Oliver Zeigermann

JavaScript für Java-Entwickler

Oliver Zeigermann

JavaScript für Java-Entwickler

schnell+kompakt

entwickler.press

Oliver Zeigermann
JavaScript für Java-Entwickler
schnell+kompakt
ISBN: 978-3-86802-295-7

ein Imprint der Software & Support Media GmbH

http://www.entwickler-press.de
http://www.software-support.biz

Ihr Kontakt zum Verlag und Lektorat: lektorat@entwickler-press.de

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

Lektorat: Sebastian Burkart
Korrektorat: Frauke Pesch
Satz: Karolina Gaspar, Dominique Kalbassi
Umschlaggestaltung: Maria Rudi

Inhaltsverzeichnis

Einleitung

JavaScript sieht syntaktisch aus wie ein vereinfachtes Java. Somit haben viele Java-Entwickler das Gefühl, die Sprache JavaScript eigentlich zu kennen und deshalb nicht lernen zu müssen. Allerdings ist das *Verhalten* von JavaScript deutlich anders als das von Java. Dies führt zu einer großen Anzahl von Java-Entwicklern, die zwar JavaScript nutzen, aber nie wirklich die Grundlagen der Sprache studiert haben.

Dieses Buch ist für Java-Entwickler gedacht, die mit so wenig Mühen wie möglich einen umfassenden Überblick über die Sprache JavaScript erlangen wollen oder müssen. Missverständnisse werden ausgeräumt und eine Beherrschung der Muster und Grundkonzepte von JavaScript werden vermittelt.

Bibliotheken und Frameworks werden in diesem Buch ganz bewusst und ausdrücklich nicht behandelt. Hier geht es ausschließlich um die *Sprache* JavaScript und Patterns, die für uns als Java-Entwickler wichtig sind. Die Beschreibung der Sprache und der Patterns geschieht unabhängig von allen Bibliotheken und Frameworks, d. h. egal, welches Framework oder welche Bibliothek ihr einsetzen wollt, dieses Buch vermittelt euch die dazu notwendigen Grundlagen der Sprache JavaScript.

Dieses Buch ist keine Referenz und erhebt keinen Anspruch auf Vollständigkeit. Ich verzichte auf alle Details, die nicht wirklich notwendig für das Verständnis der Sprache sind. Zu jedem Thema gibt es aber Referenzen auf die *ECMAScript*-Spezifikation,

Erklärungen beim *Mozilla Development Network* [5] oder andere passende Links. Damit sollten keine Fragen offen bleiben.

Ich beziehe mich ausschließlich auf die aktuellste ECMAScript-Version 5.1 [1, 2]. Diese Version wird von allen modernen Browsern ab Internet Explorer 9 unterstützt. Eine kurze Erklärung zu ECMAScript: ECMAScript ist der Standard und JavaScript ist dazu eine Implementierung.

Ich rege dazu an, allen Beispielcode inklusive aller Zwischenschritte selbst auszuprobieren. Wem das Abtippen bzw. Kopieren zu mühsam ist, der kann auch gern das *GitHub*-Repository zu diesem Buch auschecken. In ihm sind sämtliche Beispiele nach Kapiteln geordnet zu finden [3].

Inhalt

Ihr könnt dieses Buch von vorn bis hinten durchlesen. In diesem Fall werden alle notwendigen Grundlagen der Sprache JavaScript vermittelt – zugeschnitten auf Java-Entwickler.

Je nach Interesse, Zeit und Vorkenntnissen könnt ihr aber auch nur einzelne Kapitel lesen. Wenn ihr noch wenig oder keine Erfahrung mit JavaScript gemacht habt, solltet ihr zumindest Kapitel 2 (Grundlagen) und 3 (Funktionen) lesen. Sie bilden die Grundlage für die folgenden Kapitel. Insbesondere in Kapitel 3 sehen wir einiges, was auch für manche erfahrene JavaScript-Programmierer neu sein könnte.

In Kapitel 4 (Objekte, Prototypen und Vererbung) gucken wir uns an, wie Vererbung in JavaScript funktioniert und wie man die aus Java bekannten Mechanismen von Klassen und Vererbung auch in der JavaScript-Welt anwenden kann. Dazu nutzen wir einige Best Practices. Dieses Kapitel ist das konzeptionell anspruchsvollste des Buchs. Ich empfehle, es an einem Stück und evtl. zweimal zu lesen.

