DOM

Prof. Dr.-Ing. Andreas Heil

v1.1.1

Lernziele

Grundlagen des Document Object Models kennen lernen

DOM - Document Object Model

- Das Document Object Model (DOM) ist eine API, die den programmatischen Zugriff auf HTML (und XML) Dokumente ermöglicht
- JavaScript und DOM waren ursprünglich stark gekoppelt
- Damalige Browser (Netscape oder Internet Explorer) wiesen eine eigene DOM Implementierung auf
- Inzwischen (seit 2001) ist DOM ein eigener W3C Standard[^1]

Geschichte

- Einfaches DOM bereits in Netscape 2.0
- Ab Netscape 4.0 und IE 4.0 divergierten die DOMs der Browser sehr stark
- W3C DOM Level 1 wurde bereist im Oktober 1998
- Durch die Standardisierung nun nicht mehr nur via JavaScript adressierbar (auch Java, WebAssembly etc.)

DOM Struktur

- Hierarchische Struktur
- Windows als Übergeordnetes Element einer Web-Seite
- Document ist Kind mit den zu manipulierenden Elementen

```
window
  ├─ location
  ├─ frames
  ├─ ...
  └─ document
       ├─ links
       ├─ anchors
       ├─ images
       ├─ ...
       ├─ stylesheets
       └─ body
```

Adressierung von Objekten im DOM

- Über Ihre ID oder über Ihren Namen adressiert werden (muss eindeutig in der gesamten Baumstruktur sein)
- Über den Index in der Hierarchie (Position im Array)
- Über die Beziehung zum Eltern-, Kind- oder Geschwisterelement (`parentNode` , `previousSibling` , `nextSibling` , `firstChild` , `lastChild` , `childNodes` -Array)

DOM Beispiel

- Das erste `div`-Element besitzt die ID *firstName*
- Es enthält ein Textelement, dass über `childNodes[0]` adressiert werden kann
- D.h. der Text ist kein Wert des `div`-Elements sondern der Wert des ersten Kindelements des `div`-Elements

```
<div id="firstName">
Andreas
</div>
<div id="lastName">
Heil
</div>
```

▶ Beispiel: mouseover.html

Event Handler

- Wenn ein Event (dt. Ereignis) auftritt, wird ein sog. *Event Handler*^2 ausgeführt
- Beispiele hierfür:
  - `mouseover` oder `mouseout`

Event Handler Beispiel

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>mouseover Example</title>
  <meta http-equiv="Content-Script-Type" content="text/javascript"></meta>
</head>
<body>
  <div id="firstName"
       onmouseover="document.getElementById('firstName').childNodes[0].textContent = 'Andreas'"
       onmouseout="document.getElementById('firstName').childNodes[0].textContent = 'A.'">
      A.
  </div>
  <div id="lastName">Heil</div>
</body>
</html>
```

Hinweise

- Standardsprache für Skripte, die in Attributen wie im Beispiel
- DOM Elemente können via Skript modifiziert werden

```
<head>
  ...
  <meta http-equiv="Content-Script-Type" content="text/javascript"></script>
  ...
<head>
```

DOM Aufbau

```
├─ DOCTYPE: html
└─ HTML xmlns="http://www.w3.org/1999/xhtml"
   ├─ HEAD
   │  ├─ #text:
   │  ├─ TITLE
   │  │  └─ #text: mouseover Example
   │  └─ #text:
   ├─ #text:
   └─ BODY
      ├─ #text:
      ├─ DIV id="firstName"
      │      onmouseover="document.getElementById('firstName').childNodes[0].textContent = 'Andreas'"
      │      onmouseout="document.getElementById('firstName').childNodes[0].textContent = 'A.'"
      │  └─ #text: A.
      ├─ #text:
      ├─ DIV id="lastName"
      │  └─ #text: Heil
      └─ #text:
```

Document Tree (1)

```
<body>
  <div id="content">
    <h1>Prof. Dr.-Ing. Andreas Heil</h1>
    <p>Methoden des Software Engineerings</p>
    <p>Unterrichtet <em>hot s**t</em> Fächer.</p>
    <hr>
  </div>

  <div id="nav">
    <ul>
      <li>Web Application Development</li>
      <li>DevOps</li>
      <li>Cloud Computing</li>
    </ul>
  </div>
</body>
```

Document Tree (2)

