Cay Horstmann

JavaScript für Ungeduldige

Der schnelle Einstieg in modernes JavaScript

dpunkt.verlag

Cay Horstmann ist Hauptautor von *Core Java*™, Band I und II, 11. Auflage (Pearson, 2018), *Scala for the Impatient*, 2. Auflage (Addison-Wesley, 2016) und *Core Java SE 9 for the Impatient* (Addison-Wesley, 2017). Er ist emeritierter Professor für Informatik an der San José State University (Kalifornien, USA), Java-Champion und häufiger Redner auf Konferenzen der Computerbranche.

Papier plus+ PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus+:

www.dpunkt.plus

Cay Horstmann

JavaScript für Ungeduldige

Der schnelle Einstieg in modernes JavaScript

dpunkt.verlag

Cay Horstmann

Lektorat: Melanie Andrisek
Übersetzung: Volkmar Gronau
Copy-Editing: Alexander Reischert, *www.aluan.de*
Satz: G&U Language & Publishing Services GmbH, Flensburg, *www.GundU.com*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

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

ISBN:
Print 978-3-86490-801-9
PDF 978-3-96910-093-6
ePub 978-3-96910-094-3
mobi 978-3-96910-095-0

1. Auflage 2021
Copyright © 2021 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Authorized translation from the English language edition, entitled MODERN JAVASCRIPT FOR THE IMPATIENT, 1st Edition by CAY HORSTMANN, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2020 Pearson Education, Inc
All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: hallo@dpunkt.de.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag noch Übersetzer können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Inhalt

Vorwort .... **xv**

1 Werte und Variable **1**

1.1 JavaScript ausführen 2
1.2 Typen und der Operator typeof 5
1.3 Kommentare 5
1.4 Variablendeklarationen 6
1.5 Bezeichner 8
1.6 Zahlen 9
1.7 Arithmetische Operatoren 10
1.8 Boolesche Werte 13
1.9 null und undefined 13
1.10 String-Literale 14
1.11 Template-Literale 17
1.12 Objekte 18
1.13 Objektliteral-Syntax 19
1.14 Arrays 20
1.15 JSON 22
1.16 Destrukturierung 24
1.17 Destrukturierung für Fortgeschrittene 26
1.17.1 Mehr zum Thema Objektstrukturierung 26
1.17.2 Restdeklarationen 27
1.17.3 Standardwerte 27
1.18 Übungen 28

2 Steuerstrukturen **31**

2.1 Ausdrücke und Anweisungen 31
2.2 Semikolonergänzung 33

2.3 Verzweigungen 36
2.4 Falsy- und Truthy-Werte 39
2.5 Vergleichs- und Gleichheitsoperatoren 39
2.6 Vergleiche unterschiedlicher Typen 41
2.7 Boolesche Operatoren 43
2.8 Die switch-Anweisung 45
2.9 while- und do-Schleifen 46
2.10 for-Schleifen 47
2.10.1 Die klassische for-Schleife 47
2.10.2 Die for-of-Schleife 48
2.10.3 Die for-in-Schleife 49
2.11 break und continue 51
2.12 Ausnahmen abfangen 53
2.13 Übungen 54

3 Funktionen und funktionale Programmierung 57

3.1 Funktionen deklarieren 58
3.2 Funktionen höherer Ordnung 59
3.3 Funktionsliterale 60
3.4 Pfeilfunktionen 61
3.5 Funktionale Array-Verarbeitung 63
3.6 Closures 64
3.7 Harte Objekte 66
3.8 Strikter Modus 68
3.9 Argumenttypen prüfen 70
3.10 Mehr oder weniger Argumente bereitstellen 71
3.11 Standardargumente 72
3.12 Restparameter und der Verteilungsoperator 73
3.13 Benannte Argumente durch Destrukturierung simulieren 75
3.14 Hoisting 76
3.15 Exceptions auslösen 79
3.16 Exceptions abfangen 79
3.17 Die finally-Klausel 81
3.18 Übungen 82

4 Objektorientierte Programmierung ... 85

4.1 Methoden ... 86
4.2 Prototypen ... 87
4.3 Konstruktoren ... 90
4.4 Die Klassensyntax ... 92
4.5 Get- und Set-Methoden ... 93
4.6 Instanzfelder und private Methoden ... 94
4.7 Statische Methoden und Felder ... 95
4.8 Teilklassen ... 96
4.9 Methoden überschreiben ... 98
4.10 Konstruktion von Teilklassen ... 99
4.11 Klassenausdrücke ... 100
4.12 Der Verweis this ... 101
4.13 Übungen ... 105

5 Zahlen und Datumsangaben ... 109

5.1 Zahlenliterale ... 109
5.2 Zahlenformatierung ... 110
5.3 Parsen von Zahlen ... 111
5.4 Funktionen und Konstanten der Klasse Number ... 112
5.5 Funktionen und Konstanten der Klasse Math ... 114
5.6 Große Integer ... 115
5.7 Datumsangaben konstruieren ... 115
5.8 Funktionen und Methoden der Klasse Date ... 119
5.9 Datumsformatierung ... 120
5.10 Übungen ... 121

6 Strings und reguläre Ausdrücke ... 125

6.1 Konvertierung zwischen Strings und Codepunktfolgen ... 125
6.2 Teil-Strings ... 126
6.3 Weitere String-Methoden ... 128
6.4 Tagged-Template-Literale ... 132
6.5 Rohe Template-Literale ... 133
6.6 Reguläre Ausdrücke ... 135
6.7 Literale für reguläre Ausdrücke ... 138

6.8 Flags 139
6.9 Reguläre Ausdrücke und Unicode 140
6.10 Die Methoden der Klasse RegExp 141
6.11 Gruppen 143
6.12 String-Methoden für reguläre Ausdrücke 146
6.13 Mehr über das Ersetzen mit regulären Ausdrücken 147
6.14 Exotische Merkmale 149
6.15 Übungen 150

7 Arrays und Sammlungen 153

7.1 Arrays konstruieren 153
7.2 Die Eigenschaft length und die Indexeigenschaften 155
7.3 Elemente löschen und hinzufügen 156
7.4 Weitere Methoden zur Veränderung von Arrays 158
7.5 Elemente erstellen 161
7.6 Elemente finden 162
7.7 Alle Elemente durchlaufen 163
7.8 Dünn besetzte Arrays 165
7.9 Reduzierung 167
7.10 Maps 170
7.11 Mengen 173
7.12 Schwache Maps und Mengen 174
7.13 Typisierte Arrays 175
7.14 Array-Puffer 178
7.15 Übungen 179

8 Internationalisierung 183

8.1 Gebietsschemata 183
8.2 Ein Gebietsschema angeben 185
8.3 Zahlenformatierung 188
8.4 Datum und Uhrzeit 190
8.4.1 Date-Objekte formatieren 190
8.4.2 Datumsbereiche 192
8.4.3 Relative Zeitangaben 192
8.4.4 Zerlegung in Teilangaben 192

8.5 Sortierung 193
8.6 Weitere gebietsschemaabhängige String-Methoden 195
8.7 Pluralregeln und Listen 196
8.8 Verschiedene gebietsschemaabhängige Merkmale 198
8.9 Übungen 200

9 Asynchrone Programmierung 203

9.1 Parallele Aufgaben in JavaScript 204
9.2 Promises erstellen 207
9.3 Unmittelbar erledigte Promises 210
9.4 Ergebnisse von Promises abrufen 210
9.5 Promises verketten 211
9.6 Umgang mit abgelehnten Promises 213
9.7 Mehrere Promises ausführen 215
9.8 Wettlauf mehrerer Promises 216
9.9 async-Funktionen 217
9.10 Rückgabewerte von async-Funktionen 219
9.11 Gleichzeitiges Warten 221
9.12 Ausnahmen in async-Funktionen 222
9.13 Übungen 223

