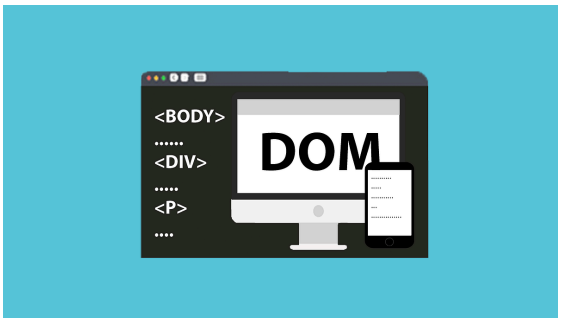
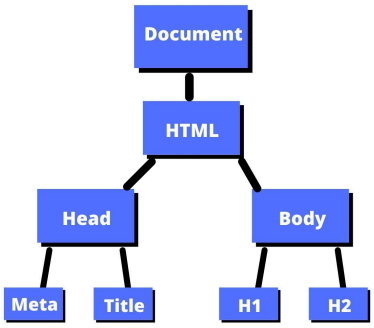
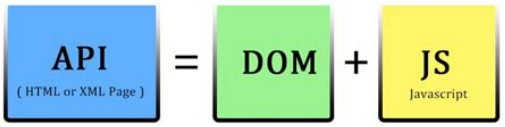
Bildungszentrum Uster
Höhere Berufsbildung Uster
HBU

04 HTML-DOM (Document Object Model) und Javascript







Dozent Bernhard Frei

edu.bernhard.frei@gmail.com

Document Object Model (DOM)

Was ist das Document Object Model (DOM)?

Das Document Object Model (dt. Dokumenten-Objekt-Modell), kurz DOM, ist eine standardisierte Programmierschnittstelle für die Strukturierung von HTML- und XML-Dokumenten. Entwickelt und veröffentlicht wurde sie vom World Wide Web Consortium (W3C), der 1994 von Web-Erfinder Tim Berners-Lee gegründeten Organisation für den Entwurf und das Etablieren von Standards für das World Wide Web.

Zweck des Document Object Models ist es, Programmierern den Zugriff auf die Komponenten eines Webprojekts und damit das Hinzufügen, Löschen oder Bearbeiten von Inhalten, Attributen und Styles so einfach wie möglich zu machen. DOM dient als plattformunabhängiges und sprachneutrales Bindeglied zwischen Skriptsprachen wie JavaScript und dem zugrundeliegenden Webdokument, indem es den Aufbau des Dokuments in einer Baumstruktur darstellt, in der jeder Knoten ein eigenständiges, ansteuerbares Objekt ist. Aufgrund dieser Strukturierungsform bezeichnet man die auf diese Art dargestellte Variante eines Webprojekts auch als DOM-Tree (dt. DOM-Baum).

Anders als es der Name vermuten lässt, handelt es sich bei DOM eigentlich nicht um ein Modell, sondern wie bereits erwähnt um eine Programmierschnittstelle. Allerdings kann ein Document Object Model im übertragenen Sinne als Modell für die Art und Weise des Zugriffs auf die als Objekt dargestellten Webdaten angesehen werden.

Die Einführung von JavaScript Mitte der 1990er-Jahre markierte einen Wendepunkt für das World Wide Web. Vor der Entstehung von JavaScript bestanden Websites hauptsächlich aus statischen HTML-Dokumenten, die lediglich serverseitig generiert und als solche ohne weitere interaktive Elemente dem Nutzer präsentiert wurden. Mit JavaScript wurde es erstmals möglich, clientseitig – also direkt im Browser des Nutzers – Inhalte dynamisch zu verändern, ohne dass eine neue Anfrage an den Server gestellt werden musste.

Browser-Hersteller wie Netscape und Microsoft reagierten auf diese Entwicklung durch die Implementierung von JavaScript-Interpretern, die es ermöglichten, JavaScript-Code zur Laufzeit auszuführen. Gleichzeitig entwickelten sie eigene Modelle und Ansätze für dynamisches HTML (DHTML), um das Verhalten von HTML-Dokumenten mithilfe von JavaScript flexibel zu gestalten. Diese frühen DHTML-Modelle beinhalteten die Möglichkeit, DOM-Knoten zu manipulieren, CSS-Stile dynamisch anzupassen und Ereignisse (Events) abzufangen und zu verarbeiten.

Da diese Implementierungen jedoch proprietär waren und sich in den verschiedenen Browsern voneinander unterschieden, standen Webentwickler vor der Herausforderung, ihre dynamischen Inhalte für jede Browser-Engine anzupassen und kompatibel zu gestalten. Dies führte zu erheblichem Mehraufwand und hinderte die Interoperabilität zwischen den Browsern.

Das World Wide Web Consortium (W3C) veröffentlichte daher 1998 die erste offizielle Spezifikation des Document Object Models (DOM). Das DOM definiert eine standardisierte Programmierschnittstelle, die den Zugriff auf und die Manipulation von HTML- und XML-Dokumenten ermöglicht. Es abstrahiert die Struktur eines Dokuments als hierarchischen Baum, in dem jedes Element, Attribut und jeder Textknoten als Objekt betrachtet wird, das über JavaScript manipulierbar ist. Diese Standardisierung legte den Grundstein für die konsistente Entwicklung von Webanwendungen, unabhängig vom verwendeten Browser.

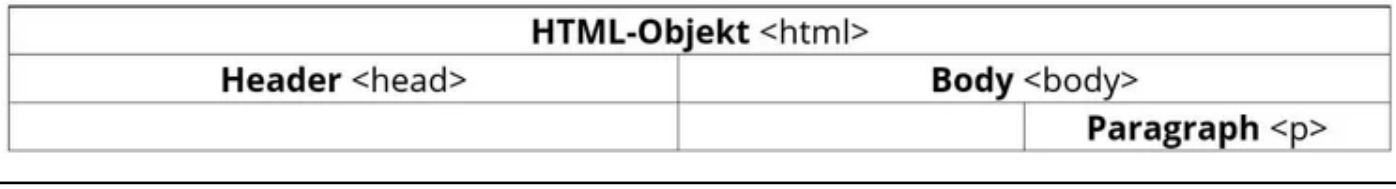
Warum ist der DOM so wichtig?

Das DOM stellt sicher, dass Webentwickler auf konsistente Weise mit der Struktur und dem Inhalt von HTML-Dokumenten arbeiten können. Es abstrahiert die Komplexität der zugrunde liegenden Darstellung des Dokuments und bietet eine universelle Methode zur Interaktion mit den verschiedenen Teilen eines Webdokuments. Ohne das DOM müssten Entwickler mit unterschiedlichen, proprietären Schnittstellen für jeden Browser umgehen. Mit der Standardisierung des DOM hat das W3C eine Grundlage geschaffen, auf der komplexe, dynamische und interaktive Webanwendungen effizient entwickelt werden können.

Durch die universelle Anwendung des DOM können Webentwickler plattformübergreifende und benutzerfreundliche Applikationen erstellen, die unabhängig vom Browser des Nutzers korrekt funktionieren.

Aufbau des DOM

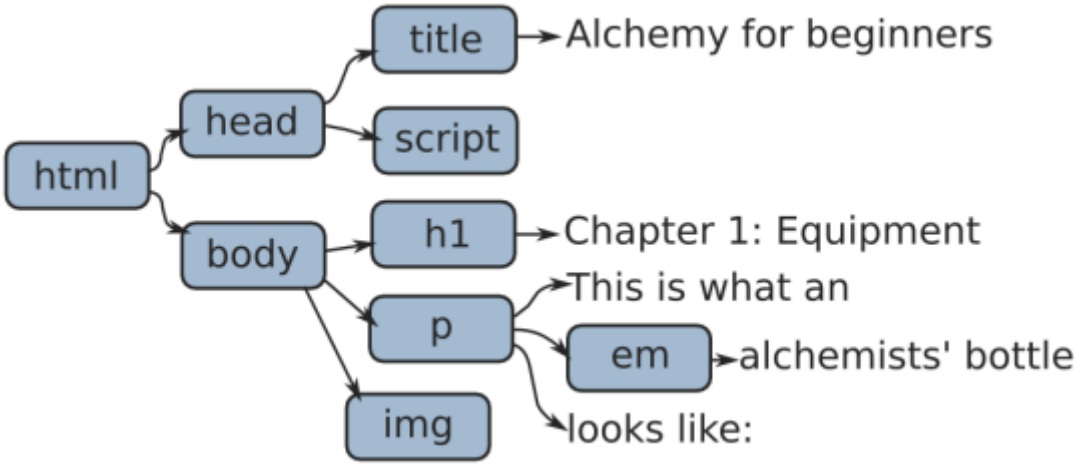
HTML-Markups definieren Beziehungen zwischen den verschiedenen enthaltenen Tags. So sind die per Tag gekennzeichneten Elemente eines Webdokuments beispielsweise abhängig von ihrer Rolle im Webprojekt über- oder untergeordnet. Zudem können manche Tags in anderen Tags enthalten sein. Um eben diese Hierarchien auch im Document Object Model adäquat wiederzugeben, greift die Schnittstelle auf die bereits erwähnte Baumstruktur zurück, die es möglich macht, die gerenderten Objekte entsprechend zu arrangieren. Wie genau ein DOM-Tree aufgebaut ist, hängt also immer von dem zugrundeliegenden HTML- bzw. XML-Dokument ab. Für Ersteres lässt sich aber folgende, projektübergreifende Grundhierarchie festhalten:



Die Objekte des minimalen Document Object Models spiegeln das HTML-Grundgerüst wider.

Wie im HTML-Grundgerüst steht das HTML-Objekt selbst in der Hierarchie an oberster Stelle. Untergeordnet sind ihm der Header (Kopfzeile) und der Body (Hauptteil) der Website. Bei Letzterem ist davon auszugehen, dass er mindestens einen Paragraph (Abschnitt mit Textinhalt) enthält.
Die einzelnen Abspaltungen im DOM-Tree werden als Nodes bzw. Knoten bezeichnet. Zusätzlich wird zwischen Elementknoten wie den oben zu sehenden HTML-, Body-, Header- oder Paragraph-Objekten, Attributknoten wie „align“ oder „size“ und Textknoten unterschieden.
Knoten für Elemente (die HTML-Tags) bestimmen die Struktur des Dokuments. Diese können untergeordnete Knoten haben. Ein Beispiel für einen solchen Knoten ist document.body. Einige dieser untergeordneten Knoten können Blattknoten sein, wie etwa Textstücke oder Kommentarknoten.

HTML-Dokumente haben also eine so genannte hierarchische Struktur. Jedes Element (oder Tag) außer dem obersten <html>Tag ist in einem anderen Element enthalten, seinem übergeordneten Element. Dieses Element kann wiederum untergeordnete Elemente enthalten. Sie können sich das als eine Art Stammbaum vorstellen:

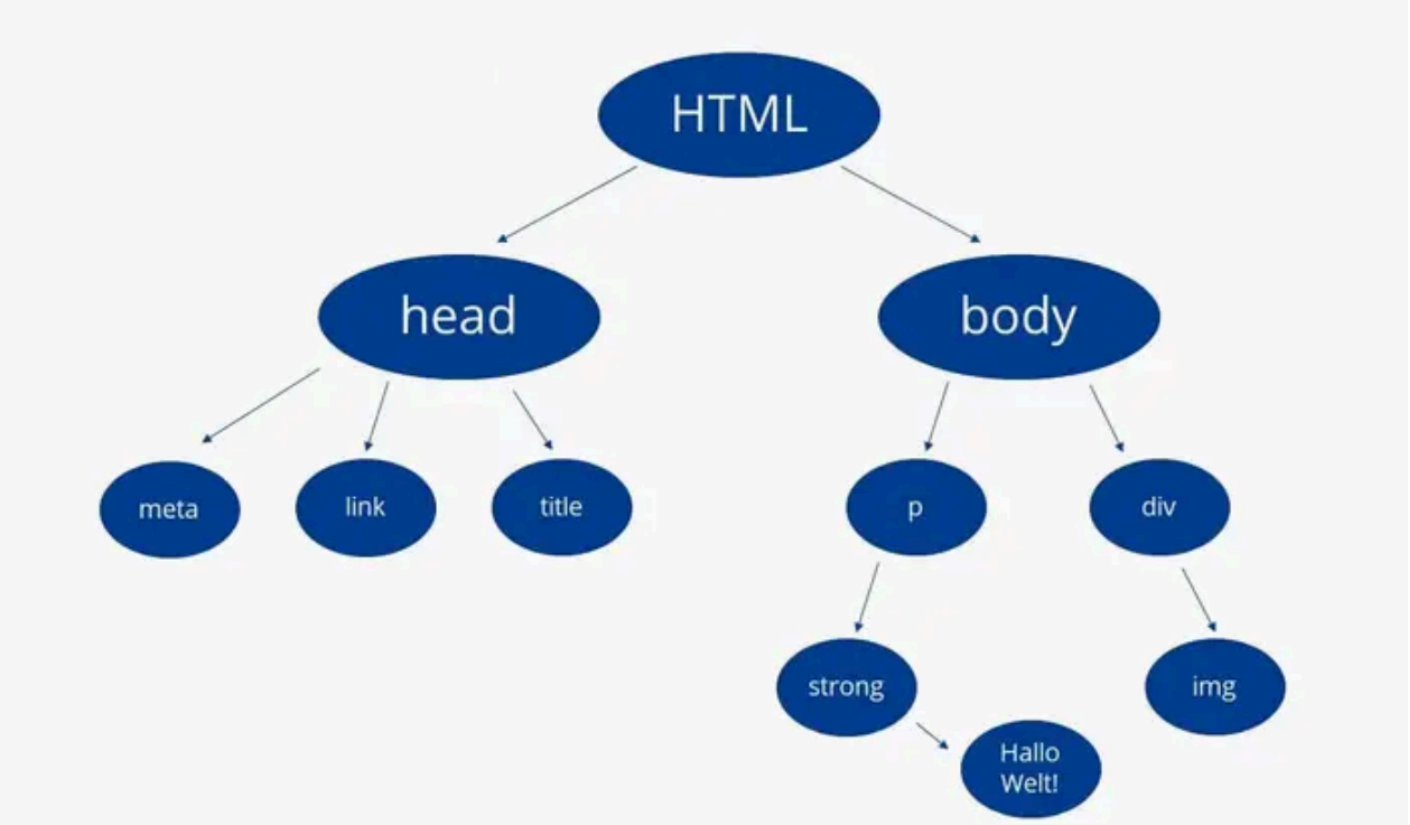


Das Dokumentobjektmodell basiert auf einer solchen Ansicht des Dokuments. Beachten Sie, dass der Baum zwei Arten von Elementen enthält: Knoten, die als blaue Kästchen dargestellt werden, und einfache Textstücke. Die Textstücke funktionieren, wie wir sehen werden, etwas anders als die anderen Elemente (zB haben sie nie untergeordnete Elemente).

Als weiteres Beispiel dient folgendes einfaches HTML-Dokument mit Header (inkl. Verlinkung auf CSS-Stylesheet) und Body sowie zwei Content-Elementen (Bild und Text):

```
<!DOCTYPE html>
<html>
    <head>
        <meta name="viewport" content="width=device-width,initial-scale=1">
        <link href="style.css" rel="stylesheet">
        <title>Title</title>
    </head>
    <body>
        <p><strong>Hallo Welt!</strong></p>
        <div><img src="bild.jpg"></div>
    </body>
</html>
```

Der aus diesem HTML-Code generierte DOM-Tree sieht folgendermaßen aus:



Das HTML-Element steht in der Hierarchie eines Webdokuments an erster Stelle.

