

Marc Jansen ▪ Till Adams

# Kurzeinführung in JavaScript

Freies Ergänzungsmaterial zu dem Buch:  
„OpenLayers –  
Webentwicklung mit dynamischen Karten  
und Geodaten

Open Source Press 2010

ISBN 978-3-937514-92-5



# Inhaltsverzeichnis

<b>1</b>	<b>Kurzeinführung in JavaScript</b>	<b>5</b>
1.1	Was ist JavaScript?	6
1.2	JavaScript, HTML und CSS	6
1.3	Basiswissen	8
1.4	Datentypen in JavaScript	11
1.4.1	Undefined	11
1.4.2	Null	12
1.4.3	String (Zeichenketten)	12
1.4.4	Number (Zahlen)	13
1.4.5	Boolean (Boolesche Variable)	14
1.5	Objekte in JavaScript	15
1.5.1	Array	15
1.5.2	Object	18
1.5.3	Function	20
1.5.4	Andere Objekte (z. B. Math, RegExp und Date)	22
1.6	Sprachelemente in JavaScript	23
1.6.1	Operatoren	23
1.6.2	Kontrollstrukturen	27
1.6.3	Fehlerbehandlung und Exceptions	30
1.7	BOM und DOM	31
1.7.1	Das Browser Object Model (BOM)	31
1.7.2	Das Document Object Model (DOM)	33
1.8	Events	35
1.9	Klassen, Vererbung, Konstruktoren und this	36
1.10	Praxistipps zum Entwickeln von JavaScript	39



# 1

# Kapitel

## Kurzeinführung in JavaScript

Dieses Kapitel war ursprünglich als Anhang zu unserem Buch „OpenLayers – Webentwicklung mit dynamischen Karten und Geodaten“ im Verlag Open Source Press geplant. Aufgrund seines allgemeinen Charakters haben wir uns aber entschieden, es als Ergänzung frei zur Verfügung zu stellen.



Abbildung 1.1:  
Till Adams, Marc  
Jansen  
„OpenLayers“  
Open Source Press,  
April 2010, ISBN:  
978-3-937514-92-5

Es soll in JavaScript einführen, erhebt jedoch keinen Anspruch auf Vollständigkeit – Ziel ist vielmehr, dass Sie sich im Quellcode der OpenLayers-Dateien schnell zurechtfinden.

### 1.1 Was ist JavaScript?

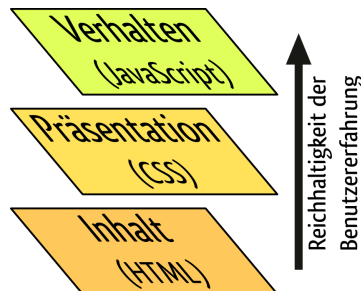
JavaScript ist eine Skriptsprache, die Ihnen vor allem im Zusammenhang mit Webbrowsern begegnet. JavaScript hat – bis auf die Namensähnlichkeit – nichts mit Java zu tun. JavaScript macht viele Funktionalitäten möglich, die man weithin mit „Web 2.0“ assoziiert, so ist z. B. die asynchrone Kommunikation zwischen Server und Client (AJAX) mit JavaScript realisiert.

JavaScript ist ein Superset bzw. Dialekt des Standards ECMAScript<sup>1</sup> und damit eine objektorientierte, dynamische und prototype-basierte Skriptsprache.

### 1.2 JavaScript, HTML und CSS

Anwendungen im WWW sind aus Einzelkomponenten in den drei Teilbereichen (bzw. Ebenen, engl. *Layer*) Inhalt, Präsentation und Verhalten aufgebaut (vgl. Abbildung 1.2). Im Englischen heißen diese Layer *content*, *presentation* und *behaviour*.<sup>2</sup>

Abbildung 1.2:  
Die drei Layer einer  
Webseite



#### Inhalt (*Content*)

Der Inhalt einer Webseite wird durch semantisches (X)HTML definiert. Dieser Teil zeichnet Inhalte nur aus, es sollte in der Auszeich-

<sup>1</sup> <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf>

<sup>2</sup> Vgl. insbesondere den Artikel „Simply JavaScript: The Three Layers of the Web“ von Kevin Yank: <http://articles.sitepoint.com/article/simply-javascript>, dieser Artikel enthält auch eine englischsprachige Version der Abbildung 1.2.

nung von Inhalten grundsätzlich keine direkte visuelle Repräsentation angestrebt werden. So sollten etwa nur tabellarische Daten mit einer Tabelle (HTML-Tag `<table>`) ausgezeichnet werden, Tabellen sollte nicht zu Designzwecken missbraucht werden.<sup>3</sup>

#### Präsentation (*Presentation*)

Das Styling (vornehmlich die visuelle Präsentation, aber z. B. auch die lautsprachliche Betonung bei einem Vorlesebrowser/Screenreader) übernimmt bei modernen Webanwendungen weitgehend CSS (Cascading Style Sheets). Über CSS lassen sich hierarchisch-strukturierte Anweisungen zur Darstellung von Elementen an den *User-Agent* (den Browser) übergeben. So kann man beispielsweise festlegen, dass alle Absätze (HTML-Tag `<p>`) in einem HTML-Dokument einen grauen Rahmen haben sollen, Absätze jedoch, die die Klasse (HTML-Attribut `class`) `important` haben, sollen zusätzlich in fetter Schrift dargestellt werden usw.

#### Verhalten (*Behaviour*)

Der Verhaltens-Layer (englisch *behavioral layer*) schließlich wird in Webanwendungen durch JavaScript umgesetzt. Dabei ergänzt und erweitert JavaScript die vom Browser standardmäßig ausgeführten Verhaltensweisen. Ein Link (HTML-Tag `<a>`) hat also beispielsweise ein Standardverhalten, das vom Browser festgelegt wird (beim Anklicken des Links wird die verlinkte Adresse im Browser geöffnet). JavaScript könnte diese Verhalten ergänzen, indem eine Frage wie „Möchten Sie die Seite verlassen?“ eingeblendet wird, die nur im Bestätigungsfall die Standardaktion des Browsers zulässt. Mit JavaScript sind die denkbaren Interaktionsmöglichkeiten des Benutzers mit einer Webseite also sehr vielfältig.

Angenommen, Sie möchten auf Ihrer Webseite einen Newsticker anbieten, der die Überschriften der neuesten zehn Newseinträge anzeigt und es dem User erlaubt, per Klick auf eine Überschrift zum tatsächlichen Newseintrag zu navigieren, so könnte eine Aufteilung in die drei o. g. Layer wie folgt aussehen:

- Der Inhalt (*Content*) könnte als Menge von Links (`<a>`, engl. *anchor*) in Form von Listenelementen (`<li>`, engl. *list item*) in einer ungeordneten Liste (`<ul>`, engl. *unordered list*) ausgezeichnet werden:

<sup>3</sup> Sollten Sie Tabellen zu Layoutzwecken verwenden wollen, empfiehlt sich zunächst die folgende Lektüre: <http://www.hotdesign.com/seybold/> und <http://shouldiuseablesforlayout.com/>.

```
<!-- HTML Fragment: -->
<ul>
  <li>
    <a href="newseintrag-1.html">
      Der Newseintrag Nummer 1 ist sehr lesenswert!
    </a>
  </li>
  <li>
    <a href="newseintrag-2.html">
      Der Newseintrag Nummer 2 aber auch!
    </a>
  </li>
  <!-- etc. -->
</ul>
```

- Per CSS könnten Sie dafür sorgen, dass Ihr HTML-Code visuell ansprechend ausgegeben wird: So sollten etwa die Aufzählungszeichen vor den `<li>`-Elementen unterdrückt werden und die Linkunterstreichung farblich dem Design der Seite entsprechen.
- Das JavaScript schließlich soll dafür sorgen, dass immer nur ein Listenelement gleichzeitig sichtbar ist. Nach einem festzulegenden Zeitintervall soll das aktuell dargestellte Element aus- und das nächste in der Liste eingeblendet werden. Natürlich soll ein Element nur dann ausgeblendet werden, wenn der User nicht mit der Maus auf einen Listeneintrag zeigt – er könnte ja auf den soeben angezeigten Eintrag klicken wollen.

Dieses Verhalten der Website würde die Benutzererfahrung deutlich erweitern. Dieses sogenannte *progressive enhancement* – also die schrittweise Erweiterung bestehender (Standard-)Funktionalität – ist generell eine empfehlenswerte Strategie für Ihre Webanwendungen. Wenn ein Browser ohne Support für CSS oder JavaScript Ihre Seite besucht, so sind die Inhalte doch immer noch erreichbar, wenn auch nicht so ansprechend präsentiert. Hätten Sie einen anderen Ansatz verfolgt (z. B. die Newsheader per AJAX abgefragt und anschließend in ihre Seite eingefügt) ist dies nicht immer gewährleistet.