10 Module 229

10.1 Das Prinzip von Modulen 230
10.2 ECMAScript-Module 230
10.3 Standardimporte 231
10.4 Benannte Importe 232
10.5 Dynamische Importe 233
10.6 Exporte 234
10.6.1 Benannte Exporte 234
10.6.2 Der Standardexport 235
10.6.3 Exporte sind Variable 236
10.6.4 Reexport 237
10.7 Module verpacken 238
10.8 Übungen 239

11 Metaprogrammierung ... 243

11.1 Symbole ... 244
11.2 Anpassung mithilfe von Symboleigenschaften ... 245
11.2.1 Die Methode toString anpassen ... 246
11.2.2 Die Typumwandlung steuern ... 247
11.2.3 species ... 248
11.3 Attribute von Eigenschaften ... 248
11.4 Eigenschaften auflisten ... 251
11.5 Das Vorhandensein einer einzelnen Eigenschaft prüfen ... 253
11.6 Objekte schützen ... 253
11.7 Objekte erstellen und ändern ... 254
11.8 Auf den Prototyp zugreifen und ihn ändern ... 255
11.9 Objekte klonen ... 256
11.10 Funktionseigenschaften ... 259
11.11 Argumente binden und Methoden aufrufen ... 260
11.12 Proxys ... 261
11.13 Die Klasse Reflect ... 264
11.14 Proxy-Invarianten ... 267
11.15 Übungen ... 269

12 Iteratoren und Generatoren ... 275

12.1 Iterierbare Werte ... 276
12.2 Iterierbare Objekte implementieren ... 277
12.3 Abschließbare Iteratoren ... 279
12.4 Generatoren ... 280
12.5 Verschachtelte yield-Anweisungen ... 282
12.6 Generatoren als Verbraucher ... 285
12.7 Generatoren in der asynchronen Verarbeitung ... 286
12.8 async-Generatoren und -Iteratoren ... 288
12.9 Übungen ... 291

13 Einführung in TypeScript ... 297

13.1 Typanmerkungen ... 298
13.2 TypeScript ausführen ... 300
13.3 Typterminologie ... 301
13.4 Primitive Typen ... 303

13.5 Zusammengesetzte Typen 304
13.6 Typinferenz 306
13.7 Untertypen 310
13.7.1 Die Substitutionsregel 310
13.7.2 Optionale und überzählige Eigenschaften 312
13.7.3 Untertypbeziehungen von Array- und Objekttypen 313
13.8 Klassen 314
13.8.1 Klassen deklarieren 314
13.8.2 Der Instanztyp einer Klasse 316
13.8.3 Der statische Typ einer Klasse 317
13.9 Strukturelle Typisierung 318
13.10 Schnittstellen 319
13.11 Indizierte Eigenschaften 321
13.12 Komplexe Funktionsparameter 322
13.12.1 Optionale, Standard- und Restparameter 322
13.12.2 Parameter destrukturieren 323
13.12.3 Untertypbeziehungen von Funktionstypen 325
13.12.4 Überladung 327
13.13 Generische Programmierung 329
13.13.1 Generische Klassen und Typen 330
13.13.2 Generische Funktionen 331
13.13.3 Typeinschränkungen 332
13.13.4 Löschung 333
13.13.5 Untertypbeziehungen von generischen Typen 334
13.13.6 Bedingte Typen 335
13.13.7 Zugeordnete Typen 336
13.14 Übungen 338

Stichwortverzeichnis 343

Für Chi, die geduldigste Person in meinem Leben

Vorwort

Erfahrene Programmierer, die mit Sprachen wie Java, C# oder C++ vertraut sind, finden sich oft in Situationen wieder, in denen sie mit JavaScript arbeiten müssen. Das liegt daran, dass es immer mehr webgestützte Benutzerschnittstellen gibt und JavaScript nun einmal die *Lingua franca* der Browser ist. Das Electron-Framework hat die Anwendung dieser Sprache auf Rich-Client-Anwendungen ausgedehnt, und es gibt verschiedene Möglichkeiten, um JavaScript-Apps für Mobilgeräte zu erstellen. Auch serverseitig wird JavaScript immer häufiger eingesetzt.

Vor vielen Jahren galt JavaScript als eine Sprache zur Programmierung im Kleinen. Ihre Features konnten für umfangreiche Programme ziemlich verwirrend und fehleranfällig sein. Mit den heutigen Standardisierungsbemühungen und dem Angebot an Werkzeugen hat die Sprache sich jedoch weit über diese bescheidenen Anfänge hinaus entwickelt.

Leider ist es schwierig, modernes JavaScript zu lernen, ohne mit veraltetem JavaScript überschüttet zu werden. In den meisten Büchern, Kursen und Blogposts geht es um den Übergang von älteren JavaScript-Versionen, was für Personen nicht hilfreich ist, die von anderen Sprachen kommen.

Das ist die Lücke, die ich mit diesem Buch füllen möchte. Ich gehe davon aus, dass Sie bereits ein kompetenter Programmierer sind und sich mit Verzweigungen und Schleifen, Funktionen, Datenstrukturen und den Grundlagen der objektorientierten Programmierung auskennen. Ich erkläre Ihnen, wie Sie in modernem

JavaScript produktiv arbeiten können, und erwähne veraltete Merkmale nur am Rande. Sie lernen hier, wie Sie modernes JavaScript nutzen, ohne über die Fallstricke der Vergangenheit zu stolpern.

JavaScript ist nicht perfekt, aber es hat sich für die Programmierung von Benutzerschnittstellen und für viele serverseitige Aufgaben als gut geeignet erwiesen. Wie Jeff Atwood es einmal vorausschauend formulierte: »Jede Anwendung, die in JavaScript geschrieben werden *kann*, *wird* irgendwann auch in JavaScript geschrieben.«

Arbeiten Sie dieses Buch durch, um zu lernen, wie Sie die nächste Version Ihrer Anwendung in modernem JavaScript schreiben können.

Fünf goldene Regeln

Wenn Sie auf einige wenige klassische Features von JavaScript verzichten, können Sie das Maß an geistiger Anstrengung erheblich verringern, um die Sprache zu erlernen und anzuwenden. Die folgenden Regeln werden Ihnen jetzt zwar noch nicht viel sagen, aber ich führe Sie hier trotzdem zum späteren Nachschlagen auf – auch um Ihnen zu zeigen, wie beruhigend wenige es sind:

1. Deklarieren Sie Variablen mit `let` oder `const`, nicht mit `var`.
2. Verwenden Sie den strikten Modus.
3. Seien Sie sich immer über die Typen im Klaren und vermeiden Sie die automatische Typkonvertierung.
4. Machen Sie sich damit vertraut, wie Prototypen funktionieren, aber verwenden Sie für Klassen, Konstruktoren und Methoden die moderne Syntax.
5. Verwenden Sie `this` nicht außerhalb von Konstruktoren und Methoden.

Darüber hinaus gibt es noch eine Metaregel: *Schauen Sie sich keinen Wat-Code an*, also diese verwirrenden Ausschnitte aus JavaScript-Code, die mit einem sarkastischen (und orthografisch nicht korrekten) »Wat?!« kommentiert sind. Manche Leute haben Spaß daran zu zeigen, wie furchtbar JavaScript angeblich ist, indem sie obskuren Code analysieren. Aus solchen Übungen habe ich jedoch nie etwas Sinnvolles mitgenommen. Welchen Vorteil bietet es Ihnen etwa zu wissen, dass `2 * ['21']` den Wert `42` ergibt, `2 + ['40']` aber nicht, wenn Ihnen die goldene Regel Nr. 3 klipp und klar sagt, dass Sie nicht mit Typkonvertierungen herumdoktern sollten? Wenn ich in eine solche verwirrende Situation gerate, frage ich mich normalerweise, wie ich sie vermeiden kann, anstatt sie in allen unnützen Einzelheiten zu erklären.

