KAPITEL 4

Ausdrücke und Operatoren

In diesem Kapitel geht es um Ausdrücke und die Operatoren, mit denen viele von ihnen gebildet werden. Ein *Ausdruck* ist in JavaScript eine Zeichenfolge, die ausgewertet werden kann, um einen Wert zu liefern. 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 genau dem Wert ausgewertet wird, der dieser Variablen zuvor zugewiesen wurde. Komplexe Ausdrücke werden aus einfacheren Ausdrücken aufgebaut. Ein vollständiger Ausdruck für den Zugriff auf ein Array setzt sich beispielsweise zusammen aus einem Ausdruck, der zu einem Array ausgewertet wird, einer folgenden öffnenden eckigen Klammer, einem Ausdruck, der zu einer ganzen Zahl ausgewertet wird, und schließlich einer schließenden eckigen Klammer. Dieser neue, zusammengesetzte und komplexere Ausdruck wird zu dem Wert ausgewertet, der an der entsprechenden Indexposition im angegebenen Array gespeichert ist. In ähnlicher Weise besteht ein Funktionsaufruf aus einem Ausdruck, der zu einem Funktionsobjekt ausgewertet wird, und null oder mehr zusätzlichen Ausdrücken, die als die Argumente für die Funktion verwendet werden.

Am häufigsten setzt man für den Aufbau komplexer Ausdrücke aus einfacheren Ausdrücken *Operatoren* ein. Ein Operator kombiniert die Werte seiner (meist zwei) Operanden auf vorgegebene Weise und wird zu einem neuen Wert ausgewertet. Der Multiplikationsoperator * ist ein einfaches Beispiel dafür: Der Ausdruck x * y wird zum Produkt der Werte der Ausdrücke x und y ausgewertet. Der Einfachheit halber spricht man manchmal auch davon, dass ein Operator einen Wert *zurückgibt*, und nicht, dass er zu einem Wert »auswertet«.

Dieses Kapitel dokumentiert alle Operatoren von JavaScript und erläutert außerdem Ausdrücke, die keine Operatoren nutzen, wie die Array-Indizierung und Funktionsaufrufe. Wenn Sie sich mit Sprachen wie C auskennen, wird Ihnen die Syntax von JavaScript bereits vertraut erscheinen.

4.1 Elementare Ausdrücke

Als *elementare Ausdrücke* bezeichnet man solche Ausdrücke, die nicht weiter in einfachere Teilausdrücke zerlegt werden können. Elementare Ausdrücke sind in JavaScript konstante oder *literale* Werte, bestimmte 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 String-Literal.
/pattern/    // Ein Regex-Literal.
```

Die Syntax für Zahlliterale haben wir in 3.2 behandelt. String-Literale wurden in 3.3 besprochen. Die Syntax für Regex-Literale wurde in 3.3.5 bereits vorgestellt und wird in 11.3 noch ausführlich behandelt.

Einige der reservierten Wörter von JavaScript sind elementare Ausdrücke:

```
true       // Wird zum booleschen Wert true ausgewertet.
false      // Wird zum booleschen Wert false ausgewertet.
null       // Wird zum Wert null ausgewertet.
this       // Wird zum "aktuellen" Objekt ausgewertet.
```

`true`, `false` und `null` haben wir in 3.4 und in 3.5 kennengelernt. Im Unterschied zu den anderen Schlüsselwörtern ist `this` keine Konstante – es wird an unterschiedlichen Stellen eines Programms zu unterschiedlichen Werten ausgewertet. `this` wird in der objektorientierten Programmierung genutzt. Im Körper einer Methode wird `this` zu dem Objekt ausgewertet, auf dem die Methode aufgerufen wurde. Mehr zu `this` finden Sie in 4.5, in Kapitel 8 (insbesondere 8.2.2) und in Kapitel 9.

Die dritte Form elementarer Ausdrücke sind Verweise auf eine Variable, Konstante oder Eigenschaft des globalen Objekts:

```
i            // Wird zum Wert der Variablen i ausgewertet.
sum          // Wird zum Wert der Variablen sum ausgewertet.
undefined    // Der Wert der Eigenschaft "undefined" des globalen Objekts.
```

Erscheint ein Identifier für sich alleine stehend in einem Programm, geht JavaScript davon aus, dass es sich um eine Variable oder Konstante oder um eine Eigenschaft des globalen Objekts handelt. Der Versuch, eine nicht vorhandene Variable auszuwerten, löst einen ReferenceError, einen Referenzfehler, aus.

4.2 Initialisierungsausdrücke von Objekten und Arrays

Objekt- und *Array-Initialisierer* sind Ausdrücke, deren Wert ein neu erzeugtes Objekt oder Array ist. Diese Initialisierungsausdrücke werden auch als *Objektliterale* und *Array-Literale* bezeichnet. Im Unterschied zu echten Literalen sind sie jedoch

keine elementaren Ausdrücke, da sie mehrere Teilausdrücke mit Eigenschafts- und Elementwerten enthalten. Array-Initialisierer haben eine etwas einfacherere Syntax als Objektinitialisierer. Mit ihnen 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 erstelltes Array. Die Elemente dieses neuen Arrays werden mit den Werten der durch Kommata getrennten Ausdrücke initialisiert:

```
[]        // Ein leeres Array. Keine Ausdrücke in den Klammern heißt:
          // keine Elemente.
[1+2,3+4] // Ein Array mit 2 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 mit solchen Ausdrücken verschachtelte Arrays erstellt werden können:

```
let 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, der Wert eines Array-Initialisierungsausdrucks kann bei jeder Auswertung anders lauten.

Undefinierte Elemente können in Array-Literale aufgenommen werden, indem zwischen zwei Kommata einfach kein Wert angegeben wird. Das folgende Array enthält beispielsweise fünf Elemente, von denen drei undefiniert sind:

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

Nach dem letzten Ausdruck in einem Array-Initialisierer ist ein nachstehendes Komma erlaubt, ohne dass dabei ein undefiniertes Element erstellt wird. Jeder Ausdruck zum Zugriff auf einen höheren Index als den des letzten Ausdrucks wird jedoch zwangsläufig zu `undefined` ausgewertet.

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

```
let p = { x: 2.3, y: -1.2 };  // Ein Objekt mit 2 Eigenschaften.
let q = {};                   // Ein leeres Objekt ohne Eigenschaften.
q.x = 2.3; q.y = -1.2;        // Jetzt hat q die gleichen Eigenschaften wie p.
```

Ab ES6 haben Objektliterale eine wesentlich funktionsreichere Syntax (Details finden Sie in 6.10). Objektliterale können ebenfalls verschachtelt werden, zum Beispiel:

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

Objektliterale werden ausführlich in den Kapiteln 6 und 7 behandelt.

4.3 Ausdrücke zur Funktionsdefinition

Ein *Ausdruck zur Funktionsdefinition* 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 in runden Klammern folgenden optionalen durch Kommata getrennten Liste weiterer Identifier (der Parameternamen) und einem Block mit JavaScript-Code (dem Funktionskörper) in geschweiften Klammern, zum Beispiel:

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

Ein Funktionsdefinitionsausdruck kann auch einen Namen für die Funktion enthalten. Funktionen können ebenfalls mit einer Funktionsanweisung statt eines Funktionsausdrucks definiert werden. Und in ES6 und später können Funktionsausdrücke eine kompakte neue Syntax verwenden, die mit sogenannten *Pfeilfunktionen* arbeitet. Alle Details zu Funktionsdefinitionen finden Sie in Kapitel 8.

4.4 Ausdrücke für den Eigenschaftszugriff

Ein *Ausdruck zum Eigenschaftszugriff* bzw. ein *Eigenschaftszugriffsausdruck* wertet den Wert einer Objekteigenschaft oder eines Array-Elements aus. JavaScript definiert zwei Syntaxformen für den Eigenschaftszugriff:

```
Ausdruck . Identifier
Ausdruck [ Ausdruck ]
```

Bei der ersten Art des Eigenschaftszugriffs wird ein Ausdruck genutzt, auf den ein Punkt und dann ein Identifier folgen. Der Ausdruck gibt das Objekt an und der Identifier 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 einige konkrete Beispiele:

```
let o = {x: 1, y: {z: 3}}; // Ein Beispielobjekt.
let a = [o, 4, [5, 6]];    // Ein Beispielarray, 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 an Indexposition 1 des
                           // Ausdrucks a.
a[2]["1"]                  // => 6: Das Element an Indexposition 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 `.` oder [zuerst ausgewertet. Ist der Wert `null` oder `undefined`, löst der Eigenschaftszugriff einen TypeError aus, da diese beiden JavaScript-Werte keine Eigenschaften haben können. Folgen auf den Objektausdruck ein Punkt und ein Identifier, wird der Wert der dadurch benannten Eigenschaft nachgeschlagen und 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 mit dem durch den String angegebenen Namen. In beiden Fällen lautet der Wert des Eigenschaftszugriffsausdrucks `undefined`, wenn es die angegebene Eigenschaft nicht gibt.

Die Identifier-Syntax mit Punktnotation 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 zulässigen Identifier-Namen hat und Sie den Namen des Identifiers beim Schreiben des Programms bereits kennen. Wenn der Eigenschaftsname Leerzeichen oder Interpunktionszeichen enthält oder eine Zahl ist (bei Arrays), müssen Sie die Notation mit 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 6.3.1).

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

4.4.1 Bedingter Zugriff auf Eigenschaften

Mit ES2020 sind zwei neue Möglichkeiten für den Eigenschaftszugriff hinzugekommen:

```
Ausdruck ?. Identifier
Ausdruck ?.[ Ausdruck ]
```

In JavaScript sind `null` und `undefined` die beiden einzigen eigenschaftslosen Werte. In einem regulären Ausdruck für den Eigenschaftszugriff mit `.` oder `[]` erhalten Sie einen TypeError, wenn der Ausdruck auf der linken Seite zu `null` oder `undefined` ausgewertet wird. Sie können aber `?.` und `?.[]` verwenden, um sich vor Fehlern dieser Art zu schützen.

Betrachten Sie den Ausdruck `a?.b`. Wenn `a` gleich `null` oder `undefined` ist, wird der Ausdruck zu `undefined` ausgewertet, ohne dass überhaupt versucht wird, auf die Eigenschaft `b` zuzugreifen. Wenn `a` einen anderen Wert hat, wird `a?.b` genauso wie sonst `a.b` ausgewertet (und wenn `a` keine Eigenschaft namens `b` hat, wird das Ergebnis `undefined` lauten).

Diese Form des Ausdrucks für den Eigenschaftszugriff wird manchmal als *optionale Verkettung* (*Optional Chaining*) bezeichnet, weil sie auch für längere, »verkettete« Ausdrücke für den Eigenschaftszugriff wie diesen funktioniert:

```
let a = { b: null };
a.b?.c.d   // => undefined
```

a ist ein Objekt, also ist a.b ein gültiger Ausdruck für den Eigenschaftszugriff. Aber der Wert von a.b ist null, sodass a.b.c einen TypeError auslösen würde. Durch die Verwendung von ?. anstelle von . vermeiden wir diesen TypeError, und a.b?.c wird zu undefined ausgewertet. Das wiederum bedeutet, dass (a.b?.c).d einen TypeError auslöst, weil dieser Ausdruck versucht, auf eine Eigenschaft mit dem Wert undefined zuzugreifen. Aber – und das ist entscheidend bei der optionalen Verkettung – a.b?.c.d (ohne die Klammern) ergibt einfach undefined und löst keinen Fehler aus. Das liegt daran, dass der Zugriff auf Eigenschaften mit ?. vorzeitig unterbrochen wird (im Englischen wird dabei häufig von *short-circuiting* gesprochen, also »kurzschließen«): Wenn der Unterausdruck links von ?. zu null oder undefined ausgewertet wird, dann wird der gesamte Ausdruck sofort zu undefined ausgewertet, ohne dass überhaupt noch versucht wird, auf die nachfolgend angegebenen Eigenschaften zuzugreifen.