Alternativ finden Sie manchmal auch folgende Präsentationsform für das hier verwendete Document-Object-Model-Beispiel:

```
DOCTYPE: HTML
HTML
----head
-  ---- meta
-  ---- link
-  ---- title
----body
-  ---- p
-    ---- strong
-      ---- TEXT: Hallo Welt!
-  ---- div
-    ---- img
```

Alternative Darstellung des DOM

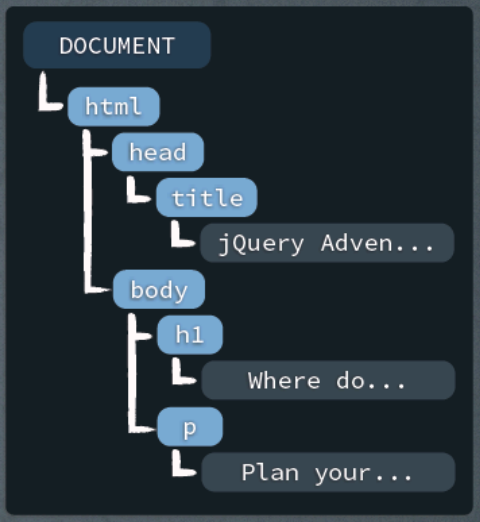
Hier ein weiteres Beispiel zum HTML und dem dazugehörigen DOM:

What does that DOM structure look like?

HTML document

```
<!DOCTYPE html>
<html>
<head>
  <title>jQuery Adventures</title>
</head>
<body>
  <h1>Where do you want to go?</h1>
  <p>Plan your next adventure.</p>
</body>
</html>
```

The DOM



Inside the DOM, HTML elements become “nodes” which have relationships with one another.

node types: element text

Wo und wann kommen DOM-Trees zum Einsatz?

Das Document Object Model wurde für den Einsatz im World Wide Web entwickelt und ist auch vor allem dort im Einsatz. Genauer gesagt sind es die jeweiligen Browser, mit denen Nutzer auf die Angebote des Webs zurückgreifen, die von der standardisierten Schnittstelle Gebrauch machen.

So nutzen gängige Webclients DOM bzw. auf DOM basierende Schnittstellen, um aufgerufene HTML- bzw. XML-Seiten zu rendern. Bei diesem Prozess werden die einzelnen Komponenten als Knoten zusammengefasst und in einem individuellen DOM-Tree organisiert. Parallel lädt der jeweilige Browser diese gerenderte Version des Webdokuments in den lokalen Speicher, um sie dort zu analysieren bzw. zu verarbeiten und abschließend die Seite in der vom Entwickler vorgesehenen Form präsentieren zu können. Für das Rendering setzen die Browser auf verschiedene Engines (Rendering-Software) wie Gecko (Firefox), WebKit (Safari) oder Blink (Chrome, Edge, Opera), die sich ebenfalls auf den DOM-Standard stützen.

Als objektorientierte Präsentation eines Webdokuments bleibt das Document Object Model außerdem auch nach der Ausgabe relevant – als Schnittstelle für sämtliche programmierten dynamischen Inhalte und somit für sämtliche Nutzerinteraktion, die das Aussehen der Seite während der Ausgabe verändern können.

Wenn der Browser das HTML einer Website lädt, werden diese Daten (die HTML - Elemente) in einer Baumhierarchie im Speicher abgelegt. Diese Daten nennt man das Document Object Model (DOM) und ist die Darstellung von HTML als Baum von Objekten, die die Struktur und den Inhalt eines Dokuments im Web bilden. Das HTML-DOM ist auch ein Programmierinterface für Webdokumente (im Normalfall wird Javascript dafür benutzt). Es stellt das Dokument als Baumstruktur dar, in der jedes Element als Knoten dargestellt wird. Mit dem DOM können wir HTML-Dokumente dynamisch ändern, also den Inhalt, die Struktur und das Styling einer Webseite nachträglich anpassen, ohne die Webseite neu laden zu müssen.

Der DOM als Standard

Das HTML DOM ist ein Objektmodell und eine Programmierschnittstelle für HTML. Es definiert:

- Die HTML-Elemente als Objekte
- Die Eigenschaften aller HTML-Elemente
- Die Methoden für den Zugriff auf alle HTML-Elemente
- Die Ereignisse für alle HTML-Elemente
- Die Programmierschnittstelle sind die Eigenschaften und Methoden jedes Objekts.
- Eine Eigenschaft ist ein Wert, den Sie abrufen oder festlegen können (wie das Ändern des Inhalts eines HTML-Elements).
- Eine Methode ist eine Aktion, die Sie ausführen können (z. B. das Hinzufügen oder Löschen eines HTML-Elements).

Das HTML DOM ist auch eine API (Programmierschnittstelle) für JavaScript :

- JavaScript kann HTML-Elemente hinzufügen/ändern/entfernen
- JavaScript kann HTML-Attribute hinzufügen/ändern/entfernen
- JavaScript kann CSS-Stile hinzufügen/ändern/entfernen
- JavaScript kann auf HTML-Ereignisse reagieren
- JavaScript kann HTML-Ereignisse hinzufügen/ändern/entfernen

Das DOM stellt sicher, dass Webentwickler auf konsistente Weise mit der Struktur und dem Inhalt von HTML-Dokumenten arbeiten können. Es abstrahiert die Komplexität der zugrunde liegenden Darstellung des Dokuments und bietet eine universelle Methode zur Interaktion mit den verschiedenen Teilen eines Webdokuments. Ohne das DOM müssten Entwickler mit unterschiedlichen, proprietären Schnittstellen für jeden Browser umgehen. Mit der Standardisierung des DOM hat das W3C eine Grundlage geschaffen, auf der komplexe, dynamische und interaktive Webanwendungen effizient entwickelt werden können.

Durch die universelle Anwendung des DOM können Webentwickler plattformübergreifende und benutzerfreundliche Applikationen erstellen, die unabhängig vom Browser des Nutzers korrekt funktionieren.

Javascript und der DOM

Das Document Object Model (DOM) ist eine API (Application Programming Interface), die speziell dafür entwickelt wurde, um die Struktur und den Inhalt von HTML- und XML-Dokumenten zu modellieren. Das DOM ist eine standardisierte Schnittstelle, die es Programmierern ermöglicht, die verschiedenen Komponenten eines Dokuments — wie Elemente, Attribute und Textknoten — zu identifizieren und direkt zu manipulieren. Diese API wurde vom World Wide Web Consortium (W3C) entwickelt, um eine einheitliche Grundlage für die Interaktion mit Webdokumenten zu schaffen, unabhängig von der verwendeten Programmiersprache oder Plattform.

Javascript direkt im HTML

Der wichtigste HTML-Tag `<script>` ermöglicht uns, ein JavaScript-Programm direkt in ein Dokument einzufügen.

```
<!DOCTYPE html>
<html lang="de">
<head>
    <title>Testseite</title>
</head>
<body>
    <h1>Teste alert</h1>
    <script>
        alert("Hallo!");
    </script>
</body>
</html>
```

Ein solches Script wird ausgeführt, sobald der Browser beim Lesen des HTML-Dokuments auf den `<script>`-Tag stößt. In diesem Fall wird beim Öffnen der Seite ein Dialogfeld angezeigt – die `alert`-Funktion - zeigt nur eine Nachricht an “Hallo”.

Externe JavaScript-Dateien einbinden

Große Programme direkt in HTML-Dokumente einzufügen, ist oft unpraktisch. Der `<script>`-Tag kann ein `src`-Attribut enthalten, um eine Script-Datei (eine Textdatei mit einem JavaScript-Programm) von einer URL abzurufen.

```
<!DOCTYPE html>
<html lang="de">
<head>
    <title>Testseite</title>
</head>
<body>
    <h1>Teste alert</h1>
    <script src="hello.js"></script>
</body>
</html>
```

Die hier eingebundene Datei `hello.js` enthält dasselbe Programm – `alert("Hallo!");`. Wenn eine HTML-Seite andere URLs als Teil von sich selbst referenziert, wie z.B. Bilddateien oder ein Script, ruft der Webbrowser diese sofort ab und fügt sie in die Seite ein.

Ein `<script>`-Tag muss immer mit `</script>` geschlossen werden, selbst wenn es auf eine Script-Datei verweist und keinen eigenen Code enthält. Wenn du dies vergisst, wird der Rest der Seite als Teil des Scripts interpretiert.

JavaScript in HTML-Attributen

Einige HTML-Attribute können ebenfalls JavaScript enthalten. Zum Beispiel unterstützt der `<button>`-Tag (der als Schaltfläche angezeigt wird) ein `onclick`-Attribut. Der Wert dieses Attributs wird jedes Mal ausgeführt, wenn der Button geklickt wird.

```
<!DOCTYPE html>
<html lang="de">
<head>
    <title>Button Alert</title>
</head>
<body>
    <h1>Teste den Button</h1>
    <button onclick="alert('Boom!');">NICHT DRÜCKEN</button>
```

```
</body>
</html>
```

Beachte, dass hier einfache Anführungszeichen (') für den Text im `onclick`-Attribut verwenden werden, weil doppelte Anführungszeichen (") bereits verwendet werden, um das gesamte Attribut zu umschließen. Alternativ könnte man `&quot;` verwenden, um die inneren Anführungszeichen zu "entkommen" (damit sie nicht als Ende des Attributs verstanden werden).

Javascript Strukturen des DOM

Das DOM stellt die Struktur eines HTML- oder XML-Dokuments als Baum dar, in dem jeder Knoten (`Node`) ein Objekt repräsentiert. Diese Baumstruktur beginnt bei einem einzigen Wurzelknoten, dem `document`-Objekt, und verzweigt sich in verschiedene Zweige, die die verschiedenen Teile des Dokuments darstellen:

- **Dokument-Knoten** (`Document Node`): Das oberste Element in der Baumstruktur, das das gesamte Dokument repräsentiert.
- **Element-Knoten** (`Element Nodes`): Diese repräsentieren die HTML-Tags (z.B. `<div>`, `<p>`, `<a>`). Jeder Element-Knoten kann Kindknoten haben, die wiederum weitere Elemente oder Textknoten darstellen.
- **Attribut-Knoten** (`Attribute Nodes`): Diese repräsentieren die Attribute von HTML-Elementen (z.B. `id="meinElement"`, `class="beispiel"`). Attribute sind keine direkten Kinder des Element-Knotens, sondern können durch Methoden wie `getAttribute()` und `setAttribute()` manipuliert werden.
- **Text-Knoten** (`Text Nodes`): Diese enthalten den tatsächlichen Textinhalt eines Elements. Jeder Text innerhalb eines HTML-Elements wird als Text-Knoten dargestellt.

Ein einfaches HTML-Dokument, wie dieses:

```
<html>
  <body>
    <h1>Überschrift</h1>
    <p>Ein Absatz.</p>
  </body>
</html>
```

würde im DOM-Baum folgendermaßen strukturiert:

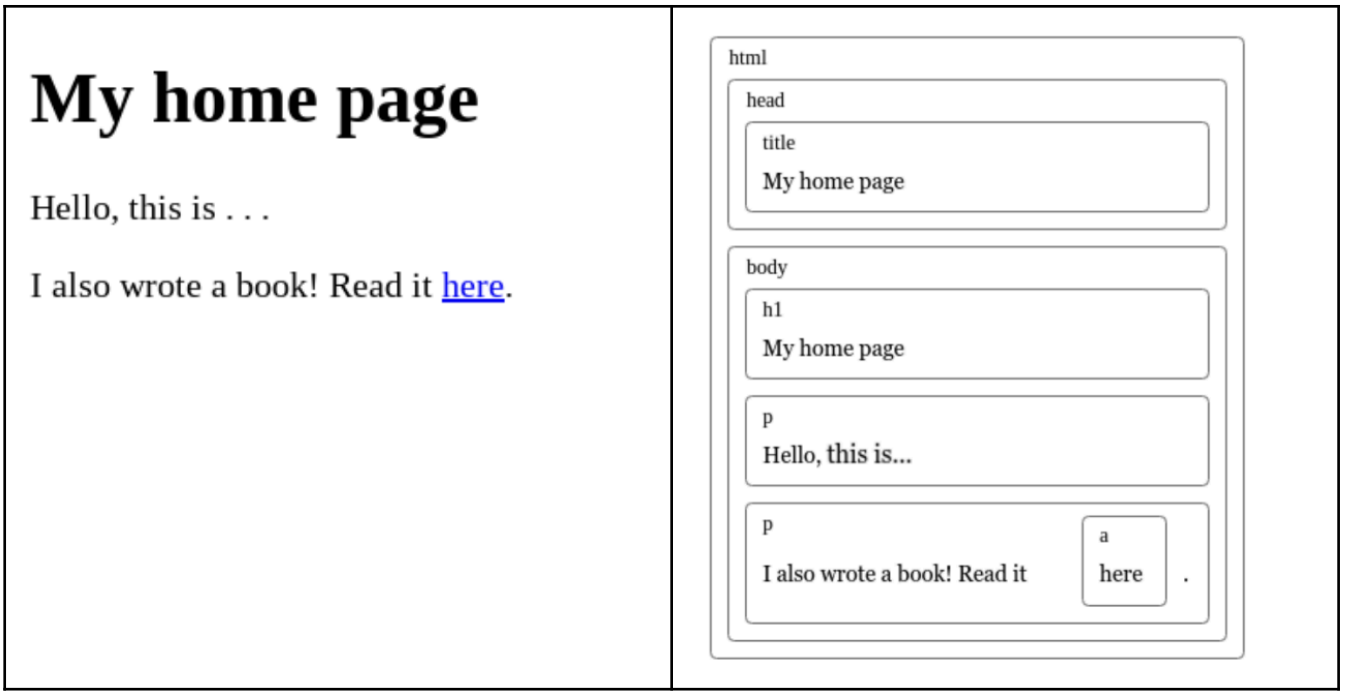
- `Document` (Document-Knoten)
  - `html` (Element-Knoten)
    - `body` (Element-Knoten)
      - `h1` (Element-Knoten)
        - Text-Knoten: "Überschrift"
      - `p` (Element-Knoten)
        - Text-Knoten: "Ein Absatz."

“document”: Der DOM im Javascript

Sie können sich ein HTML-Dokument als eine Reihe verschachtelter Boxen vorstellen. Tags wie <body> und </body>umschließen andere Tags, die wiederum andere Tags oder Text enthalten:

```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, this is ...</p>
    <p>I also wrote a book! Read it
      <a href="http://google.com">here</a>.</p>
  </body>
</html>
```

Diese Seite hat folgenden Aufbau:



Die Datenstruktur, die der Browser zur Darstellung des Dokuments verwendet, folgt dieser Struktur. Für jede Box gibt es ein Objekt, mit dem wir interagieren können, um herauszufinden, welches HTML-Tag es repräsentiert und welche weiteren Boxen und Texte es enthält. Diese Darstellung wird als Document Object Model oder kurz DOM bezeichnet.

Das globale `document`-Objekt bietet uns Zugriff auf diese untergeordneten Objekte. Seine `documentElement`-Eigenschaft verweist auf das Objekt, das das `<html>`-Tag repräsentiert, also das erste Element im HTML-Dokument. Da jedes HTML-Dokument immer einen `head` und einen `body` enthält, gibt es auch die `head`- und `body`-Eigenschaften, die auf diese Elemente zeigen.

Ein Baum ist also eine Datenstruktur und besteht aus Knoten. Jeder Knoten kann auf andere Knoten verweisen, sogenannte Kinder, die wiederum eigene Kinder haben können. Diese Form ist typisch für verschachtelte Strukturen, bei denen Elemente Unterelemente enthalten können.

Eine Datenstruktur wird als Baum bezeichnet, wenn sie eine verzweigte Struktur hat, keine Zyklen (ein Knoten darf sich nicht selbst enthalten, weder direkt noch indirekt) und eine einzige, klar definierte Wurzel besitzt. Im Fall des DOM dient `document.documentElement` als Wurzel.