Lernstoff von unterschiedlichem Niveau

Den Lernstoff in diesem Buch habe ich so angeordnet, dass Sie die benötigten Informationen leicht wiederfinden können, wenn Sie sie brauchen. Das ist allerdings nicht unbedingt die richtige Anordnung, wenn Sie das Buch zum ersten Mal lesen. Um Ihnen das Lernen zu erleichtern, habe ich jedes Kapitel mit einem Symbol für das Niveau des behandelten Stoffs versehen. Einzelne Abschnitte, die auf einem höheren Niveau angesiedelt sind, bekommen dabei ihre eigenen Symbole. Lassen Sie diese Abschnitte bei der ersten Lektüre aus und lesen Sie sie erst dann, wenn Sie dazu bereit sind.

Zur Kennzeichnung habe ich die folgenden Symbole verwendet:

Der ungeduldige Hase steht für ein **grundlegendes Thema**, das selbst die ungeduldigsten Leser nicht überspringen sollten.

Alice kennzeichnet ein Thema von **mittlerem Niveau**, mit dem sich die meisten Programmierer vertraut machen sollten, allerdings nicht unbedingt beim ersten Lesen.

Die Grinsekatze weist auf ein **fortgeschrittenes Thema** hin, das Framework-Entwicklern ein Lächeln entlocken mag. Die meisten Anwendungsentwickler können diese Abschnitte jedoch getrost ignorieren.

Der verrückte Hutmacher schließlich kennzeichnet **komplizierte Themen**, die einen in den Wahnsinn treiben können und sich nur für Leser mit krankhafter Neugier eignen.

Der Aufbau dieses Buches

In **Kapitel 1** legen wir mit den Grundlagen von JavaScript los: mit Werten und ihren Typen, mit Variablen und vor allem mit Objektliteralen. **Kapitel 2** behandelt den Steuerungsfluss. Wenn Sie mit Java, C# oder C++ vertraut sind, können Sie dieses Kapitel wahrscheinlich einfach überfliegen. In **Kapitel 3** lernen Sie Funktionen und die funktionale Programmierung kennen, die in JavaScript eine große Rolle spielt. JavaScript hat zwar ein Objektmodell, allerdings unterscheidet es sich stark von dem klassengestützter Programmiersprachen. **Kapitel 4** beschreibt dieses Objektmodell ausführlich, wobei der Schwerpunkt auf der modernen Syntax liegt. Die **Kapitel 5 und 6** behandeln die Bibliotheksklassen, die Sie am häufigsten zur Arbeit mit Zahlen, Datumsangaben, Strings und regulären Ausdrücken verwenden werden. Diese ersten sechs Kapitel sind auf grundlegendem Niveau, wobei einige etwas anspruchsvollere Abschnitte eingestreut wurden.

Die folgenden vier Kapitel behandeln Themen von mittlerem Niveau. In **Kapitel 7** erfahren Sie, wie Sie mit Arrays und anderen Sammlungstypen aus der JavaScript-Standardbibliothek arbeiten. Wenn Ihr Programm von Benutzern in aller Welt verwendet wird, sollten Sie der Internationalisierung besondere Aufmerksamkeit schenken, um die es in **Kapitel 8** geht. **Kapitel 9** über asynchrone Programmierung ist für alle Programmierer äußerst wichtig. Die asynchrone Programmierung war in JavaScript ziemlich kompliziert, ist mit der Einführung von Promises und der Schlüsselwörter `async` und `await` aber viel einfacher geworden. Außerdem hat JavaScript jetzt ein standardmäßiges Modulsystem, das Thema von **Kapitel 10** ist. Darin erfahren Sie, wie Sie Module von anderen Programmierern nutzen und ihre eigenen schreiben können.

In **Kapitel 11** geht es um Metaprogrammierung auf fortgeschrittenem Niveau. Sie sollten es lesen, wenn Sie Werkzeuge erstellen, um beliebige JavaScript-Objekte zu analysieren und umzuwandeln. **Kapitel 12** schließt die Erörterung von JavaScript mit einem weiteren fortgeschrittenen Thema ab, nämlich Iteratoren und Generatoren: zwei äußerst nützliche Mechanismen, um beliebige Folgen von Werten zu durchlaufen und zu produzieren.

Schließlich gibt es noch ein Bonuskapitel, nämlich **Kapitel 13** über TypeScript. Dabei handelt es sich um eine Erweiterung von JavaScript, die eine Typüberprüfung zur Kompilierzeit bietet. Es gehört nicht zum JavaScript-Standard, ist aber sehr populär. Lesen Sie dieses Kapitel, um selbst zu entscheiden, ob Sie beim einfachen JavaScript bleiben oder die Typisierung zur Kompilierzeit nutzen wollen.

Dieses Buch soll Ihnen solide Grundkenntnisse der *Sprache* JavaScript verleihen, sodass Sie sie souverän nutzen können. Informationen über die Werkzeuge und Frameworks, die einem ständigen Wandel unterliegen, müssen Sie dagegen an einem anderen Ort suchen.

Warum ich dieses Buch geschrieben habe

JavaScript ist eine der am häufigsten verwendeten Programmiersprachen der Welt. Wie viele Programmierer kannte ich zunächst ein bisschen Pidgin-JavaScript. Eines Tages aber musste ich ziemlich überstürzt echtes JavaScript lernen. Doch wie?

Es gab zwar eine Menge Bücher, mit denen Programmierer, die sich nur gelegentlich mit Webentwicklung befassen, ein bisschen JavaScript lernen konnten, aber so viel verstand ich von der Sprache ohnehin. Das *Nashornbuch* von Flanagan[1] war 1996 zwar großartig, setzt die Leser von heute aber zu vielen Missgriffen der Vergangenheit aus. *Das Beste an JavaScript* von Douglas Crockford[2] hat die Java-

1. David Flanagan, *JavaScript: Das umfassende Referenzwerk*, 6. Auflage (O’Reilly Verlag, 1997)
2. O’Reilly Verlag, 2008

Script-Welt 2008 wachgerüttelt, aber ein Großteil seiner Botschaft ist bereits in nachfolgende Änderungen der Sprache eingeflossen. Außerdem gibt es viele Bücher, die JavaScript-Programmierern der alten Schule die Welt der modernen Standards nahebringen, aber sie beschäftigen sich mit zu vielen klassischen JavaScript-Elementen, die mir nicht behagen.

Das Web ist voll von Blogs zum Thema JavaScript, allerdings von sehr unterschiedlicher Qualität. Einige sind korrekt, aber viele zeigen nur ein ziemlich schwaches Verständnis. Für mich war es nicht sehr sinnvoll, das Web nach Blogs zu durchsuchen und jeweils zu prüfen, wie wahrheitsgetreu sie sind.

Merkwürdigerweise konnte ich kein Buch für die Millionen Programmierer finden, die Java oder eine ähnliche Sprache kennen und JavaScript in seiner heutigen Form ohne den historischen Ballast lernen wollen.

Also musste ich es selbst schreiben.

Danksagung

Ein weiteres Mal möchte ich meinem Herausgeber Greg Doench für die Unterstützung dieses Projekts danken sowie Dmitry Kirsanov und Alina Kirsanova für das Korrekturlesen und den Satz des Buches. Mein besonderer Dank geht an meine Lektoren Gail Anderson, Tom Austin, Scott Davis, Scott Good, Kito Mann, Bob Nicholson, Ron Mak und Henri Tremblay, die sorgfältig Fehler aufgespürt und wohlüberlegte Vorschläge für Verbesserungen gemacht haben.