```
...
|
body
├─ div
|  ├─ h1
|  ├─ p
|  ├─ p
|  |  └─ em
|  └─ hr
└─ div
   └─ ul
      ├─ li
      ├─ li
      └─ li
```

Document Tree and Nodes

- Jedes HTML-Dokument kann als Baum verstanden werden
- Die Dokumenten-Struktur ist relevant, z.B. für CSS-Selektoren
- Jedes Element verfügt über Eigenschaften, die durch das Node-Objekt vorgegeben

Nodes - Properties (1)

Property	Beschreibung
nodeType	Nummer, die den Typ des Nodes beschreibt (s. treeOutline.html)
nodeName	Name des Node,s abhängig vom Typ
parentNode	Referenz zum übergeordneten Node
childNodes	Nur-Lese Array mit den Kind-Nodes, Länge 0 wenn keine vorhanden

Nodes - Properties(2)

Property	Beschreibung
{previous|next}Sibling	Vorheriges bzw. nächstes Element, *null* wenn kein Element existiert
attributes	Nur-Lese Array, das *Attr*-Instanzen als Attribute des Nodes enthält

Nodes - Methoden (1)

Methode	Beschreibung
hasAttributes()	Liefert wahr falls der Node Attribute besitzt
hasChildNodes()	Liefert wahr, falls der Node untergeordnete Elemente besitzt
appendChilde(Node)	Fügt den spezifizierten Node an an das Ende der untergeordneten Elemente hinzu
insertBefore(Node1, Node2)	Fügt Node1 direkt vor Node2 in die Liste der untergeordneten Elemente hinzu

Nodes - Methoden (2)

Methode	Beschreibung
removeChildNode(Node)	Entfernt den spezifizierten Node aus den untergeordneten Elementen
replaceChild(Node1, Node2)	Ersetzt Node2 durch Node1 in der Liste der untergeordneten Elemente

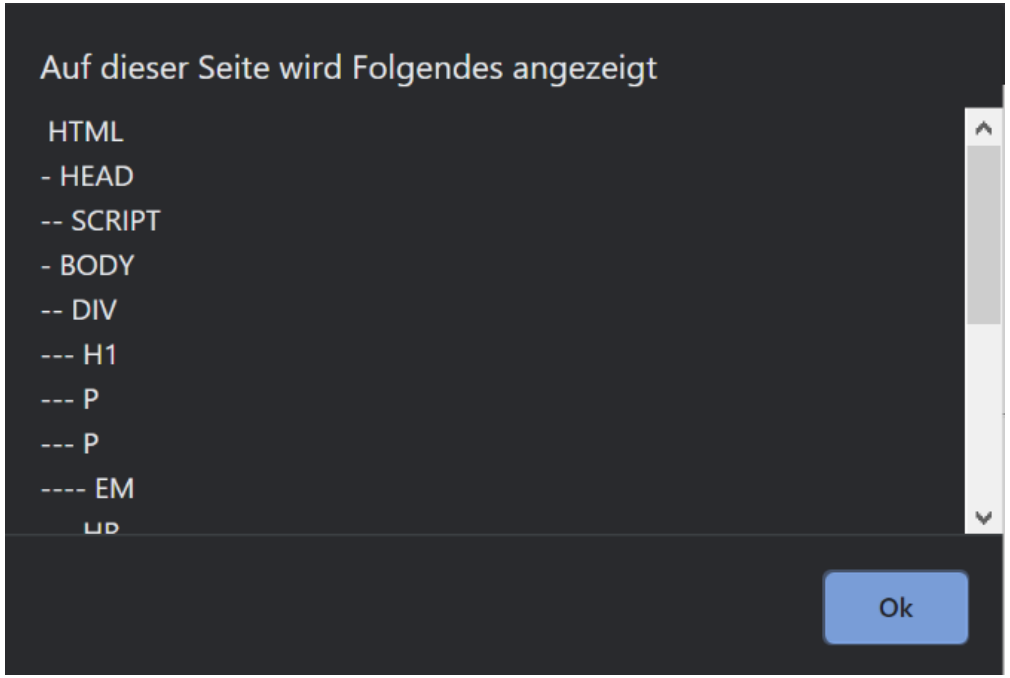
Traversieren des DOM

- HTML-Element über

  `document.documentElement`

```
function treeOutline() {
      return subtree(document.documentElement, 0);
}
```

