JS

2. Auflage

DOM, Node.js, jQuery, Web-APIs, Ajax, mobile Anwendungen, ECMAScript

```
function addEvent(element, eventType, eventHandler)
{
 if (window.addEventListener) {
  element.addEventListener(
   eventType, eventHandler, false
  );
 }
 else if (window.attachEvent) {
  element.attachEvent('on' + eventType, eventHandler);
 }
 else {
  element['on' + seventType] = eventHandler;
 }
}
```

Philip Ackermann

JavaScript

Das umfassende Handbuch

- Grundlagen, Anwendung, Referenz
- OOP, aktuelle ECMAScript-Features, mobile Anwendungen
- Inkl. Web-APIs, Node.js und Internet of Things

Mit allen Beispielen zum Download

Rheinwerk Computing

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 gewissermaßen 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 Stylesheets*) innerhalb einer Webseite steht, wie man JavaScript in eine Webseite einbindet und wie man Ausgaben erzeugen kann.

2.1 Einführung JavaScript und 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 *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, die 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 es 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 zu sehen, 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. Wobei in so einem Fall die Webseite nur weniger schick (ohne CSS) und weniger interaktiv und nutzerfreundlich (ohne JavaScript) ist (siehe wiederum Abbildung 2.1).

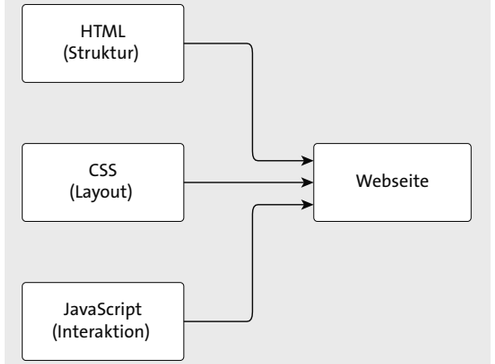

Abbildung 2.5 In der Regel wird innerhalb einer Webseite eine Kombination aus HTML, CSS und JavaScript verwendet.

Merke

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

Definition

Web- und Software-Entwickler 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.

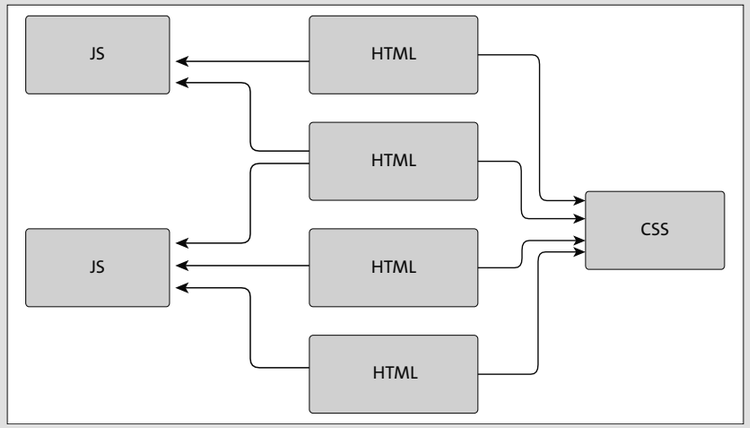

Abbildung 2.6 Wenn Sie CSS- und JavaScript-Code nicht direkt in den HTML-Code schreiben, sondern in separate Dateien, erleichtert das die Wiederverwendbarkeit.

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 ein Webdesigner sich 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 auch *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 auch ein einfacher Texteditor ausreichen (und für einfache Codebeispiele ist dies auch durchaus in Ordnung). Was Sie sich allerdings früher oder später zulegen sollten, ist ein guter Editor, 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 dahingehend, 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*),

Coda 2 (*https://panic.com/coda/*) oder auch die noch jüngeren Editoren Atom (*https://atom.io*) oder Microsoft Visual Studio Code (*https://code.visualstudio.com*) beliebt.

Sublime Text (siehe Abbildung 2.7) kostet 70 US$ und steht für Windows, macOS und Linux zur Verfügung. Coda 2 (siehe Abbildung 2.8) kostet 99 US$, ist allerdings nur für macOS verfügbar. Der kostenlose, relativ junge Editor Atom (siehe Abbildung 2.9) dagegen ist wie Sublime Text für alle drei Betriebssysteme verfügbar, ebenso der Editor Visual Studio Code von Microsoft (siehe Abbildung 2.10).

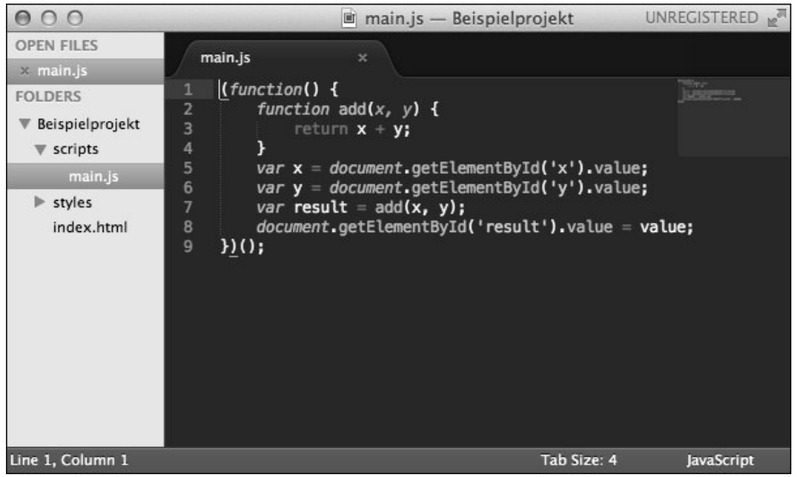

Abbildung 2.7 Der Editor Sublime Text

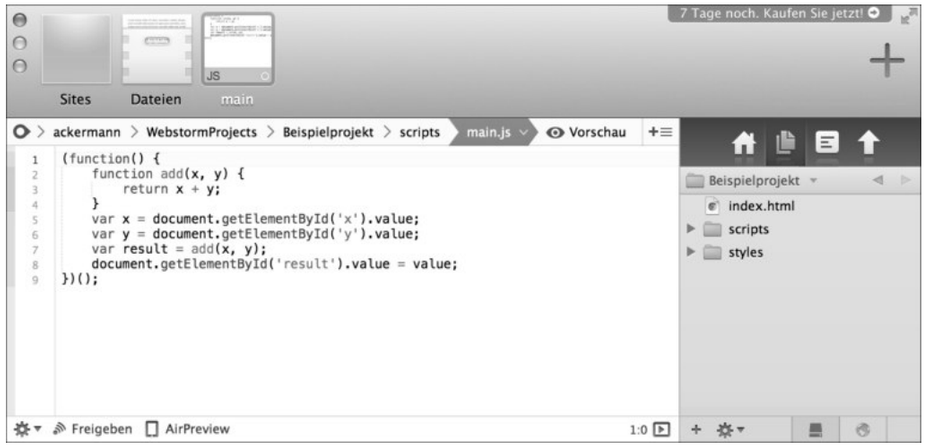

Abbildung 2.8 Der Editor Coda 2

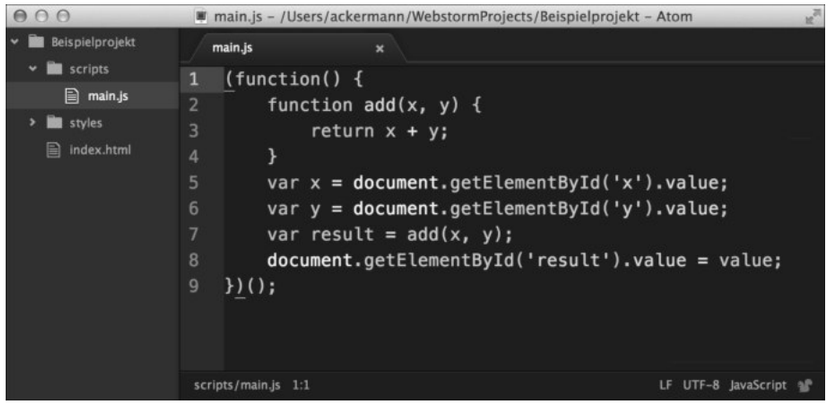

Abbildung 2.9 Der Editor Atom

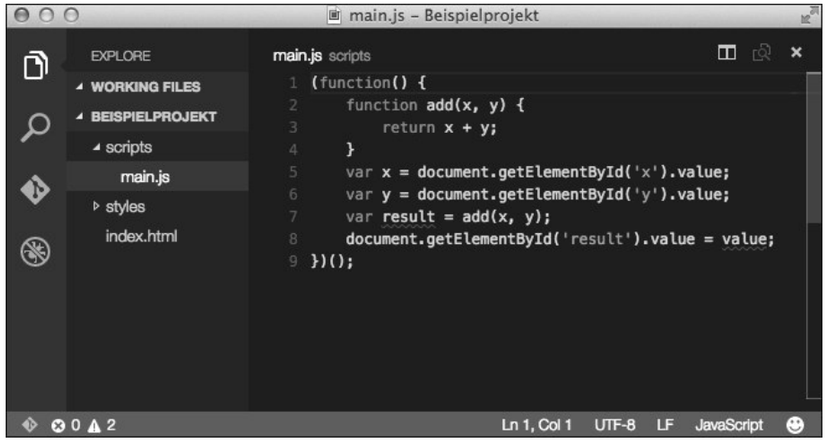

Abbildung 2.10 Der Editor Microsoft Visual Studio Code

Im Detail haben alle der genannten Editoren ihre eigenen Features und Stärken, sind prinzipiell aber doch recht ähnlich. Probieren Sie einfach aus, welcher Ihnen am meisten zusagt (für Sublime Text und Coda 2 stehen übrigens auf den jeweiligen Homepages kostenlose Testversionen zum Download bereit).