In Kapitel 5 (Module) schauen wir auf Modulkonzepte und Closures. Für dieses Kapitel solltet ihr das Wissen aus Kapitel 2 und 3 haben, Kapitel 4 ist nicht unbedingt notwendig. Wir schließen das Kapitel mit einer Betrachtung der gängigen Modulformate *AMD* und *CommonJS*.

Im letzten Kapitel (Fortgeschrittene Themen) kommen alle wichtigen Themen, die keine Grundlagen mehr sind. Hier müsst ihr nicht alles lesen, sondern könnt euch die Themen herauspicken, die euch interessieren. Besonders spannend sind dabei zusätzliche OO-Muster, die allerdings ein Verständnis der Themen aus Kapitel 4 erfordern.

Der Autor und warum dieses Buch?

Ich – das ist Oliver Zeigermann – programmiere seit über zehn Jahren in Java. Im Laufe der letzten Jahre habe ich mich – zunächst widerwillig – mit JavaScript angefreundet und programmiere nun sogar lieber JavaScript als Java. Dazu musste ich mich durch viele Missverständnisse und Wirrung kämpfen, die oft durch meinen Hintergrund als Java-Programmierer bedingt waren.

Idealerweise erspart euch die Lektüre dieses Buchs viel von dem Frust und der Verwirrung, die ich selbst erlebt habe. Das würde mich sehr freuen! Für Rückmeldungen oder Fragen könnt ihr mich gern per E-Mail kontaktieren. Meine Kontaktdaten findet ihr unter [4].

Danksagung

Ich möchte mich bei allen bedanken, die bei diesem Buch mitgeholfen haben. Das sind in alphabetischer Reihenfolge: Sebastian Burkart, Daniel Florey, Nils Hartmann, Lutz Hühnken, Markus Klink, Charlotte Krause, Oliver Langer, René Preißel, Christian

Schmidt, Alexander Weber und Stefan Zörner. Vielen Dank an euch und die anderen, die ich nur deshalb nicht erwähnt habe, weil ich vergesslich bin.

Links & Literatur

[1] HTML-Version der ECMAScript-Spezifikation 5.1: *http://www.ecma-international.org/ecma-262/5.1*

[2] PDF-Version der ECMAScript-Spezifikation 5.1: *http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf*

[3] Das GitHub-Repository für dieses Buch: *https://github.com/DJCordhose/javascript-fuer-java-entwickler*

[4] Meine Homepage: *http://zeigermann.eu/*

[5] Das Mozilla Development Network für JavaScript: *https://developer.mozilla.org/en-US/docs/Web/JavaScript*

Grundlagen

2.1 Hallo Welt

Die typische Ablaufumgebung für ein JavaScript-Programm ist nach wie vor der Browser. Zwar gibt es seit einiger Zeit mit Node.js [6] die Möglichkeit, JavaScript auch auf dem Server und von der Kommandozeile aufzurufen, wir werden uns hier aber auf die Ausführung im Browser beschränken.

Im Browser läuft JavaScript durch Einbetten in eine HTML-Seite oder durch Ausführen in der JavaScript-Konsole. Alle Browser haben eine solche Konsole. In meinen Beschreibungen beziehe ich mich speziell auf den Chrome-Browser, weil er der am weitesten verbreitete Browser ist und auf allen Plattformen läuft. Zudem sind die Entwicklertools von Chrome sehr gut.

PROFITIPP: Die Entwicklertools des Chrome-Browsers sind sehr mächtig und helfen beim schnellen Ausprobieren und Debuggen von JavaScript-Code.

Für den Firefox-Browser gibt es die Firebug-Erweiterung [7], die ebenfalls eine sehr gute Unterstützung bietet.

Die JavaScript-Konsole

Starten wir mit einem kurzen „Hallo-Welt"-Programm, das wir in der JavaScript-Konsole des Browsers laufen lassen. Dazu lernen wir das globale Object *console* (nicht zu verwechseln mit der JavaScript-Konsole) kennen, das die Funktion *log()* bietet. Mit dieser Funktion könnt ihr – ähnlich wie *System.out.println()* in Java – eine Ausgabe auf der Konsole erzeugen:

```
console.log("Hallo, Welt");
```

Im Chrome kommt ihr an die Konsole über die Tastenkombinaten *CTRL + Shift + J* (auf dem Mac ⌥⌘*J*) oder über das so genannte „Hotdog-Menü" rechts oben heran [8]. Einen Screenshot dazu könnt ihr in Abbildung 2.1 sehen.

