Kapitel 2
Erste Schritte

Nach wie vor wird JavaScript hauptsächlich für die Erstellung dynamischer Webseiten, sprich innerhalb eines Browsers, eingesetzt. Bevor wir uns in späteren Kapiteln im Detail mit anderen Anwendungsgebieten befassen, werde ich Ihnen in diesem Kapitel zeigen, auf welche Weisen Sie JavaScript in eine Webseite einbinden und einfache Ausgaben erzeugen können. Dieses Kapitel bildet somit die Grundlage für die folgenden Kapitel.

Bevor wir uns ausführlicher mit der Sprache JavaScript an sich beschäftigen, sollten Sie zunächst wissen, in welchem Zusammenhang JavaScript mit *HTML* (*Hypertext Markup Language*) und *CSS* (*Cascading Style Sheets*) innerhalb einer Webseite steht, wie man JavaScript in eine Webseite einbindet und wie man Ausgaben erzeugen kann.

2.1 Einführung in JavaScript und die Webentwicklung

Die wichtigsten drei Sprachen für die Erstellung von Web-Frontends sind sicherlich HTML, CSS und JavaScript. Jede dieser Sprachen hat dabei ihre eigene Bestimmung.

2.1.1 Der Zusammenhang zwischen HTML, CSS und JavaScript

Mithilfe von HTML legen Sie über *HTML-Elemente* die *Struktur* einer Webseite und die *Bedeutung* (die *Semantik*) einzelner Komponenten auf einer Webseite fest. Sie beschreiben beispielsweise, welcher Bereich auf der Webseite den Hauptinhalt darstellt, welcher die Navigation, und definieren Komponenten wie Formulare, Listen, Schaltflächen, Eingabefelder oder, wie in Abbildung 2.1 zu sehen, Tabellen.

Artist	Album	Release Date	Genre
Monster Magnet	Powertrip	1998	Spacerock
Kyuss	Welcome to Sky Valley	1994	Stonerrock
Ben Harper	The Will to Live	1997	Singer/Songwriter
Tool	Lateralus	2001	Progrock
Beastie Boys	Ill Communication	1994	Hip Hop

Abbildung 2.1 HTML verwenden Sie, um die Struktur einer Webseite zu definieren.

Über CSS dagegen gestalten Sie mithilfe von speziellen *CSS-Regeln*, wie die einzelnen Komponenten, die Sie zuvor in HTML definiert haben, dargestellt werden sollen, sprich, Sie legen das *Design* und das *Layout* einer Webseite fest. Sie definieren hierbei beispielsweise Textfarbe, Textgröße, Umrandungen, Hintergrundfarben, Farbverläufe etc. In Abbildung 2.2 ist zu sehen, wie CSS dazu genutzt wurde, Schriftart und Schriftgröße der Tabellenüberschriften sowie der Tabellenzellen anzupassen, Rahmen zwischen Tabellenspalten und Tabellenzeilen hinzuzufügen und die Hintergrundfarbe der Tabellenzeilen im Wechsel mit einer jeweils anderen Hintergrundfarbe einzufärben. Das Ganze sieht dann schon um einiges ansprechender aus als die Variante ohne CSS.

Artist	Album	Release Date	Genre
Monster Magnet	Powertrip	1998	Spacerock
Kyuss	Welcome to Sky Valley	1994	Stonerrock
Ben Harper	The Will to Live	1997	Singer/Songwriter
Tool	Lateralus	2001	Progrock
Beastie Boys	Ill Communication	1994	Hip Hop

Abbildung 2.2 Mit CSS definieren Sie das Layout und das Aussehen einzelner Elemente der Webseite.

JavaScript zu guter Letzt dient dazu, der Webseite (bzw. den Komponenten auf einer Webseite) *dynamisches Verhalten* hinzuzufügen bzw. die Interaktivität auf der Webseite zu erhöhen. Beispiele hierfür sind die bereits in Kapitel 1, »Grundlagen und Einführung«, angesprochene Sortierung und Filterung von Tabellendaten (siehe Abbildung 2.3 und Abbildung 2.4). Während CSS also für das Design einer Webseite zuständig ist, kann mithilfe von JavaScript die Nutzerfreundlichkeit und die Interaktivität einer Webseite erhöht werden.

Search artist

Artist ▾	Album	Release Date	Genre
Beastie Boys	Ill Communication	1994	Hip Hop
Ben Harper	The Will to Live	1997	Singer/Songwriter
Kyuss	Welcome to Sky Valley	1994	Stonerrock
Monster Magnet	Powertrip	1998	Spacerock
Tool	Lateralus	2001	Progrock

Abbildung 2.3 JavaScript ermöglicht Ihnen, eine Webseite nutzerfreundlicher und interaktiver zu gestalten, z. B. um wie hier die Daten in einer Tabelle sortierbar ...

Be

Artist ▾	Album	Release Date	Genre
Beastie Boys	Ill Communication	1994	Hip Hop
Ben Harper	The Will to Live	1997	Singer/Songwriter

Abbildung 2.4 ... oder, wie hier gezeigt, die Daten filterbar zu machen.

Eine Webseite besteht also (in den allermeisten Fällen) aus einer Kombination von HTML-, CSS- und JavaScript-Code (siehe Abbildung 2.5) – wobei gilt: Auch wenn ich eben gesagt habe, dass JavaScript für das Verhalten einer Webseite zuständig ist, kann man funktionsfähige Webseiten auch gänzlich ohne JavaScript erstellen. Ja, prinzipiell kann man Webseiten auch ohne CSS erstellen. Prinzipiell schon. Dann wird eben nur das HTML vom Browser ausgewertet. In so einem Fall ist die Webseite aber weniger schick (ohne CSS) und weniger interaktiv und nutzerfreundlich (ohne JavaScript), siehe erneut Abbildung 2.1.

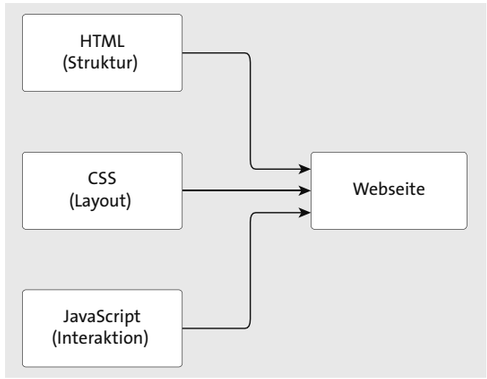

Abbildung 2.5 In der Regel wird innerhalb einer Webseite eine Kombination aus HTML, CSS und JavaScript verwendet.

Merke

HTML dient der Struktur einer Webseite, CSS dem Layout und dem Design, JavaScript dem Verhalten und der Interaktivität.

Definition

Web- und Softwareentwickler sprechen in diesem Zusammenhang auch gerne von drei Schichten: HTML bildet die *Inhaltsschicht*, CSS die *Darstellungsschicht* und JavaScript die *Verhaltensschicht*.

Trennen des Codes für die einzelnen Schichten

Guter Entwicklungsstil sieht vor, die einzelnen Schichten nicht zu vermischen, sprich HTML-, CSS- und JavaScript-Code unabhängig voneinander und in separaten Dateien vorzuhalten. Dies erleichtert den Überblick über ein Webprojekt und sorgt letztendlich dafür, dass Sie effektiver entwickeln können. Darüber hinaus können Sie auf diese Weise ein und dieselben CSS- und JavaScript-Dateien auch in verschiedenen HTML-Dateien einbinden (siehe Abbildung 2.6) und damit dieselben CSS-Regeln bzw. denselben JavaScript-Quelltext in verschiedenen HTML-Dateien wiederverwenden.

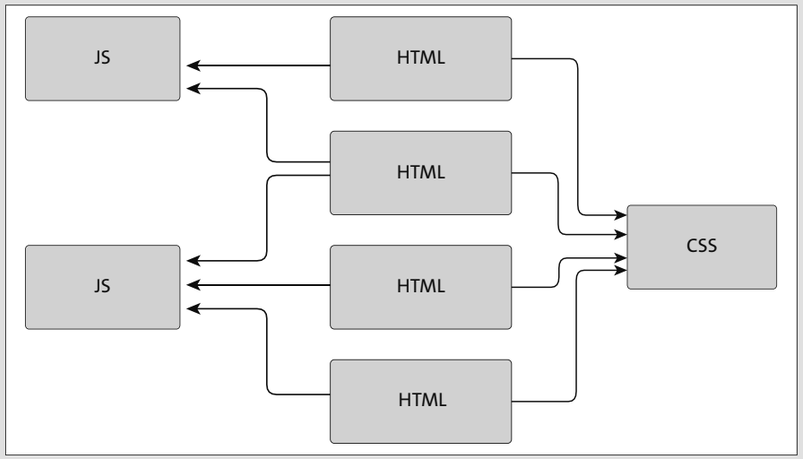

Abbildung 2.6 Wenn Sie CSS- und JavaScript-Code nicht direkt in den HTML-Code schreiben, sondern in separate Dateien, erleichtert das die Wiederverwendbarkeit.

Eine gute Vorgehensweise bei der Entwicklung einer Webseite ist es, sich erst über deren Struktur Gedanken zu machen: Welche Bereiche gibt es auf der Webseite? Welche Überschriften gibt es? Gibt es Daten, die in tabellarischer Form dargestellt werden? Aus welchen Einträgen besteht die Navigation? Welche Informationen sind im Fußbereich der Seite enthalten, welche im Kopfbereich? Hierbei verwendet man ausschließlich HTML. Die Webseite sieht dann zwar noch nicht schön aus und ist nur wenig interaktiv, aber darum soll es in diesem ersten Schritt bewusst nicht gehen, um nicht vom Wesentlichen, dem Inhalt der Webseite, abzulenken.

Aufbauend auf dieser strukturellen Grundlage, setzt man anschließend das Design mit CSS und das Verhalten der Webseite mit JavaScript um. Dabei können diese beiden Schritte prinzipiell parallel auch von verschiedenen Personen vorgenommen werden. Beispielsweise kann sich ein Webdesigner um das Design mit CSS kümmern, während ein Webentwickler die Funktionalität in JavaScript programmiert (in der Praxis sind zwar Webdesigner und Webentwickler häufig ein und dieselbe Person, aber insbesondere in großen Projekten mit vielen, vielen Webseiten ist eine Verteilung der Zuständigkeiten nicht selten).

Phasen der Website-Entwicklung

Bei der Entwicklung professioneller Websites gehen der reinen Entwicklung natürlich mehrere Phasen voraus. Bevor überhaupt mit der Entwicklung begonnen wird, werden in Konzept- und Designphasen Prototypen (entweder digital oder ganz klassisch mit Stift und Papier) entworfen. Das eben beschriebene schrittweise Vorgehen (erst HTML, dann CSS, dann JavaScript) bezieht sich somit nur auf die Entwicklung.

Auszeichnungssprache HTML und Stilsprache CSS

HTML und CSS sind übrigens keine Programmiersprachen! HTML ist eine *Auszeichnungssprache* und CSS eine *Stilsprache*, nur JavaScript ist von den drei genannten eine *Programmiersprache*. Daher sind auch Aussagen wie »Das lässt sich doch mit HTML programmieren« genau genommen nicht korrekt. Vielmehr müsste man sagen: »Das lässt sich doch mit HTML umsetzen.«

Definition

Der Prozess des Darstellens einer Webseite durch den Browser wird *Rendern* genannt. Man sagt unter Entwicklern auch: »Der Browser rendert eine Webseite.« Dabei wird HTML-, CSS- und JavaScript-Code ausgewertet, ein entsprechendes Modell der Webseite erstellt (auf das wir in Kapitel 5, »Webseiten dynamisch verändern«, noch zu sprechen kommen) und die Webseite in das Browserfenster »gezeichnet«. Im Detail ist dieser Prozess recht komplex, und wenn Sie sich mehr für dieses Thema interessieren, kann ich Ihnen den Blogbeitrag unter *www.html5rocks.com/de/tutorials/internals/howbrowserswork/* empfehlen.

2.1.2 Das richtige Werkzeug für die Entwicklung

Für das Erstellen von JavaScript-Dateien würde prinzipiell zwar ein einfacher Texteditor ausreichen (und für einfache Codebeispiele ist dies auch durchaus in Ordnung), allerdings sollten Sie sich früher oder später besser einen guten Editor zulegen, der Sie beim Schreiben von JavaScript unterstützt (sofern Sie nicht ohnehin schon einen auf Ihrem Rechner installiert haben) und der speziell für die Entwicklung von JavaScript-Programmen ausgelegt ist. Ein solcher Editor unterstützt Sie beispielsweise dahin gehend, dass er den Quelltext farblich hervorhebt, Ihnen Schreibarbeit bei wiederkehrenden Quelltextbausteinen abnimmt, Fehler im Quelltext erkennt und vieles mehr.

Editoren

Es gibt mittlerweile eine Reihe wirklich guter Editoren, mit denen sich effektiv arbeiten lässt. In der Entwickler-Community sind beispielsweise Sublime Text (*www.sublimetext.com*, siehe Abbildung 2.7) und Atom (*https://atom.io*, siehe Abbildung 2.8) beliebt, die beide für

Windows, macOS und Linux zur Verfügung stehen. Während Ersterer derzeit 99 US$ kostet (Stand: Juni 2021), ist der Editor Atom kostenlos. Im Detail haben beide Editoren ihre eigenen Features und Stärken, sind sich prinzipiell aber doch recht ähnlich. Probieren Sie einfach aus, welcher Ihnen mehr zusagt.

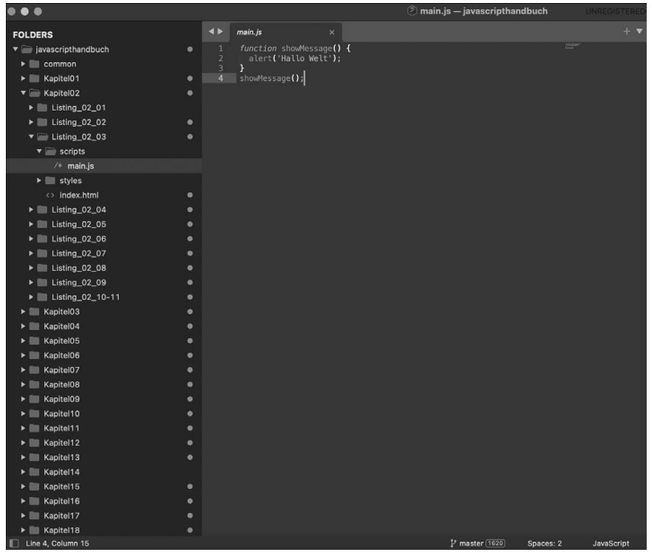

Abbildung 2.7 Der Editor Sublime Text

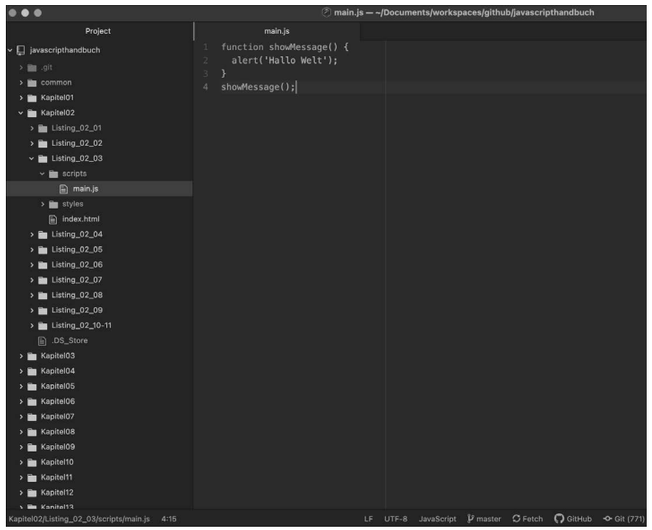

Abbildung 2.8 Der Editor Atom

Entwicklungsumgebungen

Softwareentwickler, die von Sprachen wie Java oder C++ zu JavaScript wechseln, sind von »ihren Programmiersprachen« in den meisten Fällen sogenannte *Entwicklungsumgebungen* gewohnt (im Englischen kurz *IDE* für *Integrated Development Environment*). Eine Entwicklungsumgebung können Sie sich gewissermaßen wie einen sehr, sehr mächtigen Editor vorstellen, der gegenüber einem »normalen« Editor noch diverse andere Features bereitstellt, wie beispielsweise die Synchronisation mit einem *Source-Verwaltungssystem*, das Ausführen von *automatischen Builds* oder die Integration von *Test-Frameworks*. (Wenn Sie jetzt nur verständnislos den Kopf schütteln und sich fragen, was sich hinter all diesen Begriffen verbirgt, warten Sie bis Kapitel 21, »Einen professionellen Entwicklungsprozess aufsetzen«, denn dort gehe ich auf diese fortgeschrittenen Themen der Softwareentwicklung mit JavaScript ein.)

WebStorm von IntelliJ (*www.jetbrains.com/webstorm/*, siehe Abbildung 2.9) ist ein Beispiel für eine sehr beliebte und, wie ich finde, auch wirklich sehr gute Entwicklungsumgebung. Eine Einzellizenz für WebStorm kostet derzeit 129 € (für die persönliche Nutzung gibt es noch eine Version, die derzeit 59 € kostet). Wer das Programm zunächst testen möchte, kann sich aber vorab eine 30-Tage-Testversion von der Homepage herunterladen. WebStorm steht dabei sowohl für Windows als auch für macOS und Linux zur Verfügung.

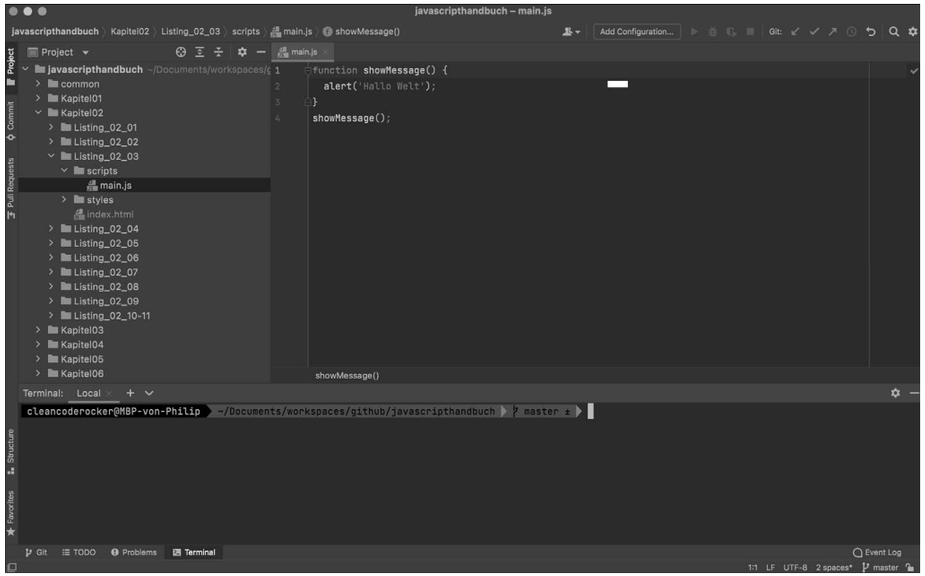

Abbildung 2.9 Die WebStorm-IDE

Mein persönlicher Favorit unter den Entwicklungsumgebungen ist mittlerweile – und hier hat sich meine Meinung seit der vorigen Auflage dieses Buchs geändert – Visual Studio Code von Microsoft (*https://code.visualstudio.com/*, siehe Abbildung 2.10). Es steht kostenlos zum

Download bereit, kann flexibel über Plug-ins erweitert werden und ist gefühlt deutlich performanter als etwa WebStorm.

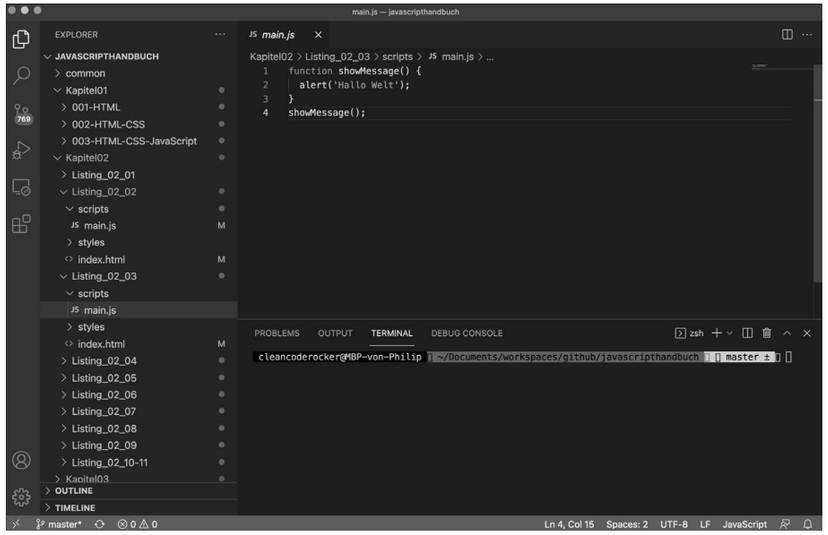

Abbildung 2.10 Microsoft Visual Studio Code

Eine kurze Übersicht über die oben genannten Editoren und Entwicklungsumgebungen finden Sie in Tabelle 2.1.

Name	Preis	macOS	Linux	Windows	Editor/Entwicklungsumgebung
Sublime Text	99 US$	ja	ja	ja	Editor
Atom	kostenlos	ja	ja	ja	Editor
Microsoft Visual Studio Code	kostenlos	ja	ja	ja	Entwicklungsumgebung
WebStorm	129 €/59 €	ja	ja	ja	Entwicklungsumgebung

Tabelle 2.1 Empfehlenswerte Editoren und Entwicklungsumgebungen für die Entwicklung mit JavaScript

Tipp

Für den Anfang – also beispielsweise für das Ausprobieren der Codebeispiele in diesem Buch – empfehle ich Ihnen, einen der genannten Editoren zu verwenden und (noch) keine Entwicklungsumgebung. Letztere haben nämlich den Nachteil, dass sie teils mit Menüs und

Funktionalitäten überfrachtet sind, sodass Sie sich – zusätzlich zum Lernen von JavaScript – auch noch mit dem Erlernen der Entwicklungsumgebung beschäftigen müssen. Das möchte ich Ihnen für den Moment zumindest möglichst ersparen.

Zudem sind Entwicklungsumgebungen eigentlich auch erst ab einer gewissen Projektgröße sinnvoll, für kleinere Projekte und die Beispiele in diesem Buch reicht ein Editor allemal (nicht dass wir nicht auch komplexe Themen behandeln werden!). Hinzu kommt, dass die Editoren in der Regel im Hinblick auf die Ausführungsgeschwindigkeit schneller als die Entwicklungsumgebungen sind.

2.2 JavaScript in eine Webseite einbinden

Da ich davon ausgehe, dass Sie bereits wissen, wie man eine HTML-Datei erstellt und wie man eine CSS-Datei einbindet und Sie »nur« hier sind, um JavaScript zu lernen, will ich auch keine weitere Zeit mit Details über HTML und CSS verschwenden, sondern gleich mit JavaScript loslegen. Keine Sorge: Das Einbinden und Ausführen einer JavaScript-Datei gestaltet sich alles andere als schwierig.

Traditionsgemäß starte ich (wie nahezu jedes Buch über Programmiersprachen) mit einem sehr einfachen sogenannten *Hello World*-Beispiel, das lediglich die Ausgabe `Hello World` (bzw. in unserem Fall die Ausgabe `Hallo Welt`) erzeugt. Das ist zwar noch wenig spannend, aber momentan geht es ja darum, Ihnen zu zeigen, wie Sie überhaupt erst mal eine JavaScript-Datei in eine HTML-Datei einbinden und den in der JavaScript-Datei enthaltenen Quelltext ausführen können. Um die komplexen Dinge kümmern wir uns dann später.

2.2.1 Eine geeignete Ordnerstruktur vorbereiten

Für den Anfang und das Durcharbeiten der folgenden Beispiele empfehle ich Ihnen, die in Abbildung 2.11 gezeigte Verzeichnisstruktur für jedes Beispiel zu verwenden. Auf oberster Ebene liegt die HTML-Datei, denn das ist für den Browser der Einstiegspunkt und damit die Datei, die Sie gleich im Browser aufrufen werden.

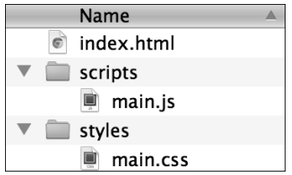

Abbildung 2.11 Exemplarische Ordnerstruktur

Für die CSS- und JavaScript-Dateien dagegen bietet es sich an, jeweils verschiedene Ordner anzulegen. Die Namen *styles* (für CSS-Dateien) und *scripts* (für JavaScript-Dateien) sind dabei

recht üblich. Insbesondere wenn Sie es während der Entwicklung mit vielen verschiedenen JavaScript- und CSS-Dateien zu tun haben, erleichtert diese Aufteilung nämlich (bzw. generell eine Aufteilung mit Unterordnern) die Übersicht über Ihr Projekt.

Startpunkt einer JavaScript-Anwendung

Die meisten Beispiele in diesem Buch folgen auch der in Abbildung 2.11 gezeigten Aufteilung, da wir für den Anfang den JavaScript-Code nur im Browser ausführen werden und dazu die HTML-Datei *index.html* gewissermaßen als Einstiegspunkt in das jeweilige Programm verwenden.

Später in Kapitel 17, »Serverseitige Anwendungen mit Node.js erstellen«, werde ich Ihnen zeigen, wie Sie JavaScript auch unabhängig von einem Browser und damit unabhängig von einer entsprechenden HTML-Datei ausführen können. In diesem Fall benötigen Sie dann keine HTML- und damit auch keine CSS-Dateien.

JavaScript im Browser ausführen

Sie können zwar auch innerhalb eines Browsers JavaScript ausführen, ohne dafür eine HTML-Datei zu erstellen, die das entsprechende Skript einbindet (nämlich über spezielle durch die Browser zur Verfügung gestellte Entwicklerwerkzeuge, siehe Abschnitt 2.3.2, »Auf die Konsole schreiben«), für den Anfang wollen wir dieses Feature aber noch nicht verwenden.

2.2.2 Eine JavaScript-Datei erstellen

JavaScript-Code sollte man also besser in einer separaten Datei (oder mehreren separaten Dateien) speichern und diese dann in den HTML-Code einbinden. Sie benötigen also als Erstes eine JavaScript-Datei. Dazu öffnen Sie einfach den Editor Ihrer Wahl (oder falls Sie nicht auf meinen Rat gehört haben: die Entwicklungsumgebung Ihrer Wahl), erstellen eine neue Datei, geben folgende Zeilen Quelltext dort hinein und speichern die Datei anschließend unter dem Namen *main.js*.

```
function showMessage() {
  alert('Hallo Welt');
}
```

Listing 2.1 Ein ganz einfaches JavaScript-Beispiel, in dem eine Funktion definiert wird

Merke

JavaScript-Dateien haben die Endung *.js*. Prinzipiell sind zwar auch andere Dateiendungen möglich, allerdings hat die Endung *.js* den Vorteil, dass Editoren, Entwicklungsumgebungen und Browser direkt wissen, worum es sich bei dem Inhalt handelt. Sie sollten also alle JavaScript-Dateien immer mit der Endung *.js* abspeichern (Browser erkennen JavaScript-Dateien,

die von einem Webserver geliefert werden, übrigens an dem sogenannten *Content-Type-Header*, einer Information, die mit der jeweiligen Datei vom Server mitgeliefert wird).

Definiert ist in Listing 2.1 eine sogenannte *Funktion* mit Namen `showMessage`, die ihrerseits eine andere Funktion (mit Namen `alert`) aufruft und dieser die Meldung `Hallo Welt` übergibt. Die Funktion `alert` ist eine Standardfunktion von JavaScript, auf die ich später in diesem Kapitel noch mal kurz zu sprechen komme. Mit Funktionen im Allgemeinen werden wir uns dagegen detailliert in Kapitel 3, »Sprachkern«, beschäftigen.

Downloadbereich zum Buch

Dieses und alle folgenden Codebeispiele finden Sie auch im Downloadbereich zum Buch (siehe *www.rheinwerk-verlag.de/5392*). Von dort können Sie den Code bequem herunterladen und in Ihrem Editor oder direkt im Browser öffnen (wobei ich ja der Meinung bin, dass man am effektivsten lernt, wenn man die Beispiele selbst abtippt und dabei Schritt für Schritt nachvollzieht).

2.2.3 Eine JavaScript-Datei in eine HTML-Datei einbinden

Um den JavaScript-Quelltext jetzt innerhalb einer Webseite verwenden zu können, müssen Sie die JavaScript-Datei mit der Webseite verknüpfen bzw. die JavaScript-Datei in die HTML-Datei einbinden. Das geschieht über das HTML-Element `<script>`.

Dieses Element kann prinzipiell auf zwei verschiedene Arten genutzt werden: Zum einen können – wie ich direkt im Anschluss zeigen werde – externe JavaScript-Dateien in das HTML eingebunden werden, zum anderen kann JavaScript-Quelltext auch direkt zwischen das öffnende `<script>`-Tag und das schließende `</script>`-Tag geschrieben werden.

Zu Letzterem zeige ich Ihnen später noch ein Beispiel, allerdings ist diese Vorgehensweise eher nur in Ausnahmefällen sinnvoll, weil dann JavaScript-Code und HTML-Code vermischt, sprich in einer Datei gespeichert werden (was wiederum aus genannten Gründen keine Best Practice ist). Schauen wir uns also erst an, wie man es richtig macht und eine separate Datei einbindet.

Das `<script>`-Element hat insgesamt sechs Attribute, von denen das `src`-Attribut mit Sicherheit das wichtigste ist: Hierüber wird der Pfad zu der JavaScript-Datei angegeben, die eingebunden werden soll (eine Übersicht darüber, wofür auch die anderen Attribute gedacht sind, zeigt Tabelle 2.2).