Wäre a.b ein Objekt und besäße dieses Objekt keine Eigenschaft namens c, würde a.b?.c.d natürlich wieder einen TypeError auslösen – und wir müssten einen weiteren bedingten Eigenschaftszugriff verwenden:

```
let a = { b: {} };
a.b?.c?.d  // => undefined
```

Der bedingte Eigenschaftszugriff ist auch mit ?.[] anstelle von [] möglich. Wenn im Ausdruck a?.[b][c] der Wert von a gleich null oder undefined ist, wird der gesamte Ausdruck sofort zu undefined ausgewertet, ohne dass bei den Teilausdrücken b und c eine Auswertung überhaupt noch versucht wird. Weist einer dieser Ausdrücke Nebeneffekte auf, treten diese nicht auf, wenn a gar nicht definiert ist:

```
let a;          // Hoppla, wir haben vergessen, diese Variable zu initialisieren!
let index = 0;
try {
    a[index++]; // Löst einen TypeError aus.
} catch(e) {
    index       // => 1: Inkrementierung erfolgt, bevor der TypeError
                // ausgelöst wird.
}
a?.[index++]    // => undefined: weil a undefined ist.
index           // => 1: Nicht erneut inkrementiert, weil ?.[] vorzeitig
                // unterbrochen ("kurzgeschlossen") hat.
a[index++]      // !TypeError: undefined kann nicht indiziert werden.
```

Der bedingte Eigenschaftszugriff mit ?. und ?.[] ist eine der neuesten Funktionen von JavaScript. Stand Anfang 2020 wird diese neue Syntax in den aktuellen oder Betaversionen der meisten gängigen Browser unterstützt.

4.5 Aufrufausdrücke

Mit einem *Aufrufausdruck* ruft man in JavaScript eine Funktion oder Methode auf bzw. führt sie aus. Er 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 keine Funktion, wird ein TypeError ausgelöst.) Danach werden die Argumentwerte der Reihe nach den Parameternamen zugewiesen, die in der Definition der Funktion angegeben sind, bevor anschließend der Code im Körper der Funktion ausgeführt wird. Nutzt die Funktion eine `return`-Anweisung, um einen Wert zurückzuliefern, wird dieser zum Wert des Aufrufausdrucks. Andernfalls ist der Wert des Aufrufausdrucks `undefined`. Sämtliche Informationen zum Funktionsaufruf – auch dazu, was geschieht, wenn die Anzahl von Argumentausdrücken nicht mit der Anzahl der Parameter in der Funktionsdefinition übereinstimmt – finden Sie in Kapitel 8.

Jeder Aufrufausdruck enthält ein Klammernpaar, vor dem ein Ausdruck steht. Greift der Ausdruck vor den Klammern auf eine Eigenschaft zu, bezeichnet man den Aufruf selbst als *Methodenaufruf*. Bei Methodenaufrufen wird das Objekt oder Array, das Gegenstand des Eigenschaftszugriffs ist, während der Ausführung des Codes im Funktionskörper zum Wert des `this`-Schlüsselworts. Das ist das Fundament des objektorientierten Programmierparadigmas, bei dem Funktionen als »Methoden« bezeichnet werden und auf dem Objekt operieren, zu dem sie gehören. Einzelheiten dazu finden Sie in Kapitel 9.

4.5.1 Bedingter Aufruf

Ab ES2020 können Sie eine Funktion auch mit `?.()` anstelle von `()` aufrufen. Normalerweise wird beim Aufruf einer Funktion ein TypeError ausgelöst, wenn der Ausdruck links von der Klammer `null` oder `undefined` lautet oder keine Funktion ist. Bei der neuen Aufrufsyntax der Form `?.()` wird der gesamte Aufrufausdruck zu `undefined` ausgewertet, wenn der Ausdruck links vom `?.` zu `null` oder `undefined` ausgewertet wird, und zudem wird keine Ausnahme ausgelöst.

Array-Objekte verfügen über eine `sort()`-Methode, der optional ein Funktionsargument übergeben werden kann, das die gewünschte Sortierreihenfolge für die Array-Elemente festlegt. Wollte man vor ES2020 eine Methode wie `sort()` schreiben, die ein optionales Funktionsargument benötigt, hätte man normalerweise mit einer `if`-Anweisung überprüft, ob das Funktionsargument definiert wurde. Erst danach hätte man es dann im Funktionskörper aufgerufen:

```
function square(x, log) { // Das zweite Argument ist eine optionale Funktion.
    if (log) {            // Falls die optionale Funktion übergeben wird,
        log(x);           // wird sie aufgerufen.
    }
```

```
    return x * x;         // Das Quadrat des Arguments wird zurückgegeben.
}
```

Mit der bedingten Aufrufsyntax von ES2020 können Sie den Funktionsaufruf jedoch einfach mit `?.()` formulieren. Damit stellen Sie sicher, dass der Aufruf nur dann erfolgt, wenn es auch tatsächlich einen aufzurufenden Wert gibt:

```
function square(x, log) { // Das zweite Argument ist eine optionale Funktion.
    log?.(x);             // Rufen Sie die Funktion auf, falls es eine gibt.
    return x * x;         // Das Quadrat des Arguments wird zurückgegeben.
}
```

Denken Sie aber daran, dass `?.()` nur prüft, ob die linke Seite `null` oder `undefined` ist, aber nicht, ob der Wert tatsächlich eine Funktion ist. Die `square()`-Funktion des obigen Beispiels würde also immer noch eine Ausnahme auslösen, wenn Sie ihr z. B. zwei Zahlen übergeben (anstelle einer Zahl und einer Funktion).

Wie die Ausdrücke für den bedingten Eigenschaftszugriff (siehe 4.4.1) ist der Funktionsaufruf mit `?.()` »kurzschließend« (*short-circuiting*): Die Auswertung wird vorzeitig unterbrochen, wenn der Wert links von `?.` `null` oder `undefined` lautet. Keiner der Argumentausdrücke innerhalb der Klammern wird dann noch ausgewertet:

```
let f = null, x = 0;
try {
    f(x++); // Löst einen TypeError aus, weil f null ist.
} catch(e) {
    x       // => 1: x wurde inkrementiert, bevor die Ausnahme ausgelöst wurde.
}
f?.(x++)    // => undefined: f ist null, aber es wird keine Ausnahme ausgelöst.
x           // => 1: Inkrementierung wurde (wegen "Kurzschließung") übersprungen.
```

Bedingte Aufrufausdrücke mit `?.()` funktionieren für Methoden genauso wie für Funktionen. Da der Methodenaufruf jedoch auch den Zugriff auf Eigenschaften beinhaltet, sollten Sie sich einen Moment Zeit nehmen, um sicherzustellen, dass Sie die Unterschiede zwischen den folgenden Ausdrücken verstehen:

```
o.m()     // Normaler (regulärer, nicht bedingter) Zugriff auf Eigenschaften,
          // normaler Aufruf.
o?.m()    // Bedingter Zugriff auf Eigenschaften, normaler Aufruf.
o.m?.()   // Normaler Zugriff auf Eigenschaften, bedingter Aufruf.
```

Im ersten Ausdruck muss `o` ein Objekt mit einer Eigenschaft `m` und der Wert dieser Eigenschaft muss eine Funktion sein. Wenn im zweiten Ausdruck `o` `null` oder `undefined` ist, wird der Ausdruck zu `undefined` ausgewertet. Hat `o` aber einen anderen Wert, muss es eine Eigenschaft `m` besitzen, deren Wert eine Funktion ist. Und im dritten Ausdruck darf `o` nicht `null` oder `undefined` sein. Falls es keine Eigenschaft `m` besitzt oder wenn der Wert dieser Eigenschaft `null` ist, wird der gesamte Ausdruck zu `undefined` ausgewertet.

Der bedingte Aufruf mit `?.()` ist – wie der bedingte Eigenschaftszugriff – eine der neuesten Funktionen von JavaScript. Stand Anfang 2020 wird auch diese neue Syntax in den aktuellen oder Betaversionen der meisten gängigen Browser unterstützt.

4.6 Ausdrücke zur Objekterstellung

Ein *Ausdruck zur Objekterstellung* bzw. ein *Objekterstellungsausdruck* erstellt – erwartungsgemäß – ein neues Objekt und ruft eine (als Konstruktor bezeichnete) Funktion auf, um die Eigenschaften dieses Objekts zu initialisieren. Ausdrücke zur Objekterstellung ähneln den Aufrufausdrücken, abgesehen davon, dass ihnen das Schlüsselwort `new` vorangestellt wird:

```
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
```

Der Wert eines Objekterstellungsausdrucks ist das neu erstellte Objekt. Konstruktoren werden in Kapitel 9 ausführlicher erläutert.

4.7 Operatoren im Überblick

Operatoren werden in JavaScript für arithmetische Ausdrücke, Vergleichsausdrücke, logische Ausdrücke, Zuweisungsausdrücke und anderes verwendet. Tabelle 4-1 fasst die Operatoren zusammen und dient als praktische Referenz.

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

In Tabelle 4-1 sind die Operatoren anhand ihres Vorrangs sortiert. Die zuerst aufgeführten Operatoren haben Vorrang vor den nachfolgend aufgeführten. Operatoren, zwischen denen eine horizontale Linie steht, gehören unterschiedlichen Vorrangstufen an. 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) betragen. Spalte »N« gibt die Anzahl von Operanden an. Die Spalte mit der Bezeichnung »Typen« führt die erwarteten Datentypen der Operanden und – nach dem Symbol → – den Ergebnistyp des Operators auf. Dabei habe ich mich an den Ergebniswerten orientiert, die der `typeof`-Operator zurückgibt (siehe Tabelle 4-3), und diese abgekürzt. Außerdem steht *bel* für »beliebig« und *lval* für »Lvalue«. In den Unterabschnitten im Anschluss an die Tabelle werden die Konzepte des Vorrangs, der Assoziativität und des Operandentyps erläutert. Die Operatoren selbst werden im Anschluss an diese Abschnitte einzeln beschrieben.

