

DOM jQuery Repetition

Framework vs Library

Frameworks vs. Libraries

■ Framework

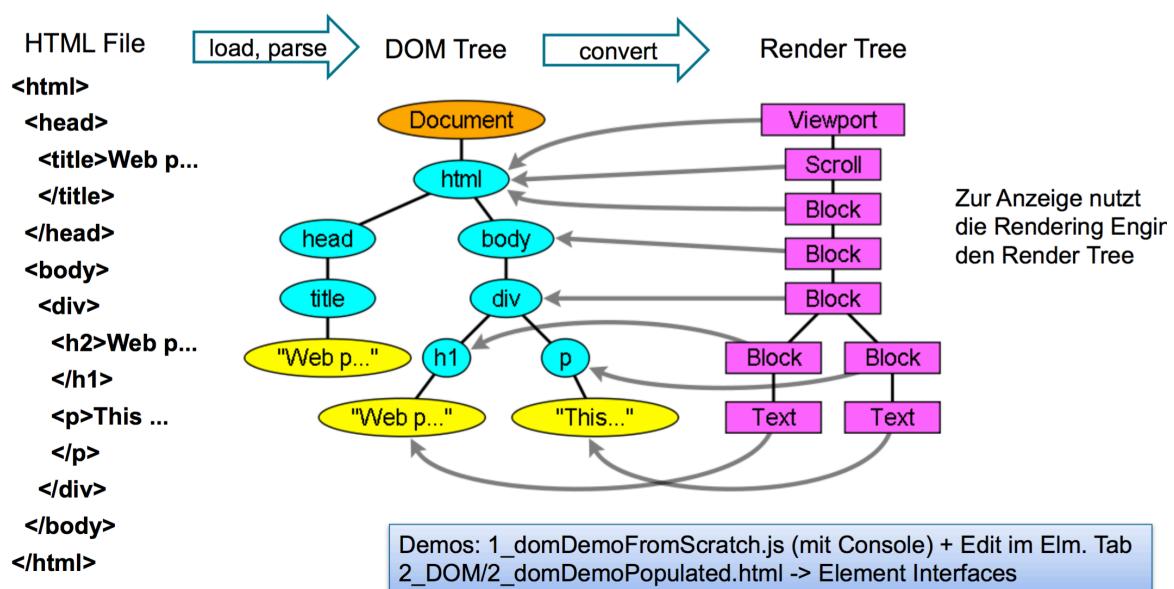
- Entwickelter Code registriert sich beim Framework
- Entwickelter Code wird vom Framework aufgerufen
- Kontrolle beim Frameworks
- Beispiele:
 - Browser-Events
 - Node: Konsole-API, Networking, Files,

■ Library

- Entwickelter Code ruft Library-Funktionen auf
- Entwickelter Code hat Kontrolle
- Beispiel:
 - JS: Math
 - Underscore

HTML to DOM parsing: HTML wird geparsst. Resultat ist das "DOM"

