O'Reillys Taschenbibliothek

4. Auflage

JavaScript

kurz & gut

David Flanagan

Übersetzung von Thomas Demmig, Jørgen W. Lang & Lars Schulten

O'REILLY®

4. AUFLAGE

JavaScript

kurz & gut

David Flanagan

Deutsche Übersetzung von
Th. Demmig, J. Lang, L. Schulten

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag GmbH & Co. KG
Balthasarstr. 81
50670 Köln
E-Mail: kommentar@oreilly.de

1. Auflage 1998
2. Auflage 2003
3. Auflage 2007
4. Auflage 2012

Die Originalausgabe erschien 2012 unter dem Titel *JavaScript Pocket Reference, Third Edition* bei O'Reilly Media, Inc.

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.de* abrufbar.

Übersetzung und deutsche Bearbeitung: Thomas Demmig, Mannheim, Jørgen W. Lang, Hatten & Lars Schulten, Köln
Lektorat: Imke Hirschmann, Köln
Korrektorat: Friederike Daenecke, Zülpich
Produktion: Karin Driesen, Köln
Umschlaggestaltung: Karen Montgomery, Sebastopol & Michael Oreal, Köln
Satz: Reemers Publishing Services GmbH, Krefeld, www.reemers.de
Belichtung, Druck und buchbinderische Verarbeitung: fgb freiburger graphische betriebe, www.fgb.de

ISBN: 978-3-86899-388-2

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Vorwort

JavaScript ist die Programmiersprache des Web. Die überwiegende Mehrheit moderner Websites setzt JavaScript ein. Außerdem enthalten alle modernen Browser – auf Desktop-Rechnern, Spielekonsolen, Tablets und Smartphones – einen JavaScript-Interpreter. Damit ist JavaScipt die am weitesten verbreitete Programmiersprache der Geschichte. JavaScript gehört zu den drei Technologien, die alle Web-Entwickler lernen müssen: HTML, um den Inhalt von Webseiten zu definieren, CSS, um festzulegen, wie die Inhalte dargestellt werden sollen, und JavaScript, um das Verhalten der Elemente einer Webseite zu steuern. Mit der Entwicklung von Node (*http://nodejs.org*) wird JavaScript außerdem auf Webservern immer wichtiger.

Dieses Buch ist ein Auszug aus *JavaScript – Das umfassende Referenzwerk, 6. Auflage*, in dem JavaScript noch wesentlich ausführlicher behandelt wird. Da das ursprüngliche Buch recht groß ist und etwas einschüchternd wirken kann, hoffe ich, dass dieses kürzere und kompaktere Buch für einige Leser nützlicher ist. Diese Kurzreferenz folgt grundsätzlich der Struktur seines großen Bruders: Die Kapitel 1 bis 9 beschäftigen sich mit dem Kern von JavaScript. Hier geht es um grundsätzliche Dinge wie die Syntax der Sprache, Typen, Werte, Variablen, Operatoren, Anweisungen. Danach beschäftigen wir uns mit JavaScript-Objekten, Arrays, Funktionen und Klassen. Diese Kapitel befassen sich mit der Sprache selbst. Sie sind nicht nur für die Verwendung von JavasScript in Webbrowsern, sondern auch beim Einsatz von Node auf der Serverseite relevant.

Damit man eine Sprache nutzen kann, muss sie entweder eine Plattform oder eine Standardbibliothek mit Funktionen besitzen, um zum Beispiel die Ein- und Ausgabe zu ermöglichen. Der Sprachkern von JavaScript legt eine minimale API für den Umgang mit Text, Arrays, Datumswerten und regulären Ausdrücken fest, besitzt aber keine Funktionalität für die Ein- und Ausgabe. Diese (und fortgeschrittenere Features wie Netzwerkzugriffe, Speicher und Grafik) liegen ganz in der Verantwortung der »Host-Umgebung«, in die JavaScript eingebettet ist. Am häufigsten kommt als Host ein Webbrowser zum Einsatz. Die Kapitel 1 bis 9 behandeln die in die Sprache integrierte Minimal-API. In den Kapiteln 10 bis 14 geht es um die Host-Umgebung des Webbrowsers. Außerdem wird hier erklärt, wie Sie »clientseitiges JavaScript« für die Erstellung dynamischer Webseiten und -applikationen verwenden können.

Die Anzahl der von Webbrowsern implementierten JavaScript-APIs ist in den letzten Jahren regelrecht explodiert. Daher ist es leider nicht möglich, sie in einem Buch dieser Größe zu behandeln. In den Kapiteln 10 bis 14 finden Sie Informationen zu den wichtigsten Elementen von clientseitigem JavaScript: Fenster, Dokumente, Elemente, Stile, Events, Netzwerke und Speicherung. Mit diesem Grundlagenwissen ist es einfach, die Verwendung weiterer clientseitiger APIs zu lernen. Weitere Informationen hierzu finden Sie ebenfalls im Buch *JavaScript – Das umfassende Referenzwerk, 6. Auflage* (oder auch in *Canvas – kurz & gut* und *jQuery Pocket Reference*, die ebenfalls Auszüge aus *JavaScript – Das umfassende Referenzwerk, 6. Auflage* sind).

Auch wenn die Programmierumgebung Node immer wichtiger wird, bietet diese Kurzreferenz leider nicht den nötigen Platz, um serverseitiges JavaScript zu behandeln. Sie finden weitere Informationen unter *http://nodejs.org*. Auch zu diesem Thema möchte ich Ihnen das Buch *JavaScript – Das umfassende Referenzwerk, 6. Auflage* empfehlen. Alternativ finden Sie auch online eine Reihe hervorragender JavaScript-Referenzen, wie beispielsweise das Mozilla Developer Network auf *http://developer.mozilla.org/*.