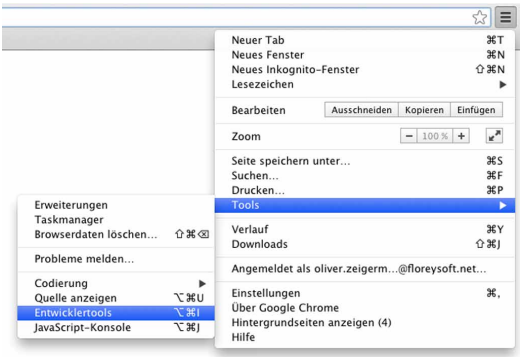

Abbildung 2.1: So kommt ihr im Chrome an die Entwicklertools bzw. JavaScript-Konsole

Hier könnt ihr nun direkt den JavaScript-Schnipsel von oben aufrufen und seht als Ausgabe den erwarteten „Hallo, Welt"-String. Zusätzlich erscheint eine Zeile mit der Ausgabe *undefined*, da die Chrome-JavaScript-Konsole auch immer die Rückgabe einer Funktion ausgibt. Die ist in diesem Fall eben *undefined*. Salopp kann man sich diese Rückgabe so vorstellen wie das, was bei einer *void*-Methode in Java zurückgeliefert wird, nämlich nichts.

JavaScript in HTML-Seiten

Für JavaScript-Code, der mehr als ein kleines Experiment ist, empfiehlt es sich, ihn aus einer HTML-Seite heraus aufzurufen. Das geht, indem man den Code direkt in die HTML-Seite einbettet wie in dem folgenden Beispiel.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Hallo, Welt!</title>
    <script>
        alert("Hallo, Welt");
    </script>
</head>
<body>
</body>
</html>
```

Wenn ihr einen Rechner zur Hand habt, tippt dieses kleine Beispiel einmal mit einem Texteditor ab, speichert es unter *index.html* und ruft es mit dem Browser auf. Es sollte sich eine Nachrichtenbox mit dem Text „Hallo, Welt" zeigen. Im Chrome sieht das Ganze in etwa wie in Abbildung 2.2 aus.

Abbildung 2.2: Die Ausgabe unseres zweiten Hallo-Welt-Programms

Der Nachteil des kleinen Skripts: es vermischt HTML und JavaScript. Das wollen wir bei dem nächsten Beispiel anders machen und beides trennen. Zudem wollen wir einer weiteren Best Practice folgen und den JavaScript-Code erst am Ende einer HTML-Seite einfügen. Dadurch müssten wir bei der Darstellung der Seite keine Verzögerung während des Ladens und Parsens des JavaScript hinnehmen:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Hallo, Welt!</title>
</head>
<body>
    <script src="hallo.js"></script>
</body>
</html>
```

Dazu nun noch die passende JavaScript-Datei hallo.js:

```
alert("Hello World");
```

Die Ausgabe dieses Beispiels unterscheidet sich nicht von der des vorherigen. Wir haben hier also unser erstes JavaScript-Refactoring gesehen!

IDEs

Sobald wir den Bereich der Spielereien mit JavaScript verlassen und größere Projekte angehen, stellt sich die Frage nach einer Entwicklungsumgebung. Die meisten Java-Entwickler sind komfortable IDEs wie z. B. Eclipse, Netbeans oder IntelliJ IDEA gewohnt. Häufig werden solche Werkzeuge aber von JavaScript-Entwicklern belächelt. Diese bevorzugen gern Texteditoren wie *vi*, *Emacs*, *TextMate* oder *Sublime Text*.

Ihr könnt euch nun entscheiden, ob ihr euch auf diese neue Welt einlassen oder lieber bei einer komfortablen IDE bleiben wollt – was meine Empfehlung wäre. In diesem Fall eignet sich gerade *IDEA Ultimate* hervorragend als Alleskönner, sowohl für Java als auch für JavaScript. Wem das zu teuer oder zu schwergewichtig ist, dem empfehle ich die kleine Schwester WebStorm, ebenfalls von JetBrains [9]. Diese kann im JavaScript-Bereich nicht weniger, ist aber einfacher und kostet weniger. Einen Eindruck von WebStorm bei der Arbeit könnt ihr in Abbildung 2.3 bekommen.