Entwicklungsumgebungen

Software-Entwickler, 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 vor-

stellen, der gegenüber einem »normalen« Editor noch diverse andere Features bereitstellt, wie beispielsweise die Synchronisation mit einem *Sourceverwaltungssystem*, 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 20, »Mikrocontroller mit JavaScript steuern«, da gehe ich auf diese fortgeschrittenen Themen der Software-Entwicklung mit JavaScript ein.)

WebStorm von IntelliJ (*www.jetbrains.com/webstorm/*, siehe Abbildung 2.11) ist ein Beispiel für eine sehr beliebte und, wie ich finde, auch wirklich sehr gute Entwicklungsumgebung, die ich persönlich auch im beruflichen Alltag nutze. Eine Einzellizenz für WebStorm kostet 99 €. Wer das Programm zunächst testen möchte, kann eine 30-Tage-Testversion von der Homepage herunterladen. WebStorm steht dabei sowohl für Windows als auch für macOS und Linux zur Verfügung.

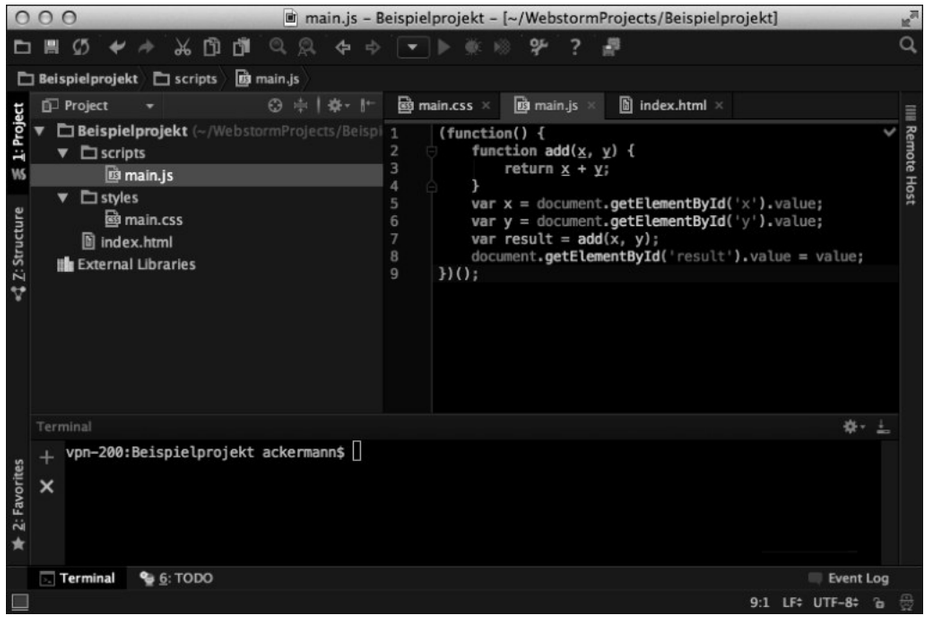

Abbildung 2.11 Die WebStorm-IDE

Als kostenlose Alternative dazu kann ich Ihnen die NetBeans IDE (*https://netbeans.org*, siehe Abbildung 2.12) empfehlen, die ursprünglich hauptsächlich für die Java-Entwicklung verwendet wird, aber auch mit JavaScript gut umgehen kann. NetBeans steht ebenfalls für alle drei genannten Betriebssysteme zur Verfügung und kann entsprechend von der Homepage heruntergeladen werden.

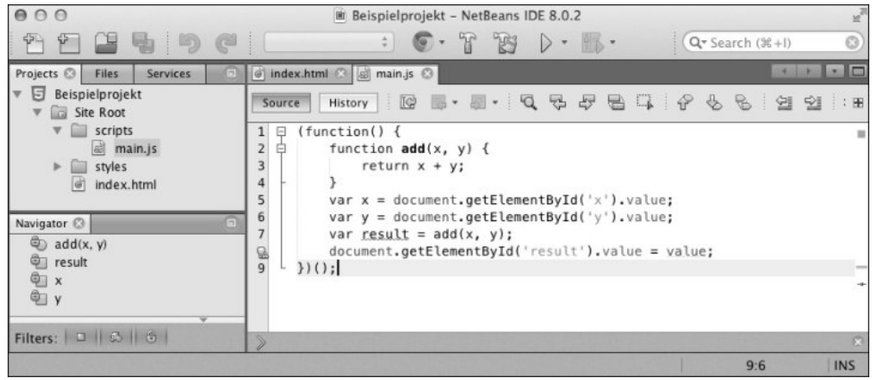

Abbildung 2.12 Die NetBeans IDE

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	70 US$	ja	ja	ja	Editor
Coda 2	99 US$	ja	nein	nein	Editor
Atom	kostenlos	ja	ja	ja	Editor
Microsoft Visual Studio Code	kostenlos	ja	ja	ja	Editor
WebStorm	99 €	ja	ja	ja	Entwicklungsumgebung
NetBeans	kostenlos	ja	ja	ja	Entwicklungsumgebung

Tabelle 2.1 Empfehlenswerte Editoren und Entwicklungsumgebungen für die Entwicklung mit JavaScript

Tipp

Für den Anfang 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 machen Entwicklungsumgebungen eigentlich auch erst ab einer gewissen Projektgröße Sinn, 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.

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, welches 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, welches 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, welche auch als *DOM-Baum* bezeichnet wird. Ein DOM-Baum wiederum setzt sich aus sogenannten *Knoten* (engl.: *Nodes*) zusammen, welche 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 Falle vom DOM steht das Ausgangselement (die Wurzel) ganz oben, und der Baum »wächst« von dort nach unten.

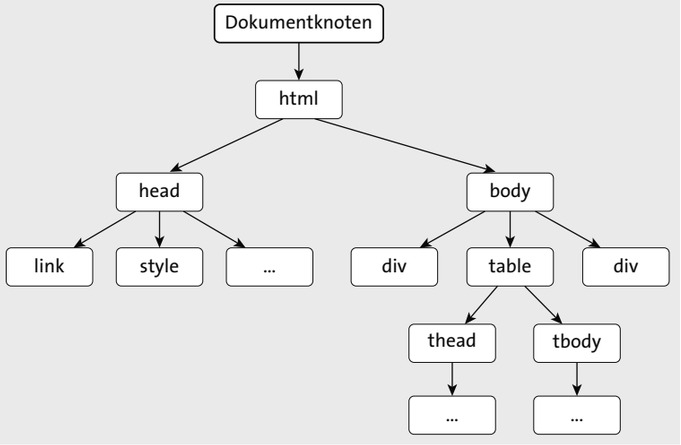

Abbildung 5.1 Aufbau eines DOM-Baumes

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 welcher Sie den HTML-Code für eine einfache Tabelle zur Darstellung einer Kontaktliste sehen. Das entsprechende Document Object Model ist in Abbildung 5.2 dargestellt (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-Baumes. Er wird durch das globale Objekt `document` repräsentiert, welches 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.

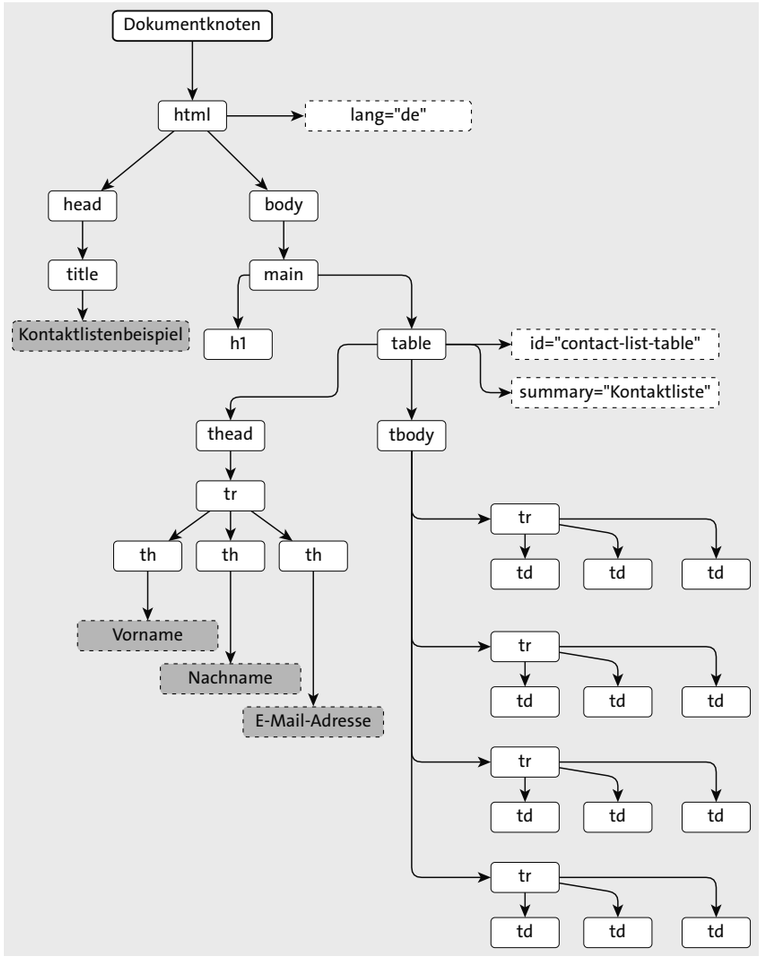

Abbildung 5.2 Aufbau des DOM-Baumes für das Beispiel

- *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>`.
- *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, die sogenannten *Textknoten* (in Abbildung 5.2 gestrichelt umrandet und grau eingefärbt). Im Beispiel sind das beispielsweise die Knoten für die Texte `Kontaktlistenbeispiel`,

`Kontaktliste`, `Vorname`, `Nachname` und `E-Mail-Adresse`. Textknoten können selbst keine Kindknoten haben und sind damit zwangsweise 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. Sie können das testen, indem Sie innerhalb des DOM-Baumes 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.