Typografische Konventionen

Ich habe die folgenden Formatierungskonventionen in diesem Buch verwendet:

Kursiv
: Wird zum Hervorheben genutzt und um die erste Verwendung eines Begriffs zu kennzeichnen. *Kursiv* wird ebenfalls für E-Mail-Adressen, URLs und Dateinamen verwendet.

`Nichtproportionalschrift`
: Wird in allen JavaScript-Codeblöcken- und HTML-Listings verwendet sowie allgemein für alles, was man beim Programmieren wörtlich eingeben würde.

`Nichtproportionalschrift kursiv`
: Wird für die Namen von Funktionsargumenten verwendet und allgemein als Platzhalter eingesetzt, um Elemente zu kennzeichnen, die in Ihren Programmen durch tatsächliche Werte ersetzt werden sollten.

Nutzung der Codebeispiele

Die Beispiele, die wir hier zeigen, dürfen Sie generell in Ihren Programmen und Dokumentationen verwenden. Sie brauchen uns nicht um Genehmigung zu bitten, sofern Sie nicht große Teile des Codes reproduzieren. Wenn Sie zum Beispiel ein Programm schreiben, das mehrere Codeabschnitte aus diesem Buch wiederverwendet, brauchen Sie nicht unser Einverständnis.

Eine Quellenangabe ist zwar nicht notwendig, aber dennoch willkommen. Bitte verwenden Sie in diesem Fall die Form: »Aus *JavaScript – kurz & gut*, 4. Auflage von David Flanagan (O'Reilly). Copyright 2012 David Flanagan, 978-3-86899-388-2.« Wenn Sie nicht wissen, ob Ihre Benutzung des Codes von dieser Genehmigung abgedeckt ist, nehmen Sie bitte unter der Adresse *permissions@oreilly.com* Kontakt mit uns auf.

Danksagungen

Vielen Dank an meinen Lektor Simon St. Laurent für den Vorschlag, *JavaScript – Das umfassende Referenzwerk, 6. Auflage* auf diese etwas besser handhabbare Größe zu verdichten. Ich danke außerdem den O'Reilly-Mitarbeitern aus der Produktion, die es immer wieder schaffen, meine Bücher richtig gut aussehen zu lassen.

KAPITEL 1

Die lexikalische Struktur

JavaScript-Programme werden im Unicode-Zeichensatz geschrieben. Unicode ist eine Obermenge von ASCII und Latin-1 und unterstützt praktisch alle geschriebenen Sprachen, die aktuell auf diesem Planeten genutzt werden.

JavaScript ist eine Sprache, die Groß-/Kleinschreibung unterstützt. Das heißt, dass Schlüsselwörter der Sprache, die Namen von Variablen und Funktionen und andere *Bezeichner* immer unter konsistenter Verwendung von Groß- und Kleinbuchstaben geschrieben werden müssen. Das Schlüsselwort `while` muss hingegen immer »while« geschrieben werden, nicht »While« oder »WHILE«. Gleichermaßen sind `online`, `Online`, `OnLine` und `ONLINE` vier verschiedene Variablennamen.

Kommentare

JavaScript unterstützt zwei Kommentarstile. Jeglicher Text, der zwischen der Zeichenfolge `//` und dem Ende einer Zeile steht, wird als Kommentar betrachtet und von JavaScript ignoriert. Jeglicher Text zwischen den Zeichenfolgen `/*` und `*/` wird ebenfalls als Kommentar betrachtet. Diese Kommentare können mehrere Zeilen lang sein, dürfen aber nicht geschachtelt werden. Die folgenden Codezeilen sind gültige JavaScript-Kommentare:

```
// Das ist ein einzeiliger Kommentar.
/* Das ist ebenfalls ein Kommentar */
// Und hier ist ein weiterer.
/*
```

```
 * Noch ein Kommentar,
 * der mehrere Zeilen lang ist.
 */
```

Bezeichner und reservierte Wörter

Ein *Bezeichner* ist einfach ein Name. In JavaScript werden Bezeichner genutzt, um Variablen und Funktionen zu benennen und um Marken für bestimmte Schleifen in JavaScript-Code anzugeben. Ein JavaScript-Bezeichner muss mit einem Buchstaben, einem Unterstrich (_) oder einem Dollarzeichen ($) beginnen. Nachfolgende Zeichen können Buchstaben, Ziffern, Unterstriche und Dollarzeichen sein.

JavaScript reserviert einige Bezeichner als Schlüsselwörter der Sprache. Sie dürfen diese Wörter in Ihren Programmen nicht als Bezeichner nutzen:

```
break     delete    function  return    typeof
case      do        if        switch    var
catch     else      in        this      void
continue  false     instanceof throw     while
debugger  finally   new       true      with
default   for       null      try
```

JavaScript reserviert außerdem einige Wörter, die von der Sprache aktuell nicht genutzt werden, in zukünftigen Versionen aber genutzt werden könnten. ECMAScript 5 reserviert die folgenden Wörter:

```
class  const  enum  export  extends  import  super
```

Zusätzlich sind die folgenden Wörter, die in gewöhnlichem JavaScript-Code zulässig sind, im »strengen Modus« (strict mode) reserviert:

```
implements  let      private    public  yield
interface   package  protected  static
```