Cay Horstmann
Berlin
März 2020

1
Werte und Variable

In diesem Kapitel lernen Sie die Datentypen kennen, mit denen Sie in JavaScript-Anwendungen arbeiten können: Zahlen, Strings und andere primitive Typen sowie Objekte und Arrays. Sie erfahren hier, wie Sie solche Werte in Variablen speichern, wie Sie Werte von einem Typ in einen anderen umwandeln und wie Sie sie mithilfe von Operatoren kombinieren.

Selbst begeisterte JavaScript-Programmierer geben zu, dass einige Konstrukte von JavaScript – die eigentlich dabei helfen sollen, Programme möglichst kurz und knapp zu schreiben – zu widersinnigen Ergebnissen führen können und daher am besten vermieden werden sollten. In diesem und den folgenden Kapiteln werde ich solche Probleme aufzeigen und einige einfache Regeln für sicheres Programmieren vorstellen.

1.1 JavaScript ausführen

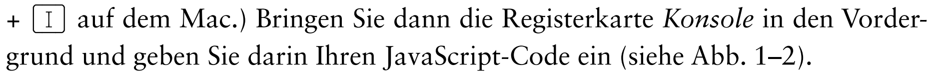
Es gibt verschiedene Möglichkeiten, um während der Lektüre dieses Buches JavaScript-Programme auszuführen. Da JavaScript ursprünglich zur Ausführung in einem Browser gedacht war, können Sie JavaScript-Code in eine HTML-Datei einbetten. Um Werte anzuzeigen, rufen Sie darin die Methode `window.alert` auf. Eine solche Datei sieht wie folgt aus:

```
<html>
   <head>
      <title>My First JavaScript Program</title>
      <script type="text/javascript">
         let a = 6
         let b = 7
         window.alert(a * b)
      </script>
   </head>
   <body>
   </body>
</html>
```

Wenn Sie die Datei in einem Browser öffnen, wird das Ergebnis wie in Abbildung 1–1 in einem Dialogfeld angezeigt:

Abb. 1–1 *Ausführung von JavaScript-Code in einem Browser*

Sie können auch kurze Folgen von Anweisungen in die Konsole eingeben, die zu den Entwicklerwerkzeugen des Browsers gehört. Suchen Sie den Menübefehl oder die Tastenkombination zur Anzeige dieser Werkzeuge. (In vielen Browsern ist das die Taste `F12` oder die Kombination `Strg` + `Alt` + `I` bzw. `cmd` + `alt`



+ [I] auf dem Mac.) Bringen Sie dann die Registerkarte *Konsole* in den Vordergrund und geben Sie darin Ihren JavaScript-Code ein (siehe Abb. 1–2).

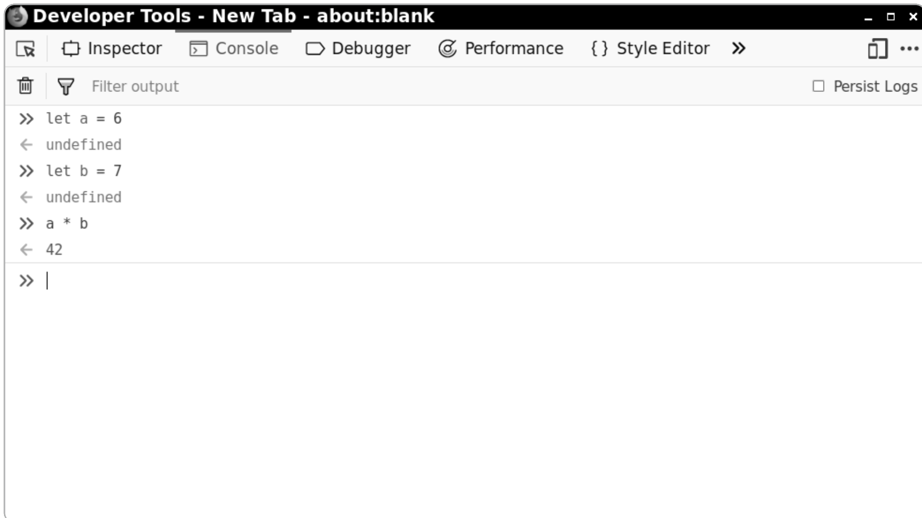


Abb. 1–2 *Ausführung von JavaScript-Code in der Entwicklerkonsole*

Eine dritte Möglichkeit besteht darin, Node.js von `http://node.js.org` zu installieren, ein Terminalfenster zu öffnen und darin das Programm `node` auszuführen. Dadurch wird eine JavaScript-»REPL« gestartet (Read-Eval-Print Loop, also etwa »Lese-, Ausführungs- und Ausgabeschleife«). Darin können Sie Befehle eingeben und sich die Ergebnisse anzeigen lassen (siehe Abb. 1–3).

Abb. 1–3 *Ausführung von JavaScript-Code in der Node.js-REPL*

Wenn Sie längere Codefolgen ausführen wollen, schreiben Sie die Anweisungen in eine Datei. Für Ausgaben verwenden Sie dabei die Methode `console.log`. Beispielsweise können Sie die folgenden Anweisungen in eine Datei aufnehmen, die Sie `first.js` nennen:

```
let a = 6
let b = 7
console.log(a * b)
```

Führen Sie anschließend den folgenden Befehl aus:

```
node first.js
```

Im Terminal wird nun die Ausgabe des Befehls `console.log` angezeigt.

Sie können auch eine Entwicklungsumgebung wie Visual Studio Code, Eclipse, Komodo IDE oder WebStorm verwenden. Darin können Sie, wie in Abbildung 1–4 gezeigt, JavaScript-Code bearbeiten und ausführen:

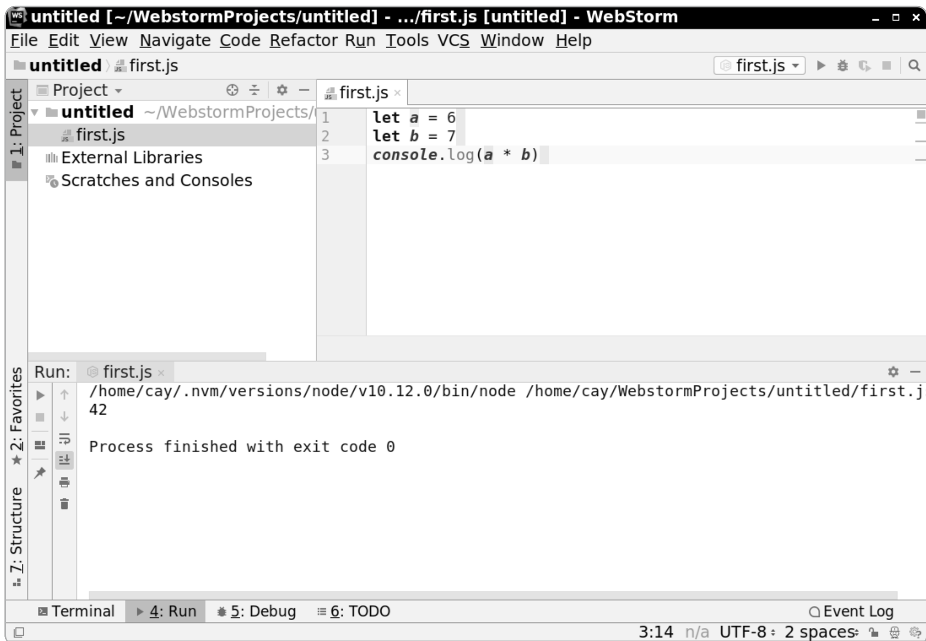


Abb. 1–4 *Ausführung von JavaScript-Code in einer Entwicklungsumgebung*

1.2 Typen und der Operator typeof

Werte in JavaScript sind jeweils von einem der folgenden Typen:

- eine Zahl
- einer der booleschen Werte `false` oder `true`
- einer der besonderen Werte `null` oder `undefined`
- ein String
- ein Symbol
- ein Objekt

Die Typen, die keine Objekte sind, werden zusammengenommen als *primitive Typen* bezeichnet.

