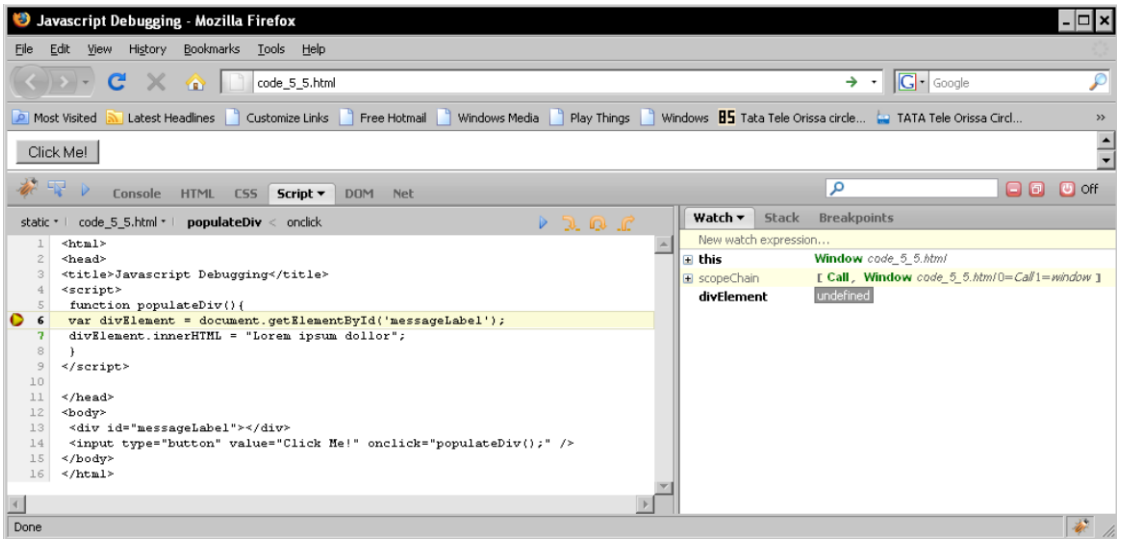
- **Continue** (F8)
  Setzt das Script nach dem Breakpoint fort bis zum nächsten Breakpoint oder zum Scriptende.

- **Step Into** (F11)
  Erlaubt das Eintreten in eine Funktion vom Ort ihres Aufrufs, um innerhalb des Funktionsblocks zu debuggen.

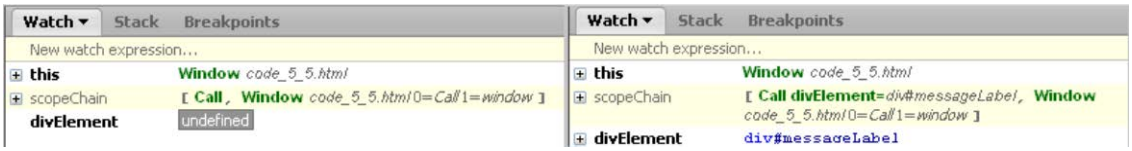
- **Step Over** (F10)
  Überspringt den Funktionsblock einer ausgeführten Funktion um das Debuggen nach der Funktion im selben Scope fortzusetzen.

- **Step Out**
  Verlässt den Scope einer aktuell debuggten Funktion und springt zurück zum Ort des Funktionsaufrufs (vorheriger Scope).

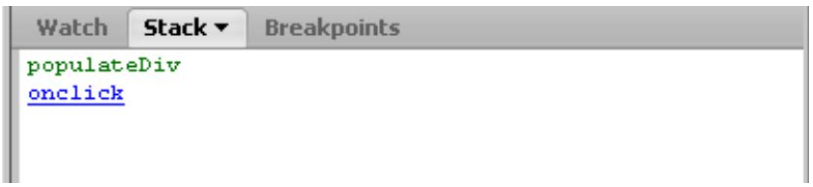
Klicken Sie auf „Step Over“ bzw. drücken Sie F10, um Zeile 6 auszuführen und zu Zeile 7 weiterzugehen. Sie sehen den aktuellen Wert der Variable `divElement` im **Watch-Panel** auf der rechten Seite.



Die Variable war vor der Ausführung von Zeile 6 *undefined* und enthält nun eine DOM-Referenz auf ein HTML Div-Element.