Außerdem beschränkt der Strict-Modus die Verwendung der folgenden Bezeichner. Sie sind nicht vollständig reserviert, dürfen aber nicht als Namen für Variablen, Funktionen oder Parameter genutzt werden:

```
arguments   eval
```

ECMAScript 3 reservierte alle Schlüsselwörter der Programmiersprache Java. Obgleich dies in ECMAScript 5 gelockert wurde, sollten Sie folgende Wörter in Ihrem Code als Bezeichner vermeiden, wenn Sie vorhaben, Ihren Code in einer ECMAScript 3-konformen JavaScript-Implementierung laufen zu lassen:

```
abstract   double     goto         native     static
boolean    enum       implements   package    super
byte       export     import       private    synchronized
char       extends    int          protected  throws
class      final      interface    public     transient
const      float      long         short      volatile
```

Optionale Semikola

Wie viele andere Programmiersprachen nutzt JavaScript das Semikolon (;) um Anweisungen voneinander abzugrenzen (siehe Kapitel 4). Das ist wichtig, um die Bedeutung Ihres Codes zu verdeutlichen: Ohne ein Trennzeichen könnte es scheinen, als sei das Ende der einen Anweisung der Anfang der nächsten oder umgekehrt. In JavaScript können Sie das Semikolon zwischen zwei Anweisungen gewöhnlich weglassen, wenn diese Anweisungen auf eigenständigen Zeilen stehen. (Außerdem können Sie das Semikolon am Ende eines Programms weglassen oder dann, wenn das nächste Token im Programm eine schließende geschweifte Klammer } ist.) Viele JavaScript-Programmierer (und der Code in diesem Buch) nutzen Semikola, um explizit das Ende einer Anweisung zu markieren, auch an Stellen, an denen sie nicht verwendet werden müssten. Ein anderer Stil ist, Semikola immer wegzulassen, wenn das möglich ist, und sie nur in den wenigen Situationen zu verwenden, in denen sie erforderlich sind. Unabhängig davon, für welchen Stil Sie sich entscheiden, sollten Sie ein paar Details zu optionalen Semikola in JavaScript begriffen haben.

Betrachten Sie den folgenden Code. Da die beiden Anweisungen auf zwei separaten Zeilen stehen, könnte das erste Semikolon weggelassen werden:

```
a = 3;
b = 4;
```

Schreibt man die Anweisungen jedoch folgendermaßen, ist das erste Semikolon erforderlich:

```
a = 3; b = 4;
```

Beachten Sie, dass JavaScript nicht jeden Zeilenumbruch als ein Semikolon betrachtet: Normalerweise betrachtet es einen Zeilenumbruch nur dann als ein Semikolon, wenn sich der Code ohne dieses Semikolon nicht parsen ließe. Genauer gesagt: JavaScript interpretiert einen Zeilenumbruch als Semikolon, wenn der Umbruch entweder auf eines der Schlüsselwörter `return`, `break` bzw. `continue` folgt, wenn er vor den Operatoren ++ oder -- steht oder wenn das folgende Nicht-Leerzeichen nicht als Fortsetzung der gegenwärtigen Anweisung interpretiert werden kann.

Diese Regeln zum Abschluss von Anweisungen können zu einigen überraschenden Fällen führen. Folgender Code scheint aus zwei eigenständigen Anweisungen zu bestehen, die durch einen Zeilenumbruch getrennt werden:

```
var y = x + f
(a+b).toString()
```

Aber die Klammern zu Anfang der zweiten Codezeile können als ein Funktionsaufruf auf dem `f`-Element der ersten Zeile gelesen werden, und JavaScript interpretiert diesen Code folgendermaßen:

```
var y = x + f(a+b).toString();
```

KAPITEL 2

Typen, Werte und Variablen

Computer-Programme basieren auf der Manipulation von Werten wie der Zahl 3,14 oder dem Text »Hallo Welt«. Die Arten der Werte, die in einer Programmiersprache repräsentiert und manipuliert werden können, bezeichnet man als *Typen*. Wenn ein Programm einen Wert zur späteren Verwendung aufbewahren muss, weist es den Wert einer *Variablen* zu (oder »speichert« ihn in einer). Eine Variable definiert einen symbolischen Namen für einen Wert und ermöglicht es, den Wert über diesen Namen zu referenzieren.

Die Typen von JavaScript können in zwei Kategorien eingeteilt werden: *elementare Typen* und *Objekttypen*. Zu den elementaren Typen von JavaScript zählen Zahlen, Zeichenfolgen (die man als *Strings* bezeichnet) und boolesche Wahrheitswerte. Diese werden in den ersten Abschnitten dieses Kapitels behandelt. (Die Kapitel 5, 6 und 7 behandeln drei Typen von JavaScript-Objekten.)

JavaScript wandelt Werte großzügig von einem Typ in einen anderen Typ um. Übergeben Sie zum Beispiel einem Programm, das einen String erwartet, eine Zahl, wandelt es diese Zahl automatisch für Sie in einen String um. Geben Sie an einer Stelle, an der ein boolescher Wert erwartet wird, einen Wert an, der kein boolescher Wert ist, führt JavaScript die erforderliche Umwandlung durch. Die Regeln für Wertumwandlungen werden im Abschnitt »Typumwandlungen« auf Seite 16 erläutert.