Erstellen Sie nun also eine HTML-Datei mit Namen *index.html* und fügen Sie dort den folgenden in Listing 2.2 gezeigten Inhalt ein.

```
<!DOCTYPE html>
<html>
<head lang="de">
```

```
  <meta charset="UTF-8">
  <title>Beispiel</title>
  <link rel="stylesheet" href="styles/main.css" type="text/css">
</head>
<body>
<!--Hier wird die JavaScript-Datei eingebunden -->
<script src="scripts/main.js"></script>
</body>
</html>
```

Listing 2.2 Einbinden von JavaScript in HTML

Wenn Sie jetzt diese HTML-Datei im Browser öffnen, passiert noch nichts, denn die Funktion, die wir in Listing 2.1 definiert haben, wird noch an keiner Stelle aufgerufen. Ergänzen Sie daher am Ende der JavaScript-Datei den Aufruf `showMessage()` und laden Sie die Webseite im entsprechenden Browser neu. Dann sollte sich ein kleiner Hinweisdialog öffnen, der die Meldung `Hallo Welt` enthält und je nach Browser ein etwas anderes Aussehen hat (siehe Abbildung 2.12).

```
function showMessage() {
  alert('Hallo Welt');
}
showMessage();
```

Listing 2.3 Funktionsdefinition und Funktionsaufruf

Attribut	Bedeutung	Anmerkung
`async`	Gibt an, ob das Herunterladen der verlinkten JavaScript-Datei asynchron stattfinden soll und das Herunterladen anderer Dateien nicht unterbrochen wird (siehe Abschnitt 2.2.5). Ergibt nur in Kombination mit dem `src`-Attribut Sinn.	optional
`charset`	Gibt den Zeichensatz des Quelltextes an, der über das `src`-Attribut eingebunden wird. Ergibt nur in Kombination mit dem `src`-Attribut Sinn, wird aber selten verwendet, weil die meisten Browser dieses Attribut nicht beachten. Zudem ist es besserer Stil, innerhalb einer Website überall UTF-8 zu verwenden und dies im `<meta>`-Element über das `charset`-Attribut zu definieren.	optional
`defer`	Gibt an, ob mit dem Ausführen der verlinkten JavaScript-Datei bis zu dem Zeitpunkt gewartet werden soll, zu dem der Inhalt der Webseite komplett verarbeitet wurde (siehe Abschnitt 2.2.5). Ergibt nur in Kombination mit dem `src`-Attribut Sinn, wird aber insbesondere von älteren Browsern nicht unterstützt.	optional

Tabelle 2.2 Die Attribute des <script>-Elements

Attribut	Bedeutung	Anmerkung
`language`	Ursprünglich dazu gedacht, die Version des verwendeten JavaScript-Codes anzugeben, wird von Browsern aber weitestgehend nicht beachtet.	veraltet
`src`	Gibt den Pfad zu der JavaScript-Datei an, die eingebunden werden soll.	optional
`type`	Dient der Angabe des *MIME-Types* (siehe Kasten), um die Skriptsprache (in unserem Fall JavaScript) zu identifizieren. Prinzipiell können Sie das Attribut jedoch weglassen, da in diesem Fall standardmäßig `text/javascript` verwendet wird, das von den meisten Browsern unterstützt wird.	optional

Tabelle 2.2 Die Attribute des <script>-Elements (Forts.)

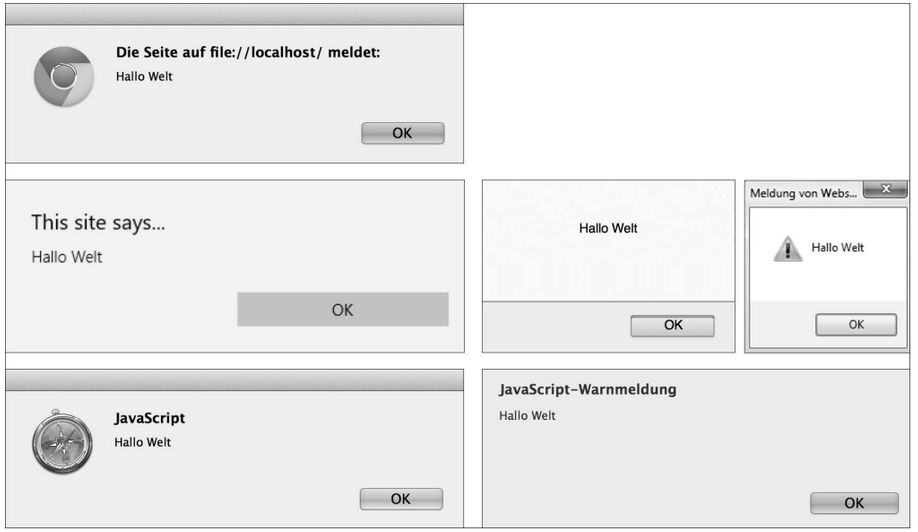

Abbildung 2.12 Hinweisdialoge in den verschiedenen Browsern

Definition

MIME-Types (*Multipurpose Internet Mail Extension*, auch *Internet Media Type* oder *Content Type* genannt) waren ursprünglich dafür gedacht, innerhalb von E-Mails, die verschiedene Inhalte (wie Bilder, PDF-Dateien etc.) enthalten, zwischen den einzelnen Inhaltstypen zu unterscheiden. Mittlerweile werden MIME-Types aber nicht nur im Zusammenhang mit E-Mails verwendet, sondern immer, wenn Daten über das Internet übertragen werden. Sen-

det ein Server eine Datei mit einem speziellen MIME-Type, weiß der Client (z. B. der Browser) direkt, um welchen Typ es sich bei den übertragenen Daten handelt.

Für JavaScript war der MIME-Type lange nicht standardisiert, sodass es direkt mehrere MIME-Types gab, wie beispielsweise `application/javascript`, `application/ecmascript`, `text/javascript` und `text/ecmascript`. Seit 2006 gibt es aber einen offiziellen Standard (*www.rfc-editor.org/rfc/rfc4329.txt*), der die erlaubten MIME-Types für JavaScript definiert. Demnach sind `text/javascript` und `text/ecmascript` beide veraltet, stattdessen sollten `application/javascript` und `application/ecmascript` verwendet werden. Paradoxerweise ist es am sichersten, im Fall von JavaScript (im `<script>`-Element) keinen MIME-Type anzugeben, da das `type`-Attribut ohnehin von den meisten Browsern ignoriert wird.

Mehrere JavaScript-Dateien einbinden

Sie können innerhalb einer HTML-Datei selbstverständlich auch mehrere JavaScript-Dateien einbinden. Dann verwenden Sie für jede Datei, die eingebunden werden soll, einfach ein eigenes `<script>`-Element.

2.2.4 JavaScript direkt innerhalb des HTML definieren

Der Vollständigkeit halber zeige ich Ihnen noch, wie Sie JavaScript auch direkt innerhalb einer HTML-Datei definieren können. Das ist zwar in der Regel nicht ratsam, weil man auf diese Weise HTML- und JavaScript-Code in einer Datei mischt, zu wissen, dass es trotzdem geht, schadet aber nicht.

Dazu schreiben Sie den entsprechenden JavaScript-Code einfach innerhalb des `<script>`-Elements, statt ihn über das `src`-Attribut zu verlinken. Listing 2.4 zeigt das Beispiel von eben, verwendet aber keine separate JavaScript-Datei für den JavaScript-Code, sondern bindet diesen direkt in das HTML ein. Das `src`-Attribut fällt daher komplett weg.

```
<!DOCTYPE html>
<html>
<head lang="de">
  <meta charset="UTF-8">
  <title>Beispiel</title>
  <link rel="stylesheet" href="styles/main.css" type="text/css">
</head>
<body>
<script>
  function showMessage() {
    alert('Hallo Welt');
  }
  showMessage();
</script>
</body>
</html>
```

Listing 2.4 Nur in Ausnahmefällen sinnvoll: Definition von JavaScript direkt in einer HTML-Datei

Hinweis

Beachten Sie, dass `<script>`-Elemente, die das `src`-Attribut verwenden, keinen Quelltext zwischen `<script>` und `</script>` haben dürfen. Sollte dies dennoch der Fall sein, wird dieser Quelltext ignoriert.

Tipp

Verwenden Sie separate JavaScript-Dateien für Ihren Quelltext, statt ihn direkt in ein `<script>`-Element zu schreiben. Das schafft eine saubere Trennung zwischen der Struktur (HTML) und dem Verhalten (JavaScript) einer Webseite.

Das <noscript>-Element

Über das `<noscript>`-Element können Sie einen HTML-Abschnitt definieren, der angezeigt wird, wenn JavaScript im Browser nicht unterstützt wird oder vom Nutzer deaktiviert wurde. Wird dagegen JavaScript unterstützt bzw. ist es aktiviert, wird der Inhalt des `<noscript>`-Elements nicht angezeigt.

```
<noscript>
  JavaScript ist nicht verfügbar oder es ist deaktiviert. <br />
  Bitte verwenden Sie einen Browser, der JavaScript unterstützt,
  oder aktivieren Sie JavaScript in Ihrem Browser.
</noscript>
```

Listing 2.5 Beispiel für die Verwendung des <noscript>-Elements

2.2.5 Platzierung und Ausführung der <script>-Elemente

Hätten Sie vor einigen (vielen) Jahren einen Webentwickler gefragt, an welcher Stelle ein `<script>`-Element innerhalb einer Webseite einzubinden ist, hätte dieser wahrscheinlich dazu geraten, es im `<head>`-Bereich der Webseite unterzubringen. In den Anfangstagen der Webentwicklung war man nämlich der Ansicht, verlinkte Dateien wie CSS-Dateien und eben JavaScript-Dateien sollten an einer zentralen Stelle innerhalb des HTML-Codes platziert werden.

Mittlerweile ist man allerdings wieder davon abgekommen. CSS-Dateien werden zwar weiterhin im `<head>`-Bereich platziert, JavaScript-Dateien dagegen sollten vor dem schließenden

</body>-Tag eingebunden werden. Der Grund dafür ist folgender: Wenn der Browser eine Webseite lädt, lädt er neben dem HTML-Code auch die eingebundenen Dateien wie beispielsweise Bilder, CSS-Dateien und JavaScript-Dateien. Je nach Prozessorleistung und Speicherauslastung sind moderne Browser dazu in der Lage, mehrere solcher Dateien parallel herunterzuladen. Wenn der Browser allerdings ein <script>-Element vorfindet, fängt er sofort damit an, den entsprechenden Quelltext zu verarbeiten und mithilfe des JavaScript-Interpreters auszuwerten. Um dies aber zu können, muss der entsprechende JavaScript-Quelltext zunächst vollständig heruntergeladen werden. Während das passiert, pausiert der Browser jedoch das Herunterladen aller anderen Dateien und das *Parsen* (also das Verarbeiten) des HTML-Codes, was wiederum zur Folge hat, dass für den Nutzer das Aufbauen der Webseite gefühlt länger dauert (siehe Abbildung 2.13).

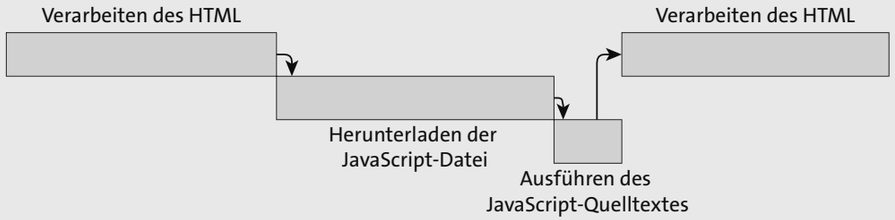

Abbildung 2.13 Standardmäßig wird die Verarbeitung des HTML-Codes gestoppt, wenn der Browser auf ein <script>-Element trifft.

Hinzu kommt, dass man innerhalb des JavaScript-Quelltextes häufig auf HTML-Elemente der jeweiligen Webseite zugreifen möchte (wie das genau funktioniert, zeige ich Ihnen in Kapitel 5, »Webseiten dynamisch verändern«). Wenn dann der JavaScript-Code ausgeführt wird, bevor die Verarbeitung dieser HTML-Elemente stattgefunden hat, kommt es zu einem Zugriffsfehler (siehe Abbildung 2.14). Platzieren Sie dagegen das <script>-Element vor dem schließenden </body>-Tag, sind Sie diesbezüglich auf der sicheren Seite (siehe Abbildung 2.15), da in dem Fall alle Elemente, die sich innerhalb des <body>-Elements befinden, bereits geladen sind (mit Ausnahme anderer <script>-Elemente selbstverständlich).

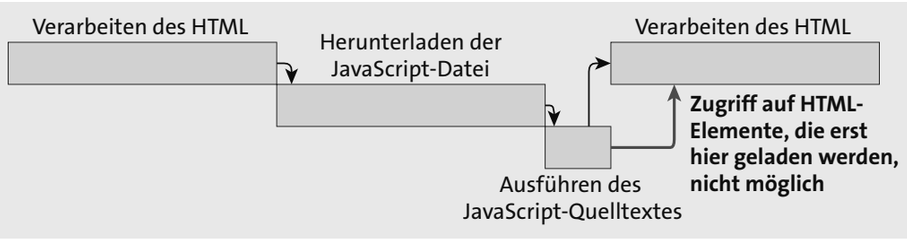

Abbildung 2.14 Wenn JavaScript auf noch nicht geladene HTML-Elemente zugreift, kommt es zu einem Fehler.

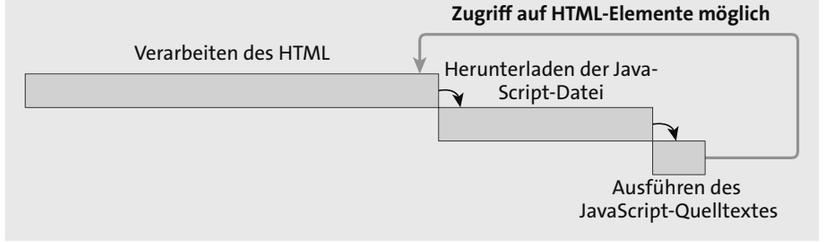

Abbildung 2.15 Wird das <script>-Element vor das schließende </body>-Tag platziert, sind dagegen alle Elemente innerhalb des <body>-Elements geladen.

Merke

In der Regel sollten Sie <script>-Elemente am Ende des <body>-Elements positionieren. Das liegt daran, dass der Browser bei jedem <script>-Element zunächst den dort enthaltenen bzw. eingebundenen JavaScript-Quelltext auswertet, bevor er mit dem Laden weiterer HTML-Elemente fortfährt.

Zwei Attribute, über die man das Ladeverhalten von JavaScript beeinflussen kann, sind die Attribute async und defer, die ich ja eben schon kurz erwähnt hatte (siehe Tabelle 2.2). Ersteres sorgt dafür, dass das Verarbeiten des HTML-Codes nicht pausiert wird, wenn der Browser auf ein <script>-Element trifft. Das Herunterladen der JavaScript-Datei passiert quasi asynchron (daher der Name async). Das Prinzip davon zeigt Abbildung 2.16.

Wie Sie sehen können, wird der JavaScript-Code auch hier direkt ausgeführt, sobald die entsprechende JavaScript-Datei vollständig heruntergeladen wurde.

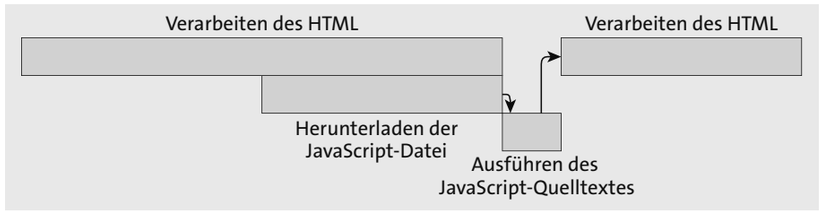

Abbildung 2.16 Über das Attribut »async« wird der HTML-Code so lange weiterverarbeitet, bis das entsprechende JavaScript heruntergeladen wurde.

Einen Schritt weiter geht das Attribut defer. Dieses Attribut sorgt nämlich zum einen dafür, dass – wie bei async – das Verarbeiten des HTML-Codes nicht pausiert wird. Zum anderen wird der JavaScript-Quelltext erst ausgeführt, nachdem der HTML-Code vollständig verarbeitet wurde (siehe Abbildung 2.17). Die Ausführung des JavaScript-Codes wird quasi verschoben (daher der Name defer).

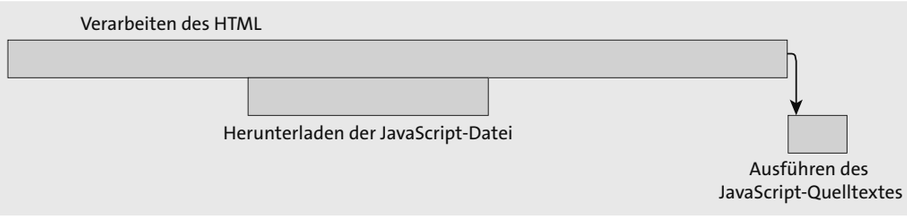

Abbildung 2.17 Über das Attribut »defer« erreicht man, dass das entsprechende JavaScript erst dann ausgeführt wird, nachdem der gesamte HTML-Code der Webseite geladen wurde.

Wann sollten Sie also welches Attribut einsetzen? Für den Moment können Sie sich merken, dass Sie wahrscheinlich am besten bedient sind, wenn Sie standardmäßig keines der beiden Attribute verwenden. Das Attribut `async` eignet sich eigentlich nur für solche Skripte, die komplett eigenständig funktionieren und quasi nichts mit dem HTML auf der Webseite »zu tun haben«. Ein Beispiel hierfür wäre die Verwendung von Google Analytics. Das Attribut `defer` dagegen wird momentan noch nicht von allen Browsern unterstützt, sodass Sie auch hier eine Verwendung mit Vorsicht abwägen sollten.

Definition

Eine weitere Möglichkeit, sicherzustellen, dass der gesamte Inhalt der Webseite geladen wurde, bevor JavaScript-Code ausgeführt wird, ist die Verwendung sogenannter *Ereignis-Handler* und *Ereignis-Listener* (auch *Event-Handler* und *Event-Listener* genannt). Beide werde ich Ihnen in Kapitel 6, »Ereignisse verarbeiten und auslösen«, detailliert vorstellen, an dieser Stelle zeige ich Ihnen aber schon mal grob, wie beide verwendet werden, weil sie in den Quelltextbeispielen im Buch schon vor den Beispielen zu Kapitel 6 auftauchen.

Beide, sowohl Ereignis-Handler als auch Ereignis-Listener, dienen allgemein gesagt dazu, auf bestimmte Ereignisse, die bei der Ausführung eines Programms auftreten, zu reagieren und bestimmten Code auszuführen (es gibt zwar einen kleinen, feinen Unterschied zwischen Ereignis-Handlern und Ereignis-Listenern, der für den Moment aber nicht wichtig ist und den ich Ihnen dann in Kapitel 6 erklären werde). Ereignisse können Mausklicks, Tastatureingaben, Änderungen der Fenstergröße und vieles mehr sein. Auch für Webseiten gibt es verschiedene Ereignisse, die ausgelöst werden und auf die man mit solchen Ereignis-Handlern und Ereignis-Listenern reagieren kann. So wird ebenfalls ein Ereignis ausgelöst, wenn der Inhalt einer Webseite vollständig geladen wurde.

Um einen Ereignis-Handler für dieses Ereignis zu definieren, können Sie das Attribut `onload` verwenden: Der Code, den Sie hier als Wert für ein solches Attribut angeben, wird aufgerufen, wenn die Webseite vollständig geladen wurde. Als Wert kann man hier eine JavaScript-*Anweisung* angeben, z. B. den Aufruf einer Funktion, wie in Listing 2.6 gezeigt:


```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Beispiel</title>
  <link rel="stylesheet" href="styles/main.css" type="text/css">
</head>
<body onload="showMessage()">
<script src="scripts/main.js"></script>
</body>
</html>
```

Listing 2.6 Verwenden eines Event-Handlers

Ereignis-Listener dagegen können nicht über HTML definiert werden, stattdessen verwendet man die Funktion `addEventListener()` des Objekts `document` (dazu später mehr), der man den Namen des Ereignisses übergibt sowie die Funktion, die ausgeführt werden soll, wenn das Ereignis ausgelöst wird. Listing 2.7 zeigt ein entsprechendes Beispiel.

```
function showMessage() {
  alert('Hallo Welt');
}
document.addEventListener('DOMContentLoaded', showMessage);
```

Listing 2.7 Verwenden von Event-Listenern

Den Aufruf `showMessage()`, den Sie eben an das Ende der Datei *main.js* angefügt hatten, müssten Sie in beiden Fällen wieder entfernen, sonst wird die Funktion zweimal aufgerufen (einmal durch das Skript selbst und einmal durch den Ereignis-Handler/Ereignis-Listener), und entsprechend wird zweimal hintereinander ein Hinweisdialog angezeigt.

2.2.6 Den Quelltext anzeigen

Alle Browser bieten in der Regel eine Möglichkeit, sich den Quelltext einer Webseite anzeigen zu lassen. Dies kann in vielen Fällen hilfreich sein, beispielsweise um zu schauen, wie ein bestimmtes Feature auf einer Website, die Sie entdeckt haben, implementiert ist.

Unter Chrome können Sie den Quelltext einsehen, indem Sie den Menüpunkt Anzeigen • Entwickler • Quelltext anzeigen aufrufen (siehe Abbildung 2.18), in Firefox über Extras • Browser-Werkzeuge • Seitenquelltext anzeigen (siehe Abbildung 2.19), in Safari über Entwickler • Seitenquelltext einblenden (siehe Abbildung 2.20), in Opera über Entwickler • Quelltext Anzeigen (siehe Abbildung 2.21) und in Microsoft Edge über Extras • Entwickler • Quelle anzeigen (siehe Abbildung 2.22).

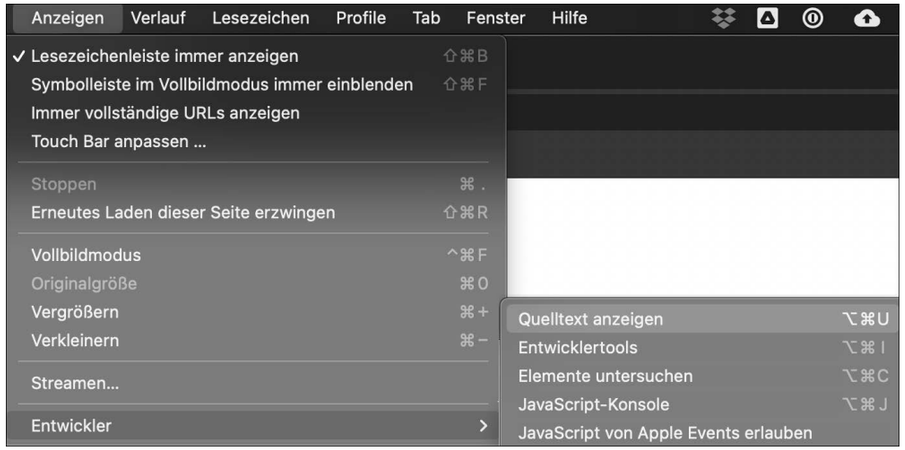

Abbildung 2.18 Quelltext anzeigen in Chrome

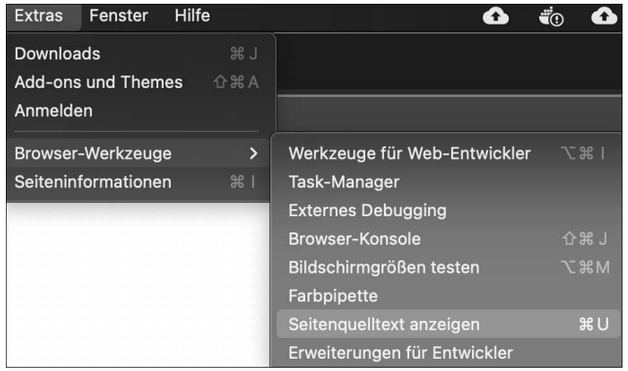

Abbildung 2.19 Quelltext anzeigen in Firefox

Quelltext von umfangreicheren Webseiten

Wenn Sie sich den Quelltext von umfangreicheren Webseiten anschauen, ist dieser häufig sehr unübersichtlich. Das hat in der Regel verschiedene Gründe: Zum einen werden Inhalte häufig dynamisch generiert, zum anderen wird JavaScript durch Webentwickler oft bewusst komprimiert und unkenntlich gemacht – Ersteres, um Platz zu sparen, Letzteres, um den Quelltext vor fremden Blicken zu schützen. Mit Komprimierung und Unkenntlichmachung des Quelltextes beschäftigen wir uns in diesem Buch nicht. Wenn Sie Interesse daran haben, kann ich Ihnen mein Buch *Professionell entwickeln mit JavaScript: Design, Patterns und Praxistipps* (2018, ISBN 978-3-8362-5687-2) empfehlen, das sich mit solch fortgeschrittenen Themen befasst und ebenfalls beim Rheinwerk Verlag erschienen ist.

Abbildung 2.20 Quelltext anzeigen in Safari

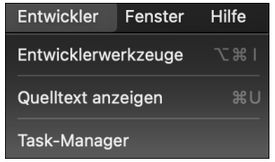

Abbildung 2.21 Quelltext anzeigen in Opera

Abbildung 2.22 Quelltext anzeigen in Microsoft Edge

Wenn man sich (egal in welchem Browser) den Quelltext einer Webseite anzeigen lässt, befindet man sich natürlich erst mal im entsprechenden HTML-Code der Webseite. Praktischerweise sind eingebundene Dateien wie beispielsweise CSS-Dateien oder JavaScript-Dateien aber in dieser Quelltextansicht verlinkt (siehe Abbildung 2.23), sodass Sie hierüber bequem auch zum Quelltext der verlinkten Datei gelangen (siehe Abbildung 2.24).

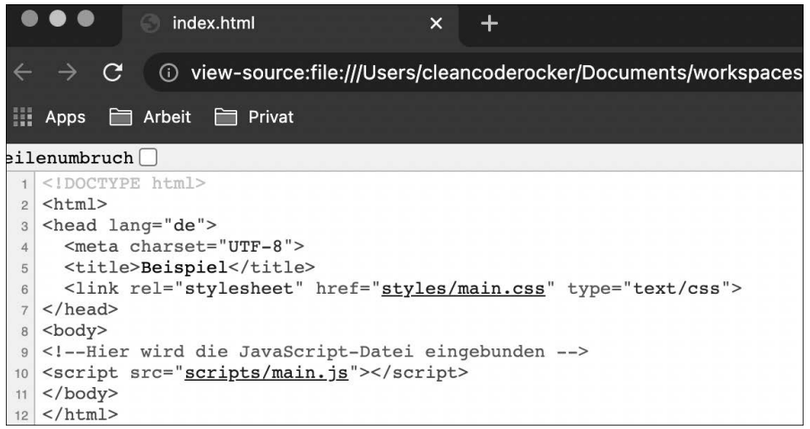

Abbildung 2.23 Quelltextansicht für HTML in Chrome

Abbildung 2.24 Quelltextansicht für JavaScript in Chrome

2.3 Eine Ausgabe erzeugen

Im *Hello World*-Beispiel haben Sie ja bereits gesehen, wie Sie über einen Aufruf der Funktion `alert()` eine einfache Ausgabe erzeugen können. Es gibt aber noch verschiedene andere Möglichkeiten.

2.3.1 Standarddialogfenster anzeigen

Neben dem bereits bekannten Hinweisdialog über den Aufruf der Funktion `alert()` (siehe Abbildung 2.25) gibt es noch zwei weitere im Sprachumfang von JavaScript enthaltene Standardfunktionen zur Darstellung von Dialogfenstern. Die erste ist die Funktion `confirm()`. Sie dient der Darstellung von *Bestätigungsdialogen*, sprich Ja/Nein-Entscheidungen (siehe Abbildung 2.27). Im Gegensatz zum Hinweisdialog verfügt der Bestätigungsdialog über zwei

Schaltflächen: eine zum Bestätigen der entsprechenden Meldung, eine zum Abbrechen. Die zweite ist die Funktion `prompt()`. Diese öffnet einen *Eingabedialog*, in dem Nutzer einen Text eingeben können (siehe Abbildung 2.26).

Abbildung 2.25 Ein einfacher Hinweisdialog

Abbildung 2.26 Ein einfacher Eingabedialog

Abbildung 2.27 Ein einfacher Bestätigungsdialog

In der Praxis kommen diese Standarddialoge für Hinweise, Bestätigungen und Eingaben jedoch eher selten zum Einsatz, da sie zum einen in den Ausdrucksmöglichkeiten begrenzt sind, zum anderen aber auch – wie Sie beim Hinweisdialog bereits sehen konnten – optisch dem Layout des jeweiligen Browsers entsprechen und in der Regel nicht zum Layout der Webseite passen.

Aus diesem Grund greift man als Webentwickler gerne auf eine der diversen JavaScript-Bibliotheken zurück, die schickere und funktionalere Dialoge anbieten (siehe Abbildung 2.28). Eine dieser Bibliotheken ist jQuery UI, die auf der bekannten Bibliothek jQuery aufbaut und diese um verschiedene UI-Komponenten erweitert. Die Hauptbibliothek jQuery sowie jQuery UI nehmen wir in Kapitel 10, »Aufgaben vereinfachen mit jQuery«, genauer unter die Lupe.

Abbildung 2.28 Ein individueller Bestätigungsdialog mit JavaScript

2.3.2 Auf die Konsole schreiben

Häufig ist es so, dass Sie bei der Entwicklung von JavaScript-Anwendungen eine Ausgabe nur zu Testzwecken für sich selbst erzeugen möchten, beispielsweise um ein Zwischenergebnis auszugeben. Für solche nur zu Testzwecken gedachten Ausgaben ergibt es natürlich keinen Sinn, diese in Dialogen zu zeigen, die auch Nutzer zu Gesicht bekommen würden. Aus diesem Grund bieten mittlerweile alle aktuellen Browser eine sogenannte *Konsole* an, die für genau solche Zwecke geeignet ist und auf die Sie innerhalb eines JavaScript-Programms zugreifen können, um Meldungen auszugeben. Standardmäßig ist diese Konsole ausgeblendet, da Nutzer einer Webseite in der Regel wenig damit anfangen können.