Abend
DD

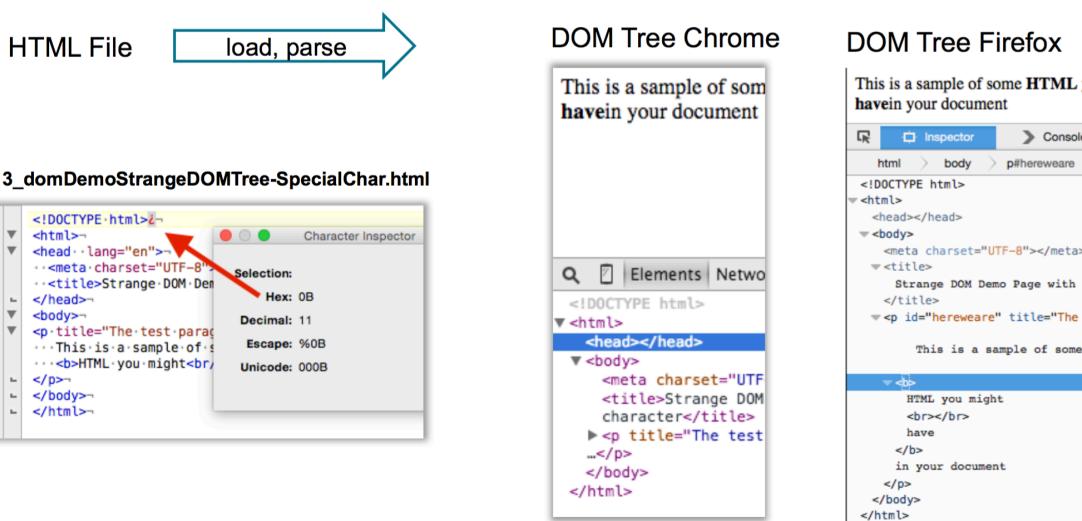


- Der Text im html File wird in das DOM übersetzt ("Parsing").
- Dies geht Schritt für Schritt (auch Scripts)
- ACHTUNG: DOM erst gesamthaft verfügbar bei/nach dem `window.onload` Event bzw. `document.DOMContentLoaded` (später)

<nt
<he
</h
<bo
<di

HTML to DOM parsing: HTML-Parsing kann durch "special characters" fehlerhaft sein

(Chrome und Firefox)



- Mögliche Layout Probleme:
Umbruch-Fehler durch zusätzlichen Whitespace.
Z.B. 3 div mit `box-sizing: border-box;` und `width: calc(100% / 3);`
-> 2_DOM/4_whitespaceDemo.html
- Abhilfe:
 - Whitespace mittels JavaScript aus dem DOM-Baum entfernen
 - Während der Minification kann Whitespace eliminiert werden
 - Die Returns mit Kommentaren umschließen

```
<div class="box">
  <div>width: calc( 100% / 3 );
    |   box-sizing: border-box;</div>
</div><!--
--><div class="box">
  <div>width: calc( 100% / 3 );
    |   box-sizing: border-box;</div>
```

DOM Standard API

- Globale Browser Objekte
- DOM Schnittstellen / Interfaces
 - Document
 - Node
 - Element
 - NodeList
- DOM Manipulation
- DOM Events, Event-Registrierung, Callback-Funktionen

- **document implementiert auch das Node Interface**

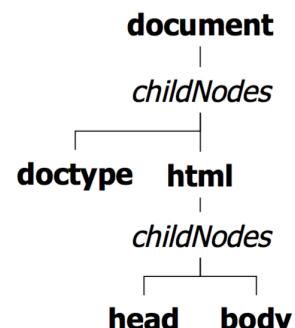
- **Node Interface**

<https://developer.mozilla.org/en-US/docs/Web/API/Node>

- Property: nodeType (z.B. ELEMENT_NODE, TEXT_NODE)
- Property: nodeValue (nur bei Text_Nodes gefüllt)
- Property: childNodes (gibt NodeList zurück)
- Property: firstChild
- Property: firstElementChild kein Text etc.
- Property: nextSibling / nextElementSibling
- Method: appendChild()
- Method: removeChild()
- Method: addEventListener()

- **Beispiel: Das body-Element ist erreichbar über**

- `document.childNodes[1].childNodes[1]`
(Annahme kein **White-Space** zwischen head & body)
- `document.body`
- `document.firstChild.firstElementChild.nextSibling`



DOM Manipulation (e: Element)

- e.insertAdjacentHTML(posStr, htmlStr)
 //posStr: 'beforebegin', 'afterbegin', 'beforeend', 'afterend'
- e.insertAdjacentText(posStr, textStr)
- e.insertAdjacentElement(posStr, element)

- e.removeChild(<childNode>)

- e.appendChild(<newDocNode>)
 e.insertAfter(<childNode>, <newDocNode>)

- e.textContent = "...."; e.innerHTML = "...." ; e.innerHTML += "...."