```
 function subtree(node, level) {
       var retVal = "";
       var elementType = window.Node ? Node.ELEMENT_NODE : 1;
       if (node.nodeType == elementType) {
           retVal += printName(level, node.nodeName);
           var children = node.childNodes;
           for (var i = 0; i < children.length; i++) {
               retVal += subtree(children[i], level + 1);
           }
       }
       return retVal;
   }
```



▶ Beispiel: treeOutline.html

JavaScript - Grundlagen

Was ist JavaScript?

Wikipedia^3 sagt...

- ... dynamisch typisierte, objektorientierte, aber klassenlose Skriptsprache
- ... unter anderem auf der Basis von Prototypen
- ... lässt sich je nach Bedarf objektorientiert, prozedural oder funktional programmieren

▶ Skriptsprache bedeutet interpretiert, wenig bis keine Deklarationen

Woher kommt JavaScript?

- Hieß ursprünglich *LiveScript*
- Wurde zum Einbetten von Java-Applets genutzt
- Basiert auf dem standardisieren ECMAScript

▶ Ähnelt C mehr als Java

C vs. JavaScript

```
i = 42;
i = i * 10 + (i / 42);

while(i >= 0) {
  sum += i*i; // Kommentar
}

for (i = 0; i < 100; i++) {
  /* Kommentar */
}

if (i < 3) {
  i = foo(i);
} else {
  i =* .01;
}
```

Sprachkonstrukte

Die meisten Operatoren aus C existieren in JavaScript

```
* / % + - ! >= <= > < > && || ?:
```

Funktionen

```
function foo(i)
{
  return i;
}
```

Schleifenkonstrukte

```
continue / break / return
```

Dynamische Typisierung

```
var i;         // kein Typ
typeof i == 'undefined'
i = 32;        // Jetzt typeof i == typeof 32 == 'number'
i = 'foobar' // Jetzt typeof i == typeof ''foobar' == 'string'
i = 'true'   // Jetzt typeof i == `boolean
```

- `use strict` *strict*-Mode erfordert das deklarieren von Variablen (ab ECMAScript 5)^4
- Variablen haben immer den Typ der letzten Zuweisung
- Primitive Typen in JS: *undefined*, *number*, *string*, *boolean*, *function*, *object*

Gültigkeitsbereiche von Variablen

Zwei Gültigkeitsbereiche (engl. scopes): *global* und *function local*

```
var globaleVariable;

function foobar() {
  var lokaleVariable; // lokaleVariable2 gibt es schon
                      // mit Typ 'undefined'
  if (globaleVariable > 0) {
    var lokaleVariable2 = 2;
  }
  // lokaleVariable2 hat hier noch Gültigkeit 😱
}
```

Hoisting

Spezielles Verhalten von JavaScript-Interpretern bei er Deklaration von Variablen

- Beim Aufruf einer Funktion sucht der Interpreter **alle** lokal definierten variablen in der Funktion
- Deklaration findet somit sofort bei Eintritt in Funktion statt
- Initialisierung bzw. Zuweisung findet erst bei der Nutzung der Variable statt

Hoisting (Forts.)

```
var foo = 42;
function foobar() {
  if (globaleVariable > 0) {
    var foo = 2 ;
  }
}
```

Sieht im Interpreter ungefähr so aus:

```
var foo = 42;
function foobar() {
  var foo;        // undefined, überlagert globales foo
  if (globaleVariable > 0) {
    var foo = 2; //  Zuweisung erst hier
  }
}
```

Hoisting bei Funktionen

Hoisting findet auch bei Funktionen-Deklarationen statt

- Funktionen können vor der Deklaration aufgerufen werden (vgl. C-Prototypen, Single-Pass-Compiler vs. Multi-Pass-Compiler)
- Gilt nicht für anonyme Methoden

```
document.write(typeof foo);  // liefert function
foo();                       // wird ausgeführt
document.write(typeof bar); // liefert undefined
bar();                       // liefet fehler
function foo() {};           // Hoisting von Deklaration und Implementierung
var bar = function () {};    // Nur Hoisting der Deklaration
```

Funktions-Deklarationen gemäß ECMAScript (1/5)

Deklaration

```
function sum(a, b) {
  return a + b;
}
```

Aufruf

```
document.write(sum(40, 2));

function sum(a, b) {
  return a + b;
}
```

Funktions-Deklarationen gemäß ECMAScript (2/5)

Anonyme Funktion / Funktions-Ausdruck

Deklaration