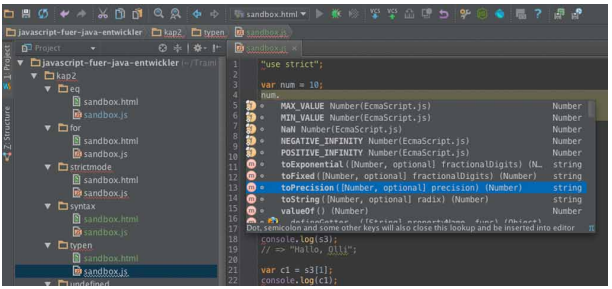

Abbildung 2.3: WebStorm im „coolen“ Darcula-Theme

2.2 Typen

Wir steigen nun nach und nach in die relevanten Details der Sprache JavaScript ein. Es geht los mit Typen. Entgegen einer weit verbreiteten Meinung existieren diese in JavaScript. Neben *object* gibt es auch die drei primitiven Typen *string*, *number* und *boolean*.

Objekte

Ein Objekt in JavaScript können wir uns vereinfacht wie eine Map in Java vorstellen. Es gibt Properties als Name/Wert-Paare, die jederzeit hinzugefügt und auch wieder gelöscht werden können. Werte können ebenso jederzeit in Inhalt und Typ verändert werden. Wir hätten damit so etwas wie *Map<String, Object>*.

Ein Objekt erzeugen wir mit der folgenden Syntax, die „object literal" oder Objekt-Literal genannt wird:

```
var obj1 = {};
```

In diesem Fall haben wir ein Objekt ohne Properties erzeugt und speichern die Referenz auf das Objekt in der Variablen *obj1*.

HINWEIS: Wir gehen später in diesem Kapitel genauer auf Variablen und das Schlüsselwort *var* ein. Für unsere Zwecke reicht es hier aus zu wissen, dass Variablen mit dem Schlüsselwort *var* deklariert werden und bei der Deklaration bereits mit einem Wert initialisiert werden können.

Ihr könnt schon beim Erzeugen eines Objekts beliebige Properties angeben:

```
var obj2 = {
         name1: "Wert1",
         name2: "Wert2",
         "Beliebiger String": "Wert3"
};
```

Hier erzeugen wir ein neues Objekt mit den Properties *name1*, *name2* und *Beliebiger String*.

Der Name einer Property kann ohne Anführungszeichen stehen, wenn er wie ein Identifier aufgebaut ist. Die kompletten Regeln für den Aufbau eines Identifiers stehen unter [1]. In Kurzform darf der Identifier keine Leerzeichen und keine Zeichen enthalten, die auch sonst als Syntax dienen. Damit fallen z. B. der Doppelpunkt, das Komma, das Semikolon und Klammern weg.

Der String *Beliebiger String* muss in Anführungszeichen stehen, da er ein Leerzeichen enthält.

Der Zugriff auf Properties von Objekten kann genau wie in Java mit dem Punkt-Operator erfolgen oder – anders als in Java - mit dem []-Operator:

```
console.log(obj2.name1);
// => "Wert1"
console.log(obj2["Beliebiger String"]);
// => "Wert3"
```

Bestehende Properties können jederzeit überschrieben und neue jederzeit eingeführt werden. Auch Referenzen auf Funktionen sind möglich:

```
obj2.name1 = "Neuer Wert";
console.log(obj2.name1);
// => "Neuer Wert"
obj2.func = function() { return "Called"; };
```

HINWEIS: Wir werden später noch genauer auf Funktionen eingehen, wichtig ist hier, dass Funktionen in JavaScript auch Objekte sind und wir somit Referenzen darauf in Properties halten können.

Properties können vom jedem beliebigen JavaScript-Typ sein und können ihn auch jederzeit ändern. Auch Referenzen auf andere Objekte sind möglich. Properties können mit *delete* wieder gelöscht werden:

```
var obj3 = {
         name1: "Wert1",
         name2: "Wert2",
         bool: true,
         zahl1: 10,
         ref1: obj2, // Referenz auf obj2
         ref2: { name: "Neues Objekt" }
};
obj3.zahl1 = "zahl1 ist nun ein String!";
delete obj3.bool;
console.log(obj3.bool);
// => undefined
```

Arrays

Arrays sind in JavaScript Objekte mit erweiterten Eigenschaften. Es gibt eine spezielle Syntax, um ein Array zu erzeugen. Dazu nimmt man den *[]*-Operator:

```
var array = ["a", "b", "c"];
console.log(array);
// => ["a", "b", "c"]
```

Mit *typeof* kann man den Typ eines Ausdrucks bestimmen. Wie oben erwähnt, sind Arrays Objekte:

```
console.log(typeof array);
// => "object";
```