### 1.3 Basiswissen

Um eine HTML-Seite mit JavaScript zu erweitern, muss der JavaScript-Code eingebunden werden, entweder über Code, der inline in den HTML-Code eingebettet ist oder über den Verweis auf eine externe JavaScript-Datei:

```
<html>
  <head>
    <title>Seitentitel</title>
```

```

<!-- JavaScript inline einbinden: -->
<script type="text/javascript">
    // Hier kann JavaScript-Code stehen,
    // der als solcher erkannt wird
</script>
<!-- JavaScript aus externer Datei einbinden: -->
<script type="text/javascript" src="pfad/zur/datei.js"></script>
</head>
<body>
</body>
</html>

```

Diese Varianten können Sie *nicht* mischen:

```

<!-- JavaScript darf nur inline ODER extern eingebunden werden! -->
<script type="text/javascript" src="pfad/oder/URL/zur/datei.js">
    // Hier darf KEIN weiterer JavaScript-Code stehen
</script>

```

JavaScript ist *case-sensitive*, die Groß- und Kleinschreibung ist von Belang. Eine Anweisung in JavaScript sollte grundsätzlich mit einem Semikolon beendet werden. *Whitespace* (Zeilenumbrüche, Leerstellen und Tabulatorenzeichen zwischen Sprachbestandteilen) sind i. d. R. irrelevant. JavaScript unterstützt sowohl einzeilige (eingeleitet durch //) als auch mehrzeilige Kommentare (umschlossen von /\* und \*/). Variablen sollten mit dem Begriff var deklariert werden, auch wenn dies nicht vorgeschrieben ist:

```

// Ein einzeiliger Kommentar
// Wir deklarieren eine Variable:
var foo = 'bar';
// ... syntaktisch gleichbedeutend mit
var baz = 'trump';
/*
  Ein
  mehrzeiliger
  Kommentar
*/

// Einige wichtige Ausnahmen:
// Achtung... die Anweisung
return
  (a + b);
// gibt - wg. des Zeilenumbruchs - NICHT
// das Ergebnis von (a + b) zurück!
// Hier ist whitespace also relevant

/*
  Mehrzeilige Kommentare dürfen NICHT
  geschachtelt werden... dieser Kommentar-
  Block ist also syntaktisch falsch...
*/

```

```
/*
... weil hier ein weiterer Blockkommentar steht
*/
*/
```

Häufig werden Sie Bezeichnungen selbst vergeben, etwa wenn Sie Variablen oder Funktionen erzeugen. Diese Bezeichnungen müssen dabei gewissen Regeln folgen:

- Bezeichnungen müssen mit einem Buchstaben (a-z und A-Z) oder einem erlaubten Sonderzeichen (s. u.) beginnen.
- Ansonsten dürfen Sie zusätzlich zu Buchstaben die Ziffern 0-9 verwenden.
- Der Variablenname muss aus einem Wort bestehen, Leerstellen sind nicht erlaubt.
- Erlaubte Sonderzeichen sind der Unterstrich `_` und das Dollarzeichen `$`.
- Eine Bezeichnung darf weder zu einem Schlüsselwort (wie z. B. `if` oder `while`) noch einem reservierten Wort<sup>4</sup> identisch sein.

Beispiele für gültige Variablenamen sind etwa:

```
var humpty;           // gültig
var Dumpty;          // gültig
var the_Crimson_King; // gültig
var Winston_Smith_1984; // gültig
var $__important;    // gültig
```

Ungültige Variablen sind z. B.:

```
var 1984_roman;      // ungültig: beginnt mit Ziffer
var test variable;  // ungültig, enthält Leerzeichen
var summe-oben;     // ungültig, enthält Bindestrich
var continue;       // ungültig, ist reserviertes Wort
```

Der Zuweisungsoperator bei Variablen ist das einfache Gleichheitszeichen (`=`):

```
// Eine Variable deklarieren (var)
// und einen Wert zuweisen:
// var variablenname = <neuer Wert>; z.B.
var foo = 'baz';
```

Der folgende Abschnitt beschreibt mögliche Datentypen in JavaScript.

<sup>4</sup> Eine Liste der reservierten Wörter und Schlüsselwörter in JavaScript finden Sie z. B. unter <http://de.selfhtml.org/javascript/sprache/reserviert.htm>.

## 1.4 Datentypen in JavaScript

Datentypen in JavaScript werden von den Inhalten der Variablen bestimmt, nicht durch explizite Deklaration einer Variable. Während z. B. in Java eine Variable grundsätzlich einen festen Datentyp hat, kann sich der Datentyp von Werten, die in einer Variablen in JavaScript gespeichert werden, ändern (dynamische Typisierung).

JavaScript kennt die folgenden elementaren Datentypen:

- Undefined
- Null
- String (Zeichenketten)
- Number (Zahlen)
- Boolean (Boolesche Variable)

Welchen Datentyp eine Variable enthält, kann man einfach – u. U. jedoch nicht immer verlässlich<sup>5</sup> – mit dem `typeof`-Operator testen:

```
var mixedVariable;
alert( typeof mixedVariable );    // 'undefined'
mixedVariable = 'Fred Feuerstein';
alert( typeof mixedVariable );    // 'string'
mixedVariable = 19;
alert( typeof mixedVariable );    // 'number'
```

### 1.4.1 Undefined

Der Wert `undefined` wird allen nicht initialisierten Variablen zugewiesen und auch zurückgegeben, wenn man auf nicht vorhandene Eigenschaften von Objekten (vgl. Kapitel 1.5.2) zugreifen möchte.

```
// Variable nur deklarieren, keinen Wert zuweisen
var deklarierte_variable;
alert( deklarierte_variable );
// Ein Objekt erzeugen (s.u.)
var andere_variable =
alert( andere_variable.fehlendeEigenschaft )
```

<sup>5</sup> Insbesondere wenn ein Array oder `null` in einer Variablen vorgehalten wird: Der `typeof`-Operator gibt in solchen Fällen `object` zurück. Wenn Objekte explizit per Konstruktor-Funktion und nicht literal erzeugt wurden – also z. B. mit `var str = new String('Text')` anstelle von `var str = 'Text'` geschrieben wird, gibt `typeof` die Zeichenkette `object` zurück.

### 1.4.2 Null

Das Objekt Null hat genau eine Instanz, nämlich `null`. Dieser Datentyp wird verwendet, wenn eine Variable deklariert und initialisiert wurde, ihr Zustand jedoch leer sein soll.

```
// deklarieren und initialisieren einer Variable:  
var wichtigeVariable = 1234;  
// arbeite mit der Variable  
...  
// setze den Wert der Variablen auf null (leer)  
wichtigeVariable = null;  
alert( wichtigeVariable );
```

Verwirrenderweise gibt der `typeof`-Operator bei `null` als Datentyp `object` zurück:

```
var platzhalter = null;  
alert( typeof platzhalter ); // object!
```

### 1.4.3 String (Zeichenketten)

Eine Stringvariable dient dazu, Zeichenketten zu speichern, die Sie in der Regel über die literale Schreibweise erzeugen:

```
// Deklarieren und belegen einer Zeichenkettenvariable  
var zeichenkette01 = 'Eine beliebige Zeichenkette';  
// einfache oder doppelte Anführungszeichen können verwendet werden  
var zeichenkette02 = "Eine andere Zeichenkette";
```

Wenn Sie Anführungszeichen in Zeichenkettenvariablen speichern möchten, müssen sie sich von den Begrenzern des String-Objektliterals unterscheiden oder *escaped* werden:

```
// Das einfache Anführungszeichen ' kann innerhalb  
// von doppelten Anführungszeichen " verwendet werden  
// (umgekehrt ebenfalls)  
var testString1 = 'Er sagte: "Wir schaffen es!"';  
var testString2 = "Sie antwortete: 'Na, wenn Du es sagst.'";  
// Alternativ können die Anführungszeichen mit  
// dem Backslash \ escaped werden:  
var testString3 = 'Er: \'Also, worauf warten wir?\'';
```

Zeichenketten in JavaScript sind Objekte, d. h. sie verfügen über Eigenschaften und Methoden.

```

// Eine Eigenschaft auslesen
var meinString = 'OpenLayers';
// Gibt die Anzahl an Zeichen (10) zurück
alert( meinString.length );

// Eine Methode des Objekts verwenden;
// wandle jedes Zeichen in einen Großbuchstaben um
alert( meinString.toUpperCase() );
// Gibt die 'OPENLAYERS' zurück

```

Eine Auflistung aller Attribute und Methoden des String-Objekts finden Sie z. B. auf den Seiten des W3C.<sup>6</sup>

### 1.4.4 Number (Zahlen)

Es gibt in JavaScript einen Datentypen, der numerische Informationen repräsentiert: `Number`. Es gibt keine separaten Datentypen für Ganzzahlen (häufig `integer` in anderen Sprachen) oder Gleitkommazahlen (`float`, `double` etc. in anderen Sprachen).