Die Konsole anzeigen

Um die Konsole zu aktivieren, gehen Sie je nach Browser wie folgt vor (auf Screenshots habe ich an dieser Stelle verzichtet, weil die Menüpunkte jeweils an ähnlicher Stelle zu finden sind wie weiter oben in diesem Kapitel die Menüpunkte, um den Quelltext anzuzeigen):

- Unter Chrome wählen Sie Anzeigen • Entwickler • JavaScript-Konsole.
- In Firefox öffnen Sie die Konsole über Extras • Browser-Werkzeuge • Browser-Konsole.
- Unter Safari öffnen Sie die Konsole über Entwickler • JavaScript-Konsole einblenden.
- In Opera müssen Sie zunächst Entwickler • Entwicklerwerkzeuge auswählen und anschließend den Tab Console.
- In Microsoft Edge öffnen Sie die Konsole über Extras • Entwickler • JavaScript-Konsole.

Abbildung 2.29 zeigt exemplarisch für den Browser Chrome, wie die Konsole aussieht: Wie Sie sehen, nichts wirklich Besonderes, allerdings wird dies eines Ihrer Hauptwerkzeuge sein, wenn Sie JavaScript für die Webentwicklung einsetzen möchten. Neben Ausgaben können Sie über die Konsole nämlich auch Eingaben tätigen (dazu in wenigen Momenten mehr). Mehr oder weniger handelt es sich bei der Konsole so gesehen um eine Art Terminal (oder Eingabeaufforderung, wenn Sie Windows-Nutzer sind), über das Sie JavaScript-Befehle absetzen können, die dann im Kontext der jeweils geladenen Webseite ausgeführt werden.

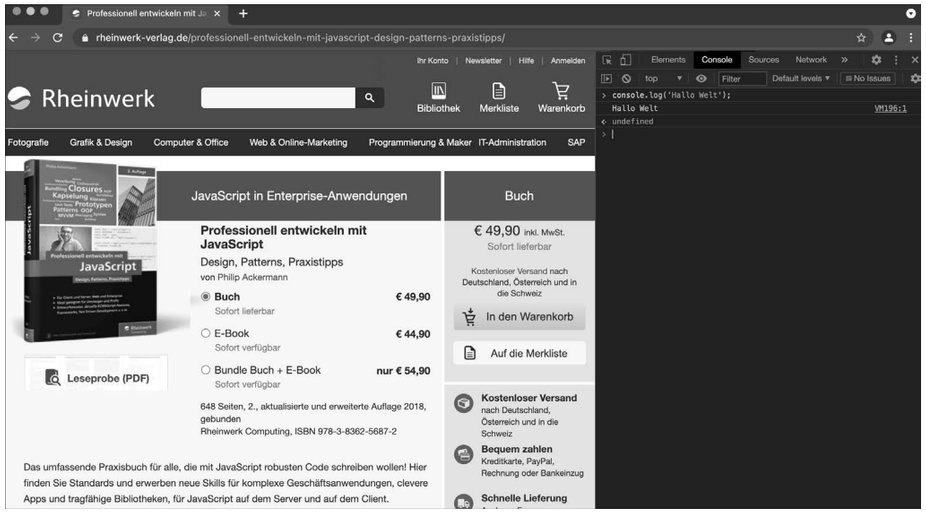

Abbildung 2.29 Standardmäßig wird die Konsole am rechten oder am unteren Rand des Browserfensters angezeigt (hier Google Chrome).

Ausgaben auf die Konsole schreiben

Um auf die Konsole schreiben zu können, stellen Browser das `console`-Objekt zur Verfügung. Dabei handelt es sich um ein JavaScript-Objekt, das erstmals durch das Firefox-Plug-in Firebug (*https://getfirebug.com*) eingeführt wurde und verschiedene Möglichkeiten bietet, Ausgaben auf der Konsole zu erzeugen. Mittlerweile steht das `console`-Objekt (obwohl immer noch nicht im ECMAScript-Standard enthalten) in nahezu jeder JavaScript-Laufzeitumgebung zur Verfügung.

> **Standardisierte API für die Arbeit mit der Konsole**
>
> Die einzelnen Methoden, die das Objekt zur Verfügung stellt, unterscheiden sich jedoch von Laufzeitumgebung zu Laufzeitumgebung. Um dem entgegenzuwirken, gibt es daher bereits Bestrebungen, an einer standardisierten API zu arbeiten.

Immer unterstützt wird die Methode `log()`, mit der eine einfache Konsolenausgabe erzeugt werden kann. Um die Verwendung der Konsole auszuprobieren, ersetzen Sie den Quelltext der Datei *main.js* einfach mit folgendem Quelltext und rufen die Webseite erneut auf.

```
// scripts/main.js
function showMessage() {
  console.log('Hallo Entwicklerwelt');
}
```

Listing 2.8 Ein einfaches JavaScript-Beispiel

Das Ergebnis sollte – je nach Browser – wie in folgender Abbildung aussehen:

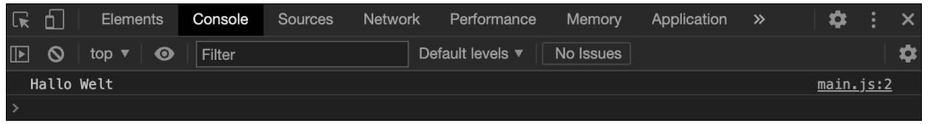

Abbildung 2.30 Ausgabe auf der Konsole in Chrome

Neben der Methode `log()` bietet `console` noch einige weitere Methoden an, von denen Tabelle 2.3 eine Übersicht der wichtigsten zeigt.

Methode	Beschreibung
`clear()`	Leert die Konsole.
`debug()`	Dient dazu, eine für das *Debuggen* (sprich die *Fehlerbehebung*) gedachte Meldung auszugeben (eventuell müssen Sie in den entsprechenden Entwicklertools erst einstellen, dass diese Art der Ausgaben ausgegeben werden sollen).
`error()`	Dient dazu, eine Fehlermeldung auszugeben. In manchen Browsern wird innerhalb der Konsole ein Fehlersymbol neben der ausgegebenen Meldung dargestellt.
`info()`	Hierdurch wird eine Infomeldung auf der Konsole ausgegeben. Chrome beispielsweise gibt zusätzlich ein Infosymbol mit aus.
`log()`	Die wohl am häufigsten verwendete Methode von `console`. Erzeugt eine normale Ausgabe auf der Konsole.
`trace()`	Gibt den sogenannten *Stack-Trace*, also den *Methodenaufruf-Stack* (siehe auch Kapitel 3, »Sprachkern«), auf der Konsole aus.
`warn()`	Dient dazu, eine Warnung auf der Konsole auszugeben. Auch hier wird in den meisten Browsern ein entsprechendes Symbol neben der Meldung ausgegeben.

Tabelle 2.3 Die wichtigsten Methoden des »console«-Objekts

Listing 2.9 zeigt den entsprechenden Quelltext für die Verwendung des `console`-Objekts. Die Ausgaben für die einzelnen Methoden werden je nach Browser farblich oder durch Symbole hervorgehoben (siehe Abbildung 2.31).

```
console.log('Hallo Entwicklerwelt');     // Ausgabe einer normalen Meldung
console.debug('Hallo Entwicklerwelt');   // Ausgabe einer Debug-Meldung
console.error('Hallo Entwicklerwelt');   // Ausgabe einer Fehlermeldung
console.info('Hallo Entwicklerwelt');    // Ausgabe einer Infomeldung
console.warn('Hallo Entwicklerwelt');    // Ausgabe einer Warnung
```

Listing 2.9 Verwendung des »console«-Objekts

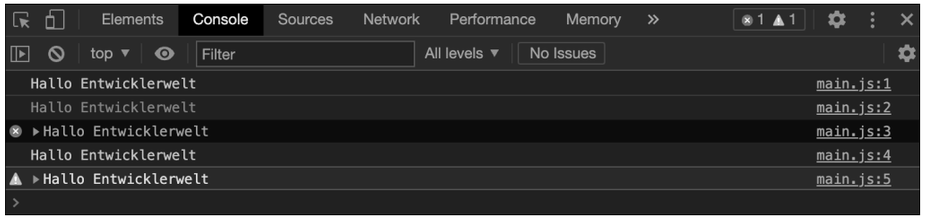

Abbildung 2.31 Die verschiedenen Meldungstypen werden farblich oder durch Symbole hervorgehoben.

Eingaben auf der Konsole schreiben

Wenn Sie sich die Screenshots genauer ansehen, werden Sie vielleicht unterhalb der Ausgabe das >-Zeichen bemerkt haben. An dieser Stelle können Sie beliebigen JavaScript-Code eingeben und direkt ausführen lassen. Ein prima Weg, um schnell einfache Skripte auszuprobieren, und für die Webentwicklung eigentlich unersetzlich. Probieren Sie es aus: Schreiben Sie den Befehl `showMessage()` in die Eingabe und drücken Sie anschließend die ↵-Taste, um den Befehl auszuführen. Das Ergebnis sehen Sie in Abbildung 2.32.

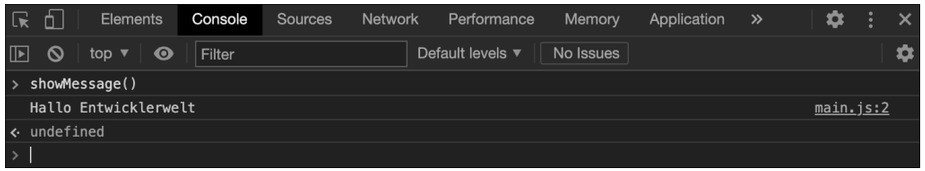

Abbildung 2.32 Über die Konsole können Sie auch Quelltext ausführen.

Merke

Das Konsolenfenster und das `console`-Objekt sind wichtige Werkzeuge für Webentwickler. Machen Sie sich mit beidem bei Gelegenheit gut vertraut.

Logging-Bibliotheken

Das `console`-Objekt eignet sich gut für die schnelle Ausgabe während der Entwicklung. Wird eine Webseite dagegen live geschaltet bzw. eine JavaScript-Anwendung produktiv eingesetzt, will man eigentlich keine `console`-Aufrufe mehr haben (auch wenn der Nutzer sie in der Regel ohnehin nicht sehen würde). In der Praxis verwendet man daher häufig spezielle *Logging-Bibliotheken*, mit deren Hilfe sich Konsolenausgaben über bestimmte Konfigurationseinstellungen einschalten (für die Entwicklung), aber auch wieder abschalten lassen (für den Produktiveinsatz). Für den Anfang und auch die Beispiele in diesem Buch soll uns aber die Verwendung des `console`-Objekts ausreichen.

2.3.3 Bestehende UI-Komponenten verwenden

Während der Einsatz von `alert()`, `confirm()` und `prompt()` eher veraltet und nur zum schnellen Testen sinnvoll und die Ausgabe über das `console`-Objekt ohnehin nur Entwicklern vorbehalten ist, braucht man natürlich noch einen Weg, um eine ansprechende Ausgabe für den Nutzer einer Webseite zu erzeugen. Dazu können Sie die Ausgabe eines Programms in bestehende UI-Komponenten wie Textfelder etc. hineinschreiben.

Listing 2.10, Listing 2.11 und Abbildung 2.33 zeigen hierzu ein Beispiel. Sie sehen hier ein einfaches Formular, über das sich das Ergebnis der Addition zweier Zahlen ermitteln lässt. Die beiden Zahlen können dabei in zwei Textfelder eingegeben werden, die Addition wird durch Betätigen der Schaltfläche ausgelöst und das Ergebnis in das dritte Textfeld hineingeschrieben.

Den Code für dieses Beispiel müssen Sie jetzt noch nicht verstehen, und ich gehe an dieser Stelle auch noch gar nicht auf die Details ein. Für den Moment müssen Sie sich nur merken, dass man bei der Webentwicklung mit JavaScript relativ häufig auf HTML-Komponenten zurückgreifen muss, um Ausgaben eines Programms an den Nutzer zu »senden«.

```
// scripts/main.js
function calculateSum() {
  const x = parseInt(document.getElementById('field1').value);
  const y = parseInt(document.getElementById('field2').value);
  const result = document.getElementById('result');
  console.log(x + y);
  result.value = x + y;
}
```

Listing 2.10 Der JavaScript-Code der Datei »main.js«

```
<!DOCTYPE html>
<html>
<head lang="de">
  <meta charset="UTF-8">
  <title>Beispiel</title>
  <link rel="stylesheet" href="styles/main.css" type="text/css">
</head>
<body>
<div class="container">
  <div class="row">
    <label for="field1">X</label> <input id="field1" type="text" value="5">
  </div>
  <div class="row">
    <label for="field2">Y</label> <input id="field2" type="text" value="5">
  </div>
```



```
  <div class="row">
    <label for="result">Ergebnis: </label> <input id="result" type="text">
    <button onclick="calculateSum()">Summe berechnen</button>
  </div>
</div>
<script src="scripts/main.js"></script>
</body>
</html>
```

Listing 2.11 Der HTML-Code für die Beispielanwendung

Abbildung 2.33 Beispielanwendung

DOM-Manipulation

Am komplexesten wird es, wenn Sie eine Webseite dynamisch ändern, um eine Ausgabe zu erzeugen, beispielsweise dynamisch eine Tabelle, um tabellarisch strukturierte Daten darzustellen. Dieses Thema der sogenannten DOM-Manipulation werden wir noch detailliert in Kapitel 5, »Webseiten dynamisch verändern«, besprechen.

2.4 Zusammenfassung

In diesem Kapitel haben Sie gelernt, wie Sie JavaScript-Dateien erzeugen und in HTML einbinden. Sie besitzen nun die Grundlage dafür, die Beispiele aus den nächsten Kapiteln ausführen zu können. Die wichtigsten Punkte aus diesem Kapitel sind:

- Für die Frontend-Entwicklung sind drei Sprachen wichtig: HTML als *Auszeichnungssprache*, um die Struktur einer Webseite festzulegen, CSS als *Stilsprache*, um Design und Layout zu definieren, und JavaScript als *Programmiersprache*, um einer Webseite zusätzliches Verhalten und Interaktivität hinzuzufügen.
- Sie können JavaScript entweder direkt innerhalb des `<script>`-Elements angeben oder über das `src`-Attribut des `<script>`-Elements eine separate JavaScript-Datei einbinden. Ich empfehle Ihnen Letzteres, da so eine saubere Trennung zwischen Struktur (HTML) und Verhalten (JavaScript) der Webseite sichergestellt ist.
- Sie sollten `<script>`-Elemente immer vor dem schließenden `</body>`-Tag platzieren, da so sichergestellt ist, dass der Inhalt der Webseite vollständig geladen ist.

- JavaScript bietet von Haus aus drei Funktionen für das Erzeugen einer Ausgabe: `alert()` für das Erstellen von Hinweisdialogen, `confirm()` für das Erzeugen von Bestätigungsdialogen und `prompt()` für das Erzeugen von Eingabedialogen.
- In der Praxis verwendet man aber statt dieser (mehr oder weniger veralteten) Funktionen schickere Dialoge, wie sie beispielsweise die Bibliothek jQuery anbietet.
- Darüber hinaus bieten alle aktuellen Browser über eine Konsole die Möglichkeit, Ausgaben zu erzeugen, die eher für Sie als Entwickler gedacht sind.

Kapitel 5

Webseiten dynamisch verändern

Bisher haben wir den Browser mehr als Mittel zum Zweck eingesetzt, nämlich für die Ausführung relativ einfacher Beispiele. Seine volle Geltung erreicht die Sprache innerhalb des Browsers allerdings erst, wenn man mit ihr eine dynamische Webanwendung erstellt. Eine wichtige Grundlage hierbei ist das sogenannte Document Object Model, das den Aufbau einer Webseite in Form einer Baumstruktur verwaltet und mithilfe von JavaScript dynamisch verändert werden kann.

Auch wenn einige der bisherigen Beispiele bereits dynamisch Inhalte innerhalb einer HTML-Seite erzeugt haben, müssen wir uns dieses Thema noch etwas genauer anschauen.

5.1 Aufbau einer Webseite

Sie wissen ja schon, dass man bei der objektorientierten Programmierung versucht, Objekte aus der realen Welt bei der Modellierung von Programmen ebenfalls als Objekte zu beschreiben. Auch eine Webseite (bei der man sich streiten kann, ob sie zur realen Welt gehört) wird intern im Browser als Objekt repräsentiert.

5.1.1 Document Object Model

Jedes Mal, wenn Sie eine Webseite aufrufen, erstellt der Browser im Arbeitsspeicher ein entsprechendes Modell der Webseite, das als sogenanntes *Document Object Model* oder kurz *DOM* bezeichnet wird. Das DOM dient in erster Linie dazu, per JavaScript auf Inhalte der Webseite zugreifen zu können, beispielsweise um bestehende Inhalte zu verändern oder neue Inhalte hinzuzufügen. Es stellt die Komponenten einer Webseite hierarchisch in einer *Baumdarstellung* dar, die auch als *DOM-Baum* bezeichnet wird. Ein DOM-Baum wiederum setzt sich aus sogenannten *Knoten* (engl.: *Nodes*) zusammen, die durch ihre hierarchische Anordnung den Aufbau einer Webseite widerspiegeln (siehe Abbildung 5.1).

Hintergrundinfo

Die *Baumdarstellung* ist eine in der Informatik und Programmierung häufig verwendete *Datenstruktur*, die insbesondere dann zum Einsatz kommt, wenn Teile-Ganzes-Beziehungen

repräsentiert werden sollen. Im Fall des DOM steht das Ausgangselement (die Wurzel) ganz oben, und der Baum »wächst« von dort nach unten.

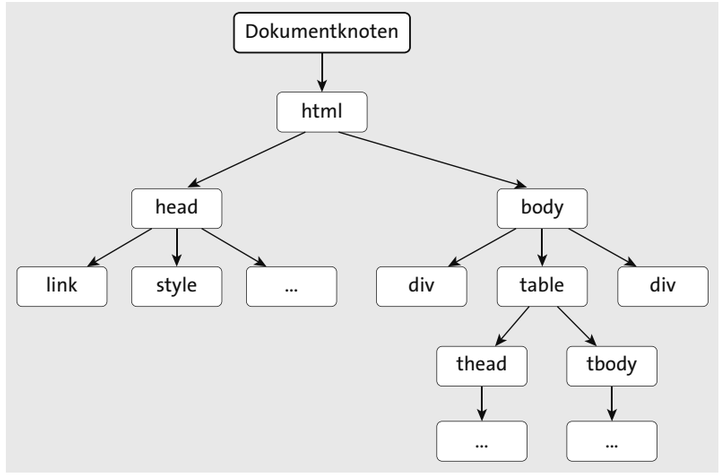

Abbildung 5.1 Aufbau eines DOM-Baums

5.1.2 Die verschiedenen Knotentypen

Insgesamt gibt es vier wesentliche Typen von Knoten (es gibt noch einige mehr, insgesamt zwölf, um genau zu sein, wobei acht davon aber für den Anfang weniger relevant sind), die sich am besten anhand eines Beispiels erläutern lassen. Listing 5.1 zeigt dazu eine Beispiel-HTML-Datei, in der Sie den HTML-Code für eine einfache Tabelle zur Darstellung einer Kontaktliste sehen. Das entsprechende Document Object Model zeigt Abbildung 5.2 (wobei ich aus Platzgründen und der Übersicht wegen auf eine vollständige Abbildung verzichtet habe).

```
<!DOCTYPE html>
<html>
  <head lang="de">
    <title>Kontaktlistenbeispiel</title>
  </head>
  <body>
    <main id="main">
      <h1>Kontaktliste</h1>
      <table id="contact-list-table" summary="Kontaktliste">
        <thead>
          <tr>
            <th id="table-header-first-name">Vorname</th>
```



```
            <th id="table-header-last-name">Nachname</th>
            <th id="table-header-email">E-Mail-Adresse</th>
          </tr>
        </thead>
        <tbody>
          <tr class="row odd">
            <td>Max</td>
            <td>Mustermann</td>
            <td>max.mustermann@javascripthandbuch.de</td>
          </tr>
          <tr class="row even">
            <td>Moritz</td>
            <td>Mustermann</td>
            <td>moritz.mustermann@javascripthandbuch.de</td>
          </tr>
          <tr class="row odd">
            <td>Peter</td>
            <td>Mustermann</td>
            <td>peter.mustermann@javascripthandbuch.de</td>
          </tr>
          <tr class="row even">
            <td>Paul</td>
            <td>Mustermann</td>
            <td>paul.mustermann@javascripthandbuch.de</td>
          </tr>
        </tbody>
      </table>
    </main>
  </body>
</html>
```

Listing 5.1 Beispiel-HTML-Seite

Folgende vier Knotentypen werden Sie bei der Arbeit mit dem DOM am häufigsten verwenden:

- Der *Dokumentknoten* (in Abbildung 5.2 fett umrandet) steht für die gesamte Webseite und bildet die Wurzel des DOM-Baums. Er wird durch das globale Objekt `document` repräsentiert, das Sie ja schon in einigen Listings sehen konnten. Dieses Objekt ist gleichzeitig das Einstiegsobjekt für jegliche Arbeiten mit dem DOM. Der Dokumentknoten wird auch als *Wurzelknoten* bezeichnet.
- *Elementknoten* (in Abbildung 5.2 mit weißem Hintergrund) repräsentieren einzelne HTML-Elemente einer Webseite. Im Beispiel sind dies beispielsweise die Elemente `<main>`, `<h1>`, `<table>`, `<thead>` und `<tbody>`.

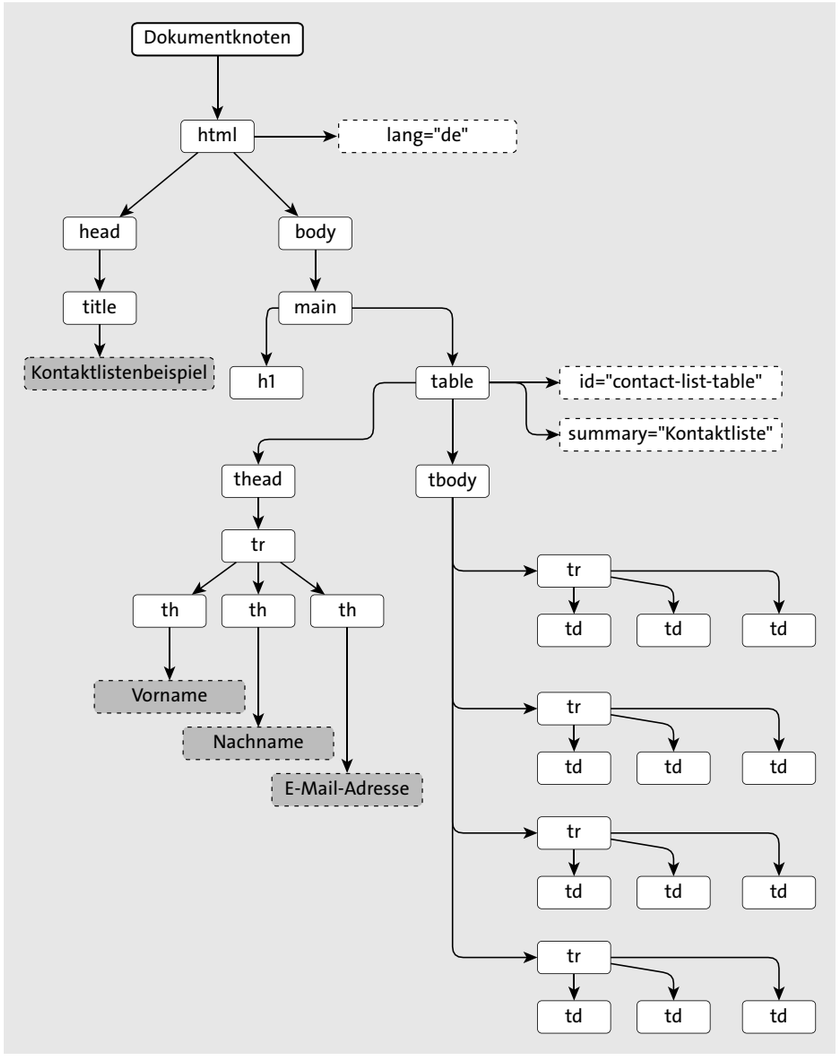

Abbildung 5.2 Aufbau des DOM-Baums für das Beispiel

- *Attributknoten* (in Abbildung 5.2 gestrichelt umrandet und mit weißem Hintergrund) stehen für Attribute von HTML-Elementen, im Beispiel die Attributknoten für die Attribute `lang`, `id` und `summary`.
- Der Text innerhalb von HTML-Elementen wird durch einen eigenen Knotentyp repräsentiert, den sogenannten *Textknoten* (in Abbildung 5.2 gestrichelt umrandet und grau eingefärbt). Im Beispiel sind das etwa die Knoten für die Texte `Kontaktlistenbeispiel`,

`Kontaktliste`, `Vorname`, `Nachname` und `E-Mail-Adresse`. Textknoten können selbst keine Kindknoten haben und sind damit zwangsläufig Blätter in dem DOM-Baum (im Beispiel sind aus genannten Platzgründen nicht alle Textknoten abgebildet).

Hinweis

Das Beispiel aus Listing 5.1 und Abbildung 5.2 bildet die Grundlage für die nächsten Abschnitte. Anhand dieses Beispiels werde ich Ihnen im Folgenden zeigen, wie Sie auf Knoten einer Webseite zugreifen und diese verändern können.

Das DOM im Browser untersuchen

Das DOM einer Webseite können Sie mit den jeweiligen JavaScript-Debugging-Tools der verschiedenen Browser in einer speziellen Ansicht einsehen. In den Chrome Developer Tools befindet sich diese Ansicht hinter der Registerkarte Elements (siehe Abbildung 5.3).

Sie können über diese Ansicht in der Regel das DOM sogar händisch ändern. Testen Sie das einmal, indem Sie innerhalb des DOM-Baums auf einen der Knoten, beispielsweise auf einen Textknoten, doppelt klicken. Anschließend können Sie den entsprechenden Text des Knotens ändern.

In der Praxis kann das recht hilfreich sein, um eben mal schnell eine gewisse Konstellation von HTML zu testen. Die Änderungen, die Sie in dieser Ansicht vornehmen, haben allerdings keine Auswirkung auf die unterliegende HTML-Datei. Wenn Sie die Datei im Browser neu laden, sind die Änderungen verloren.

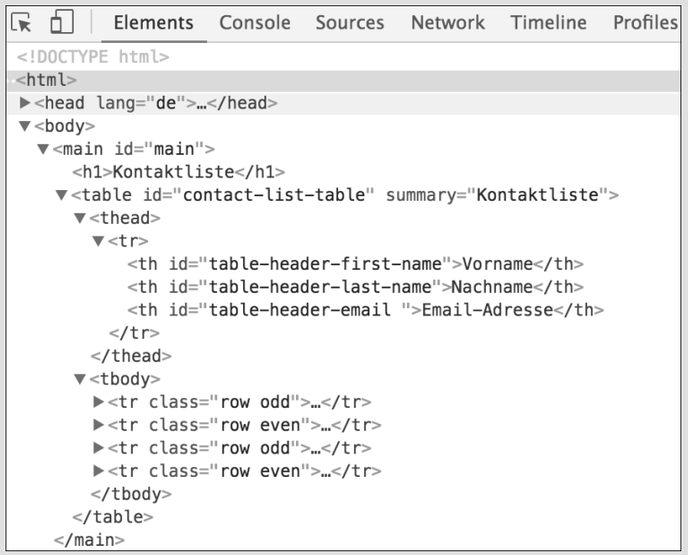

Abbildung 5.3 Darstellung des DOM in den Chrome Developer Tools

5.1.3 Der Dokumentknoten

Der Dokumentknoten stellt, wie bereits erwähnt, den Einstiegspunkt für das DOM dar und wird über das globale Objekt `document` repräsentiert, das über verschiedene Eigenschaften und Methoden verfügt.

Ausgewählte Eigenschaften sind in Tabelle 5.1 aufgelistet, auf die verschiedenen Methoden werden wir dagegen im Laufe des Kapitels im Detail eingehen.

Eigenschaft	Beschreibung
`document.title`	Enthält den Titel des aktuellen Dokuments.
`document.lastModified`	Enthält das Datum, an dem das Dokument zuletzt geändert wurde.
`document.URL`	Enthält eine URL des aktuellen Dokuments.
`document.domain`	Enthält die Domäne des aktuellen Dokuments.
`document.cookie`	Enthält eine Liste aller Cookies für das Dokument.
`document.forms`	Enthält eine Liste aller Formulare des Dokuments.
`document.images`	Enthält eine Liste aller Bilder des Dokuments.
`document.links`	Enthält eine Liste aller Links des Dokuments.

Tabelle 5.1 Ausgewählte Eigenschaften des »document«-Objekts

DOM unter Node.js

Das Document Object Model in Form der globalen `document`-Variablen steht nur in browserbasierten Laufzeitumgebungen zur Verfügung. In Node.js beispielsweise (siehe Kapitel 17, »Serverseitige Anwendungen mit Node.js erstellen«) gibt es eine solche globale Variable nicht, da Node.js in der Regel nicht dazu verwendet wird, Webseiten zu rendern. Erst über spezielle Module wie z. B. jsdom (*https://github.com/jsdom/jsdom*), mit denen man Webseiten parsen kann, lässt sich unter Node.js ein Document Object Model einer Webseite erstellen.

Der Aufbau des Document Object Model – sprich, welche Eigenschaften und Methoden zur Verfügung stehen, welche Knotentypen es gibt etc. – ist in der sogenannten *DOM API*, einer Spezifikation des *W3C* (*World Wide Web Consortium*), festgehalten. Diese API (*Application Programming Interface*) ist programmiersprachenunabhängig gehalten, d. h., es gibt nicht nur Implementierungen für JavaScript, sondern auch für andere Programmiersprachen wie Java oder C++.

Interface, Implementierung und API

In der objektorientierten Programmierung dienen *Interfaces* (auch *Schnittstellen* genannt) dazu, die Methoden zu definieren, die in *Implementierungen* (also konkreten Umsetzungen

des jeweiligen Interface) vorhanden sein müssen. Ein *Application Programming Interface* (kurz: *API*) definiert eine Menge von Interfaces, die von einem Softwaresystem zur Verfügung gestellt werden.