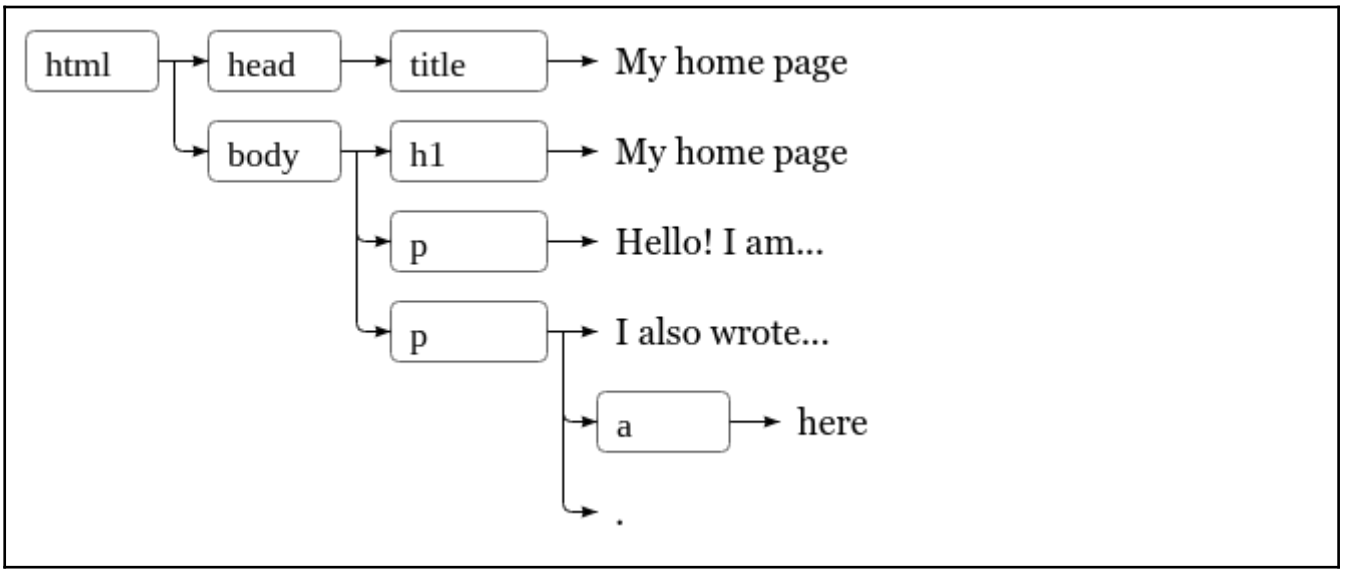
Bäume tauchen in der Informatik häufig auf. Neben der Darstellung von rekursiven Strukturen wie HTML-Dokumenten oder Programmen werden sie oft verwendet, um sortierte Datensätze zu verwalten, da Elemente in einem Baum in der Regel effizienter gefunden oder eingefügt werden können als in einem flachen Array.

Ein typischer Baum enthält verschiedene Arten von Knoten. Das Gleiche gilt für das DOM. Knoten für Elemente, die HTML-Tags repräsentieren, bestimmen die Struktur des Dokuments. Diese Knoten können wiederum Kindknoten haben. Ein Beispiel für einen solchen Knoten ist `document.body`. Einige dieser Kinder können Blattknoten sein, wie etwa Textstücke oder Kommentar-Knoten.

Jedes DOM-Knotenobjekt besitzt eine `nodeType`-Eigenschaft, die einen Code (eine Zahl) enthält, der den Knotentyp identifiziert. Elemente haben den Code 1, der auch als konstante Eigenschaft `Node.ELEMENT_NODE` definiert ist. Textknoten, die einen Abschnitt Text im Dokument repräsentieren, haben den Code 3 (`Node.TEXT_NODE`). Kommentare haben den Code 8 (`Node.COMMENT_NODE`).

Eine weitere Möglichkeit, unseren Dokumentbaum zu visualisieren, ist wie folgt:

Ein Diagramm zeigt das HTML-Dokument als Baum, mit Pfeilen von Elternknoten zu Kindknoten. Die Blätter sind Textknoten, und die Pfeile zeigen die Eltern-Kind-Beziehungen zwischen den Knoten an.


html
head
title
My home page
body
h1
My home page
p
Hello! I am...
p
I also wrote...
a
here
.

Die Programmier-Schnittstelle zum DOM

Die Verwendung von kryptischen numerischen Codes zur Darstellung von Knotentypen ist nicht ideal. Das liegt daran, dass die DOM-Schnittstelle nicht ausschließlich für JavaScript entwickelt wurde. Stattdessen ist sie als eine sprachneutrale Schnittstelle gedacht, die in verschiedenen Systemen verwendet werden kann – nicht nur für HTML, sondern zB auch für XML, ein generisches Datenformat mit einer HTML-ähnlichen Syntax.

Das ist leider ein Nachteil. Standards sind oft nützlich, aber in diesem Fall ist der Vorteil (Sprachenübergreifende Konsistenz) nicht besonders überzeugend. Eine Schnittstelle, die besser in die verwendete Sprache integriert ist, spart oft mehr Zeit, als eine, die in verschiedenen Sprachen einheitlich ist.

Ein Beispiel für diese schlechte Integration ist die `childNodes`-Eigenschaft, die Elementknoten im DOM haben. Diese Eigenschaft enthält ein array-ähnliches Objekt mit einer `length`-Eigenschaft und nummerierten Eigenschaften, um auf die Kindknoten zuzugreifen. Allerdings handelt es sich hierbei um eine Instanz des `NodeList`-Typs und nicht um ein echtes Array, weshalb Methoden wie `slice` und `map` nicht verfügbar sind.

Es gibt auch Probleme, die einfach durch schlechtes Design verursacht werden. Zum Beispiel gibt es keine Möglichkeit, einen neuen Knoten zu erstellen und ihm sofort Kinder oder Attribute hinzuzufügen. Stattdessen muss man ihn zuerst erstellen und dann die Kinder und Attribute einzeln hinzufügen. Code, der stark mit dem DOM interagiert, neigt dazu, lang, repetitiv und unübersichtlich zu werden.

Aber diese Mängel sind nicht fatal. Da JavaScript es uns erlaubt, eigene Abstraktionen zu schaffen, ist es möglich, verbesserte Methoden zu entwickeln, um die Operationen, die wir durchführen, auszudrücken. Viele Bibliotheken für die Programmierung im Browser bieten solche Werkzeuge an.

Zugriff und Manipulation mit JavaScript

JavaScript nutzt die DOM-API, um auf die Struktur und Inhalte eines HTML-Dokuments zuzugreifen und diese zu manipulieren. Einige der häufigsten DOM-Methoden und -Eigenschaften in JavaScript sind:

- **`document.getElementById('id')`**: Liefert das HTML-Element mit einer bestimmten `id`.
- **`document.querySelector('selector')`**: Liefert das erste Element, das einem CSS-Selector entspricht.
- **`element.innerHTML`**: Greift auf den HTML-Inhalt eines Elements zu oder setzt diesen.
- **`element.setAttribute('attribute', 'value')`**: Setzt ein Attribut eines Elements auf einen bestimmten Wert.
- **`element.appendChild(childNode)`**: Fügt ein neues Kind-Element zu einem bestehenden Element hinzu.
- **`element.removeChild(childNode)`**: Entfernt ein Kind-Element von einem bestehenden Element.

Diese Methoden ermöglichen es Entwicklern, gezielt auf Elemente des DOM zuzugreifen und sie zur Laufzeit zu verändern. Zum Beispiel könnte man mit dem folgenden JavaScript-Code eine bestehende Überschrift auf einer Webseite verändern:

```
<!DOCTYPE html>
<html lang="de">
<head>
    <title>Ändere Überschrift</title>
</head>
<body>
    <h1 id="ueberschrift">Originale Überschrift</h1>

    <script>
        // Zugriff auf das Element mit der ID "ueberschrift"
        const ueberschrift = document.getElementById('ueberschrift');

        // Änderung des Textinhalts der Überschrift
        ueberschrift.textContent = 'Neue Überschrift';
    </script>
</body>
</html>
```

Verwendung von Events im DOM

Eine der leistungsfähigsten Funktionen des DOM ist seine Fähigkeit, auf Benutzerereignisse (Events) zu reagieren, wie z.B. Mausklicks, Tastendrücke oder das Laden der Seite. Entwickler können Event-Listener (event listeners) hinzufügen, die auf bestimmte Ereignisse warten und darauf reagieren.
Beispiel für einen Event-Listener:

```
<!DOCTYPE html>
<html lang="de">
<head>
    <title>Button Event</title>
</head>
<body>
    <h1>Test Button Event</h1>
    <button id="meinButton">Klick mich!</button>

    <script>
        // Zugriff auf einen Button
        const button = document.getElementById('meinButton');

        // Event-Listener für Klick-Ereignis hinzufügen
        button.addEventListener('click', function() {
          alert('Button wurde geklickt!');
        });
    </script>
</body>
</html>
```

In diesem Fall wartet der Event-Listener auf ein `click`-Ereignis auf dem Button und führt dann eine Funktion aus, die eine Nachricht anzeigt.

Beispiel: Button drücken

Hier ist ein einfaches Beispiel mit HTML und JavaScript, das einen Button enthält. Wenn du auf den Button klickst, wird der Text „Button wurde geklickt“ in der Konsole ausgegeben.

Schritt 1: Erstelle die HTML + Javascript - Datei

1. Öffne deinen Text-Editor:
Verwende einen einfachen Text-Editor wie Notepad (Windows), TextEdit (Mac, im "Nur-Text"-Modus), oder einen Code-Editor wie Visual Studio Code oder Atom.

2. Erstelle eine neue Datei:
Erstelle eine neue Datei und speichere sie als `index.html` auf deinem Computer.

3. Schreibe den HTML + Javascript -Code:

```
<!DOCTYPE html>
 <html lang="de">
 <head>
     <meta charset="UTF-8">
     <meta name="viewport" content="width=device-width, initial-scale=1.0">
     <title>Button-Klick Beispiel</title>
 </head>
 <body>
     <!-- Ein Button auf der Webseite -->
     <button onclick="buttonClicked()">Klicke mich!</button>

     <!-- Verlinkung zur JavaScript-Funktion -->
     <script>
         // Diese Funktion wird ausgeführt, wenn der Button geklickt wird
```

```
        function buttonClicked() {
            // Ausgabe auf der Webseite selber als Overlay
            alert("Button wurde geklickt");

            // Ausgabe von "Button wurde geklickt" in der Konsole
            console.log("Button wurde geklickt");
        }
    </script>
</body>
</html>
```

4. Speichere die Datei:

Stelle sicher, dass du die Datei unter dem Namen `index.html` speicherst.

Schritt 2: Öffne die HTML-Datei im Browser

1. Öffne deinen Webbrowser:

Starte einen Webbrowser deiner Wahl (z. B. Chrome, Firefox, Edge).

2. Öffne die HTML-Datei im Browser:

- Gehe im Browser zu `Datei` > `Datei öffnen...` (in einigen Browsern heißt es auch `Open File...`).
- Wähle die `index.html`-Datei aus dem Ordner, in dem du sie gespeichert hast.

Alternativ kannst du auch den Ordner öffnen, in dem sich deine `index.html` befindet, und die Datei per Doppelklick direkt im Browser öffnen.

Schritt 3: Verwende die Entwicklerwerkzeuge (DevTools)

1. Öffne die Entwicklertools im Browser:

- Rechtsklicke auf die Seite und wähle "Untersuchen" (Chrome, Edge) oder "Element untersuchen" (Firefox).
- Alternativ kannst du die Entwicklertools auch über die Taste `F12` öffnen.

2. Gehe zum "Konsole"-Tab:

- In den Entwicklertools findest du oben verschiedene Tabs. Klicke auf "Konsole". Hier kannst du JavaScript-Befehle direkt ausführen und die Ausgaben sehen.

3. Teste die Webseite:
   - Klicke auf den Button "Klicke mich!" auf der Webseite.
   - Auf der Bildschirm erscheint “Button wurde geklickt”)
   - Schaue in die Konsole. Du solltest die Nachricht "Button wurde geklickt" sehen.

Erklärung des Codes Schritt für Schritt

1. HTML-Grundstruktur:
   - Die Datei beginnt mit dem `<!DOCTYPE html>`, das dem Browser mitteilt, dass dies ein HTML5-Dokument ist.
   - Der `<html>`-Tag umschließt das gesamte Dokument. Der `lang="de"`-Attribut gibt an, dass die Sprache des Dokuments Deutsch ist.

2. <head>`-Bereich:
   - Im `<head>`-Bereich befinden sich Meta-Informationen, wie die Zeichenkodierung (`UTF-8`) und die Ansichtseinstellungen (`viewport`). Diese helfen sicherzustellen, dass die Seite auf verschiedenen Geräten gut aussieht.
   - Der `<title>`-Tag definiert den Titel der Seite, der im Browser-Tab angezeigt wird.

3. <body>`-Bereich:
   - Hier befindet sich der eigentliche Inhalt der Seite. In diesem Fall gibt es einen Button (`<button>`).
   - Der Button hat ein `onclick`-Attribut. Dieses Attribut sagt dem Button, dass er die Funktion `buttonClicked()` aufrufen soll, wenn er angeklickt wird.

4. JavaScript innerhalb des `<script>`-Tags:
   - Das `<script>`-Tag enthält den JavaScript-Code, der die Funktion `buttonClicked()` definiert.
   - Funktion `buttonClicked()`:
     - Diese Funktion wird ausgeführt, wenn der Button angeklickt wird.
     - Es wird der Text “Button wurde geklickt” angezeigt
     - Die Funktion enthält eine `console.log`-Anweisung, die die Nachricht „Button wurde geklickt“ in der Konsole des Browsers anzeigt.

Mit diesem Beispiel hast du gelernt, wie man eine einfache Webseite mit einem Button erstellt und JavaScript verwendet, um eine Nachricht in der Konsole anzuzeigen, wenn der Button geklickt wird. Dies ist eine grundlegende Interaktion, die in vielen Webseiten verwendet wird

Die Javascript-Console

Die JavaScript-Konsole in den Developer Tools eines Browsers ist eine Umgebung zur direkten Ausführung von JavaScript-Code. Sie ermöglicht Entwicklern, mit der JavaScript-Engine des Browsers zu interagieren, indem sie in Echtzeit Skripte ausführen, Variablen inspizieren und Funktionen testen. Die Konsole dient außerdem der Ausgabe von Fehlern (Error Logging) und bietet Debugging-Funktionen, die beim Auffinden und Beheben von Problemen im Code nützlich sind. Sie erlaubt das Managen des Document Object Models (DOM), das Erstellen von Breakpoints, und das Auslesen von Netzwerkdaten, um den Zustand und die Interaktionen der Webanwendung zu überwachen und zu analysieren.

Debugging Funktionen

Die JavaScript-Konsole in den Developer Tools bietet eine Reihe von Debugging-Funktionen, die Entwicklern helfen, Fehler zu finden und ihren Code zu optimieren. Hier sind die wichtigsten Funktionen:

1. Konsolenausgaben (`console.log`, `console.error`, etc.)

- **`console.log()`**: Gibt beliebige Informationen in der Konsole aus, z. B. Variablen oder Nachrichten, um den Programmablauf zu verfolgen.
- **`console.error()`**: Zeigt Fehlermeldungen in der Konsole an.
- **`console.warn()`**: Gibt Warnungen aus, die weniger kritisch sind als Fehler, aber auf mögliche Probleme hinweisen.
- **`console.table()`**: Gibt Daten in tabellarischer Form aus, ideal für Arrays oder Objekte.

2. Breakpoints setzen

- Breakpoints stoppen die Ausführung des Codes an einer bestimmten Zeile. Entwickler können dann den aktuellen Zustand der Variablen untersuchen und Schritt für Schritt den weiteren Codeverlauf nachvollziehen.
- **Line Breakpoints**: Stoppt an einer bestimmten Codezeile.
- **Conditional Breakpoints**: Stoppt nur, wenn eine bestimmte Bedingung erfüllt ist.
- **XHR Breakpoints**: Stoppt, wenn eine bestimmte HTTP-Anfrage (XHR) ausgeführt wird.

3. Steuerung der Programmausführung

- **Weiter (Resume)**: Setzt die Ausführung des Skripts bis zum nächsten Breakpoint fort.
- **Schrittweise Ausführung (Step Over, Step Into, Step Out)**:
    - **Step Over**: Führt die nächste Zeile aus, ohne in Funktionsaufrufe hineinzuspringen.

- **Step Into**: Springt in den Funktionsaufruf hinein und verfolgt den Code innerhalb der Funktion.
- **Step Out**: Springt aus der aktuellen Funktion heraus und setzt das Skript im übergeordneten Kontext fort.

