WBE: BROWSER-TECHNOLOGIEN

JAVASCRIPT IM BROWSER

ÜBERSICHT

- JavaScript im Browser
- Document Object Model
- Event Handling im Browser
- Kleiner Exkurs: jQuery
- SVG, Canvas und Web-Storage

ÜBERSICHT

- JavaScript im Browser
- Document Object Model
- Event Handling im Browser
- Kleiner Exkurs: jQuery
- SVG, Canvas und Web-Storage

JAVASCRIPT IM BROWSER

- Ohne Browser gäbe es kein JavaScript
- Für den Einsatz im Browser entwickelt
- Brendan Eich, 1995: Netscape Navigator

Speaker notes

Zur Erinnerung:

Mitte der 1990er Jahre war der Netscape Navigator der vorherrschende Browser, bevor ihm der Internet Explorer den Rang abgelaufen hat.

Der name der neuen Script-Sprache war zunächst *LiveScript*, wurde dann aber schnell in *JavaScript* umbenannt, um von dem zunehmenden Hype um die neue Programmiersprache *Java* zu profitieren. Leider hat diese Namens-Ähnlichkeit immer wieder zu Verwechslungen geführt.

HTML UND JAVASCRIPT

- Element `script` (End-Tag notwendig)
- Vom Browser beim Lesen des HTML-Codes ausgeführt
- Oder Code als Reaktion auf Ereignis ausführen

```
<!-- Code ausführen -->
<script>alert("hello!")</script>

<!-- Code aus JavaScript-Datei ausführen -->
<script src="code/hello.js"></script>

<!-- Code als Reaktion auf Ereignis ausführen -->
<button onclick="alert('Boom!')">DO NOT PRESS</button>
```

HTML UND JAVASCRIPT

- Laden von ES-Modulen möglich
- Angabe von `type="module"`

```
<script type="module" src="code/date.js"></script>
```

https://eloquentjavascript.net/10_modules.html#h_hF2FmOVxw7

SANDBOX

- Ausführen von Code aus dem Internet ist potentiell gefährlich
- Möglichkeiten im Browser stark eingeschränkt
- Zum Beispiel kein Zugriff auf Filesystem, Zwischenablage etc.
- Trotzdem häufig Quelle von Sicherheitslücken
- Abwägen: Nützlichkeit vs. Sicherheit

Sicherheitslücken werden meist schnell geschlossen.
Immer die neuesten Browser-Versionen verwenden.

ÜBERSICHT

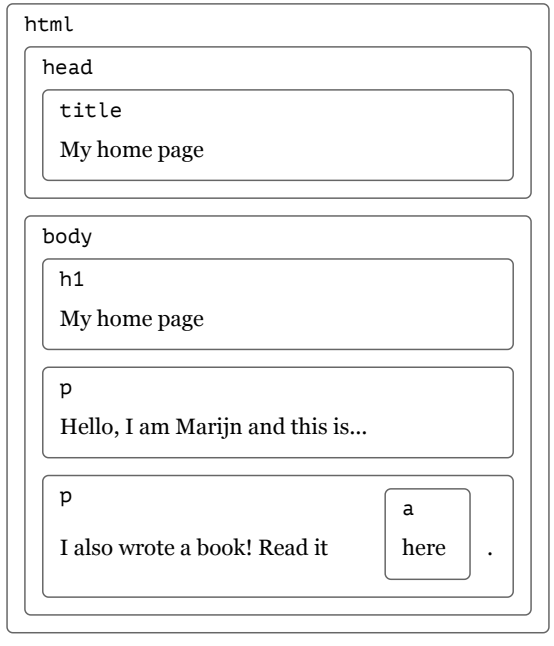
- JavaScript im Browser
- Document Object Model
- Event Handling im Browser
- Kleiner Exkurs: jQuery
- SVG, Canvas und Web-Storage

WEBSITE IM BROWSER-SPEICHER

- Browser parst HTML-Code
- Baut ein Modell der Dokumentstruktur auf
- Basierend auf dem Modell wird die Seite angezeigt
- Auf diese Datenstruktur haben Scripts Zugriff
- Anpassungen daran wirken sich live auf die Anzeige aus