Objekte des Datentyps `Number` erzeugen Sie wie folgt:

```

// Eine Zahl (Number) definieren
// (Keine Anführungszeichen, Dezimaltrenner ist der Punkt)
var zahl01 = 10.2;
// ohne fraktionales Part:
var zahl02 = 519;
// wissenschaftliche Notation
var zahl03 = 3.934e3 // <=> 3.934 * 10 * 10 * 10 = 3934

```

Instanzen des `Number`-Objekts haben keine Eigenschaften, jedoch einige Methoden wie etwa `toFixed()` und `toString()`:

```

// Die Methode toFixed rundet eine Zahl kaufmännisch
// auf die angegebenen Nachkommastellen
var zahl04 = 10.256789;
alert( zahl04.toFixed(3) ); // gibt 10.257 aus

// Die Methode toString konvertiert eine Zahl zu
// einer Zeichenkette, der optionale Parameter
// bestimmt die Basis des Zahlensystems (default 10)
var zahl05 = 1024;
alert( zahl05.toString() ); // gibt 1024 als String aus
alert( zahl05.toString(10) ); // dito, explizite Basis
alert( zahl05.toString(8) ); // gibt 2000 als String aus
alert( zahl05.toString(16) ); // gibt 400 als String aus
alert( zahl05.toString(2) ); // gibt 10000000000 als String aus

```

<sup>6</sup> [http://www.w3schools.com/jsref/jsref\\_obj\\_string.asp](http://www.w3schools.com/jsref/jsref_obj_string.asp)

Eine Auflistung aller Attribute und Methoden des Number-Objektes finden Sie wieder auf den Seiten des W3C.<sup>7</sup>

### 1.4.5 Boolean (Boolesche Variable)

Boolesche Variablen in JavaScript sind `true` und `false`; sie werden in logischen Vergleichen von JavaScript verwendet und zurückgegeben und entsprechen den Begriffen *Wahr* und *Unwahr*:

```
var isValidEmail = false;
var usernameStartsWithJ = true;
```

Neben booleschen Werten gibt es eine Reihe von Werten, die als *Wahr*-ähnlich (englisch *truthy*) bzw. *Unwahr*-ähnlich (englisch *falsy*) ausgewertet werden:

```
var testVariable;
testVariable = 1;
if (testVariable) {
    alert( '1 ist truthy' );
}

testVariable = "0";
if (testVariable) {
    alert( 'Jede nicht leere Zeichenkette ist truthy (auch "0")' );
}

testVariable = {};
if (testVariable) {
    alert( 'Jedes Objekt (s.u.) ist truthy, auch ein leeres' );
}

testVariable = [];
if (testVariable) {
    alert( 'Jedes Array (s.u.) ist truthy, auch ein leeres' );
}

testVariable = 0;
if( !testVariable ) {
    alert( '-1 ist falsy' );
}

testVariable = null;
if( !testVariable ) {
    alert( 'null ist falsy' );
}
```

<sup>7</sup> [http://www.w3schools.com/jsref/jsref\\_obj\\_number.asp](http://www.w3schools.com/jsref/jsref_obj_number.asp)

```
var testVariable2;
if( !testVariable2 ) {
    alert( 'undefined ist falsy' );
}
```

## 1.5 Objekte in JavaScript

Neben den o.g. Objekten bietet der JavaScript-Sprachkern noch weitere Objekte:

- Array
- Object
- Function

### 1.5.1 Array

Ein Array (auch *Liste* oder manchmal *Feld* genannt) ist eine Sammlung von – typischerweise gleichartigen – Einzelwerten in einem eigenen Variablentyp. So kann man beispielsweise Aufzählungen in der Regel hervorragend als Arrays darstellen:

```
// Meine Freunde als Einzelvariablen
var freund_1 = 'Peter';
var freund_2 = 'Paul';
var freund_3 = 'Mary';
// oder als Sammlung in der Form eines Arrays
var freunde = [
    'Peter',
    'Paul',
    'Mary'
];
```

Die gebräuchlichste Form der Erzeugung von Arrays ist die oben gezeigte literale Schreibweise []. Zwischen eckigen Klammern werden die einzelnen Elemente kommagetrennt aufgeführt. Nach dem letzten Element darf kein Komma stehen. Einige Beispiele:

```
var leeresArray = [];
var gefuelltesArray = [
    'element1',
    'element2'
];
// Alle Datentypen können Elemente von Arrays sein
var archivierteJahrgaenge = [
```

```
    1980,  
    1984,  
    1992,  
    2000  
  ];
```

JavaScript unterstützt nur numerische Arrays und nicht die aus anderen Sprachen wie etwa PHP bekannten assoziativen Arrays. Der Zugriff auf Arrayelemente erfolgt über deren Position (Index) im Array. Das erste Element eines Arrays hat den Index 0, das zweite den Index 1 etc.:

```
var personalpronomen = [  
  'ich',  
  'du',  
  'es,sie,er'  
  'wir',  
  'ihr',  
  'sie'  
];  
// Zugriff auf Arrayelemente  
alert( personalpronomen[3] ) // gibt 'wir' aus  
// Ändern einzelner Elemente  
personalpronomen[2] = 'er,sie,es'  
alert( personalpronomen[2] ) // gibt nun 'er,sie,es' aus
```

Arrays können als Elemente wiederum Arrays (und auch Objekte) beinhalten und man darf Arrays beliebig tief verschachteln:

```
// Die dritte Person im Singular  
// besteht aus mehreren Elementen,  
// daher kann man sie gut als weiteres  
// Array darstellen:  
var personalpronomen = [  
  'ich',  
  'du',  
  [  
    'er',  
    'sie',  
    'es'  
  ],  
  'wir',  
  'ihr',  
  'sie'  
];  
// Wenn Sie auf die feminine Form der  
// dritten Person im Singular zugreifen  
// wollen, schreiben Sie:  
alert( personalpronomen[2][1] ) // gibt 'sie' aus
```

Arrays haben einige Eigenschaften und Methoden, die Ihnen häufig begegnen werden:

```

var beatles = [
  'John',
  'Paul',
  'George',
  'Ringo'
];
// eine Eigenschaft
// Anzahl an Elementen im Array:
alert( beatles.length ) // gibt 4 aus
// Einige Methoden:
// Sortieren:
beatles.sort(); // George hat nun Index 0, John 1 etc.
// Reihenfolge umkehren:
beatles.reverse(); // Ringo hat nun Index 0, Paul 1 etc.
// Element anfügen
beatles.push( 'Yoko' ); // Index 4 wird von Yoko belegt
// Elemente als String verketteten
alert( beatles.join( ', ' ) ); // gibt 'Ringo, Paul, ... Yoko' aus

```

Die häufigsten Probleme, die bei der Arbeit mit JavaScript-Arrays auftreten, sind unserer Erfahrung nach folgende:

- Der Index beginnt bei 0, d. h. das erste Element hat den Index 0.
- Nach dem letzten Element darf kein Komma stehen.
- Die Eigenschaft `length` kann verwirrende Ergebnisse liefern, wenn explizite Indizes des Arrays gesetzt werden:

```

var arr = [
  'foo',
  'bar',
  'plonk'
];
arr[99] = 'fled';
alert( arr.length ); // gibt einhundert aus, obwohl
// die Indizes 3-98 undefined sind

```

- Um Elemente in einem Array zu löschen, kann der Operator `delete` verwendet werden, die Länge des Arrays ändert sich aber dabei *nicht*:

```

var arr = [
  'Dortmund',
  'Köln',
  'Schalke'
];
delete arr[2] // lösche 'Schalke'
alert( arr.length ); // gibt 3 aus, der Index
// 2 ist nun undefined

```

Häufig werden Sie über alle Elemente eines Arrays iterieren wollen, etwa wenn Sie nacheinander die Teile des Arrays an eine Funktion übergeben. Am einfachsten verwenden Sie hierzu die bereits bekannte Eigenschaft `length` und eine `for`-Schleife (vgl. Kapitel 1.6.2):

```
var familieYoung = [
  'Angus',
  'Malcolm'
];
// Fange mit Index 0 an und erhöhe den
// index um 1 bis das alle Elemente verwendet
// wurden:
for(var i = 0; i < familieYoung.length; i++) {
  // gibt zunächst 'Angus', dann 'Malcolm' aus
  alert( familieYoung[ i ] );
}
```