Die DOM API ist demnach eine Menge von Interfaces, die Browser für die Arbeit mit Webseiten zur Verfügung stellen.

Die API vs. das API

Die grammatisch korrekte Bezeichnung lautet *das API* (denn es ist ja die Abkürzung für *das Application Programming Interface*). Es ist aber durchaus üblich, den Artikel der deutschen Übersetzung – *die Programmierschnittstelle* – zu wählen, wonach es dann *die API* heißt.

5.2 Elemente selektieren

Egal ob Sie bestehende Informationen einer Webseite ändern wollen oder neue Informationen hinzufügen möchten: In beiden Fällen müssen Sie zunächst ein Element auf der Webseite *selektieren*, sprich auswählen, das Sie ändern bzw. an das Sie die neuen Informationen anfügen möchten. Dazu bietet die DOM API verschiedene Eigenschaften und Methoden an, von denen Tabelle 5.2 Ihnen eine Übersicht zeigt.

Wie Sie sehen, gibt es einige Methoden, die mehrere Elemente zurückgeben, und einige Methoden, die einzelne Elemente zurückgeben. Die Details schauen wir uns in den folgenden Abschnitten an.

Eigenschaft/Methode	Beschreibung	Rückgabewert	Abschnitt
`getElementById()`	Wählt ein Element anhand einer ID aus.	einzelnes Element	Abschnitt 5.2.1, »Elemente per ID selektieren«
`getElementsByClassName()`	Wählt Elemente anhand eines Klassennamens aus.	Liste von Elementen	Abschnitt 5.2.2, »Elemente per Klasse selektieren«
`getElementsByTagName()`	Wählt alle Elemente mit dem angegebenen Elementnamen aus.	Liste von Elementen	Abschnitt 5.2.3, »Elemente nach Elementnamen selektieren«
`getElementsByName()`	Wählt Elemente anhand ihres Namens aus.	Liste von Elementen	Abschnitt 5.2.4, »Elemente nach Namen selektieren«

Tabelle 5.2 Die verschiedenen Methoden und Eigenschaften für das Auswählen von Elementen

Eigenschaft/Methode	Beschreibung	Rückgabewert	Abschnitt
`querySelector()`	Gibt das erste Element zurück, das auf einen gegebenen CSS-Selektor passt.	einzelnes Element	Abschnitt 5.2.5, »Elemente per Selektor selektieren«
`querySelectorAll()`	Gibt alle Elemente zurück, die auf einen gegebenen CSS-Selektor passen.	Liste von Elementen	Abschnitt 5.2.5, »Elemente per Selektor selektieren«
`parentElement`	Gibt für einen Knoten das Elternelement zurück.	einzelnes Element	Abschnitt 5.2.6, »Das Elternelement eines Elements selektieren«
`parentNode`	Gibt für einen Knoten den Elternknoten zurück.	einzelner Knoten	Abschnitt 5.2.6, »Das Elternelement eines Elements selektieren«
`previousElementSibling`	Gibt für einen Knoten das vorhergehende Geschwisterelement zurück.	einzelnes Element	Abschnitt 5.2.8, »Die Geschwisterelemente eines Elements selektieren«
`previousSibling`	Gibt für einen Knoten den vorhergehenden Geschwisterknoten zurück.	einzelner Knoten	Abschnitt 5.2.8, »Die Geschwisterelemente eines Elements selektieren«
`nextElementSibling`	Gibt für einen Knoten das nachfolgende Geschwisterelement zurück.	einzelnes Element	Abschnitt 5.2.8, »Die Geschwisterelemente eines Elements selektieren«
`nextSibling`	Gibt für einen Knoten den nachfolgenden Geschwisterknoten zurück.	einzelner Knoten	Abschnitt 5.2.8, »Die Geschwisterelemente eines Elements selektieren«
`firstElementChild`	Gibt für einen Knoten das erste Kindelement zurück.	einzelnes Element	Abschnitt 5.2.7, »Die Kindelemente eines Elements selektieren«
`firstChild`	Gibt für einen Knoten den ersten Kindknoten zurück.	einzelner Knoten	Abschnitt 5.2.7, »Die Kindelemente eines Elements selektieren«

Tabelle 5.2 Die verschiedenen Methoden und Eigenschaften für das Auswählen von Elementen (Forts.)

Eigenschaft/Methode	Beschreibung	Rückgabewert	Abschnitt
`lastElementChild`	Gibt für einen Knoten das letzte Kindelement zurück.	einzelnes Element	Abschnitt 5.2.7, »Die Kindelemente eines Elements selektieren«
`lastChild`	Gibt für einen Knoten den letzten Kindknoten zurück.	einzelner Knoten	Abschnitt 5.2.7, »Die Kindelemente eines Elements selektieren«
`childNodes`	Gibt für einen Knoten alle Kindknoten zurück.	Liste von Knoten	Abschnitt 5.2.7, »Die Kindelemente eines Elements selektieren«
`children`	Gibt für einen Knoten alle Kindelemente zurück.	Liste von Elementen	Abschnitt 5.2.7, »Die Kindelemente eines Elements selektieren«

Tabelle 5.2 Die verschiedenen Methoden und Eigenschaften für das Auswählen von Elementen (Forts.)

Selektionsmethoden

Selektionsmethoden und die Eigenschaften stehen nicht nur für den Dokumentknoten zur Verfügung, sondern auch für andere Knoten (siehe Abschnitt 5.2.9, »Selektionsmethoden auf Elementen aufrufen«).

5.2.1 Elemente per ID selektieren

Elementen auf einer Webseite kann über das `id`-Attribut eine (auf der jeweiligen Webseite eindeutige) ID zugewiesen werden. Diese ID kann zum einen in CSS-Regeln verwendet werden, zum anderen können Sie per JavaScript über die Methode `getElementById()` des Objekts `document` das entsprechende Element auswählen. Sie übergeben der Methode lediglich die ID des Elements, das selektiert werden soll, in Form einer Zeichenkette.

In Listing 5.2 wird das Element mit der ID `main` ausgewählt (siehe auch Abbildung 5.4) und in der Variablen `mainElement` gespeichert. Anschließend wird das `class`-Attribut des Elements über die Eigenschaft `className` auf den Wert `border` geändert, was im Beispiel zur Folge hat, dass das Element einen roten Rahmen mit abgerundeten Ecken erhält (siehe Abbildung 5.5, das vollständige Beispiel inklusive HTML- und CSS-Code finden Sie wie immer im Downloadbereich zum Buch).

```
const mainElement = document.getElementById('main');
mainElement.className = 'border';
```

Listing 5.2 Zugriff auf ein Element über die ID

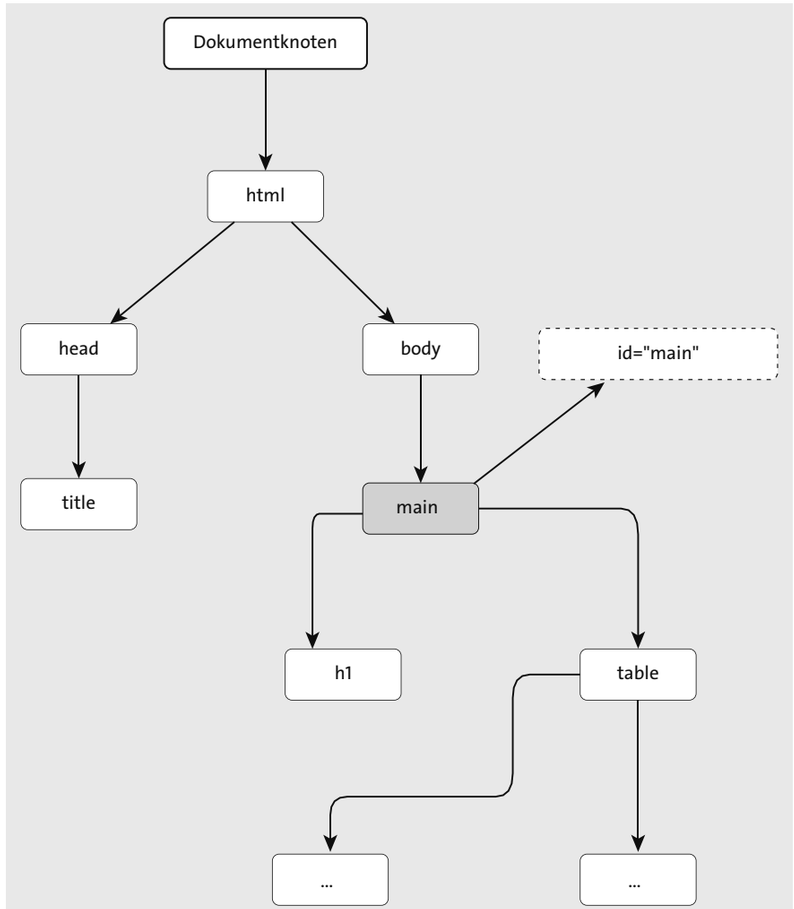

Abbildung 5.4 Mit »getElementById()« wird maximal ein Element selektiert.

Kontaktliste

Vorname	Nachname	E-Mail-Adresse
Max	Mustermann	max.mustermann@javascripthandbuch.de
Moritz	Mustermann	moritz.mustermann@javascripthandbuch.de
Peter	Mustermann	peter.mustermann@javascripthandbuch.de
Paul	Mustermann	paul.mustermann@javascripthandbuch.de

Abbildung 5.5 Dem zurückgegebenen Element wird eine neue CSS-Klasse zugewiesen, wodurch das Element einen hervorgehobenen Rahmen bekommt.

Tipp

In der Praxis ist es nicht schlecht, etwas *defensiver* zu programmieren und zu testen, ob eine Variable, auf die zugegriffen werden soll, nicht `null` oder `undefined` ist. Das gilt auch für das Arbeiten mit dem Document Object Model. Die Methode `getElementById()` gibt nämlich den Wert `null` zurück, falls kein Element mit der übergebenen ID gefunden wurde. Wenn Sie dann versuchen, auf eine Eigenschaft oder Methode auf dem vermeintlichen Element zuzugreifen, kommt es zu einem Laufzeitfehler. Um dem vorzubeugen, sollten Sie, wie in Listing 5.3 gezeigt, über eine `if`-Abfrage sicherstellen, dass der Rückgabewert von `getElementById()` nicht `null` ist.

```
const mainElement = document.getElementById('main'); // Wähle Element mit ID aus.
if(mainElement !== null) {                           // Falls Element nicht
                                                     // leer ist,
  mainElement.className = 'border';                  // weise neue CSS-Klasse zu.
}
```

Listing 5.3 Sicher ist sicher: Für den Fall, dass es kein Element mit der ID »main« gibt (im Beispiel-HTML oben ist das nicht der Fall), wird nicht auf die Variable zugegriffen.

Alternativ dazu können Sie das Ganze auch verkürzen und sich die Tatsache zunutze machen, dass beim `&&`-Operator der zweite Operand nur dann ausgewertet wird, wenn der erste Operand ein `true` zurückgibt. So wird in folgendem Listing der Operand (`mainElement.className = 'border'`) nur dann ausgewertet (bzw. ausgeführt), wenn `mainElement` einen Wert hat, der nicht zu `false` evaluiert – mit anderen Worten: nicht `null` ist.

```
const mainElement = document.getElementById('main'); // Wähle Element mit ID aus.
mainElement && (mainElement.className = 'border');
```

Listing 5.4 Über den &&-Operator können Sie die Überprüfung von oben verkürzen.

Und noch kürzer geht die Überprüfung mit dem *Optionale-Verkettung-Operator* (*Optional Chaining Operator*) `?`, der aber erst seit ES2020 zur Verfügung steht. In folgendem Listing wird auf die Eigenschaft `className` nur dann zugegriffen, wenn `mainElement` nicht `null` oder `undefined` ist:

```
const mainElement = document.getElementById('main'); // Wähle Element mit ID aus.
mainElement?.className = 'border';
```

Listing 5.5 Über den ?-Operator können Sie prüfen, ob eine Eigenschaft definiert ist.

Performance von Selektionsmethoden

Die Auswahl eines Elements per ID ist hinsichtlich der Performance im Vergleich zu anderen Selektionsmethoden recht schnell, da es auf einer Webseite nicht erlaubt ist, mehrere Elemente mit der gleichen ID zu haben, und somit die Suche sehr schnell das entsprechende Element für eine ID finden kann. Andere Selektionsmethoden wie beispielsweise die im

nächsten Abschnitt vorgestellte Methode `getElementsByClassName()` sind im Vergleich deutlich langsamer, weil hierbei jedes Element auf der Webseite überprüft werden muss. Auch wenn Sie den Geschwindigkeitsunterschied in der Regel nicht merken werden, sollten Sie ihn doch im Hinterkopf behalten.

Tipp

Bei der Verwendung von DOM-Methoden sollten Sie nicht zu verschwenderisch umgehen. Wenn Sie innerhalb eines Programms das Ergebnis einer DOM-Methode an mehreren Stellen verwenden müssen, speichern Sie das Ergebnis in einer Variablen, anstatt immer wieder die DOM-Methode aufzurufen. Bedenken Sie: Jeder Aufruf einer DOM-Methode, bei der nach Elementen im DOM-Baum gesucht wird, kostet Rechenzeit. Über Variablen, in denen Sie Ergebnisse zwischenspeichern, lässt sich diese Rechenzeit minimieren.

5.2.2 Elemente per Klasse selektieren

Ähnlich wie für IDs können auf einer Webseite einzelnen Elementen *CSS-Klassen* zugeordnet werden. Verwaltet werden diese Klassen über das `class`-Attribut. Ein Element kann dabei mehrere Klassen haben, und im Unterschied zu IDs können auch mehrere Elemente die gleiche Klasse haben.

Dies wiederum hat zur Folge, dass die entsprechende DOM-Methode `getElementsByClassName()` – mit der eine Selektion nach CSS-Klassen möglich ist – nicht nur ein einzelnes Element zurückgibt, sondern gegebenenfalls auch mehrere Elemente.

Als Argument übergibt man der Methode den Klassennamen als Zeichenkette, wie in Listing 5.6 zu sehen. In diesem Beispiel werden alle Elemente selektiert, die die CSS-Klasse `even` enthalten, sprich die beiden »geraden« Tabellenzeilen (siehe Abbildung 5.6).

```
const tableRowsEven = document
    .getElementsByClassName('even');      // Selektiere alle geraden Tabellenzeilen.
```

Listing 5.6 Zugriff auf ein Element über Klassennamen

Der Rückgabewert von `getElementsByClassName()` ist eine *Knotenliste* (genauer gesagt, ein Objekt vom Typ `NodeList`), die ähnlich wie ein Array zu verwenden ist (bei der es sich aber nicht um ein Array handelt, dazu gleich mehr). Diese Knotenliste enthält die Elemente in genau der Reihenfolge, wie sie auf der Webseite auftreten.

Auch wenn Knotenlisten auf den ersten Blick wie Arrays aussehen, sind es keine Arrays. Eine Tatsache, die man sich als JavaScript-Einsteiger immer wieder bewusst machen muss und deren Nichtbeachtung nicht selten zu Fehlern im Programm führt.

Mit Arrays gemeinsam haben Knotenlisten, dass man auf die einzelnen Elemente in einer Knotenliste über einen Index zugreifen kann, d. h., über `tableRowsEven[0]` greift man beispielsweise auf das erste Element zu, über `tableRowsEven[1]` auf das zweite Element und so weiter. Ebenfalls eine Gemeinsamkeit ist die Eigenschaft `length`, über die sich die Anzahl an Elementen in der Knotenliste herausfinden lässt.

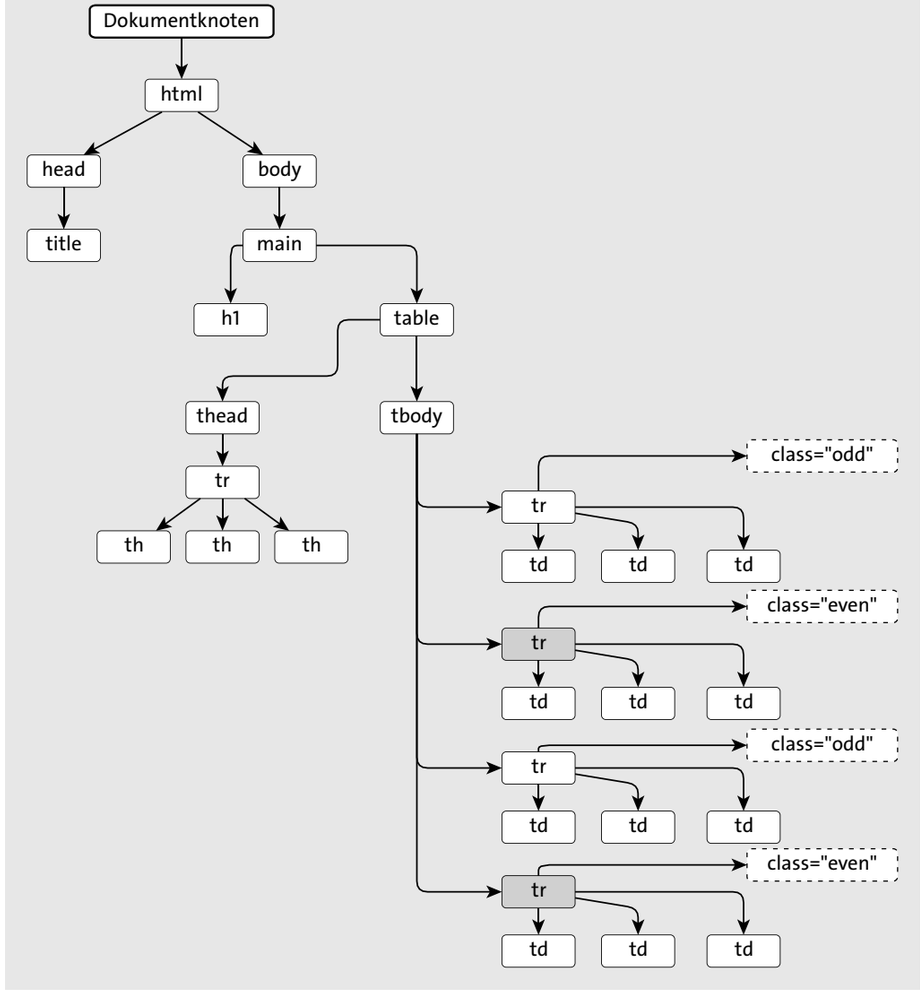

Abbildung 5.6 Die Methode »getElementsByClassName()« kann mehrere Elemente zurückgeben.

Um also beispielsweise über alle Elemente einer Knotenliste zu iterieren, geht man wie in Listing 5.7 vor. Hier wird mithilfe einer `for`-Schleife über alle Elemente der Liste iteriert. Wie

bei der Iteration über echte Arrays können Sie dabei die Eigenschaft `length` und den Zugriff per Index verwenden. Im Beispiel wird auf diese Weise jedem Element in der Liste eine neue Hintergrundfarbe zugewiesen (siehe Abbildung 5.7).

```
const tableRowsEven = document
  .getElementsByClassName('even');              // Selektiere alle geraden
                                                // Tabellenzeilen.
if(tableRowsEven.length > 0) {                  // Wenn mindestens ein Element
                                                // gefunden wurde.
  for(let i=0; i<tableRowsEven.length; i++) {   // Gehe alle Elemente durch.
    const tableRow = tableRowsEven[i];          // Weise Element einer Variablen zu.
    tableRow.style.backgroundColor = '#CCCCCC'; // Setze neue Hintergrundfarbe.
  }
}
```

Listing 5.7 Iteration über eine Knotenliste unter Verwendung der Array-Syntax

Kontaktliste

Vorname	Nachname	E-Mail-Adresse
Max	Mustermann	max.mustermann@javascripthandbuch.de
Moritz	Mustermann	moritz.mustermann@javascripthandbuch.de
Peter	Mustermann	peter.mustermann@javascripthandbuch.de
Paul	Mustermann	paul.mustermann@javascripthandbuch.de

Abbildung 5.7 Den geraden Tabellenzellen wurde per JavaScript eine andere Hintergrundfarbe zugewiesen.

Das CSS eines Elements verändern

Über die Eigenschaft `style` eines Elements können Sie an die CSS-Eigenschaften eines Elements gelangen bzw. diese auch verändern. Das in dieser Eigenschaft hinterlegte Objekt enthält alle CSS-Eigenschaften als Objekteigenschaften (also beispielsweise `style.color`, `style.border` etc.). Für CSS-Eigenschaften wie beispielsweise `background-color`, die einen Bindestrich enthalten, sind die entsprechenden Objekteigenschaften in CamelCase-Schreibweise definiert (beispielsweise `style.backgroundColor` oder `style.fontFamily`).

Alternativ zu der »Array-Syntax« mit eckigen Klammern lässt sich über die Methode `item()` auf einzelne Knoten einer Knotenliste zugreifen. Hier übergeben Sie ebenfalls als Argument den Index des Elements, das zurückgegeben werden soll. Die Schleife von eben ließe sich also auch wie folgt umformulieren:


```
const tableRowsEven = document
  .getElementsByClassName('even');              // Selektiere alle geraden
                                                // Tabellenzeilen.
if(tableRowsEven.length > 0) {                  // Wenn mindestens ein Element
                                                // gefunden wurde.
  for(let i=0; i<tableRowsEven.length; i++) {   // Gehe alle Elemente durch.
    const tableRow = tableRowsEven.item(i);     // Weise Element einer Variablen zu.
    tableRow.style.backgroundColor = '#CCCCCC'; // Setze neue Hintergrundfarbe.
  }
}
```

Listing 5.8 Iteration über eine Knotenliste unter Verwendung der Methode »item()«

Method Borrowing

Da es sich bei Knotenlisten um keine echten Arrays (sondern um Objekte vom Typ `NodeList`), wohl aber um Array-ähnliche Objekte handelt (wie das `arguments`-Objekt, Sie erinnern sich?), verwendet man in der Praxis häufig auch die Technik des *Method Borrowing* (siehe Kapitel 4, »Mit Referenztypen arbeiten«), um dennoch Methoden von `Array` verwenden zu können (siehe Listing 5.9).

```
Array.prototype.forEach.call(tableRowsEven, (tableRow) => {
  tableRow.style.backgroundColor = '#CCCCCC';
});
```

Listing 5.9 Iteration über eine Knotenliste über Method Borrowing

Aktive Knotenlisten vs. statische Knotenlisten

Man unterscheidet bei Knotenlisten zwischen den *aktiven* und den *statischen Knotenlisten*. Erstere bezeichnen Knotenlisten, bei denen Änderungen, die an einzelnen Knoten in der Liste vorgenommen werden, direkte Auswirkungen auf die Webseite haben, d. h., dass die Änderungen direkt in der Webseite widergespiegelt werden.

Bei Letzteren dagegen haben Änderungen an Knoten innerhalb der Knotenliste keine direkten Auswirkungen auf die Webseite, werden also nicht direkt in der Webseite widergespiegelt. Die Methoden `getElementsByClassName()`, `getElementsByTagName()` und `getElementsByName()` geben aktive Knotenlisten zurück, die Methode `querySelectorAll()` dagegen eine statische Knotenliste.

5.2.3 Elemente nach Elementnamen selektieren

Über die Methode `getElementsByTagName()` lassen sich Elemente anhand ihres Elementnamens selektieren. Die Methode erwartet dabei den Namen des Elements. Um beispielsweise alle Tabellenzellen zu selektieren (siehe Abbildung 5.8), gehen Sie wie in Listing 5.10 vor.

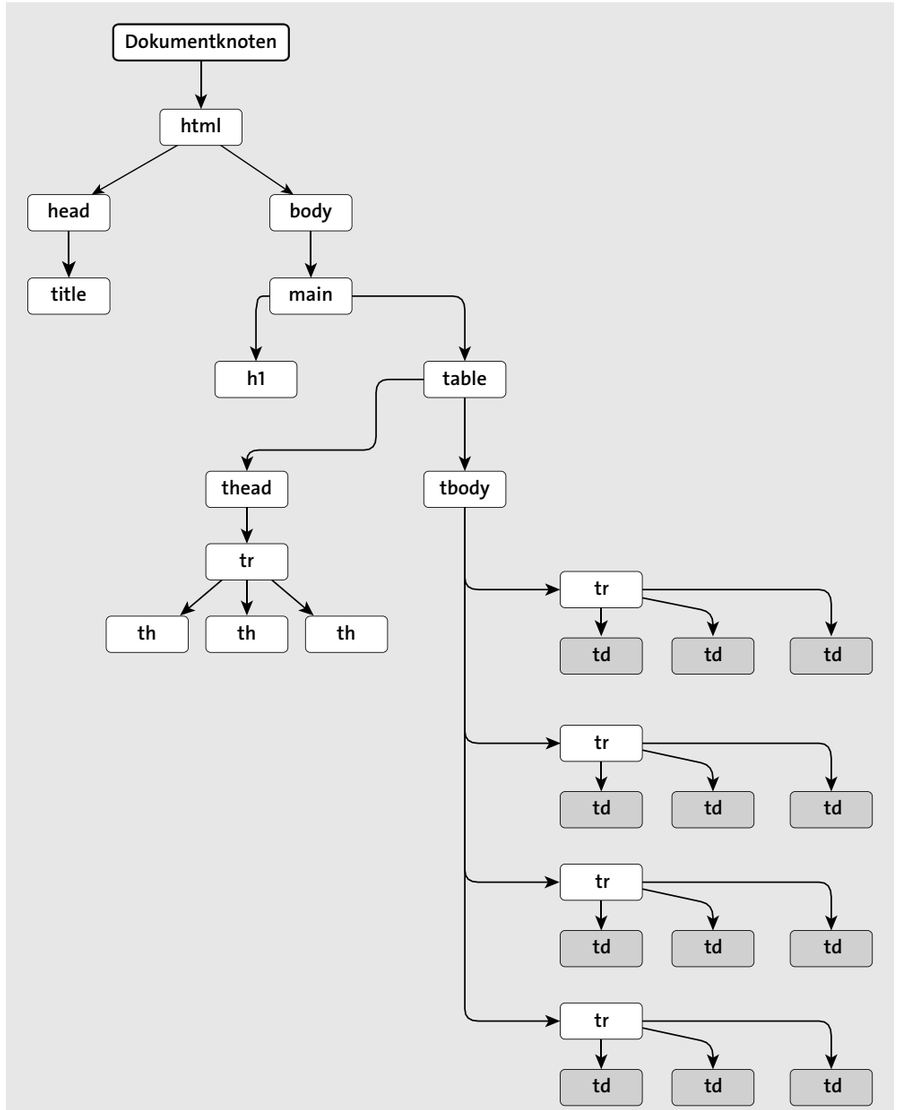

Abbildung 5.8 Die Methode »getElementsByTagName()« selektiert Elemente nach ihrem Elementnamen.

Hier werden zunächst über die Methode `getElementsByTagName()` alle Tabellenzellen ausgewählt, und anschließend wird jedem Element eine neue Schriftart sowie eine neue Schriftgröße zugewiesen. Das Ergebnis sehen Sie in Abbildung 5.9.


```
const tableCells = document.getElementsByTagName('td');
if(tableCells.length > 0) {                  // Wenn mindestens ein Element gefunden
                                             // wurde.
  for(let i=0; i<tableCells.length; i++) {   // Gehe alle Elemente durch.
    const tableCell = tableCells[i];         // Weise Element einer Variablen zu.
    tableCell.style.fontFamily = 'Verdana';  // Setze neue Schriftart.
    tableCell.style.fontSize = '9pt';        // Setze neue Schriftgröße.
  }
}
```

Listing 5.10 Zugriff auf ein Element über Elementnamen

Kontaktliste

Vorname	Nachname	E-Mail-Adresse
Max	Mustermann	max.mustermann@javascripthandbuch.de
Moritz	Mustermann	moritz.mustermann@javascripthandbuch.de
Peter	Mustermann	peter.mustermann@javascripthandbuch.de
Paul	Mustermann	paul.mustermann@javascripthandbuch.de

Abbildung 5.9 Die Tabellenzellen erhalten eine neue Schriftart und Schriftgröße.

Hinweis

Beachten Sie, dass Sie der Methode `getElementsByTagName()` wirklich nur den Namen des Elements übergeben und nicht etwa zusätzliche spitze Klammern. Beispielsweise würde der Aufruf `getElementsByTagName('<td>')` nicht funktionieren.

5.2.4 Elemente nach Namen selektieren

Einigen Elementen kann in HTML ein `name`-Attribut zugewiesen werden, beispielsweise `<input>`-Elementen vom Typ `radio`, um deren Zusammengehörigkeit mit einer Auswahlgruppe zu kennzeichnen. In Listing 5.11 beispielsweise werden darüber die drei Radiobuttons der Gruppe `genre` zugewiesen.

```
<form action="">
  <label for="artist">K&uuml;nstler</label>
  <input id="artist" type="text" name="artist">
  <br>
  <label for="album">Album</label>
  <input id="album" type="text" name="album">
  <br>
  <p>Genre:</p>
```

```
  <fieldset>
    <input type="radio" id="st" name="genre" value="Stonerrock">
    <label for="st">Stonerrock</label>
    <br>
    <input type="radio" id="sp" name="genre" value="Spacerock">
    <label for="sp">Spacerock</label>
    <br>
    <input type="radio" id="ha" name="genre" value="Hardrock">
    <label for="ha">Hardrock</label>
  </fieldset>
</form>
```

Listing 5.11 Ein einfaches HTML-Formular

Mithilfe der Methode `getElementsByName()` können Elemente ausgehend von diesem `name`-Attribut selektiert werden. In Listing 5.12 werden auf diese Weise alle Elemente selektiert, deren `name`-Attribut den Wert `genre` hat (die anderen beiden Formularelemente mit den Werten `artist` und `album` dagegen werden nicht selektiert, siehe Abbildung 5.10).