Mehr über diese Typen erfahren Sie in den folgenden Abschnitten. Die einzige Ausnahme bilden die Symbole, die erst in Kapitel 11 behandelt werden.

Den Typ eines gegebenen Werts können Sie mit dem Operator `typeof` herausfinden, der einen der Strings `'number'`, `'boolean'`, `'undefined'`, `'object'`, `'string'` oder `'symbol'` oder einen von wenigen weiteren möglichen Strings zurückgibt. Beispielsweise ergibt `typeof 42` den String `'number'`.

Hinweis

Obwohl der Typ `null` nicht identisch mit dem Typ `object` ist, ergibt `typeof null` den String `'object'`. Das ist leider historisch so gewachsen.

Vorsicht

Ähnlich wie in Java können Sie in JavaScript Objekte als Wrapper für Zahlen, boolesche Werte und Strings konstruieren. Beispielsweise werden sowohl `typeof new Number(42)` als auch `typeof new String('Hello')` zu `'object'` ausgewertet. Allerdings gibt es in JavaScript keinen sinnvollen Grund dafür, solche Wrapper-Instanzen zu erstellen. Da sie eher für Verwirrung sorgen, ist ihre Verwendung in vielen Programmierstandards untersagt.

1.3 Kommentare

In JavaScript können Sie zwei verschiedene Arten von Kommentaren einfügen. Einzeilige Kommentare beginnen mit // und laufen bis zum Ende der Zeile:

```
// Einzeiliger Kommentar
```

Dagegen können mit /* und */ abgegrenzte Kommentare mehrere Zeilen umspannen:

```
/*
   mehrzeiliger
   Kommentar
*/
```

In diesem Buch verwende ich eine Serifenschrift, um die Kommentare in Listings deutlicher hervorzuheben. In Ihrem Texteditor werden die Kommentare wahrscheinlich farbig gekennzeichnet sein.

Hinweis

Anders als Java bietet JavaScript keine besondere Formatierung für Kommentare zur Dokumentation. Für diesen Zweck können Sie jedoch Drittanbieterwerkzeuge wie JSDoc (`http://usejsdoc.org`) nutzen.

1.4 Variablendeklarationen

Mit der Anweisung `let` können Sie einen Wert in einer Variablen speichern:

```
let counter = 0
```

In JavaScript haben Variable keinen festen Typ. Daher können Sie in jeder Variablen Werte beliebigen Typs speichern. Beispielsweise ist es zulässig, den Inhalt von `counter` durch einen String zu ersetzen:

```
counter = 'zero'
```

Während es wohl kaum jemals sinnvoll sein dürfte, so etwas zu tun, erleichtern untypisierte Variablen es Ihnen in manchen Situationen, generischen Code zu schreiben, der mit unterschiedlichen Typen funktioniert.

Wenn Sie eine Variable nicht initialisieren, hat sie den besonderen Wert `undefined`:

```
let x // Deklariert die Variable x und setzt sie auf undefined
```

Hinweis

Vielleicht ist Ihnen aufgefallen, dass die vorstehenden Anweisungen nicht mit einem Semikolon abschließen. Ebenso wie in Python sind Semikolons am Zeilenende in JavaScript nicht erforderlich. Während in Python unnötige Semikolons als »unpythonisch« gelten, sind JavaScript-Programmierer darüber geteilter Ansicht. In Kapitel 2 werde ich die Argumente beider Seiten besprechen. Im Allgemeinen versuche ich mich aus solchen fruchtlosen Diskussionen herauszuhalten, aber in diesem Buch musste ich mich für eine Vorgehensweise entscheiden. Den semikolonfreien Stil habe ich aus einem ganz einfachen Grund gewählt: Code dieser Art lässt sich nicht mit Java oder C++ verwechseln. So lässt sich auf den ersten Blick erkennen, dass es sich um JavaScript handelt.

Wenn Sie den Wert einer Variablen niemals ändern, sollten Sie sie mit der Anweisung `const` deklarieren:

```
const PI = 3.141592653589793
```

Sollten Sie versuchen, den Wert in einer solchen Konstante zu ändern, tritt ein Laufzeitfehler auf.

In einer einzigen `let`- oder `const`-Anweisung können Sie auch mehrere Variable deklarieren:

```
const FREEZING = 0, BOILING = 100
let x, y
```

Viele Programmierer ziehen es jedoch vor, jede Variable in einer eigenen Anweisung zu deklarieren.

Vorsicht

Es gibt zwei veraltete Formen der Variablendeklaration, die Sie vermeiden sollten, nämlich die Deklaration mit dem Schlüsselwort `var` und die Deklaration ohne jegliches Schlüsselwort:

```
var counter = 0 // Veraltet
coutner = 1     // Aufgrund des Tippfehlers wird eine neue Variable
                // erstellt!
```

Die Deklaration mit `var` weist einige erhebliche Mängel auf; mehr darüber erfahren Sie in Kapitel 3. Variable bei ihrer ersten Zuweisung zu erstellen, ist offensichtlich gefährlich, denn wenn Sie den Variablennamen falsch schreiben, wird eine andere Variable angelegt. Aus diesem Grund gilt diese Vorgehensweise im *strikten Modus*, in dem solche veralteten Konstrukte verboten sind, als Fehler. Wie Sie den strikten Modus einschalten, erfahren Sie in Kapitel 3.

Tipp

Wenn Sie sich nach den im Vorwort aufgeführten fünf goldenen Regeln richten, können Sie die Verwirrung, die solche klassischen JavaScript-Merkmale verursachen, größtenteils vermeiden. Die ersten beiden dieser Regeln lauten:

1. Deklarieren Sie Variablen mit `let` oder `const`, nicht mit `var`.
2. Verwenden Sie den strikten Modus.

1.5 Bezeichner

Die Namen von Variablen müssen der allgemeinen Syntax für *Bezeichner* folgen. Bezeichner dürfen aus Unicode-Buchstaben, Ziffern sowie den Zeichen _ und $ bestehen. Ziffern dürfen nicht am Anfang stehen. In manchen Tools und Bibliotheken werden Namen mit $-Zeichen verwendet und manche Programmierer setzen einen Unterstrich an den Anfang oder das Ende von Bezeichnern, um private Merkmale zu kennzeichnen. Daher ist es am besten, wenn Sie bei den Namen, die Sie selbst festlegen, auf das Zeichen $ sowie auf _ am Anfang und Ende verzichten. Interne Unterstriche sind kein Problem, aber viele JavaScript-Programmierer bevorzugen die Camel-Case-Schreibweise mit Binnenmajuskel, um einzelne Namensbestandteile abzugrenzen.

Die folgenden Schlüsselwörter dürfen nicht als Bezeichner verwendet werden:

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

Im strikten Modus sind auch die folgenden Schlüsselwörter unzulässig:

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

Die folgenden Schlüsselwörter wurden erst kürzlich hinzugefügt. Sie können sie noch aus Gründen der Rückwärtskompatibilität als Bezeichner nutzen, sollten es aber lieber nicht tun:

```
await as async from get of set target yield
```

Hinweis

In Bezeichnern können Sie beliebige Unicode-Buchstaben und Ziffern verwenden, also beispielsweise auch Folgendes:

```
const ϖ = 3.141592653589793
```

So etwas ist jedoch unüblich, da vielen Programmierern die Möglichkeiten zur schnellen Eingabe solcher Zeichen fehlen.