### 1.5.2 Object

In einem JavaScript-Objekt können beliebig verschachtelte Schlüssel-Wert-Paare abgelegt werden. Selbst die abstraktesten Phänomene lassen sich gut mit Objekten modellieren.

```
// Ein neues Objekt in literaler
// Schreibweise erzeugen:
var unserObjekt = {};
// Bei der Erzeugung eines Objektes
// können die Werte bereits mit angegeben werden:
var australia = {
  capitol : 'Canberra',
  population : 21360000,
  area: '7.692.030 squarekilometer',
  isContinent: true
};
```

Schlüssel und Wert (also im Beispiel `capitol` bzw. `'Canberra'`) werden durch einen Doppelpunkt (`:`) voneinander getrennt. Der Schlüssel ist eine Bezeichnung, für die die auf Seite 10 genannten Regeln gelten. Der Schlüssel kann auch als String (also von einfachen oder doppelten Anführungszeichen umfasst) angegeben werden. In diesem Fall sind auch Schlüssel erlaubt, die andere Sonderzeichen (wie z.B. einen Bindestrich oder ein Leerzeichen) enthalten. Auch ansonsten verbotene Schlüsselwörter von JavaScript können Sie dann verwenden. Die einzelnen Schlüssel-Wert-Paare werden mit Kommas voneinander getrennt – nach dem letzten Paar darf kein Komma stehen. Umschlossen werden die Paare von geschwungenen Klammern (also `{` und `}`).

Betrachten wir das erste Beispiel aus Kapitel 1.5.1 zu Arrays, dort haben wir eine Liste von Freunden angelegt. Diese Personen wurden nur durch ihren

Namen repräsentiert, für viele Fragestellung ist dies ausreichend. Im Folgenden werden wir einen unserer Freunde aber als Objekt repräsentieren:

```
var meinBesterFreund = {
  vorname : 'max',
  nachname : 'mustermann',
  "Fußballverein" : 'VfL HauMichBlau',
  alter : 35,
  hobbies : [
    'lesen',
    'Fußball gucken',
    'schwimmen',
    'gut essen'
  ],
  familie : {
    elke : {
      alter: 43,
      beziehung: 'Ehefrau'
    },
    'hanna-marie' : {
      alter : 7,
      beziehung : 'Tochter',
      hobbies : [
        'reiten',
        'malen'
      ]
    }
  }
};
```

Sie können auf zwei Arten auf die so abgelegten Daten zugreifen:

```
// ... über die bereits bekannte Syntax,
// wie man grundsätzlich auf Objekteigenschaften
// zugreift:
alert( meinBesterFreund.alter ); // gibt 35 aus

// ... oder über eine Syntax, die etwa der von
// PHP sehr nahe kommt:
alert( meinBesterFreund['Fußballverein'] );
// gibt 'VfL HauMichBlau' aus

// Die letztgenannte Syntax muss verwendet werden, wenn auf
// Eigenschaften mit Sonderzeichen etc. zugegriffen
// werden soll oder wenn die Schlüssel in einer
// anderen Variable gespeichert sind:

var maedchenname = 'hanna-marie';
alert( meinBesterFreund['familie'][ maedchenname ]['beziehung'] );
// Man kann beide Arten auch in einem Aufruf mischen:
alert( meinBesterFreund.familie[ maedchenname ].beziehung );
// beide geben 'Tochter' aus.
```

Um über die Elemente eines Objekts zu iterieren, können Sie die `for...in`-Schleife (vgl. Kapitel 1.6.2) verwenden:

```
var meinObject = {
  foo: true,
  bar: 'baz',
  birk: 'fled'
};
// Für jeden Durchgang wird die Variable schluesssel
// auf einen tatsächlichen Schlüssel des Objekts
// gesetzt, eine bestimmte Reihenfolge des Iterierens ist
// *nicht* vorhersehbar:
for( var schluesssel in meinObject ) {
  // gibt nacheinander true, 'baz' und 'fled' aus
  alert( meinObject[ schluesssel ] );
}
```

### 1.5.3 Function

In JavaScript sind Funktionen Objekte erster Ordnung (engl. *First-class objects*), d. h. Sie können z. B. zur Laufzeit erzeugt, in Variablen gespeichert und wie andere Objekte zwischen verschiedenen Elementen übergeben werden. Funktionen kapseln Teile einer Programmlogik und sind eine Voraussetzung für wiederverwendbaren und wartbaren Code. Funktionen kann man auf mehrere Arten erzeugen, im Folgenden werden wir zwei Varianten kennenlernen:

```
// In einer an andere Sprachen erinnernden Form:
function hallo() {
  alert( 'Guten Tag!' );
}

// speichern einer Funktion in einer Variable:
var tschuess = function() {
  alert( 'Auf Wiedersehen!' );
};
```

Funktionen die Sie erzeugt haben, rufen Sie wie andere JavaScript-Methoden auf:

```
// Rufe die Funktion 'hallo' auf:
hallo();
// anschließend die Funktion 'tschuess':
tschuess();
```

Funktionen haben immer einen Rückgabewert, entweder wird ein Ausdruck explizit per `return` zurückgegeben oder der Rückgabewert ist `undefined`.

Funktionen können mit Parametern aufgerufen werden:

```

var rueckgabeTest = 'Ich werde überschrieben';
rueckgabeTest = hallo();
alert( typeof rueckgabeTest ) // gibt undefined aus

// Parameter in einer Funktion:
function addTwoValues( valueA, valueB ) {
    return ( valueA + valueB );
}
alert( addTwoValues( 15, 4 ) ); // Ergebnis von 15 + 4 ==> 19

```

Funktionen können Sie schachteln, etwa um innerhalb einer Funktion Teilaufgaben erledigen zu lassen:

```

// Definiere die äußere Funktion:
function begruessung( valueA, valueB ) {
    var text = '';
    // definiere die innere Funktion:
    function setzteText() {
        text = 'Herzlich Willkommen! Treten Sie ein...';
    };
    // Rufe die innere Funktion auf...
    setzteText();
    // und begruesse anschließend:
    alert( text );
}
// Rufe die äußere Funktion auf:
begruessung();

```

Da Funktionen Objekte sind, kann man sie wie alle Variablen an andere Strukturen übergeben:

```

function func1() {
    alert( 'EINS: Ich komme von func1' );
};
function func2() {
    alert( 'ZWEI: Ich komme von func2' );
};

// Eine weitere Funktion, die eine beliebige
// übergebene Funktion ausführt:
function zwischenFunktion( aufzurufendeFunktion ) {
    aufzurufendeFunktion();
}

// Wir übergeben die Funktionsobjekte an die
// generische zwischenFunktion:
zwischenFunktion( func1 );
zwischenFunktion( func2 );

```

Solange die Funktionen (func1 und func2 im o.g. Beispiel) keine Argumente erwarten, können Sie sie wie angegeben aufrufen. Ansonsten kön-

nen Sie z. B. auf die Funktionsmethode `call()` bzw. `apply()` zurückgreifen:

```
function addiere( a, b ) {
    alert( a + b );
};
function multipliziere( a, b ) {
    alert( a * b );
};

var ersteZahl = 12;
var zweiteZahl = 5;

// Rufe die Funktion addiere auf und übergib
// den Kontext der Funktion (this) und die Parameter:
addiere.call( this, ersteZahl, zweiteZahl );
// Bei apply muss der Kontext und ein Array an Argumenten
// übergeben werden:
multipliziere.apply( this, [ersteZahl, zweiteZahl] );
```

Was `this` in diesem Fall ist und bedeutet, wird in Abschnitt 1.9 erläutert.