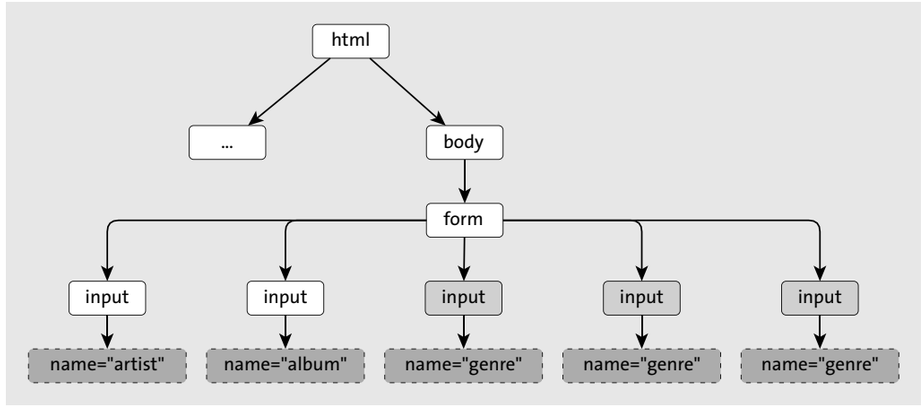

Abbildung 5.10 Die Methode »getElementsByName()« selektiert Elemente nach ihrem »name«-Attribut.

In der anschließenden Schleife werden die Werte dieser Elemente (`inputElement.value`) ausgegeben: `Stonerrock`, `Spacerock` und `Hardrock` (hach, was für ein tolles Beispiel).

```
const inputElementsForGenre = document
  .getElementsByName('genre');                    // Selektiere alle Elemente
                                                  // mit Namen.
if(inputElementsForGenre.length > 0) {            // Wenn mindestens ein
                                                  // Element gefunden wurde.
  for(let i=0; i<inputElementsForGenre.length; i++) {   // Gehe alle Elemente durch.
```



```
    const inputElement = inputElementsForGenre[i];      // Weise Element einer
                                                        // Variablen zu.
    console.log(inputElement.value);                    // Ausgabe: Stonerrock,
                                                        // Spacerock, Hardrock
  }
}
```

Listing 5.12 Zugriff auf Elemente über Elementnamen

Browsersupport von »getElementsByName()«

Die Methode `getElementsByName()` funktioniert nicht in allen Browsern konsistent. In einigen Versionen des Internet Explorer und des Opera-Browsers beispielsweise liefert die Methode nicht nur solche Elemente zurück, deren `name`-Attribut mit dem übergebenen Wert übereinstimmt, sondern auch solche, deren `id`-Attribut mit dem übergebenen Wert übereinstimmt. Meine Meinung ist, dass Sie mit den anderen (bisher vorgestellten und gleich noch vorzustellenden) Selektionsmethoden ausreichende Möglichkeiten zur Selektion von Elementen haben und somit eigentlich auf diese Methode in der Praxis verzichten können.

5.2.5 Elemente per Selektor selektieren

Mit den bisher vorgestellten DOM-Methoden zur Selektion von Elementen lässt sich schon einiges erreichen, allerdings ist man in der Ausdrucksform doch etwas begrenzt. Nicht immer ist es so, dass das Element, das man selektieren möchte, überhaupt eine ID oder Klasse hat, sodass die Methoden `getElementById()` oder `getElementsByClassName()` in solchen Fällen nicht weiterhelfen. Die Methode `getElementsByTagName()` dagegen ist sehr unspezifisch, weil tendenziell eher viele Elemente selektiert werden. Und `getElementsByName()` ist aus genannten Gründen ohnehin mit Vorsicht zu genießen.

Deutlich vielseitiger und ausdrucksstärker sind da schon die Methoden `querySelector()` und `querySelectorAll()`, um Elemente für einen gegebenen CSS-Selektor zurückzugeben. Erstere Methode liefert dabei als Rückgabewert das *erste Element*, das auf den entsprechenden CSS-Selektor zutrifft, letztere Methode liefert dagegen *alle Elemente*, die auf den übergebenen CSS-Selektor zutreffen.

Listing 5.13 zeigt ein Beispiel für die Verwendung von `querySelector()`. Übergeben wird hier der CSS-Selektor `#main table td`, der in CSS zunächst die zweiten Tabellenzellen jeder Zeile (`td:nth-child(2)`) innerhalb einer Tabelle (`table`) innerhalb eines Elements mit der ID `main` (`#main`) beschreibt. Da die Methode `querySelector()` aber nur das erste auf einen Selektor zutreffende Element selektiert, wird nur das erste `<td>`-Element zurückgegeben.

```
const tableCell = document.querySelector('#main table td:nth-child(2)');
tableCell.style.border = 'thick solid red';
```

Listing 5.13 Zugriff auf ein Element über CSS-Selektor

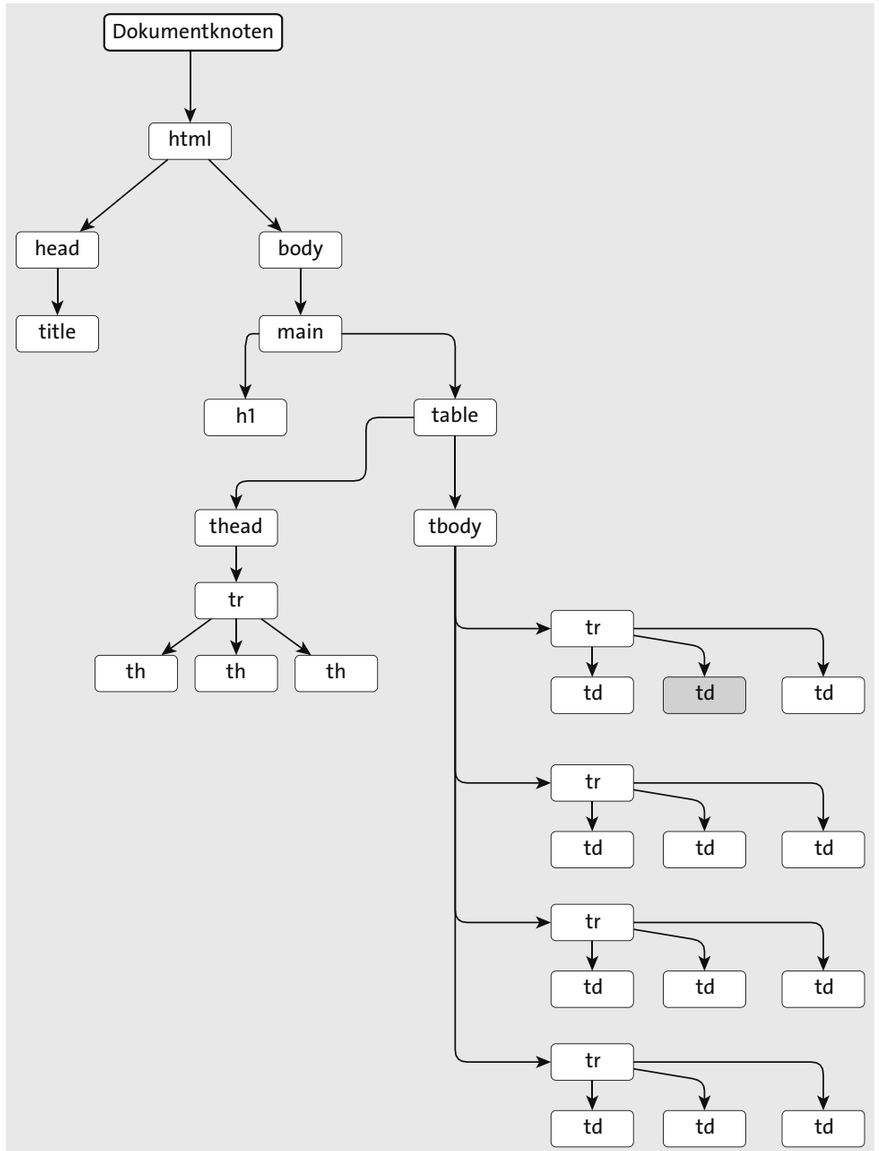

Abbildung 5.11 Die Methode »querySelector()« liefert maximal ein Element zurück.

Listing 5.14 zeigt dagegen die Anwendung der Methode `querySelectorAll()`. Auch hier wird der gleiche CSS-Selektor wie eben verwendet. Diesmal erhält man jedoch *alle* Elemente, die auf diesen Selektor zutreffen, sprich alle zweiten `<td>`-Elemente (siehe Abbildung 5.13).

Kontaktliste

Vorname	Nachname	E-Mail-Adresse
Max	Mustermann	max.mustermann@javascripthandbuch.de
Moritz	Mustermann	moritz.mustermann@javascripthandbuch.de
Peter	Mustermann	peter.mustermann@javascripthandbuch.de
Paul	Mustermann	paul.mustermann@javascripthandbuch.de

Abbildung 5.12 Die Methode »querySelector()« liefert das erste Element zurück, das auf den CSS-Selektor zutrifft.

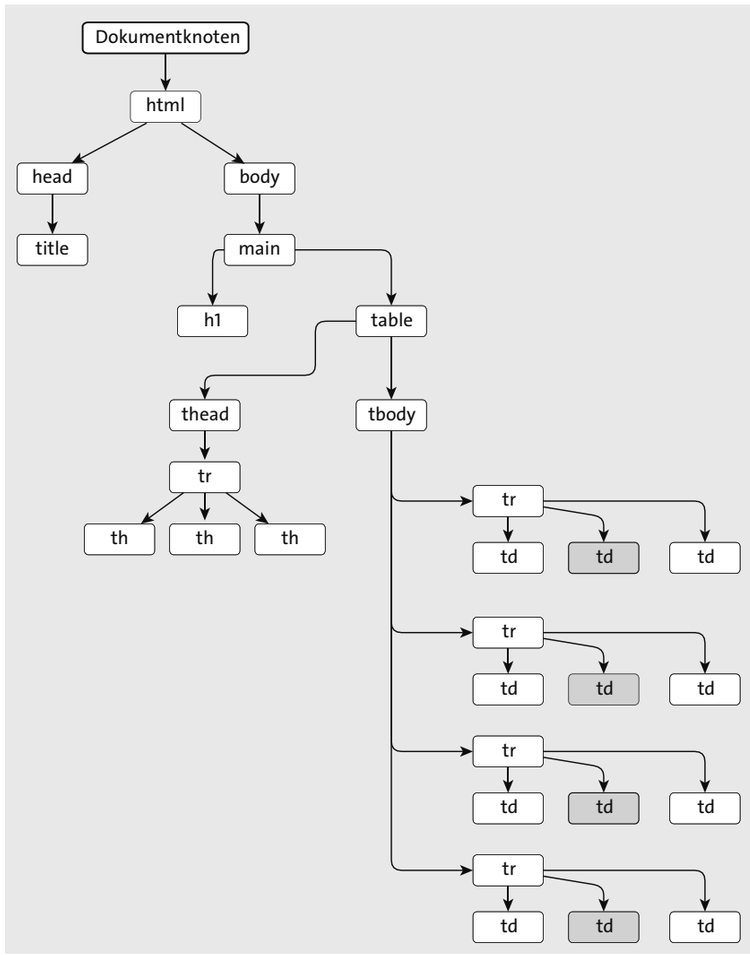

Abbildung 5.13 Die Methode »querySelectorAll()« kann mehrere Elemente zurückgeben.

Innerhalb der Schleife werden diese Elemente dann auf die gleiche Weise wie eben mit einem roten Rahmen versehen (siehe Abbildung 5.14).

```
const tableCells = document.querySelectorAll('#main table td:nth-child(2)');
if(tableCells.length > 0) {
  for(let i=0; i<tableCells.length; i++) {
    const tableCell = tableCells[i];
    tableCell.style.border = 'thick solid red';
  }
}
```

Listing 5.14 Zugriff auf mehrere Elemente über CSS-Selektor

Abbildung 5.14 Die Methode »querySelectorAll()« liefert alle Elemente zurück, die auf den übergebenen CSS-Selektor zutreffen.

Auf die Möglichkeiten, die `querySelector()` und `querySelectorAll()` bieten, haben Webentwickler lange gewartet. Vor Einführung der sogenannten *Selector API* (aktuelle Version siehe *www.w3.org/TR/selectors4*), die unter anderem diese beiden wichtigen Methoden definiert, musste man mit den weiter oben vorgestellten DOM-Methoden zur Selektion von Elementen vorliebnehmen.

Die Bibliothek jQuery hat diese Einschränkung schon beizeiten erkannt und entsprechende Helferfunktionen bereits recht früh zur Verfügung gestellt. In Kapitel 10, »Aufgaben vereinfachen mit jQuery«, werden wir unter anderem auch auf diesen Aspekt dieser bekannten JavaScript-Bibliothek eingehen.

Insgesamt erleichtern die Methoden zur Selektion über CSS-Selektoren die Arbeit eines JavaScript-Entwicklers erheblich. Eine Übersicht über die verschiedenen CSS-Selektoren zeigt Tabelle 5.3.

Selektor	Beschreibung	Seit CSS-Version
`*`	Selektiert jedes Element.	2
`E`	Selektiert Elemente vom Typ `E`.	1
`[a]`	Selektiert Elemente mit dem Attribut `a`.	2
`[a="b"]`	Selektiert Elemente mit dem Attribut `a`, das den Wert `b` hat.	2
`[a~="b"]`	Selektiert Elemente mit dem Attribut `a`, das als Wert eine Liste von Werten hat, von denen einer gleich `b` ist.	2
`[a^="b"]`	Selektiert Elemente mit dem Attribut `a`, dessen Wert mit `b` beginnt.	3
`[a$="b"]`	Selektiert Elemente mit dem Attribut `a`, dessen Wert mit `b` endet.	3
`[a*="b"]`	Selektiert Elemente mit dem Attribut `a`, dessen Wert `b` als Substring enthält.	3
`[a|="b"]`	Selektiert Elemente, deren Werte des Attributs `a` eine Reihe von mit Minuszeichen getrennten Werten ist, wobei der erste Wert `b` ist.	2
`:root`	Selektiert das Wurzelelement eines Dokuments.	3
`:nth-child(n)`	Selektiert das n-te Kindelement eines Elements.	3
`:nth-last-child(n)`	Selektiert das n-te Kindelement eines Elements von hinten.	3
`:nth-of-type(n)`	Selektiert das n-te Geschwisterelement bestimmten Typs eines Elements.	3
`:nth-last-of-type(n)`	Selektiert das n-te Geschwisterelement bestimmten Typs eines Elements von hinten.	3
`:first-child`	Selektiert das erste Kindelement eines Elements.	2
`:last-child`	Selektiert das letzte Kindelement eines Elements.	3
`:first-of-type`	Selektiert das erste Geschwisterelement eines Elements.	3

Tabelle 5.3 Die verschiedenen Selektoren in CSS3

Selektor	Beschreibung	Seit CSS-Version
`:last-of-type`	Selektiert das letzte Geschwisterelement eines Elements.	3
`:only-child`	Selektiert Elemente, die das einzige Kindelement ihres Elternelements sind.	3
`:only-of-type`	Selektiert Elemente, die das einzige Element ihres Typs unter ihren Geschwisterelementen sind.	3
`:empty`	Selektiert Elemente, die keine Kindelemente haben.	3
`:link`	Selektiert Links, die noch nicht angeklickt wurden.	2
`:visited`	Selektiert Links, die bereits angeklickt wurden.	2
`:active`	Selektiert Links, die gerade in dem Moment angeklickt werden.	2
`:hover`	Selektiert Links, über denen sich gerade die Maus befindet.	2
`:focus`	Selektiert Links, die gerade den Fokus haben.	2
`:target`	Selektiert Sprungmarken, die über Links innerhalb einer Webseite erreicht werden können.	3
`:lang(de)`	Selektiert Elemente, deren `lang`-Attribut den Wert `de` hat.	2
`:enabled`	Selektiert Formularelemente, in die Werte eingegeben bzw. die bedient werden können (und nicht deaktiviert sind).	3
`:disabled`	Selektiert Formularelemente, die nicht bedient werden können bzw. für die über das `disabled`-Attribut die Eingabe gesperrt wurde.	3
`:checked`	Selektiert Checkboxen und Radiobuttons, die aktiviert sind.	3
`.className`	Selektiert Elemente, deren `class`-Attribut den Wert `className` hat.	1
`#main`	Selektiert Elemente, deren `id`-Attribut den Wert `main` hat.	1

Tabelle 5.3 Die verschiedenen Selektoren in CSS3 (Forts.)

Selektor	Beschreibung	Seit CSS-Version
`:not(s)`	Selektiert Elemente, die nicht auf den in Klammern angegebenen Selektor `s` zutreffen.	3
`E F`	Selektiert Elemente vom Typ `F`, die irgendwo innerhalb eines Elements vom Typ `E` vorkommen.	1
`E > F`	Selektiert Elemente vom Typ `F`, die Kindelemente eines Elements vom Typ `E` sind.	2
`E + F`	Selektiert Elemente vom Typ `F`, die direkt nachfolgende Geschwisterelemente eines Elements vom Typ `E` sind.	2
`E ~ F`	Selektiert Elemente vom Typ `F`, die Geschwisterelemente eines Elements vom Typ `E` sind.	3

Tabelle 5.3 Die verschiedenen Selektoren in CSS3 (Forts.)

5.2.6 Das Elternelement eines Elements selektieren

Elementknoten verfügen über verschiedene Eigenschaften, mit denen Sie auf verwandte Elemente zugreifen können. Verwandte Elemente sind Elternknoten bzw. -elemente, Kindknoten bzw. -elemente und Geschwisterknoten bzw. -elemente.

Für die Selektion von Elternknoten/-elementen stehen die Eigenschaften `parentNode` und `parentElement` zur Verfügung, für die Selektion von Kindknoten/-elementen die Eigenschaften `firstChild`, `firstElementChild`, `lastChild`, `lastElementChild`, `childNodes` und `children`, und für die Selektion von Geschwisterknoten/-elementen gibt es die Eigenschaften `previousSibling`, `previousElementSibling`, `nextSibling` und `nextElementSibling`.

Lassen Sie mich auf diese Eigenschaften im Folgenden etwas genauer eingehen. Beginnen wir dabei mit der Selektion von Elternknoten bzw. -elementen.

Um den Elternknoten eines Elements (bzw. Knotens) zu selektieren, steht die Eigenschaft `parentNode` zur Verfügung, um dagegen das Elternelement zu selektieren, die Eigenschaft `parentElement`. In den meisten Fällen ist der Elternknoten auch immer ein Element, sprich, die beiden Eigenschaften `parentNode` und `parentElement` enthalten den gleichen Wert (siehe Listing 5.15 und Abbildung 5.15).

```
const table = document.querySelector('table');
console.log(table.parentNode);     // <main>
console.log(table.parentElement); // <main>
```

Listing 5.15 Zugriff auf Elternknoten bzw. Elternelement

Knoten und Elemente

Nicht alle Knoten im DOM-Baum sind Elemente, aber alle Elemente sind immer Knoten.

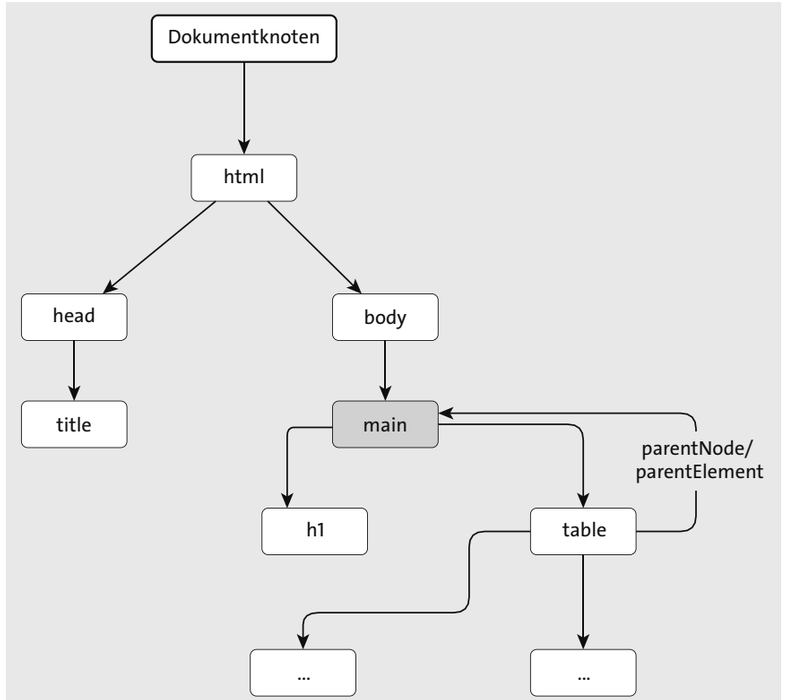

Abbildung 5.15 Selektion des Elternelements

Wichtig zu verstehen ist, dass einige der oben genannten Eigenschaften Knoten zurückgeben, andere Eigenschaften dagegen Elemente. Die Eigenschaften `parentNode`, `childNodes`, `firstChild`, `lastChild`, `previousSibling` und `nextSibling` geben Knoten zurück, während die Eigenschaften `parentElement`, `children`, `firstElementChild`, `lastElementChild`, `previousElementSibling` und `nextElementSibling` Elemente zurückgeben.

Was das konkret bedeutet, verdeutlicht folgendes Beispiel. Schauen Sie sich dazu den HTML-Code in Listing 5.16 und dessen DOM in Abbildung 5.16 an. Gezeigt ist hier eine relativ einfach aufgebaute Webseite, bei der innerhalb des `<body>`-Elements lediglich zwei `<span>`-Elemente sowie jeweils davor und dahinter Text enthalten sind.

Das entsprechende DOM enthält unterhalb des `<body>`-Elements demnach (in dieser Reihenfolge) einen Textknoten, einen Elementknoten, einen Textknoten, einen Elementknoten und wieder einen Textknoten. Für alle diese Knoten stellt das `<body>`-Element zugleich sowohl den

Elternknoten als auch das Elternelement dar. Somit liefern für alle diese Knoten die Eigenschaften `parentNode` und `parentElement` den gleichen Wert: eben das `<body>`-Element.

Auch können Sie anhand des DOM in Abbildung 5.16 sehen, dass die Eigenschaften `parentNode` und `parentElement` generell für alle Knoten immer das gleiche Element referenzieren. Einzige Ausnahme: das `<html>`-Element. Dieses Element hat nämlich kein Elternelement, sondern »nur« einen Elternknoten, sprich den Dokumentknoten. Die Eigenschaft `parentElement` liefert in diesem Fall also den Wert `null`.

Auf die anderen Beziehungen zwischen Elementen und Knoten im DOM werde ich in den folgenden Abschnitten eingehen.

```
<!DOCTYPE html>
<html>
<body>
  Text
  <span></span>
  Text
  <span></span>
  Text
</body>
</html>
```

Listing 5.16 Ein einfaches HTML-Beispiel

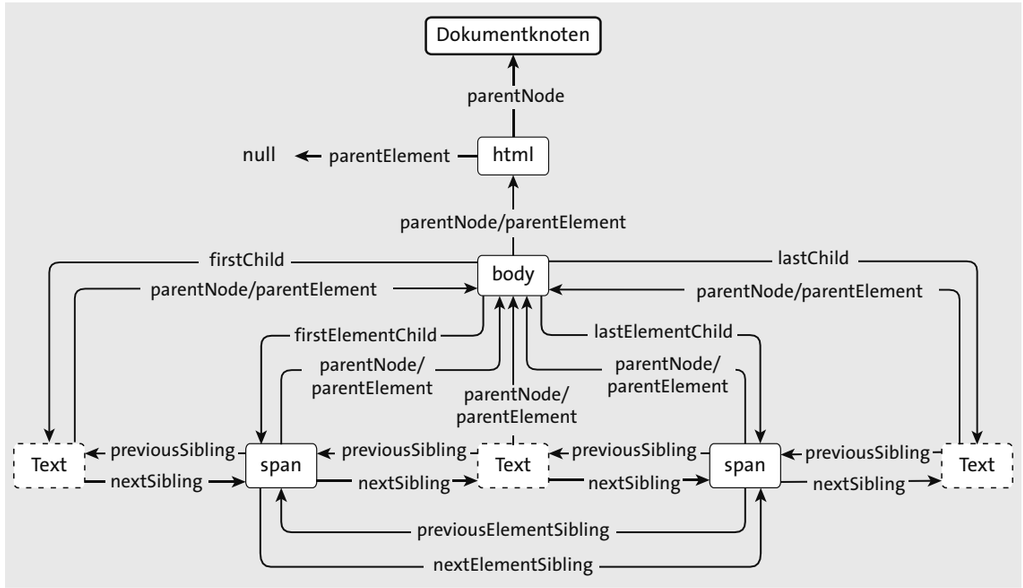

Abbildung 5.16 Übersicht über die verschiedenen Zugriffsformen

5.2.7 Die Kindelemente eines Elements selektieren

Die Kindelemente eines Elements lassen sich über die Eigenschaft `children` ermitteln, die Kindknoten über die Eigenschaft `childNodes`. Ob ein Element Kindknoten hat, lässt sich über die Methode `hasChildNodes()` bestimmen, die einen booleschen Wert zurückgibt. Ob ein Element Kindelemente hat, können Sie über die Eigenschaft `childElementCount` bestimmen: Diese enthält die Anzahl an Kindelementen.

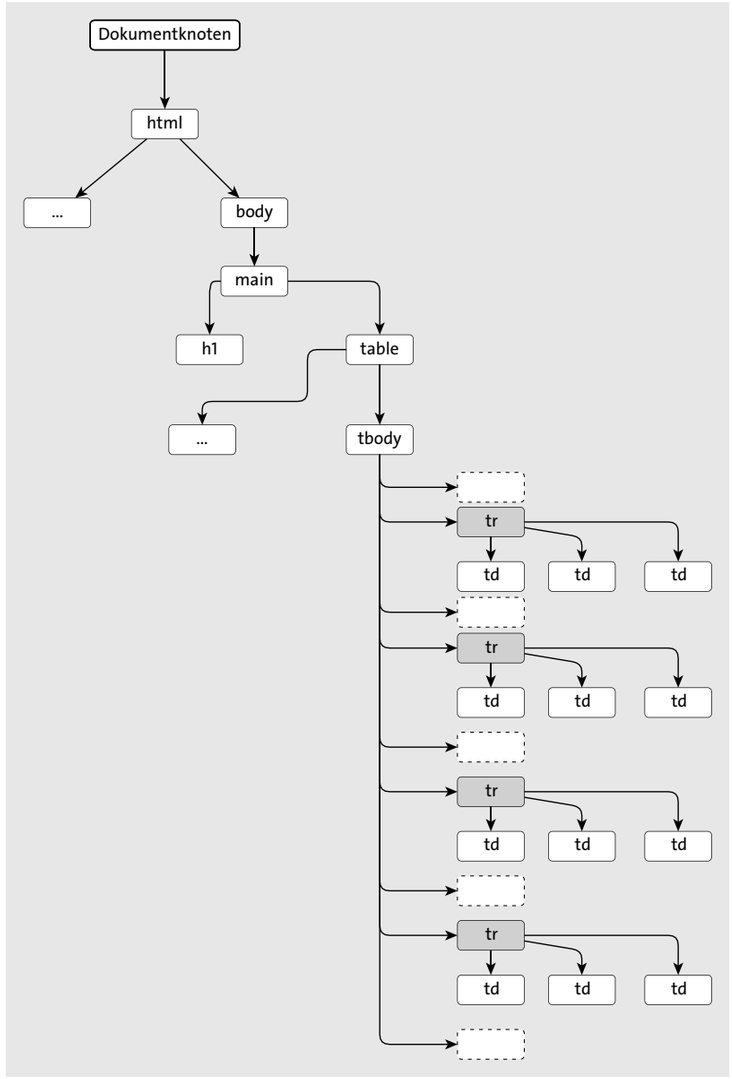

Abbildung 5.17 Selektion aller Kindelemente

Listing 5.17 zeigt hierzu einige Beispiele (bezogen wieder auf das HTML aus Listing 5.1). Sie sehen: Das Element `<tbody>` hat vier Kindelemente (nämlich die vier `<tr>`-Elemente, siehe Abbildung 5.17) und insgesamt neun Kindknoten (siehe Abbildung 5.18).

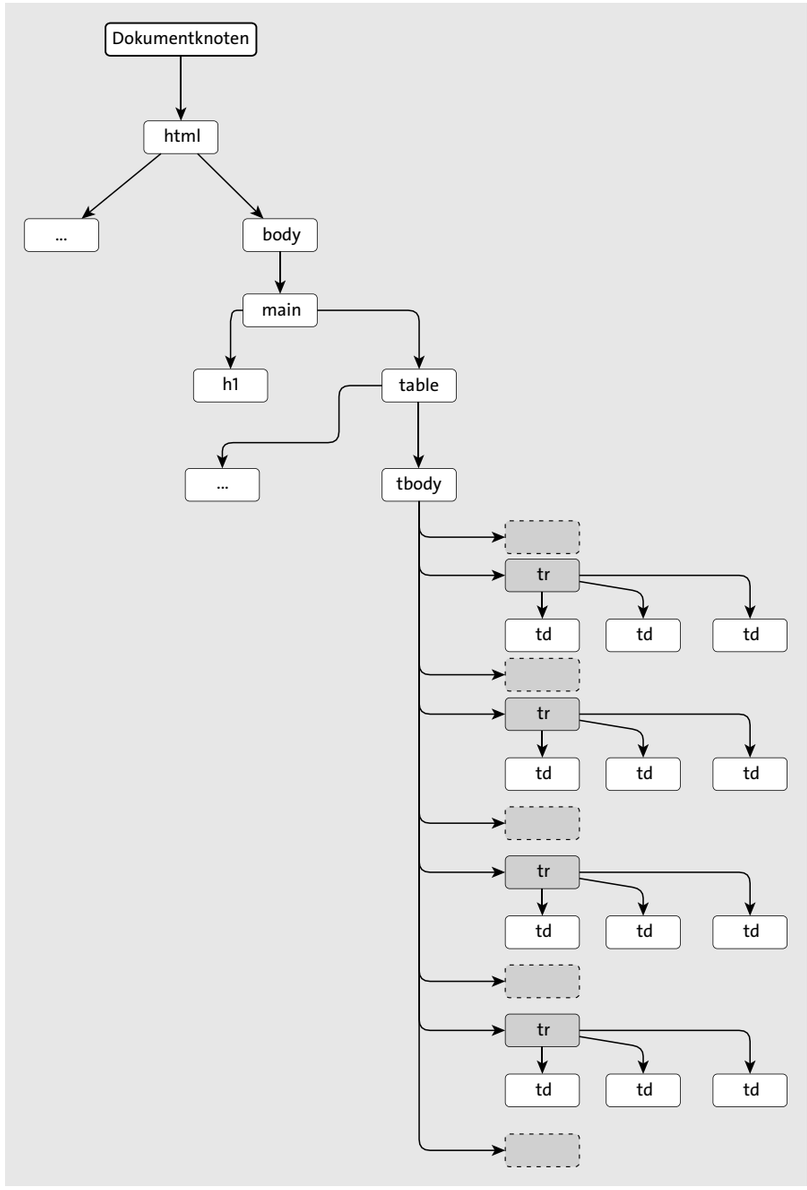

Abbildung 5.18 Selektion aller Kindknoten

Der Grund dafür ist, dass – obwohl zwischen, vor und hinter den vier `<tr>`-Elementen kein Text im HTML vorkommt – sogenannte Weißraumknoten erzeugt werden (siehe Kasten). Diese Weißraumknoten entstehen immer dann, wenn zwischen zwei Elementen beispielsweise Zeilenumbrüche im HTML verwendet werden.

```
const tbody = document.querySelector('tbody');
console.log(tbody.children.length);    // 4
console.log(tbody.childElementCount);  // 4
console.log(tbody.childNodes.length);  // 9
console.log(tbody.hasChildNodes());    // true
```

Listing 5.17 Zugriff auf Kindknoten bzw. Kindelemente

Weißraumknoten

Leerraum innerhalb des HTML-Codes, der beispielsweise durch Leerzeichen, Tabulatoren oder auch Zeilenumbrüche erzeugt wird, führt dazu, dass im DOM dafür jedes Mal Textknoten ohne Text erzeugt werden. In solchen Fällen spricht man von Weißraumknoten.