BEISPIEL

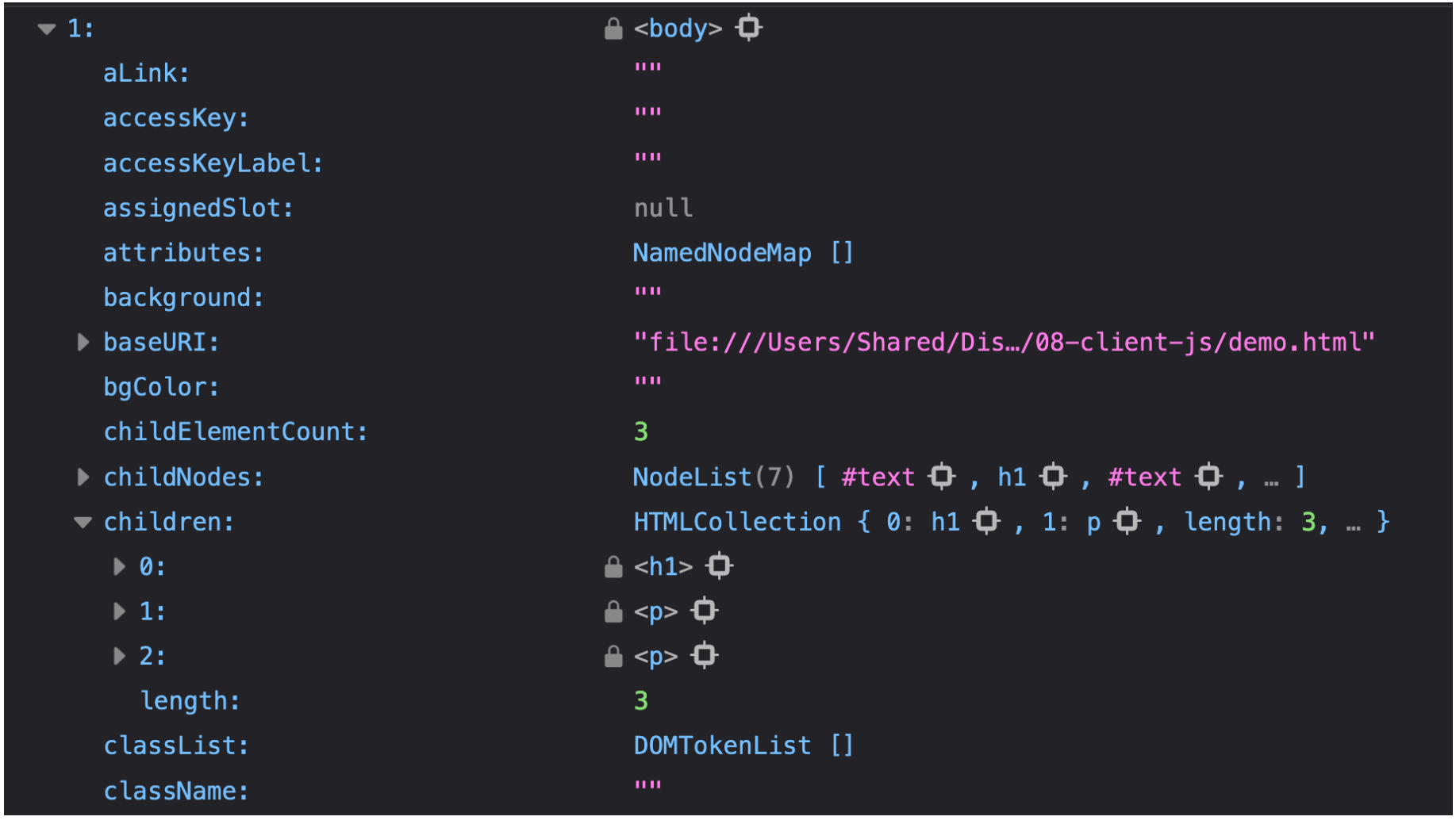
```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home
      page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```



DOCUMENT OBJECT MODEL (DOM)

- Jeder Knoten im Baum durch ein Objekt repräsentiert
- Bezeichnung: Document Object Model (DOM)
- Zugriff über das globale Objekt `document`
- Attribut `documentElement` ist Referenz auf HTML-Knoten
- Zahlreiche Attribute und Methoden

ELEMENTKNOTEN BODY

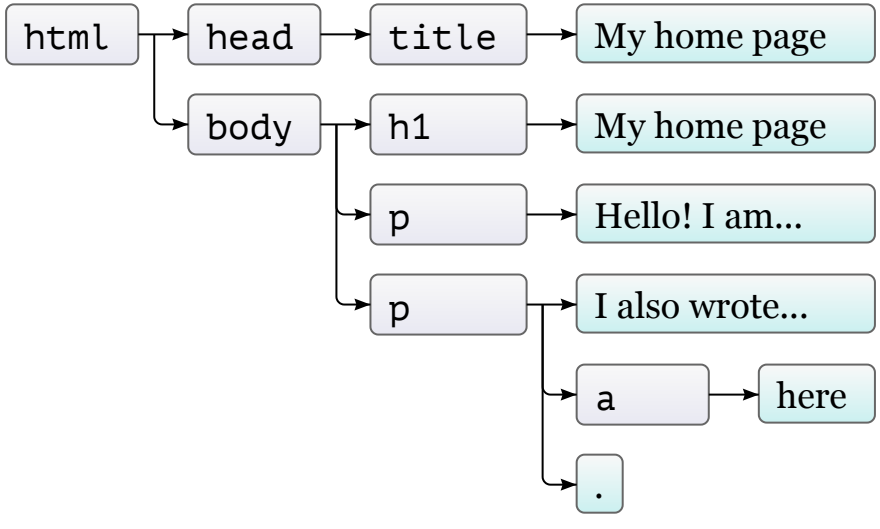


BAUMSTRUKTUR

- Jeder Knoten hat ein `nodeType`-Attribut
- HTML-Elemente haben den `nodeType` 1

NodeType	Konstante	Bedeutung
1	Node.ELEMENT_NODE	Elementknoten
3	Node.TEXT_NODE	Textknoten
8	Node.COMMENT_NODE	Kommentarknoten

BAUMSTRUKTUR

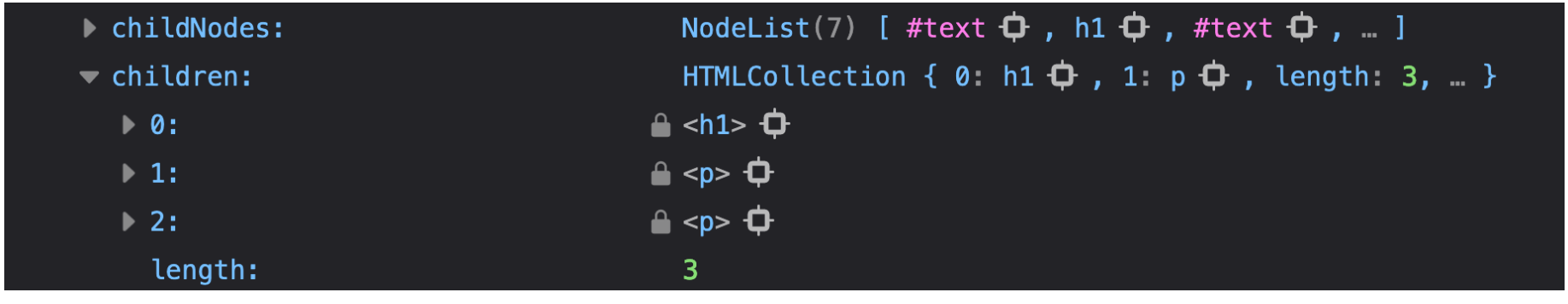


DOM ALS STANDARD

- Im Laufe der Jahre gewachsen
- Sprachunabhängig konzipiert
- Zahlreiche Redundanzen
- Kein klares und verständliches Design
- Ehrlich gesagt: ziemlich unübersichtlich

ATTRIBUT CHILDNODES