Tabelle 4-1: JavaScript-Operatoren

Operator	Operation	A	N	Typen
`++`	Prä- oder Postinkrement	R	1	lval→num
`--`	Prä- oder Postinkrement	R	1	lval→num
`-`	Zahl negieren	R	1	num→num
`+`	in eine Zahl umwandeln	R	1	bel→num
`~`	Bits invertieren	R	1	int→int
`!`	booleschen Wert invertieren	R	1	bool→bool
`delete`	Eigenschaft entfernen	R	1	lval→bool
`typeof`	Typ des Operanden ermitteln	R	1	bel→str
`void`	Wert `undefined` zurückgeben	R	1	bel→undef
`**`	Potenzieren	R	2	num,num→num
`*`, `/`, `%`	Multiplizieren, Dividieren, Rest (Modulo)	L	2	num,num→num
`+`, `-`	Addieren, Subtrahieren	L	2	num,num→num
`+`	Strings verketten	L	2	str,str→str
`<<`	nach links verschieben (»shift«)	L	2	int,int→int
`>>`	nach rechts verschieben mit Vorzeichen	L	2	int,int→int
`>>>`	nach rechts verschieben mit Nullauffüllung	L	2	int,int→int
`<`, `<=`, `>`, `>=`	Vergleich in numerischer Reihenfolge	L	2	num,num→bool
`<`, `<=`, `>`, `>=`	Vergleich in alphabetischer Reihenfolge	L	2	str,str→bool
`instanceof`	Objektklasse prüfen	L	2	obj,func→bool
`in`	Prüfung, ob eine Eigenschaft existiert	L	2	bel,obj→bool
`==`	Prüfung auf nicht-strikte Gleichheit	L	2	bel,bel→bool
`!=`	Prüfung auf nicht-strikte Ungleichheit	L	2	bel,bel→bool
`===`	Prüfung auf strikte Gleichheit	L	2	bel,bel→bool
`!==`	Prüfung auf strikte Ungleichheit	L	2	bel,bel→bool
`&`	bitweises AND berechnen	L	2	int,int→int
`^`	bitweises XOR berechnen	L	2	int,int→int
`|`	bitweises OR berechnen	L	2	int,int→int
`&&`	logisches AND berechnen	L	2	bel,bel→bel
`||`	logisches OR berechnen	L	2	bel,bel→bel
`??`	1. vorhandenen Operanden auswählen	L	2	bel,bel→bel
`?:`	2. oder 3. Operanden auswählen	R	3	bool,bel,bel→bel
`=`	Zuweisung an eine Variable oder Eigenschaft	R	2	lval,bel→bel

Tabelle 4-1: JavaScript-Operatoren (Fortsetzung)

Operator	Operation	A	N	Typen
**=, *=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=	Operation durchführen und gleichzeitig Zuweisung vornehmen	R	2	lval,bel→bel
,	1. Operanden verwerfen, 2. zurückgeben	L	2	bel,bel→bel

4.7.1 Anzahl an Operanden

Operatoren können anhand der Anzahl der von ihnen erwarteten Operanden (ihrer *Arität* oder *Stelligkeit*) kategorisiert werden. Die meisten Operatoren in JavaScript – wie beispielsweise der Multiplikationsoperator * – sind *binäre Operatoren*, die zwei Ausdrücke zu einem komplexeren Ausdruck vereinen und dementsprechend zwei Operanden erwarten. JavaScript unterstützt darüber hinaus eine Reihe von *unären Operatoren*, die einen einzelnen Ausdruck in einen einzigen komplexeren Ausdruck umwandeln. Der --Operator im Ausdruck -x ist ein unärer Operator, der auf dem Operanden x eine Negation durchführt. Schließlich 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 liefern auch einen Wert eines spezifischen Typs zurück. Die Spalte »Typen« in Tabelle 4-1 gibt (vor dem Pfeil) die Typen der Operanden und (nach dem Pfeil) den Ergebnistyp für die einzelnen Operatoren an.

Üblicherweise konvertieren die Operatoren in JavaScript den Typ ihrer Operanden bei Bedarf (siehe 3.9). Obwohl der Multiplikationsoperator * numerische Operanden erwartet, ist der Ausdruck »3« * »5« zulässig, da JavaScript die Operanden in Zahlen umwandeln kann. Und der Wert dieses Ausdrucks ist natürlich die Zahl 15, nicht der String »15«. Denken Sie bitte auch daran, dass jeder JavaScript-Wert entweder ein »irgendwie wahrer« *truthy*- oder ein »irgendwie falscher« *falsy*-Wert ist (im Sinne nicht-strikter Gleich- bzw. Ungleichheit) und Operatoren, die boolesche Operanden erwarten, deswegen mit Operanden beliebigen Typs arbeiten.

Das Verhalten einiger Operatoren ist vom Typ der verwendeten Operanden abhängig. Der auffälligste Fall ist der des +-Operators, der numerische Operanden addiert, String-Operanden hingegen verkettet. Gleichermaßen führen Vergleichsoperatoren wie < den Vergleich in Abhängigkeit vom Operandentyp auf numerische oder alphabetische Weise durch. Bei den Beschreibungen der einzelnen Operato-

ren werden ihre Typabhängigkeiten erläutert und die durchgeführten Umwandlungen angegeben.

Zuweisungsoperatoren und einige weitere der Operatoren, die in Tabelle 4-1 aufgeführt werden, erwarten einen Operanden vom Typ `lval`. *Lvalue* ist ein historischer Begriff für einen »Ausdruck, der legal auf der linken Seite eines Zuweisungsausdrucks verwendet werden darf«. In JavaScript sind Variablen, Objekteigenschaften und Array-Elemente Lvalues.

4.7.3 Seiteneffekte von Operatoren

Die Auswertung eines einfachen Ausdrucks wie `2 * 3` wirkt sich nie auf den Zustand Ihres Programms aus, d. h., keine der in der Folge von Ihrem Programm ausgeführten Berechnungen wird von dieser Auswertung beeinträchtigt. 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 aller Ausdrücke, die diese Variable bzw. Eigenschaft benutzen. Die Inkrement- und Dekrementoperatoren `++` und `--` verhalten sich ebenfalls so, da sie implizit eine Zuweisung durchführen. Auch der `delete`-Operator hat Seiteneffekte: Das Löschen einer Eigenschaft ähnelt der Zuweisung von `undefined` an diese Eigenschaft (ist aber nicht genau dasselbe).

Die anderen JavaScript-Operatoren haben keine Seiteneffekte. Man sollte jedoch beachten, dass Ausdrücke zum Funktionsaufruf und zur Objekterstellung solche Effekte haben, wenn einer der im Funktionskörper oder Konstruktor genutzten Operatoren Seiteneffekte hat.

4.7.4 Vorrang von Operatoren

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

Betrachten Sie den folgenden Ausdruck:

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

Der Multiplikationsoperator `*` hat einen höheren Vorrang als der Additionsoperator `+`, die Multiplikation wird also vor der Addition ausgeführt (und hat damit auch umgangssprachlich »Vorrang«). Der Zuweisungsoperator `=` hat schließlich den geringsten Vorrang: Die Zuweisung wird also erst ausgeführt, nachdem alle Operationen auf der rechten Seite durchgeführt 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 bitte, dass Eigenschaftszugriffs- und Aufrufausdrücke höheren Vorrang haben als alle in Tabelle 4-1 aufgeführten Operatoren. Betrachten Sie diesen Ausdruck:

```
// my ist ein Objekt mit einer Eigenschaft namens functions, deren Wert ein
// Array von Funktionen ist. Wir rufen die Funktion mit dem Index x auf,
// übergeben ihr das Argument y und fragen dann nach dem Typ des zurückgegebenen
// Werts.
typeof my.functions[x](y)
```

Obwohl `typeof` einer der Operatoren mit der höchsten Priorität ist, wird die entsprechende Operation auf dem Ergebnis von Eigenschaftszugriff, Array-Index und Funktionsaufruf ausgeführt, die alle eine höhere Priorität als Operatoren haben.

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; Zuweisungen haben sehr geringen Vorrang und werden fast immer zuletzt ausgeführt.

Wenn neue Operatoren zu JavaScript hinzugefügt werden, passen sie nicht immer nahtlos in die vorhandene Vorrangordnung. Der `??`-Operator (siehe 4.13.2) wird in der Tabelle mit niedrigerem Vorrang als `||` und `&&` dargestellt, aber tatsächlich ist sein Vorrang relativ zu diesen Operatoren nicht definiert, und ES2020 verlangt die explizite Verwendung von Klammern, wenn man `??` zusammen mit `||` oder `&&` einsetzen will. In ähnlicher Weise hat der neue Exponentialoperator `**` keine genau definierte Präzedenz gegenüber dem unären Negationsoperator, und Sie müssen Klammern verwenden, wenn Sie die Negation mit der Potenzierung kombinieren.

4.7.5 Operatorassoziativität

In Tabelle 4-1 gibt die Spalte »A« die *Assoziativität* des Operators an. Der Wert *L* besagt, 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;
```

dasselbe wie:

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

Betrachten Sie andererseits die folgenden Ausdrücke:

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

Diese entsprechen:

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

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

4.7.6 Reihenfolge der Auswertung

Operatorvorrang und Assoziativität bedingen die Reihenfolge, in der die Operationen in einem komplexen Ausdruck ausgeführt werden. Sie legen jedoch nicht fest, in welcher Reihenfolge Unterausdrücke ausgewertet werden. Ausdrücke werden in JavaScript immer streng von links nach rechts evaluiert. In dem Ausdruck `w = x + y * z` wird z. B. zuerst der Unterausdruck `w` ausgewertet, gefolgt von `x`, `y` und `z`. Dann werden die Werte von `y` und `z` multipliziert, zum Wert von `x` addiert und der durch den Ausdruck `w` spezifizierten Variablen oder Eigenschaft zugewiesen. Fügt man Ausdrücken Klammern hinzu, lässt sich dadurch die relative Reihenfolge der Multiplikation, Addition und Zuweisung ändern, nicht aber die Auswertungsreihenfolge, die von links nach rechts stattfindet.

Die Auswertungsreihenfolge wird nur dann relevant, wenn einer der ausgewerteten Ausdrücke Nebeneffekte hat, die den Wert eines anderen Ausdrucks betreffen. Inkrementiert der Ausdruck `x` eine Variable, die von 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 andere numerische Manipulationen ihrer Operanden durchführen. Die Multiplikations-, Divisions- und Subtraktionsoperatoren sind unkompliziert, und wir behandeln sie gleich zu Beginn. Der Additionsoperator erhält einen eigenen Unterabschnitt, weil er auch String-Verkettungen durchführen kann und einige ungewöhnliche Umwandlungsregeln aufweist. Die unären Operatoren und die bitweisen Operatoren werden ebenfalls in eigenen Unterabschnitten näher betrachtet.

Die meisten arithmetischen Operatoren (mit Ausnahme der nachfolgend aufgeführten) können mit BigInt-Operanden (siehe 3.2.5) oder mit regulären Zahlen verwendet werden, solange Sie die beiden Typen nicht vermischen.

Die elementaren Operatoren sind ** (Potenzierung), * (Multiplikation), / (Division), % (Modulo: Rest nach Division), + (Addition) und - (Subtraktion). Die anderen fünf elementaren Operatoren werten einfach ihre Operanden aus, wandeln die Werte bei Bedarf in Zahlen um und berechnen dann die Potenz, das Produkt, den Quotienten, den Rest oder die Differenz. Nicht numerische Operanden, die nicht in Zahlen umgewandelt werden können, werden in den `NaN`-Wert konvertiert (*Not a Number* – keine Zahl). Wenn einer der Operanden `NaN` ist (oder dazu umgewandelt wird), lautet das Ergebnis der Operation ebenfalls `NaN`.

Der **-Operator hat Vorrang vor *, / und % (die wiederum Vorrang haben vor + und -). Im Gegensatz zu den anderen Operatoren arbeitet ** von rechts nach links, also ist `2**2**3` dasselbe wie `2**8`, nicht wie `4**3`. Ausdrücke wie `-3**2` sind von Natur aus mehrdeutig. Je nach relativer Präzedenz von unärem Minus- und Exponentialoperator könnte dieser Ausdruck `(-3)**2` oder `-(3**2)` bedeuten. Verschiedene Programmiersprachen handhaben diese Situation unterschiedlich, und statt hier eine eindeutige Position zu beziehen, wurde für JavaScript einfach festgelegt, dass es zu einem Syntaxfehler führt, in diesem Fall die Klammern wegzulassen. Dadurch ist man gezwungen, einen eindeutigen Ausdruck zu formulieren. Der Exponentialoperator ** ist der neueste arithmetische Operator in JavaScript: Er wurde mit ES2016 zu JavaScript hinzugefügt. Schon seit den frühesten Versionen von JavaScript existiert die Funktion `Math.pow()`, die genau die gleiche Operation ausführt.

Der /-Operator teilt seinen ersten Operanden durch seinen zweiten Operanden. Wenn Sie an Programmiersprachen gewöhnt sind, die Ganz- und Gleitkommazahlen unterscheiden, erwarten Sie bei der Teilung einer ganzen Zahl durch eine andere vielleicht ein ganzzahliges Ergebnis. Aber da in JavaScript alle Zahlen Gleitkommazahlen sind, liefern alle Divisionsoperationen Gleitkommazahlen als Ergebnis: `5/2` wird zu `2.5` berechnet, nicht zu `2`. Die Division durch null liefert plus oder minus unendlich (`Infinity` oder `-Infinity`), während `0/0` zu `NaN` ausgewertet wird: In keinem dieser Fälle wird ein Fehler ausgelöst.

Der %-Operator berechnet den ersten Operanden modulo den zweiten Operanden. Es wird also eine Modulodivision durchgeführt, d.h. die Operation 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` und `-5 % 2` zu `-1` ausgewertet.

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

4.8.1 Der +-Operator

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