4. Watch Expressions

- Hiermit können Variablen oder komplexe Ausdrücke während der Codeausführung überwacht werden. Wenn sich der Wert einer Variablen ändert, kann der Entwickler dies in Echtzeit sehen.

5. Call Stack

- Der Call Stack zeigt die Reihenfolge, in der Funktionen aufgerufen wurden. Er ist besonders hilfreich, um nachzuvollziehen, wie der Programmfluss bei einem Fehler zu einem bestimmten Punkt gelangt ist.

6. Scope Inspection (Variablenüberwachung)

- Zeigt den aktuellen Gültigkeitsbereich (Scope) an, einschließlich globaler und lokaler Variablen. Dies ermöglicht es Entwicklern, den Zustand der Variablen während der Ausführung des Codes zu überprüfen.

7. Network Panel

- Zeigt alle Netzwerkaktivitäten (z. B. HTTP-Anfragen), die beim Laden der Seite auftreten. Es gibt Aufschluss über Ladezeiten, Datenmengen und Statuscodes.

8. Performance Panel

- Hilft, die Leistung der Anwendung zu analysieren, indem es CPU-Nutzung, Rendering und die Dauer von Skriptausführungen misst. Hier können Engpässe identifiziert werden, die zu langsamen Ladezeiten führen.

9. Source Maps

- Diese ermöglichen es Entwicklern, komprimierten oder minifizierten Code (wie in Produktionsumgebungen) zu debuggen, indem die ursprüngliche Codequelle angezeigt wird.

10. Error Stack Traces

- Zeigt detaillierte Informationen zu Fehlern an, einschließlich der Funktion, die den Fehler ausgelöst hat, und des Pfades, über den die Funktion aufgerufen wurde.

Diese Funktionen bieten eine umfassende Kontrolle über den Programmfluss und erlauben es Entwicklern,

effizient Fehler zu lokalisieren und zu beheben.

Weitere Funktionen

`console`-Objekt Funktionen

Das `console`-Objekt stellt viele Methoden bereit, die für das Debugging und die Entwicklung nützlich sind:

- **`console.log()`**: Gibt allgemeine Informationen in der Konsole aus. Diese Funktion wird oft verwendet, um Debug-Informationen oder den Status von Variablen anzuzeigen.
- **`console.error()`**: Gibt Fehlermeldungen in der Konsole aus, meist in roter Schrift. Dies wird verwendet, um echte Fehler im Code hervorzuheben.
- **`console.warn()`**: Zeigt Warnungen an, in gelber Schrift, um weniger kritische Probleme oder mögliche Gefahrenstellen im Code zu markieren.
- **`console.info()`**: Gibt Informationsmeldungen aus. Manche Browser zeigen diese mit einem speziellen Symbol an.
- **`console.assert()`**: Führt einen Ausdruck aus, der einen Test darstellt, und gibt eine Fehlermeldung aus, wenn der Test fehlschlägt. Beispiel:
    - ```
      console.assert(1 === 2, "1 ist nicht gleich 2");
      ```

      Hier wird nur dann eine Meldung ausgegeben, wenn der Ausdruck `false` ist.

- **`console.group()`** / **`console.groupEnd()`**: Diese Funktionen gruppieren mehrere Konsolenausgaben zusammen, um sie übersichtlicher zu machen. Alle Ausgaben zwischen `console.group()` und `console.groupEnd()` werden eingerückt dargestellt.
    - ```
      console.group("Testgruppe");
      console.log("Erste Ausgabe");
      console.log("Zweite Ausgabe");
      console.groupEnd();
      ```

- **`console.groupCollapsed()`**: Erzeugt eine zusammenklappbare Gruppe, die standardmäßig eingeklappt ist.
- **`console.table()`**: Zeigt Daten in einer tabellarischen Ansicht an. Diese Methode ist besonders

nützlich, um Arrays oder Objekte übersichtlich darzustellen.

    ○

```
const people = [{name: "Alice", age: 25}, {name: "Bob", age: 30}];
console.table(people);
```

- **`console.time()` / `console.timeEnd()`**: Dient zur Zeitmessung. Diese Methoden können verwendet werden, um die Zeit zu messen, die zwischen zwei Codeabschnitten verstreicht.

    ○

```
console.time("Laufzeit");
// Code, dessen Laufzeit gemessen werden soll
console.timeEnd("Laufzeit");
```

- **`console.count()`**: Zählt, wie oft ein bestimmter Ausdruck aufgerufen wird. Ideal zur Überprüfung, wie oft eine bestimmte Codezeile ausgeführt wird.

    ○

```
- console.count("Schleife durchlaufen");
```

- **`console.trace()`**: Gibt einen Stack-Trace aus, also die Liste der Funktionen, die aufgerufen wurden, bis zu dem Punkt, an dem die `console.trace()`-Anweisung steht. Nützlich, um zu sehen, von wo aus ein bestimmter Codepfad ausgelöst wurde.
- **`console.clear()`**: Löscht die gesamte Konsolenausgabe, sodass der Bildschirm leer ist und die nächste Ausgabe sauber beginnen kann.

Interaktiver Code-Ausführung

- **Direktes Ausführen von JavaScript**: Die Konsole ermöglicht es, JavaScript-Code direkt auszuführen. Dies ist ideal, um schnell Ideen zu testen oder Debugging-Code auszuführen, ohne die Seite neu zu laden.
- **Autocomplete und Intellisense**: Die Konsole unterstützt Autovervollständigung, was das Schreiben von Code erheblich beschleunigt. Sie zeigt auch Vorschläge für Objekte und Methoden an, sobald du anfängst zu tippen.
- **Historie der Befehle**: Du kannst mit den Pfeiltasten durch die Historie der Befehle navigieren, die du zuvor in der Konsole eingegeben hast.

DOM-Inspektion

- **DOM-Elemente auswählen und untersuchen**: Du kannst DOM-Elemente direkt in der Konsole inspizieren und manipulieren. Mit `$0` greifst du beispielsweise auf das zuletzt ausgewählte DOM-Element zu.
- **`document.querySelector()` und andere DOM-Methoden**: Über die Konsole kannst du DOM-Elemente direkt mit CSS-Selektoren auswählen und mit ihnen interagieren.
    -

```
let element = document.querySelector("#meineID");
console.log(element);
```

Live Editing

- **Ändern von HTML und CSS in Echtzeit**: In der Konsole kannst du nicht nur JavaScript, sondern auch HTML- und CSS-Eigenschaften in Echtzeit ändern, um sofort zu sehen, wie sich diese Änderungen auf die Seite auswirken.

Überwachung und Beobachtung (Monitoring)

- **`monitorEvents()`**: Du kannst Ereignisse auf einem Element überwachen, um zu sehen, wann sie ausgelöst werden. Zum Beispiel kannst du alle Klick-Ereignisse auf ein bestimmtes Element überwachen:
    -

```
monitorEvents(document.querySelector("#meineID"), "click");
```

- **`unmonitorEvents()`**: Um die Überwachung von Ereignissen zu stoppen.
- **Snippets**: In den Developer Tools von Chrome gibt es die Möglichkeit, wiederverwendbare Code-Snippets zu speichern und diese bei Bedarf in der Konsole auszuführen. Dies ist ideal, um sich wiederholende Aufgaben zu automatisieren.

Events Triggern und Analysieren

- **Event-Auslösung simulieren**: Über die Konsole kannst du manuell Ereignisse auslösen, um zu testen, wie deine Anwendung darauf reagiert.
    -

```
var event = new Event('click');
document.querySelector("#meineID").dispatchEvent(event);
```

Heap Snapshots und Speicherverwaltung

- **Speicherprofilierung**: In den Developer Tools gibt es auch Möglichkeiten, die Speichernutzung zu überwachen. Du kannst beispielsweise „Heap Snapshots“ nehmen, um zu sehen, wie der Speicherverbrauch zu verschiedenen Zeitpunkten aussieht.

Mobile Simulation

- **Responsive Design Mode**: Du kannst die Konsole auch verwenden, um deine Seite für mobile Geräte zu simulieren. Das schaltet die Ansicht in einen Modus, der verschiedene Bildschirmgrößen und Auflösungen nachahmt.

Beispiel: den DOM ändern

Der **DOM (Document Object Model)** ist eine Art „Datenstruktur“, die die gesamte Webseite als einen **Baum** darstellt. Stell dir diesen Baum so vor:

- Der Stamm des Baums ist das `<html>`-Element, das die Wurzel (der Anfang) der Webseite ist.
- Vom Stamm aus wachsen Äste, die zu **Knoten** führen. Jeder Knoten steht für ein Element der Webseite, wie z. B. ein Absatz (`<p>`), ein Bild (`<img>`) oder ein Bereich (`<div>`).

Diese Knoten haben eine **Eltern-Kind-Beziehung**:

- Ein **Eltern-Knoten** (Parent Node) ist ein Element, das andere Elemente enthält. Zum Beispiel ist `<body>` der Eltern-Knoten aller Inhalte, die auf der Seite angezeigt werden.
- Ein **Kind-Knoten** (Child Node) ist ein Element, das in einem anderen Element enthalten ist. Zum Beispiel sind alle `<p>`-Elemente im `<body>` Kind-Knoten des `<body>`-Elements.

Wie kann man die DOM-Struktur ändern?

Mit JavaScript kannst du die DOM-Struktur und damit die Struktur einer Webseite dynamisch (zur Laufzeit) ändern, indem du die **Eltern-Kind-Beziehungen** manipulierst. Du kannst zum Beispiel:

- **Elemente entfernen**: Bestehende Elemente löschen.
- **Elemente hinzufügen**: Neue Elemente erstellen und sie als Kinder zu anderen Elementen hinzufügen.
- **Elemente verschieben**: Elemente von einer Stelle im Baum zu einer anderen verschieben.
- **Elemente ersetzen**: Ein bestehendes Element durch ein neues Element ersetzen.

Ein Element entfernen

In diesem Beispiel entfernen wir ein <p>-Element von der Webseite.

Um ein Element zu entfernen, verwendest du die Methode **`remove`**.

Ausgabe vorher:

- Absatz 1: "Dieser Absatz wird entfernt."
- Absatz 2: "Dieser Absatz bleibt bestehen."

Ausgabe nachher:

- Nur Absatz 2: "Dieser Absatz bleibt bestehen."

```
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Element Entfernen</title>
</head>
<body>
    <p id="myParagraph">Dieser Absatz wird entfernt.</p>
    <p>Dieser Absatz bleibt bestehen.</p>

    <script>
        // Holt das <p>-Element mit der ID 'myParagraph'
        let paragraph = document.getElementById('myParagraph');
        // Entfernt das <p>-Element
        paragraph.remove();
    </script>
</body>
</html>
```

Ein Kind-Element hinzufügen

In diesem Beispiel fügen wir ein neues <p>-Element am Ende der Liste der Kinder hinzu.

Um ein neues Element hinzuzufügen, benutzt man die Methode **`appendChild`** oder **`insertBefore`**.

- **`appendChild`** fügt ein Element als letztes Kind eines Eltern-Elements hinzu.
- **`insertBefore`** fügt ein Element vor einem bestimmten anderen Kind eines Eltern-Elements hinzu.

Ausgabe vorher:

- Absatz 1: "Erster Absatz."
- Absatz 2: "Zweiter Absatz."

Ausgabe nachher:

- Absatz 1: "Erster Absatz."
- Absatz 2: "Zweiter Absatz."
- Absatz 3: "Dritter Absatz."

```
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Element Hinzufügen</title>
</head>
<body>
    <p>Erster Absatz.</p>
    <p>Zweiter Absatz.</p>

    <script>
        // Erzeugt ein neues <p>-Element
        let newParagraph = document.createElement("p");
        // Fügt Text zum neuen <p>-Element hinzu
        newParagraph.textContent = "Dritter Absatz.";

        // Fügt das neue <p>-Element am Ende des <body> hinzu
        document.body.appendChild(newParagraph);
    </script>
</body>
</html>
```

Ein Kind-Element verschieben

Hier verschieben wir das dritte <p>-Element (Drei) vor das erste <p>-Element (Eins).

Um ein Element zu verschieben, benutzt man die Methode **`insertBefore`**. Wenn du ein Element an eine neue Position setzt, wird es automatisch von seiner alten Position entfernt.

Ausgabe vorher:

- Absatz 1: "Eins"
- Absatz 2: "Zwei"
- Absatz 3: "Drei"

Ausgabe nachher:

- Absatz 1: "Drei"
- Absatz 2: "Eins"
- Absatz 3: "Zwei"

```
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Element Verschieben</title>
</head>
<body>
    <p>Eins</p>
    <p>Zwei</p>
    <p>Drei</p>

    <script>
        // Holt alle <p>-Elemente auf der Seite
        let paragraphs = document.body.getElementsByTagName("p");
        // Verschiebt das dritte <p> vor das erste <p>
        document.body.insertBefore(paragraphs[2], paragraphs[0]);
    </script>
</body>
</html>
```

Ein Kind-Element ersetzen

Hier ersetzen wir einen bestehenden Absatz durch einen neuen Absatz.

Um ein Element zu ersetzen, nutzt man die Methode **`replaceChild`**. Diese Methode ersetzt ein bestehendes Kind durch ein neues.

Ausgabe vorher:

- Absatz 1: "Alter Absatz."
- Absatz 2: "Bleibender Absatz."

Ausgabe nachher:

- Absatz 1: "Neuer Absatz."
- Absatz 2: "Bleibender Absatz."

```
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Element Ersetzen</title>
</head>
<body>
    <p id="alteId">Alter Absatz.</p>
    <p>Bleibender Absatz.</p>

    <script>
        // Erzeugt ein neues <p>-Element
        let newParagraph = document.createElement("p");
        // Fügt Text zum neuen <p>-Element hinzu
        newParagraph.textContent = "Neuer Absatz.";

        // Holt das alte <p>-Element, das ersetzt wird
        let oldParagraph = document.getElementById("alteId");
        // Ersetzt das alte <p>-Element durch das neue
        document.body.replaceChild(newParagraph, oldParagraph);
    </script>
</body>
</html>
```

Beispiel: ToDo-Liste

Die ToDo-Liste ist ein Klassiker im Programmieren, auch weil es so oft vorkommt. Wie funktioniert dieser Sourcecode?

- Ein Eingabefeld und ein Button werden bereitgestellt, um eine neue Aufgabe hinzuzufügen.
- Wenn der Benutzer auf "Aufgabe hinzufügen" klickt, wird die Funktion `addTask()` aufgerufen:
    - Sie prüft, ob das Eingabefeld leer ist und zeigt eine Warnung an, falls ja.
    - Sie erstellt ein neues Listenelement (`<li>`) und fügt den eingegebenen Text hinzu.
    - Eine "Löschen"-Schaltfläche wird erstellt und hinzugefügt, mit einer Funktion, die die Aufgabe entfernt, wenn darauf geklickt wird.
    - Die neue Aufgabe wird zur Aufgabenliste (`<ul>`) hinzugefügt.

```
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Einfache To-Do-Liste</title>
</head>
<body>
    <h1>Meine To-Do-Liste</h1>
    <input type="text" id="taskInput" placeholder="Neue Aufgabe hinzufügen">
    <button onclick="addTask()">Aufgabe hinzufügen</button>
    <ul id="taskList"></ul> <!-- Liste der Aufgaben -->

    <script>
        // Funktion zum Hinzufügen einer neuen Aufgabe
        function addTask() {
            const taskInput = document.getElementById('taskInput'); //
Eingabefeld für Aufgaben
            const taskText = taskInput.value; // Den eingegebenen Text holen

            if (taskText === '') {
                alert('Bitte eine Aufgabe eingeben!'); // Warnung, wenn das
Eingabefeld leer ist
                return;
            }
```

```
            // Neues Listen-Element (<li>) für die Aufgabe erstellen
            const newTask = document.createElement('li');
            newTask.textContent = taskText; // Den Text der Aufgabe hinzufügen

            // Schaltfläche "Löschen" erstellen
            const deleteButton = document.createElement('button');
            deleteButton.textContent = 'Löschen';
            deleteButton.onclick = function() { // Funktion zum Löschen der
Aufgabe
                newTask.remove();
            };

            // Die Schaltfläche zum Listen-Element hinzufügen
            newTask.appendChild(deleteButton);

            // Das neue Listen-Element zur To-Do-Liste hinzufügen
            document.getElementById('taskList').appendChild(newTask);

            taskInput.value = ''; // Eingabefeld leeren, nachdem die Aufgabe
hinzugefügt wurde
        }
    </script>
</body>
</html>
```

Beispiel: Taschenrechner

Dieser Code zeigt, wie man eine einfache Taschenrechner-Anwendung mit HTML und JavaScript erstellt. Er verwendet Funktionen, um Eingaben zu verarbeiten und Berechnungen durchzuführen, und demonstriert grundlegende Interaktivität auf Webseiten.