Im rechten Unterpanel finden Sie neben dem Watch-Panel das **Stack-Panel**, in dem Sie die Reihenfolge der Script-Ereignisse verfolgen können. Beachten Sie, dass das letzte Ereignis jeweils oben auf den Stack gelegt wird, die Liste also das letzte Ereignis oben zeigt. Hier ist dies der Aufruf der Funktion `populateDiv`, dem ein Click-Event vorausgeht:



Klicken Sie ein weiteres Mal auf **Step Over** um zur folgenden Zeile weiterzugehen. In diesem Fall existiert allerdings keine Folgezeile, daher ist das Script beendet.

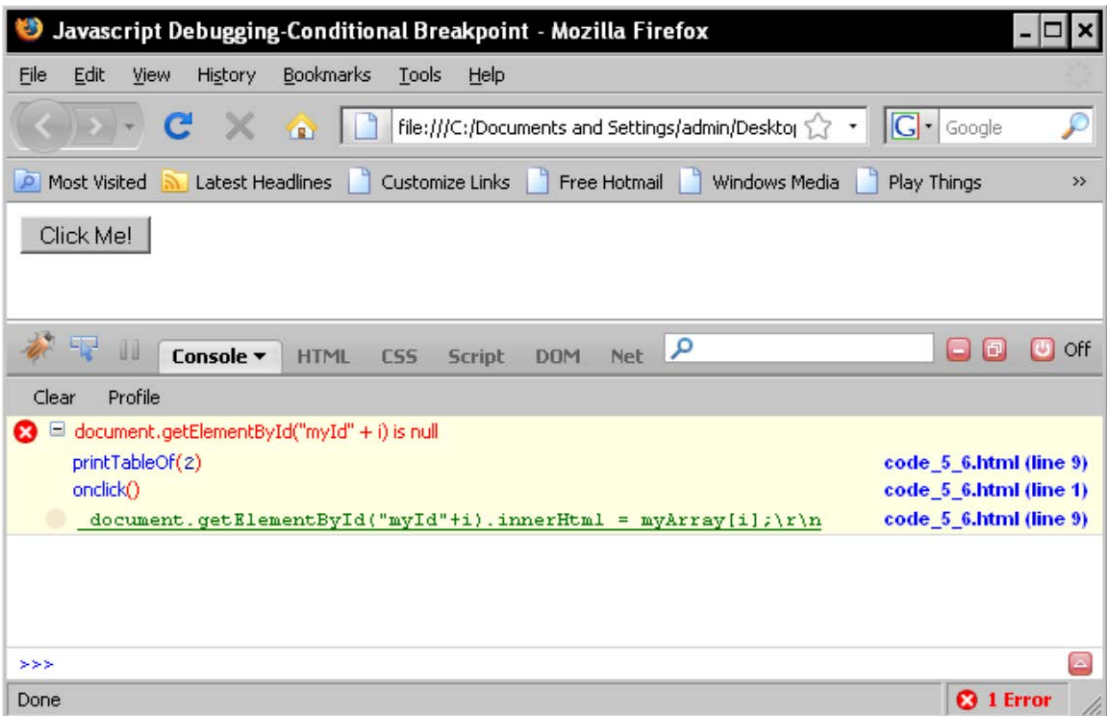
12.2.2. Bedingte Breakpoints

Häufig geschieht es, dass ein Breakpoint im Inneren einer Schleife gesetzt werden muss, weil genau hier ein Fehler auftritt. Es ist allerdings mühselig, diverse, möglicherweise tausende, Schleifendurchläufe durchzusteppen, bis der Fehlerfall tatsächlich eintritt.

Für genau diesen Fall bietet Firebug die Möglichkeit, an einen Breakpoint eine **Bedingung** zu knüpfen (*conditional breakpoint*). In diesem Fall wird der Breakpoint nur aktiv, wenn die gewählte Bedingung erfüllt ist. Als Beispieldatei verwenden wir *debug02.html*:

```
<html>
<head>
<title>Javascript Debugging-Conditional
Breakpoint</title>
<script>
   var myArray = new Array(9);
   function printTableOf(num){
      for(i = 1; i<=9; i++){
         myArray[i] = i*num;
         document.getElementById("myId"+i).innerHtml =
         myArray[i];
      }
   }
</script>
</head>
<body>
   <div id="myId1"></div>
   <div id="myId2"></div>
   <div id="myId3"></div>
   <div id="myId4"></div>
   <div id="myId5"></div>
   <input type="button" value="Klick mich!"
onclick="printTableOf(2);"/>
</body>
</html>
```

Folgendem Fehler sind wir auf der Spur – klicken Sie auf den Button in der Datei, um etwa dieses Bild in der Konsole zu sehen. Offensichtlich ist das Problem in Zeile 9 zu lokalisieren (im Code oben fett).