- **Effizientes append von mehreren Elementen:**
 "document fragment" nutzen
 const df = document.createDocumentFragment();
 songs.forEach(function (song) {
 const liElement = document.createElement('li');
 [....]
 df.appendChild(liElement);
 });
 document.getElementById('songs').appendChild(df);

Mini-Quiz: DOM API

LogOutput?

```
<head lang="en">
  <meta charset="UTF-8">
  <title>DOM API Quiz</title>
  <script>
    let outTxt = document.getElementById('p1').nextSibling.textContent;
    console.log(outTxt); // -> _____
    //outText = document.getElementById('p1')._____. // -> Text2.2

    //outTxt = document.getElementById('p1').nextElementSibling.firstElementChild.textContent;
    //console.log(outTxt); // -> _____
  </script>
</head>
<body>
  <p id="p1">Text1.1<em>Text1.2</em>Text1.3</p>
  <p>Text2.1<em>Text2.2</em>Text2.3</p>
</body>
</html>
```

DOM Loaded Events: window.onload oder document Listener für DOMContentLoaded

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>domLoadedDemo</title>
    <script>
        console.log('headScript executing');
        document.onload = _ => console.log('onload executed'); // does not fire
        document.onDOMContentLoaded = _ => console.log('onDOMContentLoaded executed'); // does not exist / fire
        document.addEventListener('DOMContentLoaded',
            _ => console.log('document DOMContentLoaded executed'));
        window.onload = _ => console.log('window.onload executed');
        console.log('headScript ended');
    </script>
</head>
<body>
<h1>Dom Loaded Demo</h1>
<script>
    console.log('endScript executing');
</script>
</body>
</html>
```

headScript executing
headScript ended
endScript executing
document DOMContentLoaded executed
window.onload executed

Event Handler, Nutzung von Closures (1)

DOM
41

- Folgendes Beispiel funktioniert nicht (Webseite mit 10 p-Elementen)

2_DOM/8_domEventsClosure_V1.htm

```
pElementsNodeList = document.querySelectorAll('p');
let i=0;
for (pElement of [...pElementsNodeList]) {
    pElement.onclick = function () {console.log('I am P-Element #' + i)};
    i++;
}
```

- Lösung: Closure 1

2_DOM/8_domEventsClosure_V2.htm

```
function createIndexedEventListener(index) {
    return function () {console.log('I am P-Element #' + index)}
}
window.onload = function () {
    pElementsNodeList = document.querySelectorAll('p');
    let i=0;
    for (pElement of [...pElementsNodeList]) {
        pElement.onclick = createIndexedEventListener(i);
        i++;
    }
}
```

Event Handler, Nutzung von Closures (2)

■ Problem (rep.)

```
let i=0;
for (pElement of [... document.querySelectorAll('p')]) {
    pElement.onclick = function () {console.log('I am P-Element #' + i)};
    i++;
}
```

■ Alternative Lösung 2: Closure 2

```
window.onload = function () {
    pElementsArray = [...document.querySelectorAll('p')];
    pElementsArray.forEach((e, i) =>
        e.onclick = () => console.log('I am P-Element #' + i));
};

8_domEventsClosure_V3
```

■ Alternative Lösung 3: Closure 3 (jedes let/const generiert eine Closure)

```
window.onload = function () {
    pElementsNodeList = document.querySelectorAll('p');
    for (let i = 0; i < pElementsNodeList.length; i++) {
        pElementsNodeList[i].onclick =
            () => console.log('I am P-Element #' + i);
    }
};

8_domEventsClosure_V4
```

Event Handler, Nutzung von data-attributes

43

■ Folgendes Beispiel funktioniert nicht (Webseite mit 10 p-Elementen)

8_domEventsClosure_V1

```
pElementsNodeList = document.querySelectorAll('p');
let i=0;
for (pElement of [...pElementsNodeList]) {
    pElement.onclick = function () {console.log('I am P-Element #' + i)};
    i++;
}
```

■ Lösung: Nutzung von data-attributes (diese könnten auch im html z.B. direkt mit data-index spezifiziert werden)

8_domEventsClosure_V5