JavaScript-Variablen sind *typlos*: Sie können einer Variable, der Sie zunächst einen Wert des einen Typs zugewiesen haben, später problemlos einen Wert eines anderen Typs zuweisen. Variablen

werden mit dem Schlüsselwort var *deklariert*. JavaScript arbeitet mit *lexikalischer Geltung*. Variablen, die nicht innerhalb einer Funktion deklariert werden, sind *globale Variablen*, die überall innerhalb eines JavaScript-Programms sichtbar sind. Variablen, die in einer Funktion deklariert werden, haben *Funktionsgeltung* und sind nur für Code sichtbar, der innerhalb der Funktion erscheint. Der Abschnitt »Variablendeklaration« auf Seite 21 geht detaillierter auf Variablen ein.

Zahlen

Anders als viele andere Sprachen macht JavaScript keinen Unterschied zwischen Ganzzahlen (oder Integern) und Gleitkommazahlen. In JavaScript werden alle Zahlen als Gleitkommawerte dargestellt. JavaScript repräsentiert Zahlen in dem 64-Bit-Gleitkommaformat, das im IEEE-Standard 754 definiert wird. Das heißt, es können Zahlen bis hinauf zu $\pm 1.7976931348623157 \times 10^{308}$ und bis hinab zu $\pm 5 \times 10^{-324}$ repräsentiert werden.

Das JavaScript-Zahlenformat ermöglicht es Ihnen, alle ganzen Zahlen zwischen –9007199254740992 (-2^{53}) und 9007199254740992 (2^{53}) exakt festzuhalten, die Grenzen eingeschlossen. Nutzen Sie größere ganzzahlige Werte, können Sie in den letzten Ziffern an Genauigkeit verlieren. Beachten Sie jedoch, dass in JavaScript bestimmte Operationen (wie die Indizierung von Arrays und die in Kapitel 3 beschriebenen Bit-Operationen) mit 32-Bit-Ganzzahlen durchgeführt werden.

Erscheint eine Zahl unmittelbar in einem JavaScript-Programm, bezeichnet man das als ein Zahlliteral. JavaScript unterstützt Zahlliterale in verschiedenen Formaten. Beachten Sie, dass jedem Zahlliteral ein Minuszeichen (-) vorangestellt werden kann, um die Zahl negativ zu machen.

Eine ganze Zahl zur Basis 10 wird in einem JavaScript-Programm als Folge von Ziffern geschrieben. Zum Beispiel:

```
0
1024
```

Neben Ganzzahlliteralen zur Basis 10 kennt JavaScript Hexadezimalliterale (zur Basis 16). Ein Hexadezimalliteral beginnt mit den Zeichen »0x« oder »0X«, auf die eine Reihe hexadezimaler Ziffern folgt. Hexadezimale Ziffern sind die arabischen Ziffern 0 bis 9 und die Buchstaben a (oder A) bis f (oder F), die die Werte von 10 bis 15 repräsentieren. Hier sind einige Beispiele für hexadezimale Ganzzahlliterale:

```
0xff  // 15*16 + 15 = 255 (Basis 10)
0xCAFE911
```

Gleitkommaliterale können einen Dezimaltrenner enthalten und nutzen die traditionelle Syntax für reelle Zahlen. Eine reelle Zahl wird mit einem ganzzahligen Teil, einem Punkt als Dezimaltrenner und dem Nachkommateil der Zahl dargestellt.

Gleitkommaliterale können auch in wissenschaftlicher Notation dargestellt werden. Bei dieser folgen auf eine reelle Zahl der Buchstabe e (oder E), ein optionales Plus- oder Minuszeichen und ein ganzzahliger Exponent. Eine in dieser Notation dargestellte Zahl entspricht der reellen Zahl mal 10 hoch dem Exponenten.

Kompakter formuliert, sieht diese Syntax so aus:

```
[Ziffern][.Ziffern][(E|e)[(+|-)]Ziffern]
```

Zum Beispiel:

```
3.14
6.02e23        // 6.02 × 10^23
1.4738223E-32  // 1.4738223 × 10^-32
```

JavaScript-Programme arbeiten bei Zahlen mit den arithmetischen Operatoren, die die Sprache bietet. Zu diesen zählen + für die Addition, - für die Subtraktion, * für die Multiplikation, / für die Division und % für die Modulodivision (den Rest nach der Division). Vollständige Angaben zu diesen und weiteren Operatoren finden Sie in Kapitel 3.

Neben diesen elementaren arithmetischen Operatoren unterstützt JavaScript komplexere mathematische Operationen über die Funktionen und Konstanten, die als Eigenschaften des `Math`-Objekts definiert sind:

```
Math.pow(2,53)  // => 9007199254740992: 2 hoch 53.
Math.round(.6)  // => 1.0: auf die nächste ganze Zahl
                // runden.
Math.ceil(.6)   // => 1.0: auf die nächste ganze Zahl
                // aufrunden.
Math.floor(.6)  // => 0.0: auf die nächste ganze Zahl
                // abrunden.
Math.abs(-5)    // => 5: Absolutwert.
Math.max(x,y,z) // Das größte der Argumente ermitteln.
Math.min(x,y,z) // Das kleinste der Argumente ermitteln.
Math.random()   // Pseudo-Zufallszahl, für deren Wert
                // 0 <= x < 1.0 gilt.
Math.PI         // π: Kreisumfang / Kreisdurchmesser.
Math.E          // e: Die Basis des natürlichen
                // Logarithmus.
Math.sqrt(3)    // Die Quadratwurzel von 3.
Math.pow(3,1/3)  // Die Kubikwurzel von 3.
Math.sin(0)     // Trigonometrie: auch Math.cos,
                // Math.atan usw.
Math.log(10)    // Natürlicher Logarithmus von 10.
Math.log(100)/Math.LN10 // Logarithmus zur Basis 10 von 100.
Math.log(512)/Math.LN2  // Logarithmus zur Basis 2 von 512.
Math.exp(3)             // Math.E hoch 3.
```