Darüber hinaus stehen verschiedene weitere Eigenschaften zur Verfügung, mit denen sich gezielt einzelne Kindelemente bzw. Kindknoten selektieren lassen:

- Die Eigenschaft `firstChild` enthält den ersten Kindknoten.
- Die Eigenschaft `lastChild` enthält den letzten Kindknoten.
- Die Eigenschaft `firstElementChild` enthält das erste Kindelement.
- Die Eigenschaft `lastElementChild` enthält das letzte Kindelement.

Listing 5.18 zeigt einige Beispiele dazu, Abbildung 5.19 das Ergebnis der Selektion des ersten und letzten Kindelements und Abbildung 5.20 das Ergebnis der Selektion des ersten und letzten Kindknotens.

```
const tbody = document.querySelector('tbody');
console.log(tbody.firstChild);          // Textknoten
console.log(tbody.lastChild);           // Textknoten
console.log(tbody.firstElementChild);   // <tr>
console.log(tbody.lastElementChild);    // <tr>
```

Listing 5.18 Zugriff auf spezielle Kindknoten und Kindelemente

Hinweis

In den meisten Fällen werden Sie wahrscheinlich mit Elementknoten arbeiten. In diesen Fällen verwenden Sie am besten Eigenschaften, die auch Elementknoten zurückgeben (wie beispielsweise `firstElementChild` und `lastElementChild`). Es gab dagegen eine Zeit, in der Webentwicklern nur Eigenschaften zur Verfügung standen, die alle Arten von Knoten zu-

rückgeben (beispielsweise `firstChild` und `lastChild`), und man anhand des Knotentyps selbst die Elementknoten herausfiltern musste. Dies ist zum Glück nicht mehr so.

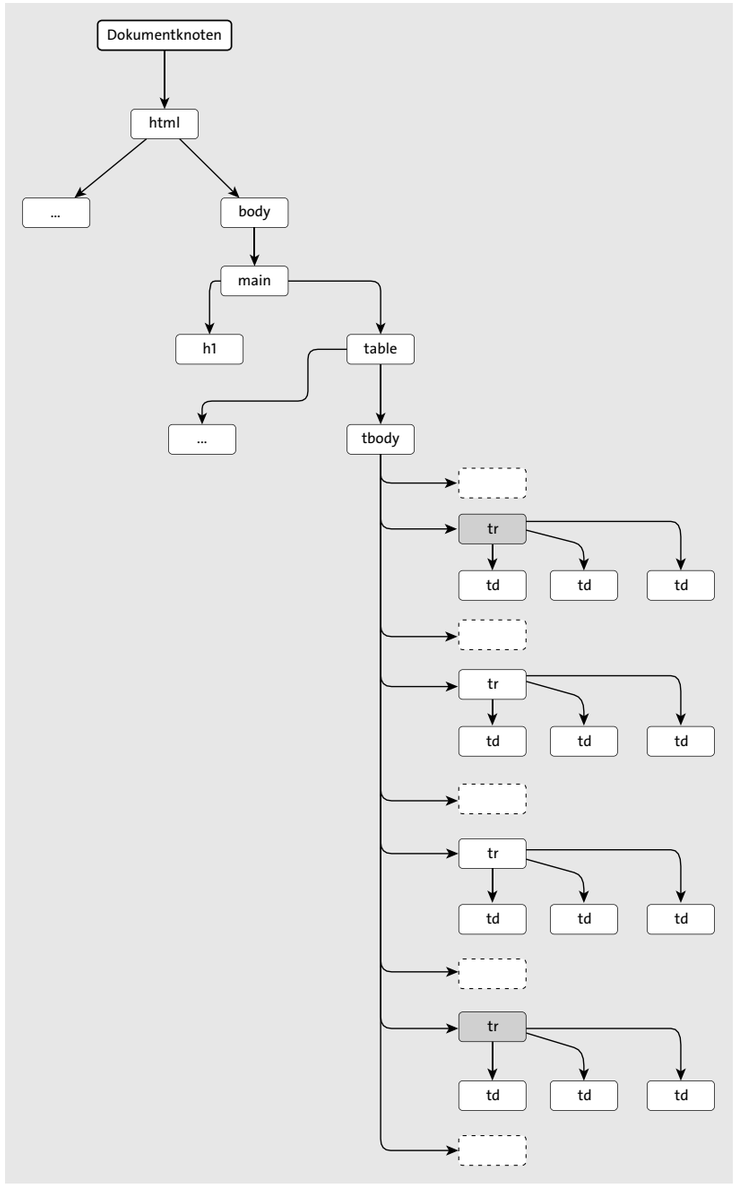

Abbildung 5.19 Selektion des ersten und des letzten Kindelements

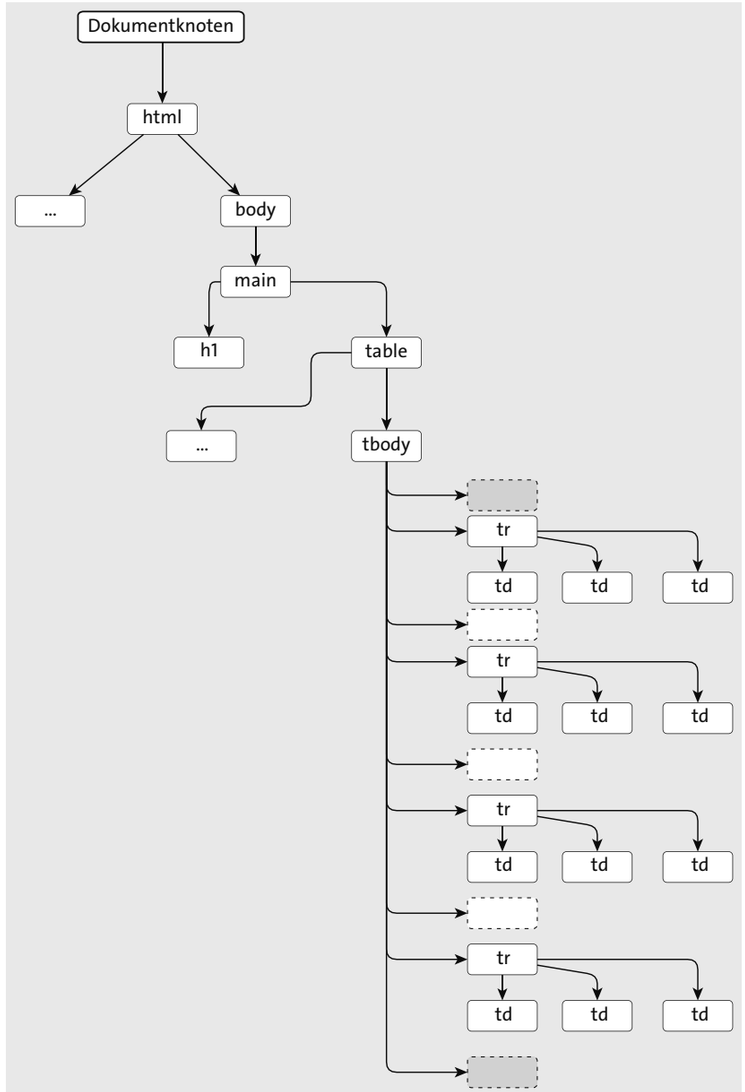

Abbildung 5.20 Selektion des ersten und des letzten Kindknotens

5.2.8 Die Geschwisterelemente eines Elements selektieren

Sie wissen jetzt also, wie Sie im DOM-Baum ausgehend von einem Knoten/Element über dessen Eigenschaften Knoten/Elemente oberhalb selektieren (Elternknoten/Elternelemente) und wie Sie Knoten/Elemente unterhalb selektieren können (Kindknoten/Kindelemente).

Zusätzlich gibt es aber auch die Möglichkeit, *innerhalb einer Ebene* des DOM die Geschwisterknoten bzw. Geschwisterelemente zu selektieren:

- Die Eigenschaft `previousSibling` enthält den vorherigen Geschwisterknoten.
- Die Eigenschaft `nextSibling` enthält den nachfolgenden Geschwisterknoten.
- Die Eigenschaft `previousElementSibling` enthält das vorherige Geschwisterelement.
- Die Eigenschaft `nextElementSibling` enthält das nachfolgende Geschwisterelement.

Listing 5.19 zeigt dazu ein Codebeispiel. Ausgehend von der zweiten Tabellenzeile, werden zunächst der vorhergehende Geschwisterknoten (über `previousSibling`) und der nachfolgende Geschwisterknoten (über `nextSibling`) selektiert, wobei es sich in beiden Fällen um Textknoten (genauer gesagt, Weißraumknoten) handelt (siehe Abbildung 5.21).

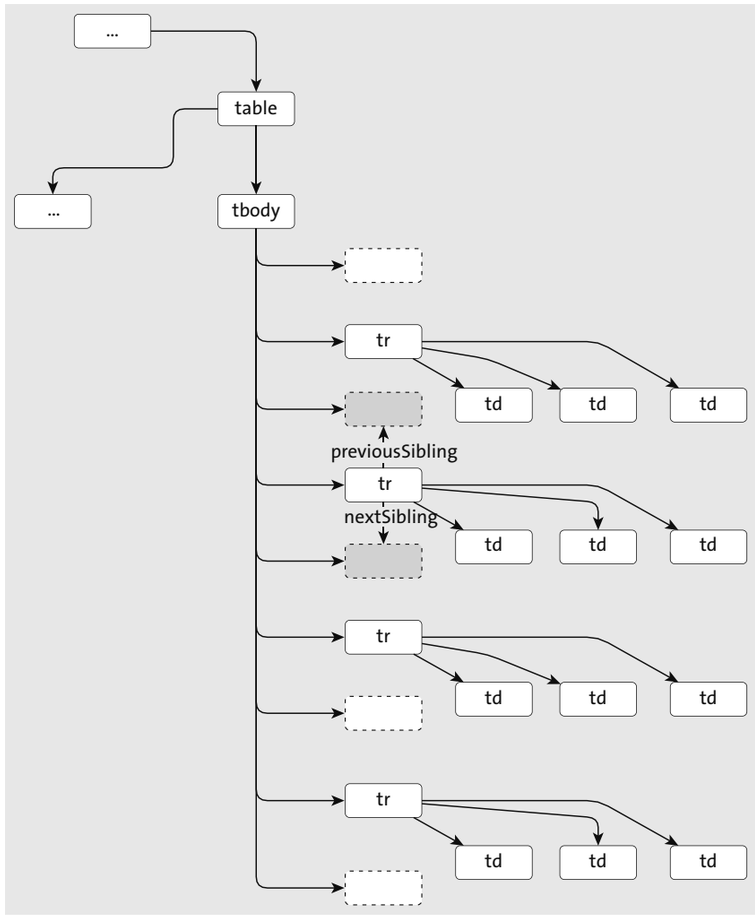

Abbildung 5.21 Selektion des vorherigen und des nachfolgenden Geschwisterknotens

Anschließend werden über `previousElementSibling` das vorhergehende Geschwisterelement und über `nextElementSibling` das nachfolgende Geschwisterelement selektiert (siehe Abbildung 5.22).

```
const tableCell = document.querySelector('tbody tr:nth-child(2)');
console.log(tableCell.previousSibling);        // Textknoten
console.log(tableCell.nextSibling);            // Textknoten
console.log(tableCell.previousElementSibling); // <tr>
console.log(tableCell.nextElementSibling);     // <tr>
```

Listing 5.19 Zugriff auf spezielle Geschwisterknoten und Geschwisterelemente

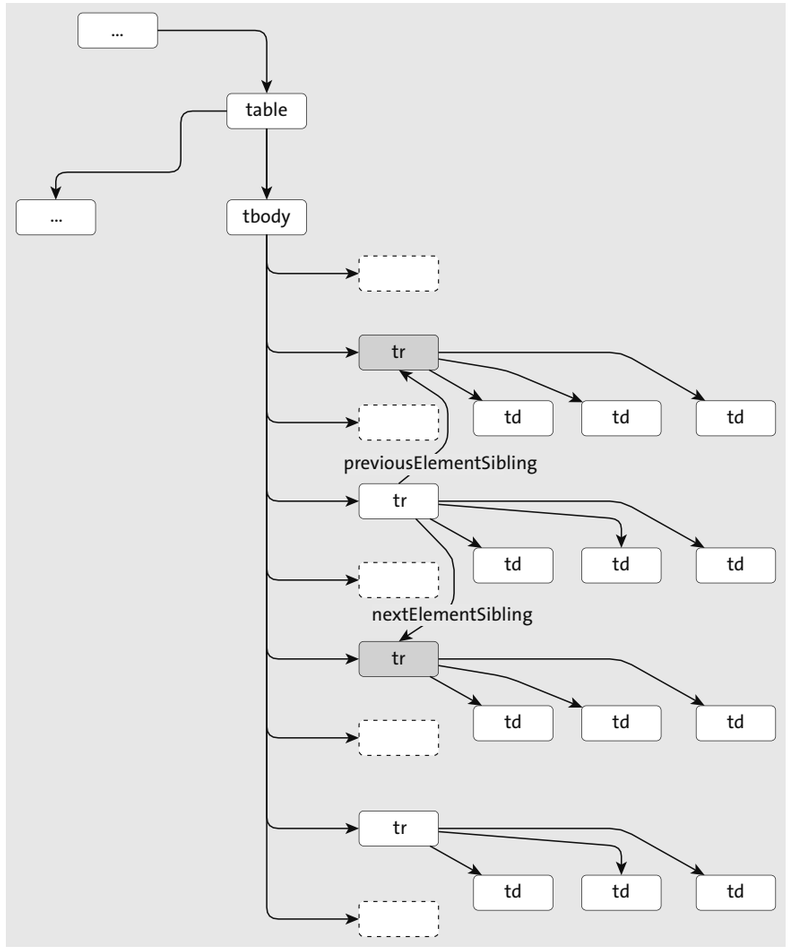

Abbildung 5.22 Selektion des vorherigen und des nachfolgenden Geschwisterelements

5.2.9 Selektionsmethoden auf Elementen aufrufen

Die meisten der vorgestellten DOM-Methoden zur Selektion von Elementen (`getElementsByClassName()`, `getElementsByTagName()`, `querySelector()` und `querySelectorAll()`) lassen sich nicht nur auf dem Dokumentknoten (also auf `document`), sondern auch auf allen anderen Elementknoten einer Webseite aufrufen (lediglich `getElementById()` und `getElementsByName()` lassen sich nur auf dem Dokumentknoten aufrufen). In diesem Fall bezieht die Suche nach den Elementen nur den Teilbaum unterhalb des Elements mit ein, auf dem die jeweilige Methode aufgerufen wurde.

Betrachten Sie dazu folgenden HTML-Code in Listing 5.20, der geschachtelte Listen enthält. Im JavaScript-Code in Listing 5.21 wird die Methode `getElementsByTagName()` mit dem Argument `li` zunächst auf dem Dokumentknoten `document` aufgerufen (wodurch alle Listeneinträge der gesamten Webseite selektiert werden, siehe Abbildung 5.23) und anschließend auf der geschachtelten Liste mit der ID `list-2` (wodurch wiederum nur die Listeneinträge selektiert werden, die in diesem Teilbaum des DOM, also unterhalb der geschachtelten Liste, vorkommen, siehe Abbildung 5.24).

```
<!DOCTYPE html>
<html>
  <head lang="de">
    <title>Beispiel zur Selektion von Elementen</title>
  </head>
  <body>
    <main id="main-content">
      <ul id="list-1">
        <li>Listeneintrag 1</li>
        <li>
          Listeneintrag 2
          <ul id="list-2">
            <li>Listeneintrag 2.1</li>
            <li>Listeneintrag 2.2</li>
            <li>Listeneintrag 2.3</li>
            <li>Listeneintrag 2.4</li>
          </ul>
        </li>
        <li>Listeneintrag 3</li>
        <li>Listeneintrag 4</li>
      </ul>
    </main>
  </body>
</html>
```

Listing 5.20 Beispiel HTML-Seite

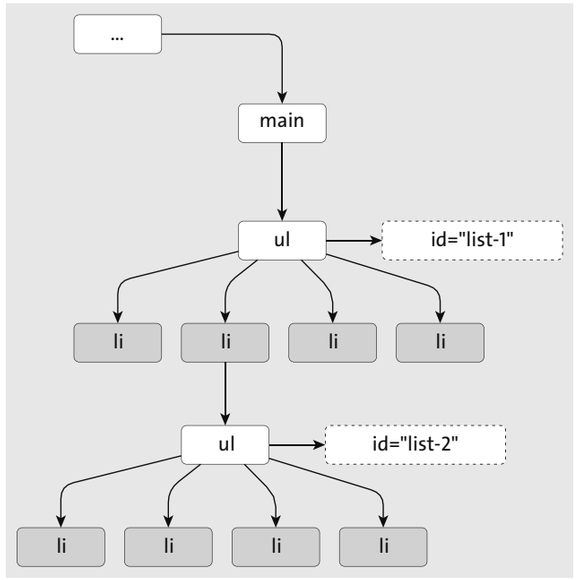

Abbildung 5.23 Aufruf der Methode »getElementsByTagName()« auf dem Dokumentknoten

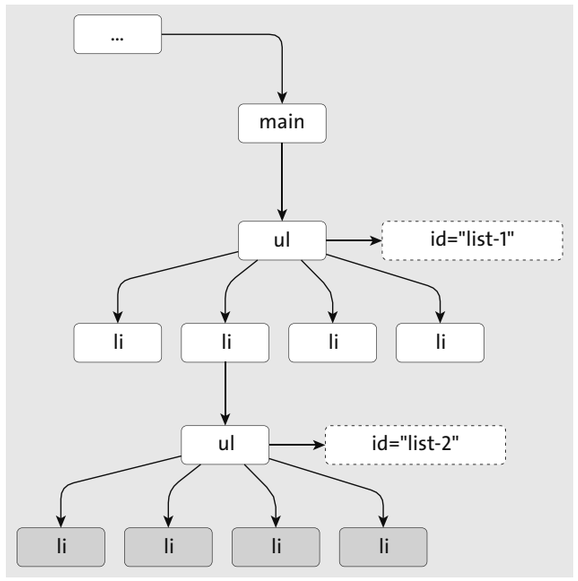

Abbildung 5.24 Aufruf der Methode »getElementsByTagName()« auf dem -Element mit der ID »list-2«


```
const allListItemElements = document.getElementsByTagName('li');
console.log(allListItemElements.length); // Ausgabe: 8
const subList = document.getElementById('list-2');
const subListListItems = subList.getElementsByTagName('li');
console.log(subListListItems.length); // Ausgabe: 4
```

Listing 5.21 Selektion von Elementen ausgehend von einem Elternelement

5.2.10 Elemente nach Typ selektieren

Neben den vorgestellten Selektionsmethoden bietet das `document`-Objekt verschiedene Eigenschaften, um auf bestimmte Elemente einer Webseite direkt zugreifen zu können. Über die Eigenschaft `anchors` können beispielsweise alle Anker (sprich Sprungelemente) auf einer Webseite selektiert werden, über `forms` alle Formulare, über `images` alle Bilder und über `links` alle Links. Zudem kann über die Eigenschaft `head` direkt auf das `<head>`-Element und über die Eigenschaft `body` direkt auf das `<body>`-Element zugegriffen werden.

Eigenschaft	Beschreibung
`document.anchors`	Enthält eine Liste aller Anker der Webseite.
`document.forms`	Enthält eine Liste aller Formulare der Webseite.
`document.images`	Enthält eine Liste aller Bilder der Webseite.
`document.links`	Enthält eine Liste aller Links der Webseite.
`document.head`	Zugriff auf das `<head>`-Element der Webseite.
`document.body`	Zugriff auf das `<body>`-Element der Webseite.

Tabelle 5.4 Verschiedene Eigenschaften zur Selektion von Elementen nach Typ

5.3 Mit Textknoten arbeiten

Wenn Sie ein oder mehrere Elemente selektiert haben, können Sie diese verändern: Sie können Text hinzufügen oder entfernen, Attribute hinzufügen oder entfernen sowie Elemente hinzufügen oder entfernen. Folgende Tabelle zeigt einen Überblick über die wichtigsten Eigenschaften und Methoden, die dafür zur Verfügung stehen und die wir in den folgenden Abschnitten im Detail besprechen werden.

Wie ich ja bereits sagte, wird jeglicher Text auf einer Webseite innerhalb des DOM-Baums als Textknoten repräsentiert. Schauen wir uns nun an, wie Sie auf die Textinhalte zugreifen und diese auch verändern können.

Eigenschaft/Methode	Beschreibung	Abschnitt
`textContent`	Über diese Eigenschaft können Sie auf den Textinhalt eines Knotens zugreifen.	Abschnitt 5.3.1, »Auf den Textinhalt eines Elements zugreifen«
`nodeValue`	Über diese Eigenschaft können Sie auf den Inhalt eines Knotens zugreifen.	Abschnitt 5.3.1, »Auf den Textinhalt eines Elements zugreifen«
`innerHTML`	Über diese Eigenschaft können Sie auf den HTML-Inhalt eines Knotens zugreifen.	Abschnitt 5.3.3, »Das HTML unterhalb eines Elements verändern«
`createTextNode()`	Mit dieser Methode können Sie Textknoten erstellen.	Abschnitt 5.3.4, »Textknoten erstellen und hinzufügen«
`createElement()`	Mit dieser Methode können Sie Elemente erstellen.	Abschnitt 5.4.1, »Elemente erstellen und hinzufügen«
`createAttribute()`	Mit dieser Methode können Sie Attributknoten erstellen.	Abschnitt 5.5.3, »Attributknoten erstellen und hinzufügen«
`appendChild()`	Mit dieser Methode können Sie dem DOM-Baum Knoten hinzufügen.	Abschnitt 5.4.1, »Elemente erstellen und hinzufügen«
`removeChild()`	Mit dieser Methode können Sie Knoten aus dem DOM-Baum entfernen.	Abschnitt 5.4.2, »Elemente und Knoten entfernen«

Tabelle 5.5 Die verschiedenen Methoden und Eigenschaften für das Verändern von Elementen

5.3.1 Auf den Textinhalt eines Elements zugreifen

Um auf den reinen Textinhalt eines Elements zugreifen zu können, verwenden Sie am besten die Eigenschaft `textContent`. Das Praktische an dieser Eigenschaft ist, dass eventuelle HTML-Auszeichnungen (Markup) innerhalb des jeweiligen Elements ignoriert werden und in dem Wert, den man zurückerhält, nicht enthalten sind. Die folgenden beiden Listings machen dies deutlich: In Listing 5.22 sehen Sie eine einfache HTML-Liste mit einem Eintrag, wobei der dort enthaltene Text durch `<strong>`- und `<em>`-Elemente ausgezeichnet ist.

```
<ul id="news">
  <li>
    <strong>Platten-News: </strong>Neues Album von <em>Ben Harper</em> erschienen.
  </li>
</ul>
```

Listing 5.22 HTML mit geschachtelten Elementen

Greifen Sie jetzt wie in Listing 5.23 auf die Eigenschaft `textContent` zu, sehen Sie, dass diese nur den reinen Text des `<li>`-Elements enthält, nicht aber die darin enthaltenen Auszeichnungen `<strong>` und `<em>`.

```
const textContent = document.querySelector('#news li:nth-child(1)').textContent;
console.log(textContent);
// Ausgabe: Platten-News: Neues Album von Ben Harper erschienen.
```

Listing 5.23 Die Eigenschaft »textContent« ignoriert Markup innerhalb des entsprechenden Elements.

Merke

Die Eigenschaft `textContent` ist sehr praktisch, da man in der Praxis bei Zugriff auf den Textinhalt eines Elements häufig eben nicht daran interessiert ist, ob und welche zusätzlichen Auszeichnungen verwendet wurden.

5.3.2 Den Textinhalt eines Elements verändern

Möchten Sie den Textinhalt eines Elements neu setzen, verwenden Sie ebenfalls die Eigenschaft `textContent`. Als Wert übergeben Sie einfach den neuen Text, wie in Listing 5.24 zu sehen. Hier wird dem Listenelement von eben ein neuer Text zugewiesen.

```
const element = document.querySelector('#news li:nth-child(1)');
element.textContent = 'Platten-News: Neues Album von Tool endlich
  erschienen.';
```

Listing 5.24 Über die Eigenschaft »textContent« lässt sich der Textinhalt eines Elements neu setzen.

Zu beachten ist dabei aber, dass es über `textContent` nicht möglich ist, Markup, sprich HTML-Auszeichnungen, hinzuzufügen: Obwohl die übergebene Zeichenkette in folgendem Listing Auszeichnungen enthält, werden diese nicht interpretiert, sondern als Text dargestellt (siehe Abbildung 5.25).

```
const element = document.querySelector('#news li:nth-child(1)');
element.textContent = '<strong>Platten-News:</strong> Neues Album von
  <em>Tool</em> endlich erschienen.';
```

Listing 5.25 Das Markup innerhalb der angegebenen Zeichenkette wird nicht ausgewertet.

- Platten-News: Neues Album von Tool immer noch nicht erschienen.

Abbildung 5.25 Über »textContent« angegebenes Markup wird nicht ausgewertet.

»textContent« vs. »innerText«

In einigen Browsern steht Ihnen noch die Eigenschaft `innerText` zur Verfügung, die so ähnlich arbeitet wie `textContent`, sich im Detail allerdings etwas unterscheidet und zudem nicht in der DOM API enthalten ist und daher beispielsweise auch nicht von Firefox unterstützt wird. Ich rate Ihnen daher, auf `innerText` zu verzichten und stattdessen wie gezeigt `textContent` zu verwenden.

5.3.3 Das HTML unterhalb eines Elements verändern

Möchten Sie nicht nur Text, sondern auch HTML in ein Element einfügen, können Sie die Eigenschaft `innerHTML` verwenden. Sie werden zwar später mit der sogenannten *DOM-Bearbeitung* noch eine weitere Möglichkeit kennenlernen, die in der Praxis häufiger zum Einsatz kommt, um HTML in das DOM einzubauen, aber für den Anfang bzw. für einfache HTML-Bausteine, die hinzugefügt werden sollen, reicht zunächst `innerHTML`. Listing 5.26 zeigt dazu ein Beispiel: Hier wird der gleiche HTML-Baustein wie schon in Listing 5.25 hinzugefügt, diesmal allerdings auch als HTML interpretiert (siehe Abbildung 5.26).

```
const element = document.querySelector('#news li:nth-child(1)');
element.innerHTML = '<strong>Platten-News:</strong> Neues Album von <em>Tool ↩
  </em> endlich erschienen.';
```

Listing 5.26 Mit der Eigenschaft »innerHTML« wird in der übergebenen Zeichenkette enthaltenes Markup ausgewertet.

- **Platten-News:** Neues Album von *Tool* immer noch nicht erschienen.

Abbildung 5.26 Wie erwartet: Das per »innerHTML« eingefügte HTML wird ausgewertet.

Umgekehrt können Sie über `innerHTML` auch den HTML-Inhalt eines Elements auslesen. Als Ergebnis erhalten Sie wie schon bei `textContent` eine Zeichenkette, in der nun allerdings nicht nur der Textinhalt, sondern auch die HTML-Auszeichnungen enthalten sind (siehe Listing 5.27).

```
const innerHTML = document.querySelector('#news li:nth-child(1)').innerHTML;
console.log(innerHTML);
// Ausgabe: <strong>Platten-News: </strong>Neues Album von
// <em>Ben Harper</em> erschienen.
```

Listing 5.27 Die Eigenschaft »innerHTML« enthält auch die HTML-Auszeichnungen.

5.3.4 Textknoten erstellen und hinzufügen

Alternativ zu den gezeigten Möglichkeiten, über die Eigenschaften `textContent` und `innerHTML` auf den Text innerhalb einer Webseite zuzugreifen oder diesen zu verändern, gibt es noch die Möglichkeit, Textknoten zu erstellen und diese manuell dem DOM-Baum hinzuzufügen. Dazu bietet die DOM API die Methode `createTextNode()` an. In Listing 5.28 wird über diese Methode ein Textknoten (mit dem Text `Beispiel`) erstellt und anschließend über die Methode `appendChild()` (dazu später noch mehr) einem bestehenden Element als Kindknoten hinzugefügt (dieser zweite Schritt ist notwendig, da über die Methode `createTextNode()` der Textknoten noch nicht dem DOM-Baum hinzugefügt wird).

```
const element = document.getElementById('container');
const textNode = document.createTextNode('Beispiel');
element.appendChild(textNode);
```

Listing 5.28 Erstellen und Hinzufügen eines Textknotens

Weitere Methoden für das Erstellen von Knoten

Neben der Methode `createTextNode()` gibt es weitere Methoden für das Erstellen von Knoten, unter anderem die Methoden `createElement()` für das Erstellen von Elementknoten (siehe Abschnitt 5.4.1, »Elemente erstellen und hinzufügen«) und `createAttribute()` für das Erstellen von Attributknoten (siehe dazu Abschnitt 5.5.3, »Attributknoten erstellen und hinzufügen«).