1. **HTML-Struktur:**
    - Wir haben ein Eingabefeld (`<input type="text">`), das als Display des Taschenrechners fungiert. Es ist "disabled", was bedeutet, dass der Benutzer nicht direkt hineinschreiben kann. Die Einträge werden nur über die Tasten gemacht.
    - Es gibt mehrere Tasten (`<input type="button">`), die jeweils eine Zahl oder einen mathematischen Operator darstellen (wie `+`, `-`, `*`, `/`).
    - Jede Taste hat ein `onclick`-Attribut, das eine bestimmte JavaScript-Funktion aufruft, wenn sie geklickt wird.
2. **JavaScript-Funktionen:**
    - **`addToDisplay(value)`**:
        - Fügt das angeklickte Zeichen (Zahl oder Operator) zum Text im Display hinzu.
        - Beispiel: Wenn der Benutzer die "1" drückt, wird "1" zum Text im Display hinzugefügt.
    - **`clearDisplay()`**:
        - Löscht das Display, wenn die "C"-Taste gedrückt wird.
        - Setzt den Text im Display auf einen leeren String (`''`).
    - **`calculate()`**:
        - Berechnet den mathematischen Ausdruck, der im Display steht, wenn die "="-Taste gedrückt wird.
        - Nutzt die JavaScript-Funktion `eval()`, um den Ausdruck zu berechnen. `eval()` interpretiert den Text im Display als JavaScript-Code und führt ihn aus.
        - Wenn der Benutzer etwas Ungültiges eingegeben hat, fängt die `try...catch`-Struktur

den Fehler ab und zeigt eine Warnung an. Danach wird das Display gelöscht.

```
<!DOCTYPE html>
<html lang="de">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Taschenrechner</title>
  <style>
      /* CSS-Stil für die Tasten des Taschenrechners */
      input[type="button"] {
          width: 50px; /* Breite jeder Taste */
          height: 50px; /* Höhe jeder Taste */
          font-size: 20px; /* Schriftgröße des Textes auf den Tasten */
      }
  </style>
</head>
<body>
  <h1>Taschenrechner</h1>
  <!-- Eingabefeld, das als Display des Taschenrechners dient. "disabled"
bedeutet, dass der Benutzer nicht direkt hineinschreiben kann. -->
  <input type="text" id="display" disabled>
  <br>
  <!-- Die Tasten des Taschenrechners: Jede Taste ruft beim Klicken eine
Funktion auf. -->
  <!-- Erste Reihe mit den Tasten "1", "2", "3" und "+" -->
  <input type="button" value="1" onclick="addToDisplay('1')">
  <input type="button" value="2" onclick="addToDisplay('2')">
  <input type="button" value="3" onclick="addToDisplay('3')">
  <input type="button" value="+" onclick="addToDisplay('+')">
  <br>
  <!-- Zweite Reihe mit den Tasten "4", "5", "6" und "-" -->
  <input type="button" value="4" onclick="addToDisplay('4')">
  <input type="button" value="5" onclick="addToDisplay('5')">
  <input type="button" value="6" onclick="addToDisplay('6')">
  <input type="button" value="-" onclick="addToDisplay('-')">
  <br>
  <!-- Dritte Reihe mit den Tasten "7", "8", "9" und "*" -->
  <input type="button" value="7" onclick="addToDisplay('7')">
  <input type="button" value="8" onclick="addToDisplay('8')">
  <input type="button" value="9" onclick="addToDisplay('9')">
  <input type="button" value="*" onclick="addToDisplay('*')">
  <br>
  <!-- Vierte Reihe mit der Taste "0", der Löschen-Taste "C", der
```

```
Berechnungstaste "=" und der Division "/" -->
  <input type="button" value="0" onclick="addToDisplay('0')">
  <input type="button" value="C" onclick="clearDisplay()"> <!-- Löscht den
aktuellen Inhalt des Displays -->
  <input type="button" value="=" onclick="calculate()"> <!-- Berechnet das
Ergebnis des Ausdrucks -->
  <input type="button" value="/" onclick="addToDisplay('/')"> <!-- Fügt das
Division-Zeichen zum Display hinzu -->

  <script>
      // Funktion, die aufgerufen wird, wenn eine Zahl oder ein Operator (wie
+, -, *, /) gedrückt wird.
      // Diese Funktion fügt das gedrückte Zeichen (z.B. '1', '2', '+') zum
aktuellen Text im Display hinzu.
      function addToDisplay(value) {
          const display = document.getElementById('display'); // Holt das
Display-Element anhand seiner ID
          display.value += value; // Fügt das gedrückte Zeichen zum Display
hinzu
      }

      // Funktion, die aufgerufen wird, wenn die "C"-Taste gedrückt wird, um
das Display zu löschen.
      function clearDisplay() {
          document.getElementById('display').value = ''; // Löscht den
aktuellen Text im Display
      }

      // Funktion, die aufgerufen wird, wenn die "="-Taste gedrückt wird, um
die Berechnung durchzuführen.
      function calculate() {
          const display = document.getElementById('display'); // Holt das
Display-Element anhand seiner ID
          try {
              // Berechnet den mathematischen Ausdruck im Display mit der
Funktion eval()
              // eval() führt den Text im Display als JavaScript-Code aus
              display.value = eval(display.value);
          } catch (e) {
              // Wenn ein Fehler auftritt (z.B. weil der Benutzer einen
ungültigen Ausdruck eingegeben hat), zeigt eine Warnung an
              alert('Ungültige Eingabe!');
              clearDisplay(); // Löscht das Display, wenn ein Fehler auftritt
          }
```

```
      }
  </script>
</body>
</html>
```

Beispiel: Slideshow

Der Code erstellt eine einfache Bild-Slideshow, die es dem Benutzer ermöglicht, durch verschiedene Bilder zu blättern, indem er auf die "Zurück"- und "Weiter"-Schaltflächen klickt.

1. **HTML-Struktur:**
    - `<h1>`: Überschrift der Seite, die "Bild-Slideshow" anzeigt.
    - `<div id="slideshow">`: Ein Bereich, der die Slideshow enthält. Darin gibt es:
        - `<img id="slide">`: Ein Bild, das in der Slideshow angezeigt wird. Die `src`-Eigenschaft enthält die URL des Bildes, das gerade angezeigt wird.
        - Zwei **Schaltflächen**: "Zurück" und "Weiter", um zwischen den Bildern zu wechseln.
2. **JavaScript-Funktionen:**
    - `const images`: Eine Liste (Array) mit den URLs der Katzenbilder, die in der Slideshow angezeigt werden.
    - `let currentIndex`: Speichert den aktuellen Index des angezeigten Bildes. Startet bei 0 (dem ersten Bild).
    - `prevImage()`: Funktion, die das vorherige Bild anzeigt. Wenn das erste Bild angezeigt wird und "Zurück" gedrückt wird, springt die Funktion zum letzten Bild.
    - `nextImage()`: Funktion, die das nächste Bild anzeigt. Wenn das letzte Bild angezeigt wird und "Weiter" gedrückt wird, springt die Funktion zum ersten Bild.
    - `updateImage()`: Funktion, die das Bild auf dem Bildschirm aktualisiert, indem es die `src`-Eigenschaft des Bildes ändert.

Was passiert beim Ausführen?

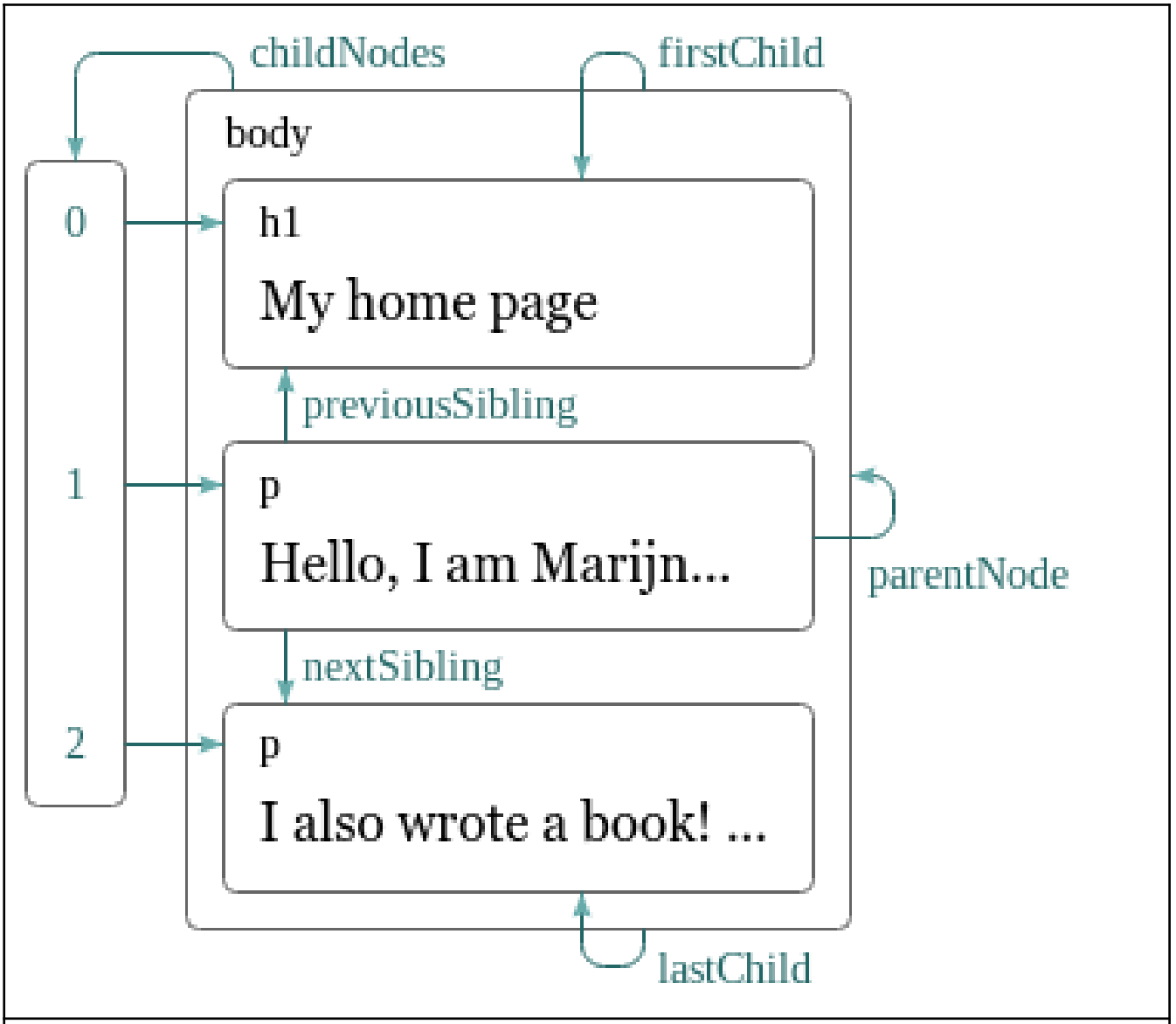
- Die Slideshow beginnt mit dem ersten Bild in der Liste `images`.
- Wenn du auf "Zurück" klickst, zeigt die Slideshow das vorherige Bild an.
- Wenn du auf "Weiter" klickst, zeigt die Slideshow das nächste Bild an.
- Die Slideshow wechselt automatisch zum ersten Bild, wenn das Ende der Liste erreicht ist, und umgekehrt.

```
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Bild-Slideshow</title>
    <style>
        /* Stil für die Slideshow */
        #slideshow {
            width: 300px; /* Breite des Bildbereichs */
            height: 200px; /* Höhe des Bildbereichs */
            margin: auto; /* Zentriert den Bereich in der Mitte */
            text-align: center; /* Zentriert den Text und die Bilder */
        }

        /* Stil für das Bild in der Slideshow */
        #slide {
            width: 300px; /* Passt die Breite des Bildes an den Bereich an */
            height: 200px; /* Passt die Höhe des Bildes an den Bereich an */
            object-fit: cover; /* Passt das Bild proportional in den Bereich
ein */
        }
    </style>
</head>
<body>
    <h1>Bild-Slideshow</h1>
    <div id="slideshow">
        <!-- Das Bild, das in der Slideshow angezeigt wird -->
        <img id="slide" src="cat1.jpg"alt="Slideshow">
        <br>
        <!-- Schaltflächen, um zwischen den Bildern zu wechseln -->
        <button onclick="prevImage()">Zurück</button>
        <button onclick="nextImage()">Weiter</button>
    </div>

    <script>
        // Liste der Bilder für die Slideshow (Bilder von Unsplash)
        const images = [
            "cat2.jpg", // Bild 2 von einer Katze
            "cat3.jpg", // Bild 3 von einer anderen Katze
            "cat4.jpg"  // Bild 4 von noch einer anderen Katze
```

```
        ];

        let currentIndex = 0; // Startet die Slideshow mit dem ersten Bild
(Index 0)

        // Funktion zum Anzeigen des vorherigen Bildes
        function prevImage() {
            currentIndex--; // Geht ein Bild zurück
            if (currentIndex < 0) currentIndex = images.length - 1; // Geht zum
letzten Bild, wenn wir am Anfang sind
            updateImage(); // Ruft die Funktion auf, um das Bild zu
aktualisieren
        }

        // Funktion zum Anzeigen des nächsten Bildes
        function nextImage() {
            currentIndex++; // Geht ein Bild weiter
            if (currentIndex >= images.length) currentIndex = 0; // Geht zum
ersten Bild, wenn wir am Ende sind
            updateImage(); // Ruft die Funktion auf, um das Bild zu
aktualisieren
        }

        // Funktion zum Aktualisieren des angezeigten Bildes
        function updateImage() {
            document.getElementById('slide').src = images[currentIndex]; //
Ändert das Bild, das angezeigt wird
        }
    </script>
</body>
</html>
```

Mit JS im DOM-Baum navigieren

DOM-Knoten enthalten eine Vielzahl von Verweisen auf andere, nahegelegene Knoten. Das folgende Diagramm verdeutlicht diese Verbindungen:



Diagramm, das die Verbindungen zwischen DOM-Knoten zeigt

Der `body`-Knoten wird als Box dargestellt, mit einem `firstChild`-Pfeil, der auf den `h1`-Knoten am Anfang zeigt, einem `lastChild`-Pfeil, der auf den letzten Paragraph-Knoten zeigt, und einem `childNodes`-Pfeil, der auf ein Array von Links zu all seinen Kindern zeigt. Der mittlere Paragraph hat einen `previousSibling`-Pfeil, der auf den Knoten vor ihm zeigt, einen `nextSibling`-Pfeil, der auf den Knoten nach ihm zeigt, und einen `parentNode`-Pfeil, der auf den `body`-Knoten verweist.

Obwohl das Diagramm nur einen Link von jedem Typ zeigt, besitzt jeder Knoten eine

`parentNode`-Eigenschaft, die auf den Knoten verweist, zu dem er gehört.