In JavaScript lösen arithmetische Operationen keine Fehler aus, wenn es zu Wertüberläufen, Wertunterläufen oder einer Division durch null kommt. Ist das Ergebnis einer numerischen Operation größer als die größte darstellbare Zahl (Überlauf), ist das Ergebnis ein spezieller Wert für unendlich, den JavaScript als `Infinity` ausgibt. Wird ein negativer Wert kleiner als die kleinste darstellbare negative Zahl, ist das Ergebnis minus unendlich und wird als `-Infinity` ausgegeben. Unendliche Werte verhalten sich, wie man erwarten würde: Additions-, Subtraktions-, Multiplikations- und Divisionsoperationen auf einem unendlichen Wert liefern als Ergebnis wieder einen unendlichen Wert (eventuell mit umgekehrtem Vorzeichen).

Die Division durch null ist in JavaScript kein Fehler: Sie liefert einfach Infinity oder -Infinity. Es gibt allerdings eine Ausnahme: Null geteilt durch null hat keinen wohldefinierten Wert. Das Ergebnis dieser Operation ist ein spezieller »Keine-Zahl«-Wert, der als `NaN` (kurz für Not-a-number) ausgegeben wird. `NaN` resultiert ebenfalls, wenn Sie versuchen, unendlich durch unendlich zu teilen, die Quadratwurzel einer negativen Zahl zu ermitteln, oder wenn Sie

arithmetische Operatoren auf nicht numerischen Operanden nutzen, die sich nicht in Zahlen umwandeln lassen.

JavaScript definiert die globalen Variablen `Infinity` und `NaN` zur Repräsentation von plus unendlich und »Keine-Zahl«.

NaN hat in JavaScript eine ungewöhnliche Eigenschaft: Dieser Wert ist keinem anderen Wert gleich, nicht einmal sich selbst. Das heißt, Sie können nicht `x == NaN` schreiben, um zu prüfen, ob der Wert einer Variablen `x` gleich `NaN` ist. Stattdessen sollten Sie `x != x` schreiben. Dieser Ausdruck ist nur dann wahr, wenn `x` gleich `NaN` ist. Alternativ können Sie die Funktion `isNaN()` nutzen. Diese liefert `true`, wenn ihr Argument `NaN` oder ein nicht-numerischer Wert wie ein String oder ein Objekt ist, der nicht in einen numerischen Wert umgewandelt werden kann. Die verwandte Funktion `isFinite()` liefert `true`, wenn ihr Argument eine Zahl ist, die nicht `NaN`, `Infinity` oder `-Infinity` ist.

Es gibt unendlich viele reelle Zahlen, von denen im JavaScript-Gleitkommaformat aber nur eine endliche Anzahl (18437736874454810627, um genau zu sein) exakt repräsentiert werden kann. Das heißt, dass die Darstellungen der Gleitkommazahlen, mit denen Sie in JavaScript arbeiten, häufig nur Näherungswerte für die eigentlichen Zahlen sind und es daher zu Rundungsfehlern kommen kann.

Text

Ein *String* ist eine unveränderliche Folge von 16-Bit-Werten, die üblicherweise Unicode-Zeichen darstellen – Strings sind der JavaScript-Typ zur Darstellung von Text. Die *Länge* eines Strings ist die Anzahl an 16-Bit-Werten, die er enthält. In JavaScript sind die Indizes für Strings (und Arrays) nullbasiert: Der erste 16-Bit-Wert befindet sich an Position 0, der zweite an Position 1 und so weiter. Der *leere String* ist der String der Länge 0. JavaScript hat keinen speziellen Typ zur Darstellung eines einzelnen String-Elements. Zur Darstellung eines einzelnen 16-Bit-Werts nutzen Sie einfach einen String der Länge 1.

Stringliterale

In ein JavaScript-Programm schließen Sie ein Stringliteral ein, indem Sie die Zeichen des Strings einfach in ein Paar einfacher oder doppelter Anführungszeichen setzen (' oder "). Doppelte Anführungszeichen können in Strings eingebettet werden, die von einfachen Anführungszeichen eingefasst werden. Einfache Anführungszeichen können umgekehrt in Strings eingebettet werden, die von doppelten Anführungszeichen eingefasst werden. Hier sind einige Beispiele für Stringliterale:

```
""  // Der leere String: Er enthält keine Zeichen.
'name="formular"'
"Hätten Sie nicht lieber ein O'Reilly-Buch?"
"Dieser String\nhat zwei Zeilen."
"π = 3.14"
```

Das Backslash-Zeichen (\) hat in JavaScript-Strings eine spezielle Bedeutung. Gemeinsam mit dem auf es folgenden Zeichen repräsentiert es ein Zeichen, das auf andere Weise nicht im String dargestellt werden kann. Beispielsweise ist `\n` eine *Escape-Sequenz*, die ein Zeilenumbruchzeichen vertritt.