1.6 Zahlen

JavaScript hat keinen expliziten Typ für Integer. Alle Zahlen sind Fließkommazahlen mit doppelter Genauigkeit. Natürlich können Sie ganzzahlige Werte verwenden; kümmern Sie sich einfach nicht um die Unterschiede zwischen `1` und `1.0`. Aber wie sieht es mit Rundungen aus? Alle ganzen Zahlen zwischen `Number.MIN_SAFE_INTEGER` ($-2^{53} + 1$ gleich –9.007.199.254.740.991) und `Number.MAX_SAFE_INTEGER` ($+2^{53} - 1$ gleich 9.007.199.254.740.991) werden exakt dargestellt. Das ist ein größeres Intervall als das für Integer in Java. Solange die Ergebnisse innerhalb dieses Intervalls bleiben, sind arithmetische Operationen auf Integern exakt. Außerhalb dieses Bereichs dagegen kann es zu Rundungsfehlern kommen. Beispielsweise wird `Number.MAX_SAFE_INTEGER * 10` zu `90071992547409900`.

Hinweis

Wenn der Integer-Bereich nicht ausreicht, können Sie auch große Integer mit einer beliebigen Anzahl von Stellen verwenden. Mehr darüber erfahren Sie in Kapitel 5.

Beim Rechnen mit Fließkommazahlen kann es wie in allen Programmiersprachen zu Rundungsfehlern kommen. Beispielsweise wird `0.1 + 0.2` wie in Java, C++ und Python zu `0.30000000000000004` ausgewertet. So etwas ist unvermeidlich, da es keine exakte binäre Darstellung für Dezimalbrüche wie `0.1`, `0.2` und `0.3` gibt. Wenn Sie mit Euro- und Centbeträgen rechnen müssen, sollten Sie daher alle Werte als ganzzahlige Vielfache eines Cent angeben. In Kapitel 5 werden Sie noch weitere Formen von numerischen Literalen kennenlernen, z. B. Hexadezimalzahlen.

Um einen String in eine Zahl umzuwandeln, können Sie die Funktionen `parseFloat` und `parseInt` verwenden:

```
const notQuitePi = parseFloat('3.14') // Die Zahl 3.14
const evenLessPi = parseInt('3')      // Der Integer 3
```

Mit der Methode `toString` dagegen konvertieren Sie eine Zahl in einen String:

```
const notQuitePiString = notQuitePi.toString() // Der String '3.14'
const evenLessPiString = (3).toString()        // Der String '3'
```

Hinweis

Anders als in Java, aber ebenso wie in C++ gibt es in JavaScript sowohl Funktionen als auch Methoden. Bei den Funktionen `parseFloat` und `parseInt` handelt es sich nicht um Methoden. Deshalb werden sie nicht mit der Punktschreibweise aufgerufen.

Hinweis

Wie der vorige Code zeigt, ist es möglich, Methoden auf numerische Literale anzuwenden. Dabei müssen Sie die Literale jedoch in Klammern einschließen, damit der Punkt nicht fälschlicherweise als Dezimaltrennzeichen aufgefasst wird.

Vorsicht

Was geschieht, wenn Sie in einem Fall, in dem ein Integer erwartet wird, einen Dezimalbruch verwenden? Das hängt von der jeweiligen Situation ab. Nehmen wir an, Sie wollen aus einem String einen Teil-String entnehmen. Dabei werden Dezimalbrüche als Positionsangaben abgeschnitten, sodass sich der nächstkleinere Integer ergibt:

```
'Hello'.substring(0, 2.5) // Der String 'He'
```

Geben Sie aber einen Dezimalbruch als Index an, lautet das Ergebnis `undefined`:

```
'Hello'[2.5] // undefined
```

Es lohnt nicht, sich damit zu beschäftigen, wann ein Dezimalbruch anstelle eines Integers verwendet werden kann und wann nicht. Machen Sie in solchen Situationen deutlich, was Sie beabsichtigen, indem Sie ausdrücklich `Math.trunc(x)` oder `Math.round.(x)` aufrufen, um die Nachkommastellen abzuschneiden bzw. die Zahl auf den nächsten Integer zu runden.

Bei einer Division durch null lautet das Ergebnis `Infinity` oder `-Infinity`. Allerdings wird `0 / 0` zu `NaN` ausgewertet, der Konstante »not a number«. Manche Funktionen, die Zahlen generieren, geben `NaN` zurück, um auf eine fehlerhafte Eingabe hinzuweisen. Beispielsweise wird `parseFloat('pie')` zu `NaN` ausgewertet.

1.7 Arithmetische Operatoren

JavaScript verfügt über die üblichen Operatoren `+`, `-`, `*` und `/` für Addition, Subtraktion, Multiplikation und Division. Beachten Sie, dass der Operator `/` stets eine

Fließkommazahl ergibt, selbst wenn beide Operanden Integer sind. Beispielsweise ergibt `1 / 2` die Zahl `0.5` und nicht `0`, wie es in Java oder C++ der Fall wäre.

Ebenso wie in Java, C++ und Python ergibt der Operator `%` den Rest der Division zweier nichtnegativer Integer-Operanden. Wenn `k` ein nichtnegativer Integer ist, wird `k % 2` für ein gerades `k` zu `0` ausgewertet und für ein ungerades `k` zu `1`.

Sind `k` und `n` positiv (und möglicherweise nicht ganzzahlig), dann ist `k % n` der Wert, der sich ergibt, wenn fortgesetzt `n` von `k` subtrahiert wird, bis das Ergebnis kleiner als `n` ist. Beispielsweise ergibt `3.5 % 1.2` den Wert `1.1`, das Ergebnis der zweimaligen Subtraktion von `1.2`. Was bei negativen Operanden geschieht, erfahren Sie in Übung 3.

Der Operator ** steht ebenso wie in Python (und schon in Fortran) für eine Potenzierung. Der Wert von `2 ** 10` ist `1024`, der von `2 ** -1` ist `0.5` und der von `2 ** 0.5` die Quadratwurzel von 2.

Ist einer der Operanden eines arithmetischen Operators der »Not-a-Number-Wert« `NaN`, so ist das Ergebnis ebenfalls `NaN`.

Wie in Java, C++ und Python können Sie Zuweisungen und arithmetische Operationen kombinieren:

```
counter += 10 // Identisch mit counter = counter + 10
```

Die Operatoren ++ und -- inkrementieren bzw. dekrementieren eine Variable:

```
counter++ // Identisch mit counter = counter + 1
```

Vorsicht

Ebenso wie Java und C++ folgt auch JavaScript dem Beispiel von C und erlaubt es, den Operator ++ vor oder nach der Variable anzugeben, was den Prä- bzw. Post-Inkrementwert ergibt:

```
let counter = 0
let riddle = counter++
let enigma = ++counter
```

Welchen Wert haben `riddle` und `enigma`? Wenn Sie es nicht wissen, können Sie es herausfinden, indem Sie die vorige Beschreibungen genau lesen, den Code ausprobieren oder sich an den großen Quell des Wissens wenden, das Internet. Allerdings rate ich Ihnen dringend, niemals Code zu schreiben, für den Sie solche Kenntnisse benötigen.

Manche Programmierer halten die Operatoren ++ und -- für so verwerflich, dass sie sich weigern, diese zu benutzen. Es gibt auch keinen echten Grund dafür, denn schließlich ist `counter += 1` nicht viel länger als `counter++`. In diesem Buch werde ich die Operatoren ++ und -- zwar verwenden, aber niemals in Zusammenhängen, in denen ihr Wert erfasst wird.

Wie in Java wird der Operator + auch zur String-Verkettung verwendet. Wenn s ein String ist und x ein Wert eines beliebigen Typs, dann sind sowohl s + x als auch x + s Strings, die dadurch zustande kommen, dass x in einen String umgewandelt und mit s verkettet wird.