Ebenso hat jeder Elementknoten (Knotentyp 1) eine `childNodes`-Eigenschaft, die auf ein array-ähnliches Objekt verweist, das seine Kinder enthält: Jeder Elementknoten (wie ein `<div>`, `<p>`, oder `<ul>` in HTML) kann andere Knoten "in sich" haben, also Kinder. Die `childNodes`-Eigenschaft ist wie ein Behälter, in dem alle diese Kinder gesammelt werden. Dieser Behälter sieht aus und funktioniert so ähnlich wie eine Liste, in der die Kinder in der Reihenfolge stehen, in der sie im HTML-Dokument auftauchen.

Hier ein Beispiel:

```
<div>
  <p>Erstes Kind</p>
  <p>Zweites Kind</p>
</div>
```

Hier ist das `<div>` der "Elternknoten", und die beiden `<p>`-Tags sind seine "Kinder". Wenn du in JavaScript auf die `childNodes`-Eigenschaft des `<div>`-Knotens zugreifst, erhältst du eine Liste (ein sogenanntes "array-ähnliches Objekt") von den beiden `<p>`-Elementen.

Das bedeutet, dass `childNodes` dir alle Kinder zeigt, die zu einem bestimmten HTML-Element gehören.

Theoretisch könnte man sich nur mithilfe dieser `parentNode`- und `childNodes`-Verweise durch den gesamten Baum bewegen. JavaScript bietet jedoch auch eine Reihe zusätzlicher Verknüpfungen für bequemere Zugriffe:

- **`firstChild` und `lastChild`:** Diese Eigenschaften zeigen auf das erste bzw. letzte Kind eines Knotens oder haben den Wert `null`, wenn der Knoten keine Kinder hat.

- **`previousSibling` und `nextSibling`:** Diese Eigenschaften zeigen auf die benachbarten Knoten, d.h., die Knoten mit demselben Elternknoten, die direkt vor oder nach dem aktuellen Knoten stehen. Für das erste Kind ist `previousSibling` `null`, und für das letzte Kind ist `nextSibling` `null`.

Darüber hinaus gibt es die `children`-Eigenschaft, die ähnlich wie `childNodes` ist. Die `children`-Eigenschaft in einem DOM-Knoten funktioniert ähnlich wie die `childNodes`-Eigenschaft, aber es gibt einen wichtigen Unterschied:

`childNodes` enthält *alle* Arten von Kindknoten eines Elements. Das bedeutet, dass es nicht nur die HTML-Elemente (wie `<div>`, `<p>`, `<span>`, usw.) zeigt, sondern auch andere Knoten wie Textknoten (Text, der zwischen den HTML-Tags steht) oder Kommentar-Knoten (Kommentare, die mit `<!-- Kommentar -->` im HTML-Dokument hinzugefügt wurden).

`children` hingegen enthält nur die HTML-Elemente, also nur die Knoten vom Typ "Element" (Knotentyp 1). Es ignoriert alle anderen Arten von Knoten, wie zum Beispiel Textknoten oder Kommentare.

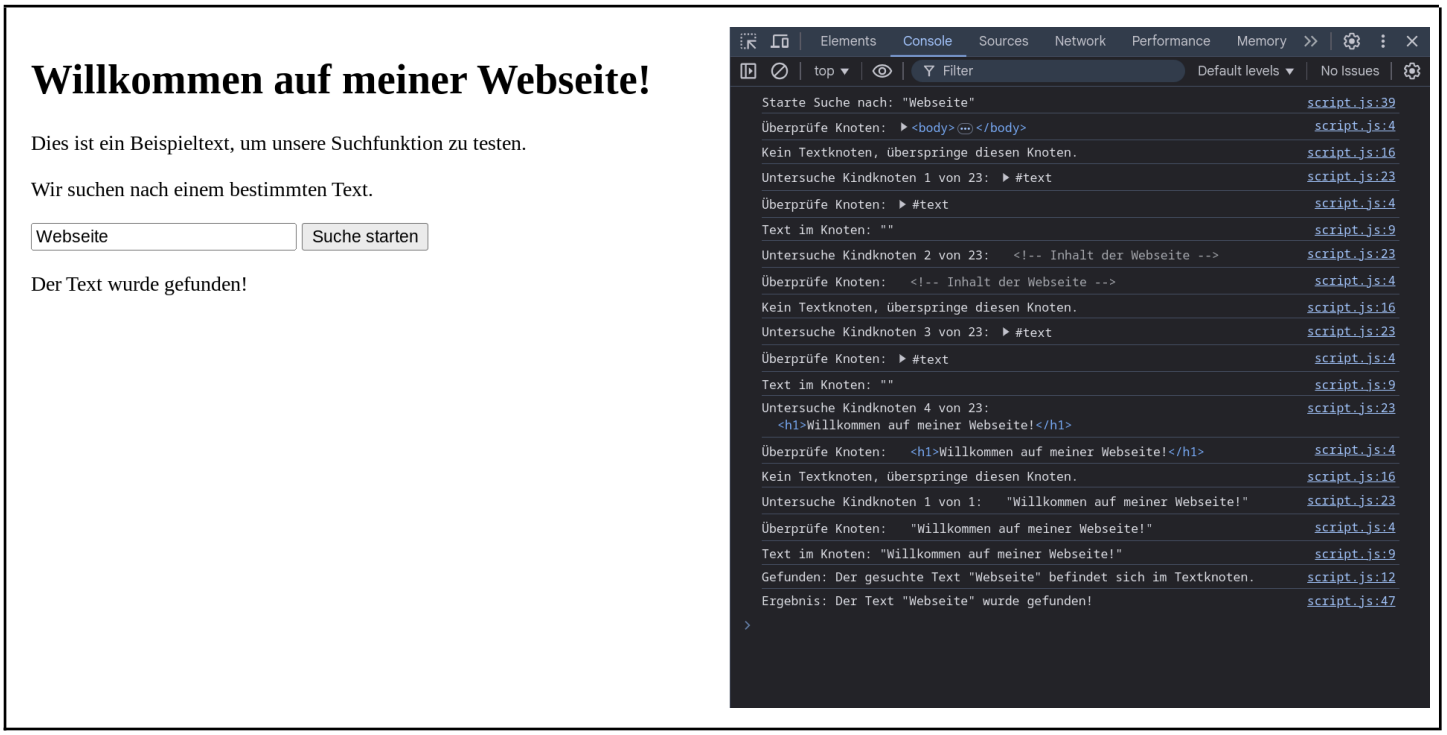
Zusammengefasst:

- **`childNodes`**: Zeigt alle Kinder an, einschließlich Text und Kommentare.
- **`children`**: Zeigt nur die HTML-Elemente an und ignoriert Text und Kommentare.

Das ist besonders nützlich, wenn man nur mit den tatsächlichen HTML-Elementen arbeiten möchte, ohne durch die ganzen Text- und Kommentar-Knoten gehen zu müssen.

Beispiel: Einen Text im HTML suchen

In diesem Beispiel soll ein Text eingegeben werden. Nach “Suche starten” wird der Text auf dieser Webseite gesucht. In der Console in den Devtools wird der aktuelle Zustand der Suche ausgegeben (in welchem Node gerade gesucht wird, was der Inhalt ist etc.)



Schritt 1: Erstelle die HTML-Datei

1. Öffne deinen Text-Editor:

Verwende einen einfachen Text-Editor wie Notepad (Windows), TextEdit (Mac, im "Nur-Text"-Modus), oder einen Code-Editor wie Visual Studio Code oder Atom.

2. Erstelle eine neue Datei:

Erstelle eine neue Datei und speichere sie als `index.html` in einem neuen Ordner auf deinem Computer. Du kannst den Ordner "Meine Webseite" oder einen anderen Namen nennen, der dir gefällt.

3. Schreibe den HTML-Code:

Kopiere den folgenden HTML-Code in die `index.html`-Datei:

index.html

```
<!DOCTYPE html>
  <html>
  <head>
      <meta charset="UTF-8">
      <title>Textsuche auf der Webseite</title>
  </head>
  <body>
      <!-- Inhalt der Webseite -->
      <h1>Willkommen auf meiner Webseite!</h1>
      <p>Dies ist ein Beispieltext, um unsere Suchfunktion zu testen.</p>
      <p>Wir suchen nach einem bestimmten Text.</p>

      <!-- Eingabefeld und Button für die Suche -->
      <input type="text" id="searchInput" placeholder="Gib den gesuchten Text ein">
      <button onclick="startSearch()">Suche starten</button>

      <!-- Bereich für die Ergebnisausgabe -->
      <p id="result"></p>

      <!-- Verlinkung zur JavaScript-Datei -->
      <script src="script.js"></script>
  </body>
  </html>

```


```
<!DOCTYPE html>
 <html>
 <head>
     <meta charset="UTF-8">
     <title>Textsuche auf der Webseite</title>
 </head>
 <body>
```

```
    <!-- Inhalt der Webseite -->
    <h1>Willkommen auf meiner Webseite!</h1>
    <p>Dies ist ein Beispieltext, um unsere Suchfunktion zu testen.</p>
    <p>Wir suchen nach einem bestimmten Text.</p>

    <!-- Eingabefeld und Button für die Suche -->
    <input type="text" id="searchInput" placeholder="Gib den gesuchten Text ein">
    <button onclick="startSearch()">Suche starten</button>

    <!-- Bereich für die Ergebnisausgabe -->
    <p id="result"></p>

    <!-- Verlinkung zur JavaScript-Datei -->
    <script src="script.js"></script>
</body>
</html>
```

4. Speichere die Datei:

Stelle sicher, dass du die Datei unter dem Namen `index.html` speicherst.

Schritt 2: Erstelle die JavaScript-Datei

1. Erstelle eine neue Datei im gleichen Ordner:
Öffne eine neue Datei in deinem Text-Editor und speichere sie als `script.js` im selben Ordner, in dem sich deine `index.html`-Datei befindet.

2. Schreibe den JavaScript-Code:
Kopiere den folgenden JavaScript-Code in die `script.js`-Datei:

Wenn man mit einer verschachtelten Datenstruktur wie dem DOM arbeitet, sind rekursive Funktionen oft nützlich. Die folgende Funktion durchsucht rekursiv ein Dokument nach Textknoten, die einen bestimmten String enthält. Rekursiv bedeutet, dass die Funktion sich für kleine Teiläste des Problems immer wieder selber aufruft (die Funktion searchTextNode).

script.js

```
// Diese Funktion sucht nach einem bestimmten Text in einem Textknoten
function searchTextNode(node, searchText) {
    // Zeige im Console, welcher Knoten gerade überprüft wird
    console.log(`Überprüfe Knoten:`, node);

    // Überprüfen, ob der aktuelle Knoten ein Textknoten ist
    // nodeType === 3 bedeutet, dass es ein Textknoten ist
    if (node.nodeType === Node.TEXT_NODE) {
        console.log(`Text im Knoten: "${node.nodeValue.trim()}"`);
        // Überprüfe, ob der Text im Knoten den gesuchten Text enthält
        if (node.nodeValue.includes(searchText)) {
            console.log(`Gefunden: Der gesuchte Text "${searchText}" befindet sich im Textknoten.`);
            return true; // Wenn der gesuchte Text gefunden wurde, gibt die Funktion 'true' zurück
        }
    } else {
        console.log(`Kein Textknoten, überspringe diesen Knoten.`);
    }

    // Wenn der aktuelle Knoten kein passender Textknoten ist, durchsuchen wir seine Kindknoten
    // Wir durchlaufen alle Kindknoten des aktuellen Knotens
    for (let i = 0; i < node.childNodes.length; i++) {
        // Ausgabe des Inhalts des Kindknotens
        console.log(`Untersuche Kindknoten ${i + 1} von ${node.childNodes.length}:`, node.childNodes[i]);

        // Ruft die Funktion rekursiv für jedes Kind auf
        if (searchTextNode(node.childNodes[i], searchText)) {
            return true; // Wenn der Text gefunden wurde, geben wir 'true' zurück
        }
    }

    // Wenn der Text nirgends gefunden wird, gibt die Funktion 'false' zurück
    return false;
}

// Diese Funktion wird aufgerufen, wenn der Benutzer auf den "Suche starten" Button klickt
function startSearch() {
    // Hol den gesuchten Text aus dem Eingabefeld
    const searchText = document.getElementById('searchInput').value;
    console.log(`Starte Suche nach: "${searchText}"`); // Ausgabe in der Konsole, um zu zeigen, dass die Suche startet

    // Suche nach dem Text im gesamten Dokument (im body)
    const found = searchTextNode(document.body, searchText);

    // Zeige das Ergebnis auf der Webseite an
    const resultElement = document.getElementById('result');
    if (found) {
        console.log(`Ergebnis: Der Text "${searchText}" wurde gefunden!`); // Ausgabe in der Konsole, wenn der Text gefunden wurde
        resultElement.textContent = 'Der Text wurde gefunden!'; // Zeigt "gefunden" an, wenn der Text da ist
    } else {
        console.log(`Ergebnis: Der Text "${searchText}" wurde nicht gefunden.`); // Ausgabe in der Konsole, wenn der Text nicht gefunden wurde
        resultElement.textContent = 'Der Text wurde nicht gefunden.'; // Zeigt "nicht gefunden" an, wenn der Text fehlt
    }
}

```

```
// Diese Funktion sucht nach einem bestimmten Text in einem Textknoten
```

```
function searchTextNode(node, searchText) {
    // Zeige im Console, welcher Knoten gerade überprüft wird
    console.log(`Überprüfe Knoten:`, node);

    // Überprüfen, ob der aktuelle Knoten ein Textknoten ist
    // nodeType === 3 bedeutet, dass es ein Textknoten ist
    if (node.nodeType === Node.TEXT_NODE) {
        console.log(`Text im Knoten: "${node.nodeValue.trim()}"`);
        // Überprüfe, ob der Text im Knoten den gesuchten Text enthält
        if (node.nodeValue.includes(searchText)) {
            console.log(`Gefunden: Der gesuchte Text "${searchText}" befindet sich im
Textknoten.`);
            return true; // Wenn der gesuchte Text gefunden wurde, gibt die Funktion
'true' zurück
        }
    } else {
        console.log(`Kein Textknoten, überspringe diesen Knoten.`);
    }

    // Wenn der aktuelle Knoten kein passender Textknoten ist, durchsuchen wir seine
Kindknoten
    // Wir durchlaufen alle Kindknoten des aktuellen Knotens
    for (let i = 0; i < node.childNodes.length; i++) {
        // Ausgabe des Inhalts des Kindknotens
        console.log(`Untersuche Kindknoten ${i + 1} von ${node.childNodes.length}:`,
node.childNodes[i]);

        // Ruft die Funktion rekursiv für jedes Kind auf
        if (searchTextNode(node.childNodes[i], searchText)) {
            return true; // Wenn der Text gefunden wurde, geben wir 'true' zurück
        }
    }

    // Wenn der Text nirgends gefunden wird, gibt die Funktion 'false' zurück
    return false;
}

// Diese Funktion wird aufgerufen, wenn der Benutzer auf den "Suche starten" Button
```

```
klickt
function startSearch() {
    // Hol den gesuchten Text aus dem Eingabefeld
    const searchText = document.getElementById('searchInput').value;
    console.log(`Starte Suche nach: "${searchText}"`); // Ausgabe in der Konsole, um zu
zeigen, dass die Suche startet

    // Suche nach dem Text im gesamten Dokument (im body)
    const found = searchTextNode(document.body, searchText);

    // Zeige das Ergebnis auf der Webseite an
    const resultElement = document.getElementById('result');
    if (found) {
        console.log(`Ergebnis: Der Text "${searchText}" wurde gefunden!`); // Ausgabe
in der Konsole, wenn der Text gefunden wurde
        resultElement.textContent = 'Der Text wurde gefunden!'; // Zeigt "gefunden" an,
wenn der Text da ist
    } else {
        console.log(`Ergebnis: Der Text "${searchText}" wurde nicht gefunden.`); //
Ausgabe in der Konsole, wenn der Text nicht gefunden wurde
        resultElement.textContent = 'Der Text wurde nicht gefunden.'; // Zeigt "nicht
gefunden" an, wenn der Text fehlt
    }
}
```

Erläuterung zu den Ausgaben auf der Konsole:

1. console.log bei jedem Schritt:
    - Bei jedem Knoten, der überprüft wird, zeigt console.log an, welcher Knoten gerade geprüft wird und ob es sich um einen Textknoten handelt.
    - Wenn der Knoten ein Textknoten ist, wird der Inhalt des Textknotens ausgegeben.
    - Wenn der Text im Knoten den gesuchten Text enthält, wird dies ebenfalls in der Konsole ausgegeben.
2. Ausgabe der Kindknoten:
    - Wenn die Funktion beginnt, die Kindknoten zu durchlaufen, wird jeder Kindknoten mit seiner Nummer (z.B. "Kindknoten 1 von 3") und seinem Inhalt ausgegeben.
    - Dies hilft dabei zu verstehen, welcher Teil des HTML-Dokuments gerade durchsucht wird.