```
window.onload = function () {
    pElementsNodeList = document.querySelectorAll('p');
    let i=0;
    for (pElement of [...pElementsNodeList]) {
        pElement.dataset.index = i;
        pElement.onclick = function (e) {console.log('I am P-Element #' + e.target.dataset.index)};
        i++;
    }
};
```

"this" in Event-Listenern

Abs

- "this" in Event-Listenern
 - = DOM-Element welches den Event ausgelöst hat (d.h. das DOM-Element bei dem der EventListener registriert wurde)
 - Event-Listener erhalten beim Aufruf als erstes Argument den aktuellen Event (= Objekt mit Informationen zum aktuellen Event)
 - **e.target** = DOM-Element mit dem Nutzer interagiert hat => Bei "bubbled" Events ist event.eventTarget != this
 - **e.currentTarget** ist immer = this = DOM-Element dem der EventListener registriert wurde
 - **Empfehlung:** Lieber mit e.target oder e.currentTarget arbeiten als mit "this"

```
<!DOCTYPE html>
<html>
<head lang="en">
<meta charset="UTF-8">
<title>DOM Events Demo</title>
<script>
    function buttonClickListener (e) {
        console.log('e.target.id=' + e.target.id);
        console.log('this.id=' + this.id);
        console.log('e.currentTarget.id=' + e.currentTarget.id)
    }
    window.onload = function () {
        par1.addEventListener("click", buttonClickListener);
        button1.addEventListener("click", buttonClickListener)
    }
</script>
</head>
<body>
    <p id="par1">
        <button id="button1">button1</button>
    </p>
</body>
</html>
```

Event Handling ACHTUNG

- **Fakten**
 - JavaScript ist "single threaded"
 - Timer-Events werden erst behandelt wenn alle User-Interactions (clicks, keys) abgearbeitet worden sind.
 - Event-Handler können den DOM-Baum inklusive registrierte Event-Handler verändern. Der veränderte Status des DOM Baums wird schon während der Behandlung von Events wie click berücksichtigt
- **Empfehlungen**
 - Event-Handler sollten nach spätestens ca. 100ms (1/10 sec) terminieren, sonst wird das UI merklich "unresponsive"

JQuery Einbinden & Starten

■ Einbinden der JQuery Lib:

(Local): <script src="js/jquery-3.2.1.min.js"></script>
(CDN): <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>

■ Startpunkt (meist) statt window.onload

in JQuery:

```
$('document').ready(function () {  
    ...  
});  
  
oder (äquivalent)  
jQuery(function ($) {  
    ...  
});  
  
oder (noch sicherer)  
(function($, window, document) {  
    $(function() {  
        ...  
    });  
}(window.jQuery, window, document));
```

Startpunkt

■ Einsatz von jQuery macht in den meisten Fällen erst Sinn nachdem das DOM bereit ist...

■ ... deshalb auf das DOM-Ready Event warten

■ ready wird ausgeführt falls DOMContentLoaded ausgelöst wurde.

■ Falls DOMContentLoaded schon gefeuert wurde – wird diese Funktion sofort aufgerufen!
■ Vorteil da normalerweise diese Funktion ignoriert werden würde.

```
$('document').on('ready', (function () {  
  
    // code ...  
    $('#container').text("Hello world");  
});  
  
// oder kürzer  
  
$(function () {  
  
    // code ...  
    $('#container').text("Hello world");  
});
```

addEventListener(DOMContentLoaded) vs. ready(...)