```
1 + 2                       // => 3
"hello" + " " + "there"     // => "hello there"
"1" + "2"                   // => "12"
```

Wenn die Werte der Operanden entweder nur Zahlen oder nur Strings sind, ist offensichtlich, was der +-Operator mit ihnen macht. In allen anderen Fällen sind hingegen Typumwandlungen erforderlich. Die Operation, die letztendlich ausgeführt wird, ist von diesen Umwandlungen abhängig. Die Umwandlungsregeln für + bevorzugen die String-Verkettung: 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 String-Verkettung wird durchgeführt. Eine Addition erfolgt nur dann, wenn keiner der Operanden stringartig ist.

Formal verhält sich der +-Operator folgendermaßen:

- Ist einer seiner Operanden ein Objekt, wird dieses mit dem in 3.9.3 beschriebenen Algorithmus in einen Primitivwert konvertiert. `Date`-Objekte werden über ihre `toString()`-Methode umgewandelt, alle anderen Objekte über ihre `valueOf()`-Methode, falls diese Methode einen primitiven Wert liefert. Die meisten Objekte besitzen jedoch keine nützliche `valueOf()`-Methode und werden deswegen ebenfalls über `toString()` umgewandelt.
- Ist nach der Objekt-zu-Primitivwert-Umwandlung einer der Operanden ein String, wird der andere ebenfalls in einen String umgewandelt, und eine String-Verkettung wird durchgeführt.
- Andernfalls werden beide Operanden in Zahlen (oder `NaN`) konvertiert, die daraufhin addiert werden.

Hier einige Beispiele:

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

Schließlich ist noch zu beachten, dass der Operator + bei Zeichenketten und Zahlen möglicherweise nicht assoziativ ist. Oder anders ausgedrückt: Das Ergebnis kann von der Reihenfolge abhängen, in der die Operationen durchgeführt werden.

Ein Beispiel:

```
1 + 2 + " blind mice"    // => "3 blind mice"
1 + (2 + " blind mice")  // => "12 blind mice"
```

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 ihre einzigen Operanden bei Bedarf in eine Zahl um. Man sollte im Hinterkopf behalten, dass + und - sowohl als unäre als auch als binäre Operatoren verwendet werden.

Es gibt die folgenden unären arithmetischen Operatoren:

Unäres Plus (+)
: Der unäre arithmetische Operator wandelt seinen Operanden in eine Zahl (oder in `NaN`) um und gibt diesen umgewandelten Wert zurück. Wird er auf einem Operanden verwendet, der bereits eine Zahl ist, wird keine Aktion durchgeführt. Dieser Operator darf nicht mit BigInt-Werten verwendet werden, da diese nicht in reguläre Zahlen konvertiert werden können.

Unäres Minus (-)
: Wenn - als unärer Operator verwendet wird, wandelt er seinen Operanden gegebenenfalls in eine Zahl um und ändert dann das Vorzeichen des Ergebnisses.

Inkrement (++)
: Der ++-Operator inkrementiert seinen einzigen Operanden. 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 von einem Präinkrement –, inkrementiert er den Wert und wird dann zum inkrementierten Wert des Operanden ausgewertet. Steht er hinter dem Operanden – dann spricht man von einem Postinkrement –, inkrementiert er den Operanden, wird aber zum *nicht inkrementierten* Wert des Operanden ausgewertet. Den Unterschied verdeutlichen diese beiden Codezeilen:

```
let i = 1, j = ++i;    // i und j sind beide 2.
let n = 1, m = n++;    // n ist 2, m ist 1.
```

  Beachten Sie, dass der Ausdruck `x++` nicht immer das Gleiche ist wie `x=x+1`. Der ++-Operator führt keine String-Verkettung durch: Er wandelt seinen Ope-

randen immer in eine Zahl um und inkrementiert sie. Wenn `x` den String-Wert »1« hat, entspricht `++x` der Zahl 2, während `x+1` zum String »11« ausgewertet wird.

Bitte beachten Sie außerdem, dass wegen JavaScripts automatischer Semikolonergänzung kein Zeilenumbruch zwischen den Postinkrementoperator und den vor ihm stehenden Operanden gesetzt werden darf. Tritt dieser Fall dennoch auf, behandelt JavaScript den Operanden als eine eigenständige Anweisung und fügt vor ihm ein Semikolon ein.

Der Inkrementoperator wird in seinen Erscheinungsformen als Prä- und Postinkrementoperator am häufigsten zur Steuerung von Zählern in `for`-Schleifen (siehe 5.4.3) eingesetzt.

Dekrement (--)

Der Operator -- erwartet einen Lvalue-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 relativ zum Operanden abhängig. Steht er vor seinem Operanden, dekrementiert er den Wert und gibt dann den dekrementierten Wert zurück. Steht er hinter seinem Operanden, dekrementiert er den Operanden, gibt aber den *nicht dekrementierten* Wert zurück. Steht der Operator hinter dem Operanden, ist zwischen Operand und Operator kein Zeilenumbruch erlaubt.

4.8.3 Bitweise Operatoren

Die *bitweisen Operatoren* (oder kurz *Bit-Operatoren*) führen elementare Manipulationen der Bits in der Binärrepräsentation von Zahlen durch. Obwohl sie keine klassischen arithmetischen Operationen ausführen, werden sie hier unter die arithmetischen Operatoren eingereiht, weil sie auf numerischen Operanden operieren und einen numerischen Wert liefern. Vier dieser Operanden führen boolesche Algebra auf den einzelnen Bits der Operanden aus und verhalten sich, als wären die einzelnen Bits des Operanden jeweils boolesche Werte (1 = wahr, 0 = falsch). Die anderen drei Bit-Operatoren werden genutzt, um Bits nach links oder rechts zu verschieben. In der JavaScript-Programmierung werden diese Operatoren nicht sehr häufig verwendet. Wenn Sie mit der binären Darstellung von Integern einschließlich der Zweierkomplement-Darstellung negativer Ganzzahlen nicht vertraut sind, können Sie den Teilabschnitt zu Verschiebungsoperatoren auch einfach überspringen.

Die bitweisen 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 Bit-Operators verwendet werden.

Alle bitweisen Operatoren – außer `>>>` – können mit regulären Zahlenoperanden oder mit BigInt-Operanden verwendet werden (siehe 3.2.5).

Bitweises UND (AND) (&)
: Der `&`-Operator führt eine logische UND-Operation auf allen Bits seiner ganzzahligen Operanden durch. Im Ergebnis wird ein Bit nur dann gesetzt, wenn das entsprechende Bit in beiden Operanden gesetzt ist. `0x1234 & 0x00FF` ergibt beispielsweise `0x0034`.

Bitweises ODER (OR) (|)
: Der `|`-Operator führt eine logische ODER-Operation auf allen Bits seiner ganzzahligen Operanden durch. Im Ergebnis wird ein Bit gesetzt, wenn das entsprechende Bit in einem oder in beiden Operanden gesetzt ist. `0x1234 | 0x00FF` ergibt beispielsweise `0x12FF`.

Bitweises XODER (XOR) (^)
: Der `^`-Operator führt eine exklusive logische ODER-Operation, also eine XODER-Operation, auf den einzelnen Bits seiner ganzzahligen Operanden durch. Exklusives ODER heißt, dass ein Bit entweder im ersten Operanden oder im zweiten Operanden wahr ist, aber nicht in beiden. Im Ergebnis der Operation wird ein Bit gesetzt, wenn das entsprechende Bit in einem (aber nicht in beiden) Operanden gesetzt ist. `0xFF00 ^ 0xF0F0` ergibt beispielsweise `0x0FF0`.

Bitweises NICHT (NOT) (~)
: Der `~`-Operator ist ein unärer Operator, der vor einem ganzzahligen Operanden erscheint. Er bewirkt, dass alle Bits im Operanden umgekehrt werden. Aufgrund der Art, in der Ganzzahlen mit Vorzeichen in JavaScript repräsentiert werden, erzeugt die Anwendung des `~`-Operators auf einen Wert das gleiche Ergebnis, als würde sein Vorzeichen geändert und dann 1 vom Zwischenergebnis abgezogen. Zum Beispiel wird `~0x0F` zu `0xFFFFFFF0` bzw. –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 muss. Zum Beispiel wird in der Operation `a << 1` das erste Bit (das Einer-Bit) von `a` zum zweiten Bit (dem Zweier-Bit), das zweite Bit von `a` zum dritten usw. Für das neue Bit wird eine Null verwendet, 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 dabei 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. `7 >> 1` ergibt beispielsweise 3, `-7 >>1` dagegen –4.

Verschiebung nach rechts mit Nullauffüllung (>>>)

Der >>>-Operator entspricht dem >>-Operator, allerdings werden die nach links verschobenen Bits immer mit null gefüllt, unabhängig vom Vorzeichen des ersten Operanden. Dies ist hilfreich, wenn Sie 32-Bit-Werte mit Vorzeichen so behandeln wollen, als wären sie vorzeichenlose Ganzzahlen. `-1 >> 4` wird beispielsweise zu –1 ausgewertet, aber `-1 >>> 4` zu `0x0FFFFFFF`. Das ist der einzige der bitweisen JavaScript-Operatoren, der nicht mit BigInt-Werten verwendet werden kann. BigInt stellt negative Zahlen nicht dar, indem das hohe Bit wie bei 32-Bit-Integern gesetzt wird, der >>>-Operator ist aber nur zur Darstellung des Zweierkomplements sinnvoll.

4.9 Relationale Ausdrücke

In diesem Abschnitt geht es um die relationalen Operatoren von JavaScript. Diese Operatoren prüfen Verhältnisse und Beziehungen (wie »gleich«, »kleiner« oder »Eigenschaft von«) zwischen zwei Werten und liefern `true` oder `false` zurück, je nachdem, ob dieses Verhältnis oder diese Beziehung besteht oder nicht. Relationale Ausdrücke werden immer zu booleschen Werten ausgewertet, die oft zur Steuerung der Programmausführung in `if`-, `while`- und `for`-Anweisungen (siehe Kapitel 5) genutzt werden. Die folgenden Unterabschnitte beschreiben die Gleichheits- und Ungleichheitsoperatoren, die Vergleichsoperatoren sowie die beiden weiteren relationalen Operatoren von JavaScript, `in` und `instanceof`.

4.9.1 Gleichheits- und Ungleichheitsoperatoren

Die Operatoren == und === prüfen beide, ob zwei Werte gleich sind – allerdings verwenden sie dabei 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 sich unterscheiden. Der ===-Operator wird als *strikter Gleichheitsoperator* (manchmal auch als *Identitätsoperator*)

bezeichnet. Er prüft die »Identität« seiner Operanden gemäß einer strengen Definition von Gleichheit. Der ==-Operator wird als *Gleichheitsoperator* bezeichnet, zur Unterscheidung oft auch als *nicht-strikter Gleichheitsoperator*. Er prüft, ob seine beiden Operanden gleich sind, indem er eine »losere« oder »entspanntere« Definition von Gleichheit verwendet, bei der auch Typumwandlungen erlaubt sind.

Die Operatoren != und !== testen auf das genaue Gegenteil der ==- und ===-Operatoren. Der !=-Ungleichheitsoperator liefert `false`, wenn zwei Werte gemäß == gleich sind, ansonsten `true`. Der !==-Operator liefert `false`, wenn zwei Werte strikt betrachtet 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 für »im strikten Sinne nicht gleich« stehen.

Die Operatoren =, == und ===

JavaScript unterstützt die Operatoren =, == und ===. Es ist wichtig, dass Ihnen die Unterschiede zwischen dem Zuweisungs-, dem Gleichheits- und dem Identitätsoperator (dem strikten Gleichheitsoperator) klar sind. Achten Sie beim Programmieren sorgfältig darauf, dass Sie tatsächlich den passenden Operator verwenden! Obgleich es verführerisch sein mag, bei allen drei Operatoren jeweils »gleich« zu lesen, können Sie möglichen Verwechslungen vorbeugen, wenn Sie für sich = mit »wird zugewiesen«, == mit »ist gleich« und === mit »ist identisch« übersetzen.

Der ==-Operator ist ein veraltetes Sprachmerkmal von JavaScript, dessen Gebrauch als überholt und gemeinhin als Fehlerquelle angesehen wird. Sie sollten in nahezu allen Fällen === anstelle von == und !== anstelle von != verwenden.

Wie in 3.8 bereits erwähnt, werden JavaScript-Objekte anhand der Referenz verglichen, nicht anhand des Werts. Dadurch ist ein Objekt nur mit sich selbst identisch – und mit keinem anderen Objekt. Zwei unabhängige Objekte, die die gleiche Anzahl an Eigenschaften mit gleichen Namen und gleichen Werten haben, sind dennoch nicht gleich. Auch sind zwei Arrays, die die gleichen Elemente in der gleichen Reihenfolge enthalten, nicht gleich.

Strikte Gleichheit

Der strikte 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 `NaN` ist oder beide es sind. (Es mag überraschend erscheinen, aber der `NaN`-Wert ist *keinem* anderen Wert gleich, nicht einmal sich selbst! Um zu prüfen, ob ein Wert `x` gleich `NaN` ist, können Sie den Ausdruck `x !== x` oder die globale Funktion `isNaN()` verwenden.)
- Die Werte sind gleich, wenn beide Zahlen sind und den gleichen Wert 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 3.3) 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 codiert sein. JavaScript führt keine Unicode-Normalisierung durch, und derartige Strings sind weder für den `===`- noch für den `==`-Operator gleich.
- 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 beide Objekte gleiche Eigenschaften besitzen.

Gleichheit mit Typumwandlung

Der nicht-strikte Gleichheitsoperator `==` verhält sich ähnlich wie der strikte Gleichheitsoperator, ist dabei in seiner Auslegung von Gleichheit aber weniger streng. Haben die Werte der beiden Operanden nicht den gleichen Typ, versucht er, die Typen umzuwandeln, und vergleicht dann die umgewandelten Werte. Dabei gilt:

- Haben zwei Werte den gleichen Typ, werden sie wie oben beschrieben auf strikte Gleichheit geprüft. Sind sie im strikten Sinne gleich, sind sie gleich. Sind sie nicht im strikten 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 und der andere ein String, wird der String in eine Zahl umgewandelt, und der Vergleich wird dann mit dem umgewandelten Wert durchgeführt.
  - Ist einer der Werte `true`, wird er in 1 umgewandelt, und der Vergleich wird dann erneut ausgeführt. Ist einer der Werte `false`, wird er in 0 umgewandelt, und der Vergleich wird dann erneut durchgeführt.
  - Ist einer der Werte ein Objekt und der andere eine Zahl oder ein String, wird das Objekt anhand des in 3.9.3 beschriebenen Algorithmus in einen primitiven Wert umgewandelt, und der Vergleich wird dann erneut vorgenommen. Bei einer solchen Umwandlung in einen Primitivwert wird ent-

weder die `toString()`- oder die `valueOf()`-Methode des Objekts benutzt. Die im Sprachkern von JavaScript integrierten Klassen versuchen zuerst, eine `valueOf()`-Umwandlung vorzunehmen, bevor eine `toString()`-Umwandlung ausprobiert wird. Nur die Klasse `Date` führt direkt eine `toString()`-Umwandlung durch.

- Alle anderen Kombinationen von Werten sind nicht gleich.

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