3. Speichere die Datei:

Stelle sicher, dass du die Datei unter dem Namen `script.js` speicherst.

Schritt 3: Öffne die HTML-Datei im Browser

1. Öffne deinen Webbrowser:

Starte einen Webbrowser deiner Wahl (z. B. Chrome, Firefox, Edge).

2. Öffne die HTML-Datei im Browser:

- Gehe im Browser zu `Datei` > `Datei öffnen...` (in einigen Browsern heißt es auch `Open File...`).
- Wähle die `index.html`-Datei aus dem Ordner, in dem du sie gespeichert hast.

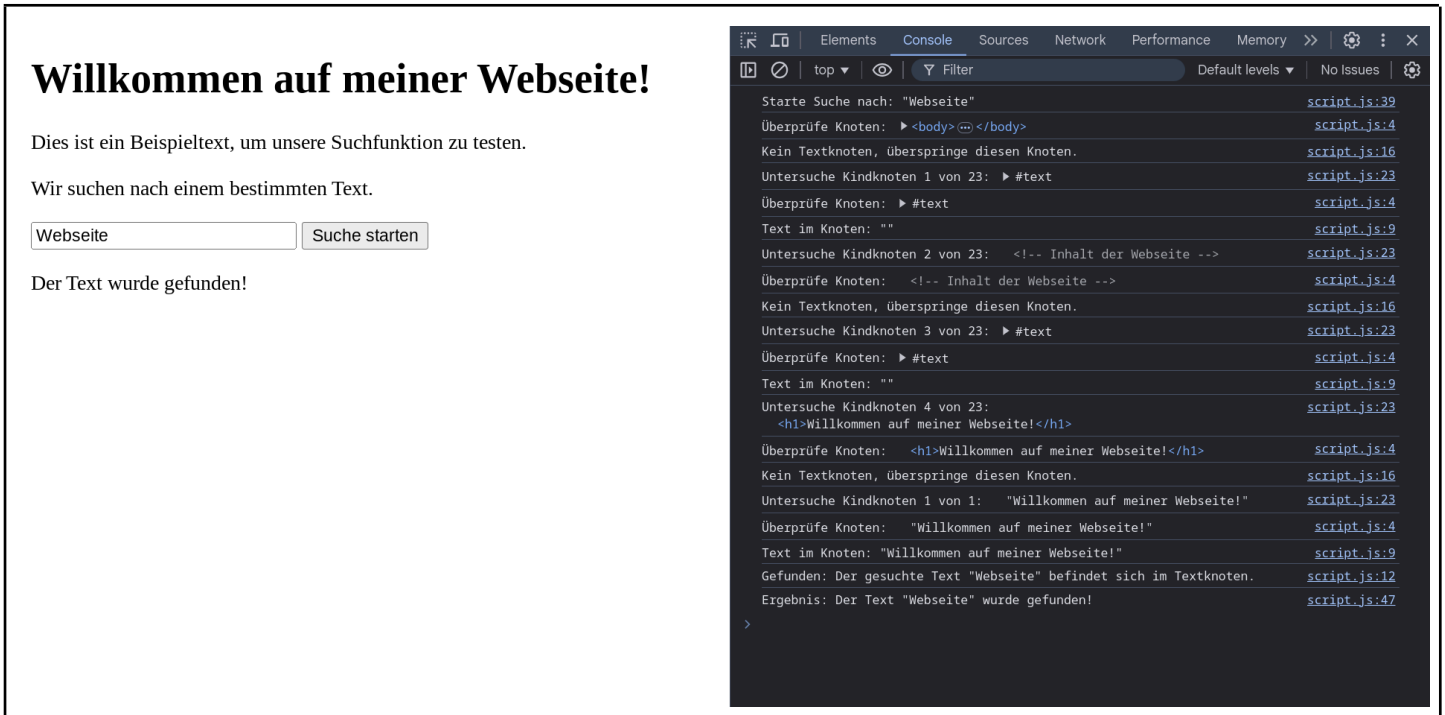
Alternativ kannst du auch den Ordner öffnen, in dem sich deine `index.html` befindet, und die Datei per Doppelklick direkt im Browser öffnen.

Schritt 4: Verwende die Entwicklerwerkzeuge (DevTools)

1. Öffne die Entwicklertools im Browser:
 - Rechtsklicke auf die Seite und wähle "Untersuchen" (Chrome, Edge) oder "Element untersuchen" (Firefox).
 - Alternativ kannst du die Entwicklertools auch über die Taste `F12` öffnen.

2. Gehe zum "Konsole"-Tab:
- In den Entwicklertools findest du oben verschiedene Tabs. Klicke auf "Konsole". Hier kannst du JavaScript-Befehle direkt ausführen und Fehlernachrichten sehen.

3. Teste deine Webseite:
- Gib einen beliebigen Text in das Eingabefeld auf deiner Webseite ein.
- Klicke auf den Button "Suche starten".
- Beobachte, wie das Ergebnis in der Konsole und auf der Webseite angezeigt wird.



Ausgabe auf der Konsole, wenn nach dem Text “Webseite” gesucht wird.

Genaue Erklärung zur Ausgabe auf der Konsole

Der Screenshot zeigt, wenn man ganz genau hinsieht, dass der erste Knoten “Untersuche Kindnoten 1 von 23” ein Textknoten (`#text`) ist. Aber anstatt wirklichen Text zu enthalten, zeigt der Knoten nur einen Zeilenumbruch (`"\n"`) und nur ein paar Leerzeichen an. Das ist der textContent ganz unten:

```
textContent: "\n        "
wholeText: "\n        "
```

Inhalt vom Kindknoten “1”

```
Starte Suche nach: "Webseite"                                          script.js:39
Überprüfe Knoten:  ▶ <body> ⋯ </body>                                  script.js:4
Kein Textknoten, überspringe diesen Knoten.                            script.js:16
Untersuche Kindknoten 1 von 23:  ▶ #text                               script.js:23
Überprüfe Knoten:                                                      script.js:4
▼ #text i
    assignedSlot: null
    baseURI: "file:///home/chronos/u-e568959cafa8e57060df481819bfd8bfe854c781/MyFiles/Downlo
  ▶ childNodes: NodeList []
    data: "\n        "
    firstChild: null
    isConnected: true
    lastChild: null
    length: 8
  ▶ nextElementSibling: h1
  ▶ nextSibling: comment
    nodeName: "#text"
    nodeType: 3
    nodeValue: "\n        "
  ▶ ownerDocument: document
  ▶ parentElement: body
  ▶ parentNode: body
    previousElementSibling: null
    previousSibling: null
    textContent: "\n        "
    wholeText: "\n        "
  ▶ [[Prototype]]: Text
Kein Textknoten, überspringe diesen Knoten.                            script.js:16
Untersuche Kindknoten 2 von 23:   <!-- Inhalt der Webseite -->         script.js:23
Überprüfe Knoten:   <!-- Inhalt der Webseite -->                       script.js:4
Kein Textknoten, überspringe diesen Knoten.                            script.js:16
Untersuche Kindknoten 3 von 23:  ▶ #text                               script.js:23
Überprüfe Knoten:  ▶ #text                                             script.js:4
Kein Textknoten, überspringe diesen Knoten.                            script.js:16
Untersuche Kindknoten 4 von 23:                                        script.js:23
  <h1>Willkommen auf meiner Webseite!</h1>
Überprüfe Knoten:   <h1>Willkommen auf meiner Webseite!</h1>           script.js:4
Kein Textknoten, überspringe diesen Knoten.                            script.js:16
Untersuche Kindknoten 1 von 1:   "Willkommen auf meiner Webseite!"     script.js:23
Überprüfe Knoten:   "Willkommen auf meiner Webseite!"                  script.js:4
Text im Knoten: "Willkommen auf meiner Webseite!"                      script.js:9
Gefunden: Der gesuchte Text "Webseite" befindet sich im Textknoten.    script.js:12
Ergebnis: Der Text "Webseite" wurde gefunden!                          script.js:47
```

Warum ist das so?

Wenn wir ein HTML-Dokument schreiben, verwenden wir oft Leerzeichen, Zeilenumbrüche oder Tabs, um den Code sauber und übersichtlich zu formatieren. Zum Beispiel:

```
<body>
    <h1>Willkommen!</h1>
    <p>Das ist ein Absatz.</p>
</body>
```

Zwischen den Tags (wie `<body>`, `<h1>`, `<p>`) gibt es einige unsichtbare Zeichen wie Leerzeichen oder Zeilenumbrüche (`\n`). Diese unsichtbaren Zeichen helfen uns, den Code besser zu lesen, sind aber für den Browser auch Knoten im DOM-Baum.

Was passiert genau?

Der Browser sieht jedes dieser unsichtbaren Zeichen – wie Leerzeichen und Zeilenumbrüche – als einen eigenen Textknoten an. Deshalb zeigt die Konsole einen `#text`-Knoten an, dessen Inhalt nur ein Zeilenumbruch (`"\n"`) oder einige Leerzeichen ist.

Das passiert, weil der JavaScript-Code alle Knoten durchläuft, auch die, die keine sichtbaren Zeichen oder echten Inhalte enthalten. Deshalb zeigt die Konsole den ersten Kindknoten als `#text` mit einem Zeilenumbruch an, der nichts anderes als ein unsichtbarer Teil des HTML-Codes ist.

Weitere Übungen

- Ändere den Text auf der Webseite und teste die Suchfunktion erneut.
- Füge neue HTML-Elemente hinzu, wie z. B. mehr Absätze oder Überschriften, um zu sehen, wie die Funktion darauf reagiert.

Genaue Erklärung zum Javascript

Grundidee des Codes

Der Code durchsucht das gesamte Dokument (alles, was du auf der Webseite siehst) nach einem bestimmten Text auf der Webseite. Dabei wird eine Technik namens **Rekursion** verwendet, die bedeutet, dass eine Funktion (die Funktion searchTextNode) sich selbst aufruft. Er beginnt an einem Punkt (dem sogenannten "Knoten") und schaut, ob dort der Text ist. Wenn der Text nicht gefunden wird, durchsucht der Code die "Kinder" dieses Punktes, also die Elemente, die unter diesem Punkt liegen. Das geht so lange weiter, bis der Text gefunden wird oder es keine weiteren Kinder mehr gibt, die durchsucht werden können.

Wie wird das umgestzt?

Der Code besteht aus zwei Funktionen:

1. `searchTextNode(node, searchText)`: Diese Funktion durchsucht das HTML-Dokument nach einem bestimmten Text. Sie geht durch alle Teile der Webseite und überprüft, ob der gesuchte Text in einem Textknoten vorhanden ist.
2. `startSearch()`: Diese Funktion wird ausgeführt, wenn der Benutzer auf den "Suche starten" Button klickt. Sie startet die Suche nach dem Text, den der Benutzer eingegeben hat, und zeigt das Ergebnis auf der Webseite an.

Erklärung des Codes Schritt für Schritt

1. **Aufruf der Funktion** `searchTextNode(node, searchText)` - Die Suchfunktion:
    - Diese Funktion nimmt zwei Eingaben:
        - `node`: Das ist ein Knoten (ein Punkt im Dokument), an dem die Suche beginnt.
        - `searchText`: Das ist der Text, den wir suchen.
2. **Prüfen, ob der Knoten ein Textknoten ist:**
    - `if (node.nodeType === Node.TEXT_NODE)`: Hier wird geprüft, ob der Knoten Text enthält (Textknoten). Wenn ja, wird der Text in der Konsole angezeigt.
    - `if (node.nodeValue.includes(searchText))`: Wenn der Text im Knoten den gesuchten Text enthält, wird eine Nachricht in der Konsole ausgegeben, und die Funktion gibt `true` zurück – das bedeutet, dass der Text gefunden wurde.
3. **Durchsuchen der Kindknoten:**
    - Wenn der aktuelle Knoten nicht der gesuchte Textknoten ist, geht die Funktion zu allen Kindknoten des aktuellen Knotens.

- `for (let i = 0; i < node.childNodes.length; i++)`: Diese Schleife durchläuft alle Kindknoten des aktuellen Knotens.
  - Für jeden Kindknoten ruft die Funktion **sich selbst** auf (`searchTextNode(node.childNodes[i], searchText)`). Das ist der rekursive Teil!
  - Die Funktion überprüft die Kindknoten auf die gleiche Weise, wie sie den Hauptknoten überprüft hat.

4. **Rückgabe der Funktion:**
   - Wenn der Text gefunden wurde, gibt die Funktion `true` zurück und die Suche endet.
   - Wenn der Text nirgends gefunden wird, gibt die Funktion `false` zurück.
5. **Die Funktion zum Starten der Suche:** `startSearch()`
   - Diese Funktion wird aufgerufen, wenn der Benutzer auf einen Button klickt.
   - Sie holt sich den Text aus einem Eingabefeld und startet die Suche im gesamten Dokument (`document.body`).
   - Das Ergebnis wird dem Benutzer auf der Webseite angezeigt.

Wie funktioniert die Rekursion?

Rekursion bedeutet, dass die Funktion **sich selbst wiederholt aufruft**, um ein Problem zu lösen. Stell dir das wie eine Kette von Aufgaben vor:

1. **Start:** Die Funktion beginnt an einem Knoten.
2. **Check:** Sie prüft, ob der gesuchte Text da ist.
3. **Kinder durchsuchen:** Wenn der Text nicht da ist, schaut sie bei allen "Kindern" des Knotens nach.
   - **Wiederholen:** Für jedes Kind ruft die Funktion sich selbst auf und wiederholt die gleichen Schritte.
4. **Ende:** Wenn die Funktion keinen Text findet oder keine Kinder mehr da sind, gibt sie `false` zurück.

Die Rekursion läuft so lange weiter, bis der gesuchte Text gefunden wird oder alle Knoten durchsucht wurden.

- **Rekursion** ist wie ein Detektiv, der seine Kollegen schickt, um in jedem Raum eines Hauses nach einem Hinweis zu suchen. Wenn der Detektiv selbst den Hinweis nicht findet, lässt er jeden Raum durchsuchen – und jeder Kollege macht das Gleiche, bis entweder der Hinweis gefunden ist oder es keine Räume mehr gibt.
- Dieser Code nutzt die Rekursion, um alle Teile einer Webseite nach einem bestimmten Text zu durchsuchen.