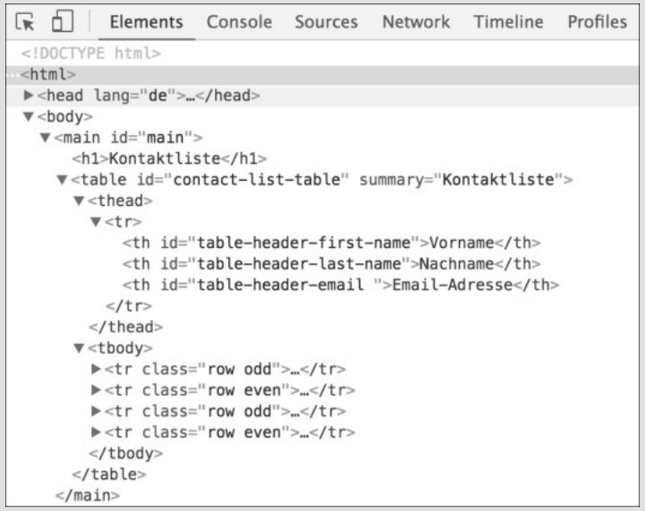

Abbildung 5.3 Darstellung des DOM in den Chrome Developer Tools

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, welches 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 einen 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 (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. domino (*https://github.com/fgnass/domino*), mit denen man Webseiten parsen kann, lässt sich unter Node.js ein Document Object Model einer Webseite erstellen.

Der Aufbau des Document Object Models, 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 Software-System 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* (weil man ja auch *das Application Programming Interface* sagen würde). Es ist aber auch durchaus üblich, den Artikel nach der deutschen Übersetzung *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, welches Sie ändern bzw. an welches 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«

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

Eigenschaft/ Methode	Beschreibung	Rückgabe-wert	Abschnitt
`getElementsBy-Name()`	Wählt Elemente anhand ihres Namens aus.	Liste von Elementen	Abschnitt 5.2.4, »Elemente nach Namen selektieren«
`querySelector()`	Gibt das erste Element zurück, das auf einen gegebenen CSS-Selektor passt.	einzelnes Element	Abschnitt 5.2.5, »Elemente per Selektor selektieren«
`querySelector-All()`	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«
`previousElement-Sibling`	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«
`nextElement-Sibling`	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«
`firstElement-Child`	Gibt für einen Knoten das erste Kindelement zurück.	einzelnes Element	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	Rückgabe-wert	Abschnitt
`firstChild`	Gibt für einen Knoten den ersten Kindknoten zurück.	einzelner Knoten	Abschnitt 5.2.7, »Die Kindelemente eines Elements selektieren«
`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, welches 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).

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

Listing 5.2 Zugriff auf ein Element über die ID

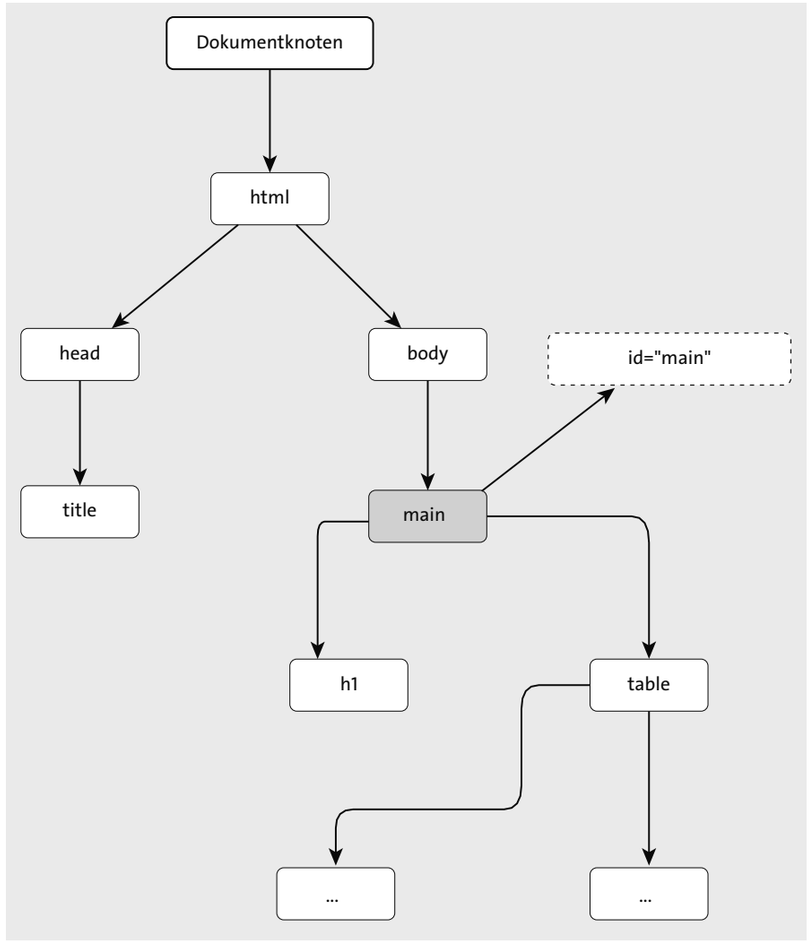

Abbildung 5.4 Mit »getElementById()« wird maximal ein Element selektiert.

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 welche 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 vorgehen.

```
let 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 nicht der Fall), wird nicht auf die Variable zugegriffen.

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 einer 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 diesen Unterschied doch im Hinterkopf haben.

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 5.4 zu sehen. In diesem Beispiel werden alle Elemente selektiert, die die CSS-Klasse `even` enthalten, sprich die beiden »geraden« Tabellenzeilen (siehe Abbildung 5.6).

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

Listing 5.4 Zugriff auf ein Element über Klassennamen

Der Rückgabewert von `getElementsByClassName()` ist eine *Knotenliste* (genauer gesagt, ein Objekt vom Typ `NodeList`), welche ähnlich wie ein Array zu verwenden ist (bei der es sich aber um kein 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 an 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 gemeinsam ist die Eigenschaft `length`, über die sich die Anzahl an Elementen in der Knotenliste herausfinden lässt.

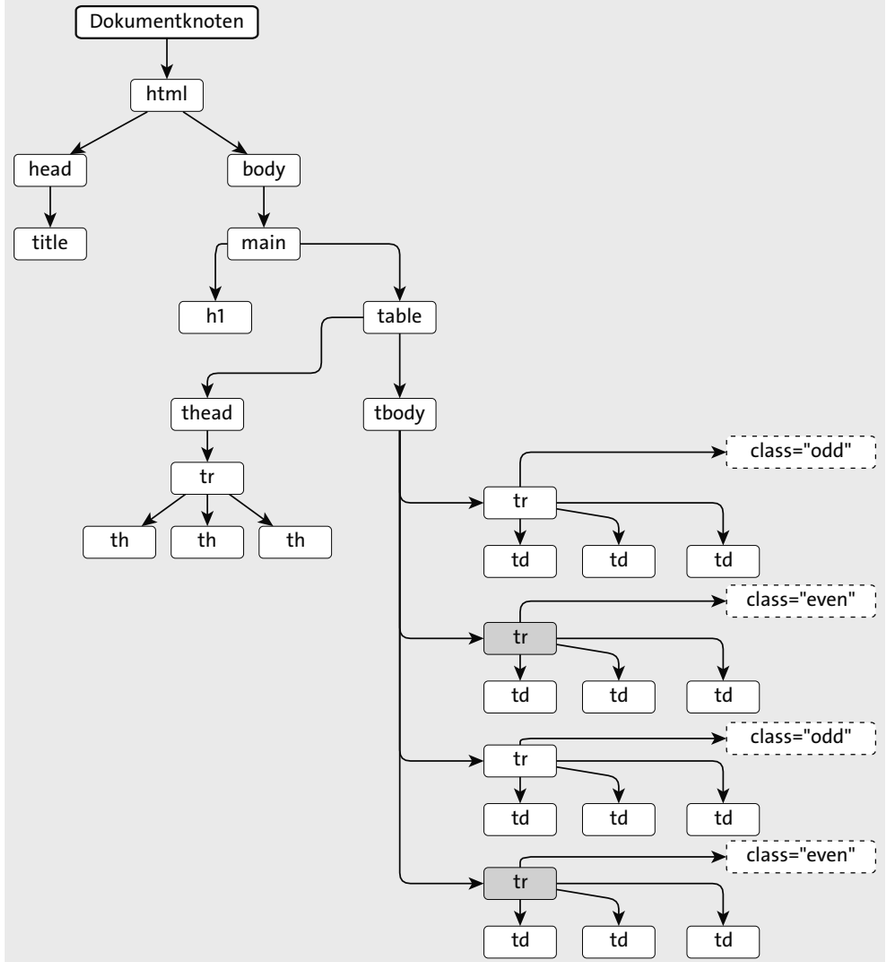

Abbildung 5.6 Die Methode »getElementsByClassName()« kann mehrere Elemente zurückgeben.

Um also beispielsweise über alle Elemente einer Knotenliste zu iterieren, geht man wie in Listing 5.5 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).