### 1.5.4 Andere Objekte (z. B. Math, RegExp und Date)

JavaScript bietet weitere Spezialobjekte, die etwa für Datumskalkulationen, reguläre Ausdrücke oder mathematische Berechnungen verwendet werden können. Alle diese Objekte haben Eigenschaften und Methoden. Einige Beispiele:

```
// Das Math-Objekt für mathematische Berechnungen --
// gib die Kreiszahl Pi (3.141...) aus:
alert( Math.PI );
// Eine Zufallszahl bestimmen:
alert( Math.random() );
// Eine Wurzel bestimmen
alert( Math.sqrt(17) );

// Das RegExp-Objekt für reguläre Ausdrücke
var regulaerer_ausdruck = new RegExp( 'matz' );
var zu_testender_string = 'Schmatzmo';
alert( regulaerer_ausdruck.test( zu_testender_string ) );
// gibt true zurueck, denn 'matz' taucht im String auf

// Reguläre Ausdrücke haben daneben eine literale Schreibweise:
var anderer_ausdruck = /^sch/i;
alert( regulaerer_ausdruck.test( zu_testender_string ) );
// gibt true zurueck, denn 'sch' taucht, wenn
// Groß-/Kleinschreibung irrelevant ist (i = Schalter),
// am Anfang des Strings auf (^ = Anker)
```

```
// Das Date-Objekt für Datumskalkulationen --  
// das heutige Datum:  
alert( new Date() );  
// gibt z.B. 'Sat Nov 28 2009 07:52:00 GMT+0100' aus
```

Unter <http://www.w3schools.com/js/default.asp> sind diese Objekte detaillierter beschrieben.

## 1.6 Sprachelemente in JavaScript

### 1.6.1 Operatoren

#### Mathematische Operatoren

Folgenden mathematischen Operatoren stellt JavaScript zur Verfügung:

```
// Addition  
alert( 12 + 7 ); // 19  
// Subtraktion  
alert( 23 - 4 ); // 19  
// Multiplikation  
alert( 20 * 3 ); // 60  
// Division  
alert( 45 / 3 ); // 15  
// Modulo (Rest der Ganzzahldivision)  
alert( 16 % 7 ); // 2
```

Die gezeigten Operatoren werden auch binäre Operatoren genannt, da sie Argumente zu beiden Seiten des Operators benötigen. Daneben existieren einige weitere (monadische oder ünäre) Operatoren, die nur ein Argument erfordern:

```
var aNumber = 12;  
// a um die Zahl 1 erhöhen (inkrementieren)  
aNumber++;  
alert( aNumber ); // aNumber is nun 13  
  
// a um die Zahl 1 verringern (dekrementieren)  
aNumber--;  
alert( aNumber ); // aNumber is nun wieder 12  
  
// ++ und -- können auch vorangestellt sein:  
++aNumber;  
--aNumber;
```

### Zuweisungsoperatoren

Den Zuweisungsoperator = haben Sie bereits in den vorangegangenen Beispielen gesehen.

```
// Weise der Variablen humpty den  
// String dumpty zu:  
var humpty = 'dumpty';
```

Weitere Zuweisungsoperatoren sind die nachfolgend aufgeführten, die neben der reinen Zuweisung zuvor eine mathematische Operation auf den bisherigen Wert der Variable anwenden:

```
// Weise der Variablen meineZahl  
// initial die Zahl (Number) 16 zu  
var meineZahl = 16;  
  
// Dividiere und weise zu:  
meineZahl /= 2;  
// meineZahl ist nun 8  
  
// Subtrahiere und weise zu:  
meineZahl -= 3;  
// meineZahl ist nun 5  
  
// Multipliziere und weise zu:  
meineZahl *= 3;  
// meineZahl ist nun 15  
  
// Addiere und weise zu:  
meineZahl += 4;  
// meineZahl ist nun 19  
  
// Modulo und weise zu:  
meineZahl %= 6;  
// meineZahl ist nun 1
```

### Vergleichsoperatoren

Auch den einfachsten Vergleichsoperator == haben Sie bereits mehrfach kennengelernt.

```
var andereZahl = 11;  
alert( andereZahl == 11 );
```

Es gibt daneben noch weitere Vergleichsoperatoren:

```
var fred = 21;  
var frieda = '21';
```

```
var franz = 25;
var fritz = 21;

// Einfacher Vergleich:
alert( fred == frieda ); // ist true

// Identitätsvergleich (gleich und gleicher Typ):
alert( fred === frieda ); // ist false

// Einfache Ungleichheit:
alert( fred != franz ) // ist true

// Nicht-identisch:
alert( fred !== fritz ) // ist false
```

Auch die Vergleiche „größer“, „größer oder gleich“, „kleiner“ und „kleiner oder gleich“ sind möglich:

```
var foo = 21;
var bar = 23;
var wos = 21.0;
var sab = 'A';
var cud = 'a';

// Kleiner
alert( foo < bar ); // ist true

// Kleiner oder gleich
alert( foo <= wos ); // ist true

// Größer (geht auch mit Strings)
alert( cud > sab ); // ist true

// Größer oder gleich (geht auch mit Strings)
alert( foo >= wos ); // ist true
```

## Stringoperatoren

Häufig werden Sie Strings aneinanderreihen müssen. Diese Verkettung oder Konkatination können Sie in JavaScript mit dem Operator +, den Sie ja bereits aus der Addition von Zahlen kennen, durchführen:

```
var start = 'Open';
var ende = 'Layers';
var zusammen = start + ende
alert( zusammen ); // Gibt 'OpenLayers' aus

// Auch einen kombinierten Verkettungs- und
// Zuweisungsoperator gibt es
```

```
zusammen += ' ist super!';  
alert( zusammen ); // Gibt 'OpenLayers ist super!' aus
```

### Boolesche Operatoren

In JavaScript können logische Ausdrücke durch `&&` (UND) sowie `||` ODER verknüpft werden, außerdem kann ein boolescher Ausdruck mittels des Operators `!` negiert werden (logisches NOT):

```
var variableOne = true;  
var variableTwo = false;  
  
var andTest = variableOne && variableOne;  
// andTest ist false  
  
var orTest = variableOne || variableOne;  
// orTest ist true  
  
var notTest = !variableOne;  
// notTest ist false
```

Die Operatoren `&&` und `||` geben bei einem Vergleich dabei je nach Implementierung (sprich JavaScript-Engine oder Browser) von JavaScript nicht zwingend einen booleschen Wert zurück. Das ist insbesondere bei Vergleichen mit reinen Booleans (vgl. z. B. Kap. 1.4.5) von Belang:

```
var check1 = ''; // wird als false interpretiert  
var check2 = 15.3; // wird als true interpretiert  
  
// Gibt check1 zurück, wenn jenes als false interpretiert wird,  
// ansonsten wird check2 zurückgegeben:  
var andTestReturn = check1 && check2;  
alert( typeof andTestReturn ); // string, nämlich check1  
  
// Gibt check1 zurück, wenn jenes als true interpretiert wird,  
// ansonsten wird check2 zurückgegeben:  
var orTestReturn = check1 || check2;  
alert( orTestReturn ); // ist 15.3 also check2  
  
// Die Ausgabe von  
alert( '' && false );  
// ist also ggf. anders als die Ausgabe von  
alert( false && '' );  
// Analog gilt dies für  
alert( true || 'fc' );  
// ...und das umgekehrte  
alert( 'fc' || true );
```

## 1.6.2 Kontrollstrukturen

### Bedingungen: `if...else` und `switch`

Eine `if`-Bedingung sieht im einfachsten Fall aus wie folgt:

```
// <ausdruck> ist ein Platzhalter
if ( <ausdruck> ) {
    // Anweisungen für den Fall, dass
    // <ausdruck> als true ausgewertet wird
} else {
    // Anweisungen, die ansonsten ausgeführt werden
}
```

Der `else`-Zweig ist optional. `if...else` Konstrukte können geschachtelt werden. Der ternäre Operator (`<bedingung> ? ... : ...`) ist eine Kurzform von `if...else`, bei ihm ist der `else`-Zweig jedoch vorgeschrieben:

```
// Dieses Statement...
var ergebnis = bedingung ? ifFall : elseFall;
// ist also gleichbedeutend mit
if (bedingung) {
    var ergebnis = ifFall;
} else {
    var ergebnis = elseFall;
}
```

Wenn Sie den Wert einer Variablen gegen verschiedene Ausprägungen prüfen wollen, können Sie `if...else`-Strukturen natürlich verschachteln. Das kann sehr schnell unübersichtlich werden, so dass eventuell die `switch`-Fallunterscheidung verwendet werden sollte:

```
// Pseudocode
switch ( <wert_der_getestet_wird> ) {
    case <vergleichswert_1>:
        // Anweisungen für Fall 1
        break;
    case <vergleichswert_2>:
        // Anweisungen für Fall 2
        break;
    // usw.
    default:
        // Anweisung, falls keiner der Einzelfälle eintrat
        break;
}
```