```
"1" == true  // => true
```

Dieser Ausdruck wird zu `true` ausgewertet – diese beiden so unterschiedlich aussehenden Werte sind also in der Tat gleich: Zuerst wird der boolesche Wert `true` (auf der rechten Seite) in die Zahl 1 umgewandelt, und der Vergleich wird erneut durchgeführt. Dann wird der String »1« (auf der linken Seite) in die Zahl 1 umgewandelt. Da beide Werte jetzt gleich sind, liefert der Vergleich als Ergebnis `true`.

4.9.2 Vergleichsoperatoren

Die Vergleichsoperatoren testen die relative Reihenfolge (numerisch oder alphabetisch) ihrer beiden Operanden:

Kleiner als (<)
: Der Operator < wird zu `true` ausgewertet, wenn sein erster Operand kleiner als sein zweiter Operand ist; andernfalls wird er zu `false` ausgewertet.

Größer als (>)
: Der Operator > wird zu `true` ausgewertet, wenn sein erster Operand größer als sein zweiter Operand ist; andernfalls wird er zu `false` ausgewertet.

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

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

Die Operanden dieser Vergleichsoperatoren dürfen beliebigen Typs sein. Vergleiche können jedoch nur auf Zahlen und Strings durchgeführt werden. Operanden, die weder Strings noch Zahlen sind, werden also umgewandelt.

Vergleich und Umwandlung erfolgen nach diesen Regeln:

- Wird einer der Operanden zu einem Objekt ausgewertet, wird dieses so in einen primitiven Wert umgewandelt, wie es am Ende von 3.9.3 beschrieben wurde: Liefert die `valueOf()`-Methode einen primitiven Wert, wird dieser genommen, andernfalls der Rückgabewert der Methode `toString()`.

- Sind nach eventuell erforderlichen Umwandlungen von Objekten in primitive Werte beide Operanden Strings, werden sie anhand ihrer alphabetischen Reihenfolge verglichen. Diese »alphabetische Reihenfolge« wird dabei durch die numerische Abfolge der 16-Bit-Unicode-Werte bestimmt, aus denen die Strings bestehen.
- Ist nach den Umwandlungen von Objekten in primitive 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` zurück. Obwohl die arithmetischen Operatoren nicht zulassen, dass BigInt-Werte mit regulären Zahlen gemischt werden, erlauben die Vergleichsoperatoren Vergleiche zwischen Zahlen und BigInts.

Denken Sie daran, dass JavaScript-Strings Folgen von 16-Bit-Ganzzahlen und String-Vergleiche nur numerische Vergleiche der in ihnen enthaltenen Werte sind. Die durch Unicode vorgegebene numerische Reihenfolge entspricht nicht notwendigerweise der traditionellen Sortierfolge, die in einer bestimmten Sprache oder einem bestimmten Gebietsschema (einer *Locale*) verwendet wird. Beachten Sie insbesondere, dass String-Vergleiche 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 würde der Operator < den String »Zoo« vor dem String »aal« einsortieren.

Einen robusteren Algorithmus für String-Vergleiche finden Sie in der Methode `String.localeCompare()`, die Locale-spezifische Definitionen der alphabetischen Reihenfolge berücksichtigt. Für Vergleiche, bei denen die Groß- und Kleinschreibung nicht berücksichtigt werden soll, können Sie Zeichenfolgen mit `String.toLowerCase()` oder `String.toUpperCase()` in Klein- oder Großbuchstaben konvertieren. Als allgemeineres und besser lokalisiertes Vergleichswerkzeug für Strings können Sie die in 11.7.3 beschriebene `Intl.Collator`-Klasse verwenden.

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

```
1 + 2        // => 3: Addition.
"1" + "2"    // => "12": Verkettung.
"1" + 2      // => "12": 2 wird in "2" umgewandelt.
11 < 3       // => false: Numerischer Vergleich.
"11" < "3"   // => true: String-Vergleich.
"11" < 3     // => false: Numerischer Vergleich, "11" wird in 11 umgewandelt.
"one" < 3    // => false: Numerischer Vergleich, "one" wird in NaN konvertiert.
```

Beachten Sie bitte auch, dass sich die Kleiner-gleich- und Größer-gleich-Operatoren <= und >= nicht auf den nicht-strikten oder strikten Gleichheitsoperator 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 dazu umgewandelt wird). In diesem Fall liefern alle vier Vergleichsoperatoren false.

4.9.3 Der in-Operator

Der Operator in erwartet als linken Operanden einen String oder ein Symbol oder aber einen Wert, der in einen String konvertiert werden kann. Er erwartet als rechten Operanden ein Objekt. 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:

```
let point = {x: 1, y: 1};  // Ein Objekt definieren.
"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.

let data = [7,8,9];        // Ein Array mit Elementen an den Indizes 0, 1 und 2.
"0" in data                // => true: Array hat ein 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 ein Objekt als Operanden und auf der rechten Seite einen Operanden, der eine *Klasse* von Objekten angibt. Der Operator wird zu true ausgewertet, wenn das linke Objekt eine Instanz der rechten Klasse ist, ansonsten zu false. Kapitel 9 erläutert, dass in JavaScript Klassen von Objekten durch die Konstruktorfunktionen definiert werden, mit denen Objektinstanzen initialisiert werden. Der rechte Operand sollte beim instanceof-Operator also eine Funktion sein. Hier einige Beispiele:

```
let 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.
let a = [1, 2, 3];   // Erstellt ein Array mit 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 keine regulären Ausdrücke.
```

Beachten Sie bitte, dass alle Objekte Instanzen von Object sind. instanceof berücksichtigt auch eventuell vorhandene übergeordnete Klassen (sogenannte »Superklassen«), wenn es entscheidet, ob ein Objekt eine Instanz einer Klasse ist. Ist der linksseitige Operand von instanceof kein Objekt, liefert der Operator false. Ist die rechte Seite keine Objektklasse, wird ein TypeError ausgelöst.

Um genau zu verstehen, wie der instanceof-Operator funktioniert, müssen wir uns die »Prototypenkette« anschauen. Damit ist der Vererbungsmechanismus von JavaScript gemeint, der in 6.3.2 beschrieben wird. Im Ausdruck `o instanceof f` wertet JavaScript zuerst den Teilausdruck `f.prototype` aus und sucht dann in der Prototypenkette von `o` nach diesem Wert. Wenn er dort gefunden wird, ist `o` eine Instanz von `f` (oder eine Unterklasse von `f`), und der Operator gibt `true` zurück. Wenn `f.prototype` kein Wert in der Prototypenkette von `o` ist, dann ist `o` keine Instanz von `f`, und `instanceof` gibt `false` zurück.