Methoden von Dokumentknoten

Die Methode `createTextNode()` und auch die im Folgenden noch beschriebenen Methoden `createElement()` und `createAttribute()` stehen nur auf dem Dokumentknoten (sprich dem Objekt `document`) zur Verfügung. Diese Methoden können nicht auf anderen Knoten (und damit auch nicht auf Elementen) aufgerufen werden.

5.4 Mit Elementen arbeiten

Auch im Fall von Elementen ist es möglich, diese manuell über Methoden zu erzeugen und sie dann dem DOM-Baum hinzuzufügen (im Unterschied zur Verwendung der Eigenschaft `innerHTML`, bei der Sie die HTML-Elemente ja indirekt in Form des Textes übergeben, den Sie der Eigenschaft zuweisen).

Wie Sie Elemente über diese Methoden erstellen und hinzufügen und generell mit Elementen arbeiten können, zeige ich Ihnen nun im Folgenden.

5.4.1 Elemente erstellen und hinzufügen

Um Elemente zu erstellen, verwenden Sie die Methode `createElement()`. Diese erwartet als Parameter den Namen des zu erstellenden Elements und gibt das neue Element zurück. Durch den Aufruf der Methode wird das neue Element allerdings (wie schon zuvor Textknoten bei Verwendung der Methode `createTextNode()`) noch nicht dem DOM hinzugefügt.

Für das Hinzufügen von erzeugten Elementen zum DOM stehen dagegen verschiedene andere Methoden zur Verfügung:

- Über `insertBefore()` lässt sich das Element als Kindelement vor einem anderen Element/einem anderen Knoten hinzufügen, sprich als vorheriges Geschwisterelement definieren.
- Über `appendChild()` lässt sich das Element als letztes Kindelement eines Elternelements hinzufügen.
- Über `replaceChild()` lässt sich ein bestehendes Kindelement (bzw. ein bestehender Kindknoten) durch ein neues Kindelement ersetzen. Die Methode wird dabei auf dem Elternelement aufgerufen und erwartet als ersten Parameter den neuen Kindknoten sowie als zweiten Parameter den zu ersetzenden Kindknoten.

Textknoten hinzufügen

Die oben genannten Methoden stehen übrigens auch für das Hinzufügen von Textknoten (siehe Abschnitt 5.3.4, »Textknoten erstellen und hinzufügen«) zur Verfügung.

Ein etwas komplexeres – dafür aber praxisrelevantes – Beispiel zeigt Listing 5.29. Hier wird auf Basis einer Kontaktliste (die in Form eines Arrays repräsentiert wird) eine HTML-Tabelle erzeugt. Die einzelnen Einträge in der Kontaktliste enthalten dabei Angaben zu Vorname, Nachname und E-Mail-Adresse des jeweiligen Kontakts.

Alles rund um das Erstellen der entsprechenden Elemente geschieht innerhalb der Funktion `createTable()`. Hier wird zunächst über die Methode `querySelector()` das `<tbody>`-Element der im HTML bereits existierenden Tabelle (siehe Listing 5.30) ausgewählt und anschließend über das Array mit den Kontaktinformationen iteriert. Für jeden Eintrag wird dabei mithilfe der Methode `createElement()` eine Tabellenzeile erzeugt (`<tr>`) und für jede der zuvor genannten Eigenschaften (`firstName`, `lastName` und `email`) eine Tabellenzelle (`<td>`). Über die Methode `createTextNode()` werden für die Werte der Eigenschaften entsprechende Textknoten erzeugt und über `appendChild()` dem jeweiligen `<td>`-Element hinzugefügt (alternativ könnte man hier auch die Eigenschaft `textContent` verwenden).

Die erzeugten Tabellenzellen werden dann – am Ende jeder Iteration – der entsprechenden Tabellenzeile als Kindelemente hinzugefügt und – in der letzten Zeile der Iteration – die Tabellenzeile als Kindelement des Tabellenkörpers, sprich des `<tbody>`-Elements. Die einzelnen Schritte werden im Listing mithilfe von Kommentaren erläutert und können anhand von Abbildung 5.27 nachvollzogen werden.


```
const contacts = [
  {
    firstName: 'Max',
    lastName: 'Mustermann',
    email: 'max.mustermann@javascripthandbuch.de'
  },
  {
    firstName: 'Moritz',
    lastName: 'Mustermann',
    email: 'moritz.mustermann@javascripthandbuch.de'
  },
  {
    firstName: 'Peter',
    lastName: 'Mustermann',
    email: 'peter.mustermann@javascripthandbuch.de'
  }
];

function createTable() {
  const tableBody = document.querySelector('#contact-table tbody');
  for(let i=0; i<contacts.length; i++) {
    // Für den aktuellen Kontakt ...
    const contact = contacts[i];
    // ... wird eine neue Zeile erzeugt.
    // (1)
    const tableRow = document.createElement('tr');
    // Innerhalb der Zeile werden verschiedene Zellen erstellt ...
    // (2)
    const tableCellFirstName = document.createElement('td');
    const ... und jeweils mit Werten befüllt.
    // (3)
    const firstName = document.createTextNode(contact.firstName);
    // (4)
    tableCellFirstName.appendChild(firstName);
    // (5)
    const tableCellLastName = document.createElement('td');
    // (6)
    const lastName = document.createTextNode(contact.lastName);
    // (7)
    tableCellLastName.appendChild(lastName);
    // (8)
    const tableCellEmail = document.createElement('td');
    // (9)
```

```
    const email = document.createTextNode(contact.email);
    // (10)
    tableCellEmail.appendChild(email);
    // (11)
    tableRow.appendChild(tableCellFirstName);
    // (12)
    tableRow.appendChild(tableCellLastName);
    // (13)
    tableRow.appendChild(tableCellEmail);
    // (14)
    tableBody.appendChild(tableRow);
  }
}
```

Listing 5.29 Erzeugen einer Tabelle auf Basis der Kontaktliste

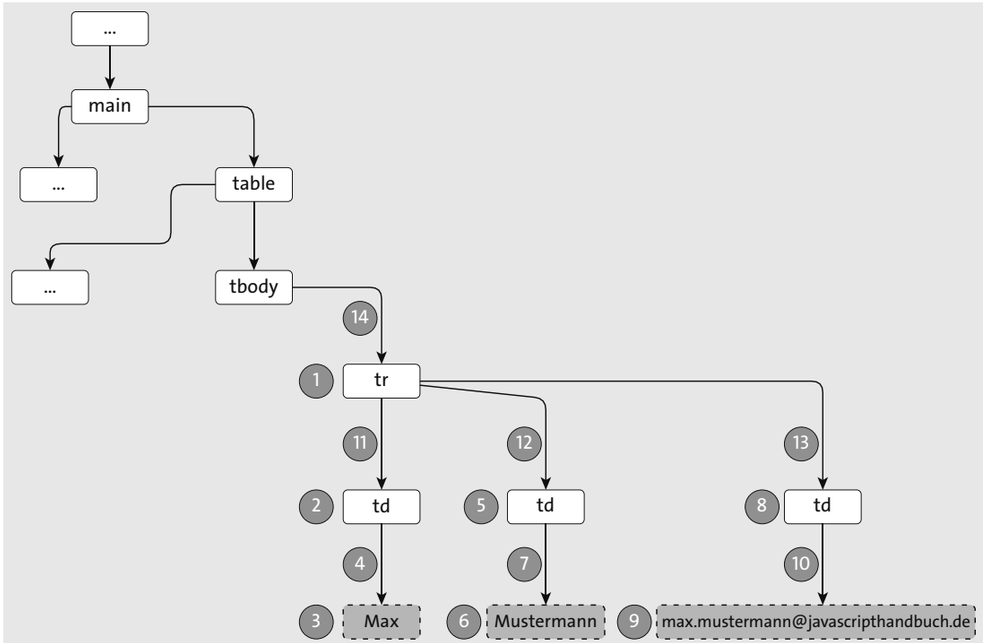

Abbildung 5.27 Reihenfolge der Schritte

```
<!DOCTYPE html>
<html>
<head lang="de">
  <title>Kontaktlistenbeispiel</title>
```



```
</head>
<body onload="createTable()">
<main id="main-content">
  <h1>Kontaktliste</h1>
  <table id="contact-table" summary="Kontaktliste">
    <thead>
    <tr>
      <th>Vorname</th>
      <th>Nachname</th>
      <th>E-Mail-Adresse</th>
    </tr>
    </thead>
    <tbody>
    </tbody>
  </table>
</main>
<script src="scripts/main.js"></script>
</body>
</html>
```

Listing 5.30 Die HTML-Vorlage

5.4.2 Elemente und Knoten entfernen

Um Elemente (bzw. allgemeiner: Knoten) von einem Elternelement (bzw. allgemeiner: einem Elternknoten) zu entfernen, steht Ihnen die Methode `removeChild()` zur Verfügung. Diese Methode erwartet das zu entfernende Element (bzw. den zu entfernenden Knoten) und gibt dieses auch als Rückgabewert zurück. In Listing 5.31 sehen Sie (auf Basis der Listings aus dem vorherigen Abschnitt) eine Methode zur Filterung von Tabellendaten (`sortByFirstName()`), bei der sich die Methode `removeChild()` zunutze gemacht wird, um alle Kindknoten und Kindelemente aus dem Tabellenkörper (also alle Tabellenzeilen) zu entfernen.

```
function sortByFirstName() {
  const tableBody = document.querySelector('#contact-table tbody');
  while (tableBody.firstChild !== null) {
    tableBody.removeChild(tableBody.firstChild);
  }
  contacts.sort(function(contact1, contact2) {
    return contact1.firstName.localeCompare(contact2.firstName);
  })
  createTable();
}
```

Listing 5.31 Beispiel für die Verwendung der Methode »removeChild()«

5.4.3 Die verschiedenen Typen von HTML-Elementen

Jedes HTML-Element wird innerhalb eines DOM-Baums durch einen bestimmten Objekttyp repräsentiert. Welche dies sind, ist in einer Erweiterung der DOM API, der sogenannten DOM-HTML-Spezifikation, festgehalten. Beispielsweise werden Verlinkungen (`<a>`-Elemente) durch den Typ `HTMLAnchorElement` repräsentiert, Tabellen (`<table>`-Elemente) durch den Typ `HTMLTableElement` etc. Eine Übersicht über die verschiedenen HTML-Elemente und ihre entsprechenden Objekttypen gibt Tabelle 5.6. Detaillierte Informationen zu Eigenschaften und Methoden finden Sie dagegen in Abschnitt B.2, »HTML-Interfaces«. Veraltete Typen sind innerhalb der Tabelle kursiv gesetzt.

Der Obertyp »HTMLElement«

Alle Objekttypen haben dabei den gleichen »Obertyp«, den Typ `HTMLElement`, Elemente, die keinen speziellen Typ haben, sind »direkt« vom Typ `HTMLElement`.

Veraltete Elemente

Die Elemente und Objekttypen, die mittlerweile veraltet sind, werden der Vollständigkeit halber noch in kursiver Schrift in der Tabelle mit aufgeführt.

HTML-Element	Typ
`<a>`	`HTMLAnchorElement`
`<abbr>`	`HTMLElement`
`<acronym>`	`HTMLElement`
`<address>`	`HTMLElement`
`<applet>`	*`HTMLAppletElement`*
`<area>`	`HTMLAreaElement`
`<audio>`	`HTMLAudioElement`
`<b>`	`HTMLElement`
`<base>`	`HTMLBaseElement`
`<basefont>`	*`HTMLBaseFontElement`*
`<bdo>`	`HTMLElement`
`<big>`	`HTMLElement`

Tabelle 5.6 Die verschiedenen HTML-Elemente und ihre jeweiligen Typen in JavaScript

HTML-Element	Typ
`<blockquote>`	`HTMLQuoteElement`
`<body>`	`HTMLBodyElement`
` `	`HTMLBRElement`
`<button>`	`HTMLButtonElement`
`<caption>`	`HTMLTableCaptionElement`
`<canvas>`	`HTMLCanvasElement`
`<center>`	`HTMLElement`
`<cite>`	`HTMLElement`
`<code>`	`HTMLElement`
`<col>, <colgroup>`	`HTMLTableColElement`
`<data>`	`HTMLDataElement`
`<datalist>`	`HTMLDataListElement`
`<dd>`	`HTMLElement`
`<del>`	`HTMLModElement`
`<dfn>`	`HTMLElement`
`<dir>`	*`HTMLDirectoryElement`*
`<div>`	`HTMLDivElement`
`<dl>`	`HTMLDListElement`
`<dt>`	`HTMLElement`
`<em>`	`HTMLElement`
`<embed>`	`HTMLEmbedElement`
`<fieldset>`	`HTMLFieldSetElement`
`<font>`	*`HTMLFontElement`*
`<form>`	`HTMLFormElement`
`<frame>`	*`HTMLFrameElement`*
`<frameset>`	*`HTMLFrameSetElement`*

Tabelle 5.6 Die verschiedenen HTML-Elemente und ihre jeweiligen Typen in JavaScript (Forts.)

HTML-Element	Typ
<h1>, <h2>, <h3>, <h4>, <h5>, <h6>	HTMLHeadingElement
<head>	HTMLHeadElement
<hr>	HTMLHRElement
<html>	HTMLHtmlElement
<i>	HTMLElement
<iframe>	HTMLIFrameElement
<img>	HTMLImageElement
<input>	HTMLInputElement
<ins>	HTMLModElement
<isindex>	*HTMLIsIndexElement*
<kbd>	HTMLElement
<keygen>	HTMLKeygenElement
<label>	HTMLLabelElement
<legend>	HTMLLegendElement
	HTMLLIElement
<link>	HTMLLinkElement
<map>	HTMLMapElement
<media>	HTMLMediaElement
<menu>	*HTMLMenuElement*
<meta>	HTMLMetaElement
<meter>	HTMLMeterElement
<noframes>	HTMLElement
<noscript>	HTMLElement
<object>	HTMLObjectElement
	HTMLOListElement
<optgroup>	HTMLOptGroupElement

Tabelle 5.6 Die verschiedenen HTML-Elemente und ihre jeweiligen Typen in JavaScript (Forts.)

HTML-Element	Typ
<option>	HTMLOptionElement
<output>	HTMLOutputElement
<p>	HTMLParagraphElement
<param>	HTMLParamElement
<pre>	HTMLPreElement
<progress>	HTMLProgressElement
<q>	HTMLQuoteElement
<s>	HTMLElement
<samp>	HTMLElement
<script>	HTMLScriptElement
<select>	HTMLSelectElement
<small>	HTMLElement
<source>	HTMLSourceElement
	HTMLSpanElement
<strike>	HTMLElement
	HTMLElement
<style>	HTMLStyleElement
<sub>	HTMLElement
<sup>	HTMLElement
<table>	HTMLTableElement
<tbody>	HTMLTableSectionElement
<td>	HTMLTableCellElement
<textarea>	HTMLTextAreaElement
<tfoot>	HTMLTableSectionElement
<th>	HTMLTableHeaderCellElement
<thead>	HTMLTableSectionElement

Tabelle 5.6 Die verschiedenen HTML-Elemente und ihre jeweiligen Typen in JavaScript (Forts.)

HTML-Element	Typ
<time>	HTMLTimeElement
<title>	HTMLTitleElement
<tr>	HTMLTableRowElement
<tt>	HTMLElement
<u>	HTMLElement
<track>	HTMLTrackElement
	HTMLUListElement
<var>	HTMLElement
	HTMLUnknownElement
<video>	HTMLVideoElement

Tabelle 5.6 Die verschiedenen HTML-Elemente und ihre jeweiligen Typen in JavaScript (Forts.)

Listing 5.32 zeigt das Prinzip dieser Objekttypen am Beispiel von Tabellen, die durch den Typ `HTMLTableElement` repräsentiert werden. Dieser Typ verfügt – wie alle anderen der in Tabelle 5.6 gezeigten Typen – über individuelle, dem Typ entsprechende Eigenschaften: unter anderem die Eigenschaft `caption`, die den Untertitel einer Tabelle enthält (und im Beispiel `null` ist, weil die Tabelle in Listing 5.1 kein `caption`-Attribut hat), die Eigenschaft `tHead`, die den Kopfbereich, sprich das `<thead>`-Element, einer Tabelle enthält, die Eigenschaft `tBodies`, die die verschiedenen Tabellenkörper, sprich `<tbody>`-Elemente, einer Tabelle enthält, die Eigenschaft `rows`, die die Tabellenzeilen (inklusive derjenigen im Kopfbereich) enthält, sowie die Eigenschaft `tFoot`, die den Fußbereich, sprich das `<tfoot>`-Element, enthält.

```
const table = document.querySelector('table');
console.log(Object.getPrototypeOf(table));  // HTMLTableElement
console.log(table.caption);                 // null
console.log(table.tHead);                   // thead
console.log(table.tBodies);                 // [tbody]
console.log(table.rows);                    // [tr, tr, tr, tr, tr]
console.log(table.tFoot);                   // null
```

Listing 5.32 Jedes HTML-Element wird durch einen eigenen Objekttyp repräsentiert, wie hier beispielsweise Tabellen durch den Typ »HTMLTableElement«.

Neben individuellen Eigenschaften haben die verschiedenen Objekttypen auch verschiedene Methoden: Beispielsweise hat der Typ `HTMLTableElement`, wie in Listing 5.33 zu sehen,

unter anderem die Methode `insertRow()`, über die sich (ohne über `document.createElement()` manuell entsprechende HTML-Elemente zu erzeugen) direkt eine neue Tabellenzeile erstellen lässt. Diese gibt wiederum ein Objekt vom Typ `HTMLTableRowElement` zurück, der wiederum unter anderem über die Methode `insertCell()` verfügt, über die sich – Sie werden es ahnen – der entsprechenden Zeile direkt eine neue Tabellenzelle hinzufügen lässt. Im Beispiel wird auf diese Weise eine neue Tabellenzeile mit drei Zellen erstellt. Das ist deutlich übersichtlicher als in Listing 5.29, finden Sie nicht?

```
const newRow   = table.insertRow(1);
const newCellFirstName  = newRow.insertCell(0);
newCellFirstName.textContent = 'Bob';
const newCellLastName  = newRow.insertCell(1);
newCellLastName.textContent = 'Mustermann';
const newCellEmail  = newRow.insertCell(2);
newCellEmail.textContent = 'bob.mustermann@javascripthandbuch.de';
```

Listing 5.33 Die verschiedenen Objekttypen haben unter anderem auch individuelle Methoden.

Eigenschaftsnamen vs. Elementnamen

Beachten Sie, dass Objekteigenschaften wie `tHead`, `tBodies` und `tFoot` in CamelCase-Schreibweise geschrieben sind, die entsprechenden HTML-Elemente dagegen in Kleinbuchstaben (`<thead>`, `<tbody>`, `<tfoot>`).

5.5 Mit Attributen arbeiten

Um mit Attributen zu arbeiten, stehen Ihnen in der DOM API verschiedene Methoden zur Verfügung.

- Über die Methode `getAttribute()` können Sie auf Attribute eines Elements zugreifen (siehe Abschnitt 5.5.1, »Den Wert eines Attributs auslesen«).
- Über die Methode `setAttribute()` können Sie den Wert eines Attributs ändern oder einem Element Attribute hinzufügen (siehe Abschnitt 5.5.2, »Den Wert eines Attributs ändern oder ein neues Attribut hinzufügen«).
- Über die Methode `createAttribute()` können Sie Attributknoten erstellen und diese über `setAttributeNode()` hinzufügen (siehe Abschnitt 5.5.3, »Attributknoten erstellen und hinzufügen«).
- Über die Methode `removeAttribute()` können Sie Attribute entfernen (siehe Abschnitt 5.5.4, »Attribute entfernen«).

5.5.1 Den Wert eines Attributs auslesen

Um auf den Wert eines Attributs zuzugreifen, verwenden Sie auf dem jeweiligen Element die Methode `getAttribute()`. Als Parameter erwartet die Methode den Namen des jeweiligen HTML-Attributs. Als Rückgabewert erhalten Sie den Wert des entsprechenden Attributs. Nehmen Sie als Ausgangspunkt den HTML-Code aus Listing 5.34: Dort ist ein Link gezeigt (ein `<a>`-Element) mit den Attributen `id`, `class` und `href`.

```
<a id="home" class="link" href="index.html">Home</a>
```

Listing 5.34 Ein HTML-Link

Um auf diese Attribute zuzugreifen, verwenden Sie die Methode `getAttribute()`, wie in Listing 5.35 gezeigt.

```
const element = document.getElementById('home');
console.log(element.getAttribute('id'));    // home
console.log(element.getAttribute('class')); // link
console.log(element.getAttribute('href'));  // index.html
```

Listing 5.35 Über die Methode »getAttribute()« können Sie auf Attribute eines HTML-Elements zugreifen.

Die Attribute eines Elements stehen in der Regel auch als gleichnamige Eigenschaften zur Verfügung – wobei das Attribut `class` eine Ausnahme darstellt: Auf dieses Attribut kann über die Eigenschaft `className` zugegriffen werden. Listing 5.36 zeigt dazu ein entsprechendes Beispiel: Die Attribute `id` und `href` können über die gleichnamigen Eigenschaften ausgelesen werden, das Attribut `class` über die Eigenschaft `className`.

```
console.log(element.id);        // home
console.log(element.className); // link
console.log(element.href);      // index.html
```

Listing 5.36 Die Attribute eines Elements stehen auch als Eigenschaften zur Verfügung.

Beachten Sie aber: Bei zwei Attributen liefert der Zugriff über die Methode `getAttribute()` einen anderen Rückgabewert als der direkte Zugriff über die Eigenschaft. Für das Attribut `style` liefert die Methode `getAttribute()` lediglich den Text, den das Attribut als Wert enthält. Der Zugriff über die Eigenschaft `style` dagegen liefert ein Objekt vom Typ `CSSStyleDeclaration`, über das sich detailliert auf die entsprechenden CSS-Informationen zugreifen lässt. Darüber hinaus liefern alle Attribute, über die sich Event-Handler definieren lassen (siehe auch Kapitel 6, »Ereignisse verarbeiten und auslösen«), über die entsprechende Eigenschaft (beispielsweise `onclick`) den auszuführenden JavaScript-Code als Funktionsobjekt zurück. Greift man auf das jeweilige Attribut dagegen über die Methode `getAttribute()` zu,

erhält man als Rückgabewert den Namen der Funktion, die ausgeführt werden soll, als Text zurück.

Schauen Sie sich dazu Listing 5.37 und Listing 5.38 an. Ersteres zeigt einen HTML-Button mit verschiedenen Attributen, unter anderem einem `style`-Attribut und einem `onclick`-Attribut. In Letzterem sehen Sie dann den Zugriff auf beides jeweils über die gleichnamige Eigenschaft und über die Methode `getAttribute()`.

```
<button id="create" class="link" style="background-color: green" onclick=
"createContact()">Kontakt anlegen</button>
```

Listing 5.37 Ein HTML-Button

```
const button = document.getElementById('create');
console.log(button.onclick);                       // Ausgabe der Funktion
console.log(typeof button.onclick);                // Ausgabe: function
console.log(button.getAttribute('onclick'));       // createContact()
console.log(typeof button.getAttribute('onclick')); // Ausgabe: string
console.log(button.style);                         // Ausgabe der
                                                   // CSSStyleDeclaration
console.log(typeof button.style);                  // Ausgabe: object
console.log(button.getAttribute('style'));         // background-color: green
console.log(typeof button.getAttribute('style'));  // Ausgabe: string
```

Listing 5.38 Der Zugriff auf Event-Handler und das »style«-Attribut liefert je nach Zugriffsart unterschiedliche Rückgabewerte.

Der Grund dafür, dass der direkte Zugriff auf Event-Handler-Attribute wie `onclick` keine Zeichenkette, sondern eine Funktion zurückgibt, ist, dass man über diese Eigenschaft Event-Handler für das jeweilige Element definieren kann. Sprich, man kann dieser Eigenschaft Funktionsobjekte zuweisen.

Der Grund dafür, dass der direkte Zugriff auf das `style`-Attribut keine Zeichenkette zurückgibt, ist, dass über dieses Attribut programmatisch auf die CSS-Informationen des jeweiligen Elements zugegriffen werden kann, auch schreibend (wie Sie in diesem Kapitel schon sehen konnten).

5.5.2 Den Wert eines Attributs ändern oder ein neues Attribut hinzufügen

Um den Wert eines Attributs zu ändern oder ein neues Attribut hinzuzufügen, verwenden Sie die Methode `setAttribute()` auf dem Element, für das das Attribut geändert werden soll. Diese erwartet zwei Parameter: den Namen des Attributs und den neuen Wert. Falls das entsprechende Element bereits über ein gleichnamiges Attribut verfügt, wird der Wert dieses Attributs mit dem neuen Wert überschrieben. Gibt es das Attribut noch nicht, wird

dem Element ein entsprechendes Attribut neu hinzugefügt. Listing 5.39 zeigt dazu ein Beispiel: Hier werden die Eigenschaften `class`, `href` und `target` für das zuvor selektierte Linkelement geändert.

```
const element = document.getElementById('home');
element.setAttribute('class', 'link active');
element.setAttribute('href', 'newlink.html');
element.setAttribute('target', '_blank');
console.log(element.getAttribute('class'));   // link active
console.log(element.getAttribute('href'));    // newlink.html
console.log(element.getAttribute('target'));  // _blank
```

Listing 5.39 Über die Methode »setAttribute()« können Sie bestehende Attribute eines HTML-Elements ändern bzw. neue Attribute hinzufügen.

Alternativ dazu können Sie über die (in der Regel) gleichnamigen Objekteigenschaften ebenfalls die Werte von Attributen ändern bzw. neue Attribute hinzufügen (siehe Listing 5.40).

```
element.className = 'link active highlighted';
element.href = 'anotherLink.html';
element.target = '_self';
console.log(element.getAttribute('class'));   // link active highlighted
console.log(element.getAttribute('href'));    // anotherLink.html
console.log(element.getAttribute('target'));  // _self
```

Listing 5.40 Attribute können ebenfalls direkt über entsprechende Eigenschaften geändert werden.

Hinweis

Im Hintergrund wird bei Verwendung der Methode `setAttribute()` ein Attributknoten erzeugt und dem DOM-Baum an dem entsprechenden Elementknoten als Kindknoten hinzugefügt.

5.5.3 Attributknoten erstellen und hinzufügen

Wie auch schon bei normalen Texten und bei der Arbeit mit Elementen haben Sie auch im Fall von Attributen die Möglichkeit, diese als Attributknoten über eine spezielle Methode zu erstellen, nämlich über die Methode `createAttribute()`. Als Argument erwartet diese Methode – wenig verwunderlich – den Namen des zu erstellenden Attributs, als Rückgabewert liefert sie den neuen Attributknoten. Auch dieser ist – wie zuvor Textknoten und Elementknoten – zunächst noch nicht direkt im DOM-Baum eingebaut. Das müssen Sie

manuell über die Methode `setAttributeNode()` am entsprechenden Element nachholen (siehe Listing 5.41).

```
const element = document.getElementById('home');
const attribute = document.createAttribute('target');
attribute.value = '_blank';
element.setAttributeNode(attribute);
console.log(element.getAttribute('target'));  // _blank
```

Listing 5.41 Erstellen und Hinzufügen eines Attributknotens

5.5.4 Attribute entfernen

Über die Methode `removeAttribute()` können Sie Attribute wieder von einem Element entfernen. In Listing 5.42 werden auf diese Weise die beiden Attribute `class` und `href` aus dem Linkelement entfernt. Anschließend liefern die beiden Attribute den Wert `null`.

```
const element = document.getElementById('home');
element.removeAttribute('class');
element.removeAttribute('href');
console.log(element.getAttribute('class'));   // null
console.log(element.getAttribute('href'));    // null
```

Listing 5.42 Über die Methode »removeAttribute()« können Sie Attribute eines HTML-Elements entfernen.

5.5.5 Auf CSS-Klassen zugreifen

Auch wenn Sie es im Laufe des Kapitels schon an einigen Beispielen (zumindest teilweise) gesehen haben, gehe ich an dieser Stelle noch einmal kurz darauf ein, wie Sie die CSS-Klassen eines Elements auslesen können.

Zunächst einmal gibt es die Ihnen schon bekannte Eigenschaft `className`, über die jedes Element auf einer Webseite (sprich jeder Elementknoten) verfügt. Diese Eigenschaft enthält einfach den Wert des `class`-Attributs des entsprechenden Elements als Zeichenkette. Hat das Element mehrere CSS-Klassen, sind diese Klassennamen innerhalb der Zeichenkette durch Leerzeichen getrennt.

In der Vergangenheit hat das teils zu etwas umständlichem Code geführt, wenn man beispielsweise einem Element neue CSS-Klassen hinzufügen oder – schlimmer noch – bestehende CSS-Klassen entfernen wollte. Warum umständlich? Weil man in jedem Fall den Wert des Attributs parsen musste.

Diesem Umstand wurde mit Version 4 der DOM API Rechnung getragen. Seitdem verfügen Elemente (in der DOM API, nicht in HTML) nämlich zusätzlich über die Eigenschaft `class-`

`List`, die die CSS-Klassen als Liste enthält. Das Hinzufügen und Entfernen einzelner CSS-Klassen zu bzw. von einem Element gestaltet sich seitdem um einiges einfacher:

- Über die Methode `add()` können der Liste neue CSS-Klassen hinzugefügt werden.
- Über die Methode `remove()` können CSS-Klassen aus der Liste entfernt werden.
- Über die Methode `toggle()` lassen sich CSS-Klassen »umschalten«, d. h., gibt es die CSS-Klasse in der Liste, wird sie gelöscht, gibt es sie dagegen nicht, wird sie hinzugefügt. Dies lässt sich sogar an boolesche Bedingungen knüpfen.
- Über die Methode `contains()` lässt sich zudem überprüfen, ob eine CSS-Klasse in der Liste enthalten ist.