Ein Beispiel: Wir wollen je nach dem Inhalt der Variable Lieblingsverein eine speziellen alert ausgeben:

```
var Lieblingsverein = '1. FC Köln';
// Zunächst mit geschachtelten if...else
if ( Lieblingsverein == 'Borussia Dortmund' ) {
    alert( 'Heja BVB!' );
} else {
    if ( Lieblingsverein == 'FC Bayern München' ) {
        alert( 'Nur der FCB!' );
    } else {
        if ( Lieblingsverein == '1. FC Köln' ) {
            alert( 'Geißbock-Power!' );
        } else {
            alert( 'Dein Verein wird nicht speziell begrüßt.' );
        }
    }
}

// Mit einer switch-Struktur sieht das
// Beispiel etwas einfach aus
switch( Lieblingsverein ) {
    case 'Borussia Dortmund':
        alert( 'Heja BVB!' );
        break;
    case 'FC Bayern München':
        alert( 'Nur der FCB!' );
        break;
    case '1. FC Köln':
        alert( 'Geißbock-Power!' );
        break;
    default:
        alert( 'Dein Verein wird nicht speziell begrüßt.' );
        break;
}
```

### Schleifen: for, for...in, while und do...while

Mit der for-Schleife können Sie Schleifen wie folgt abbilden:

```
// Pseudocode
for( <initialisierung>; <abbruchbedingung>; <variablenänderung> ) {
    // JavaScript-Code
}

// Um also die Zahlen von 1-10 auszugeben,
// kann die folgende Schleife verwendet werden:
for(var i = 1; i <= 10; i++) {
    alert( i );
}

// Der <initialisierung>-Bereich deklariert eine neue Variable
// namens i, der der Wert 0 zugewiesen wird.
// Die <abbruchbedingung> legt fest, dass die Schleife
```

```
// solange ausgeführt wird, wie i einen Wert hat, der kleiner
// oder gleich 10 ist. Die <variablenänderung> sorgt dafür,
// dass nach jedem Durchlauf in der Schleife i um eins erhöht wird
```

Um über Eigenschaften von Objekten zu iterieren, verwenden Sie das `for...in`-Statement:

```
var meinObjekt = {
  foo : 'bar',
  baz : 'bilk'
};

for ( objKey in meinObjekt ) {
  alert( objKey + ' ==> ' + meinObjekt[objKey] );
}
// Gibt zunächst 'foo ==> bar' und
// anschließend 'baz ==> bilk' aus
```

Neben der `for`-Ablaufstruktur bietet JavaScript noch kopf- bzw. fußgesteuerte Schleifen `while` und `do...while`:

```
// In Pseudocode
// Die kopfgesteuerte while-Schleife
while ( <bedingung> ) {
  // Anweisungen
}

// Und die fußgesteuerte do...while-Variante
do {
  //
} while ( <bedingung> )
```

Der Unterschied zwischen den beiden gezeigten Schleifen ist der Zeitpunkt der Überprüfung der `<bedingung>`: Bei einer `while`-Schleife erfolgt sie, bevor die JavaScript-Anweisungen zwischen den geschwungenen Klammern (`{` und `}`) ausgeführt werden – bei einer `do...while`-Schleife werden zunächst die Anweisungen ausgeführt und anschließend die `<bedingung>` geprüft. Eine `do...while`-Struktur stellt also sicher, dass der Code wenigstens ein Mal ausgeführt wird, unabhängig davon, ob die Anweisung jemals als `true` ausgewertet wurde:

```
var meinWert = 19;

// Die Bedingung ist nicht wahr,
// es wird nichts ausgegeben:
while ( meinWert > 20 ) {
  alert( 'while: ' + meinWert );
  meinWert = meinWert - 1;
}
```

```
}

// Die Bedingung ist nicht wahr,
// trotzdem wird zunächst ein alert ausgegeben:
do {
  alert( 'do...while: ' + meinWert );
  meinWert = meinWert - 1;
} while ( meinWert > 20 )
```

### 1.6.3 Fehlerbehandlung und Exceptions

Fehler in syntaktisch korrektem JavaScript-Code (also *Exceptions*) können in JavaScript abgefangen werden, um eine spezielle Fehlerbehandlung zu implementieren:

```
try {
  // Hier könnten fehlerhafte, aber syntaktisch
  // richtige Befehle stehen... vgl. Beispiel
  // weiter unten
} catch (e) {
  // ... die entsprechenden exceptions werden hier verarbeitet
  // in der Variable e steht ein Informationstext...
} finally {
  // Hier kommt man in jedem Falle hin.
}
```

Mittels der `throw`-Anweisung können Sie auch eigene *Exceptions* werfen:

```
throw 'Hier steht der Text der Exception';
```

Ein Beispiel:

```
var foo = {};

try {
  // Diese Eigenschaft existiert nicht,
  // es wird also eine exception ausgelöst...
  foo.humpty.dumpty = 'abc';
} catch (e) {
  // ... die hier abgefangen wird:
  alert('Exception abgefangen: ' + e);
} finally {
  // Hierhinein gelangt man immer:
  alert('der finally Bereich wird trotzdem erreicht');
  // Wir werfen nun eine eigene Exception,
  // die nicht abgefangen wird:
  throw 'Exception wurde von mir geworfen';
}
```

## 1.7 BOM und DOM

Neben dem Sprachkern von JavaScript gibt zwei weitere Themen, die in engem Zusammenhang mit JavaScript stehen: Das mittlerweile standardisierte *Document Object Model (DOM)*, das von Browserherstellern als Teil des breiter angelegten *Browser Object Model (BOM)* implementiert wird.

### 1.7.1 Das Browser Object Model (BOM)

Über die Objekte des *Browser Object Model (BOM)* – bzw. deren Eigenschaften und Methoden – können Sie ganz unterschiedliche Informationen wie die Breite und Höhe des Browserfensters, die Historie der besuchten Seiten oder die aktuelle URL in der Adressleiste des Browsers feststellen. Das BOM ist die hierarchische Sammlung verschiedener browserspezifischer Objekte. Besonders wichtig ist im BOM das `window`-Objekt, das den globalen Namensraum für Variablen darstellt und das Elternobjekt der anderen BOM-Objekte ist.

Das `window`-Objekt repräsentiert – vereinfacht ausgedrückt – das aktuelle Browserfenster, in dem die HTML-Datei dargestellt wird.

```
// Die uns bereits wohlbekannt Methode alert()
// ist ein Teil des window-Objekts...
alert( 'Meldung an den Benutzer ausgeben' );
// ... ist synonym zu
window.alert( 'Meldung an den Benutzer ausgeben' );
```

#### Das `window`-Objekt

Da `window` das globale Objekt in JavaScript ist, muss man technisch gesehen nicht zwingend `window` vor die entsprechenden Methoden und Variablen schreiben, da JavaScript am Ende der Interpretation einer Anweisung immer auch im globalen Namensraum sucht. Immer wenn ein Variable ohne `var` initialisiert wird, erzeugen Sie implizit eine globale Variable. Ihr Code sollte grundsätzlich nur wenige solche globale Variablen einführen, da theoretisch unerwünschte Kollisionen verschiedener Programmteile auftreten können.

```
// initialisiere eine explizite globale Variable
var globaleVariable = 1;

// Diese Funktion kann aus ihrem Kontext auf die
// gerade definierte Variable zugreifen:
function nutzeGlobaleVariable() {
    globaleVariable = 2;
}
```

```
nutzeGlobaleVariable();
// Nachdem die Funktion aufgerufen wurde, ist der Wert
// in der globalen Variablen geändert:
alert( globaleVariable );
// Es handelt sich bei der Variable um eine Eigenschaft
// des window-Objektes:
alert( window.globaleVariable );

// Diese Funktion erzeugt eine nur für die Funktion
// sichtbare Variable gleichen Namens und verändert
// daher nur die lokale, nicht die globale Variable:
function ignoriereGlobaleVariable() {
    var globaleVariable = 3;
}
ignoriereGlobaleVariable();
// Nach dem Funktionsaufruf ist der Wert der globalen
// Variable unverändert 2...
alert( window.globaleVariable );

// In der folgenden Funktion wird durch die Initialisierung
// ohne var implizit eine neue globale Variable erzeugt:
function erzeugeNeueGlobaleVariable() {
    neueGlobaleVariable = 4;
}
// Vor dem Aufruf ist window.neueGlobaleVariable == undefined
alert( window.neueGlobaleVariable );
erzeugeNeueGlobaleVariable();
// Nach dem Aufruf ist window.neueGlobaleVariable == 4
alert( window.neueGlobaleVariable );
```

### Das location-Objekt

Das Objekt `location` (oder `window.location`, s. o.) enthält Informationen zur aktuellen Adresse (URL) der Seite:

```
// Wenn wir uns auf der Seite
// http://example.com/index.html#beispiel
// befinden:
alert( location.hash ); // gibt '#beispiel' aus
alert( location.protocol ); // gibt 'http:' aus
// location ist ein Subobjekt von window:
alert( window.location.href );
// Gibt 'http://example.com/index.html#beispiel' aus
```