4.10 Logische Ausdrücke

Die logischen Operatoren `&&`, `||` und `!` führen boolesche Operationen aus und werden häufig verwendet, um zwei relationale Ausdrücke zu einem komplexeren Ausdruck zu kombinieren. Diesen logischen Operatoren wenden wir uns in den folgenden Unterabschnitten zu. Als Hintergrund möchte ich noch einmal an das Konzept der »irgendwie wahren« bzw. nicht-strikt wahren *truthy*- und »irgendwie falschen« bzw. nicht-strikt falschen *falsy*-Werte erinnern, das in 3.4 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 eine logische UND-Operation auf beiden Werten durch. Der Operator liefert nur dann `true`, wenn der erste *und* der zweite Operand wahr sind. Ist einer oder sind beide Operanden `false`, wird `false` zurückgegeben.

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

```
x === 0 && y === 0   // true nur und nur dann, 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 `x === 0 && y === 0` können also problemlos ohne Klammern geschrieben werden.

Aber bei `&&` müssen die Operanden nicht unbedingt boolesche Werte sein. Erinnern Sie sich einfach daran, dass alle JavaScript-Werte entweder *truthy* oder *falsy* sind. (Einzelheiten dazu finden Sie in 3.4. Die falsy-Werte sind `false`, `null`, `undefined`, `0`, `-0`, `NaN` und `""`. Alle anderen Werte einschließlich aller Objekte sind truthy.) In zweiter Hinsicht kann `&&` als UND-Operator für solche nicht-strikt wahren und falschen Werte verstanden werden. Sind beide Operanden truthy, liefert der Operator einen truthy-Wert. Andernfalls – d.h., wenn mindestens einer der Operanden ein falsy-Wert ist – liefert er einen falsy-Wert. In JavaScript können alle Ausdrücke oder Anweisungen, die einen booleschen Wert erwarten, mit nicht-strikt wahren

und falschen Werten umgehen. Deswegen führt es in der Praxis zu keinerlei Problemen, dass && nicht immer true oder false liefert.

Die obige Beschreibung besagt übrigens, dass der Operator zwar einen »irgendwie wahren« Wert oder einen »irgendwie falschen« Wert liefert, aber nicht angibt, welcher Wert es genau ist. Dazu müssen wir && in dritter und letzter Hinsicht beschreiben. Der Operator beginnt bei der Auswertung mit dem ersten Operanden, dem Ausdruck auf der linken Seite. Ist der Wert dieses Operanden falsy, muss auch der Wert des gesamten Ausdrucks ein falsy-Wert sein. && liefert deswegen einfach den Wert auf der linken Seite und wertet den Ausdruck auf der rechten Seite nicht einmal mehr aus.

Ist der Wert auf der linken Seite hingegen ein truthy-Wert, hängt der Wert des gesamten Ausdrucks vom Wert auf der rechten Seite ab. Ist der Wert auf der rechten Seite ebenfalls ein truthy-Wert, ist auch der gesamte Ausdruck ein truthy-Wert. Ist der Wert auf der linken Seite dagegen ein falsy-Wert, ist auch der gesamte Ausdruck falsy. Wenn der Wert auf der linken Seite also ein truthy-Wert ist, wertet der Operator einfach den Wert auf der rechten Seite aus und liefert ihn zurück:

```
let o = {x: 1};
let p = null;
o && o.x     // => 1: o ist truthy, es wird also der Wert von o.x geliefert.
p && p.x     // => null: p ist falsy, also gib p zurück; p.x wird nicht mehr
             // ausgewertet.
```

Entscheidend dabei ist, dass der Operand auf der rechten Seite von && manchmal ausgewertet wird und manchmal nicht. In diesem Codebeispiel hatte die Variable p den Wert null. Der Ausdruck p.x hätte, wäre er ausgewertet worden, zu einem TypeError geführt. Aber der Code nutzt && auf idiomatische Weise derart, dass p.x nur ausgewertet wird, wenn p truthy ist – und damit niemals null oder undefined.

Das Verhalten von && wird gelegentlich als »kurzschließend« bezeichnet (dieses Konzept eines vorzeitigen Abbruchs einer Auswertung ist uns bereits bei bedingten Eigenschaftszugriffen und bedingten Aufrufen begegnet), und Sie können gelegentlich auf Code stoßen, der es absichtlich einsetzt, damit Code nur bedingt ausgeführt wird. Die folgenden beiden Codezeilen haben beispielsweise die gleiche Wirkung:

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

Im Allgemeinen sollten Sie aufpassen, wenn Sie auf der rechten Seite von && Ausdrücke mit möglichen Seiteneffekten einsetzen (Zuweisungen, Inkremente, Dekremente, Funktionsaufrufe). Ob diese Seiteneffekte eintreten, hängt vom Wert auf der linken Seite ab.

Trotz der recht komplexen Funktionsweise dieses Operators wird er am häufigsten als einfacher logischer Operator benutzt, der mit truthy- und falsy-Werten umgehen kann.

4.10.2 Logisches ODER (||)

Der ||-Operator verknüpft die beiden Operanden mit einem logischen ODER. Sind beide Operanden truthy-Werte, liefert der Operator einen truthy-Wert. Sind beide Operanden falsy-Werte, liefert der Operator einen falsy-Wert.

Obwohl der ||-Operator in der Regel meist einfach als boolescher ODER-Operator zum Einsatz kommt, weist er wie der &&-Operator ein komplexeres Verhalten auf. Der Operator beginnt bei der Auswertung mit dem ersten Operanden, dem Ausdruck auf der linken Seite. Wenn der Wert dieses ersten Operanden truthy ist, agiert der Operator »kurzschließend« und gibt den Ausdruck auf der linken Seite zurück, ohne den Ausdruck auf der rechten Seite überhaupt auszuwerten. Wenn der Wert des ersten Operanden dagegen falsy ist, wertet || den zweiten Operanden aus und gibt den Wert dieses Ausdrucks zurück.

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

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

```
// Wenn maxWidth truthy ist, nimm diesen Wert. Untersuche andernfalls
// das Objekt preferences. // Ist es nicht truthy, nimm die angegebene Konstante.
let max = maxWidth || preferences.maxWidth || 500;
```

Beachten Sie aber bitte, dass diese Konstruktion nur korrekt funktioniert, wenn `maxWidth` niemals 0 wird, da 0 ein falsy-Wert ist. Der ??-Operator (siehe 4.13.2) bietet eine Alternative.

Vor ES6 wurde dieses Idiom häufig in Funktionen verwendet, um Standardwerte für Parameter anzugeben:

```
// Die Eigenschaften von o auf p kopieren und p zurückgeben.
function copy(o, p) {
    p = p || {};  // Wurde für p kein Objekt übergeben, nutze ein neu erstelltes.
    // Hier folgt der Funktionskörper.
}
```

In ES6 und später wird dieser Trick jedoch nicht mehr benötigt, da der Standardwert für einen Parameter einfach in die Funktionsdefinition selbst aufgenommen werden kann: `function copy(o, p={}) { ... }`.

4.10.3 Logisches NICHT (!)

Der !-Operator ist ein unärer Operator. Er steht vor seinem einzigen Operanden und hat die Aufgabe, den logischen Wahrheitswert seines Operanden zu verneinen. Ist beispielsweise `x` ein truthy-Wert, wird `!x` zu `false` ausgewertet. Wenn `x` falsy ist, wird `!x` zu `true` ausgewertet.

Im Unterschied zu den Operatoren && und || wandelt der !-Operator seinen Operanden in einen booleschen Wert um (und nutzt dabei die in Kapitel 3 beschriebenen Regeln), bevor er den dabei erhaltenen Wert negiert bzw. umkehrt. Das heißt, dass ! immer true oder false liefert und Sie deswegen einen beliebigen Wert x in seinen zugehörigen Wahrheitswert umwandeln können, indem Sie den Operator einfach zweimal anwenden: !!x (siehe 3.9.2).

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 auf folgende Weise in JavaScript-Syntax ausdrücken können:

```
// Die zwei de-morganschen Gesetze
!(p && q) === (!p || !q)  // => true: Für alle Werte von p und q.
!(p || q) === (!p && !q)  // => true: Für alle Werte von p und q.
```

4.11 Zuweisungsausdrücke

JavaScript verwendet 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 im selben Ausdruck sowohl testen wie auch 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 bitte, dass = einen sehr geringen Vorrang hat und deshalb 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 mit dem Wert 0.
```

4.11.1 Zuweisung mit Operation

Neben dem normalen Zuweisungsoperator = unterstützt JavaScript eine Reihe weiterer Zuweisungsoperatoren, bei denen die eigentliche Zuweisung mit einer anderen Operation kombiniert wird. Beispielsweise führt der +=-Operator sowohl eine Addition als auch eine Zuweisung aus. Der Ausdruck

```
total += salesTax;
```

ist gleichwertig mit diesem:

```
total = total + salesTax;
```

Wie Sie sich vielleicht schon gedacht haben, kann der +=-Operator mit Strings und Zahlen umgehen. Bei numerischen Operanden führt er Addition und Zuweisung aus, bei String-Operanden Verkettung und Zuweisung.

Ähnliche Operatoren sind -=, *=, &= usw. Tabelle 4-2 zeigt sie in einer Übersicht.

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
^=	a ^= b	a = a ^ b

In den meisten Fällen ist der Ausdruck

```
a op= b
```

(mit op als beliebigem Operator) äquivalent zum Ausdruck

```
a = a op b
```

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 hat, weil dieser Ausdruck beispielsweise einen Funktionsaufruf oder eine Inkrementierungsoperation enthält. Die beiden folgenden Zuweisungen sind beispielsweise nicht gleich:

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

4.12 Auswertungsausdrücke

Wie viele interpretierte Programmiersprachen kann JavaScript Strings interpretieren, die spracheigenen Quellcode enthalten, und diese auswerten, um einen Wert zu erzeugen. In JavaScript benutzt man dazu die globale Funktion `eval()`:

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

Die dynamische Auswertung von Strings mit Quellcode ist ein mächtiges Sprachmerkmal, das man in der Praxis allerdings fast nie braucht. Wenn Sie `eval()` einsetzen, sollten Sie noch einmal sorgfältig überdenken, ob Sie es tatsächlich benötigen. Damit `eval()` nicht zu einem Sicherheitsrisiko wird, sollten Sie vor allem niemals einen String, der auf irgendwelche Benutzereingaben zurückgeht, an `eval()` übergeben. Bei einer so komplizierten Sprache wie JavaScript besteht keine Möglichkeit, Benutzereingaben derart von möglichem Schadcode zu bereinigen, dass die Eingaben sicher mit `eval()` verwendet werden können. Wegen dieser Sicherheitsprobleme verwenden einige Webserver den HTTP-Header `Content-Security-Policy`, um `eval()` für eine ganze Website zu deaktivieren.

In den folgenden Unterabschnitten besprechen wir zuerst die allgemeine Verwendung von `eval()`, bevor wir zu zwei eingeschränkten Versionen kommen, die geringere Auswirkungen auf den JavaScript-Optimierer haben.

Ist eval() eine Funktion oder ein Operator?

