Behandelt HTML5 & ECMAScript 5

6. Auflage

JavaScript

Das umfassende Referenzwerk

David Flanagan

Deutsche Übersetzung von
Lars Schulten & Thomas Demmig

O'REILLY®

Inhalt

KAPITEL 4

Ausdrücke und Operatoren

Ein *Ausdruck* ist eine JavaScript-Äußerung, die der JavaScript-Interpreter *auswerten* kann, um einen Wert hervorzubringen. Eine Konstante, die direkt in Ihr Programm eingebettet ist, ist eine sehr einfache Art von Ausdruck. Ein Variablenname ist ebenfalls ein einfacher Ausdruck, der zu dem Wert ausgewertet wird, der dieser Variable zugewiesen wurde. Komplexe Ausdrücke werden aus einfacheren Ausdrücken aufgebaut. Ein Array-Zugriffsausdruck besteht beispielsweise aus einem Ausdruck, der zu einem Array ausgewertet wird, auf den eine öffnende eckige Klammer folgt, dann aus einem Ausdruck, der zu einer ganzen Zahl ausgewertet wird, und schließlich einer schließenden eckigen Klammer. Dieser neue, komplexere Ausdruck wird zu dem Wert ausgewertet, der unter dem entsprechenden Index im angegebenen Array gespeichert ist. Ähnlich besteht ein Funktionsaufruf aus einem Ausdruck, der zu einem Funktionsobjekt ausgewertet wird, und null oder mehr zusätzlichen Ausdrücken in einer Klammer, die als die Argumente für die Funktion verwendet werden.

Am häufigsten setzt man für den Aufbau komplexer Ausdrücke aus einfachen Ausdrücken Operatoren ein. Ein *Operator* kombiniert die Werte seiner *Operanden* (üblicherweise zwei) auf bestimmte Weise und wird zu einem neuen Wert ausgewertet. Der Multiplikationsoperator * ist ein einfaches Beispiel. Der Ausdruck `x * y` wird zum Produkt der Werte der Ausdrücke `x` und `y` ausgewertet. Der Einfachheit halber sagen wir gelegentlich, dass ein Operator einen Wert *zurückliefert*, nicht »er wird zu einem Wert ausgewertet«.

Dieses Kapitel dokumentiert alle Operatoren von JavaScript und erläutert außerdem Ausdrücke (wie die Array-Indizierung und den Funktionsaufruf), die keine Operatoren nutzen. Wenn Sie bereits eine andere Programmiersprache kennen, die eine C-ähnliche Syntax nutzt, werden Sie feststellen, dass Ihnen die Syntax der meisten Ausdrücke und Operatoren von JavaScript bereits vertraut ist.

4.1 Elementare Ausdrücke

Die einfachsten Ausdrücke, die auch als *elementare Ausdrücke* bezeichnet werden, sind die, die eigenständig sind – d.h., sie können nicht in weitere elementare Ausdrücke zerlegt

werden. Elementare Ausdrücke sind in JavaScript konstante oder *literale* Werte, einige Schlüsselwörter der Sprache und Variablenreferenzen.

Literale sind konstante Werte, die unmittelbar in Ihr Programm eingebettet sind. Sie sehen beispielsweise so aus:

```
1.23         // Ein Zahlliteral.
"hello"      // Ein Stringliteral.
/pattern/    // Ein Regex-Literal.
```

Die JavaScript-Syntax für Zahlliterale wurde in § 3.1 behandelt. Stringliterale wurden in § 3.2 dokumentiert. Die Syntax für Regex-Literale wurde in § 3.2.4 eingeführt und wird in Kapitel 10 ausführlich behandelt werden.

Einige der reservierten Wörter von JavaScript sind elementare Ausdrücke:

```
true      // Wird zum booleschen Wahr-Wert ausgewertet.
false     // Wird zum booleschen Falsch-Wert ausgewertet.
null      // Wird zum Nullwert ausgewertet.
this      // Wird zum "aktuellen" Objekt ausgewertet.
```

`true`, `false` und `null` haben wir in § 3.3 und § 3.4 kennengelernt. Im Unterschied zu den anderen Schlüsselwörtern ist `this` keine Konstante – es wird an unterschiedlichen Stellen eines Programms zu unterschiedlichen Werten ausgewertet. Das Schlüsselwort `this` wird in der objektorientierten Programmierung genutzt. Im Rumpf einer Methode wird `this` zu dem Objekt ausgewertet, auf dem die Methode aufgerufen wurde. Mehr Informationen zu `this` finden Sie in § 4.5, Kapitel 8 (insbesondere § 8.2.2) und Kapitel 9.

Die dritte Art von elementarem Ausdruck schließlich sind einfache Variablenreferenzen:

```
i            // Wird zum Wert der Variablen i ausgewertet.
sum          // Wird zum Wert der Variablen sum ausgewertet.
undefined    // undefined ist eine globale Variable, kein Schlüsselwort wie null.
```

Erscheint ein Bezeichner unabhängig in einem Programm, geht JavaScript davon aus, dass es sich um eine Variable handelt. Gibt es keine Variable dieses Namens, wird der Ausdruck zum Wert `undefined` ausgewertet. Im Strict-Modus von ECMAScript 5 führt der Versuch, eine Variable auszuwerten, die es nicht gibt, stattdessen jedoch zu einem ReferenceError.

4.2 Objekt- und Array-Initialisierer

Objekt- und Array-*Initialisierer* sind Ausdrücke, deren Wert ein neu erzeugtes Objekt oder Array ist. Diese Initialisierungsausdrücke werden gelegentlich auch als »Objektliterale« und »Array-Literale« bezeichnet. Im Unterschied zu echten Literalen sind diese jedoch keine elementaren Ausdrücke, weil sie mehrere Teilausdrücke enthalten, die Eigenschafts- und Elementwerte angeben. Array-Initialisierer haben eine etwas einfachere Syntax. Mit diesen wollen wir deswegen beginnen.

Ein Array-Initialisierer ist eine durch Kommata getrennte Liste von Ausdrücken in eckigen Klammern. Der Wert eines Array-Initialisierers ist ein neu erzeugtes Array. Die Elemente

dieses neuen Arrays sind auf die Werte der durch Kommata getrennten Ausdrücke initialisiert:

```
[]        // Ein leeres Array: Keine Ausdrücke in den eckigen Klammern heißt keine
          // Elemente.
[1+2,3+4] // Ein Array mit zwei Elementen. Das erste Element ist 3, das zweite 7.
```

Die Elementausdrücke in einem Array-Initialisierer können selbst auch wieder Array-Initialisierer sein, was bedeutet, dass diese Ausdrücke geschachtelte Arrays erstellen können:

```
var matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

Die Elementausdrücke in einem Array-Initialisierer werden jedes Mal ausgewertet, wenn der Array-Initialisierer ausgewertet wird. Das heißt, dass der Wert eines Array-Initialisierungsausdrucks bei jeder Auswertung ein anderer sein kann.

Elemente mit dem undefinierten Wert können in Array-Literale eingeschlossen werden, indem einfach zwischen zwei Kommata der Wert weggelassen wird. Das folgende Array enthält beispielsweise fünf Elemente, von denen drei undefiniert sind:

```
var sparseArray = [1,,,,5];
```

Nach dem letzten Ausdruck in einem Array-Initialisierer ist ein nachstehendes Komma erlaubt, das kein undefiniertes Element erstellt.

Objektinitialisierungsausdrücke ähneln Array-Initialisierungsausdrücken, stehen statt in eckigen Klammern aber in geschweiften Klammern. Außerdem werden jedem Teilausdruck ein Eigenschaftsname und ein Doppelpunkt vorangestellt:

```
var p = { x:2.3, y:-1.2 };  // Ein Objekt mit zwei Eigenschaften.
var q = {};                 // Ein leeres Objekt ohne Eigenschaften.
q.x = 2.3; q.y = -1.2;      // Jetzt hat q die gleichen Eigenschaften wie p.
```

Objektliterale können ebenfalls geschachtelt werden. Zum Beispiel:

```
var rectangle = { upperLeft:  { x: 2, y: 2 },
                  lowerRight: { x: 4, y: 5 } };
```

Die Ausdrücke in einem Objektinitialisierer werden jedes Mal ausgewertet, wenn der Objektinitialisierer ausgewertet wird und müssen keine konstanten Werte haben: Es können beliebige JavaScript-Ausdrücke sein. Außerdem können die Eigenschaftsnamen in Objektliteralen Strings statt Bezeichner sein (was nützlich sein kann, wenn man Eigenschaftsnamen angeben muss, die reservierte Wörter sind oder die aus anderem Grund keine zulässigen Bezeichner darstellen):

```
var side = 1;
var square = { "upperLeft":  { x: p.x, y: p.y },
               'lowerRight': { x: p.x + side, y: p.y + side}};
```

Objekt- und Array-Initialisierer werden uns in den Kapiteln 6 und 7 erneut begegnen.

4.3 Funktionsdefinitionsausdrücke

Ein Funktionsdefinitionsausdruck definiert eine JavaScript-Funktion, und der Wert eines solchen Ausdrucks ist die neu definierte Funktion. In gewisser Weise ist ein Funktionsdefinitionsausdruck ein »Funktionsliteral«, ähnlich wie ein Objektinitialisierer ein »Objektliteral« ist. Ein Funktionsdefinitionsausdruck besteht üblicherweise aus dem Schlüsselwort `function`, einer kommaseparierten Liste weiterer Bezeichner (den Parameternamen) in Klammern und einem Block JavaScript-Code (dem Funktionsrumpf) in geschweiften Klammern. Zum Beispiel:

```
// Diese Funktion liefert das Quadrat des übergebenen Werts.
var square = function(x) { return x * x; }
```

Ein Funktionsdefinitionsausdruck kann auch einen Namen für die Funktion enthalten. Funktionen können auch mit einer Funktionsanweisung statt einem Funktionsausdruck definiert werden. Vollständige Informationen zu Funktionsdefinitionen finden Sie in Kapitel 8.

4.4 Eigenschaftszugriffsausdrücke

Der Eigenschaftszugriffsausdruck wird zum Wert einer Objekteigenschaft oder eines Array-Elements ausgewertet. JavaScript definiert zwei Syntaxformen für den Eigenschaftszugriff:

```
Ausdruck . Bezeichner
Ausdruck [ Ausdruck ]
```

Bei der ersten Art des Eigenschaftszugriffs wird ein Ausdruck genutzt, auf den ein Punkt und dann ein Bezeichner folgen. Der Ausdruck gibt das Objekt an und der Bezeichner den Namen der gewünschten Eigenschaft. Bei der zweiten Art des Eigenschaftszugriffs folgt auf den ersten Ausdruck (das Objekt oder Array) ein weiterer Ausdruck in eckigen Klammern. Dieser zweite Ausdruck gibt den Namen der gewünschten Eigenschaft oder den Index des gewünschten Array-Elements an. Hier sind einige konkrete Beispiele:

```
var o = {x:1,y:{z:3}};  // Ein Beispielobjekt.
var a = [o,4,[5,6]];    // Ein Beispiel-Array, das das Objekt enthält.
o.x                     // => 1: Eigenschaft x des Ausdrucks o.
o.y.z                   // => 3: Eigenschaft z des Ausdrucks o.y.
o["x"]                  // => 1: Eigenschaft x des Objekts o.
a[1]                    // => 4: Das Element beim Index 1 des Ausdrucks a.
a[2]["1"]               // => 6: Das Element beim Index 1 des Ausdrucks a[2]
a[0].x                  // => 1: Eigenschaft x des Ausdrucks a[0].
```

Bei beiden Arten des Eigenschaftszugriffs wird der Ausdruck vor dem . bzw. der [zuerst ausgewertet. Ist der Wert `null` oder `undefined`, löst der Eigenschaftszugriff einen TypeError aus, da diese beiden JavaScript-Werte keine Eigenschaften haben können. Ist der Wert kein Objekt (oder Array), wird er in eines umgewandelt (siehe § 3.6). Folgen auf den Objektausdruck ein Punkt und ein Bezeichner, wird der Wert der durch diesen Bezeichner angegebenen Eigenschaft nachgeschlagen und wird zum Wert des gesamten Ausdrucks.

Folgt auf den Objektausdruck ein weiterer Ausdruck in eckigen Klammern, wird dieser zweite Ausdruck ausgewertet und in einen String umgewandelt. Der Wert des gesamten Ausdrucks ist dann der Wert der Eigenschaft, deren Name der String angibt. In beiden Fällen ist der Wert des Eigenschaftszugriffsausdrucks `undefined`, wenn es die angegebene Eigenschaft nicht gibt.

Die *.Bezeichner*-Syntax ist die einfachere der beiden Optionen für den Eigenschaftszugriff. Beachten Sie jedoch, dass sie nur eingesetzt werden kann, wenn die Eigenschaft, auf die Sie zugreifen wollen, einen Namen hat, der ein zulässiger Bezeichnername ist, und wenn Sie den Namen des Bezeichners bereits beim Schreiben des Programms kennen. Wenn der Eigenschaftsname einem reservierten Wort entspricht, Leerzeichen oder Interpunktionszeichen enthält oder eine Zahl ist (bei Arrays), müssen Sie die Notation mit den eckigen Klammern verwenden. Eckige Klammern werden auch genutzt, wenn der Eigenschaftsname nicht statisch, sondern selbst das Ergebnis einer Berechnung ist (ein Beispiel finden Sie in § 6.2.1).

Objekte und ihre Eigenschaften werden in Kapitel 6 ausführlich behandelt, Arrays und ihre Elemente in Kapitel 7.

4.5 Aufrufausdrücke

Als *Aufrufausdruck* bezeichnet man JavaScripts Syntax für den Aufruf (oder die Ausführung) einer Funktion oder Methode. Sie beginnt mit einem Funktionsausdruck, der die aufzurufende Funktion bezeichnet. Auf den Funktionsausdruck folgen eine öffnende Klammer, eine kommagetrennte Liste mit null oder mehr Argumenten und eine schließende Klammer. Ein paar Beispiele:

```
f(0)           // f ist der Funktionsausdruck; 0 der Argumentausdruck.
Math.max(x,y,z) // Math.max ist die Funktion; x, y und z sind die Argumente.
a.sort()       // a.sort ist die Funktion; es gibt keine Argumente.
```

Wird ein Aufrufausdruck ausgewertet, wird zuerst der Funktionsausdruck ausgewertet. Anschließend werden die Argumentausdrücke ausgewertet, um eine Liste mit Argumentwerten zu erstellen. Ist der Wert des Funktionsausdrucks kein aufrufbares Objekt, wird ein TypeError ausgelöst. (Alle Funktionen sind aufrufbar. Host-Objekte können auch aufrufbar sein, wenn sie keine Funktionen sind. Diesen Unterschied sehen wir uns in § 8.7.7 an.) Anschließend werden die Argumentwerte der Reihe nach den Parameternamen zugewiesen, die bei der Definition der Funktion angegeben wurden, bevor der Codeinhalt der Funktion ausgeführt wird. Nutzt die Funktion eine `return`-Anweisung, um einen Wert zu liefern, wird dieser Wert zum Wert des Aufrufausdrucks. Andernfalls ist der Wert des Aufrufausdrucks `undefined`. Vollständige Informationen zum Funktionsaufruf, einschließlich einer Erklärung dessen, was geschieht, wenn die Anzahl von Argumentausdrücken nicht der Anzahl an Parametern in der Funktionsdefinition entspricht, finden Sie in Kapitel 8.

Jeder Aufrufausdruck enthält ein Klammernpaar und einen Ausdruck vor diesen Klammern. Ist dieser Ausdruck ein Eigenschaftszugriffsausdruck, bezeichnet man den Aufruf

selbst als einen *Methodenaufruf*. Bei Methodenaufrufen wird das Objekt oder Array, das der Gegenstand des Eigenschaftsaufrufs ist, zum Wert des `this`-Parameters für die Ausführung des Codeinhalts der Funktion. Das ist das Fundament des objektorientierten Programmierparadigmas, bei dem Funktionen (die unter diesem Paradigma eben als »Methoden« bezeichnet werden) auf dem Objekt operieren, dessen Teil sie sind. Mehr dazu finden Sie unter Kapitel 9.

Aufrufausdrücke, die keine Methodenaufrufe sind, nutzen üblicherweise das globale Objekt als Wert des `this`-Schlüsselworts. In ECMAScript 5 erhalten Funktionen, die im Strict-Modus definiert wurden, für `this` statt des globalen Objekts den Wert `undefined`. Mehr Informationen zum Strict-Modus finden Sie unter § 5.7.3.

4.6 Objekterstellungsausdrücke

Ein *Objekterstellungsausdruck* erstellt ein neues Objekt und ruft eine (als »Konstruktor« bezeichnete) Funktion auf, um die Eigenschaften dieses Objekts zu initialisieren. Objekterstellungsausdrücke sind wie Aufrufausdrücke; ihnen wird allerdings das Schlüsselwort `new` vorangestellt:

```
new Object()
new Point(2,3)
```

Werden der Konstruktorfunktion in einem Objekterstellungsausdruck keine Argumente übergeben, kann die leere Klammer weggelassen werden:

```
new Object
new Date
```

Bei der Auswertung eines Objekterstellungsausdrucks erstellt JavaScript zunächst ein neues leeres Objekt wie das, das vom Objektinitialisierer `{}` erstellt wird. Dann wird die die angegebene Funktion mit den angegebenen Argumenten aufgerufen, und dabei wird das neue Objekt als Wert des Schlüsselworts `this` übergeben. Die Funktion kann `this` dann nutzen, um die Eigenschaften des neu erstellten Objekts zu initialisieren. Funktionen, die zur Verwendung als Konstruktoren geschrieben werden, liefern üblicherweise keinen Wert zurück. Der Wert des Objekterstellungsausdrucks ist dann das neu erstellte und initialisierte Objekt. Liefert ein Konstruktor einen Objektwert zurück, wird dieser zum Wert des Objekterstellungsausdrucks, während das neu erstellte Objekt verworfen wird.

Konstruktoren werden in Kapitel 9 ausführlicher erläutert.

4.7 Operatoren im Überblick

Operatoren werden für JavaScripts arithmetische Ausdrücke, Vergleichsausdrücke, logische Ausdrücke, Zuweisungsausdrücke und anderes genutzt. Tabelle 4-1 bietet einen Überblick über die Operatoren und kann als praktische Referenz dienen.

Beachten Sie, dass die meisten Operatoren durch Interpunktionszeichen wie + und = repräsentiert werden. Nur einige wenige werden durch Schlüsselwörter wie `delete` und `instanceof` repräsentiert. Schlüsselwortoperatoren sind ganz gewöhnliche Operatoren, die sich in keiner Weise von denen unterscheiden, die durch Interpunktionszeichen ausgedrückt werden: Sie haben einfach nur eine etwas weniger kompakte Darstellung.

Tabelle 4-1 ist anhand des Operatorenvorrangs aufgebaut. Die zuerst aufgeführten Operatoren haben einen höheren Vorrang als die nachfolgend aufgeführten. Operatoren, zwischen denen eine horizontale Linie steht, haben unterschiedliche Vorrangstufen. Die Spalte mit der Überschrift A gibt die Assoziativität des Operators an. Der Wert kann L (von links nach rechts) oder R (von rechts nach links) sein. Die Spalte N gibt die Anzahl von Operanden an. Die Spalte mit der Überschrift *Typen* führt die erwarteten Typen der Operanden und (nach dem →-Symbol) den Typ des Ergebnisses der Operatoroperation auf (dabei steht Lval für Lvalue – was unten erläutert wird –, Zahl für eine beliebige Zahl, Int für eine ganze Zahl, Bool für einen booleschen Wert, Str für String, Obj für Objekt, Funk für Funktion und bel für einen Wert eines beliebigen Typs). Die auf die Tabelle folgenden Unterabschnitte erläutern die Konzepte *Vorrang*, *Assoziativität* und *Operatortyp*. Die Operatoren selbst werden im Anschluss an diese Erläuterungen einzeln dokumentiert.

Tabelle 4-1: JavaScript-Operatoren

Operator	Operation	A	N	Typen
++	Prä- oder Post-Inkrement	R	1	Lval→Zahl
--	Prä- oder Post-Dekrement	R	1	Lval→Zahl
-	Zahl negieren	R	1	Zahl→Zahl
+	in Zahl umwandeln	R	1	Zahl→Zahl
~	Bits invertieren	R	1	Int→Int
!	booleschen Wert invertieren	R	1	Bool→Bool
delete	Eigenschaft löschen	R	1	Lval→Bool
typeof	Typ ermitteln	R	1	bel→Str
void	undefinierten Wert liefern	R	1	bel→undef
*, /, %	Multiplikation, Division, Rest	L	2	Zahl,Zahl→Zahl
+, -	Addieren, Subtrahieren	L	2	Zahl,Zahl→Zahl
+	Strings verketten	L	2	Str,Str→Str
<<	nach links verschieben	L	2	Int,Int→Int
>>	mit Vorzeichenerweiterung nach rechts verschieben	L	2	Int,Int→Int
>>>	mit Nullauffüllung nach rechts verschieben	L	2	Int,Int→Int
<, <=,>, >=	in numerischer Folge vergleichen	L	2	Zahl,Zahl→Bool
<, <=,>, >=	in alphabetischer Folge vergleichen	L	2	Str,Str→Bool
instanceof	Objektklasse prüfen	L	2	Obj,Funk→Bool
in	prüfen, ob es eine Eigenschaft gibt	L	2	Str,Obj→Bool
==	auf Gleichheit prüfen	L	2	bel,bel→Bool

Tabelle 4-1: JavaScript-Operatoren (Fortsetzung)

Operator	Operation	A	N	Typen
!=	auf Ungleichheit prüfen	L	2	bel,bel→Bool
===	auf Gleichheit im strengen Sinne prüfen	L	2	bel,bel→Bool
!==	auf Ungleichheit im strengen Sinne prüfen	L	2	bel,bel→Bool
&	bitweises UND berechnen	L	2	Int,Int→Int
^	bitweises XODER berechnen	L	2	Int,Int→Int
|	bitweises ODER berechnen	L	2	Int,Int→Int
&&	logisches UND berechnen	L	2	bel,bel→bel
||	logisches ODER berechnen	L	2	bel,bel→bel
?:	zweiten oder dritten Operanden wählen	R	3	Bool,bel,bel→bel
=	Zuweisung an eine Variable oder Eigenschaft	R	2	Lval,bel→bel
*=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=	Operation und Zuweisung	R	2	Lval,bel→bel
,	ersten Operanden verwerfen, zweiten liefern	L	2	bel,bel→bel

4.7.1 Anzahl an Operanden

Operatoren können auf Grundlage der von ihnen erwarteten Operanden (ihrer *Stelligkeit*) kategorisiert werden. Die meisten JavaScript-Operatoren sind wie beispielsweise der *-Multiplikationsoperator *binäre Operatoren*, die zwei Ausdrücke zu einem komplexeren Ausdruck vereinen. Das heißt, dass sie zwei Operanden erwarten. JavaScript unterstützt darüber hinaus eine Reihe von *unären Operatoren*, die einen einzigen Ausdruck in einen einzigen komplexeren Ausdruck umwandeln. Der --Operator im Ausdruck -x ist ein unärer Operator, der eine Negationsoperation auf dem Operanden x ausführt. Außerdem unterstützt JavaScript einen *ternären Operator*, den Bedingungsoperator ?:, der drei Ausdrücke zu einem einzigen Ausdruck kombiniert.

4.7.2 Operanden- und Ergebnistyp

Einige Operatoren arbeiten mit Werten beliebiger Typen, aber die meisten erwarten Operanden eines spezifischen Typs, und die meisten Operatoren liefern einen Wert eines spezifischen Typs (oder werden zu einem solchen ausgewertet). Die Typen-Spalte in Tabelle 4-1 gibt (vor dem Pfeil) die Typen der Operanden und (nach dem Pfeil) den Typ des Ergebnisses für die Operatoren an.

Die Operatoren von JavaScript wandeln den Typ ihrer Operanden üblicherweise nach Bedarf um (siehe § 3.8). Der Multiplikationsoperator * erwartet numerische Operanden, doch der Ausdruck "3" * "5" ist dennoch zulässig, da JavaScript die Operanden in Zahlen umwandeln kann. Der Wert dieses Ausdrucks ist natürlich die Zahl 15, nicht der String »15«. Beachten Sie auch, dass jeder JavaScript-Wert entweder ein wahrer oder ein falscher

Wert ist und dass Operatoren, die boolesche Operanden erwarten, deswegen mit Operanden beliebigen Typs arbeiten.

Das Verhalten einiger Operatoren kann vom Typ der verwendeten Operanden abhängig sein. Der auffälligste Fall ist der des +-Operators, der numerische Operanden addiert, Stringoperanden hingegen verkettet. Gleichermaßen führen Vergleichsoperatoren wie < den Vergleich in Abhängigkeit vom Operandentyp auf numerische oder alphabetische Weise durch. Bei den Beschreibungen der einzelnen Operatoren werden ihre Typabhängigkeiten erläutert und die durchgeführten Umwandlungen angegeben.

4.7.3 Lvalues

Beachten Sie, dass die Zuweisungsoperatoren und einige wenige andere der in Tabelle 4-1 aufgeführten Operatoren einen Operanden des Typs `Lval` erwarten. *Lvalue* (Linkswert) ist ein geschichtlicher Begriff, der »einen Ausdruck, der auf der linken Seite einer Zuweisungsanweisung erscheinen darf« bezeichnet. In JavaScript sind Variablen, Objekteigenschaften und die Elemente von Arrays Lvalues. Die ECMAScript-Spezifikation erlaubt eingebauten Funktionen, Lvalues zurückzuliefern, definiert aber keine Funktionen, die sich auf diese Weise verhalten.

4.7.4 Seiteneffekte von Operatoren

Die Auswertung eines einfachen Ausdrucks wie `2 * 3` wirkt sich nie auf den Zustand Ihres Programms aus, d.h., dass keine der in der Folge von Ihrem Programm ausgeführten Berechnungen von dieser Auswertung beeinträchtigt wird. Einige Ausdrücke haben jedoch *Seiteneffekte* – ihre Auswertung kann sich auf das Ergebnis späterer Auswertungen auswirken. Die Zuweisungsoperatoren sind das offensichtlichste Beispiel: Weisen Sie einer Variablen oder einer Eigenschaft einen Wert zu, ändert das den Wert jedes Ausdrucks, der diese Variable bzw. Eigenschaft nutzt. Die `++`- und `--`-Inkrement- und Dekrement-Operatoren verhalten sich ebenfalls so, da sie implizit eine Zuweisung durchführen. Auch der `delete`-Operator hat Seiteneffekte: Wird eine Eigenschaft gelöscht, ist das ähnlich (aber nicht das Gleiche), als würde der Eigeschaft `undefined` zugewiesen.

Keiner der anderen JavaScript-Operatoren hat Seiteneffekte, aber Funktionsaufruf- und Objekterstellungsausdrücke haben Seiteneffekte, wenn einer der im Codeinhalt der Funktion bzw. des Konstruktors genutzten Operatoren Seiteneffekte hat.

4.7.5 Operatorvorrang

Die in Tabelle 4-1 aufgeführten Operatoren sind nach absteigendem Vorrang angeordnet, und zwischen den verschiedenen Vorrangstufen wurden Trennlinien eingefügt. Der Operatorvorrang steuert, in welcher Abfolge Operationen ausgeführt werden. Operatoren mit höherem Vorrang (die, die weiter oben in der Tabelle stehen) werden vor denen mit niedrigerem Vorrang (denen, die weiter unten in der Tabelle stehen) ausgeführt.

Betrachten Sie die folgenden Ausdrücke:

```
w = x + y*z;
```

Der Multiplikationsoperator * hat einen höheren Vorrang als der Additionsoperator +, die Multiplikation wird also vor der Addition ausgeführt. Darüber hinaus hat der Zuweisungsoperator = den geringsten Vorrang, die Zuweisung wird also erst ausgeführt, nachdem alle Operationen auf der rechten Seite ausgeführt wurden.

Der Operatorvorrang kann durch den expliziten Einsatz von Klammern überschrieben werden. Ändern Sie den Ausdruck folgendermaßen, wenn Sie erzwingen wollen, dass die Addition zuerst ausgeführt wird:

```
w = (x + y)*z;
```

Beachten Sie,dass Eigenschaftszugriffs- und Aufrufausdrücke höheren Vorrang haben als alle in Tabelle 4-1 aufgeführten Operatoren. Betrachten Sie folgenden Ausdruck:

```
typeof my.functions[x](y)
```

Obwohl `typeof` einer der Operatoren mit dem höchsten Vorrang ist, wird die `typeof`-Operation auf dem Ergebnis der beiden Eigenschaftszugriffe und dem Funktionsaufruf ausgeführt.

Sollten Sie sich nicht sicher sein, welchen Vorrang die von Ihnen verwendeten Operatoren haben, ist es empfehlenswert, die Auswertung explizit durch die Einführung von Klammern zu steuern. Folgende Regeln sollten Sie sich auf alle Fälle merken: Multiplikation und Division werden vor Addition und Subtraktion ausgeführt; die Zuweisung hat sehr geringen Vorrang und wird fast immer zuletzt ausgeführt.

4.7.6 Operatorassoziativität

Die mit A beschriftete Spalte in Tabelle 4-1 gibt die *Assoziativität* des Operators an. Der Wert L sagt, dass der Operator linksassoziativ ist, der Wert R, dass er rechtsassoziativ ist. Die Assoziativität eines Operators definiert, in welcher Reihenfolge Operationen mit gleichem Vorrang ausgeführt werden. Linksassoziativität heißt, dass die Operationen von links nach rechts ausgeführt werden. Da der Subtraktionsoperator linksassoziativ ist, ist

```
w = x - y - z;
```

das Gleiche wie:

```
w = ((x - y) - z);
```

Betrachten Sie andererseits die folgenden Ausdrücke:

```
x = ~-y;
w = x = y = z;
q = a?b:c?d:e?f:g;
```

Diese entsprechen

```
x = ~(-y);
w = (x = (y = z));
q = a?b:(c?d:(e?f:g));
```

weil die unären Operatoren, der Zuweisungs- und der Ternäroperator rechtsassoziativ sind und die entsprechenden Operationen von rechts nach links ausgeführt werden.

4.7.7 Reihenfolge der Auswertung

Operatorvorrang und Assoziativität bedingen die Reihenfolge, in der die Operationen in einem komplexen Ausdruck ausgeführt werden. Sie bedingen jedoch nicht, in welcher Reihenfolge Unterausdrücke ausgewertet werden. Unterausdrücke werden in JavaScript immer streng von links nach rechts ausgewertet. Im Ausdruck `w=x+y*z` wird beispielsweise immer zuerst der Unterausdruck `w` ausgewertet und anschließend in dieser Reihenfolge `x`, `y` und `z`. Dann werden die Werte `y` und `z` multipliziert, dem Wert von `x` hinzugefügt und der Variablen oder Eigenschaft zugewiesen, die durch den Ausdruck `w` angegeben wird. Wenn Sie in die Ausdrücke Klammern einfügen, können Sie die relative Abfolge der Multiplikation, Addition und Zuweisung steuern, aber nicht die strikt von links nach rechts laufende Auswertung der Unterausdrücke.

Die Auswertungsabfolge wird nur dann relevant, wenn einer der ausgewerteten Ausdrücke Seiteneffekte hat, die den Wert eines der anderen Ausdrücke betreffen. Inkrementiert der Ausdruck `x` eine Variable, die vom Ausdruck `z` genutzt wird, wird der Umstand relevant, dass `x` vor `z` ausgewertet wird.

4.8 Arithmetische Ausdrücke

Dieser Abschnitt behandelt die Operatoren, die arithmetische oder anderere numerische Operationen auf ihren Operanden ausführen. Die Multiplikations-, Divisions- und Subtraktionsoperatoren sind problemlos und werden zuerst behandelt. Der Additionsoperator erhält einen eigenen Unterabschnitt, weil er auch Stringverkettungen durchführen kann und einige ungewöhnliche Umwandlungsregeln hat. Die unären Operatoren und die bitweisen Operatoren werden ebenfalls in eigenen Unterabschnitten behandelt.

Die elementaren arithmetischen Operatoren sind * (Multiplikation), / (Division), % (Modulo: Rest nach der Division), + (Addition) und - (Subtraktion). Wie bereits erwähnt, werden wir den +-Operator in einem eigenständigen Abschnitt behandeln. Die anderen vier elementaren Operatoren werten einfach ihre Operanden aus, wandeln die Werte bei Bedarf in Zahlen um und berechnen dann das Produkt, den Quotienten, den Rest oder die Differenz zwischen den Werten. Nicht-numerische Operanden, die nicht in Zahlen umgewandelt werden können, werden in den `NaN`-Wert umgewandelt. Wenn einer der Operanden `NaN` ist (oder in darin umgewandelt wird), ist das Ergebnis der Operation ebenfalls `NaN`.

Der /-Operator teilt seinen ersten Operanden durch seinen zweiten Operanden. Wenn Sie an Programmiersprachen gewöhnt sind, die Ganz- und Gleitkommazahlen unterscheiden, erwarten Sie vielleicht ein ganzzahliges Ergebnis, wenn Sie eine ganze Zahl durch eine andere ganze Zahl teilen. Aber da in JavaScript alle Zahlen Gleitkommazahlen sind, haben alle Divisionsoperationen Gleitkommazahlen zum Ergebnis: `5/2` wird zu `2.5` ausgewertet, nicht zu `2`. Die Division durch null liefert Plus- oder Minus-Unendlich, während `0/0` zu `NaN` ausgewertet wird: In keinem dieser Fälle wird ein Fehler gemeldet.

Der %-Operator führt eine Modulodivision durch, d.h., er liefert den Rest, der verbleibt, wenn man den ersten Operanden ganzzahlig durch den zweiten Operanden teilt. Das Vorzeichen des Ergebnisses entspricht dem Vorzeichen des ersten Operanden. Beispielsweise wird `5 % 2` zu `1` ausgewertet und `-5 % 2` zu `-1`.

Obgleich der Modulooperator üblicherweise mit ganzzahligen Operanden verwendet wird, funktioniert er auch bei Gleitkommawerten. Beispielsweise wird `6.5 % 2.1` zu `0.2` ausgewertet.

4.8.1 Der +-Operator

Der binäre +-Operator addiert numerische Operanden oder verkettet Stringoperanden:

```
1 + 2                    // => 3
"Hallo" + " " + "Welt"   // => "Hallo Welt"
"1" + "2"                // => "12"
```

Wenn die Werte der Operanden beide Zahlen oder beide Strings sind, ist offensichtlich, was der +-Operator tut. In allen anderen Fällen sind hingegen Typumwandlungen erforderlich. Die Operation, die letztendlich ausgeführt wird, ist von den Umwandlungen abhängig, die vorgenommen werden. Die Umwandlungsregeln für + bevorzugen die Stringverkettung: Ist einer der Operanden ein String oder ein Objekt, das in einen String umgewandelt werden kann, wird der andere Operand in einen String umgewandelt und eine Stringverkettung durchgeführt. Eine Addition wird nur dann durchgeführt, wenn keiner der Operanden stringartig ist.

Technisch verhält sich der +-Operator folgendermaßen:

- Ist einer seiner Operanden einen Objekt, wird dieser mit dem Objekt/Elementar-Algorithmus umgewandelt, der in § 3.8.3 beschrieben wurde: Date-Objekte werden über ihre `toString()`-Methode umgewandelt, alle anderen Objekte über ihre `valueOf()`-Methode, wenn diese Methode einen elementaren Wert liefert. Die meisten Objekte haben jedoch keine nützliche `valueOf()`-Methode und werden deswegen ebenfalls über `toString()` umgewandelt.
- Ist nach der Objekt/Elementar-Umwandlung einer der Operanden ein String, wird der andere in einen String umgewandelt und eine Stringverkettung durchgeführt.
- Andernfalls werden beide Operanden in Zahlen (oder `NaN`) umgewandelt und eine Addition durchgeführt.

Hier sind einige Beispiele:

```
1 + 2         // => 3: Addition.
"1" + "2"     // => "12": Verkettung.
"1" + 2       // => "12": Verkettung nach Zahl/String-Umwandlung.
1 + {}        // => "1[object Object]": Verkettung nach Objekt/String-Umwandlung.
true + true   // => 2: Addition nach Boolescher-Wert/Zahl-Umwandlung.
2 + null      // => 2: Addition nach Umwandlung von null in 0.
2 + undefined // => NaN: Addition nach Umwandlung von undefined in NaN.
```

Abschließend sollten Sie sich unbedingt merken, dass der +-Operator bei der Verwendung mit Zahlen und Strings seine Assoziativität verlieren kann, d.h., dass das Ergebnis davon abhängen kann, in welcher Reihenfolge die Operationen ausgeführt werden. Ein Beispiel:

```
1 + 2 + " blinde Mäuse";    // => "3 blinde Mäuse"
1 + (2 + " blinde Mäuse");  // => "12 blinde Mäuse"
```

Da die erste Zeile keine Klammern enthält und der +-Operator linksassoziativ ist, werden zunächst die beiden Zahlen addiert, bevor ihre Summe mit dem String verkettet wird. In der zweiten Zeile ändern Klammern die Abfolge dieser Operationen: Die Zahl 2 wird mit dem String verkettet und es wird ein neuer String erzeugt. Dann wird die Zahl 1 mit dem neuen String verkettet, um das endgültige Ergebnis zu erzeugen.

4.8.2 Unäre arithmetische Operatoren

Unäre Operatoren modifizieren den Wert eines einzelnen Operanden, um einen neuen Wert hervorzubringen. In JavaScript haben alle unären Operatoren einen hohen Vorrang und sind rechtsassoziativ. Die arithmetischen unären Operatoren (+, -, ++ und --) wandeln alle ihren einen Operanden bei Bedarf in eine Zahl um. Beachten Sie, dass die Interpunktionszeichen + und – beide als unäre und binäre Operatoren genutzt werden.

Es gibt die folgenden unären arithmetischen Operatoren:

Unäres Plus (+)
: Der unäre Plusoperator wandelt seinen Operanden in eine Zahl (oder `NaN`) um und liefert den umgewandelten Wert. Wird er auf einem Operanden verwendet, der bereits eine Zahl ist, tut er nichts.

Unäres Minus (-)
: Wird - als unärer Operator verwendet, wird der Operand bei Bedarf in eine Zahl umgewandelt, und dann wird das Vorzeichen des Umwandlungsergebnisses geändert.

Inkrement (++)
: Der ++-Operator inkrementiert seinen einen Operanden (d.h., fügt ihm 1 hinzu). Der Operand muss ein Lvalue sein (eine Variable, ein Array-Element oder eine Objekteigenschaft). Der Operator wandelt seinen Operanden bei Bedarf in eine Zahl um, fügt dieser 1 hinzu und weist den inkrementierten Wert wieder der Variablen, dem Element oder der Eigenschaft zu.

  Der Rückgabewert des ++-Operators ist von seiner Stellung in Bezug auf den Operanden abhängig. Steht er vor seinem Operanden – man spricht dann von einem

Prä-Inkrement –, inkrementiert er den Wert und wird dann zum inkrementierten Wert des Operanden ausgewertet. Steht er hinter dem Operanden – dann spricht man von einem Post-Inkrement –, inkrementiert er den Operanden, wird aber zum *nicht inkrementierten* Wert des Operanden ausgewertet. Erwägen Sie die Unterschiede zwischen diesen beiden Codezeilen:

```
var i = 1, j = ++i;    // i und j sind beide 2.
var i = 1, j = i++;    // i ist 2, j ist 1.
```

Beachten Sie, dass der Ausdruck `++x` nicht immer das Gleiche ist wie `x=x+1`. Der `++`-Operator führt nie eine Stringverkettung durch: Er wandelt seinen Operanden immer in eine Zahl um und inkrementiert ihn. Wenn `x` den Stringwert »1« hat, entspricht `++x` der Zahl 2, während `x+1` zum String »11« ausgewertet wird.

Beachten Sie außerdem, dass Sie, aufgrund von JavaScripts automatischer Semikolonergänzung, keinen Zeilenumbruch zwischen den Postinkrementoperator und den ihm vorausgehenden Operanden setzen dürfen. Tun Sie das, behandelt JavaScript den Operanden als eine eigenständige Anweisung und fügt vor ihm ein Semikolon ein.

Dieser Operator wird seinen Erscheinungsformen als Prä- und Postinkrementoperator am häufigsten zur Steuerung eines Zählers eingesetzt, der eine `for`-Schleife (siehe § 5.5.3) steuert.

Dekrement (`--`)

Der `---`Operator erwartet einen Lvalue als Operanden. Er wandelt den Wert seines Operanden in eine Zahl um, zieht 1 von ihm ab und weist den dekrementierten Wert wieder dem Operanden zu. Wie beim `++`-Operator ist der Rückgabewert von der Stellung zum Operanden abhängig. Steht der Operator vor dem Operanden, wird der Operand dekrementiert und der dekrementierte Wert geliefert, steht er hinter dem Operanden, wird der Operand dekrementiert, aber der *nicht dekrementierte* Wert geliefert. Steht der Operator hinter dem Operanden, ist zwischen Operand und Operator kein Zeilenumbruch erlaubt.

4.8.3 Bit-Operatoren

Die Bit-Operatoren führen elementare Manipulationen der Bits in der binären Repräsentation von Zahlen durch. Obwohl sie keine klassischen arithmetischen Operationen erfüllen, werden sie hier unter die arithmetischen Operatoren eingeordnet, weil sie auf numerischen Operanden operieren und einen numerischen Wert liefern. In der JavaScript-Programmierung werden diese Operatoren nicht sehr häufig verwendet. Wenn Sie mit der binären Darstellung dezimaler Ganzzahlen vertraut sind, können Sie diesen Abschnitt wahrscheinlich überspringen. Vier dieser Operanden führen boolesche Algebra auf den einzelnen Bits der Operanden aus und verhalten sich, als wären die Bits des Operanden boolesche Werte (1=wahr, 0=falsch). Die anderen drei Bit-Operatoren werden genutzt, um Bits nach links oder rechts zu verschieben.

Die Bit-Operatoren erwarten ganzzahlige Operanden und verhalten sich, als würden diese Werte als 32-Bit-Ganzzahlwerte und nicht als 64-Bit-Gleitkommawerte repräsentiert. Sie wandeln ihre Operanden bei Bedarf in Zahlen um und zwingen die numerischen Werte dann in ein 32-Bit-Ganzzahlformat, indem alle Bruchteile und Bits nach dem 32. Bit fallen gelassen werden. Die Verschiebungsoperatoren verlangen auf der rechten Seite einen Operanden zwischen 0 und 31. Nachdem dieser Operand in eine vorzeichenlose 32-Bit-Ganzzahl umgewandelt wurde, werden alle Bits nach dem 5. fallen gelassen, um einen Wert im entsprechenden Bereich zu erhalten. Überraschenderweise werden `NaN`, `Infinity` und `-Infinity` alle in 0 umgewandelt, wenn sie als Operanden eines der Bit-Operatoren verwendet werden.

Bitweises UND (`&`)
: Der `&`-Operator führt eine Boolean-UND-Operation auf allen Bits seiner ganzzahligen Operanden durch. Ein Bit im Ergebnis wird nur dann gesetzt, wenn das entsprechende Bit in beiden Operanden gesetzt ist. Beispielsweise wird `0x1234 & 0x00FF` zu `0x0034` ausgewertet.

Bitweises ODER (`|`)
: Der `|`-Operator führt eine boolesche ODER-Operation auf allen Bits seiner ganzzahligen Operanden durch. Ein Bit im Ergebnis ist gesetzt, wenn das entsprechende Bit in einem oder in beiden Operanden gesetzt ist. Beispielsweise wird `0x1234 | 0x00FF` zu `0x12FF` ausgewertet.

Bitweises XOR (`^`)
: Der `^`-Operator führt eine exklusive boolesche ODER-Operation oder XOR-Operation auf den einzelnen Bits seiner ganzzahligen Operanden durch. Exklusiv ODER heißt, dass ein Bit entweder im einen Operanden oder im anderen Operanden wahr ist, aber nicht in beiden. Ein Bit im Ergebnis der Operation ist gesetzt, wenn das entsprechende Bit in einem (aber nicht beiden) Operanden gesetzt ist. Beispielsweise wird `0xFF00 ^ 0xF0F0` zu `0x0FF0` ausgewertet.

Bitweises NICHT (`~`)
: Der `~`-Operator ist ein unärer Operator, der vor einem ganzzahligen Operanden erscheint. Er bewirkt, dass alle Bits im Operanden umgekehrt werden. Die Art, wie Ganzzahlen mit Vorzeichen in JavaScript repräsentiert werden, bewirkt, dass die Anwendung des `~`-Operators auf einen Wert das gleiche Ergebnis hat, als würde sein Vorzeichen geändert und dann 1 vom Zwischenergebnis abgezogen. Beispielsweise wird `~0x0F` zu `0xFFFFFFF0` oder –16 ausgewertet.

Verschiebung nach links (`<<`)
: Der `<<`-Operator verschiebt alle Bits in seinem ersten Operanden um die Anzahl von Stellen nach links, die durch den zweiten Operanden angegeben werden, der eine ganze Zahl zwischen 0 und 31 sein sollte. Beispielsweise wird in der Operation `a << 1` das erste Bit (das Einer-Bit) von `a` zum zweiten Bit (das Zweier-Bit), das zweite Bit von `a` das dritte Bit usw. Für das neue erste Bit wird eine Null eingesetzt, und der Wert des 32. Bits geht verloren. Eine Verschiebung um eine Position nach links entspricht einer

Multiplikation mit 2, eine Verschiebung von zwei Positionen einer Multiplikation mit 4 und so weiter. Beispielsweise wird `7 << 2` zu 28 ausgewertet.

Verschiebung nach rechts mit Vorzeichen (>>)
: Der >>-Operator verschiebt alle Bits in seinem ersten Operanden um die durch den zweiten Operanden angegebenen Stellen nach rechts. (Der zweite Operand ist eine ganze Zahl zwischen 0 und 31.) Bits, die nach rechts hinausgeschoben werden, gehen verloren. Wie die neuen Bits auf der linken Seite gefüllt werden, hängt vom Vorzeichen des ursprünglichen Operanden ab, damit das Vorzeichen im Ergebnis bewahrt bleibt. Ist der erste Operand positiv, werden die neuen hohen Bits mit Nullen aufgefüllt. Ist der erste Operand negativ, werden die neuen hohen Bits mit Einsen aufgefüllt. Eine Verschiebung um eine Position nach rechts entspricht einer ganzzahligen Division durch 2 (wobei der Rest verworfen wird), eine Verschiebung um zwei Stellen einer restlosen Division durch 4 und so weiter. Beispielsweise wird `7 >> 1` zu 3 ausgewertet und `-7 >> 1` zu –4.

Verschiebung nach rechts mit Nullauffüllung (>>>)
: Der >>>-Operator entspricht dem >>-Operator, aber die von links eingeschobenen Bits werden immer mit Null gefüllt – unabhängig davon, welches Vorzeichen der erste Operand hat. Beispielsweise wird `-1 >> 4` zu `-1` ausgewertet und `-1 >>> 4` zu `0xOFFFFFFF`.

4.9 Relationale Ausdrücke

Dieser Abschnitt beschreibt JavaScripts relationale Operatoren. Diese Operatoren prüfen Verhältnisse (wie »gleich«, »kleiner« oder »Eigenschaft von«) zwischen zwei Werten und liefern `true` oder `false`, je nachdem, ob dieses Verhältnis besteht oder nicht. Relationale Ausdrücke werden immer zu einem booleschen Wert ausgewertet, und dieser Wert wird häufig eingesetzt, um den Ablauf der Programmausführung in `if`-, `while`- und `for`-Anweisungen (siehe Kapitel 5) zu steuern. Die nachfolgenden Unterabschnitte dokumentieren die Gleichheits- und Ungleichheitsoperatoren, die Vergleichsoperatoren und die beiden anderen relationalen Operatoren von JavaScript, `in` und `instanceof`.

4.9.1 Gleichheits- und Ungleichheitsoperatoren

Die Operatoren `==` und `===` prüfen, ob zwei Werte gleich sind, nutzen dabei aber zwei unterschiedliche Definitionen von Gleichheit. Beide Operatoren akzeptieren Operanden eines beliebigen Typs, und beide liefern `true`, wenn die Operanden gleich sind, bzw. `false`, wenn sie unterschiedlich sind. Der `===`-Operator wird als strenger Gleichheitsoperator (manchmal auch Identitätsoperator) bezeichnet. Er prüft die »Identität« seiner Operanden gemäß einer strengen Definition von Gleichheit. Der `==`-Operator heißt einfach Gleichheitsoperator und prüft die »Gleichheit« seiner Operanden gemäß einer entspannteren Definition von Gleichheit, die Typumwandlungen gestattet.

JavaScript kennt `=`-, `==`- und `===`-Operatoren. Es ist wichtig, dass Ihnen die Unterschiede zwischen dem Zuweisungs-, dem Gleichheits- und dem Identitätsoperator klar sind, und

achten Sie beim Programmieren sorgfältig darauf, dass Sie tatsächlich den benötigen Operator verwenden! Obgleich es verführerisch sein mag, bei allen drei Operatoren jeweils »gleich« zu lesen, können Sie sich wahrscheinlich Irritationen ersparen, wenn Sie für `=` »wird zugewiesen«, für `==` »ist gleich« und für `===` »ist identisch mit« lesen.

Die Operatoren `!=` und `!==` testen das genaue Gegenteil von `==` und `===`. Der `!=`-Ungleichheitsoperator liefert `false`, wenn zwei Werte gemäß `==` gleich sind, `true` andernfalls. Der `!==`-Operator liefert `false`, wenn zwei Werte streng genommen gleich sind, und `true`, wenn das nicht der Fall ist. Wie Sie in § 4.10 sehen werden, berechnet der `!`-Operator die boolesche NICHT-Operation. Man kann sich also leicht merken, dass `!=` und `!==` für »nicht gleich« und »im strengen Sinne nicht gleich« stehen.

Wie in § 3.7 erwähnt wurde, werden JavaScript-Objekte anhand der Referenz verglichen, nicht anhand des Werts. Ein Objekt ist nur mit sich selbst identisch, mit keinem anderen Objekt. Zwei unabhängige Objekte, die die gleiche Menge von Eigenschaften mit gleichen Namen und Werten haben, sind nicht gleich. Zwei Arrays, die die gleichen Elemente in der gleichen Reihenfolge haben, sind nicht gleich.

Der strenge Gleichheitsoperator `===` wertet seine Operanden aus und vergleicht die beiden Werte dann folgendermaßen, ohne eine Typumwandlung durchzuführen:

- Die Werte sind nicht gleich, wenn sie unterschiedliche Typen haben.
- Die Werte sind gleich, wenn beide `null` oder beide `undefined` sind.
- Die Werte sind gleich, wenn beide den booleschen Wert `true` oder beide den booleschen Wert `false` haben.
- Die Werte sind nicht gleich, wenn einer oder beide `NaN` sind. Der `NaN`-Wert ist keinem anderen Wert gleich, nicht einmal sich selbst! Wenn Sie prüfen wollen, ob ein `x` gleich `NaN` ist, nutzen Sie `x !== x`. `NaN` ist der einzige Wert für `x`, für den dieser Ausdruck wahr ist.
- Die Werte sind gleich, wenn beide Zahlen sind und beide den gleichen Betrag haben. Sie sind ebenfalls gleich, wenn einer der Werte `0` und der andere `-0` ist.
- Die Werte sind gleich, wenn beide Strings sind und genau die gleichen 16-Bit-Werte (siehe den Kasten in § 3.2) an genau den gleichen Positionen enthalten. Sie sind nicht gleich, wenn sie eine unterschiedliche Länge oder einen unterschiedlichen Inhalt haben. Zwei Strings können die gleiche Bedeutung und die gleiche Darstellung haben, aber dennoch mit unterschiedlichen Folgen von 16-Bit-Werten kodiert sein. JavaScript führt keine Unicode-Normalisierung durch, und derartige Strings sind weder für den `===`- noch für den `==`-Operator gleich. Andere Möglichkeiten zum Vergleich von Strings finden Sie unter `String.localeCompare()` in Teil III.
- Die Werte sind gleich, wenn beide auf dasselbe Objekt, dasselbe Array oder dieselbe Funktion verweisen. Verweisen sie auf unterschiedliche Objekte, sind sie nicht gleich, selbst wenn die Eigenschaften der Objekte identisch sind.

Der Gleichheitsoperator == verhält sich wie der strenge Gleichheitsoperator, ist aber weniger streng. Haben die Werte der beiden Operanden nicht den gleichen Typ, versucht er, die Typen umzuwandeln, und vergleicht dann die umgewandelten Werte:

- Haben zwei Werte den gleichen Typ, werden sie wie oben beschrieben auf strenge Gleichheit geprüft. Sind sie im strengen Sinne gleich, sind sie gleich. Sind sie nicht im strengen Sinne gleich, sind sie nicht gleich.
- Haben die beiden Werte nicht den gleichen Typ, kann der ==-Operator sie dennoch als gleich betrachten. Er nutzt die folgenden Regeln und Typumwandlungen, um die Gleichheit zu prüfen:
  - Die Werte sind gleich, wenn einer der Werte `null` und der andere `undefined` ist.
  - Ist einer der Werte eine Zahl, der andere ein String, wird der String in eine Zahl umgewandelt und der Vergleich dann mit dem umgewandelten Wert durchgeführt.
  - Ist einer der Werte `true`, wird er in 1 umgewandelt und der Vergleich dann erneut ausgeführt. Ist einer der Werte `false`, wird er in 0 umgewandelt und der Vergleich dann erneut durchgeführt.
  - Ist einer der Werte ein Objekt, der andere eine Zahl oder ein String, wird das Objekt anhand des unter § 3.8.3 beschriebenen Algorithmus umgewandelt und der Vergleich dann erneut probiert. Objekte werden entweder über Ihre `toString()`- oder ihre `valueOf()`-Methode in einen String umgewandelt. Die eingebauten Klassen des Sprachkerns von JavaScript versuchen eine `valueOf()`-Umwandlung, bevor die `toString()`-Umwandlung genutzt wird. Nur die Klasse Date führt direkt eine `toString()`-Umwandlung durch. Objekte, die nicht Teil des Sprachkerns von JavaScript sind, können sich selbst auf implementierungsspezifische Weise in elementare Werte umwandeln.
  - Alle anderen Kombinationen von Werten sind nicht gleich.

Schauen Sie sich als Beispiel für die Prüfung auf Gleichheit den folgenden Vergleich an:

```
"1" == true
```

Dieser Ausdruck wird zu `true` ausgewertet – diese beiden so unterschiedlich aussehenden Werte sind also in der Tat gleich. Erst wird der boolesche Wert `true` in die Zahl 1 umgewandelt und der Vergleich erneut durchgeführt. Dann wird der String `"1"` in die Zahl 1 umgewandelt. Da jetzt beide Werte gleich sind, liefert der Vergleich `true`.

4.9.2 Vergleichsoperatoren

Die Vergleichsoperatoren prüfen die relative Abfolge (numerisch oder alphabetisch) ihrer beiden Operanden:

Kleiner als (<)
: Der <-Operator wird zu `true` ausgewertet, wenn sein erster Operand kleiner ist als der zweite, andernfalls mit `false`.

Größer als (>)
: Der >-Operator wird zu `true` ausgewertet, wenn sein erster Operand größer ist als der zweite, andernfalls zu `false`.

Kleiner oder gleich (<=)
: Der <=-Operator wird zu `true` ausgewertet, wenn sein erster Operand kleiner oder gleich dem zweiten Operanden ist, andernfalls zu `false`.

Größer oder gleich (>=)
: Der >=-Operator wird zu `true` ausgewertet, wenn sein erster Operand größer oder gleich dem zweite Operanden ist, andernfalls zu `false`.

Die Operanden dieser Vergleichsoperatoren können einen beliebigen Typ haben. Vergleiche können jedoch nur auf Zahlen und Strings durchgeführt werden. Operanden, die weder Strings noch Zahlen sind, werden also umgewandelt. Vergleich und Umwandlung erfolgen folgendermaßen:

- Wird einer der Operanden zu einem Objekt ausgewertet, wird dieses Objekt so in einen elementaren Wert umgewandelt, wie es am Ende von § 3.8.3 beschrieben wurde: Liefert die `valueOf()`-Methode einen elementaren Wert, wird dieser Wert genommen, andernfalls der Rückgabewert der `toString()`-Methode.
- Sind nach eventuell erforderlichen Umwandlungen von Objekten in elementare Werte beide Operanden Strings, werden die beiden Strings nach ihrer alphabetischen Reihenfolge verglichen. Die »alphabetische Reihenfolge« wird dabei durch die numerische Abfolge der 16-Bit-Unicode-Werte bestimmt, die die Strings bilden.
- Ist nach den Umwandlungen von Objekten in elementare Werte mindestens einer der Operanden kein String, werden beide Operanden in Zahlen umgewandelt und numerisch verglichen. `0` und `-0` werden als gleich betrachtet. `Infinity` ist größer als jede Zahl außer sich selbst, und `-Infinity` ist kleiner als jede Zahl außer sich selbst. Wenn einer der Operanden `NaN` ist (oder in diesen Wert umgewandelt wird) liefern die Vergleichsoperatoren immer `false`.

Denken Sie daran, dass JavaScript-Strings Folgen von 16-Bit-Ganzzahlwerten sind und dass Stringvergleiche nur numerische Vergleiche der Werte in zwei Strings sind. Die numerische Kodierungsfolge, die Unicode definiert, entspricht nicht notwendigerweise der traditionellen Sortierfolge, die in einer bestimmten Sprache oder einem bestimmten Kulturraum verwendet wird. Beachten Sie insbesondere, dass Stringvergleiche Groß-/Kleinschreibung berücksichtigen und dass alle ASCII-Großbuchstaben »kleiner als« alle ASCII-Kleinbuchstaben sind. Diese Regel kann zu verwirrenden Ergebnissen führen, wenn Sie auf dieses Verhalten nicht eingestellt sind. Beispielsweise kommt für den <-Operator der String »Zoo« vor dem String »aal«.

Einen robusteren Algorithmus für Stringvergleiche finden Sie bei der Methode `String.localeCompare()`, die Locale-spezifische Definitionen der alphabetischen Reihenfolge berücksichtigt. Wollen Sie Vergleich ohne Berücksichtigung von Groß-/Kleinschreibung durchführen, müssen Sie alle Strings zunächst in Groß- oder Kleinbuchstaben umwandeln, indem Sie `String.toLowerCase()` bzw. `String.toUpperCase()` einsetzen.

Der +-Operator und die Vergleichsoperatoren verhalten sich bei Zahl- und Stringoperanden unterschiedlich. + bevorzugt Strings: Es wird eine Verkettung durchgeführt, wenn einer der Operanden ein String ist. Die Vergleichsoperatoren bevorzugen Zahlen und führen nur dann einen Stringvergleich durch, wenn beide Operanden Strings sind:

```
1 + 2        // Addition. Ergebnis ist 3.
"1" + "2"    // Verkettung. Ergebnis ist "12".
"1" + 2      // Verkettung. 2 wird in "2" umgewandelt. Ergebnis ist "12".
11 < 3       // Numerischer Vergleich. Ergebnis ist false.
"11" < "3"   // Stringvergleich. Ergebnis ist true.
"11" < 3     // Numerischer Vergleich. "11" wird in 11 umgewandelt. Ergebnis ist false.
"eins" < 3   // Numerischer Vergleich. "eins" wird in NaN umgewandelt. Ergebnis ist
             // false.
```

Beachten Sie außerdem, dass die Operatoren <= (kleiner gleich) und >= (größer gleich) sich nicht auf den Gleichheits- oder Identitätsoperator stützen, wenn sie prüfen, ob zwei Werte »gleich« sind. Stattdessen ist Kleiner-gleich einfach als »nicht größer als« und Größer-gleich als »nicht kleiner als« definiert. Die einzige Ausnahme tritt ein, wenn einer der Operanden `NaN` ist (oder darin umgewandelt wird). In diesem Fall liefern alle vier Vergleichsoperatoren `false`.

4.9.3 Der in-Operator

Der `in`-Operator erwartet auf seiner linken Seite einen Operanden, der ein String ist oder in einen String umgewandelt werden kann, und auf seiner rechten Seite einen Operanden, der ein Objekt ist. Er wird zu `true` ausgewertet, wenn der Wert auf der linken Seite der Name einer Eigenschaft des Objekts auf der rechten Seite ist. Zum Beispiel:

```
var point = { x:1, y:1 };  // Definiert ein Objekt
"x" in point               // => true: Objekt hat eine Eigenschaft namens "x".
"z" in point               // => false: Objekt hat keine Eigenschaft namens "z".
"toString" in point        // => true: Objekt erbt toString()-Methode.

var data = [7,8,9];        // Ein Array mit den Elementen 0, 1 und 2.
"0" in data                // => true: Array hat Element "0".
1 in data                  // => true: Zahlen werden in Strings umgewandelt.
3 in data                  // => false: Kein Element 3.
```

4.9.4 Der instanceof-Operator

Der `instanceof`-Operator erwartet auf der linken Seite einen Operanden, der ein Objekt ist, und auf der rechten Seite einen Operanden, der eine Klasse von Objekten angibt. Der Operator wird mit `true` ausgewertet, wenn das Objekt auf der linken Seite eine Instanz der Klasse auf der rechten Seite ist, andernfalls mit `false`. Kapitel 9 erläutert, dass in JavaScript Klassen von Objekten durch die Konstruktorfunktion definiert werden, die sie initialisiert. Der rechte Operand für den `instanceof`-Operator sollte also eine Funktion sein. Hier sind einige Beispiele:

```
var d = new Date();  // Erstellt ein neues Objekt mit dem Date()-Konstruktor.
d instanceof Date;   // true; d wurde mit Date() erstellt.
d instanceof Object; // true; alle Objekte sind Instanzen von Object.
d instanceof Number; // false; d ist kein Number-Objekt.
var a = [1, 2, 3];   // Erstellt ein Array mit der Array-Literal-Syntax.
a instanceof Array;  // true; a ist ein Array.
a instanceof Object; // true; alle Arrays sind Objekte.
a instanceof RegExp; // false; Arrays sind kein regulären Ausdrücke.
```

Beachten Sie, dass alle Objekte Instanzen von `Object` sind. `instanceof` betrachtet die »Oberklassen«, wenn es entscheidet, ob ein Objekt eine Instanz einer Klasse ist. Ist der linksseitige Operand von `instanceof` kein Objekt, liefert `instanceof` `false`. Ist der rechtsseitige Operand keine Funktion, löst es einen `TypeError` aus.

Wenn Sie verstehen wollen, wie der `instanceof`-Operator funktioniert, müssen Sie die »Prototypkette« verstehen. Das ist der Vererbungsmechanismus von JavaScript, der in § 6.2.2 beschrieben wird. Zur Auswertung des Ausdrucks `o instanceof f` wertet JavaScript `f.prototype` aus und sucht dann nach diesem Wert in der Prototypkette von `o`. Findet es ihn, ist `o` eine Instanz von `f` (oder einer Oberklasse von `f`), und der Operator liefert `true`. Ist `f.prototype` keiner der Werte in der Prototypkette von `o`, ist `o` keine Instanz von `f`, und `instanceof` liefert `false`.

4.10 Logische Ausdrücke

Die logischen Operatoren `&&`, `||` und `!` führen boolesche Algebra durch und werden häufig gemeinsam mit den relationalen Operatoren genutzt, um zwei relationale Ausdrücke zu einem komplexeren Ausdruck zu vereinen. Diese Operatoren werden in den nachfolgenden Unterabschnitten beschrieben. Zum besseren Verständnis sollten Sie sich gegebenenfalls noch einmal das Konzept »wahrer« und »falscher« Werte ansehen, das in § 3.3 eingeführt wurde.

4.10.1 Logisches UND (&&)

Der `&&`-Operator kann auf drei verschiedenen Ebenen betrachtet werden. Auf der einfachsten Stufe, wenn er mit booleschen Operanden verwendet wird, führt `&&` einfach die boolesche UND-Operation auf den beiden Werten durch. Der Operator liefert nur dann `true`, wenn der erste *und* der zweite Operand `true` sind. Wenn einer oder beide Operanden `false` sind, liefert er `false`.

`&&` wird häufig zum Verbinden zweier relationaler Ausdrücke genutzt:

```
x == 0 && y == 0   // true, wenn x und y beide 0 sind.
```

Da relationale Ausdrücke immer zu `true` oder `false` ausgewertet werden, liefert auch der Operator selbst `true` oder `false`, wenn er auf diese Weise verwendet wird. Relationale Operatoren haben einen höheren Vorrang als `&&` (und `||`), Ausdrücke wie dieser können also problemlos ohne Klammern geschrieben werden.

Aber && verlangt nicht, dass seine Operanden boolesche Werte sind. Erinnern Sie sich, dass alle JavaScript-Werte entweder »wahr« oder »falsch« sind. (Mehr Informationen finden Sie in § 3.3. Die falschen Werte sind `false`, `null`, `undefined`, `0`, `-0`, `NaN` und `""`. Alle anderen Werte, Objekte eingeschlossen, sind wahr.) In zweiter Hinsicht kann && also als boolesches UND für wahre und falsche Werte verstanden werden. Sind beide Operanden wahre Werte, liefert der Operator einen wahren Wert. Andernfalls, d.h., wenn mindestens einer der Operanden ein falscher Wert ist, liefert er einen falschen Wert. In JavaScript können alle Ausdrücke oder Anweisungen, die einen booleschen Wert erwarten, mit wahren und falschen Werten umgehen. Deswegen verursacht der Umstand, dass && nicht immer `true` oder `false` liefert, in der Praxis keinerlei Probleme.

Beachten Sie, dass die Beschreibung oben sagt, dass der Operator einen »einen wahren Wert« oder »einen falschen Wert« liefert, aber nicht angibt, was dieser Wert tatsächlich ist. Dazu müssen wir && in dritter und letzter Hinsicht beschreiben. Der Operator beginnt damit, dass er seinen ersten Operanden, den Ausdruck zu seiner Linken, auswertet. Ist der Wert dieses Operanden ein falscher Wert, muss auch der Wert des gesamten Ausdrucks ein falscher Wert sein. && liefert deswegen einfach den Wert auf der linken Seite und wertet den Ausdruck auf der rechten Seite nicht einmal aus.

Ist der Wert auf der linken Seite hingegen ein wahrer Wert, hängt der Wert des gesamten Ausdrucks vom Wert auf der rechten Seite ab. Ist der Wert auf der rechten Seite ein wahrer Wert, muss der gesamte Ausdruck ein wahrer Wert sein; ist er ein falscher Wert, so ist auch der gesamte Ausdruck falsch. Wenn der Wert auf der linken Seite ein wahrer Wert ist, wertet der &&-Operator also einfach den Wert auf der rechten Seite aus und liefert ihn zurück:

```
var o = { x : 1 };
var p = null;
o && o.x     // => 1: o ist wahr, es wird also der Wert von o.x geliefert.
p && p.x     // => null: p ist falsch und wird geliefert; p.x wird nicht ausgewertet.
```

Es ist wichtig, dass Sie begreifen, dass der Operand auf der rechten Seite von && manchmal ausgewertet wird, manchmal nicht. Im Code oben war die Variable `p` auf `null` gesetzt. Der Ausdruck `p.x` hätte, wäre er ausgewertet worden, zu einem TypeError geführt. Aber der Code nutzt && auf idiomatische Weise, damit `p.x` nur ausgewertet wird, wenn `p` wahr ist – nicht `null` oder `undefined`.

Das Verhalten von && wird gelegentlich als »Kurzschlussverhalten« bezeichnet, und Sie können gelegentlich auf Code stoßen, der es absichtlich einsetzt, um Code bedingt auszuführen. Die folgenden beiden Zeilen JavaScript-Code haben beispielsweise die gleiche Wirkung:

```
if (a == b) stop();   // stop() nur aufrufen, wenn a == b.
(a == b) && stop();   // Macht das Gleiche.
```

Im Allgemeinen müssen Sie aufpassen, wenn Sie auf der rechten Seite von && Ausdrücke mit Seiteneffekten angeben (Zuweisung, Inkrement, Dekrement, Funktionsaufruf). Ob diese Seiteneffekte eintreten, hängt vom Wert auf der linken Seite ab.

Trotz der einigermaßen komplexen Funktionsweise dieses Operators wird er am häufigsten als einfacher boolescher Operator eingesetzt, der mit wahren und falschen Werten umgehen kann.

4.10.2 Logisches ODER (||)

Der ||-Operator führt auf seinen beiden Operanden eine boolesche ODER-Operation durch. Sind beide Operanden wahre Werte, liefert er einen wahren Wert; sind beide Operanden falsche Werte, liefert er einen falschen.

Obwohl der ||-Operator in der Regel meist einfach als boolescher ODER-Operator zum Einsatz kommt, hat er wie && ein komplexeres Verhalten. Er beginnt damit, dass er seinen ersten Operanden, den Operanden auf seiner linken Seite, auswertet. Ist der Wert dieses Operanden ein wahrer Wert, liefert er diesen wahren Wert. Andernfalls wertet er seinen zweiten Operanden, den Ausdruck auf seiner rechten Seite, aus, und liefert der Wert dieses Ausdrucks.

Wie beim &&-Operator sollten Sie auf der rechten Seite Operanden vermeiden, die Seiteneffekte haben, es sei denn, Sie haben explizit vor, den Umstand auszunutzen, dass der Ausdruck auf der rechten Seite eventuell nicht ausgewertet wird.

Eine idiomatische Verwendungsweise dieses Operators ist sein Einsatz zur Auswahl des ersten wahren Wertes in einer Menge von Alternativen:

```
// max_width nutzen, wenn es definiert ist. Andernfalls einen Wert
// im Objekt preferences nachschlagen.
// Wenn es auch den nicht gibt, auf eine Konstante zurückgreifen.
var max = max_width || preferences.max_width || 500;
```

Dieses Konstrukt wird häufig im Codeinhalt von Funktionen genutzt, um Standardwerte für Parameter anzugeben:

```
// Die Eigenschaften von o in p kopieren und p liefern.
function copy(o, p) {
   p = p || {};  // Wurde für p kein Objekt übergeben, ein neu erstelltes nutzen.
   // Inhalt der Funktion
}
```

4.10.3 Logisches NICHT (!)

Der !-Operator ist ein unärer Operator, der vor einem einzigen Operanden steht. Sein Zweck ist es, den booleschen Wert seines Operanden zu negieren. Ist x ein wahrer Wert, wird !x zu false ausgewertet. Ist x ein falscher Wert, wird !x zu true ausgewertet.

Im Unterschied zu && und || wandelt der !-Operator seinen Operanden in einen booleschen Wert um (und nutzt dabei die in Kapitel 3 beschriebenen Regeln), bevor er den konvertierten Wert nutzt. Das heißt, dass ! immer true oder false liefert und dass Sie deswegen einen beliebigen Wert x in den entsprechenden booleschen Wert umwandeln können, indem Sie diesen Operator zweimal anwenden: !!x (siehe § 3.8.2).

Da ! ein unärer Operator ist, hat er hohen Vorrang und bindet stark. Wenn Sie den Wert eines Ausdrucks wie p && q invertieren wollen, müssen Sie deswegen Klammern nutzen: !(p && q). Es lohnt sich, zwei Theoreme der booleschen Algebra festzuhalten, die wir mit JavaScript-Syntax ausdrücken können:

```
// Diese beiden Äquivalenzen gelten für alle Werte von p und q.
!(p && q) === !p || !q
!(p || q) === !p && !q
```

4.11 Zuweisungsausdrücke

JavaScript nutzt den =-Operator, um einer Variablen oder Eigenschaft einen Wert zuzuweisen. Zum Beispiel:

```
i = 0           // Setzt die Variable i auf 0.
o.x = 1         // Setzt die Eigenschaft x von Objekt o auf 1.
```

Der =-Operator erwartet, dass der Operand auf seiner linken Seite ein Lvalue ist: eine Variable oder eine Objekteigenschaft (oder ein Array-Element). Als rechtsseitigen Operanden erwartet er einen beliebigen Wert eines beliebigen Typs. Der Wert eines Zuweisungsausdrucks ist der Wert seines rechtsseitigen Operanden. Als Seiteneffekt weist der =-Operator den Wert auf seiner rechten Seite der Variablen oder Eigenschaft auf seiner linken Seite zu, damit spätere Referenzen auf die Variable oder Eigenschaft zu diesem Wert ausgewertet werden.

Obwohl Zuweisungsausdrücke in der Regel recht einfach sind, können Sie gelegentlich darauf stoßen, dass der Wert eines Zuweisungsausdrucks als Teil eines größeren Ausdrucks genutzt wird. Beispielsweise können Sie einen Wert mit Code wie dem folgenden in einem Ausdruck testen und zuweisen:

```
(a = b) == 0
```

Wenn Sie derartige Dinge tun, sollten Sie darauf achten, dass Ihnen der Unterschied zwischen den Operatoren = und == vollkommen klar ist! Beachten Sie, dass = einen sehr geringen Vorrang hat und deswegen meist Klammern erforderlich sind, wenn der Wert einer Zuweisung in einem größeren Ausdruck verwendet werden soll.

Der Zuweisungsoperator ist rechtsassoziativ. Das bedeutet, dass mehrere Zuweisungsoperatoren in einem Ausdruck von rechts nach links ausgewertet werden. Sie können also Code wie diesen schreiben, um mehreren Variablen den gleichen Wert zuzuweisen:

```
i = j = k = 0;       // Initialisiert 3 Variablen auf 0.
```

4.11.1 Zuweisung mit Operation

Neben dem normalen =-Zuweisungsoperator unterstützt JavaScript eine Reihe weiterer Zuweisungsoperatoren, die Kurzformen zur Kombination einer Zuweisung mit einer anderen Operation darstellen. Beispielsweise führt der +=-Operator eine Addition und eine Zuweisung aus. Der Ausdruck

```
total += sales_tax
```

ist gleichwertig zu diesem:

```
total = total + sales_tax
```

Wie man erwarten könnte, kann der +=-Operator mit Strings und Zahlen umgehen. Bei numerischen Operanden führt er Addition und Zuweisung aus, bei Stringoperanden, Verkettung und Zuweisung.

Ähnliche Operatoren sind unter anderem -=, *=, &= und so weiter. Tabelle 4-2 führt alle auf.

Tabelle 4-2: Zuweisungsoperatoren

Operator	Beispiel	Äquivalent
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b
>>>=	a >>>= b	a = a >>> b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b

In den meisten Fällen ist der Ausdruck

```
a op= b
```

(*op* ist ein beliebiger Operator) zum Ausdruck

```
a = a op b
```

äquivalent. In der ersten Zeile wird der Ausdruck a einmal ausgewertet, in der zweiten zweimal. Die beiden Fälle unterscheiden sich also nur dann, wenn a Seiteneffekte wie einen Funktionsaufruf oder eine Inkrementierungsoperation einschließt. Die beiden folgenden Zuweisungen sind beispielsweise nicht gleich:

```
data[i++] *= 2;
data[i++] = data[i++] * 2;
```

4.12 Auswertungsausdrücke

Wie viele interpretierte Sprachen hat JavaScript die Fähigkeit, Strings mit JavaScript-Quellcode zu interpretieren und auszuwerten, um einen Wert hervorzubringen. In JavaScript erfolgt das mit der globalen Funktion `eval()`:

```
eval("3+2")    // => 5
```

Die dynamische Auswertung von Strings mit Quellcode ist eine mächtige Spracheinrichtung, die in der Praxis fast nie notwendig ist. Wenn Sie auf `eval()` zurückgreifen, sollten Sie sorgfältig überlegen, ob Sie es tatsächlich benötigen.

Die nachfolgenden Unterabschnitte erläutern die Grundlagen der Verwendung von `eval()` und erläutern dann zwei eingeschränkte Versionen, die geringere Auswirkungen auf den Optimizer haben.

Ist eval() eine Funktion oder ein Operator?

`eval()` ist eine Funktion, wird aber trotzdem in diesem Kapitel über Ausdrücke behandelt, weil es eigentlich ein Operator hätte sein sollen. Die frühesten Versionen der Sprache definierten eine `eval()`-Funktion, und seitdem haben Sprachentwickler und Interpreterautoren sie mit Einschränkungen versehen, die sie immer mehr einem Operator gleichen lassen. Moderne JavaScript-Interpreter führen umfangreiche Codeanalysen und -optimierungen durch. `eval()` aber hat das Problem, dass der Code, den es auswertet, in der Regel nicht analysierbar ist. Allgemein gesagt: Ruft eine Funktion `eval()` auf, kann der Interpreter diese Funktion nicht mehr optimieren. Daran, dass `eval()` als Funktion definiert ist, ist problematisch, dass man ihr einen anderen Namen geben kann:

```
var f = eval;
var g = f;
```

Wenn das zulässig ist, kann der Interpreter keine Funktion mehr verlässlich optimieren, die `g()` aufruft. Dieses Problem könnte man vermeiden, wäre `eval` ein Operator (und ein reserviertes Wort). Wir werden unten (in § 4.12.2 und § 4.12.3) die Einschränkungen kennenlernen, denen man `eval()` unterworfen hat, um es operatorähnlicher zu machen.

4.12.1 eval()

`eval()` erwartet ein Argument. Übergeben Sie einen anderen Wert als einen String, liefert es einfach diesen Wert zurück. Übergeben Sie einen String, versucht es, diesen String als JavaScript-Code zu parsen, und löst einen SyntaxError aus, wenn dieser Versuch scheitert. Kann es den String erfolgreich parsen, wertet es den Code aus und liefert den Wert des letzten Ausdrucks oder der letzten Anweisung im String oder `undefined`, wenn der letzte Ausdruck bzw. die letzte Anweisung keinen Wert hatte. Löst der String eine Exception aus, leitet `eval()` diese Exception weiter.

Das Wesentliche bei `eval()` (wenn es auf diese Weise aufgerufen wird) ist, dass es die Variablenumgebung des Codes nutzt, der es aufruft. Das heißt, dass es die Werte von Variablen auf gleiche Weise nachschlägt und neue Variablen und Funktionen auf gleiche Weise definiert wie lokaler Code. Definiert eine Funktion eine lokale Variable `x` und ruft dann `eval("x")`auf, erhält sie den Wert der lokalen Variablen. Ruft sie `eval("x=1")` auf, ändert sie den Wert der lokalen Variablen. Und wenn die Funktion `eval("var y = 3;")` aufruft, hat sie eine neue lokale Variable `y` definiert. Mit Code wie dem folgenden kann eine Funktion auf vergleichbare Weise auch eine lokale Funktion deklarieren:

```
eval("function f() { return x+1; }");
```

Rufen Sie `eval()` aus Code der obersten Ebene auf, operiert es natürlich auf globalen Variablen und globalen Funktionen.

Beachten Sie, dass der Code-String, den Sie an `eval()` übergeben, eine eigenständige syntaktische Einheit sein muss. Sie können es nicht nutzen, um Codefragmente in eine Funktion einzusetzen. Beispielsweise ist der Ausdruck `eval("return;")` sinnlos, weil `return` nur in Funktionen erlaubt ist und der Umstand, dass der ausgewertete String die gleiche Variablenumgebung nutzt wie die aufrufende Funktion, den String nicht zu einem Teil der Funktion macht. Könnte Ihr String ein eigenständigs Skript darstellen (und wenn es nur ein ganz kurzes wie `x=0` wäre), kann er an `eval()` übergeben werden. Andernfalls löst `eval()` einen SyntaxError aus.

4.12.2 Globales eval()

Das, was `eval()` für JavaScript-Optimierer so problematisch macht, ist seine Fähigkeit, lokale Variablen zu ändern. Um das zu umgehen, führen Interpreter einfach weniger Optimierungen auf Funktionen aus, die `eval()` aufrufen. Aber was soll ein JavaScript-Interpreter tun, wenn ein Skript ein Alias für `eval()` definiert und die Funktion einfach unter einem anderen Namen aufruft? Damit sich JavaScript-Implementierer über dieses Problem nicht den Kopf zerbrechen mussten, legte der ECMAScript 3-Standard fest, dass Interpreter das nicht zulassen müssen. Wenn die Funktion `eval()` unter einem anderen Namen als »eval« aufgerufen wird, war es ihnen gestattet, einen EvalError auszulösen.

In der Praxis aber machten die meisten Implementierer etwas anderes. Wenn `eval()` unter anderem Namen aufgerufen wurde, werteten sie den String aus, als wäre es Code der obersten Ebene. Der ausgewertete Code konnte neue globale Variablen oder Funktionen deklarieren und globale Variablen setzen, konnte aber keine Variablen in der aufrufenden Funktion mehr nutzen oder verändern und kam so den lokalen Optimierungen nicht mehr ins Gehege.

ECMAScript 5 erklärt EvalError für veraltet und standardisiert das De-facto-Verhalten von `eval()`. Ein »direktes Eval« ist ein Aufruf der `eval()`-Funktion mit einem Ausdruck, der den genauen, nicht qualifizierten Namen »eval« nutzt (der sich damit immer mehr wie ein reserviertes Wort anfühlt). Direkte Aufrufe von `eval()` nutzen die Variablenumgebung des aufrufenden Kontexts. Jeder andere Aufruf – ein indirekter Aufruf – nutzt das globale Objekt als Variablenumgebung und kann lokale Variablen oder Funktionen nicht lesen, schreiben oder definieren. Der folgende Code führt dies vor:

```
var geval = eval;               // Ein anderer Name führt zu einem globalen Eval.
var x = "global", y = "global"; // Zwei globale Variablen.
function f() {                  // Die Funktion führt eval() local aus.
    var x = "lokal";            // Eine lokale Variable definieren.
    eval("x += 'geändert';");   // Das direkte Eval setzt die lokale Variable.
    return x;                   // Den geänderten lokalen Wert zurückliefern.
}
```

```
function g() {                    // Diese Funktion führt ein globales Eval aus.
    var y = "lokal";              // Eine lokale Variable.
    geval("y += 'geändert';");    // Ein indirektes Eval setzt die globale Variable.
    return y;                     // Liefert den unveränderten lokalen Wert.
}
console.log(f(), x); // Lokale Variable geändert: ergibt "lokalgeändert global":
console.log(g(), y); // Globale Variable geändert: ergibt "lokal globalgeändert":
```

Beachten Sie, dass diese Möglichkeit, ein globales Eval durchzuführen, nicht bloß eine Verbeugung vor den Anforderungen des Optimierers ist. Es ist tatsächlich eine unglaublich nützliche Einrichtung, die es Ihnen ermöglicht, Strings mit Code auszuführen, als wären es unabhängige Skripten der obersten Ebene. Wie bereits zu Anfang dieses Abschnitts erwähnt wurde, kommt es nur äußerst selten vor, dass man Strings mit Code auswerten muss. Aber wenn Sie feststellen, dass Sie dazu gezwungen sind, ist es eher wahrscheinlich, dass Sie ein globales Eval brauchen und als ein lokales.

Vor dem IE9 verhielt sich der IE anders als anderen Browser: Er führt kein globales Eval aus, wenn `eval()` unter einem anderen Namen aufgerufen wird. (Er löst auch keinen EvalError aus, sondern macht brav ein lokales Eval.) Aber der IE definiert eine globale Funktion namens `execScript()`, die ihr Stringargument ausführt, als wäre es ein Top-Level-Skript. (Im Unterschied zu `eval()` liefert `execScript()` jedoch immer `null`.)

4.12.3 eval() im Strict-Modus

Der ECMAScript 5-Strict-Modus (siehe § 5.7.3) unterwirft das Verhalten der Funktion `eval()` und selbst die Verwendung des Bezeichners »eval« weiteren Einschränkungen. Wird `eval()` aus Code im Strict-Modus aufgerufen oder beginnt der auszuwertende Code selbst mit der Direktive »use strict«, führt `eval()` ein lokales Eval mit einer privaten Variablenumgebung aus. Das bedeutet, dass so ausgewerteter Code im Strict-Modus lokale Variablen abfragen und setzen, aber keine neuen Variablen oder Funktionen im lokalen Geltungsbereich definieren kann.

Außerdem macht der Strict-Modus `eval()` noch operatorartiger, indem er »eval« praktisch zu einem reservierten Wort macht. Man darf die `eval()`-Funktion nicht mit einem neuen Wert überschreiben, und man darf keine Variable, keine Funktion, keinen Funktions- oder Catch-Blockparameter mit dem Namen »eval« deklarieren.

4.13 Verschiedene Operatoren

JavaScript unterstützt eine Reihe von Operatoren zu anderen Zwecken, die in den folgenden Abschnitten beschrieben werden.

4.13.1 Der Bedingungsoperator (?:)

Der Bedingungsoperator ist der einzige Ternäroperator (drei Operanden) in JavaScript und wird in der Tat gelegentlich einfach als »der Ternäroperator« bezeichnet. Dieser Operator wird gelegentlich in der Form `?:` geschrieben, obwohl er im Code nicht ganz in dieser Form

erscheint. Weil dieser Operator drei Operatoren hat, kommt der erste vor das ?, der zweite zwischen ? und : und der dritte nach dem :. Er wird folgendermaßen verwendet:

```
x > 0 ? x : -x     // Der Absolutwert von x
```

Die Operanden des Bedingungsoperators können einen beliebigen Typ haben. Der erste Operand wird ausgewertet und als boolescher Wert interpretiert. Ist der erste Operand ein wahrer Wert, wird der zweite Operand ausgewertet und sein Wert zurückgeliefert. Ist der erste Operand hingegen ein falscher Wert, wird der dritte Operand ausgewertet und sein Wert zurückgeliefert. Es wird immer entweder der erste oder der zweite Operand ausgewertet, nie beide.

Obgleich man mit der if-Anweisung (siehe § 5.4.1) Ähnliches erreichen kann, erweist sich der h?:-Operator häufig als eine praktische Kurzform. Hier ist eine typische Verwendung, in der geprüft wird, dass eine Variable definiert ist (und einen sinnvollen, wahren Wert hat) und diese gegebenenfalls verwendet, andernfalls hingegen auf einen Vorgabewert ausgewichen wird:

```
greeting = "Hallo " + (username ? username : "Welt!");
```

Das ist zur folgenden if-Anweisung äquivalent, aber erheblich kompakter:

```
greeting = "Hallo ";
if (username)
    greeting += username;
else
    greeting += "Welt";
```

4.13.2 Der typeof-Operator

typeof ist ein unärer Operator, der seinem Operanden vorangestellt wird, der einen beliebigen Typ haben kann. Sein Wert ist ein String, der den Typ des Operanden angibt. Die folgende Tabelle gibt den Wert des typeof-Operators für alle JavaScript-Werte an:

x	typeof x
undefined	"undefined"
null	"object"
true oder false	"boolean"
eine beliebige Zahl oder NaN	"number"
ein beliebiger String	"string"
eine beliebige Funktion	"function"
jedes native Objekt, das keine Funktion ist	"object"
ein beliebiges Host-Objekt	Ein implementierungsabhängiger String, aber nicht »undefined«, »boolean«, »number« oder »string«.

Sie könnten den typeof-Operator in einem Ausdruck wie diesem nutzen:

```
(typeof value == "string") ? "'" + value + "'" : value
```

Der typeof-Operator ist auch bei der Verwendung in einer switch-Anweisung praktisch (siehe § 5.4.3). Beachten Sie, dass Sie den Operanden für typeof in eine Klammer stellen können, was typeof wie einen Funktionsnamen statt wie ein Operatorenschlüsselwort aussehen lässt:

```
typeof(i)
```

Beachten Sie, dass typeof »object« liefert, wenn der Wert des Operanden null ist. Wenn Sie null von Objekten unterscheiden müssen, müssen Sie diesen speziellen Wert explizit prüfen. typeof kann bei Host-Objekten einen anderen String als »object« liefern. In der Praxis ist es jedoch so, dass die meisten Host-Objekte in clientseitigem JavaScript den Typ »object« haben.

Weil typeof für alle Objekt- und Array-Werte außer Funktionen zu »object« ausgewertet wird, ist der Operator nur zur Unterscheidung von Objekten und anderen, elementaren Typen geeignet. Wollen Sie eine Klasse von Objekten von einer anderen unterscheiden, müssen Sie andere Technologien wie den instanceof-Operator (siehe § 4.9.4), das *class*-Attribut (siehe § 6.8.2) oder die constructor-Eigenschaft (siehe § 6.8.1 und § 9.2.2) nutzen.

Obwohl Funktionen in JavaScript eine Art von Objekt sind, betrachtet der typeof-Operator Funktionen als so anders, dass es einen eigenen Rückgabewert für sie gibt. JavaScript macht einen feinen Unterschied zwischen Funktionen und »aufrufbaren Objekten«. Alle Funktionen sind aufrufbar, aber man kann auch aufrufbare Objekte haben – Objekte, die sich wie Funktionen aufrufen lassen –, die keine echten Funktionen sind. Die ECMAScript 3-Spezifikation sagt, dass der typeof-Operator für alle nativen Objekte, die aufrufbar sind, »function« liefert. Die ECMAScript 5-Spezifikation erweitert das und verlangt, dass typeof »function« für alle aufrufbaren Objeket liefert, unabhängig davon, ob es native Objekte oder Host-Objekte sind. Die meisten Browser-Hersteller nutzen native JavaScript-Funktionsobjekte für die Methoden ihrer Host-Objekte. Microsoft jedoch hat schon immer nicht-native aufrufbare Objekte für seine clientseitigen Methoden verwendet. Vor dem IE 9 lieferte der typeof-Operator »object« für sie, obwohl sie sich wie Funktionen verhalten. Im IE9 sind diese clientseitigen Methoden jetzt echte native Funktionsobjekte. Mehr Informationen zum Unterschied zwischen echten Funktionen und aufrufbaren Objekten finden Sie unter § 8.7.7.

4.13.3 Der delete-Operator

delete ist ein unärer Operator, der versucht, die Objekteigenschaft oder das Array-Element zu löschen, die bzw. das als sein Operand angegeben wird.[1] Wie die Operatoren für Zuweisung, Inkrement und Dekrement wird delete üblicherweise wegen seines Seiten-

1 Sollten Sie C++-Programmierer sein, müssen Sie beachten, dass das delete-Schlüsselwort von JavaScript keinerlei Ähnlichkeit mit dem delete-Schlüsselwort von C++ hat. In JavaScript erfolgt die Speicherfreigabe automatisch durch die Garbage Collection. Sie müssen sich also nie darüber Gedanken machen, Speicher explizit freizugeben. Ein C++-artiges delete zum Löschen vollständiger Objekte ist also überhaupt nicht erforderlich.

effekts, Eigenschaften zu löschen, eingesetzt, und nicht für den Wert, den es liefert. Einige Beispiele:

```
var o = { x: 1, y: 2}; // Ein Objekt definieren.
delete o.x;            // Eine seiner Eigenschaften löschen.
"x" in o               // => false: Die Eigenschaft gibt es nicht mehr.

var a = [1,2,3];       // Ein Array definieren.
delete a[2];           // Sein letztes Element löschen.
a.length               // => 2: Jetzt hat es nur noch zwei Elemente.
```

Beachten Sie, dass gelöschte Eigenschaften oder Array-Elemente nicht einfach auf `undefined` gesetzt werden. Wird eine Eigenschaft gelöscht, endet ihre Existenz. Ein Versuch, eine nicht existierende Eigenschaft zu lesen, liefert `undefined`, aber die tatächliche Existenz einer Eigenschaft können Sie mit dem `in`-Operator (siehe § 4.9.3) prüfen.

`delete` erwartet, dass sein Operand ein Lvalue ist. Ist er kein Lvalue, unternimmt der Operator nichts und liefert `true`. Andernfalls versucht `delete`, den angegebenen Lvalue zu löschen. `delete` liefert `true`, wenn der angegebene Lvalue erfolgreich gelöscht wurde. Es können jedoch nicht alle Eigenschaften gelöscht werden: Einige eingebaute Eigenschaften des Sprachkerns von JavaScript und clientseitigem JavaScript sind für Löschungen immun. Auch benutzerdefinierte Variablen, die mit der `var`-Anweisung deklariert wurden, können nicht gelöscht werden. Funktionen, die mit der `function`-Anweisung definiert wurden, und deklarierte Funktionsparameter können ebenfalls nicht gelöscht werden.

Im Strict-Modus von ECMAScript 5 löst `delete` einen SyntaxError aus, wenn sein Operand ein nicht-qualifizierter Bezeichner wie eine Variable, eine Funktion oder ein Funktionsparameter ist: Der Operator funktioniert nur, wenn der Operand ein Eigenschaftszugriffsausdruck (siehe § 4.4) ist. Der Strict-Modus gibt auch vor, dass `delete` einen TypeError auslöst, wenn es aufgefordert wird, eine nicht konfigurierbare Eigenschaft zu löschen (siehe § 6.7). Außerhalb des Strict-Modus treten in diesen Fällen keine Exceptions auf. `delete` liefert einfach `false`, um anzuzeigen, dass der Operand nicht gelöscht werden konnte.

Hier sind einige Beispiele zur Verwendung des delete-Operators:

```
var o = {x:1, y:2};  // Eine Variable deklarieren und auf ein Objekt initialisieren.
delete o.x;          // Eine der Eigenschaften des Objekts löschen; liefert true.
typeof o.x;          // Die Eigenschaft gibt es nicht; liefert "undefined".
delete o.x;          // Eine nicht vorhandene Eigenschaft löschen; liefert true.
delete o;            // Deklarierte Variablen können nicht gelöscht werden; liefert
                     // false.
                     // Würde im Strict-Modus eine Exception auslösen.
delete 1;            // Argument ist kein Lvalue: liefert true.
this.x = 1;          // Ohne var eine Eigenschaft auf dem globalen Objekt definieren.
delete x;            // Versuchen, diese zu löschen: Liefert im normalen Modus true,
                     // im Strict-Modus eine Exception. Nutzen Sie stattdessen
                     // 'delete this.x'.
x;                   // Laufzeitfehler: x ist nicht definiert.
```

Der `delete`-Operator wird uns in § 6.3 wieder begegnen.

4.13.4 Der void-Operator

`void` ist ein unärer Operator, der vor seinem Operanden steht, der einen beliebigen Typ haben kann. Dieser Operator ist ungewöhnlich und wird nur selten eingesetzt: Er wertet seinen Operanden aus, verwirft aber dann den Wert und liefert `undefined`. Da der Wert des Operanden verworfen wird, ist `void` nur sinnvoll, wenn der Operand Seiteneffekte hat.

Der gebräuchliste Verwendungszweck für diesen Operator ist in einer clientseitigen `javascript:`-URL. Dort ermöglicht er die Auswertung eines Ausdrucks um seiner Seiteneffekte wegen, ohne dass der Browser den Wert des ausgewerteten Ausdrucks anzeigt. Beispielsweise könnten Sie den `void`-Operator folgendermaßen in einem HTML `<a>`-Tag einsetzen:

```
<a href="javascript:void window.open();">Neues Fenster öffnen</a>
```

Dieses HTML hätte mit einem `onclick`-Event-Handler statt einer `javascript:`-URL natürlich besser lesbar geschrieben werden können, in dem der `void`-Operator nicht mehr erforderlich gewesen wäre.

4.13.5 Der Kommaoperator (,)

Der Kommaoperator ist ein binärer Operator, der mit Operanden beliebigen Typs umgehen kann. Er wertet seinen linken Operanden aus, wertet seinen rechten Operanden aus und liefert dann den Wert des rechten Operanden. Die folgende Zeile

```
i=0, j=1, k=2;
```

wird mit 2 ausgewertet und entspricht im Prinzip:

```
i = 0; j = 1; k = 2;
```

Der Ausdruck auf der linken Seite wird immer ausgewertet, aber sein Wert wird verworfen. Der Einsatz des Kommaoperators ist also nur sinnvoll, wenn der linksseitige Ausdruck Seiteneffekte hat. Der einzige Ort, an dem der Kommaoperator häufig verwendet wird, ist eine `for`-Schleife (siehe § 5.5.3) mit mehreren Schleifenvariablen:

```
// Das erste Komma unten ist Teil der Syntax der var-Anweisung,
// das zweite der Kommaoperator: Quetschen wir 2
// Ausdrücke (i++ und j--) in eine Anweisung (die for-Schleife), die einen erwartet.
for(var i=0,j=10; i < j; i++,j--)
    console.log(i+j);
```