- Instanz von `NodeList`
- Array-ähnliches Objekt (aber kein Array)
- Numerischer Index und `length`-Attribut
- Alternative: `children`-Attribut
  - Instanz von `HTMLCollection`
  - enthält nur die untergeordneten Elementknoten



Speaker notes

Da es kein Array ist, funktionieren Array-Methoden wie `slice` und `map` nicht, wohl aber `cnodes[0]` oder `cnodes.length`. Man kann aber mit `Array.from` ein Array erstellen.

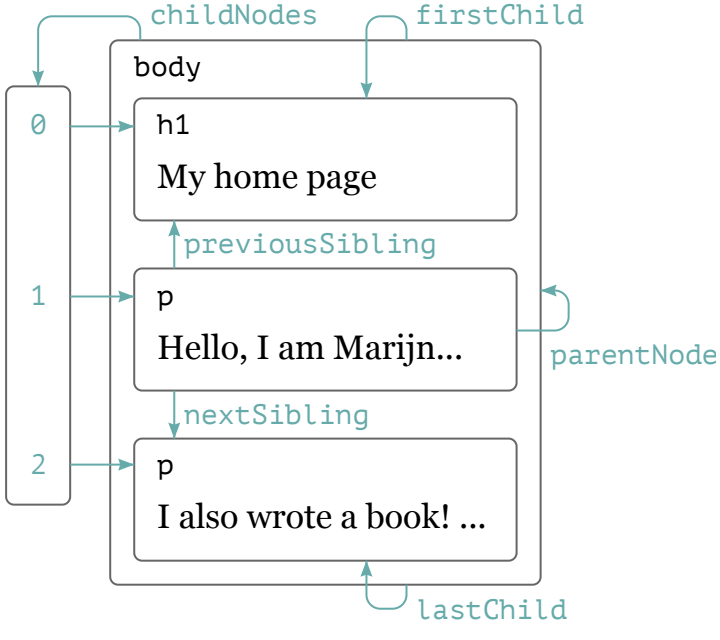
ELEMENT HINZUFÜGEN

- Element erzeugen: `document.createElement`
- Attribute erzeugen: `document.createAttribute`
- Und hinzufügen: `<element>.setAttribute`
- Element in Baum einfügen: `<element>.appendChild`

„Code that interacts heavily with the DOM tends to get long, repetitive, and ugly.” (Eloquent JavaScript)

Gut dagegen: JavaScript erlaubt es, problemlos eigene Abstraktionen zu definieren

BAUMSTRUKTUR ABARBEITEN



- Diverse Attribute und Methoden zur Navigation im DOM-Baum
- Häufig: Array-ähnliche Objekte

BEISPIEL