Einzelne Elemente eines Arrays bekommt man über den *[]*-Operator …

```
var el = array[2];
console.log(el);
// => "c"
```

… und kann auch neue Werte direkt darüber setzen:

```
array[1] = 20;
console.log(array);
// => ["a", 20, "c"]
```

Das Pendant zur *add*-Methode in Java ist in JavaScript die *push*-Methode:

```
// fügt die 4 am Ende hinzu
array.push(4);
console.log(array);
// => ["a", 20, "c", 4]
```

Das Entfernen und das Einfügen von Elementen macht dieselbe Methode, nämlich *splice*: Zuerst das Entfernen von Elementen …

```
// Ab Position 1 werden 2 Elemente entfernt
// und zurückgegeben
array.splice(1, 2);
console.log(array);
// => ["a", 4]
```

... dann das Einfügen:

```
// An Position 1 werden 0 Elemente
// (also keine) entfernt und zurückgegeben
// Zudem wird an Position 1 "x" hinzugefügt
array.splice(1, 0, "x");
console.log(array);
// => ["a", "x", 4]
```

Eine komplette Referenz der Operationen auf Arrays ist unter [13] zu finden.

string

String in JavaScript entspricht in etwa der aus Java bekannten *String*-Klasse. Man kann Strings über Zeichenketten angeben, die entweder in einfachen oder doppelten Anführungszeichen eingeschlossen sind. Anders als bei Java sind dabei einfache oder doppelte Anführungszeichen semantisch äquivalent.

```
var string1 = "Zeichenkette!";
var string2 = 'Geht auch!';
```

PROFITIPP: Entscheidet euch für eine Variante der Anführungszeichen und haltet euch konsequent daran, das vermeidet Missverständnisse. JSON-Objekte (Kapitel 6) erfordern zwingend doppelte Anführungszeichen, während sich bei vielen JavaScript-Programmierern eine Tendenz zu einfachen Anführungszeichen herausbildet.

typeof kennt ihr ja bereits. Damit können wir wieder den Typ bestimmen:

```
console.log(typeof string1);
// => string
```

Einzelne Zeichen, die auch wieder vom Typ String sind, können auf zwei Arten extrahiert werden: zum einen über die Funktion *charAt* und zum anderen über den *[]*-Operator:

```
var s3 = "Hallo";
var c1 = s3[1];
console.log(c1);
// => "a"
console.log(typeof c1);
// => "string"
console.log(s3.charAt(1) === c1);
// => true
```

Strings können wie in Java mit dem +-Operator aneinandergehängt werden:

```
var s1 = "Hallo, ";
var s2 = "Olli";

var s3 = s1 + s2;
console.log(s3);
// => "Hallo, Olli";
```

Wenn man viele Strings aneinanderhängen möchte, gibt es dafür eine effizientere Art. Hier wird das Erzeugen von vielen Zwischenobjekten vermieden:

```
var builder = ["a", "b", "c"];
var s4 = builder.join("");
console.log(s4);
// => "abc";
```

Dies ist im *Effekt* – jedoch nicht in der Implementierung – vergleichbar mit *StringBuilder* bzw. *StringBuffer* in Java.

HINWEIS: Wir benutzen hier die Methode *join*, die auf Arrays definiert ist. Der angegebene String – hier der Leer-String – dient als Trennzeichen.

Die komplette Referenz für Strings findet ihr unter [12].

number

Im Gegensatz zu Java kennt JavaScript nur einen einzigen Zahlentyp, nämlich *number*, der im Wesentlichen *double* in Java entspricht.

```
var int = 1;
console.log(typeof int);
// => "number";

var float = 1.0;
console.log(typeof float);
// => "number";
```

Obwohl beide Variablen vom Typ *number* sind, könnt ihr euch bei der Wandlung von einem String zwischen *int* oder *float* entscheiden:

```
var int2 = parseInt("1000.1");
console.log(int2);
// => 1000

var float2 = parseFloat("1000.1");
console.log(float2);
// => 1000.1
```

parseInt und *parseFloat* sind global definierte Funktionen und damit direkt aufrufbar. Mehr zu Sichtbarkeitsbereichen und Funktionen in den folgenden Kapiteln.