### Das screen-Objekt

Im `screen`-Objekt (`window.screen`) sind Informationen zum Browserfenster abgelegt. Die Werte sind erfahrungsgemäß leider nicht immer aussagekräftig:

```

alert( 'Ihr Bildschirm?: '
      + screen.width + ' x '
      + window.screen.height );

```

## Das history-Objekt

Mit dem `history`-Objekt kann man in der lokalen Browserhistorie navigieren:

```

// Zur zuvor besuchten Seite wechseln:
history.back();
// Zwei Seiten in der lokalen History zurückspringen:
history.go(-2);

```

### 1.7.2 Das Document Object Model (DOM)

Das *Document Object Model* (DOM) ist eine Konvention zur Repräsentation und Interaktion von Objekten in strukturierten Sprachen wie HTML und XML. Durch das DOM ist es möglich, Dokumente in den genannten Sprachen einfach abzubilden, abzufragen und zu verändern. Seit 1998 wurde das DOM vom W3-Konsortium (W3C) in verschiedenen Stufen standardisiert.<sup>8</sup> Sehr häufig werden Sie mit JavaScript das DOM benutzen, etwa wenn Sie HTML-Elemente oder deren Attribute in Ihrem Dokument auslesen oder wenn Sie ändernd in die Struktur einer Seite eingreifen.

Das DOM definiert für Elemente in der HTML-Seite zahlreiche Verwandtschaftsbeziehungen der einzelnen Elemente der Seite. Oft werden die Einzelbestandteile in einem sogenannten *document-tree* (Dokument-Baum) von Elementknoten (*nodes*) dargestellt. Schauen wir uns hierzu ein Beispiel an:

```

<ul>
  <li>
    erster Punkt
  </li>
  <li id='second-bullet'>
    zweiter Punkt
  </li>
  <li>
    dritter Punkt
    <ol>
      <li>
        Subpunkt
        <acronym title="Document Object Model">
          DOM
        </acronym>

```

<sup>8</sup> <http://www.w3.org/DOM/>

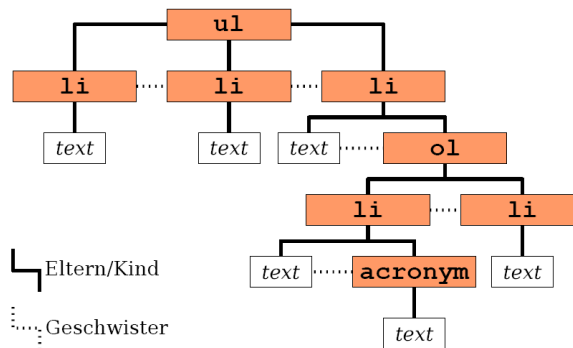
```

    </li>
    <li>
      Subpunkt Zwei
    </li>
  </ol>
</li>
</ul>

```

Dieses HTML-Fragment entspricht den folgenden hierarchischen DOM-Struktur:

Abbildung 1.3:  
OM-Struktur eines  
HTML-Fragments



Auszugsweise bestehen die folgenden Verwandtschaften:

- Alle `<li>`-Knoten sind Kinder (*children*) von `<ul>`.
- Dementsprechend kann `<ul>` als Elternknoten (*parentnode*) angesehen werden.
- Die `<li>`-Knoten haben die anderen `<li>` als Geschwister *siblings*.
- Die ersten beiden `<li>`-Knoten beinhalten nur Textteile; diese Bestandteile werden Textknoten (englisch *textnodes*) genannt. Achtung: Textknoten werden auch für Sonderzeichen wie den Zeilenumbruch (`\n`, bzw. `\r\n`) erzeugt.
- Das zweite `<li>` hat ein HTML-Attribut `id` mit dem Wert `second-bullet`. Das ist kein Subknoten im DOM, sondern eine Eigenschaft des `<li>`.

Vom `<ul>` kann man über die Verbindungen im Tree jeden anderen Knoten erreichen.

Die häufigsten DOM-Methoden (bereitgestellt vom `document`-Objekt), die Ihnen in JavaScript begegnen – und die beide der Selektion einzelner DOM-Knoten dienen – sind `document.getElementById( ... )` und `document.getElementsByTagName( ... )`:

```
// Beide nachfolgenden Methoden adressieren das zweite
// LI-Element;
// zunächst per id:
alert( document.getElementById( 'second-bullet' ) );
// nun über den tagname (mit dem Index 1 trifft man den zweiten node)
alert( (document.getElementsByTagName( 'LI' ))[1] );
```

Von einem einzelnen Knoten schließlich kann man auf seine Verwandtschaft zugreifen, diese kann z. B. so aussehen:

```
// Das erste Kind des OL-Tags ausgeben:
console.log( document.getElementsByTagName( 'OL' )[0].firstChild )
// Das kann z.B. etwa \n sein...
// In jedem Falle ist der mehrfach Elternknoten das UL-Element:
console.log(
  document.getElementsByTagName( 'OL' )[0] // das OL
  .firstChild // Textknoten oder LI
  .parentNode // ==> OL
  .parentNode // ==> LI
  .parentNode // ==> UL
);
```

Daneben bestehen z. B. zahlreiche Methoden, um neue Elemente zu erzeugen und bestehenden DOM-Strukturen hinzuzufügen, Attribute von Knoten zu verändern usw.<sup>9</sup>

## 1.8 Events

In Kapitel 1.2 unseres Buches haben Sie die drei Ebenen einer Webseite kennengelernt. Wie soll nun der durch JavaScript definierte Verhaltens-Layer erfahren, wann welches Verhalten vom User erwartet wird? Über vom User ausgelöste Events. Im Hintergrund einer Webseite wird vom Browser z. B. in regelmäßigen Abständen die Position der Maus in Relation zu Elementen auf der Seite festgestellt und gegebenenfalls an JavaScript weitergeleitet.

Die meisten Aktionen, die ein User auf einer Webseite ausführt – also etwa das Überfahren eines <div>-Elements mit der Maus, das Anklicken eines Buttons, die Eingabe eines Zeichens in einem Formularfeld, das Klicken und Halten der rechten Maustaste etc. – können mit JavaScript Funktionen belegt werden. Event-Handler, die verschiedene Ereignisse abfangen, sind somit ein wichtiges Bindeglied zwischen HTML und JavaScript.

Die Zuweisung eines JavaScript-Event-Handlers ist in verschiedenen Browsern zum Teil sehr unterschiedlich implementiert. Die meisten aktuellen

<sup>9</sup> Weitere Informationen finden Sie z. B. unter <http://www.howtcreate.co.uk/tutorials/javascript/domstructure>.

Browser verstehen die Definition eines solchen Handlers direkt im HTML-Quelltext über entsprechende HTML-Attribute. Wenn Sie z. B. bei einem Mausklick (Event `click`) auf einem `<div>` eine Funktion (in unserem Beispiel eine Funktion namens `func`) aufrufen möchten, können Sie dies über das HTML-Attribut `onclick` tun:

```
<div onclick="func();">  
    Event-Handler per HTML-Attribut  
</div>
```

In dieser Form haben Sie schon wenigstens einen weiteren Event in diesem Buch kennengelernt: Den `load`-Event des `<body>`, der ausgelöst wird, sobald die HTML-Seite und alle weiteren Ressourcen (wie etwa Bilder sowie CSS- und JavaScript-Dateien) vollständig geladen wurden und die Seite damit im Browser dargestellt werden kann. In vielen der im Buch gezeigten OpenLayers-Beispiele haben Sie eine Funktion (meist hieß sie `init`) aufgerufen, die Ihre OpenLayers-Applikation geladen hat:

```
<body onload="init();">  
    <!-- Die restlichen Seitenelemente -->  
</body>
```

Natürlich können Sie diese Eventhandler auch per JavaScript an Elemente binden. Das o. g. Beispiel sähe dann so aus:

```
// Das DOM Element per ID bekommen:  
var div = document.getElementById( 'id_des_divs' );  
div.onclick = func;
```

## 1.9 Klassen, Vererbung, Konstruktoren und `this`

JavaScript ist *per se* klassenfrei und kann ein aus der objektorientierten Welt bekanntes Konzept wie Vererbung (engl. *inheritance*) nur über Umwege implementieren. Es finden sich im Internet zahlreiche Versuche, diesem (vermeintlichen) Mangel entgegenzuwirken und eine „klassische Vererbung“ (wie man sie etwa von Java kennt) möglich zu machen.