```
// scans a document for text nodes containing a given string and
// returns true when it has found one
function talksAbout (node, string) {
  if (node.nodeType == Node.ELEMENT_NODE) {
    for (let i = 0; i < node.childNodes.length; i++) {
      if (talksAbout(node.childNodes[i], string)) {
        return true
      }
    }
    return false
  } else if (node.nodeType == Node.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1
  }
}

console.log(talksAbout(document.body, "book"))
// → true
```

Speaker notes

Da

```
childNodes
```

kein echtes Array liefert, ist eine Iteration mit for/of hier nicht möglich. Alternativ hätte mit

```
Array.from
```

zuerst ein Array erstellt werden können.

ELEMENTE AUFFINDEN

```
let aboutus = document.getElementById("aboutus")
let aboutlinks = aboutus.getElementsByTagName("a")
let aboutimportant = aboutus.getElementsByClassName("important")

let navlinks = document.querySelectorAll("nav a")
```

- Gezielte Suche im ganze Dokument oder Teilbaum
- Zum Beispiel alle Elemente mit bestimmtem Tagnamen
- Oder nach bestimmtem Wert des `id`- oder `class`-Attributs
- Alternativ mit Hilfe eines CSS-Selektors

Speaker notes

Auch `querySelectorAll` liefert wie viele andere DOM-Methoden ein Array-ähnliches Objekt als Ergebnis (konkret: es ist eine Instanz von `NodeList`).

Im Gegensatz zu `querySelectorAll` liefert `querySelector` nur das erste passende Element oder null, wenn es kein solches gibt.

https://eloquentjavascript.net/14_dom.html#h_5ooQzToxht

DOKUMENT ANPASSEN

- Diverse Methoden zum Knoten entfernen, einfügen, löschen oder verschieben
- Zum Beispiel: `appendChild`, `remove`, `insertBefore`

```
<p>One</p>
<p>Two</p>
<p>Three</p>

<script>
  let paragraphs = document.body.getElementsByTagName("p")
  document.body.insertBefore(paragraphs[2], paragraphs[0])
</script>
```

Speaker notes

Das Beispiel wählt alle Absätze aus und verschiebt den letzten Absatz vor den ersten. Durch das Einfügen des Absatzes wird er automatisch an der ursprünglichen Stelle entfernt, da ein Knoten nur an einer Stelle im Dokument vorkommen kann.

Wenn statt `insertBefore` die Methode `replaceChild` verwendet wird, wird der erste Absatz durch den dritten ersetzt und es resultiert:

Three
Two

TEXTKNOTEN ERZEUGEN