Listing 5.43 zeigt zu diesen Methoden einige Beispiele.

```
const element = document.getElementById('home');
console.log(element.classList);                    // ["link"]
element.classList.add('active');                   // Klasse hinzufügen
console.log(element.classList);                    // ["link", "active"]
element.classList.remove('active');                // Klasse entfernen
console.log(element.classList);                    // ["link"]
element.classList.toggle('active');                // Klasse umschalten
console.log(element.classList);                    // ["link", "active"]
element.classList.toggle('active');                // Klasse umschalten
console.log(element.classList);                    // ["link"]
console.log(element.classList.contains('link'));   // true
console.log(element.classList.contains('active')); // false
const i = 5;
const condition = i > 0;
element.classList.toggle('active', condition);     // Klasse umschalten
console.log(element.classList);                    // ["link", "active"]
```

Listing 5.43 Mithilfe der Eigenschaft »classList« von Elementen lässt sich sehr einfach mit CSS-Klassen arbeiten.

5.6 Zusammenfassung

In diesem Kapitel haben Sie gelernt, wie Sie auf Inhalte von Webseiten per JavaScript zugreifen und diese dynamisch verändern können. Fassen wir die wichtigsten Punkte zusammen:

- Das *Document Object Model* (kurz *DOM*) stellt das Modell für eine Webseite dar, eine hierarchisch aufgebaute Baumstruktur.
- Die einzelnen Komponenten in dieser Baumstruktur werden *Knoten* genannt, wobei es verschiedene Arten von Knoten gibt. Die wichtigsten sind *Dokumentknoten*, *Elementknoten*, *Textknoten* und *Attributknoten*. Die Elementknoten werden zudem über verschiedene Typen repräsentiert, ausgehend vom Typ `HTMLElement`.

- Die *DOM API* definiert Eigenschaften und Methoden, über die Sie an die Daten auf einer Webseite gelangen oder diese verändern können.
- Sie können mithilfe der DOM API beispielsweise Elemente hinzufügen, Elemente löschen, Texte verändern, Attribute hinzufügen und löschen.
- Elemente auf einer Webseite können auf verschiedene Weisen selektiert werden: nach ID, nach CSS-Klasse, nach Elementname, nach `name`-Attribut sowie nach CSS-Selektor.
- Ausgehend von einem Element bzw. Knoten können über verschiedene Eigenschaften das Elternelement/der Elternknoten, die Kindelemente/Kindknoten sowie die Geschwisterelemente/Geschwisterknoten selektiert werden.
- Über die Eigenschaft `textContent` kann auf den Textinhalt eines Knotens zugegriffen bzw. der Textinhalt gesetzt werden, über die Eigenschaft `innerHTML` kann dagegen auf den HTML-Inhalt eines Elements zugegriffen werden.
- Über `createTextNode()` können Sie Textknoten erstellen, über `createElement()` Elementknoten und über `createAttribute()` Attributknoten.
- Nachdem Sie einen Knoten erstellt haben, müssen Sie ihn erst dem DOM-Baum hinzufügen, wobei verschiedene Methoden zur Verfügung stehen: `insertBefore()`, `appendChild()` und `replaceChild()`.

Kapitel 19
Desktopanwendungen mit JavaScript

Auch Desktopanwendungen lassen sich mittlerweile mit JavaScript implementieren. Im Wesentlichen haben sich in den letzten Jahren dafür zwei Frameworks herausgebildet, die im Folgenden vorgestellt werden sollen.

Webanwendungen sind dank moderner Webtechnologien in den letzten Jahren immer populärer geworden und haben in vielen Fällen klassische Desktopanwendungen verdrängt bzw. warten mit gleichwertigen Lösungen auf (Stichwort *Rich Internet Applications*). Dennoch sind Desktopanwendungen je nach Anwendungsfall bzw. Anforderungen nach wie vor in vielen Fällen sinnvoller.

Zu den Vorteilen von Desktopanwendungen gegenüber Webanwendungen zählen unter anderem:

- **Zugriff auf native Features**: Im Gegensatz zu Webanwendungen, die gar nicht bzw. nur sehr eingeschränkt (beispielsweise über entsprechende Web-APIs oder Browser-Plug-ins) auf native Features und Hardwareressourcen des Rechners zugreifen können, gilt diese Einschränkung für Desktopanwendungen nicht. Klassisches Beispiel hierzu ist der Zugriff auf das Dateisystem: Während man innerhalb einer Webanwendung (über die File API, siehe Abschnitt 12.5, »Auf das Dateisystem zugreifen«) nur auf Dateien zugreifen kann, die durch den Nutzer explizit ausgewählt wurden, hat man innerhalb einer Desktopanwendung (entsprechende Rechte vorausgesetzt) prinzipiell Zugriff auf das gesamte Dateisystem.
- **Kein Aufwand bezüglich Browserversionen**: Bei der Entwicklung von Webanwendungen muss in der Regel ein beachtlicher Teil der Entwicklungszeit dem Thema Browserkompatibilität gewidmet werden. Dazu zählen Fragestellungen wie: Welches Feature wird von welchem Browser unterstützt? Und ab welcher Browserversion? Welche Besonderheiten oder Bugs gibt es in welchem Browser? Wie können Letztere behoben werden? Natürlich können in diesem Zusammenhang Polyfills helfen, sprich Bibliotheken, die für den Fall, dass ein Browser ein bestimmtes Feature nicht unterstützt, dieses Feature emulieren. Auch Cross-Browser-Testing-Tools, die eine Webanwendung automatisch in verschiedenen Konstellationen aus Browser, Version und Betriebssystem testen, stellen eine enorme Hilfe bei der Entwicklung dar. Bei Desktopanwendungen allerdings fällt dieses Thema komplett weg, da man es erst gar nicht mit verschiedenen Browsern bzw. Browserengines zu tun hat (wobei man ehrlicherweise anmerken muss, dass man bei Desktopanwendungen in etwa vergleichbaren Stress mit den verschiedenen Betriebssystemen, Betriebssystemversionen, Architekturen etc. hat).

- **Kein Internetzugang erforderlich**: Webanwendungen setzen eine Verbindung zum Internet voraus. Auch wenn sich dies über Offline-First-Technologien wie Service Worker, IndexedDB und Web Storage weitestgehend minimieren lässt, lassen sich Desktopanwendungen in der Regel deutlich einfacher so gestalten, dass sie auch ohne Internetzugang rundlaufen.
- **Keine Downloadzeit**: Die Komplexität von Webanwendungen und die Anzahl an eingebundenen Fremdbibliotheken und Frameworks wirken sich entsprechend auf die Zeit aus, die es braucht, um die Anwendung initial zu starten. Dauert dies lange, wird hierdurch die Nutzerfreundlichkeit einer Anwendung negativ beeinflusst. Caching-Mechanismen der verschiedenen Browser wirken dem zwar entgegen, für Desktopanwendungen stellt sich dieses Problem aber erst gar nicht. Dennoch: Desktopanwendungen müssen natürlich initial einmal heruntergeladen werden, damit man sie auf dem eigenen Rechner installieren kann.
- **Performance bei hohem Nutzeraufkommen**: Bei Webanwendungen können sich hohe Zugriffszahlen negativ auf die Performance auswirken. Bei Desktopanwendungen spielt die Anzahl an gleichzeitig aktiven Nutzern (zumindest für den UI-Code) keine Rolle. Lediglich für den Fall, dass die Desktopanwendung externe (Web-)Services einbindet, können diese zum Bottleneck der Anwendung werden.

Andersherum bieten Webanwendungen natürlich auch eine Menge Vorteile gegenüber Desktopanwendungen, darunter ein ganz wesentlicher: Cross-Plattform-Fähigkeit, sprich die Fähigkeit einer Anwendung, auf allen Betriebssystemen inklusive mobiler Betriebssysteme (Windows, Linux, macOS, Android, iOS) zu laufen, eine entsprechende Laufzeitumgebung, die in Form eines Browsers daherkommt, vorausgesetzt. Der Aufwand, entsprechende Cross-Plattform-fähige Desktopanwendungen nativ zu erstellen, und die notwendigen Programmierkenntnisse sind im Vergleich entsprechend hoch.

Das ist genau der Punkt, an dem die im Folgenden vorgestellten Frameworks *NW.js* (*https://nwjs.io/*) und *Electron* (*https://www.electronjs.org/*) ansetzen: Sie kombinieren moderne Webtechnologien mit der Möglichkeit, diese für die Erstellung von Desktopanwendungen zu verwenden. Die Ansätze beider Frameworks sind ähnlich, weisen bei genauerer Betrachtung aber Unterschiede auf.

19.1 NW.js

Bei NW.js (*https://nwjs.io/*) handelt es sich um ein Open-Source-Framework zur Erstellung von Desktopanwendungen in HTML, CSS und JavaScript, das 2011 von Intel entwickelt wurde. Die Idee von NW.js besteht darin, die JavaScript-Laufzeitumgebung Node.js mit der Browserengine WebKit zu kombinieren (der ursprüngliche Name lautete daher auch Node WebKit) und plattformunabhängig zur Verfügung zu stellen.

Durch die Kombination von WebKit und Node.js ist es NW.js zum einen möglich, innerhalb eines Desktopfensters Anwendungen darzustellen, die in HTML, CSS und JavaScript implementiert wurden (dies ist möglich durch WebKit), und zum anderen, mit dem zugrunde liegenden Betriebssystem zu interagieren, sprich native Funktionalitäten zu nutzen (dies wiederum ist möglich durch Node.js).

Vereinfacht gesagt, können mithilfe von NW.js Webentwickler, die »nur« HTML, CSS und JavaScript beherrschen, auch Desktopanwendungen implementieren. Und das Ganze funktioniert sogar plattformübergreifend, denn NW.js kümmert sich darum, basierend auf einer einzigen Codebasis in den genannten Sprachen die passenden Anwendungsdateien für die verschiedenen Betriebssysteme Windows, Linux und macOS zu generieren (dazu später mehr).

19.1.1 Installation und Erstellen einer Anwendung

Die Installation von NW.js gestaltet sich recht einfach: Auf der Website des Projekts (*https://nwjs.io/*) finden Sie für alle Betriebssysteme entsprechende Installationsdateien. Für die Entwicklung von NW.js-basierten Anwendungen ist es dabei notwendig, dass Sie die SDK-Version herunterladen (*SDK* für *Software Development Kit*).

Eine minimale NW.js-Anwendung besteht aus zwei bis drei Dateien: einer Konfigurationsdatei, einer HTML-Datei, die den Einstiegspunkt der Anwendung darstellt, sowie üblicherweise mindestens einer JavaScript-Datei, die die Logik der Anwendung enthält. Lassen Sie uns kurz schauen, wie diese drei Dateien für die Beispielanwendung aussehen.

Über die Konfigurationsdatei *package.json* (die man auch als Manifest-Datei bezeichnet) werden wie auch bei Node.js-Packages allgemeine Metadaten zur Anwendung verwaltet, beispielsweise Name, Versionsnummer, die Angabe der Datei, die den Einstiegspunkt für die Anwendung darstellt, sowie externe Abhängigkeiten.

Name, Version und Einstiegspunkt bilden zugleich die Minimalanforderungen an eine *package.json*-Datei für NW.js-Anwendungen. Die Kombination der Eigenschaften `name` und `version` dient dabei als Identifier für die Anwendung, die Eigenschaft `main` gibt an, welche HTML-Datei beim Start geladen wird.

Eine minimale Konfigurationsdatei für eine Anwendung mit Namen *helloworld* in der Version 1.0.0, bei der die Datei *index.html* den Einstiegspunkt markiert, sieht also wie folgt aus:

```
{
  "name": "helloworld",
  "version": "1.0.0",
  "main": "index.html"
}
```

Listing 19.1 Exemplarischer Aufbau der Datei »package.json«

Folgendes Listing zeigt den Inhalt der entsprechenden HTML-Datei. Wie Sie sehen, handelt es sich dabei um normales HTML. Die Logik wiederum wird sinnigerweise in eine separate JavaScript-Datei ausgelagert (dazu gleich mehr).

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello World!</title>
</head>
<body>
  <h1>Hello World!</h1>
  <div>
    Ihr Betriebssystem lautet: <span id="platform"></span>
  </div>
  <script type="text/javascript" src="./scripts/main.js"></script>
</body>
</html>
```

Listing 19.2 Den Einstiegspunkt bildet eine HTML-Datei.

NW.js stellt seine API über ein globales Objekt `nw` zur Verfügung, über das sich beispielsweise UI-Komponenten wie Kontextmenüs oder Ähnliches erstellen lassen. Auf die DOM API und die verschiedenen Node.js-APIs lässt sich dagegen direkt zugreifen. Folgendes Beispiel zeigt eine Kombination der beiden Letztgenannten: Am `document`-Objekt wird hier ein Event-Listener für das `DOMContentLoaded`-Event registriert (DOM API), in dem dann über das Node.js-Modul `os` die Informationen zu der verwendeten Plattform ausgelesen (Node.js-API) und diese Informationen dann (wieder mithilfe der DOM API) in das Element mit der ID `platform` geschrieben werden.

```
const os = require('os');
// Event-Listener registrieren (DOM API)
document.addEventListener('DOMContentLoaded', () => {
  // Informationen zum Betriebssystem (bzw. zur Plattform) auslesen (Node.js-API)
  const platform = os.platform();
  // Informationen in den DOM-Baum schreiben (DOM API)
  document.getElementById('platform').textContent = platform;
});
```

Listing 19.3 Zugriff auf die Node.js-API

Hinweis

Innerhalb einer NW.js-Anwendung lassen sich die vollständige DOM API und die vollständige Node.js-API nutzen (plus zusätzlicher Bibliotheken und Packages, versteht sich). Darüber hinaus erweitert NW.js die DOM API und die Node.js-API um einige Features, beispielsweise um zusätzliche Eigenschaften für Texteingabefelder oder zusätzliche Eigenschaften für das `process`-Objekt.

19.1.2 Starten der Anwendung

Um die oben gezeigte Anwendung zu starten, benötigen Sie das am Anfang dieses Kapitels installierte SDK für NW.js. Gehen Sie dann auf der Kommandozeile in das Wurzelverzeichnis des Projekts und führen Sie den Befehl `<NWJS_TOOLS> .` aus, wobei sich der Platzhalter `<NWJS_TOOLS>` wie im Folgenden gezeigt von Betriebssystem zu Betriebssystem unterscheidet:

```
# Für Windows:
/pfad/zu/nwjs/nw.exe .

# Für Linux
/pfad/zu/nwjs/nw .

# Für macOS
/pfad/zu/nwjs/nwjs.app/Contents/MacOS/nwjs .
```

Listing 19.4 Starten einer NW.js-Anwendung

Nach dem Starten sollte die Anwendung so aussehen wie im folgenden Screenshot. Da ich die Anwendung unter macOS starte, ist im Screenshot der Wert `darwin` zu sehen. Wenn Sie ein anderes Betriebssystem verwenden, steht hier dann natürlich ein anderer Wert.

Abbildung 19.1 Die gestartete NW.js-Anwendung

19.1.3 Packaging der Anwendung

Für das Packaging von NW.js-Anwendungen, also die »Paketierung« von Quelltextdateien zu einer ausführbaren Datei, bietet sich das Package nwjs-builder-phoenix (*https://github.com/evshiron/nwjs-builder-phoenix*) an. Es kann über den Befehl `npm install nwjs-builder-phoenix --save-dev` als Entwicklungsabhängigkeit installiert werden, woraufhin diese entsprechend in der *package.json*-Datei unter dem Bereich `devDependencies` auftaucht (siehe Listing 19.5).

```
{
  "name": "helloworld",
  "version": "1.0.0",
  "main": "index.html",
  "devDependencies": {
    "nwjs-builder-phoenix": "^1.15.0"
  }
}
```

Listing 19.5 Exemplarischer Aufbau der Datei »package.json«

Um nun das Projekt für die verschiedenen Betriebssysteme zu »verpacken«, benötigen Sie in dieser Konfigurationsdatei außerdem noch zwei weitere Einträge: Zum einen müssen Sie über den Bereich `build` konfigurieren, welche Version von NW.js verwendet werden soll. Zum anderen ist es hilfreich, für den doch recht langen `build`-Befehl für das Verpacken der Anwendung einen entsprechenden Eintrag unter `scripts` einzufügen. Auf diese Weise lässt sich der Build-Prozess mit dem sehr viel kürzeren Befehl `npm run dist` aufrufen. Für welche Plattformen Sie die Anwendung paketieren möchten, geben Sie über den Parameter `--tasks` an. Im Beispiel werden auf diese Weise insgesamt drei Packages generiert: eines für Windows, eines für Linux und eines für macOS (jeweils in der 64-Bit-Variante).

```
{
  "name": "helloworld",
  "version": "1.0.0",
  "main": "index.html",
  "scripts": {
    "dist": "build --tasks win-x64,linux-x64,mac-x64 --mirror https://dl.nwjs.io/ .",
  },
  "devDependencies": {
    "nwjs-builder-phoenix": "^1.15.0"
  },
  "build": {
    "nwVersion": "0.14.7"
  }
}
```

Listing 19.6 Die um Build-Informationen und ein Build-Skript erweiterte Datei »package.json«

19.1.4 Weitere Beispielanwendungen

Die gezeigte Beispielanwendung ist natürlich sehr einfach gehalten. Prinzipiell können Sie natürlich Ihren Kenntnissen in HTML, CSS, JavaScript und Node.js freien Lauf lassen und die tollsten Desktopanwendungen implementieren. Ein paar Inspirationen, die zeigen, was alles mit NW.js möglich ist, zeigt Tabelle 19.1.

Name	Art der Anwendung	URL
Facebook Messenger Desktop	Messenger	*https://github.com/Aluxian/Facebook-Messenger-Desktop*
Gitter Desktop	Gitter-Client	*https://github.com/gitterHQ/desktop*
Mango	Markdown-Editor	*https://github.com/egrcc/Mango*
WhatsApp Desktop	Messenger (nicht offiziell)	*https://github.com/bcalik/Whatsapp-Desktop*

Tabelle 19.1 Anwendungen auf Basis von NW.js

19.2 Electron

Das zweite populäre Framework für die Erstellung von Desktopanwendungen in JavaScript ist Electron (*https://electron.atom.io/*). Electron wurde ursprünglich von GitHub im Rahmen der Entwicklung des Codeeditors Atom eingesetzt, im Jahr 2013 aber als separates Framework der Öffentlichkeit zugänglich gemacht.

Vom Prinzip her ist Electron ähnlich wie NW.js. Kein Wunder – einer der ursprünglichen Entwickler von NW.js, der zuvor bei Intel arbeitete, entwickelt mittlerweile für GitHub an Electron weiter. Dennoch gibt es einen wesentlichen Unterschied zwischen NW.js und Electron, der für Entwickler wichtig zu verstehen ist: Während sich in NW.js Node.js und WebKit einen einzelnen JavaScript-Kontext teilen, gibt es in Electron mehrere Kontexte: einen für den Hintergrundprozess, der die Anwendung steuert, und jeweils einen für jedes Anwendungsfenster. Ein weiterer Unterschied besteht in der Definition des Startpunkts einer Anwendung: In NW.js ist dies wie bereits beschrieben eine HTML-Datei, in Electron dagegen eine JavaScript-Datei.

Weitere Unterschiede

Im Gegensatz zu Electron ermöglicht NW.js zudem die Kompilierung des JavaScript-Codes in *nativem* Code, um zu verhindern, dass der Quelltext in lesbarer Form mit der Anwendung ausgeliefert wird. Allerdings läuft der kompilierte Code dann um etwa 30 % langsamer. Darüber hinaus verfügt NW.js über einen integrierten PDF-Viewer und eine integrierte Druckvorschau. Unter Electron dagegen muss man auf externe Bibliotheken wie pdf.js (*https://mozilla.github.io/pdf.js/*) ausweichen.

Electron unterscheidet, wie bereits gesagt, zwischen zwei verschiedenen Arten von Prozessen: Hauptprozess und Renderer-Prozess. Der Hauptprozess (üblicherweise in einer Datei *main.js* enthalten) stellt den Einstiegspunkt für eine Electron-Anwendung dar und kontrol-

liert den Lebenszyklus einer Anwendung. Aus dem Hauptprozess heraus kann beispielsweise auch auf native Komponenten wie etwa auf das Dateisystem zugegriffen werden.

Dem stehen die Renderer-Prozesse gegenüber, die im Wesentlichen ein (Browser-)Fenster innerhalb einer Electron-Anwendung repräsentieren und eine Kombination aus HTML, CSS und JavaScript enthalten. Folglich hat man innerhalb eines Renderer-Prozesses (bzw. des entsprechenden JavaScript-Codes) Zugriff auf das DOM des entsprechenden Fensters. Zusätzlich hat man innerhalb eines Renderer-Prozesses aber auch Zugriff auf die vollständige Node.js-API.

Hinweis

Innerhalb einer Anwendung kann es daher durchaus mehrere Renderer-Prozesse geben, jedoch verständlicherweise nur einen Hauptprozess.

19.2.1 Installation und Erstellen einer Anwendung

Der einfachste Weg, sich ein Electron-Projekt generieren zu lassen, führt über das Git-Repository electron-quick-start, das sich über den Befehl `git clone https://github.com/electron/electron-quick-start` klonen lässt und eine minimale Beispielanwendung enthält, bestehend aus der *package.json*-Datei, einer *.gitignore*-Datei, einer Lizenz- und Readme-Datei sowie folgenden Dateien:

- einer JavaScript-Datei, in der sich der Code für den Hauptprozess befindet, in dem wiederum das Hauptfenster der Anwendung erzeugt wird (*main.js*)
- einer HTML-Datei, die in dieses Hauptfenster geladen wird (*index.html*) und den Code für den Renderer-Prozess dieses Fensters bereitstellt
- einer JavaScript-Datei, die von dieser HTML-Datei eingebunden wird (*renderer.js*) und den Code enthält, der durch den Renderer-Prozess verarbeitet wird (zu Beginn ist diese Datei bis auf einige Kommentare noch leer)

Eine etwas angepasste Version des generierten Hauptprozesscodes (*main.js*) zeigt folgendes Listing. Wie von Node.js-Anwendungen gewohnt, wird das Package electron über die Funktion `require()` eingebunden. Die hierüber importierte Variable `app` repräsentiert dabei die Anwendung, `BrowserWindow` ein einzelnes Anwendungsfenster. Der Code im Listing sorgt dafür, dass, sobald die Anwendung geladen wurde, über die Funktion `createWindow()` ein neues Anwendungsfenster mit einer Breite von 800 Pixeln und einer Höhe von 600 Pixeln erstellt wird.

```
const { app, BrowserWindow } = require('electron');
const path = require('path');
```



```
function createWindow() {
  const mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js')
    }
  })
  mainWindow.loadFile('index.html');
}

app.whenReady().then(() => {
  createWindow();

  app.on('activate', () => {
    if (BrowserWindow.getAllWindows().length === 0) {
      createWindow();
    }
  })
});

app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') {
    app.quit();
  }
});
```

Listing 19.7 Code für den Hauptprozess

Über den Aufruf von `loadFile()` wiederum wird die angegebene HTML-Datei *index.html* in das erstellte Anwendungsfenster geladen. Der Code dieser Datei ist in folgendem Listing zu sehen:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="Content-Security-Policy" content="default-src 'self'; ↩
script-src 'self'">
    <title>Hello World!</title>
  </head>
  <body>
```

```
    <h1>Hello World!</h1>
    <div>Node.js Version: <span id="node-version"></span></div>
    <div>Chromium Version: <span id="chrome-version"></span></div>
    <div>Electron Version: <span id="electron-version"></span></div>
    <script src="./renderer.js"></script>
  </body>
</html>
```

Listing 19.8 Code für das Hauptfenster

Darüber hinaus wird beim Aufruf des Konstruktors von `BrowserWindow` eine weitere JavaScript-Datei geladen (*preload.js*), die wiederum dafür sorgt, dass die Informationen zu den verwendeten Versionen von Chrome, Node.js sowie Electron in die dafür vorgesehenen HTML-Platzhalterelemente geschrieben werden:

```
window.addEventListener('DOMContentLoaded', () => {
  const replaceText = (selector, text) => {
    const element = document.getElementById(selector);
    if (element) {
      element.innerText = text;
    }
  }

  for (const type of ['chrome', 'node', 'electron']) {
    replaceText(`${type}-version`, process.versions[type]);
  }
})
```

Listing 19.9 Inhalt der Datei preload.js

Schauen wir uns als Nächstes an, wie sich die gezeigte Beispielanwendung starten lässt. Doch bevor dies möglich ist, müssen Sie zunächst über den Befehl `npm install` die benötigten Abhängigkeiten installieren.

19.2.2 Starten der Anwendung

Das Starten der Anwendung ist nicht sonderlich schwer: Es reicht, den Befehl `npm start` auszuführen, den Rest erledigt Electron im Hintergrund. Anschließend öffnet sich das Anwendungsfenster, das wir vorhin über die Funktion `createWindow()` erstellt hatten und das für das Beispiel wie folgt aussieht:

Abbildung 19.2 Die gestartete Electron-Anwendung

19.2.3 Packaging

Genau wie mit NW.js lassen sich Anwendungen auch mit Electron für verschiedene Betriebssysteme paketieren. Eine Möglichkeit führt über das Package electron-packager (*https://github.com/electron-userland/electron-packager*). Dieses Package lässt sich sowohl programmatisch als auch über die Kommandozeile aufrufen. Für Ersteres reicht die lokale Installation über den Befehl `npm install electron-packager --save-dev`, für Letzteres muss das Modul über den Befehl `npm install -g electron-packager` global installiert werden.

Auf der Kommandozeile startet man das Packaging über den Befehl `electron-packager`, wobei als Parameter das Quellverzeichnis, der Name der Anwendung, die Zielplattformen (`darwin`, `linux`, `mas`, `win32`) sowie deren Architektur (`ia32`, `x64`, `armv7l`) anzugeben sind. Der Befehl `electron-packager . hello-world --platform=win32,linux,darwin --arch=x64` beispielsweise paketiert die Anwendung für Windows, Linux und macOS in 64 Bit.

Da der Befehl für das Paketieren der Anwendung unter Umständen verhältnismäßig lang werden kann, bietet es sich an – wie schon zuvor bei NW.js –, ein entsprechendes Skript in die *package.json*-Datei einzufügen (siehe Listing 19.10). Anschließend lässt sich der Build-Prozess über ein einfaches `npm run dist` aufrufen.

```
{
  "name": "electron-helloworld",
  "version": "1.0.0",
  "main": "main.js",
  "scripts": {
    "start": "electron .",
    "dist": "electron-packager . hello-world --platform=win32,linux,darwin --arch=x64"
  },
  "devDependencies": {
    "electron": "^13.1.5"
  }
}
```

Listing 19.10 Konfigurationsdatei »package.json« für die Electron-Beispielanwendung

19.2.4 Weitere Beispielanwendungen

Die Auswahl an Beispielanwendungen, die mithilfe von Electron erstellt wurden, ist deutlich umfangreicher als die der Anwendungen, die mithilfe von NW.js erstellt wurden. Gefühlt ist Electron also das beliebtere Framework (was man im Übrigen auch an der Anzahl der »Stargazer« bei GitHub erkennt).

Eine kleine Auswahl von Electron-basierten Anwendungen zeigt Tabelle 19.2. Wie Sie sehen, befinden sich darunter auch einige, die man als Webentwickler vielleicht bereits tagtäglich nutzt (z. B. Atom, Postman, Slack oder Visual Studio Code), bei denen man sich aber vielleicht nicht der zugrunde liegenden Technologie bewusst war.

Name	Art der Anwendung	URL
Abricotine	Markdown-Editor	*https://github.com/brrd/Abricotine*
Atom	Codeeditor	*https://github.com/atom/atom*
Caret	Markdown-Editor	*http://caret.io/*
Flow	Task-Management	*https://www.getflow.com/*
GitKraken	Git-Client	*https://www.gitkraken.com/*
Kitematic	Docker Container Management	*https://kitematic.com/*
Min	Webbrowser	*https://github.com/minbrowser/min*
Mongotron	MongoDB-Management-Tool	*https://github.com/officert/mongotron*
Nocturn	Twitter-Client	*https://github.com/k0kubun/Nocturn*
Nylas N1	E-Mail-Client	*https://www.nylas.com/*
Playback	Videoplayer	*https://github.com/mafintosh/playback*
Postman	REST- bzw. HTTP-Client	*https://www.getpostman.com/*
Slack	Desktopanwendung für Slack	*https://slack.com/downloads*
Space Radar	Festplatten-visualisierung	*https://github.com/zz85/space-radar*
Visual Studio Code	Codeeditor	*https://github.com/Microsoft/vscode*

Tabelle 19.2 Anwendungen auf Basis von Electron

Name	Art der Anwendung	URL
Wagon	SQL-Analysetool	*http://www.wagonhq.com/*
WebTorrent	Torrent-Client	*https://github.com/feross/webtorrent-desktop*
WhatsApp	Messenger	*https://www.whatsapp.com/download/*
WordPress Desktop	CMS	*https://desktop.wordpress.com/*

Tabelle 19.2 Anwendungen auf Basis von Electron (Forts.)

19.3 Zusammenfassung

Mit NW.js und Electron ist es relativ einfach, plattformunabhängige Desktopanwendungen auf Basis von HTML, CSS und JavaScript zu erstellen.

Folgende Punkte sollten Sie sich aus diesem Kapitel merken:

- NW.js und Electron funktionieren vom Prinzip her ähnlich, indem sie WebKit und Node.js miteinander kombinieren.
- Beide Frameworks ermöglichen es, sowohl die DOM API als auch die verschiedenen APIs von Node.js zu verwenden.
- Der Einstiegspunkt bei NW.js ist eine HTML-Datei, bei Electron eine JavaScript-Datei.
- Während sich in NW.js Node.js und WebKit einen einzelnen JavaScript-Kontext teilen, gibt es in Electron mehrere Kontexte, weil hier zwischen Hauptprozess und Renderer-Prozessen unterschieden wird.
- Sowohl NW.js als auch Electron erlauben es, Anwendungen für verschiedene Betriebssysteme und Architekturen zu paketieren.