```
document.addEventListener("DOMContentLoaded", function() {
    console.log("DOMContentLoaded");
});
$(document).on('ready', function () {
    console.log("ready");
});

setTimeout(function() {
    document.addEventListener("DOMContentLoaded", function() {
        console.log("setTimeout - DOMContentLoaded");
    });
    $(document).ready(function () {
        console.log("setTimeout - ready");
    });
}, 10);
```

Output:

```
ready
DOMContentLoaded
setTimeout - ready
```

> |

\$.ready vs. DOMContentLoaded vs. onload vs. load

■ Load Events werden erst ausgelöst, wenn der gesamte Content geladen ist

- Bilder
- Flash
- Vorteil: Es ist ALLES geladen
- Nachteil: Es muss ALLES geladen sein.

■ Ready Events

- Keine Garantie für das Bilder schon vorhanden sind.
- Frühester Zeitpunkt welcher DOM Manipulationen möglich sind.
- In den meisten Fällen die bessere Wahl.

\$.ready vs. DOMContentLoaded vs. onload vs. load

```
window.onload = function () {
    console.log("load", "window.onload");
};

$(window).on('load', (function(){
    console.log("load", "$.load");
}));

window.addEventListener('load', function () {
    console.log("load", "window.addEventListener");
    console.log("img height", $("img").first().height());
}, false);

$(function(){
    console.log("ready","$.ready");
    console.log("img height", $("img").first().height());
});

document.addEventListener('DOMContentLoaded', function () {
    console.log("ready", "document.addEventListener");
}, false);
```

ready	\$.ready
ready	img height 0
ready	document.addEventListener
load	window.onload
load	\$.load
load	window.addEventListener
load	img height 100

noConflict

- Problem: Das \$ Zeichen wird auch von anderen Frameworks genutzt.

■ Details: <https://learn.jquery.com/using-jquery-core/avoid-conflicts-other-libraries>

- jQuery kann einen «noConflict» Modus aktivieren.

- \$ wird wieder auf die alte Referenz gelegt.
- jQuery ist nur noch über jQuery zugreifbar.
- Es kann ein anderer Alias vergeben werden.

```
var $j = jQuery.noConflict();
```

- Lösung mit IIFE

```
(function($) {
    // $ ist jQuery
}) (jQuery);
```

- Lösung über JQuery Parameter Injection

```
jQuery(function($) {
    // $ ist jQuery
    console.log($(".em"));
});
```

JQuery Pitfalls & Antipatterns

Pitfalls

- <http://www.codeproject.com/Articles/346904/Common-Pitfalls-of-jQuery>

■ Wichtige Punkte / Achtung bei

- Unnötigen jQuery Abfragen statt Zwischenspeichern von Resultaten
- Ineffiziente Selektoren, z.B. #eluids nicht genutzt um Suche einschränken
- Shortcuts (wie hide, show, toggle, empty) nicht genutzt -> ineffizient & unleserlich
- Repetitive Selektoren

Antipatterns

- <http://blog.javascripting.com/2015/01/12/real-world-javascript-anti-patterns-part-two/>
- SCHLECHT: .addClass und .removeClass statt .toggleClass
- SCHLECHT: registrieren von generischen Handlern, dann Switch/Case zu Runtime
- GUT: Nutzung von Event-Namespace zu Gruppen-de-Registrierung von Handlern \$(window).off('.mycomponent')

Client Templating mit Handlebars – Details (1)

■ Templates können pre-compiled werden

(siehe <http://handlebarsjs.com/precompilation.html>)

oder das Template HTML wird in einem Script-Tag definiert z.B.

```
<script id="entry-template" type="text/x-handlebars-template">
  <div class="entry">
    <h1>{{name}}</h1>
    <div class="body">
      {{adress}}
    </div>
  </div>
</script>
```

■ Valide Handlebars Expressions

<h1>{{name}}</h1> (lookup im Context Objekt)

<h1>{{adress.zip}}</h1> (paths sind erlaubt: adress ist eine Eigenschaft des Context Objekts; mit ../ Navigation zum Parent)

■ Standard Block Helpers -> http://handlebarsjs.com/block_helpers.html

- {{#each this}}
 Id: {{userId}} Name: {{name}}
{{/each}}
- {{#if isActive}}
 ![Active](star.gif)
{{/if}}