```
let 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.
    let tableRow = tableRowsEven[i];            // Weise Element einer Variablen zu.
    tableRow.style.backgroundColor = '#CCCCCC'; // Setze neue Hintergrundfarbe.
  }
}
```

Listing 5.5 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 wird 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 dieser Eigenschaft hinterlegte Objekt enthält alle CSS-Eigenschaften als Objekteigenschaften (also beispielsweise `style.color`, `style.border` usw.). 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 auch über die Methode `item()` auf einzelne Knoten einer Knotenliste zugreifen. Auch hier übergeben Sie als Argument den Index des Elements, welches zurückgegeben werden soll. Die Schleife von eben ließe sich also auch wie folgt umformulieren:

```
let 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.
    let tableRow = tableRowsEven.item(i);       // Weise Element einer Variablen zu.
    tableRow.style.backgroundColor = '#CCCCCC'; // Setze neue Hintergrundfarbe.
  }
}
```

Listing 5.6 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 Borrowings* (siehe Kapitel 4, »Mit Objekten und Referenztypen arbeiten«), um dennoch Methoden von `Array` verwenden zu können (siehe Listing 5.7).

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

Listing 5.7 Iteration über eine Knotenliste über Method Borrowing

Aktive Knotenlisten vs. statische Knotenlisten

Man unterscheidet bei Knotenlisten zwischen sogenannten *aktiven* und *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 5.8 vor.

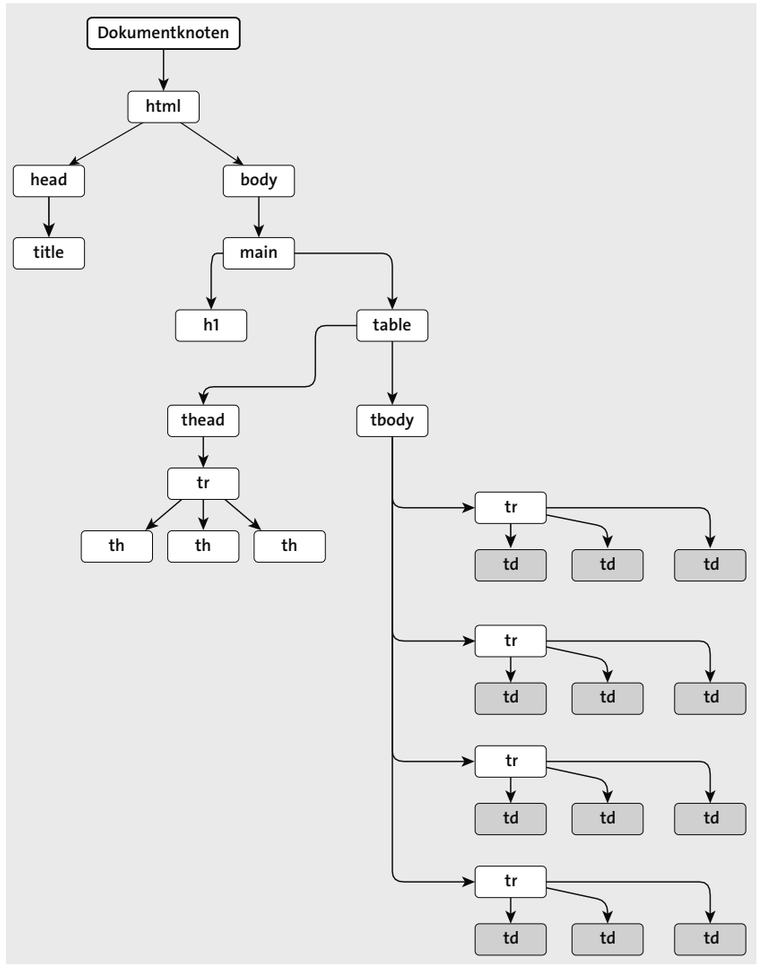

Abbildung 5.8 Die Methode »getElementsByTagName()« selektiert Elemente nach ihrem Elementnamen.

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

```
let 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.
    let tableCell = tableCells[i];           // Weise Element einer Variablen zu.
```



```
    tableCell.style.fontFamily = 'Verdana';  // Setze neue Schriftart.
    tableCell.style.fontSize = '9pt';        // Setze neue Schriftgröße.
  }
}
```

Listing 5.8 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 zu einer Auswahlgruppe zu kennzeichnen. In Listing 5.9 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 5.9 Ein einfaches HTML-Formular

Mithilfe der Methode `getElementsByName()` können Elemente ausgehend von diesem `name`-Attribut selektiert werden. In Listing 5.10 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).

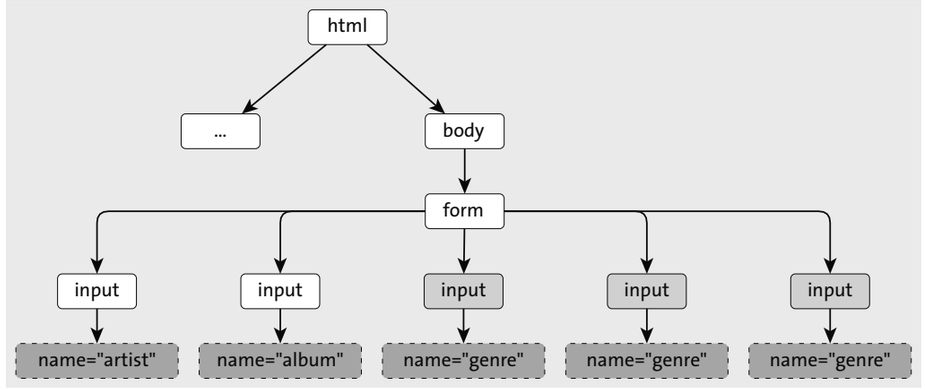

Abbildung 5.10 Die Methode »getElementsByName()« selektiert Elemente nach ihrem »name«-Attribut.

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

```
let 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.
    let inputElement = inputElementsForGenre[i];  // Weise Element einer
                                                  // Variablen zu.
```