`eval()` ist eine Funktion, wird aber trotzdem in diesem Kapitel über Ausdrücke und Operatoren behandelt, weil es eigentlich ein Operator hätte sein sollen. Die `eval()`-Funktion gibt es bereits seit den frühesten Versionen von JavaScript, aber die Sprachentwickler und Programmierer des Interpreters haben sie seitdem kontinuierlich mit Einschränkungen versehen, die zu immer größerer Ähnlichkeit mit einem Operator geführt haben. Moderne JavaScript-Interpreter führen eine Vielzahl von Codeanalysen und -optimierungen durch. Allgemein gesagt: Ruft eine Funktion `eval()` auf, kann der Interpreter diese Funktion nicht mehr optimieren. Weil `eval()` als Funktion vorliegt, kann man ihr einen anderen Namen geben – und das ist das Problem:

```
let f = eval;
let g = f;
```

In solchen Fällen kann der Interpreter nicht mehr mit Sicherheit erkennen, welche Funktionen `eval()` aufrufen, und deshalb nicht mehr aggressiv optimieren. Dieses Problem hätte man vermeiden können, wäre `eval()` tatsächlich ein Operator (und ein reserviertes Wort). Wir werden weiter unten (in 4.12.2 und 4.12.3) die Einschränkungen kennenlernen, denen man `eval()` unterworfen hat, damit es eher einem Operator ähnelt.

4.12.1 eval()

`eval()` erwartet ein Argument. Übergeben Sie einen anderen Wert als einen String, liefert es einfach diesen Wert zurück. Ansonsten versucht es, den übergebenen String als JavaScript-Code zu parsen, und löst einen SyntaxError aus, falls 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 zurück oder aber `undefined`, wenn der letzte Ausdruck bzw. die letzte Anweisung keinen Wert hat. Löst die ausgewertete Zeichenkette eine Ausnahme aus, wird diese Ausnahme an `eval()` weitergereicht.

Wird `eval()` auf diese Weise eingesetzt (und das ist hier wesentlich), nutzt es die Variablenumgebung des Codes, aus dem es aufgerufen wird. Damit liest die Funktion Werte von Variablen auf gleiche Weise aus und definiert neue Variablen und Funktionen auf gleiche Weise wie der lokale Umgebungscode. 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, deklariert sie eine neue lokale Variable `y`. Wird in der auszuwertenden Zeichenkette dagegen `let` oder `const` verwendet, gilt die deklarierte Variable oder Konstante nur lokal innerhalb von `eval()` (d.h. innerhalb der Auswertung) und wird nicht in der aufrufenden Umgebung definiert.

In ähnlicher Weise kann eine Funktion beispielsweise eine lokale Funktion deklarieren:

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

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

Beachten Sie bitte, dass der Codestring, den Sie an `eval()` übergeben, syntaktisch für sich stehend sinnvoll sein muss. Sie können `eval()` nicht nutzen, um Codefragmente in eine Funktion einzubringen. 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. Wäre Ihr String ein eigenständiges Skript (und sei es nur ein ganz kurzes wie `x=0`), dürfte er an `eval()` übergeben werden. Andernfalls löst `eval()` einen SyntaxError aus.

4.12.2 Globales eval()

`eval()` ist für den JavaScript-Optimierer deshalb so problematisch, weil es lokale Variablen ändern kann. Um das zu umgehen, führen Interpreter bei Funktionen, die `eval()` aufrufen, einfach weniger Optimierungen aus. Aber was soll ein JavaScript-Interpreter tun, wenn ein Skript ein Alias für `eval()` definiert und die Funktion einfach unter einem anderen Namen aufruft? Die JavaScript-Spezifikation sieht vor, dass bei einem Aufruf von `eval()` unter einem anderen Namen die Zeichenkette so ausgewertet werden soll, als würde es sich um globalen Code der höchsten Programmebene handeln. Der ausgewertete Code darf neue globale Variablen oder Funktionen deklarieren und globale Variablen erstellen, darf aber in der aufrufenden Funktion keine Variablen mehr nutzen oder verändern und kommt so den lokalen Optimierungen nicht mehr ins Gehege.

Ein »direktes Eval« ist ein Aufruf der `eval()`-Funktion mit einem Ausdruck, der den genauen, nicht qualifizierten Namen »eval« verwendet (der damit immer mehr wie ein reserviertes Wort wirkt). Direkte Aufrufe von `eval()` nutzen die Variablenumgebung des aufrufenden Kontexts. Jeder andere – indirekte – Aufruf nutzt als Variablenumgebung das globale Objekt und kann lokale Variablen oder Funktionen weder lesen noch schreiben noch definieren. (Sowohl direkte als auch indirekte Aufrufe können neue Variablen nur mit `var` definieren. Die Verwendung von `let` und `const` innerhalb einer auszuwertenden Zeichenkette erzeugt Variablen und Konstanten, die lokal zur Auswertung gehören und die aufrufende oder globale Umgebung nicht verändern.)

Der folgende Code verdeutlicht dies:

```
const geval = eval;               // Die Verwendung eines anderen Namens führt zu
                                  // einem globalen Eval.
let x = "global", y = "global";   // Zwei globale Variablen.
function f() {                    // Diese Funktion führt ein lokales Eval aus.
    let x = "local";              // Eine lokale Variable definieren.
    eval("x += 'changed';");      // Das direkte Eval setzt die lokale Variable.
    return x;                     // Den geänderten lokalen Wert zurückliefern.
}
function g() {                    // Diese Funktion führt ein globales Eval aus.
    let y = "local";              // Eine lokale Variable.
    geval("y += 'changed';");     // Ein indirektes Eval setzt die globale
                                  // Variable.
    return y;                     // Liefert den unveränderten lokalen Wert.
}
console.log(f(), x); // Lokale Variable geändert: ergibt "localchanged global".
console.log(g(), y); // Globale Variable geändert: ergibt "local globalchanged".
```

Die Möglichkeit, ein globales Eval durchzuführen, ist übrigens mehr als eine Anpassung an die Anforderungen des Optimierers. Das globale Eval ist tatsächlich eine unglaublich nützliche Einrichtung, mit der man Strings mit Code ausführen kann, als enthielten sie unabhängige Skripte der obersten Ebene. Wie bereits zu Anfang dieses Abschnitts erwähnt, kommt es nur äußerst selten vor, dass man

Strings mit Code auswerten muss. Wenn Sie aber das Gefühl haben, es sei notwendig, ist meist ein globales Eval die bessere Lösung als ein lokales.

4.12.3 eval() im strict-Modus

Der `strict`-Modus (siehe 5.6.3) unterwirft das Verhalten der Funktion `eval()` und sogar die Verwendung des Identifiers »eval« weiteren Einschränkungen. Wird `eval()` aus Code aufgerufen, der im `strict`-Modus ausgeführt wird, oder beginnt der auszuwertende Code selbst mit der Direktive `"use strict"`, führt `eval()` ein lokales Eval mit einer privaten Variablenumgebung aus. Das bedeutet, dass im `strict`-Modus evaluierter Code lokale Variablen abfragen und setzen, aber keine neuen Variablen oder Funktionen im lokalen Bereich definieren kann.

Außerdem gleicht der `strict`-Modus `eval()` einem Operator noch stärker an, indem er »eval« praktisch zu einem reservierten Wort umfunktioniert. Es ist nicht erlaubt, die `eval()`-Funktion mit einem neuen Wert zu überschreiben. Und es ist nicht erlaubt, eine Variable, eine Funktion, einen Funktionsparameter oder einen Parameter eines `catch`-Blocks mit dem Namen »eval« zu deklarieren.

4.13 Weitere Operatoren

JavaScript unterstützt eine Reihe weiterer Operatoren mit unterschiedlichen Aufgaben, die in den folgenden Abschnitten beschrieben werden.

4.13.1 Der Bedingungsoperator (?:)

Der Bedingungsoperator ist der einzige *Ternäroperator* – also ein Operator, der mit drei Operanden arbeitet – in JavaScript und wird in der Tat gelegentlich einfach als »der Ternäroperator« bezeichnet. Dieser Operator wird auch oft in der Form `?:` geschrieben, obwohl er im Code nicht ganz in dieser Form erscheint. Weil dieser Operator drei Operanden hat, steht der erste vor dem `?`, der zweite zwischen `?` und `:` und der dritte nach dem `:`. Er wird wie folgt verwendet:

```
x > 0 ? x : -x     // Der Absolutwert (Betrag) von x.
```

Die Operanden des Bedingungsoperators können beliebigen Typs sein. Der erste Operand wird ausgewertet und als boolescher Wert interpretiert. Ist der erste Operand ein nicht-strikt wahrer Wert, wird der zweite Operand ausgewertet, und dessen Wert wird zurückgeliefert. Ist der erste Operand hingegen ein nicht-strikt (»irgendwie«) falscher Wert, wird der dritte Operand ausgewertet, und dessen Wert wird zurückgeliefert. Es wird immer *entweder* der erste *oder* der zweite Operand ausgewertet.

Obgleich man mit der `if`-Anweisung (siehe 5.3.1) Ähnliches erreichen kann, erweist sich der `?:`-Operator häufig als praktische Kurzform. Das folgende Beispiel zeigt eine typische Verwendung. Dabei wird geprüft, ob eine Variable definiert ist

und einen sinnvollen, wahren Wert hat. Ist das der Fall, wird dieser Wert verwendet, andernfalls wird auf einen Vorgabewert ausgewichen:

```
greeting = "hello " + (username ? username : "there");
```

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

```
greeting = "hello ";
if (username) {
    greeting += username;
} else {
    greeting += "there";
}
```

4.13.2 ?? – der Erstdefiniert-Operator

Der Operator `??` wird zu seinem *ersten definierten* Operanden ausgewertet: Wenn sein linker Operand nicht `null` und nicht `undefined` ist, wird dieser Wert zurückgeliefert. Andernfalls gibt er den Wert des rechten Operanden zurück. Wie die Operatoren `&&` und `||` ist auch `??` ein »kurzschließender« Operator: Er wertet seinen zweiten Operanden nur dann aus, wenn der erste Operand `null` oder `undefined` ergibt. Wenn der Ausdruck `a` keine Seiteneffekte hat, ist der Ausdruck `a ?? b` äquivalent zu:

```
(a !== null && a !== undefined) ? a : b
```

`??` ist eine nützliche Alternative zu `||` (siehe 4.10.2), wenn Sie den ersten *definierten* Operanden und nicht den ersten »irgendwie wahren« Operanden auswählen möchten. Obwohl `||` nominell ein logischer ODER-Operator ist, wird er in JavaScript auch gern verwendet, um den ersten »irgendwie falschen« Operanden auszuwählen, beispielsweise so:

```
// Wenn maxWidth truthy ist, nimm diesen Wert, andernfalls einen Wert
// im Objekt preferences. Ist das nicht truthy, nimm die angegebene Konstante.
let max = maxWidth || preferences.maxWidth || 500;
```

Das Problem bei dieser Verwendung ist, dass null (0), die leere Zeichenkette und `false` allesamt nicht-strikt falsche Werte sind, die unter bestimmten Umständen durchaus valide sein können. Wenn in diesem Codebeispiel `maxWidth` gleich null (0) ist, wird dieser Wert ignoriert. Aber wenn wir den `||`-Operator durch `??` ersetzen, erhalten wir einen Ausdruck, in dem 0 ein gültiger Wert ist:

```
// Wenn maxWidth definiert ist, nimm diesen Wert, andernfalls einen Wert
// im Objekt preferences. Wenn es auch den nicht gibt, greife auf die Konstante
// zurück.
let max = maxWidth ?? preferences.maxWidth ?? 500;
```

Hier sind weitere Beispiele, die zeigen, wie `??` funktioniert, wenn der erste Operand nicht-strikt falsch ist. Ist er zwar falsy, aber definiert, gibt `??` diesen ersten Operanden zurück. Nur wenn er »nullish« ist (also »nullartig«: `null` oder `undefiniert`), wertet dieser Operator den zweiten Operanden aus und gibt diesen zurück:

```
let options = { timeout: 0, title: "", verbose: false, n: null };
options.timeout ?? 1000     // => 0: wie im Objekt definiert.
options.title ?? "Untitled" // => "": wie im Objekt definiert.
options.verbose ?? true     // => false: wie im Objekt definiert.
options.quiet ?? false      // => false: Eigenschaft ist nicht definiert.
options.n ?? 10             // => 10: Eigenschaft ist null.
```

Beachten Sie bitte, dass die Ausdrücke `timeout`, `title` und `verbose` hier unterschiedliche Werte hätten, wenn wir `||` anstelle von `??` verwendeten.