Erklärung des Javascript Zeile für Zeile

- Die Funktion `searchTextNode` durchsucht das gesamte HTML-Dokument nach einem bestimmten Text.
- Die Funktion `startSearch` wird aufgerufen, wenn der Benutzer auf den Button klickt, und zeigt das Ergebnis der Suche auf der Webseite und in der Konsole an.
- Der Code verwendet rekursive Aufrufe und Konsolenausgaben, um zu zeigen, welcher Teil des Dokuments durchsucht wird und ob der gesuchte Text gefunden wurde oder nich

Funktion `searchTextNode(node, searchText)`

Diese Funktion hat zwei Parameter:

- `node`: Der aktuelle Knoten (Teil) des Dokuments, den wir gerade untersuchen.
- `searchText`: Der Text, nach dem wir suchen.

```
// Zeige im Console, welcher Knoten gerade überprüft wird
console.log(`Überprüfe Knoten:`, node);
```

In der Konsole des Browsers wird angezeigt, welcher Knoten gerade überprüft wird. Das hilft uns, zu sehen, was der Code macht.

```
// Überprüfen, ob der aktuelle Knoten ein Textknoten ist
// nodeType === 3 bedeutet, dass es ein Textknoten ist
if (node.nodeType === Node.TEXT_NODE) {
```

Ein Textknoten ist ein Teil der Webseite, der nur Text enthält (also keine HTML-Tags wie `<div>` oder `<p>`). Der Code überprüft hier, ob der aktuelle Knoten ein Textknoten ist.

```
console.log(`Text im Knoten: "${node.nodeValue.trim()}"`);
```

Wenn der Knoten ein Textknoten ist, wird der Text, den er enthält, in der Konsole angezeigt. Die `.trim()` Methode entfernt unnötige Leerzeichen am Anfang und Ende des Textes.

```
// Überprüfe, ob der Text im Knoten den gesuchten Text enthält
if (node.nodeValue.includes(searchText)) {
    console.log(`Gefunden: Der gesuchte Text "${searchText}" befindet sich im
Textknoten.`);
    return true; // Wenn der gesuchte Text gefunden wurde, gibt die Funktion
```

```
'true' zurück
}
```

Der Code prüft, ob der gesuchte Text (`searchText`) im Text des Knotens (`node.nodeValue`) enthalten ist. Wenn ja, gibt die Funktion `true` zurück, was bedeutet, dass der gesuchte Text gefunden wurde, und zeigt in der Konsole an, dass der Text gefunden wurde.

```
} else {
    console.log(`Kein Textknoten, überspringe diesen Knoten.`);
}
```

Wenn der Knoten kein Textknoten ist (also ein HTML-Tag wie `<div>` oder `<p>`), wird in der Konsole angezeigt, dass dieser Knoten übersprungen wird.

```
// Wenn der aktuelle Knoten kein passender Textknoten ist, durchsuchen wir seine
Kindknoten
// Wir durchlaufen alle Kindknoten des aktuellen Knotens
for (let i = 0; i < node.childNodes.length; i++) {
    // Ausgabe des Inhalts des Kindknotens
    console.log(`Untersuche Kindknoten ${i + 1} von ${node.childNodes.length}:`,
node.childNodes[i]);

    // Ruft die Funktion rekursiv für jedes Kind auf
    if (searchTextNode(node.childNodes[i], searchText)) {
        return true; // Wenn der Text gefunden wurde, geben wir 'true' zurück
    }
}
```

Wenn der gesuchte Text nicht im aktuellen Knoten gefunden wurde, durchsucht die Funktion alle Kindknoten (die Teile, die in diesem Knoten enthalten sind). Das macht sie rekursiv, d. h. die Funktion ruft sich selbst auf, um jeden Kindknoten zu durchsuchen. Wenn der gesuchte Text in einem der Kindknoten gefunden wird, gibt die Funktion `true` zurück.

```
// Wenn der Text nirgends gefunden wird, gibt die Funktion 'false' zurück
return false;
```

Wenn der gesuchte Text in keinem der Knoten und Kindknoten gefunden wurde, gibt die Funktion `false` zurück. Das bedeutet, dass der Text nicht im Dokument vorhanden ist.

Funktion `startSearch()`

Diese Funktion wird ausgeführt, wenn der Benutzer auf den "Suche starten"-Button klickt.

```
// Hol den gesuchten Text aus dem Eingabefeld
const searchText = document.getElementById('searchInput').value;
console.log(`Starte Suche nach: "${searchText}"`); // Ausgabe in der Konsole, um
zu zeigen, dass die Suche startet
```

Der Text, den der Benutzer im Eingabefeld eingegeben hat, wird ausgelesen und in der Variable `searchText` gespeichert. In der Konsole wird angezeigt, dass die Suche beginnt und nach welchem Text gesucht wird.

```
// Suche nach dem Text im gesamten Dokument (im body)
const found = searchTextNode(document.body, searchText);
```

Die Funktion `searchTextNode` wird aufgerufen, um im gesamten Dokument (innerhalb des `<body>`-Tags) nach dem eingegebenen Text zu suchen. Das Ergebnis (`true` oder `false`) wird in der Variable `found` gespeichert.

```
// Zeige das Ergebnis auf der Webseite an
const resultElement = document.getElementById('result');
if (found) {
    console.log(`Ergebnis: Der Text "${searchText}" wurde gefunden!`); // Ausgabe
in der Konsole, wenn der Text gefunden wurde
    resultElement.textContent = 'Der Text wurde gefunden!'; // Zeigt "gefunden"
an, wenn der Text da ist
} else {
    console.log(`Ergebnis: Der Text "${searchText}" wurde nicht gefunden.`); //
Ausgabe in der Konsole, wenn der Text nicht gefunden wurde
    resultElement.textContent = 'Der Text wurde nicht gefunden.'; // Zeigt "nicht
gefunden" an, wenn der Text fehlt
}
```

Wenn der gesuchte Text gefunden wurde (`found` ist `true`), zeigt die Konsole an, dass der Text gefunden wurde, und der Text „Der Text wurde gefunden!“ wird auf der Webseite angezeigt.

Wenn der Text nicht gefunden wurde (`found` ist `false`), zeigt die Konsole an, dass der Text nicht gefunden wurde, und der Text „Der Text wurde nicht gefunden.“ wird auf der Webseite angezeigt.

Variablen in Javascript (const und let)

In JavaScript gibt es zwei wichtige Arten, Variablen zu deklarieren: `const` und `let`. Beide Schlüsselwörter werden verwendet, um Variablen zu deklarieren, aber sie haben unterschiedliche Bedeutungen und Anwendungsfälle.

Unterschied zwischen const und let (und var)

1. **`const` (Konstante):**
    - Wenn du eine Variable mit `const` deklarierst, bedeutet das, dass diese Variable *nicht erneut zugewiesen* werden kann. Das bedeutet, dass ihr Wert nach der Deklaration nicht mehr verändert werden darf. Die Variable bleibt „konstant“.
    - Wichtig: Bei `const` kannst du den Inhalt von Objekten oder Arrays verändern (zum Beispiel, Eigenschaften zu einem Objekt hinzufügen oder Werte in einem Array ändern), aber du kannst die Variable selbst nicht auf ein anderes Objekt oder Array zeigen lassen.
2. **`let`:**
    - Eine mit `let` deklarierte Variable kann *neu zugewiesen* werden, d.h. ihr Wert kann im Verlauf des Programms geändert werden.
    - `let` eignet sich, wenn du eine Variable hast, deren Wert du später anpassen möchtest, zum Beispiel in einer Schleife.
3. **`var`:**
    - War die ursprüngliche Definition von Variablen in JS und sollte auf keinen Fall mehr verwendet werden! Ist nur für die Rückwärtskompatiblität mit altem JS Code noch vorhanden. Bringt viele Probleme mit sich, weil es nicht so funktioniert, wie man meinen könnte.

Warum const im Code-Beispiel?

In dem JavaScript-Code oben wurde `const` für die Deklaration von `searchText` und `resultElement` verwendet, weil:

- **`searchText`:** Diese Variable speichert den Text, den der Benutzer ins Eingabefeld eingibt. Sie wird nur einmal beim Start der Suche zugewiesen und danach nicht mehr verändert. Da der Wert dieser Variablen nach ihrer Deklaration nicht erneut zugewiesen wird, ist `const` die richtige Wahl.
- **`resultElement`:** Diese Variable speichert die Referenz auf das HTML-Element (`<p id="result">`), in das das Ergebnis der Suche geschrieben wird. Wir weisen `resultElement` nur einmal einen Wert zu (die Referenz auf das Element) und ändern diese Referenz nicht mehr, daher verwenden wir `const`.

Wann sollte man let verwenden?

Du solltest `let` verwenden, wenn der Wert einer Variablen während der Ausführung deines Codes verändert werden soll. Ein Beispiel wäre die Verwendung von `let` in einer Schleife:

```
for (let i = 0; i < 10; i++) {
  console.log(i); // 'i' wird in jeder Schleifeniteration neu zugewiesen
}
```

Hier wird `let` verwendet, weil `i` sich in jeder Schleifeniteration verändert.

Fazit

- Verwende **`const`**, wenn du sicher bist, dass eine Variable nach ihrer Initialisierung nicht erneut zugewiesen wird. Das macht den Code sicherer und besser lesbar, weil klar ist, dass diese Variable im Verlauf des Programms nicht geändert wird.
- Verwende **`let`**, wenn der Wert der Variable im Verlauf des Programms geändert oder neu zugewiesen werden muss.
- In den Anfängen von JS gab es auch **`var`**, das unbedingt vermeiden: kann durch seine Scoping-Regeln und das Hoisting unerwartete Fehler im Code verursachen.

Javascript - Wichtige Methoden

Die folgenden Methoden werden verwendet, um ein Element oder eine Gruppe von Elementen basierend auf ihrem Typ, ihren Eigenschaften, ihren Eigenschaftswerten oder ihrer Position auf dem Bildschirm relativ zum Ansichtsfenster abzurufen.

getElementById

Dies getElementById ist eine grundlegende Methode zur DOM-Manipulation. Sie ermöglicht es uns, ein bestimmtes HTML-Element anhand seiner eindeutigen Kennung (ID) auszuwählen und darauf zuzugreifen. Diese Methode ist effizient und wird häufig für gezielte Interaktionen mit Elementen auf einer Seite verwendet.

```
let element = document.getElementById('myElement');
```

getElementsByClassName

Mit dieser getElementsByClassName Methode rufen wir eine Sammlung von Elementen anhand ihres Klassennamens ab. Sie ist nützlich, um Änderungen an mehreren Elementen vorzunehmen, die eine gemeinsame Klasse haben.

```
let elements = document.getElementsByClassName('myClass');
```

getElementsByTagName

Mit getElementsByTagName können wir Elemente anhand ihres Tag-Namens auswählen. Diese Methode ist praktisch, wenn Sie mit einer Gruppe von Elementen desselben Typs arbeiten.

```
let paragraphs = document.getElementsByTagName('p');
```

Abfrageselektor

Mit dieser querySelector Methode können wir das erste Element auswählen, das einem angegebenen CSS-Selektor entspricht . Sie bietet eine leistungsstarke Möglichkeit, auf Elemente mit komplexeren Kriterien zuzugreifen.

```
let element = document.querySelector('.myClass');
```

querySelectorAll

Gibt ähnlich wie der querySelector eine NodeList zurück, querySelectorAll die alle Elemente enthält, die einem angegebenen CSS-Selektor entsprechen. Dies ist nützlich, um mehrere Elemente gleichzeitig zu verarbeiten.

```
let elements = document.querySelectorAll('.myClass');
```

elementFromPoint

Mit dieser elementFromPoint Methode können wir das oberste Element an einem bestimmten Koordinatensatz auf der Seite abrufen. Dies ist für Szenarien von Vorteil, in denen präzise Interaktionen basierend auf Maus- oder Berührungsereignissen des Benutzers erforderlich sind.

```
let element = document.elementFromPoint(x, y);
```

Die folgenden Methoden werden verwendet, um einen neuen Knoten zu erstellen und ihn entweder am Ende, vor oder nach einem bestimmten Knoten an ein übergeordnetes Element anzuhängen (hinzuzufügen). Diese Methoden werden auch verwendet, um ein Element zu entfernen und untergeordnete Elemente innerhalb eines Elements zu ersetzen, entweder von einem Element zu einem anderen oder durch Leeren.

createElement

Die createElementMethode ist wichtig für die dynamische Erstellung von HTML-Elementen in JavaScript. Wir können diese Elemente dann nach Bedarf bearbeiten und an das DOM anhängen. Diese Methode wird häufig beim Erstellen von Single Page Applications verwendet.

```
let newElement = document.createElement('div');
```

appendChild

Mit dieser Methode haben wir am Ende ein neues untergeordnetes Element in ein vorhandenes übergeordnetes Element eingefügt. Sie wird häufig verwendet, um einer Webseite dynamisch Inhalt hinzuzufügen.

```
parentElement.appendChild(newElement);
```

after

Die after Methode, Teil der ParentNode-Schnittstelle, fügt eine Reihe von Node-Objekten oder

DOMString-Objekten unmittelbar nach dem angegebenen untergeordneten Objekt ein.

```
element.after(newChild1, newChild2, ...);
```

before

Ähnlich wie after fügt die before Methode einen Satz von Node-Objekten oder DOMString-Objekten unmittelbar vor dem angegebenen untergeordneten Element ein.

```
element.before(newChild1, newChild2, ...);
```

replaceChildren

Die replaceChildren Methode vereinfacht den Prozess des Ersetzens der untergeordneten Elemente eines Elements durch einen neuen Satz von Knoten. Dies kann insbesondere dann nützlich sein, wenn der Inhalt eines Containerelements dynamisch aktualisiert und alle untergeordneten Elemente entfernt werden.

```
parentElement.replaceChildren(newChild1, newChild2, ...);
```

remove

Die removeMethode entfernt den angegebenen Knoten aus dem DOM. Dies ist eine sauberere Alternative zur Einstellung parentNode.removeChild(node).

```
element.remove();
```

Die folgenden Methoden werden verwendet, um die Attribute und Stile eines Elements dynamisch zu bearbeiten und so dynamische und reaktionsschnelle Anpassungen in einer Webanwendung zu ermöglichen.

setAttribute

Mit dieser setAttribute Methode können wir den Wert eines Attributs für ein bestimmtes Element festlegen oder ändern, was Flexibilität bei der Änderung von HTML-Elementen bietet. Mit dieser removeAttribute Methode können Sie das Attribut eines Elements auch vollständig entfernen.

```
element.setAttribute('src', 'new-image.jpg');
```

classList

Die classList Eigenschaft bietet Methoden wie „Hinzufügen“, „Entfernen“ und „Umschalten“, um die Klassen eines Elements einfach zu bearbeiten. Dies ist besonders für dynamisches Styling nützlich.

```
// add a css class
element.classList.add('newClass');

// remove a css class
element.classList.add('newClass');
```

Die folgenden Methoden werden zum dynamischen Scrollen einer Webseite oder eines Scroll-Containers verwendet und ermöglichen so eine reibungslose und kontrollierte Navigation oder Präsentation von Inhalten.

scrollInToView

Die scrollIntoView Methode ist eine praktische Möglichkeit, das angegebene Element reibungslos in den sichtbaren Bereich des Browserfensters zu scrollen. Dies ist insbesondere bei langen Seiten oder dynamisch geladenen Inhalten nützlich.

```
element.scrollIntoView({ behavior: 'smooth' });
```

scrollBy

Mit dieser scrollBy Methode können wir von der aktuellen Scrollposition aus einen bestimmten Betrag scrollen. Dies ist nützlich, um sanfte Scrolleffekte zu implementieren oder auf Benutzerinteraktionen zu reagieren.

```
window.scrollBy({
  top: 100,
  left: 0,
  behavior: 'smooth',
});
```

Die folgenden beiden Methoden dienen zum reagieren auf Benutzerinteraktionen. Auf Grundlage dieser Interaktionen können Aktionen ausgeführt werden, oder sie können bei Bedarf verhindert werden.

addEventListener

Mit addEventListener können wir Ereignishandler an Elemente anhängen, sodass wir auf Benutzerinteraktionen achten und entsprechend handeln können.

```
element.addEventListener('click', function() {
    // Handle click event
});
```

preventDefault

Obwohl es sich nicht um eine Methode des DOM selbst handelt, wird die Methode preventDefault häufig in Eventhandlern verwendet, um die mit einem Event verknüpfte Standardaktion zu verhindern, z. B. das Verhindern des Absendens eines Formulars oder der Navigation über einen Link. Dies kann als die Essenz oder der Kern von Single Page Applications betrachtet werden.

```
element.addEventListener("click",(event)=>{
    event.preventDefault();
});
```

animate

Die animate Methode dient zum sanften Animieren von Änderungen an CSS-Eigenschaften. Sie gibt ein Animation-Objekt zurück, das gesteuert und nach Informationen zur Animation abgefragt werden kann.

```
element.animate(keyframes, options);
```

requestFullscreen

Mit dieser requestFullscreenMethode können wir ein Element den gesamten Bildschirm einnehmen lassen. Dies ist besonders nützlich für immersive Web-Erlebnisse oder die Medienwiedergabe.

```
element.requestFullscreen();
```