```
    console.log(inputElement.value);              // Ausgabe: Stonerrock,
                                                  // Spacerock, Hardrock
  }
}
```

Listing 5.10 Zugriff auf Elemente über Elementnamen

Browsersupport von »getElementsByName()«

Die Methode `getElementsByName()` funktioniert nicht in allen Browsern konsistent. In einigen Versionen des Internet Explorers und des Opera-Browsers beispielsweise liefert die Methode nicht nur solche Elemente zurück, deren `name`-Attribut mit dem übergebenen Wert übereinstimmt, sondern auch solche Elemente, 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, welches 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*, welches auf den entsprechenden CSS-Selektor zutrifft. Letztere Methode liefert *alle Elemente*, die auf den übergebenen CSS-Selektor zutreffen.

Listing 5.11 zeigt ein Beispiel für die Verwendung von `querySelector()`. Übergeben wird hier der CSS-Selektor `#main table td`, welcher in CSS zunächst die zweiten Tabellenzellen jeder Zeile (`td:nth-child(2)`) innerhalb einer Tabelle (`table`) innerhalb eines Elements mit 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.

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

Listing 5.11 Zugriff auf ein Element über CSS-Selektor

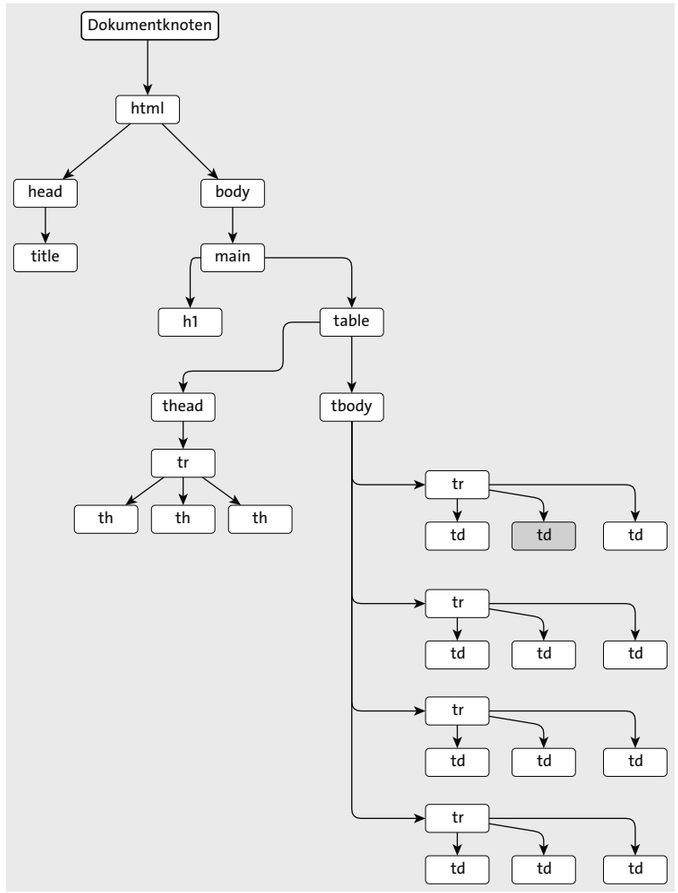

Abbildung 5.11 Die Methode »querySelector()« liefert maximal ein Element zurück.

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.

Listing 5.12 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).

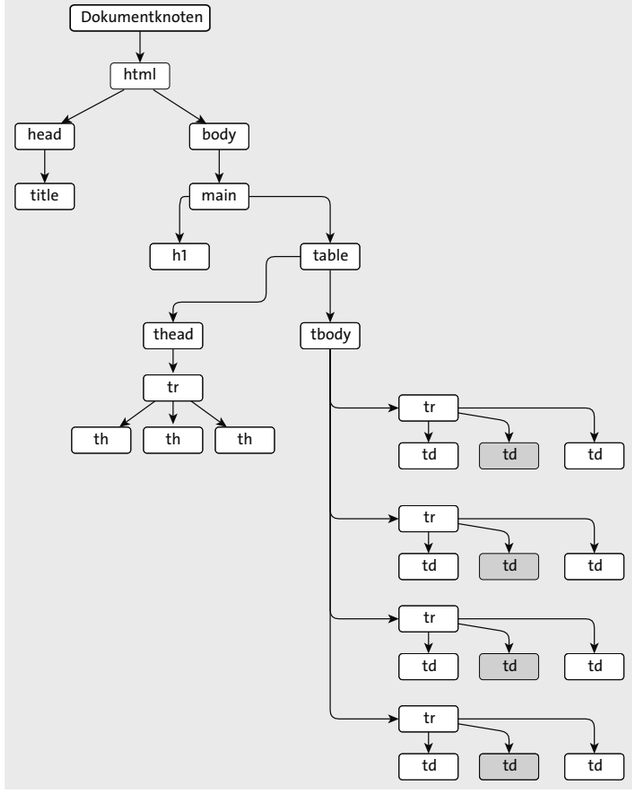

Abbildung 5.13 Die Methode »querySelectorAll()« kann mehrere Elemente zurückgeben.

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

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

Listing 5.12 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*), welche u. a. diese beiden wichtigen Methoden definiert, musste man mit den anderen, vorhin vorgestellten DOM-Methoden zur Selektion von Elementen vorliebnehmen.

Die Bibliothek jQuery hat diese Einschränkung schon frühzeitig erkannt und entsprechende Helferfunktionen bereits recht früh zur Verfügung gestellt. In Kapitel 10, »Aufgaben vereinfachen mit jQuery«, werden wir u. a. 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 Attribut `a`.	2
`[a="b"]`	Selektiert Elemente mit Attribut `a`, welches den Wert `b` hat.	2
`[a~="b"]`	Selektiert Elemente mit Attribut `a`, welches als Wert eine Liste von Werten hat, von denen einer gleich `b` ist.	2

Tabelle 5.3 Die verschiedenen Selektoren in CSS3

Selektor	Beschreibung	Seit CSS-Version
`[a^="b"]`	Selektiert Elemente mit Attribut `a`, dessen Wert mit `b` beginnt.	3
`[a$="b"]`	Selektiert Elemente mit Attribut `a`, dessen Wert mit `b` endet.	3
`[a*="b"]`	Selektiert Elemente mit 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
`: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

Tabelle 5.3 Die verschiedenen Selektoren in CSS3 (Forts.)

Selektor	Beschreibung	Seit CSS-Version
`: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
`: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

Tabelle 5.3 Die verschiedenen Selektoren in CSS3 (Forts.)

Selektor	Beschreibung	Seit CSS-Version
`E > F`	Selektiert Elemente vom Typ `F`, die Kindelemente eines Elements vom Typ `E` sind.	2
`E + F`	Selektiert Elemente vom Typ `F`, die direkte 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 5.13 und Abbildung 5.15).

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

Listing 5.13 Zugriff auf Elternknoten bzw. Elternelement

Knoten und Elemente

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

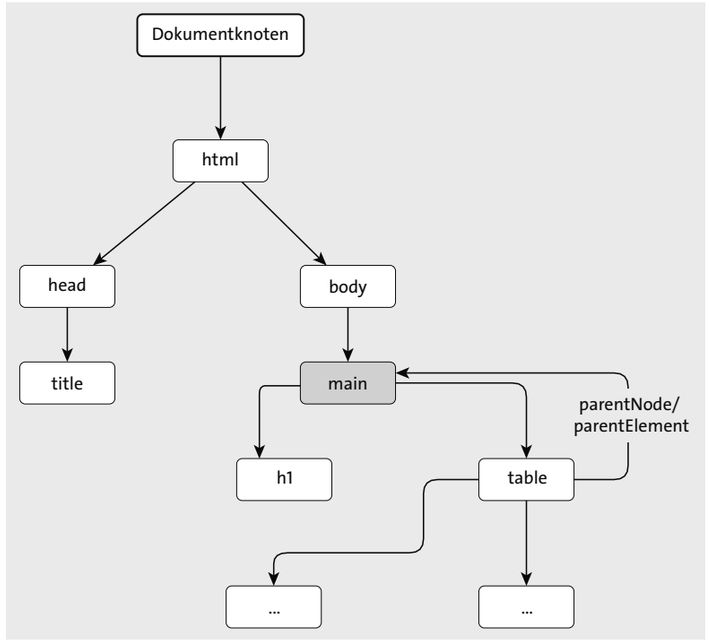

Abbildung 5.15 Selektion des Elternelements

Wichtig zu verstehen ist, dass einige der oben genannten Eigenschaften Knoten zurückgeben, andere Eigenschaften dagegen Elemente zurückgeben. 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 5.14 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 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 nun in den folgenden Abschnitten eingehen.

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

Listing 5.14 Ein einfaches HTML-Beispiel

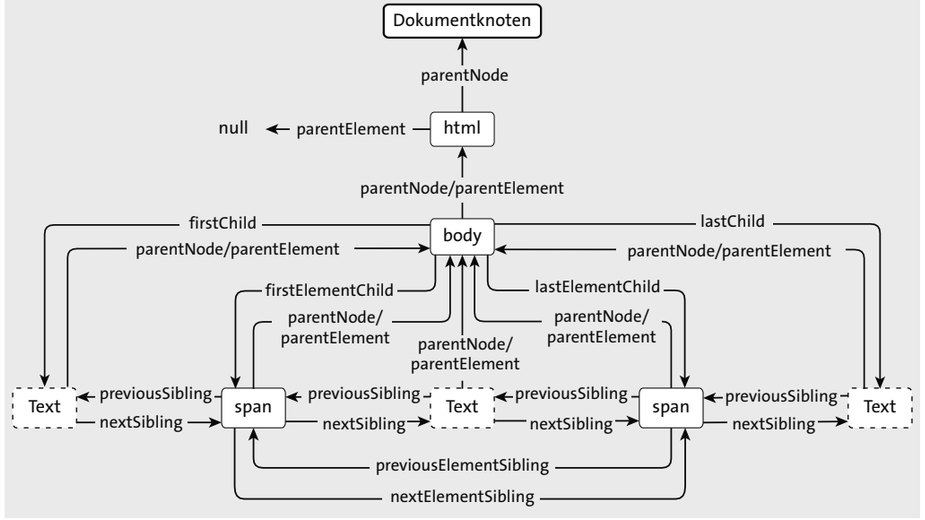

Abbildung 5.16 Übersicht über die verschiedenen Zugriffsformen

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, welche einen booleschen Wert zurückgibt. Ob ein

Element Kindelemente hat, können Sie über die Eigenschaft `childElementCount` bestimmen: Diese enthält die Anzahl an Kindelementen.

Listing 5.15 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).

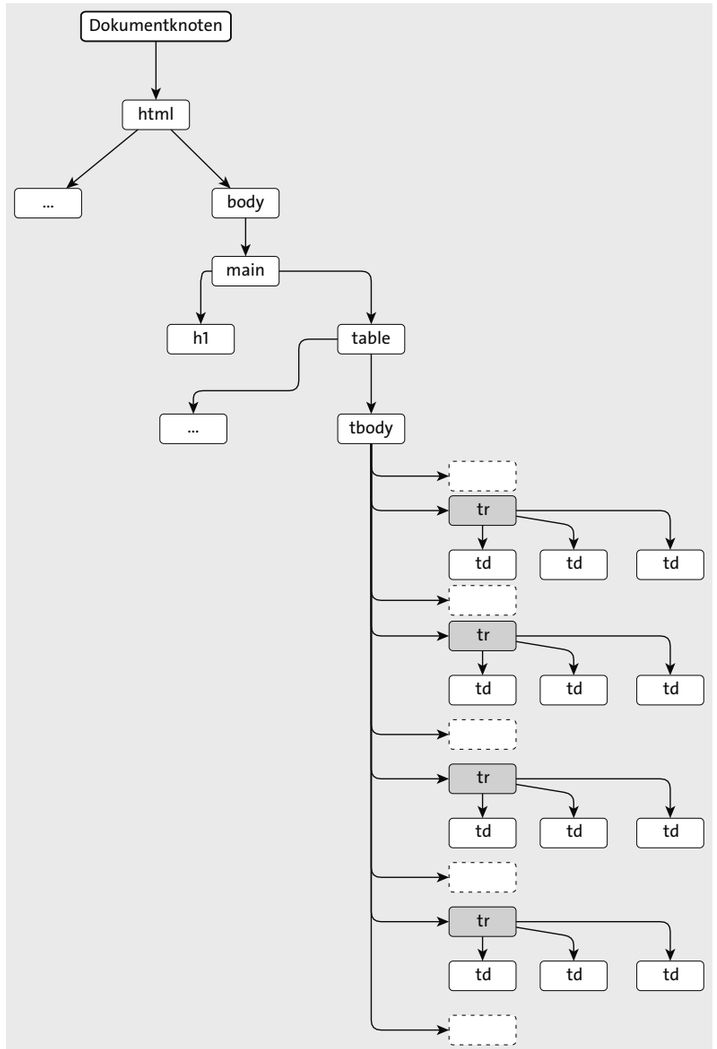

Abbildung 5.17 Selektion aller Kindelemente

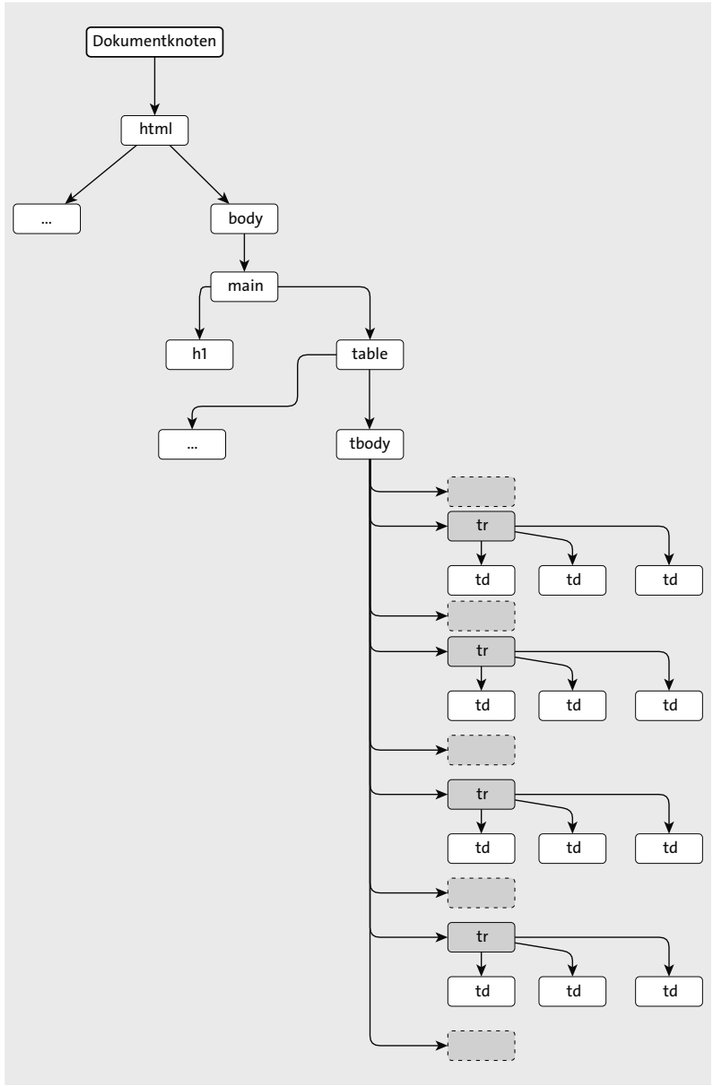

Abbildung 5.18 Selektion aller Kindknoten

Der Grund dafür ist, dass – obwohl zwischen und 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.