Ein weiteres Beispiel ist die Escape-Sequenz `\'`, die ein einfaches Anführungszeichen (oder einen Apostroph) repräsentiert. Diese Escape-Sequenz ist hilfreich, wenn Sie einen Apostroph in ein Stringliteral einschließen müssen, das von einfachen Anführungszeichen eingefasst ist. Escape-Sequenzen nutzen Sie, um die übliche Interpretation eines Zeichens zu verhindern. Deswegen sagt man auch, dass der Backslash das nachfolgende Zeichen (wie hier das einfache Anführungszeichen) maskiert. Er sorgt dafür, dass der Apostroph nicht als vorzeitiger Abschluss des Strings verstanden wird:

```
'You\'re right, it can\'t be a quote'
```

Tabelle 2-1 führt die JavaScript-Escape-Sequenzen und die Zeichen auf, die sie jeweils repräsentieren. Zwei Escape-Sequenzen sind allgemein und können eingesetzt werden, um beliebige Zeichen anhand ihres Latin-1- oder Unicode-Zeichencodes in hexadezimaler Form anzugeben. Beispielsweise repräsentiert die Escape-Sequenz `\xA9` das Copyright-Symbol, dessen Latin-1-Kodierung hexadezimal

A9 ist. Die vergleichbare Escape-Sequenz \u repräsentiert ein beliebiges Unicode-Zeichen, das mit vier hexadezimalen Ziffern angegeben wird. \u03c0 steht beispielsweise für das Zeichen π.

Tabelle 2-1: JavaScript-Escape-Sequenzen

Sequenz	Zeichen
\0	Das NUL-Zeichen (\u0000)
\b	Backspace (\u0008)
\t	Horizontaler Tabulator (\u0009)
\n	Zeilenumbruch (\u000A)
\v	Vertikaler Tabulator (\u000B)
\f	Seitenvorschub (\u000C)
\r	Wagenrücklauf (\u000D)
\"	Doppeltes Anführungszeichen (\u0022)
\'	Apostroph oder einfaches Anführungszeichen (\u0027)
\\	Backslash (\u005C)
\x *XX*	Das Latin-1-Zeichen, das durch die hexadezimalen Ziffern *XX* angegeben wird.
\u *XXXX*	Das Unicode-Zeichen, das durch die vier hexadezimalen Ziffern *XXXX* angegeben wird.

Steht das \-Zeichen vor einem anderen Zeichen als den in Tabelle 2-1 aufgeführten, wird der Backslash einfach ignoriert (obwohl zukünftige Versionen der Sprache natürlich neue Escape-Sequenzen definieren können). Beispielsweise ist \# das Gleiche wie #. ECMAScript 5 erlaubt einen Backslash vor einem Zeilenumbruch, um ein Stringliteral auf mehrere Zeilen zu verteilen.

Eine der eingebauten Einrichtungen von JavaScript ist die Fähigkeit, Strings zu *verketten*. Wenn Sie den +-Operator mit Zahlen als Operanden verwenden, addiert er diese. Verwenden Sie ihn mit Strings als Operanden, verbindet er diese, indem er den zweiten String an den ersten anhängt. Zum Beispiel:

```
msg = "Hallo " + "Welt";   // => "Hallo Welt"
```

Die Länge eines Strings – also die Anzahl von 16-Bit-Werten, die er enthält – ermitteln Sie mit der length-Eigenschaft des Strings. Die Länge eines Strings s ermitteln Sie also folgendermaßen:

```
s.length
```

Neben der length-Eigenschaft gibt es eine Reihe von Methoden, die Sie auf Strings aufrufen können (wie immer finden Sie alle Details im Referenzabschnitt):

```
var s = "Hallo Welt" // Definiert einen Text.
s.charAt(0)          // => "H": Das erste Zeichen.
s.charAt(s.length-1) // => "t": Das letzte Zeichen.
s.substring(1,4)     // => "all": Das 2., 3. und 4. Zeichen.
s.slice(1,4)         // => "all": Das Gleiche.
s.slice(-3)          // => "elt": Die letzten drei Zeichen.
s.indexOf("l")       // => 2: Position des ersten
                     // Vorkommens des Buchstabens l.
s.lastIndexOf("l")   // => 9: Position des letzten
                     // Vorkommens des Buchstabens l.
s.indexOf("l", 3)    // => 3: Position des ersten
                     // "l" nach oder bei der 3. Position.
s.split(", ")        // => ["Hallo", "Welt"]:
                     // In Substrings aufgetrennt.
s.replace("h", "H")  // => "hallo Welt": Ersetzt Vorkommen.
s.toUpperCase()      // => "HALLO WELT": Alles in
                     // Großbuchstaben.
```

Denken Sie daran, dass Strings in JavaScript immer unveränderbar sind. Methoden wie replace() und toUpperCase() liefern neue Strings – den String, auf dem sie aufgerufen wurden, verändern sie nicht.

In ECMAScript 5 können Strings wie schreibgeschützte Arrays behandelt werden. Das heißt, Sie können auf die einzelnen Zeichen (16-Bit-Werte) eines Strings mit eckigen Klammern statt mit der Methode charAt() zugreifen:

```
s = "Hallo Welt";
s[0]                    // => "H"
s[s.length-1]           // => "t"
```

Boolesche Werte

Ein boolescher Wert repräsentiert Wahrheit oder Falschheit, an oder aus, ja oder nein. Für diesen Typ gibt es folglich nur zwei mögliche Werte. Die reservierten Wörter true und false werden zu diesen beiden Werten ausgewertet.

Boolesche Werte sind in der Regel die Ergebnisse von Vergleichen, die Sie in Ihren JavaScript-Programmen anstellen. Zum Beispiel:

```
a == 4
```

Dieser Code prüft, ob der Wert der Variablen a gleich der Zahl 4 ist. Ist das der Fall, ist das Ergebnis dieses Vergleichs der boolesche Wert true. Ist a nicht gleich 4, ist das Ergebnis des Vergleichs false.