Der `??`-Operator ähnelt den Operatoren `&&` und `||`, hat aber weder höheren noch niedrigeren Vorrang als diese. Wenn Sie ihn in einem Ausdruck zusammen mit einem dieser Operatoren einsetzen, müssen Sie explizite Klammern verwenden, um die Ausführungsreihenfolge festzulegen:

```
(a ?? b) || c   // ?? zuerst, dann ||
a ?? (b || c)   // || zuerst, dann ??
a ?? b || c     // SyntaxError: Hier sind Klammern erforderlich.
```

Der `??`-Operator wurde mit ES2020 eingeführt und wird Ende 2020 von allen gängigen Desktopbrowsern unterstützt (bei Browsern für Mobilgeräte bestehen noch kleine Lücken). Dieser Operator wird formal als »Nullish-Coalescing«-Operator bezeichnet (wörtlich im Deutschen etwa »nullartiger Zusammenführungsoperator«). Ich vermeide den Begriff aber, weil dieser Operator zwar einen seiner Operanden auswählt, sie aber in keiner für mich erkennbaren Weise »zusammenführt«. Deshalb bezeichne ich ihn lieber als *Erstdefiniert-Operator* (*First-defined Operator*), weil er den ersten definierten Operanden auswählt.

4.13.3 Der typeof-Operator

`typeof` ist ein unärer Operator, der einem Operanden beliebigen Typs vorangestellt wird. Sein Ergebniswert ist ein String, der den Typ des Operanden angibt. Tabelle 4-3 zeigt den Wert des `typeof`-Operators für alle JavaScript-Typen.

Tabelle 4-3: Werte, die vom typeof-Operator zurückgegeben werden

x	typeof x
undefined	"undefined"
null	"object"
true oder false	"boolean"
eine Zahl oder NaN	"number"
ein BigInt	"bigint"
ein String	"string"
ein Symbol	"symbol"
eine Funktion	"function"
ein Objekt, das keine Funktion ist	"object"

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

```
// Wenn der Wert eine Zeichenkette ist, wird sie in Anführungszeichen gesetzt,
// andernfalls umgewandelt.
(typeof value === "string") ? "'" + value + "'" : value.toString()
```

Beachten Sie, dass typeof »object« liefert, wenn der Wert des Operanden null ist. Wenn Sie null von Objekten unterscheiden wollen, müssen Sie explizit auf diesen Sonderfall testen.

Obwohl JavaScript-Funktionen so etwas wie Objekte sind, betrachtet der typeof-Operator Funktionen als ausreichend anders – deshalb gibt es für sie einen eigenen Rückgabewert.

Weil typeof für alle Objekt- und Array-Werte außer Funktionen zu »object« ausgewertet wird, ist der Operator nur zur Unterscheidung zwischen Objekten und anderen primitiven Typen geeignet. Wollen Sie eine Klasse von Objekten von einer anderen unterscheiden, müssen Sie andere Techniken wie den instanceof-Operator (siehe 4.9.4), das class-Attribut (siehe 14.4.3) oder die constructor-Eigenschaft (siehe 9.2.2 und 14.3) nutzen.

4.13.4 Der delete-Operator

delete ist ein unärer Operator, der versucht, die Objekteigenschaft oder das Array-Element zu löschen, das als sein Operand angegeben ist. Wie die Operatoren für Zuweisung, Inkrementierung und Dekrementierung wird delete üblicherweise wegen seines Seiteneffekts, Eigenschaften zu löschen, eingesetzt und nicht wegen des Werts, den er liefert. Ein paar Beispiele:

```
let o = { x: 1, y: 2}; // Beginnen wir mit einem Objekt.
delete o.x;            // Eine seiner Eigenschaften löschen.
"x" in o               // => false: Die Eigenschaft gibt es nicht mehr.

let a = [1,2,3];       // Ein Array definieren.
delete a[2];           // Das letzte Array-Element löschen.
2 in a                 // => false: Das Array-Element mit Index 2 existiert
                       // nicht mehr.
a.length               // => 3: Beachten Sie, dass sich die Länge des Arrays
                       // nicht ändert.
```

Übrigens wird eine gelöschte Eigenschaft oder ein gelöschtes Array-Element nicht einfach auf undefined gesetzt. Wird eine Eigenschaft gelöscht, endet ihre Existenz. Ein Versuch, eine nicht (mehr) vorhandene Eigenschaft zu lesen, liefert undefined. Sie können aber auf die tatsächliche Existenz einer Eigenschaft mit dem in-Operator (siehe 4.9.3) prüfen. Löscht man ein Array-Element, entsteht eine »Lücke« im Array – die Länge des Arrays ändert sich dabei nicht. Das resultierende Array ist ein sogenanntes *Sparse-Array* (siehe 7.3).

delete erwartet, dass sein Operand ein Lvalue ist. Ist er kein Lvalue, unternimmt der Operator nichts und liefert true. Andernfalls versucht delete, das angegebene

Lvalue zu löschen. delete gibt true zurück, wenn der Operator das Lvalue erfolgreich löschen konnte. Allerdings sind nicht alle Eigenschaften löschbar: Nicht konfigurierbare Eigenschaften (siehe 14.1) sind immun gegen Löschungen.

Im strict-Modus löst delete einen SyntaxError aus, wenn sein Operand ein nicht qualifizierter Identifier wie eine Variable, eine Funktion oder ein Funktionsparameter ist. Der Operator funktioniert also nur, wenn der Operand ein Eigenschaftszugriffsausdruck (siehe 4.4) ist. Der strict-Modus sorgt auch dafür, dass delete einen TypeError auslöst, wenn versucht wird, eine nicht konfigurierbare Eigenschaft zu löschen. Außerhalb des strict-Modus treten in diesen Fällen keine Ausnahmen auf. delete liefert einfach false, um anzuzeigen, dass der Operand nicht gelöscht werden konnte.

Hier sind einige Beispiele zur Verwendung des delete-Operators:

```
let o = {x: 1, y: 2};
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 1;     // Das ergibt zwar keinen Sinn, aber es liefert als Ergebnis true.
// Eine Variable kann nicht gelöscht werden; gibt false oder – im strict-Modus –
// einen SyntaxError zurück:
delete o;
// Nicht löschbare Eigenschaft; gibt false oder – im strict-Modus – einen
// TypeError zurück:
delete Object.prototype;
```

Der delete-Operator wird uns noch einmal in 6.4 begegnen.

4.13.5 Der await-Operator

await wurde in ES2017 als Möglichkeit eingeführt, die asynchrone Programmierung in JavaScript natürlicher zu gestalten. Um diesen Operator genau kennenzulernen, empfehle ich Ihnen Kapitel 13. Man kann den Operator aber kurz so beschreiben: await erwartet ein Promise-Objekt (das eine asynchrone Operation repräsentiert) als einzigen Operanden und veranlasst Ihr Programm, sich so zu verhalten, als würde es auf den Abschluss der asynchronen Operation warten. (Dies geschieht aber, ohne die Programmausführung tatsächlich zu blockieren, und es verhindert auch nicht, dass andere asynchrone Operationen zur gleichen Zeit ablaufen.) Der Wert des await-Operators ist der Erfüllungswert des Promise-Objekts. Wichtig dabei ist, dass await nur innerhalb von Funktionen zulässig ist, die mit dem Schlüsselwort async als asynchron deklariert wurden. Alle weiteren Einzelheiten finden Sie in Kapitel 13.

4.13.6 Der void-Operator

void ist ein unärer Operator, der vor einem Operanden beliebigen Typs steht. Dieser Operator ist ungewöhnlich und wird nur selten eingesetzt: Er wertet seinen

Operanden aus, verwirft den Wert aber dann und liefert `undefined`. Da der Wert des Operanden verworfen wird, ist `void` nur sinnvoll, wenn der Operand Seiteneffekte hat.

Der Operator `void` ist derart seltsam, dass es schwierig ist, ein praktisches Beispiel für seine Anwendung zu finden. Aber nehmen wir an, Sie möchten eine Funktion definieren, die nichts zurückgibt, dabei aber die Pfeilnotation verwendet (siehe 8.1.3), bei der der Körper der Funktion also ein einzelner Ausdruck ist, der ausgewertet und zurückgegeben wird. Wenn Sie den Ausdruck nur wegen seiner Seiteneffekte auswerten und seinen Wert nicht zurückgeben wollen, ist es am einfachsten, den Funktionskörper in geschweifte Klammern einzufassen. Als Alternative könnten Sie in diesem Fall aber auch den `void`-Operator verwenden:

```
let counter = 0;
const increment = () => void counter++;
increment()   // => undefined
counter       // => 1
```

4.13.7 Der Kommaoperator (,)

Der Kommaoperator ist ein binärer Operator, dessen Operanden beliebigen Typs sein können. Er wertet seinen linken Operanden aus, dann den rechten Operanden und liefert schließlich den Wert des rechten Operanden zurück. Die folgende Zeile

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

wird deshalb zu 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. Die einzige Situation, in der der Kommaoperator häufig verwendet wird, ist eine `for`-Schleife (siehe 5.4.3), die mehrere Schleifenvariablen hat:

```
// Das erste Komma unten ist Teil der let-Anweisung,
// das zweite der Kommaoperator: Damit quetschen wir zwei
// Ausdrücke (i++ und j--) in eine Anweisung (die for-Schleife),
// die nur einen erwartet.
for(let i=0,j=10; i < j; i++,j--) {
    console.log(i+j);
}
```

4.14 Zusammenfassung

Dieses Kapitel deckt eine Vielzahl von Themen ab und enthält eine Menge Referenzmaterial, zu dem Sie vielleicht im weiteren Verlauf Ihrer Arbeit mit JavaScript zurückkehren möchten. Einige wichtige, merkenswerte Punkte aus diesem Kapitel:

- Ausdrücke sind die (in 1.3 so bezeichneten) »Phrasen« eines JavaScript-Programms.
- Jeder Ausdruck kann zu einem JavaScript-Wert *ausgewertet* werden.
- Ausdrücke erzeugen nicht nur Werte, sondern können auch Seiteneffekte haben, beispielsweise eine Variablenzuweisung.
- Einfache Ausdrücke wie Literale, Variablenreferenzen und Eigenschaftszugriffe können mit Operatoren kombiniert werden, um komplexere Ausdrücke zu erzeugen.
- JavaScript bietet Operatoren für Arithmetik, Vergleiche, boolesche Logik, Zuweisungen und bitweise Manipulationen sowie verschiedene andere Operatoren einschließlich des ternären Bedingungsoperators.
- Der Operator + kann sowohl Zahlen addieren als auch Strings verketten.
- Die logischen Operatoren && und || haben ein spezielles »kurzschließendes« Verhalten und werten manchmal nur eines ihrer Argumente aus. Da von diesem speziellen Verhalten in vielen Fällen Gebrauch gemacht wird, sollten Sie die Funktionsweise dieser Operatoren genau verstehen.