```
let 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 5.15 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 5.16 zeigt einige Beispiele dazu, Abbildung 5.19 das Ergebnis der Selektion des ersten und letzten Kindelements, Abbildung 5.20 das Ergebnis der Selektion des ersten und letzten Kindknotens.

```
let 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 5.16 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, da standen Webentwicklern nur Eigenschaften zur Verfügung, die alle Arten von Knoten zurückgeben (beispielsweise `firstChild` und `lastChild`), und man anhand des Knotentyps selbst die Elementknoten herausfiltern musste. Dies ist zum Glück nicht mehr so.

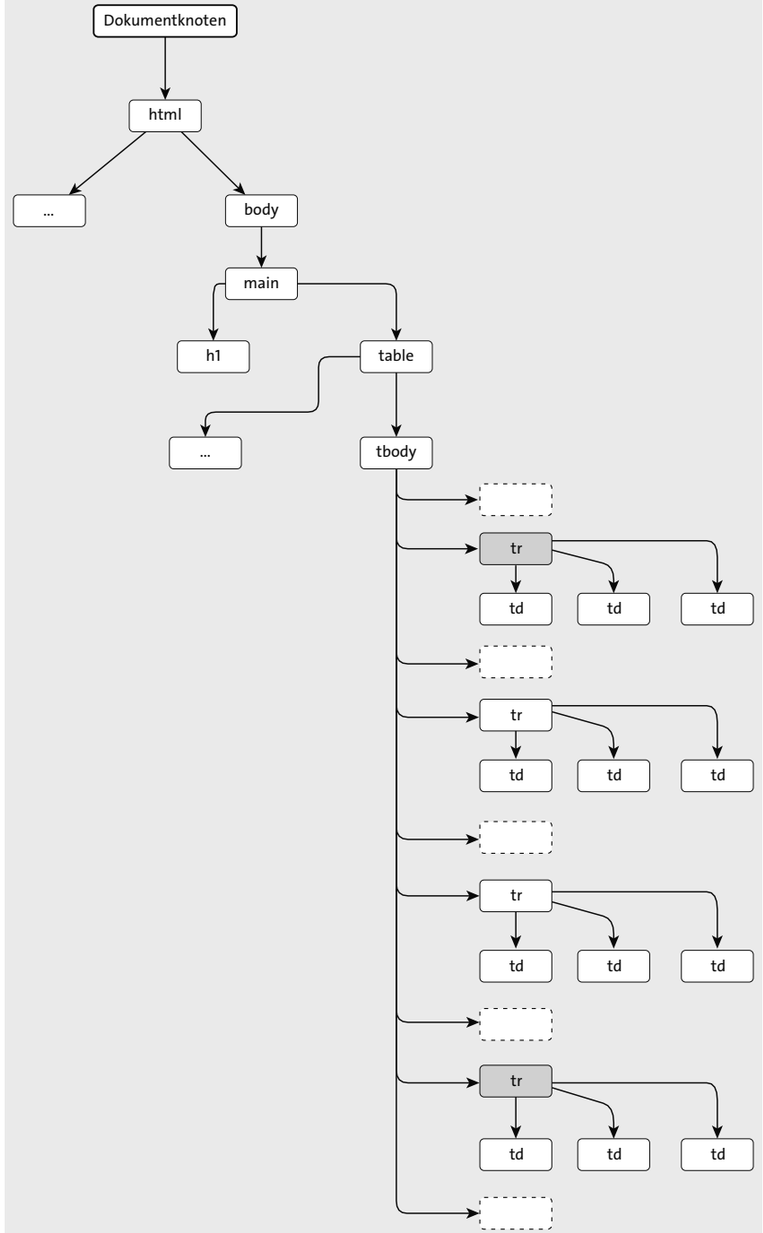

Abbildung 5.19 Selektion des ersten und des letzten Kindelements

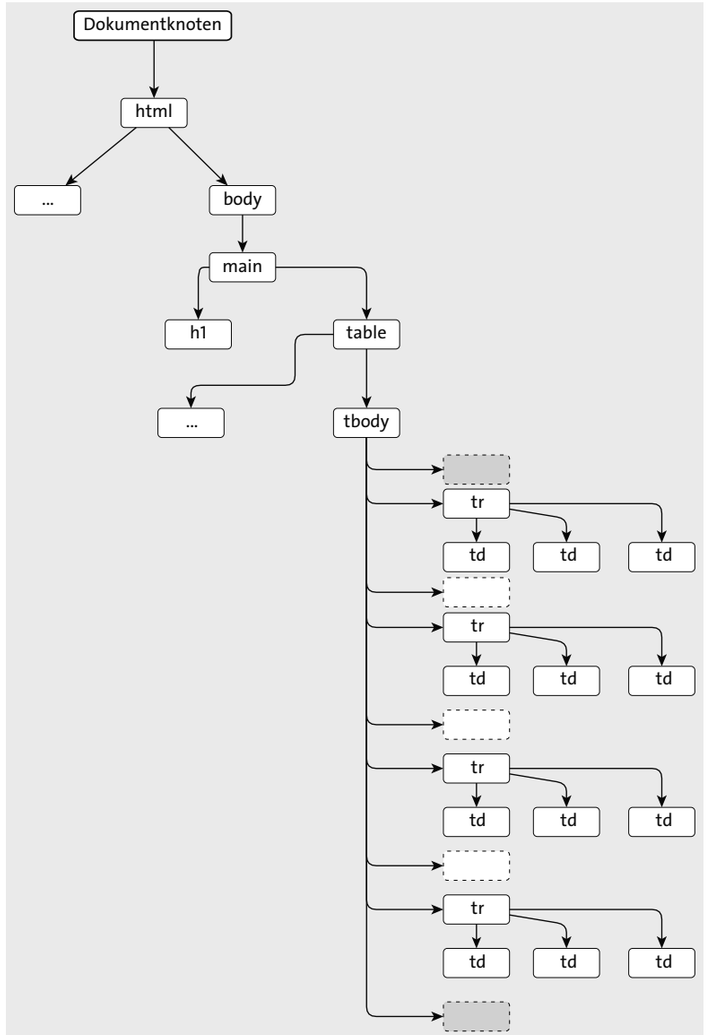

Abbildung 5.20 Selektion des ersten und des letzten Kindknotens

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 vorigen Geschwisterknoten.
- Die Eigenschaft `nextSibling` enthält den nachfolgenden Geschwisterknoten.
- Die Eigenschaft `previousElementSibling` enthält das vorige Geschwisterelement.
- Die Eigenschaft `nextElementSibling` enthält das nachfolgende Geschwisterelement.

Listing 5.17 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).

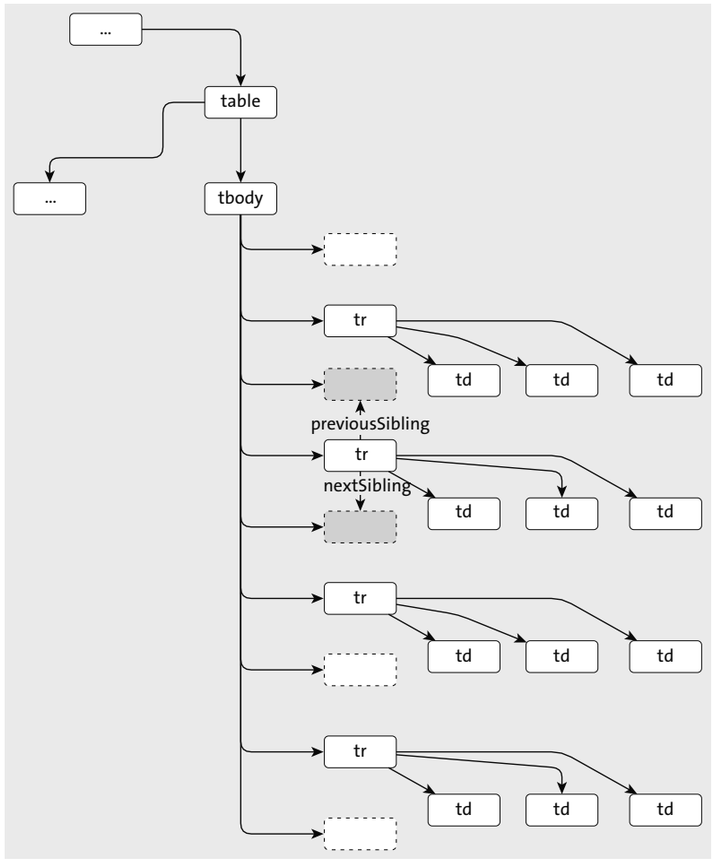

Abbildung 5.21 Selektion des vorigen und nachfolgenden Geschwisterknotens

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