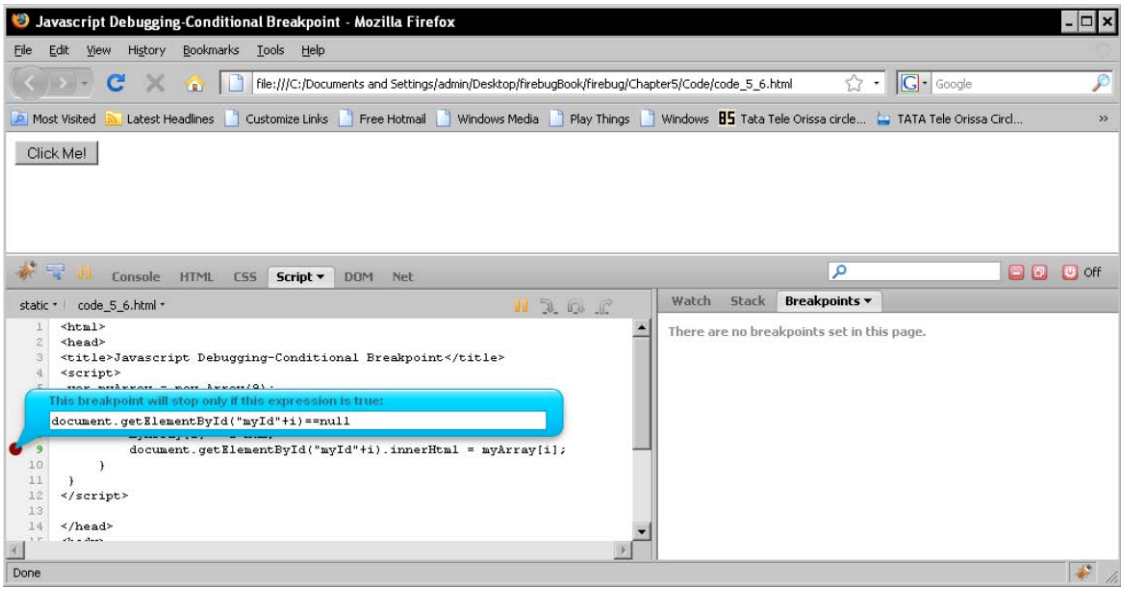
Wechseln Sie nun zum Script-Panel, laden Sie die Seite neu (diesmal noch nicht auf den Button klicken) und setzen Sie einen Breakpoint in Zeile 9. Klicken sie anschließend mit der rechten Maustaste auf den gesetzen Breakpoint. Firebug zeigt nun ein **Dialog-Tooltip**, in das eine Bedingung eingegeben werden kann. Sinnvollerweise wird die Bedingung so gewählt, dass sie die Fehlerursache berücksichtigt. Folgendes war in der Konsole zu sehen (rote Schriftfarbe):

```
document.getElementById("myId"+i) is null
```

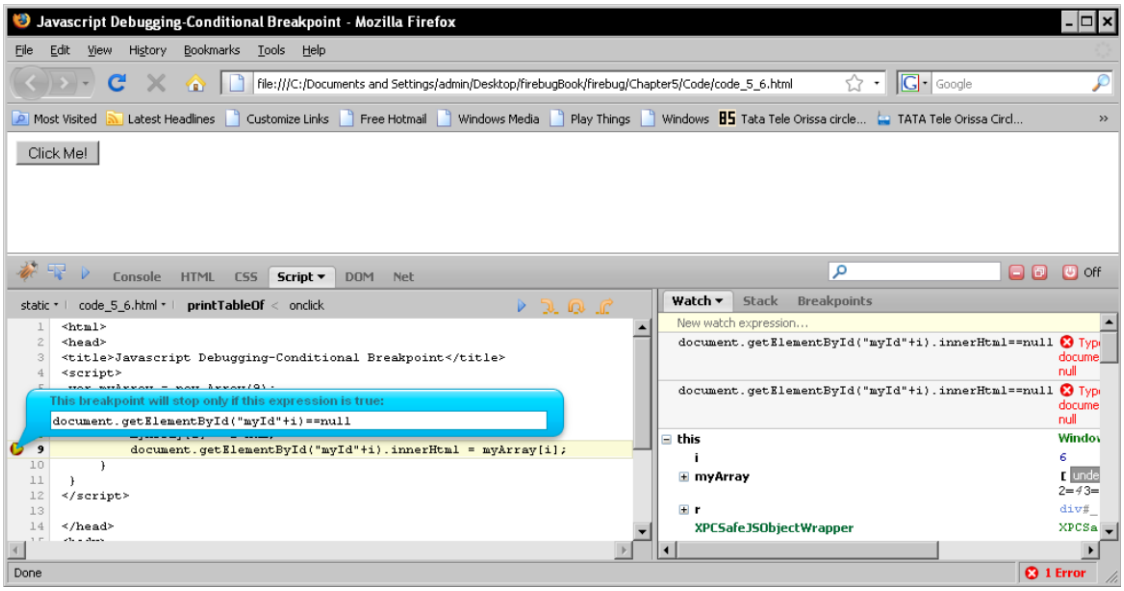
Folglich wählen wir genau dies als Bedingung. Geben Sie also den Fehler über den Dialog ein:

```
document.getElementById("myId"+i) == null
```