Betrachten Sie dazu das folgende Beispiel:

```
let counter = 7
let agent = '00' + counter // Der String '007'
```

Vorsicht

Wie Sie gesehen haben, ist der Ausdruck x + y eine Zahl, wenn beide Operanden Zahlen sind, und ein String, wenn es sich bei mindestens einem Operanden um einen String handelt. In allen anderen Fällen sind die Regeln ziemlich kompliziert und die Ergebnisse nur selten sinnvoll. Entweder werden beide Operanden in Strings verwandelt und verkettet oder in Zahlen konvertiert und addiert. Beispielsweise wird der Ausdruck `null + undefined` zu der numerischen Addition `0 + NaN` ausgewertet, die wiederum `NaN` ergibt (siehe Tabelle 1–1). Bei den anderen arithmetischen Operatoren wird nur eine Umwandlung in Zahlen versucht. So ergibt beispielsweise `6 * '7'` den Wert `42`, da der String `'7'` in die Zahl 7 konvertiert wird.

Wert	Umwandlung in Zahl	Umwandlung in String
Eine Zahl	Die Zahl selbst	Ein String aus den Ziffern dieser Zahl
Ein String aus Ziffern, die eine Zahl bilden	Der Wert der Zahl	Der String selbst
Der leere String `''`	`0`	`''`
Jeder andere String	`NaN`	Der String selbst
`false`	`0`	`'false'`
`true`	`1`	`'true'`
`null`	`0`	`'null'`
`undefined`	`NaN`	`'undefined'`
Das leere Array []	`0`	`''`
Ein Array, das eine einzige Zahl enthält	Die Zahl	Ein String aus den Ziffern der Zahl
Andere Arrays	`NaN`	Die Elemente, umgewandelt in Strings und durch Kommata getrennt, z. B. `'1,2,3'`.
Objekte	Standardmäßig `NaN`, kann aber angepasst werden	Standardmäßig `'[object Object]'`, kann aber angepasst werden.

Tab. 1–1 *Umwandlung in Zahlen und Strings*

Tipp

Verlassen Sie sich nicht auf die automatische Typumwandlung bei arithmetischen Operatoren. Die Regeln sind kompliziert und können zu unerwarteten Ergebnissen führen. Wenn Sie Strings oder einelementige Arrays als Operanden verarbeiten wollen, dann wandeln Sie sie explizit um.

Tipp

Verwenden Sie lieber Template-Literale (siehe Abschnitt 1.11, »Template-Literale«) als die String-Verkettung. Dadurch müssen Sie sich nicht merken, was der Operator + bei nichtnumerischen Operanden macht.

1.8 Boolesche Werte

Der boolesche Typ kann die beiden Werte `false` und `true` annehmen. In einer Bedingung werden Werte beliebiger Typen in einen booleschen Wert umgewandelt. Dabei werden `0`, `NaN`, `null`, `undefined` und der leere String zu `false` konvertiert und alle anderen zu `true`.

Das klingt zwar ganz einfach, aber wie Sie im folgenden Kapitel noch sehen werden, kann das zu verwirrenden Resultaten führen. Um Unklarheiten auf ein Minimum zu reduzieren, ist es sinnvoll, in Bedingungen grundsätzlich echte boolesche Werte zu verwenden.

1.9 null und undefined

JavaScript kennzeichnet das Fehlen eines Wertes auf zwei verschiedene Arten. Wenn eine Variable deklariert, aber nicht initialisiert wird, ist ihr Wert `undefined`. Das geschieht häufig bei Funktionen: Wenn Sie eine Funktion aufrufen, aber keinen Parameter bereitstellen, hat die Parametervariable den Wert `undefined`. Der Wert `null` dagegen dient dazu, die beabsichtigte Abwesenheit eines Wertes zu kennzeichnen.

Ist diese Unterscheidung sinnvoll? Darüber gehen die Meinungen auseinander. Manche Programmierer meinen, dass die Verwendung zweier solcher Verlegenheitswerte fehleranfällig ist. Deshalb raten sie dazu, nur einen davon zu verwenden. Das sollte dann `undefined` sein, da es nicht möglich ist, diesen Wert in JavaScript zu vermeiden, wohingegen Sie auf `null` (fast) immer verzichten können.

Nach der gegenteiligen Ansicht sollten Sie weder Werte auf `undefined` setzen noch `undefined` von einer Funktion zurückgeben lassen, sondern für fehlende Werte

stets `null` verwenden. Dadurch bleibt `undefined` als ein Signal reserviert, das auf ernste Probleme hindeutet.

Tipp

Einigen Sie sich bei jedem Projekt auf die eine oder die andere Vorgehensweise, also entweder `undefined` oder `null` zur Anzeige der beabsichtigten Abwesenheit eines Wertes zu verwenden. Dadurch ersparen Sie sich später endlose philosophische Diskussionen und unnötige Prüfungen auf `undefined` und `null`.

Vorsicht

Im Gegensatz zu `null` ist `undefined` *kein* reserviertes Wort, sondern eine Variable im globalen Gültigkeitsbereich. Früher war es sogar möglich, der globalen Variablen `undefined` einen neuen Wert zuzuweisen! So etwas zu tun, ist natürlich eine furchtbare Idee, und heutzutage ist `undefined` eine Konstante. Allerdings können Sie immer noch *lokale* Variable mit dem Namen `undefined` deklarieren. Das ist allerdings nach wie vor eine schlechte Idee. Deklarieren Sie auch keine lokalen Variablen mit den Namen `NaN` und `Infinity`.

1.10 String-Literale

String-Literale sind in einfache oder doppelte Anführungszeichen eingeschlossen, also z. B. `'Hallo'` oder `"Hallo"`. In diesem Buch verwende ich dazu immer einfache Anführungszeichen.

Wenn innerhalb eines Strings ein Anführungszeichen der gleichen Art steht, mit der der String begrenzt ist, dann müssen Sie es mit einem Backslash maskieren. Auch Backslashs selbst und die Steuerzeichen aus Tabelle 1–2 müssen Sie mit Backslashs maskieren.

Beispielsweise ist `'\\\'\'\\\n'` ein String von fünf Zeichen Länge, der die Zeichenfolge `\''\` gefolgt von einem Zeilenumbruch enthält.

Um beliebige Unicode-Zeichen in einen JavaScript-String aufzunehmen, können Sie sie einfach eingeben oder kopieren, wobei die Quelldatei jedoch eine geeignete Kodierung verwenden muss (z. B. UTF-8):

```
let greeting = 'Hello 🌐'
```

Wenn Ihre Dateien unbedingt ASCII-Format haben müssen, können Sie stattdessen die Schreibweise `\u{`*Codepunkt*`}` verwenden:

```
let greeting = 'Hello \u{1F310}'
```

Maskierungssequenz	Maskiertes Zeichen	Unicode-Wert
`\b`	Rückschritt	`\u{0008}`
`\t`	Tabulator	`\u{0009}`
`\n`	Zeilenvorschub	`\u{000A}`
`\r`	Wagenrücklauf	`\u{000D}`
`\f`	Seitenvorschub	`\u{000C}`
`\v`	Vertikaler Tabulator	`\u{000B}`
`\'`	Einfaches Anführungszeichen	`\u{0027}`
`\"`	Doppeltes Anführungszeichen	`\u{0022}`
`\\`	Backslash	`\u{005C}`

\Zeilenumbruch	Fortsetzung in der nächsten Zeile	Nichts – es erfolgt kein Zeilenumbruch:
		`"Hel\`
		`lo"`
		ergibt den String `"Hello"`.

Tab. 1–2 *Maskierungssequenzen für Sonderzeichen*

Leider gibt es bei der Verwendung von Unicode in JavaScript einen bösen Haken. Um die Feinheiten zu verstehen, müssen wir einen Blick auf die Geschichte von Unicode werfen. Vor der Erfindung von Unicode gab es verschiedene, nicht miteinander vereinbare Systeme zur Zeichenkodierung, wobei ein und dieselbe Bytefolge für Benutzer in Europa, Russland oder China ganz andere Bedeutungen haben konnte.