JavaScript ist prototype-basiert, das bedeutet, dass Objekte von Objekten erben, indem diese als prototypisch für das aktuelle Objekt gelten. Es gibt hierfür in JavaScript jedoch keinen direkten Operator, so dass man sich um die entsprechende Funktionalität selber kümmern muss:

```
// Eine Funktion, die ein Auto-Objekt darstellt:  
function Car () {  
    // Kilometerstand initialisieren:
```

```

this.mileage = 0;
// Eine Methode um das Auto zu bewegen:
this.drive = function (km) {
    var newMileage = this.getMileage() + Math.abs(km);
    this.setMileage(newMileage);
};
// Eine Methode um den Kilometerstand auszulesen:
this.getMileage = function() {
    return this.mileage;
};
// Eine Methode um den Kilometerstand zu setzen:
this.setMileage = function(newMileage) {
    this.mileage = newMileage;
    return this.mileage === newMileage;
};
}

// Verwenden der Funktion
// Wichtig: Die Funktion muss mit new aufgerufen werden!
var auto1 = new Car();
var auto2 = new Car();
// Bewege beide Autos
auto1.drive(100);
auto2.drive(23);
auto2.drive(-2);
// Wie ist der aktuelle Kilometerstand?
alert('Kilometerstand Auto 1 (Car): ' + auto1.getMileage());
alert('Kilometerstand Auto 2 (Car): ' + auto2.getMileage());

```

Im Beispiel wird eine „Klasse“ Car definiert, die eine Autoobjekt repräsentiert: Das Auto hat einen Kilometerstand, man kann es bewegen und man kann den Kilometerstand setzen und auslesen.

Beachten Sie die Verwendung von `this` in der Funktion: `this` repräsentiert im Beispiel immer die aktuelle Instanz der „Klasse“. Sie müssen die Methode mittels `new` aufrufen, damit `this` einen korrekten Wert enthält. Immer wenn eine Funktion mit dem `new`-Operator aufgerufen wird, sprechen wir von der Funktion auch als Konstruktor-Funktion.

Wir wollen nun eine „Klasse“ Cabriolet entwerfen, die alle Eigenschaften und Methoden von Car erben soll und zusätzlich Informationen zum Zustand des Verdecks speichern kann. Hierzu entwickeln wir zunächst nur die neue Funktion Cabriolet, die die Unterschiede zu Car abbildet:

```

// Eine Funktion, die ein Cabrioletobjekt darstellt:
function Cabriolet() {
    // Das Verdeck soll initial geschlossen sein:
    this.verdeckOffen = false;
    // Eine Methode, um das Verdeck zu öffnen:
    this.oeffneVerdeck = function() {
        this.verdeckOffen = true;
    };
}

```

```
};  
// Eine Methode, um das Verdeck zu schließen:  
this.schliesseVerdeck = function() {  
    this.verdeckOffen = false;  
};  
// Eine Methode, die den Verdeckstatus zurückgibt:  
this.isVerdeckOffen = function() {  
    return this.verdeckOffen;  
}  
}
```

Die Instanzen von `Cabriolet` können sich derzeit jedoch noch nicht fortbewegen und haben keinen Kilometerstand. Wenn wir aber die Eigenschaft `prototype` des Funktionsobjektes `Cabriolet` mit einem fortan prototypisch zu verwendenden `Car`-Objekt überschreiben, haben alle Instanzen von `Cabriolet` alle Eigenschaften und Methoden von `Car`:

```
// Eine Funktion, die ein Cabriolet-Objekt darstellt:  
function Cabriolet() {  
    // implementieren der Differenzen zu Car wie oben  
}  
// Den Prototypen für Cabriolet-Objekte festlegen:  
Cabriolet.prototype = new Car();  
  
// Testen wir, ob man Cabriolets nun bewegen kann  
// und ob sie die anderen Eigenschaften/Methoden geerbt haben;  
// ein neues Cabriolet instanziiieren:  
var auto3 = new Cabriolet();  
// Einige Kilometer fahren...  
auto3.drive(350);  
// ...das Verdeck öffnen...  
auto3.oeffneVerdeck();  
// ...und den Status des Objektes abfragen:  
alert('Kilometerstand Auto 3 (Cabriolet): ' + auto3.getMileage()  
    + ', Verdeck offen? ' + auto3.isVerdeckOffen());
```

Wie erwartet, kann man nun auch auf die vom prototypisch angegebenen Objekt geerbten Methoden und Eigenschaften zurückgreifen. Auch mehrfache Vererbung ist möglich:

```
// Eine Funktion, die ein VolkswagenCabriolet-Objekt darstellt:  
function VolkswagenCabriolet() {  
    // Den Hersteller des wagens festlegen:  
    this.hersteller = 'Volkswagen';  
    // Den Hersteller des Wagens zurückgeben:  
    this.getHersteller = function() {  
        return this.hersteller;  
    };  
}  
// Ein Volkswagen-Cabriolet soll alle Eigenschaften eines
```

```

// Cabriolets (und damit von Car) erben:
VolkswagenCabriolet.prototype = new Cabriolet();

// Testen wir die multiple Vererbung:
// Erzeugen einer Instanz (new nicht vergessen!)
var auto4 = new VolkswagenCabriolet();
// Einige Kilometer fahren (Methode von Car geerbt):
auto4.drive(140);
// Das Verdeck oeffnen (Methode von Cabriolet geerbt):
auto4.oeffneVerdeck();
// Status des Objektes abfragen:
alert('Kilometerstand Auto 4 (VolkswagenCabriolet): '
      + auto4.getMileage()
      + ', Hersteller: ' + auto4.getHersteller() + '. '
      + ', Verdeck offen? ' + auto4.isVerdeckOffen());

```

Es gibt zahlreiche JavaScript-Frameworks, die die Klassenfunktionalität deutlich erweitert abbilden. Zu nennen ist hier z. B. Prototype JS, das großen Einfluss auf OpenLayers genommen hat.

Im Kapitel ?? haben Sie bereits gesehen, wie man OpenLayers um neue Funktionalität erweitert und dabei die Vorteile von objektorientiertem Code ausnutzen kann.

## 1.10 Praxistipps zum Entwickeln von JavaScript

Wenn Sie JavaScript Code entwickeln, werden Sie die Erweiterung Firebug<sup>10</sup> für den Mozilla Firefox kennen. . . und wenn nicht, sollten Sie sie kennenlernen. Auf der Webseite des Projekts stehen Hinweise für die Verwendung dieses praktischen Tools – man kann sich heute gar nicht mehr vorstellen, wie es war, JavaScript zu entwickeln, als Firebug noch nicht zur Verfügung stand.

Für den Internet Explorer – der Ihnen hier und da sicherlich einen Streich mit wunundersamen Interpretationen von JavaScript spielen wird – gibt es die von Microsoft herausgegebene Internet Explorer Developer Toolbar<sup>11</sup>, die eine gute Hilfe bei der Entwicklung, aber leider nicht mit dem oben angesprochenen Firebug vergleichbar ist.

Natürlich sollten Sie die Grundzüge der Sprache JavaScript kennen, um guten Code schreiben zu können. Für die Praxis aber ist es sehr empfehlenswert, auf ein JavaScript-Framework zu setzen. Diese Bibliotheken sind praxiserprobt, meist gut dokumentiert, und erleichtern die Entwicklung von Grundfunktionalität für Ihre Anwendungen, die dann auch in einer Vielzahl von Browsern funktioniert.

<sup>10</sup> <http://www.getfirebug.com>

<sup>11</sup> <http://www.microsoft.com/downloads/details.aspx?familyid=e59c3964-672d-4511-bb3e-2d5e1db91038&displaylang=en>

Hier als Auswahl die JavaScript-Bibliotheken:

- jQuery<sup>12</sup>,
- Prototype JS<sup>13</sup>,
- Ext JS<sup>14</sup>,
- Moo Tools<sup>15</sup>
- qooxdoo<sup>16</sup>.

Möchten Sie weiter in die Sprache JavaScript eintauchen, empfehlen wir die Werke von John Resig<sup>17</sup>, Douglas Crockford<sup>18</sup>, Dean Edwards<sup>19</sup> und Peter-Paul Koch<sup>20</sup> (PPK).

<sup>12</sup> <http://jquery.com/>  
<sup>13</sup> <http://www.prototypejs.org/>  
<sup>14</sup> <http://www.extjs.com/>  
<sup>15</sup> <http://mootools.net/>  
<sup>16</sup> <http://qooxdoo.org/>  
<sup>17</sup> <http://ejohn.org/>  
<sup>18</sup> <http://www.crockford.com/>  
<sup>19</sup> <http://dean.edwards.name/>  
<sup>20</sup> <http://www.quirksmode.org/>