```
// anonyme Funktion / Funktions-Ausdruck
let sum = function(a, b) {
  return a + b;
}
```

Aufruf

```
let sum = function(a, b) {
  return a + b;
}

document.write(sum(40, 2));
```

Funktions-Deklarationen gemäß ECMAScript (3/5)

Anonyme Funktion / Funktions-Ausdruck

Deklaration

```
(function(a, b) {
   return a + b;
})();
```

Aufruf

```
document.write((function(a, b) {
   return a + b;
})(40,2));
```

Funktions-Deklarationen gemäß ECMAScript (4/5)

Konstruktor-Methode

```
let sum = new Function('a','b', 'return a + b');
```

Aufruf

```
let sum = new Function('a','b', 'return a + b');
document.write(sum(40, 2));
```

Funktions-Deklarationen gemäß ECMAScript (5/5)

Arrow Function (Lambda Äquivalent)

Deklaration

```
var sum = (a, b) => {
  return a + b;
}
```

Aufruf

```
var sum = (a, b) => {
  return a + b;
}

document.write(sum(40, 2));
```

Vorteile bei Arrow-Funktionen

- Nicht "überschreibbar", wenn `const` genutzt
- Scope der *this*-Referenz bezieht sich auf die umgebende Funktion!

```
const sum = (a, b) => {
return a+b;
}

sum = (a, b) => {
return a*b;
}

document.write(sum(40,2));
```

Probleme bei Gültigkeitsbereichen

- Globale Variablen in Browsern können Konflikte mit anderen Modulen verursachen (gleiche global Variable)
- Hoisting durch Überlagern globaler Variablen vor der Initialisierung
- Hoisting bei Funktionen nur bei Funktions-Deklarationen
- Manche JavaScript Guidelines empfehlen alle `var`-Deklaration am Funktionsanfang
- ECMAScript 6 führte Non-Hoisting, 'let' mit Gültigkeitsbereichen, 'const' mit expliziten Gültigkeitsbereiten ein
- Manche Entwicklungsumgebungen lassen zwar kein `var` aber dafür aber `let` und `const` zu

Type: number

- Es gibt nur einen eonzigen Typ für Zahlen: *number*
- *number*-Variablen werden immer als 64-bit Floating Point gespeichert
- `NaN`, `Infinity` sind ebenfalls vom Typ *number*
- `1/0 == Infinity`
- `Math.sqrt(-1) == NaN`
- `(0.1 + 0.2) == 0.3` ❌ Fließkommaarithmetik
- Bitweise Operatoren (~, &, |, ^, >>, <<, >>>) sind 32Bit-Operationen!

Type: string

- Variable Länge
- `+` ist Operator für Konkatenation
- Zahlreiche hilfreiche Funktionen
  - `indexOf()`, `charAt()`, `match()`, `search()`, `replace()`, `toUpperCaser()`, `toLowerCase()`, `slice()`, `substr()` etc.
  - `'foo'.toUpperCase() // FOO`

Type: boolean

- Entweder *true* oder *false*
- Werte werden entweder als wahr oder falsch interpretiert
- Falsch
  - `false`, `0`, `null`, `undefinded`, `NaN`
- Wahr
  - Alles was nicht falsch ist, alle Objekte, nichtleere Strings, Zahlen ungleich Null (0), Funktionen etc.)

Type: undefined und null

- `undefined` - kein Wert zugewiesen
- `null` - Gemäß ECMAScript Spezifikation: "null is a primitive value that represents the intentional absence of any object value."^5
  - `typeof null // liefert object`
  - `null` ist "falsch"
  - Beim Zugriff auf `null` wird ein `TypeError` geworfen

Type: function

- Hoisting bei "normalen" Deklarationen, können also vor Deklaration genutzt werden
- Können mit mehr oder weniger Argumenten als in der Deklaration aufgerufen werden
- Nicht spezifizierte Argumente haben den Wert *undefined*
- Liefern immer einen Wert zurück (*undefined*)

```
var foobar = function foobar(x) {
  if (x <= 1) {
    return 1;
  }
  return x * foobar(x-1);
}
document.write(typeof foobar == 'function'); // true
document.write(foobar.name == 'foobar'); // truename == 'foobar';
```

Type: object

- Ungeordnete Paare von Werte-Paaren (engl. name value pair): *Properties*
- `var foo = {}`
- `var bar = {name: "Heil", age: NaN, department: "Computer Science"};`
- `Zugriff über Property oder wie in einer Hash-Table
  - `bar.name` oder `bar["name"]`
  - `foo.name` ist *undefined*

Properties können hinzugefügt und entfernt werden

Hinzufügen:

```
var foo = {};
foo.Name = "Andreas"; // foo.Name liefert "Andreas"
```

Entfernen:

```
var foo = {name: "Andreas"};
delete foo.Name; // foo hat nun keine Properties mehr
```

Enumerationen (via Object.keys):

```
Object.keys({name: "Andreas", age: NaN}) = ["name", "age"]
```

Arrays

- `var arr = [1,2,3,4];`
- `typeof arr == 'object'`
- Können lückenhaft und polymorph sein
  - `arr[5] = "A. Heil"`
  - [1, 2, 3, 4, , "A. Heil"]
- Analog zu string eine Vielzahl an Methoden:
  - `push` , `pop` , `shift` , `unshift` , `sort` , `reverse` , `splice` etc.
- Kann Properties enthalten
  - `arr.Name = "Mein Array"` - Speichern von Werten in Properties
  - `arr.length = 0;` - Ups, was passiert da wohl?

Date

- `var date = new Date();`
- Vom Typ *object*
- Speichert kein Datum, sondern Anzahl der Millisekunden seit Mitternacht 1. Januar, 1970 UTC
- Muss Zeitzonen in Betracht ziehen
- Keine gute Idee für feste Daten (z.B. Geburtstag)
- Zahlreiche Methoden zur Manipulation
  - `date.valueOf() > 123459316314
  - `date.toISOString() > '2021-03-21T09:45:00.123Z'
  - `date.toLocaleString() > '21/3/2021, 09:45:00 AM'

RegEx

- `let re = /ab+c/i;` als Literal
- `let re = new RegExp('ab+c', 'i')` Konstruktor mit String-Pattern als erstes Argument
- `let re = new RegExp(/ab+c/, 'i')` Konstruktor Mit RegEx Literal als erstes Argument (ab ECMAScript 6)
- `exec()` und `test()`

Exceptions

- Wird oft genutzt um Fehler im Programm zu behandeln
- Stoppt die Programmausführung mit einem Fehler
- Exceptions wandern den Stack hinauf, können mit `try/catch` behandelt werden

```
try {
  funktionExistiertNicht();
} catch (err) {
  console.log("Fehler: Funktion exisitert nicht", err.name, err.message);
}
```

Finally

- Exceptions könne mit `throw` geworfen werden

```
try {
  throw "Fehler";
} catch (errstr) { // err === "Fehler"
  console.log('Exception: ', errstr)
} finally {
  // wird nach try/catch ausgeführt
}
```

JavaScript in HTML-Seiten einbetten

- Einbinden über dedizierte Datei:

```
<script type="text/javascript" src="foobar.js"></script>
```

- Inline

```
<head>
<script>
function foobar() {
..
}
</script>
</head>
```

Selbstausführende Funktionen (1)

Bereits kennen gelernte Probleme:

- Globaler Scope für Variablen
- Herkömmliche Strukturen wie unten in großen Anwendungen irgendwann unübersichtlich

```
var foo = 'Hello';
var bar = 'World!';

function baz(){
  return foo  + ' ' + bar;
}

alert(baz());
```

Selbstausführende Funktionen (2)

- Anonyme Funktion
- Scope von darin deklarierten Variablen ausschließlich innerhalbe der anonymen Funktion

```
(function(){
  // Scope innerhalb der anonym. Funtion
})
```

Selbstausführende Funktionen (3)

Funktion wird durch `()` direkt ausgeführt

```
(function(){
  // Scope innerhalb der anonym. Funtion
})()
```

Kurzform

```
!funtion(){
  // Code
}
```

Selbstausführende Funktionen (4)

Die letzten drei Zeilen liefern Exceptions, da nichts außerhalb der anonymen Funktion zugänglich ist.

```
(function(){
  var foo = 'Hello';
  var bar = 'World!'

  function baz(){
      return foo + ' ' + bar;
 }
})();

console.log(foo);
console.log(bar);
console.log(baz());
```

Beispiel und weiterführende Erläuterungen "http://markdalgleish.com/2011/03/self-executing-anonymous-functions/"

Referenzen