Der Breakpoint muss noch mit **Enter** bestätigt werden und ist nun einsatzbereit.

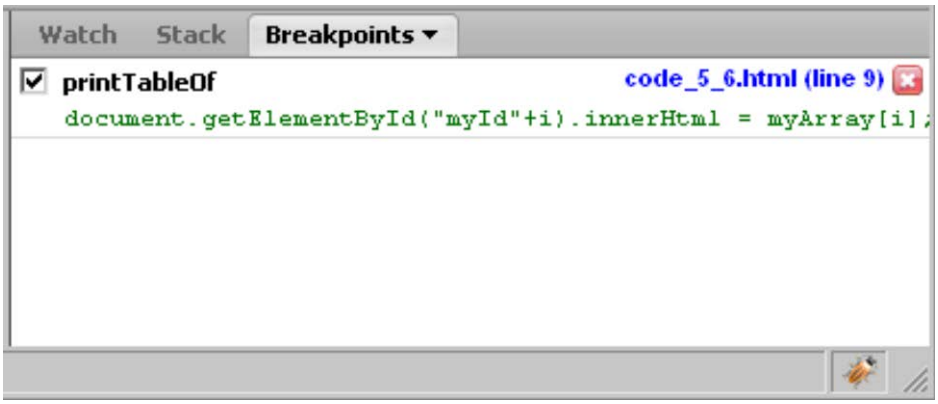


Jetzt ist der Moment, um das Script durch Klick auf den Button zu starten. Dank der Bedingung, stoppt der Breakpoint die Scriptausführung exakt bei eintreten des Fehlers. Was genau in der Zeile passiert, können Sie nun wieder im Watch-Panel betrachten. Die Variable `i` hat in diesem Augenblick den Wert `i=6`.



12.2.3. Breakpoints entfernen

Um einengesetzten Breakpoint zu entfernen, müssen Sie ihn in der Codeansicht lediglich anklicken. Alternativ wählen Sie seine Checkbox im Breakpoints-Panel im Seitenbereich ab.



13. Literatur

- **jQuery: Das Handbuch**, 3. Aufl.
  Frank Bongers, Maximilian Vollendorf
  (Galileo Computing, 2014)

- **JavaScript.** Das umfassende Referenzwerk, 6. Aufl.
  David Flanagan
  (O'Reilly, 2012)

- **JavaScript Patterns**
  Stoyan Stefanow
  (O'Reilly, 2010)

- **High Performance JavaScript**
  Nicholas C. Zakas
  (O'Reilly, 2010)

- **Secrets of the JavaScript Ninja**
  John Resig
  (Manning, 2012)

- **JavaScript: The Good Parts**
  Douglas Crockford
  (O'Reilly, 2008)

14. Onlineressourcen

14.1. Links zu JavaScript

JavaScript Dokumentation (engl.):

- developer.mozilla.org/en/docs/JavaScript

ECMA Spezifikation (engl.):

- www.ecma-international.org

Im Rahmen von HTML5 existieren eine Reihe von Spezifikationen, die JavaScript-APIs für verschiedene Funktionalitäten beschreiben. Einige davon seien hier aufgelistet:

- **Geolocation API Specification:**
  www.w3.org/TR/geolocation-API/
- **Web Storage Specification:**
  http://dev.w3.org/html5/webstorage/
- **The WebSocket API:**
  http://dev.w3.org/html5/websockets/
- **Web Workers:**
  http://dev.w3.org/html5/workers/