```
let 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 5.17 Zugriff auf spezielle Geschwisterknoten und Geschwisterelemente

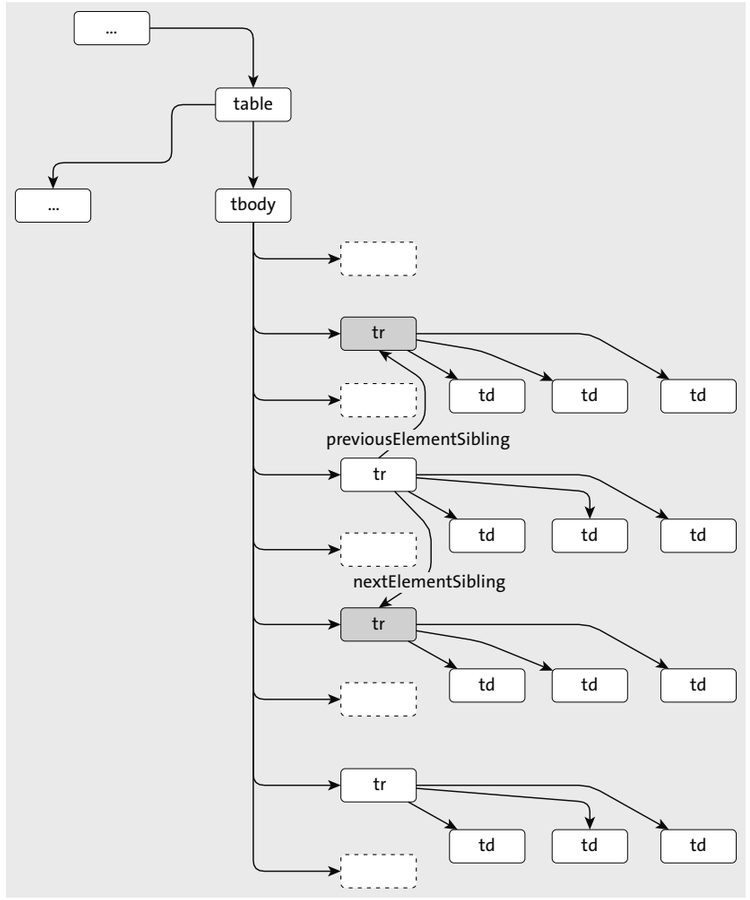

Abbildung 5.22 Selektion des vorigen und nachfolgenden Geschwisterelements

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 (nur `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 5.18, der geschachtelte Listen enthält. Im JavaScript-Code in Listing 5.19 wird die Methode `getElementsByTagName()` mit 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 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 5.18 Beispiel HTML-Seite

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

```
let subListListItems = subList.getElementsByTagName('li');
console.log(subListListItems.length); // Ausgabe: 4
```

Listing 5.19 Selektion von Elementen ausgehend von einem Elternelement

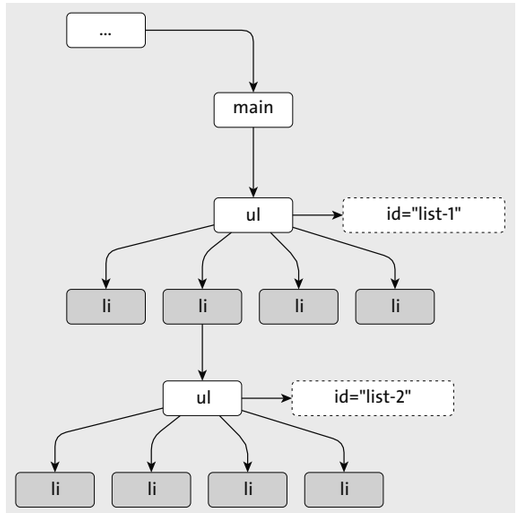

Abbildung 5.23 Aufruf der Methode »getElementsByTagName()« auf dem Dokumentknoten

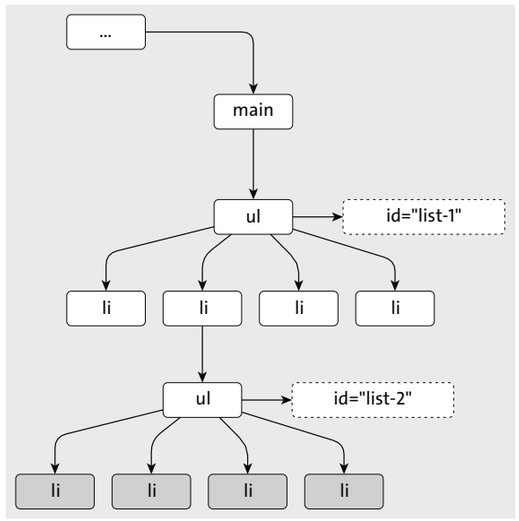

Abbildung 5.24 Aufruf der Methode »getElementsByTagName()« auf dem -Element mit ID »list-2«

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 oder Elemente hinzufügen oder entfernen. Folgende Tabelle zeigt einen Überblick über den wichtigsten Teil der entsprechenden Eigenschaften und Methoden, die dafür zur Verfügung stehen und die wir in den folgenden Abschnitten im Detail besprechen werden.

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«

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

Eigenschaft/Methode	Beschreibung	Abschnitt
`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 (Forts.)

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

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 im Wert, den man zurückerhält, nicht enthalten sind. Die folgenden beiden Listings machen dies deutlich: In Listing 5.20 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 5.20 HTML mit geschachtelten Elementen

Greifen Sie jetzt wie in Listing 5.21 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>`.

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

Listing 5.21 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 5.22 zu sehen. Hier wird dem Listenelement von eben ein neuer Text zugewiesen.

```
let element = document.querySelector('#news li:nth-child(1)');
element.textContent = 'Platten-News: Neues Album von Tool immer ¿
  noch nicht erschienen.';
```

Listing 5.22 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).

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

Listing 5.23 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 HTML in ein Element einfügen, können Sie die Eigenschaft `innerHTML` verwenden. Wir 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 5.24 zeigt dazu ein Beispiel: Hier wird der gleiche HTML-Baustein wie schon in Listing 5.23 hinzugefügt, diesmal allerdings auch als HTML interpretiert (siehe Abbildung 5.26).

```
let element = document.querySelector('#news li:nth-child(1)');
element.innerHTML = '<strong>Platten-News:</strong> Neues Album von <em>Tool¿
  </em> immer noch nicht erschienen.';
```

Listing 5.24 Bei 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 5.25).

```
let 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 5.25 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 5.26 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).

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

Listing 5.26 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, u. a. die Methoden `createElement()` für das Erstellen von Elementknoten (siehe Abschnitt 5.4.1) und `createAttribute()` für das Erstellen von Attributknoten (siehe dazu den Abschnitt 5.5.3).

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 Falle 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`, wo 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 ein anderes Element/einen anderen Knoten hinzufügen, sprich als voriges 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) zur Verfügung.

Ein etwas komplexeres – dafür aber praxisrelevantes – Beispiel zeigt Listing 5.27. 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 5.28) 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 sind durch Kommentare im Listing gekennzeichnet und anhand Abbildung 5.27 nachzuvollziehen.


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

```
    // (9)
    let 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 5.27 Erzeugen einer Tabelle auf Basis der Kontaktliste