Boolesche Werte kommen häufig in den Kontrollstrukturen von JavaScript zum Einsatz. Beispielsweise führt die if/else-Anweisung von JavaScript eine von zwei Aktionen aus, wenn ein boolescher Wert true ist, und die andere, wenn er false ist. Üblicherweise binden Sie den Vergleich, der den booleschen Wert hervorbringt, direkt in die Anweisung ein, die ihn nutzt. Das sieht dann folgendermaßen aus:

```
if (a == 4)
  b = b + 1;
else
  a = a + 1;
```

Dieser Code prüft, ob a gleich 4 ist. Ist das der Fall, addiert er 1 zu b hinzu; andernfalls addiert er 1 zu a hinzu.

Wie wir im Abschnitt »Typumwandlungen« auf Seite 16 erörtern werden, kann jeder JavaScript-Wert in einen booleschen Wert umgewandelt werden. Die folgenden Werte werden in false umgewandelt und verhalten sich deswegen auch wie false:

```
undefined
null
0
-0
NaN
""  // Der leere String.
```

Alle anderen Werte, einschließlich aller Objekte (und Arrays) werden in true umgewandelt und verhalten sich wie true. false und die sechs Werte, die in false umgewandelt werden, werden gelegentlich als *falsy*, alle anderen Werte als *truthy* bezeichnet. (Das führen wir hier nur für den Fall an, dass Sie in englischer Literatur einmal auf diese Bezeichnungen stoßen. Wir werden hier einfach von falschen und wahren Werten sprechen.) Erwartet JavaScript einen

booleschen Wert, verhält sich ein falscher Wert wie `false`, ein wahrer Wert wie `true`.

Nehmen Sie zum Beispiel an, die Variable `o` enthielte entweder ein Objekt oder den Wert `null`. Dass `o` nicht null ist, könnten Sie dann explizit mit einer `if`-Anweisung wie der folgenden prüfen:

```
if (o !== null) ...
```

Der Ungleich-Operator `!==` vergleicht `o` mit `null` und wird entweder zu `true` oder zu `false` ausgewertet. Sie können den Vergleich aber auch gleich weglassen und sich auf den Umstand stützen, dass `null` ein falscher Wert ist, Objekte hingegen wahre Werte sind:

```
if (o) ...
```

Im ersten Fall wird der Inhalt des `if`s nur ausgeführt, wenn `o` nicht `null` ist. Der zweite Fall ist weniger exakt: Der Body des `if`s wird ausgeführt, wenn `o` nicht `false` oder irgendein falscher Wert (wie `null` oder `undefined`) ist. Welche `if`-Anweisung für Ihr Programm angemessen ist, hängt vollkommen davon ab, welche Werte Sie für `o` erwarten. Wenn `null` anders behandelt werden soll als `0` und `""`, müssen Sie einen expliziten Vergleich einsetzen.

null und undefined

`null` ist ein Schlüsselwort, das zu einem speziellen Wert ausgewertet wird, der üblicherweise genutzt wird, um das Fehlen eines Wertes anzuzeigen. Wird der `typeof`-Operator auf `null` angewandt, erhält man als Ergebnis den String »object«. Das zeigt an, dass `null` als spezieller Objektwert betrachtet werden kann, der »kein Objekt« bedeutet. In der Praxis wird `null` üblicherweise jedoch als einziges Exemplar seines eigenen Typs betrachtet und kann ebenso eingesetzt werden, um für Zahlen oder Strings »kein Wert« anzuzeigen. Die meisten Programmiersprachen besitzen ein Äquivalent zu JavaScripts `null`: Vielleicht kennen Sie es als `null` oder `nil`.

JavaScript kennt noch einen weiteren Wert, der das Fehlen eines Wertes anzeigt. Der undefinierte Wert steht für ein grundlegenderes Fehlen. Es ist der Wert, den Variablen haben, die noch nicht initialisiert wurden, und der Wert, den Sie erhalten, wenn Sie den

Wert einer Objekteigenschaft oder eines Array-Elements abfragen, die bzw. das es nicht gibt. Der undefinierte Wert wird auch von Funktionen geliefert, die keinen Rückgabewert haben, und er wird als Wert für Funktionsparameter geliefert, für die kein Argument übergeben wurde. `undefined` ist eine vordefinierte globale Variable (kein Sprachschlüsselwort wie `null`), die auf den undefinierten Wert initialisiert ist. Wendet man den `typeof`-Operator auf den undefinierten Wert an, liefert er »undefined«. Das zeigt an, dass dieser Wert das einzige Exemplar eines speziellen Typs ist.

Trotz dieser Unterschiede können `null` und `undefined` beide das Fehlen eines Wertes anzeigen und häufig austauschbar verwendet werden. Der Gleichheitsoperator `==` betrachtet sie als gleich. (Nutzen Sie den strengen Gleichheitsoperator `===`, wenn Sie sie unterscheiden müssen.) Beides sind falsche Werte und verhalten sich wie `false`, wenn ein boolescher Wert benötigt wird. Weder `null` noch `undefined` haben irgendwelche Eigenschaften oder Methoden. Wenn Sie versuchen, mit `.` oder `[]` auf eine Eigenschaft oder Methode eines dieser Werte zuzugreifen, führt das sogar zu einem TypeError.