Anders herum, also von *number* nach *string*, geht es auch:

```
console.log(float2.toFixed());
// => "1000"
```

Dabei kann man optional die Anzahl der Stellen nach dem Komma festlegen. Tut man das nicht, ist der Default-Wert *0*.

```
console.log(float2.toFixed(2));
// => "1000.10"
```

Man kann mit *number* so rechnen wie man es erwartet. Dabei kann es zu besonderen Ergebnissen kommen:

```
console.log(1 / 0);
// => Infinity
console.log(typeof (1 / 0));
// => "number";

console.log(0 / 0);
// => NaN
console.log(typeof (0 / 0));
// => "number";

console.log(0 / "a");
// => NaN
```

NAN steht für „Not A Number“, also für „keine Zahl“! Dennoch sind sowohl *NaN* als auch *Infinite* weiterhin vom Typ *number* und ihr könntet mit ihnen sogar weiter rechnen.

Als Java-Programmierer würde man hier *Exceptions* erwarten, diese gab es jedoch nicht in der ersten Version von JavaScript. Dazu später mehr in diesem Kapitel.

Auch JavaScript hat eine kleine mathematische Bibliothek, die über das globale *Math*-Objekt [15] erreichbar ist und die grundlegenden Funktionen enthält.

boolean

Ebenso wie in Java gibt es auch in JavaScript einen booleschen Datentyp. Über ihn gibt es nicht viel zu sagen, außer: Es gibt die Literale *true* und *false* und der Typ heißt "*boolean*":

```
var bool = true;
console.log(typeof bool);
// => "boolean";
```

2.3 Variablen

Variablen deklariert man in JavaScript mit dem Schlüsselwort *var*, dabei kann man die Variable optional mit einem Wert initialisieren, zum Beispiel so:

```
var a = 10;
```

Es ist auch möglich, mehrere Variablen auf einmal zu deklarieren, dazu trennt man sie mit einem Komma. Variablen, die nicht initialisiert wurden, enthalten den Wert *undefined*. Versucht man, mit *undefinierten* Variablen oder Feldern zu arbeiten, führt das zu Fehlermeldungen. Mehr dazu im Unterkapitel *undefined vs null*.

Man kann jeder Variablen jederzeit einen neuen Wert zuweisen. Unveränderliche Variablen gibt es nicht:

```
var a = 10, b;
console.log(a);
// => 10
console.log(b);
// => undefined
b = 100;
console.log(b);
// => 100
```

Variablen können auch Referenzen auf Objekte oder Arrays enthalten.

```
var obj = {a: 10};
console.log(obj);
=> Object {a: 10}

var array = [10];
console.log(array);
// => [10]
```

Ebenso wie in Java wird der Speicher in JavaScript über eine *Garbage Collection* verwaltet. Zum Beispiel wird das Objekt *obj* so lange im Speicher gehalten, wie mindestens eine Variable darauf eine Referenz enthält oder eine Property eines anderen Objekts darauf zeigt. Hier also nichts Neues für uns Java-Entwickler.

PROFITIPP: Grundsätzlich ist es möglich, das *var*-Schlüsselwort wegzulassen. Das solltet ihr jedoch nie tun, da das Programm dann nur noch scheinbar das Erwartete tut. Tatsächlich wird eine globale Variable definiert. Es gibt jedoch Mittel und Wege, das zu verhindern. Mehr dazu am Ende des nächsten Kapitels.

Die Werte von Variablen haben zwar einen Typ, doch der kann sich zur Laufzeit jederzeit ändern:

```
var c;

c = 10;
console.log(typeof c);
// => number

c = true;
console.log(typeof c);
// => boolean

c = {wert: 10}
console.log(typeof c);
// => object
```

PROFITIPP: Entscheidet euch bei jeder Variable schon bei der Deklaration für einen Typ und ändert ihn danach nicht mehr. Dokumentiert, um welchen Typ es sich handelt, das vermeidet Überraschungen.

Wie man eine Variable und deren Typ am besten dokumentieren kann, seht ihr im nächsten Abschnitt.

2.4 Syntax

Die Syntax von JavaScript sieht auf den ersten Blick ähnlich wie die von Java aus. Auf den zweiten Blick fallen lediglich einige zusätzliche Schlüsselworte wie z. B. *var* und *function* auf. Man muss jedoch vorsichtig sein, da manche Konstrukte zwar vertraut aussehen, sich jedoch anders verhalten als erwartet.

Semikolon

Das Semikolon ist in JavaScript, anders als in Java, optional. Das wirft die Frage auf, wie in JavaScript denn ein Statement beendet wird. Denn wie in Java kann auch in JavaScript ein Statement