```
<p>The <img src="img/cat.png" alt="Cat"> in the
  <img src="img/hat.png" alt="Hat">.</p>

<p><button onclick="replaceImages()">Replace</button></p>

<script>
  function replaceImages () {
    let images = document.body.getElementsByTagName("img")
    for (let i = images.length - 1; i >= 0; i--) {
      let image = images[i]
      if (image.alt) {
        let text = document.createTextNode(image.alt)
        image.parentNode.replaceChild(text, image)
      }
    }
  }
</script>
```

Speaker notes

Das Beispiel ersetzt alle Bilder durch ihren im `alt`-Attribut abgelegten Alternativtext. Der Textknoten wird mit `document.createTextNode` erzeugt und das Bild mit `replaceChild` ersetzt.

https://eloquentjavascript.net/14_dom.html#h_AlX6HES+2D

Die Liste der Bilder wird hier von hinten her bearbeitet, da die `images`-Referenz auf eine *live* aktualisierte Datenstruktur verweist. Das bedeutet, dass die Anzahl der Bilder (images.length) bereits um eins reduziert ist, nachdem das erste Bild ersetzt wurde.

Das Ergebnis von `querySelectorAll` ist im Gegensatz zu `getElementsByTagName` übrigens keine "Live"-Datenstruktur.

ARRAYS

- Datenstrukturen im DOM sind häufig Array-ähnlich
- Sie haben Zahlen sowie `length` als Attribute
- Mit `Array.from` können sie in echte Arrays konvertiert werden

```
let arrayish = {0: "one", 1: "two", length: 2}
let array = Array.from(arrayish)
console.log(array.map(s => s.toUpperCase()))
// → ["ONE", "TWO"]
```

ELEMENTKNOTEN ERZEUGEN

```
function elt (type, ...children) {
  let node = document.createElement(type)
  for (let child of children) {
    if (typeof child != "string") node.appendChild(child)
    else node.appendChild(document.createTextNode(child))
  }
  return node
}
```

- Element mit Typ (1. Argument) erzeugen
- Kindelemente (weitere Argumente) hinzufügen

Speaker notes

Die Kindelemente werden als Textknoten angelegt, wenn es Strings sind. Andererseits wird angenommen, dass es selbst bereits Elemente sind, welche hinzugefügt werden können.

Beispiel:

```
<blockquote id="quote">
  No book can ever be finished. While working on it we learn
  just enough to find it immature the moment we turn away
  from it.
</blockquote>

<script>
// definition of elt ...

// demo of the elt function:
document.getElementById("quote").appendChild(
  elt("footer", "—",
      elt("strong", "Karl Popper"),
      ", preface to the second edition of ",
      elt("em", "The Open Society and Its Enemies"),
      ", 1950"))
</script>
```

ATTRIBUTE

- Viele HTML-Attribute entsprechen Attributen im DOM
- Beispiel: `href`-Attribut des `a`-Elements

```
<a href="http://eloquentjavascript.net">here</a>
```

DOM:

```
a-element
  accessKey: ""
  accessKeyLabel: ""
  attributes: NamedNodeMap [ href="http://eloquentjavascript.net" ]
  childNodes: NodeList [ #text ]
  children: HTMLCollection { length: 0 }
  classList: DOMTokenList []
  className: ""
  ...
  href: "http://eloquentjavascript.net/"
  ...
```

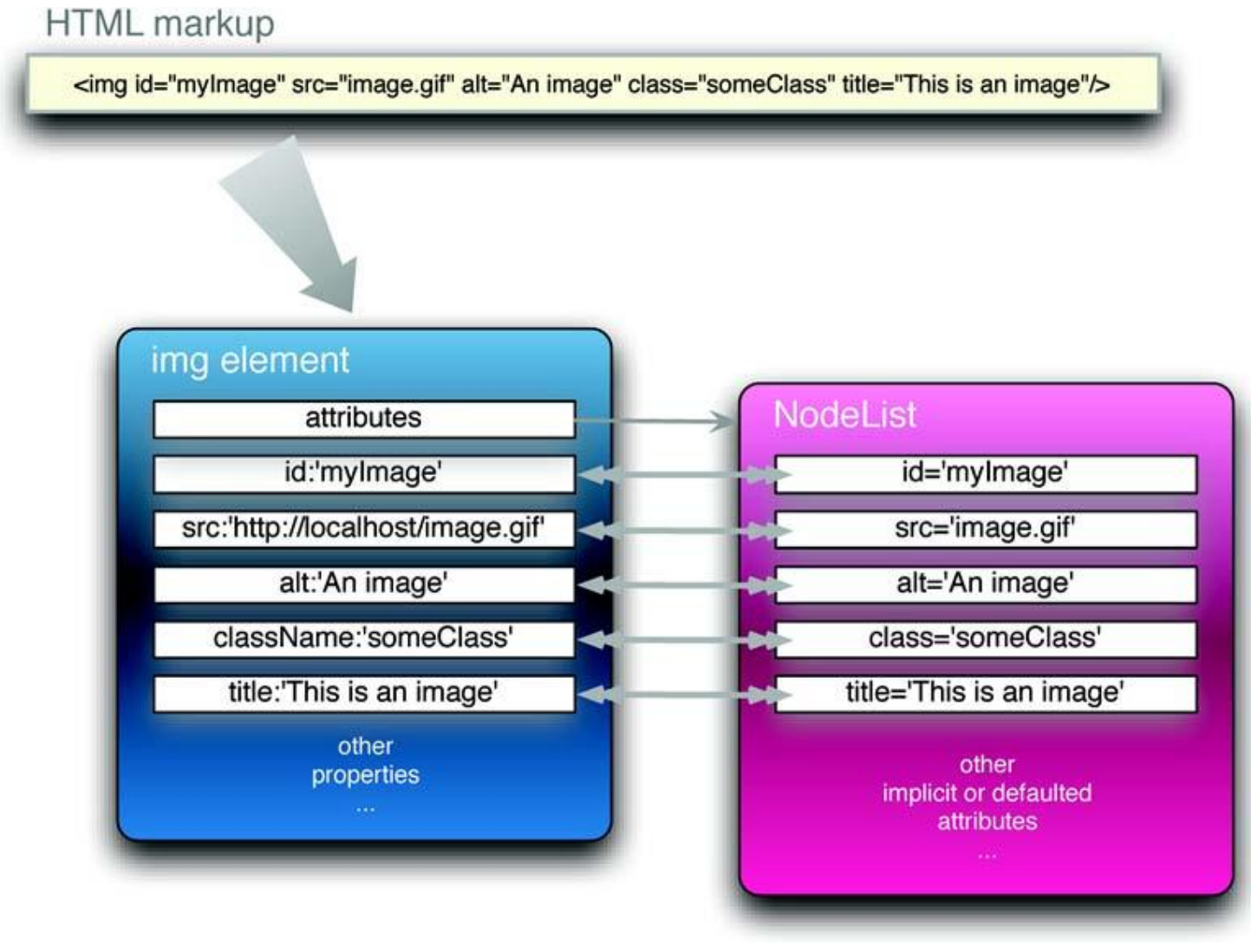
Speaker notes

In englischen Texten wird meist unterschieden zwischen `attribute`, wenn zum Beispiel HTML-Attribute gemeint sind, und `property`, wenn die "Properties" von Dom-Knoten (Objekten) gemeint sind.

In der deutschen Sprache biete ist diese Unterscheidung nicht so gut möglich, da wir nicht so gern von *Objekt-Eigenschaften* sprechen. Also verwenden wir auch in diesem Fall die Bezeichnung *Attribute* und ergänzen diese in Fällen, in denen sonst Verwechslungen möglich sind. Also:

- HTML-Attribut (engl. attribute)
- Element-Attribut (engl. attribute)
- DOM-Attribut (engl. property)
- Objekt-Attribut (engl. property)

ATTRIBUTE


HTML markup

img element
attributes
id:'myImage'
src:'http://localhost/image.gif'
alt:'An image'
className:'someClass'
title:'This is an image'
other
properties
...
NodeList
id='myImage'
src='image.gif'
alt='An image'
class='someClass'
title='This is an image'
other
implicit or defaulted
attributes
...

EIGENE ATTRIBUTE

- Beginnen mit "data-"
- DOM-Attribut `dataset` liefert DOMStringMap mit allen `data`-Attributen

```
<p data-classified="secret">The launch code is 00000000.</p>
<p data-classified="unclassified">I have two feet.</p>