Das globale Objekt

Die vorangegangenen Abschnitte haben die elementaren Typen und Werte von JavaScript behandelt. Objekttypen – Objekte, Arrays und Funktionen – werden weiter unten im Buch in eigenständigen Kapiteln behandelt. Es gibt allerdings ein sehr wichtiges Objekt, das wir jetzt behandeln müssen. Das *globale Objekt* ist ein gewöhnliches JavaScript-Objekt, das einen sehr wichtigen Zweck erfüllt: Die Eigenschaften dieses Objekts sind die global definierten Symbole, die einem JavaScript-Programm zur Verfügung stehen. Wenn der JavaScript-Interpreter startet (oder ein Webbrowser eine neue Seite lädt), erstellt er ein neues globales Objekt und gibt ihm einen anfänglichen Satz von Eigenschaften, die Folgendes definieren:

- globale Eigenschaften wie `undefined`, `Infinity` und `NaN`
- globale Funktionen wie `isNaN()`, `parseInt()` (siehe »Typumwandlungen« auf Seite 16) und `eval()` (siehe »Auswertungsausdrücke« auf Seite 46).

- Konstruktorfunktionen wie `Date()`, `RegExp()`, `String()`, `Object()` und `Array()`
- Globale Objekte wie Math und JSON (siehe »Eigenschaften und Objekte serialisieren« auf Seite 87)

Die anfänglichen Eigenschaften des globalen Objekts sind keine reservierten Wörter, verdienen es aber, als solche behandelt zu werden. Dieses Kapitel hat sich bereits mit einigen dieser globalen Eigenschaften befasst. Mit den meisten anderen werden wir uns an anderen Stellen dieses Buchs noch befassen.

Auf der obersten Codeebene – JavaScript-Code, der nicht Teil einer Funktion ist – können Sie das JavaScript-Schlüsselwort `this` nutzen, um auf das globale Objekt zu verweisen:

```
var global = this; // /Auf das globale Objekt verweisen.
```

Bei clientseitigem JavaScript dient das Window-Objekt als das globale Objekt. Dieses globale Window-Objekt hat eine selbstreferenzielle `window`-Eigenschaft, um auf das globale Objekt zu verweisen. Das Window-Objekt definiert die globalen Eigenschaften des Sprachkerns von JavaScript und darüber hinaus eine ganze Menge andere globale Eigenschaften, die für Webbrowser und clientseitiges JavaScript eigentümlich sind (siehe Kapitel 10).

Nachdem es erstellt wurde, definiert das globale Objekt alle vordefinierten globalen Werte von JavaScript. Aber dieses spezielle Objekt hält auch von Programmen definierte globale Werte fest. Wenn Ihr Code eine globale Variable deklariert, ist diese Variable eine Eigenschaft des globalen Objekts.

Typumwandlungen

JavaScript ist sehr flexibel in Bezug auf die Typen der Werte, die es verlangt. Bei booleschen Werten haben wir das bereits gesehen: Erwartet JavaScript einen booleschen Wert, können Sie einen Wert beliebigen Typs angeben, der von JavaScript in die erforderliche Form umgewandelt oder konvertiert wird. Einige Werte (wahre Werte) werden in `true` umgewandelt, andere (falsche Werte) in `false`. Das Gleiche erfolgt auch bei anderen Werten: Braucht

JavaScript einen String, wandelt es den von Ihnen übergebenen Wert in einen String um. Braucht JavaScript eine Zahl, versucht es den von Ihnen angegebenen Wert in eine Zahl umzuwandeln (oder wandelt ihn in `NaN` um, wenn er nicht in eine vernünftige Zahl umgewandelt werden kann). Ein paar Beispiele:

```
10 + " Objekte"  // => "10 Objekte". 10 -> String
"7" * "4"        // => 28: Beide Strings -> Zahlen
var n = 1 - "x"; // => NaN: "x" kann nicht in eine Zahl
                 // umgewandelt werden.
n + " objects"   // => "NaN Objekte": NaN -> "NaN"
```

Tabelle 2-2 fasst zusammen, wie Werte in JavaScript von einem Typ in einen anderen umgewandelt werden. Fettgedruckte Tabelleneinträge unterstreichen Umwandlungen, die für Sie vielleicht überraschend sein könnten. Leere Zellen zeigen an, dass keine Umwandlung erforderlich ist und auch keine durchgeführt wird.

Tabelle 2-2: JavaScript-Typumwandlungen

Wert	Umgewandelt in:			
	String	**Zahl**	**Boolescher Wert**	**Objekt**
`undefined`	`"undefined"`	`NaN`	`false`	*meldet einen TypeError*
`null`	`"null"`	**`0`**	`false`	*meldet einen TypeError*
`true`	`"true"`	**`1`**		`Boolean(true)`
`false`	`"false"`	**`0`**		`Boolean(false)`
`""` (leerer String)		**`0`**	**`false`**	`String("")`
`"1.2"` (nicht leer, numerisch)		`1.2`	`true`	`String("1.2")`
`"eins"` (nicht leer, nicht numerisch)		`NaN`	`true`	`String("ein")`
`0`	`"0"`		**`false`**	`Number(0)`
`-0`	`"0"`		**`false`**	`Number(-0)`
`NaN`	`"NaN"`		**`false`**	`Number(NaN)`
`Infinity`	`"Infinity"`		`true`	`Number(Infinity)`
`-Infinity`	`"-Infinity"`		`true`	`Number(-Infinity)`
`1` (endlich, nicht 0)	`"1"`		`true`	`Number(1)`