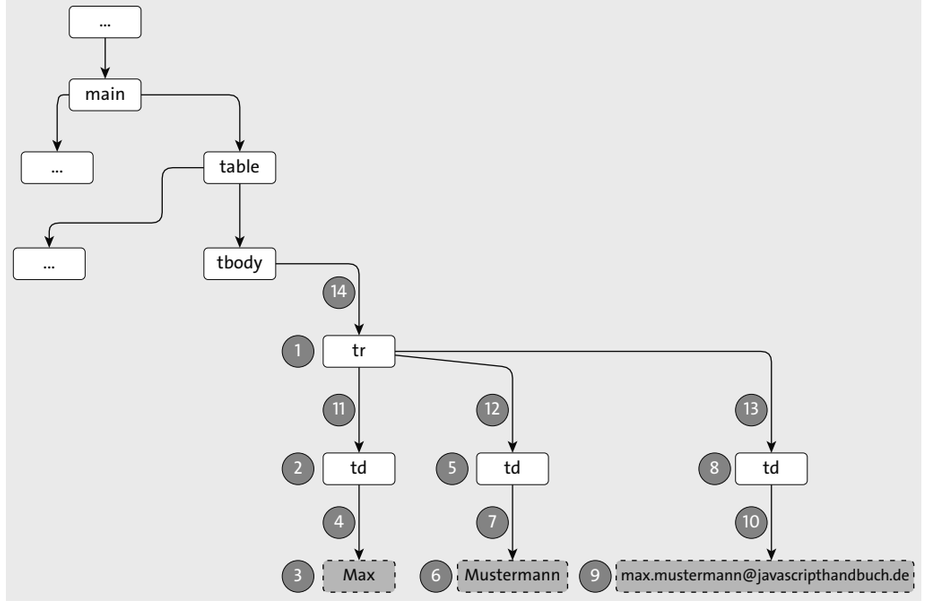

Abbildung 5.27 Reihenfolge der Schritte

```
<!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 5.28 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 5.29 sehen Sie (auf Basis der Listings aus vorigem 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() {
  let 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 5.29 Beispiel für die Verwendung der Methode »removeChild()«

5.4.3 Die verschiedenen Typen von HTML-Elementen

Jedes HTML-Element wird innerhalb eines DOM-Baumes 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` usw. 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, sind in der Tabelle der Vollständigkeit halber noch in kursiver Schrift 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`*

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

HTML-Element	Typ
<frameset>	*HTMLFrameSetElement*
<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

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

HTML-Element	Typ
<object>	HTMLObjectElement
	HTMLOListElement
<optgroup>	HTMLOptGroupElement
<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

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

HTML-Element	Typ
<td>	HTMLTableCellElement
<textarea>	HTMLTextAreaElement
<tfoot>	HTMLTableSectionElement
<th>	HTMLTableHeaderCellElement
<thead>	HTMLTableSectionElement
<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 5.30 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: u. a. 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`, welche die verschiedenen Tabellenkörper, sprich `<tbody>`-Elemente, einer Tabelle enthält, die Eigenschaft `rows`, welche die Tabellenzeilen (inklusive derer im Kopfbereich) enthält, sowie die Eigenschaft `tFoot`, welche den Fußbereich, sprich das `<tfoot>`-Element, enthält.

```
let 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 5.30 Jedes HTML-Element wird durch einen eigenen Objekttyp repräsentiert, wie hier beispielsweise Tabellen durch den Typ »HTMLTableElement«.

Neben individuellen Eigenschaften haben die verschiedenen Objekttypen je nachdem auch verschiedene Methoden: Beispielsweise hat der Typ `HTMLTableElement`, wie in Listing 5.31 zu sehen, u. a. 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, welcher wiederum u. a. 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. Deutlich übersichtlicher als in Listing 5.27, finden Sie nicht?

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

Listing 5.31 Die verschiedenen Objekttypen haben u. a. auch individuelle Methoden.

Eigenschaftsnamen vs. Elementnamen

Beachten Sie, dass die 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 erhält man den Wert des entsprechenden Attributs. Nehmen Sie als Ausgangspunkt den HTML-Code aus Listing 5.32: 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 5.32 Ein HTML-Link

Um auf diese Attribute zuzugreifen, verwenden Sie die Methode `getAttribute()`, wie in Listing 5.33 gezeigt.

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

Listing 5.33 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 5.34 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 5.34 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 5.35 und Listing 5.36 an. Ersteres zeigt einen HTML-Button mit verschiedenen Attributen, u. a. 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 5.35 Ein HTML-Button

```
let 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 5.36 Der Zugriff auf Event-Handler und das »style«-Attribut liefert je nach Zugriffsart unterschiedliche Rückgabewerte.

Der Grund, warum 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, warum der direkte Zugriff auf das `style`-Attribut keine Zeichenkette zurückgibt, ist der, 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 5.37 zeigt dazu ein Beispiel: Hier werden die Eigenschaften `class`, `href` und `target` für das zuvor selektierte Linkelement geändert.

```
let 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 5.37 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 5.38).

```
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 5.38 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 Falle 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. Dies müssen Sie manuell über die Methode `setAttributeNode()` am entsprechenden Element nachholen (siehe Listing 5.39).


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

Listing 5.39 Erstellen und Hinzufügen eines Attributknotens

5.5.4 Attribute entfernen

Über die Methode `removeAttribute()` können Sie Attribute wieder von einem Element entfernen. In Listing 5.40 werden auf diese Weise die beiden Attribute `class` und `href` aus dem Linkelement entfernt. Anschließend liefern die beiden Attribute den Wert `null`.

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

Listing 5.40 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 dies teils zu etwas umständlichem Code geführt, wenn man beispielsweise einem Element neue CSS-Klassen hinzufügen oder – schlimmer noch – beste-

hende CSS-Klassen entfernen wollte. Warum umständlich? Weil man in jedem Fall den Wert des Attributs parsen musste.

Diesem Umstand wurde mit der Version 4 der DOM API Rechnung getragen. Seitdem verfügen Elemente (in der DOM API, nicht in HTML) nämlich zusätzlich über die Eigenschaft `classList`, welche 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 5.41 zeigt zu diesen Methoden einige Beispiele.

```
let 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
let i = 5;
let condition = i > 0;
element.classList.toggle('active', condition);     // Klasse umschalten
console.log(element.classList);                    // ["link", "active"]
```

Listing 5.41 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 von dem 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 Elementnamen, 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 Geschwisterelemente/Geschwisterknoten selektiert werden.
- Über die Eigenschaft `textContent` kann auf den Textinhalt eines Knotens zugegriffen bzw. der Textinhalt gesetzt werden, über die Eigenschaft `innerHTML` dagegen auf den HTML-Inhalt eines Elements.
- 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 kurz 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 u. a.:

- **Zugriff auf native Features**: Im Gegensatz zu Webanwendungen, die gar nicht bzw. nur sehr eingeschränkt (beispielsweise über entsprechende Web-APIs oder Browser-Plugins) auf native Features und Hardware-Ressourcen 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.

- **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.
- **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.

Natürlich gibt es andersherum auch eine Menge Vorteile von Webanwendungen 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://electron.atom.io/*) ansetzen: Sie kombinieren moderne Webtechnologien mit der Möglichkeit, diese für die Erstellung von Desktopanwendungen zu verwenden. Beide Frameworks verwenden dazu einen ähnlichen Ansatz, 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, welches 2011 von Intel entwickelt wurde. Die Idee von NW.js ist es, 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 ermöglicht NW.js es zum einen, innerhalb eines entsprechenden Anwendungsfensters Anwendungen darzustellen, die in HTML, CSS

und JavaScript implementiert wurden (durch WebKit), zum anderen, mit dem zugrunde liegenden Betriebssystem zu interagieren, sprich native Funktionalitäten zu nutzen (durch Node.js).

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

19.1.1 Installation und Beispielanwendung

Die Installation von NW.js geschieht wie für Node.js-Anwendungen gewohnt über den Node.js Package Manager (NPM) mithilfe des Befehls `npm install -g nw`. Die globale Installation ist hierbei notwendig, um die zum Framework gehörenden Kommandozeilentools zu installieren.

Eine minimale NW.js-Anwendung besteht aus zwei Dateien: Über die Manifest-Datei *package.json* 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 sind 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, könnte daher wie folgt aussehen:

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

Listing 19.1 Exemplarischer Aufbau der Datei »package.json«

Der Inhalt der HTML-Datei könnte wie folgt aussehen:

```
<!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 Einstiegspunkt bildet eine HTML-Datei.

NW.js stellt seine API über ein globales Objekt `nw` zur Verfügung, über welches 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), innerhalb dessen dann über das Node.js-Modul os die Informationen zu der verwendeten Plattform ausgelesen werden (Node.js-API) und diese Informationen dann (wieder mithilfe der DOM API) in das Element mit der ID `platform` geschrieben werden.

```
'use strict'
const os = require('os');
document.addEventListener('DOMContentLoaded', () => {
  let platform = os.platform();
  document.getElementById('platform').textContent = platform;
});
```

Listing 19.3 Zugriff auf die Node.js-API

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

19.1.2 Packaging

Für das Packaging von NW.js-Anwendungen, also die »Paketierung« von Quelltext-Dateien zu einer ausführbaren Datei, steht u. a. das Modul nw-builder (*https://github.com/nwjs/nw-builder*) zur Verfügung. Das Modul kann über den Befehl `npm install -g nw-builder` installiert werden, anschließend kann über die Kommandozeile global der Befehl `nwbuild` verwendet werden. Über den Parameter `–platforms` bzw. `-p` ist es dabei möglich, eine kommaseparierte Liste von Betriebssystemen zu übergeben, für die die Anwendung erzeugt werden soll. Mögliche Werte hierbei sind `win32`, `win64`, `osx32`, `osx64`, `linux32` und `linux64`. Um also beispielsweise die Anwendung für Windows in 64 Bit und macOS in 64 Bit zu erstellen, reicht folgender Befehl:


```
nwbuild --platforms win64,osx64
```

Über weitere Parameter lassen sich u. a. die zu verwendende Version von NW.js (`--version` bzw. `-v`) und der Build-Ordner (`--buildDir` bzw. `-o`) angeben.

Alternativ zu der Verwendung auf Kommandozeile kann `nwbuild` auch programmatisch genutzt werden, beispielsweise um es in Build-Tools wie Gulp (*https://gulpjs.com/*) oder Grunt (*https://gruntjs.com/*) zu integrieren.

```
'use strict';
const NwBuilder = require('nw-builder');
cont nw = new NwBuilder({
    files: './path/to/nwfiles/**/**',
    platforms: ['win64','osx64'],
    version: '0.14.6'
});
nw.on('log',  console.log);
nw.build().then(() => {
  console.log('all done!');
}).catch(error => {
  console.error(error);
});
```

Listing 19.4 Programmatisches Packaging einer NW.js-Anwendung

Auf einen Blick

Inhalt

3 Sprachkern 89