Unicode sollte diese Probleme lösen. Als man in den 80er Jahren mit dieser Vereinheitlichung begann, schien es so, als reiche ein 16-Bit-Code völlig aus, um sämtliche Zeichen in allen Sprachen der Welt zu kodieren und dabei noch Platz für zukünftige Erweiterungen zu lassen. 1991 wurde Unicode 1.0 veröffentlicht, worin knapp die Hälfte der verfügbaren 65.536 Codewerte belegt war. Als JavaScript und Java 1995 erschienen, nutzten sie die Unicode-Kodierung. In beiden Sprachen sind Strings Folgen von 16-Bit-Werten.

Im Laufe der Zeit aber geschah das Unvermeidliche: Der Umfang von Unicode überstieg den Vorrat von 65.536 Zeichen. Heutzutage nutzt Unicode 21 Bits, was nach gängiger Meinung nun wirklich ausreichen sollte. JavaScript aber ist bei den 16-Bit-Werten stecken geblieben.

Um zu erklären, wie dieses Problem gelöst wird, müssen wir uns ein bisschen mit dem technischen Hintergrund beschäftigen. Ein Unicode-*Codepunkt* ist ein 21-Bit-Wert, der mit einem Zeichen verknüpft wird. JavaScript nutzt die UTF-16-Kodierung, die alle Unicode-Codepunkte durch einen oder zwei 16-Bit-Werte oder *Codeeinheiten* darstellt. Für Zeichen bis `\u{FFFF}` wird jeweils eine Codeeinheit

verwendet. Alle anderen Zeichen dagegen werden mit zwei Einheiten kodiert, die aus einem reservierten Bereich stammen und nicht zur Darstellung irgendwelcher anderer Zeichen genutzt werden. Beispielsweise wird `\u{1F310}` durch die Folge `0xD83C 0xDF10` kodiert. (Eine Beschreibung des Kodieralgorithmus finden Sie auf `https://de.wikipedia.org/wiki/UTF-16`.)

Mit den Einzelheiten der Kodierung müssen Sie sich nicht beschäftigen, aber wissen, dass einige Zeichen eine einzelne 16-Bit-Codeeinheit erfordern, andere dagegen zwei.

Beispielsweise hat der String `'Hello 🌐'` eine Länge von 8, obwohl er nur sieben Unicode-Zeichen enthält (einschließlich des Leerzeichens zwischen `Hello` und 🌐). Mit dem Operator `[]` können Sie auf die Codeeinheiten eines Strings zugreifen. Der Ausdruck `greeting[0]` ist ein String, der nur aus dem Zeichen `'H'` besteht. Allerdings funktioniert dieser Operator nicht bei Zeichen, die aus zwei Codeeinheiten aufgebaut sind. Die Codeeinheiten für das Zeichen 🌐 befinden sich an den Positionen 6 und 7. Die Ausdrücke `greeting[6]` und `greeting[7]` sind Strings der Länge 1 und enthalten jeweils eine einzige Codeeinheit, die aber kein Zeichen kodiert. Mit anderen Worten: Es handelt sich nicht um gültige Unicode-Strings.

Tipp

In Kapitel 2 werden Sie erfahren, wie Sie mit einer `for-of`-Schleife die einzelnen Codepunkte abrufen können.

Hinweis

Sie können in String-Literalen auch 16-Bit-Codeeinheiten angeben, müssen dabei aber die geschweiften Klammern weglassen: `\uD83C\uDF10`. Für Codeeinheiten bis `\u{0xFF}` können Sie die Hexmaskierung verwenden, also z. B. `\xA0` statt `\u{00A0]`. Allerdings kann ich mir für beides keinen guten Grund vorstellen.

In Kapitel 6 lernen Sie die Methoden für die Arbeit mit Strings kennen.

Hinweis

In JavaScript gibt es auch Literale für reguläre Ausdrücke. Mehr darüber erfahren Sie in Kapitel 6.

1.11 Template-Literale

Template-Literale sind Strings, die Ausdrücke enthalten und mehrere Zeilen umspannen können. Sie werden in Backticks (`...`) eingeschlossen:

```
let destination = 'world' // Regulärer String
let greeting = `Hello, ${destination.toUpperCase()}!` //Template-Literal
```

Die in ${...} eingebetteten Ausdrücke werden ausgewertet, bei Bedarf in einen String umgewandelt und dann in das Template eingefügt. Im vorigen Beispiel ergibt sich dadurch der folgende String:

```
Hello, WORLD!
```

Sie können auch weitere Template-Literale in dem ${...}-Ausdruck verschachteln:

```
greeting = `Hello, ${firstname.length > 0 ? `${firstname[0]}. ` : '' }
${lastname}`
```

Alle Zeilenumbrüche innerhalb des Template-Literals werden in den String aufgenommen. Betrachten Sie dazu das folgende Beispiel:

```
greeting = `<div>Hello</div>
<div>${destination}</div>
`
```

Hier wird greeting auf den String '<div>Hello</div>\n<div>World</div>\n' mit Zeilenumbrüchen hinter jeder Zeile gesetzt. (Für den resultierenden String werden die Windows-Zeilenende-Zeichen \r\n in das Unix-Zeilenende-Zeichen \n umgewandelt.)

Um in Template-Literale Backticks, Dollarzeichen und Backslashs aufzunehmen, müssen Sie sie mit Backslashs maskieren. Die Zeichenfolge `\`\$\\` enthält die drei Zeichen `$\.

Hinweis

Als *Tagged-Template-Literale* werden Template-Literale bezeichnet, denen eine Funktion vorausgeht:

```
html`<div>Hello, ${destination}</div>`
```

Hier wird die Funktion html mit den Template-Fragmenten '<div>Hello, ' und '</div>' sowie dem Wert des Ausdrucks destination aufgerufen.
In Kapitel 6 erfahren Sie, wie Sie eigene Tag-Funktionen schreiben.

1.12 Objekte

JavaScript-Objekte unterscheiden sich von denen in klassengestützten Sprachen wie Java und C++. Ein JavaScript-Objekt ist lediglich eine Menge von »Eigenschaften« genannten Name-Wert-Paaren wie dem folgenden:

```
{ name: 'Harry Smith', age: 42 }
```

Ein solches Objekt enthält nur öffentliche Daten und bietet keine Kapselung und kein Verhalten. Es ist auch keine Instanz einer bestimmten Klasse. Kurz gesagt, ist es etwas ganz anderes als ein Objekt im Sinne der objektorientierten Programmierung. Wie Sie in Kapitel 2 noch sehen werden, ist es zwar möglich, Klassen und Methoden zu definieren, doch unterscheiden sich die Mechanismen sehr stark von denen in den meisten anderen Sprachen.

Natürlich ist es möglich, ein Objekt in einer Variablen zu speichern:

```
const harry = { name: 'Harry Smith', age: 42 }
```

Bei einer solchen Variablen können Sie mit der üblichen Punktschreibweise auf die Eigenschaften des Objekts zugreifen:

```
let harrysAge = harry.age
```

Dadurch können Sie vorhandene Eigenschaften bearbeiten und neue hinzufügen:

```
harry.age = 40
harry.salary = 90000
```

Hinweis

Obwohl die Variable `harry` als `const` definiert wurde, können Sie das Objekt verändern, auf das sie verweist. Nicht möglich ist es dagegen, einer `const`-Variablen einen anderen Wert zuzuweisen:

```
const sally = { name: 'Sally Lee' }
sally.age = 28 // Zulässig: Ändert das Objekt, auf das sally verweist
sally = { name: 'Sally Albright' }
// Fehler: Einer const-Variablen kann kein neuer Wert zugewiesen
// werden
```

Mit anderen Worten: `const` verhält sich wie `final` in Java und nicht wie `const` in C++.

Um eine Eigenschaft zu entfernen, verwenden Sie den Operator `delete`:

```
delete harry.salary
```