<script>
  let paras = document.body.getElementsByTagName("p")
  for (let para of Array.from(paras)) {
    if (para.dataset.classified == "secret") {
      para.remove()
    }
  }
</script>
```

Speaker notes

Eigene HTML-Attribute werden also nicht direkt in die DOM-Knoten übernommen. Wenn sie mit `data-` beginnen, kann man via `dataset` auf sie zugreifen. Das ist die empfohlene Variante, anwendungsspezifische Attribute einzuführen.

Alternativ kann mit `getAttribute` und `setAttribute` auf HTML-Attribute zugegriffen werden. Im Beispiel daher auch möglich:

```
if (para.getAttribute("data-classified") == "secret") {...}
```

ATTRIBUT CLASS

- Mehrere Klassen durch Leerzeichen getrennt möglich
- Im DOM zugreifbar über `className` oder `classList`
- Achtung: `className` statt `class` (reservierter Name)

```
<p class="hint info">I also wrote a book!</p>
```

DOM:

```
...
classList     DOMTokenList [ "hint", "info" ]
className     "hint info"
...
```

Speaker notes

Statt `class` wird im DOM `className` verwendet, weil `class` ein reservierter Name in JavaScript ist. Mit `getAttribute` und `setAttribute` kann aber auch `class` verwendet werden.

Das Bearbeiten von Klassen über `className` ist umständlich, da mit String-Verarbeitung verbunden. Aus diesem Grund wurde später `classList` hinzugefügt, das eine komfortablere Bearbeitung erlaubt.

Tipp: Damit weiss man immer noch nicht, wie die Bearbeitung der `classList` funktioniert. Als Typ wird `DOMTokenList` angegeben. Relativ schnell findet man mehr Informationen dazu in einer Referenz wie https://devdocs.io.

Auf der Seite https://devdocs.io/dom/domtokenlist erfährt man dann, dass Methoden wie `add`, `remove` und einige andere zur Verfügung stehen.

LAYOUT

- Browser positioniert Elemente im Viewport
- Grösse und Position ebenfalls in DOM-Struktur eingetragen
- `clientWidth`: Breite von Blockelementen inkl. Padding
- `offsetWidth`: Breite inkl. Border
- Einheit: Pixel (`px`)
- Beispiel:

```
clientHeight   19
clientLeft     0
clientTop      0
clientWidth    338
```

```
offsetHeight   19
offsetLeft     8
offsetParent   <body>
offsetTop      116
offsetWidth    338
```

Speaker notes

Früher entsprach ein Pixel einem Leuchtpunkt des Displays. Bei heutigen hochauflösenden Displays ist das nicht mehr der Fall, das heisst ein Pixel entspricht meist mehreren Leuchtpunkten des Displays.

Am besten findet man die Position eines Elements im Viewport mit der Methode `getBoundingClientRect` heraus, die ein Objekt mit Attributen `top`, `bottom`, `left` und `right` zurückgibt. Um die Position relativ zum gesamten Dokument zu bestimmen, muss man noch die Scroll-Positionen `pageXOffset` und `pageYOffset` addieren.

PERFORMANZ

- Layout einer Seite aufbauen ist zeitaufwendig
- Konsequenz: Seitenänderungen via DOM möglichst zusammenfassen
- Beispiel: Warum ist folgende Sequenz ungünstig?

```
let target = document.getElementById("one")
while (target.offsetWidth < 2000) {
  target.appendChild(document.createTextNode("X"))
}
```

Speaker notes

Es sollen offenbar 'X'-Zeichen geschrieben werden, bis eine Breite von 2000px erreicht ist. Ganz unabhängig davon, ob das ein sinnvolles Beispiel ist: Nach jedem Schreiben von 'X' wird das Layout neu berechnet und die neue Breite des Absatzes abgefragt.

Besser: Breite eines 'X' bestimmen, einen String mit der passenden Anzahl von Zeichen erstellen und diesen aufs Mal ins DOM übernehmen:

```
let target = document.getElementById("two")
target.appendChild(document.createTextNode("XXXXX"))
let total = Math.ceil(2000 / (target.offsetWidth / 5))
target.firstChild.nodeValue = "X".repeat(total)
```

Diese Variante hat im Test 1ms benötigt gegenüber 19ms beim Schreiben der Zeichen in der while-Schleife.

Demo:
https://eloquentjavascript.net/14_dom.html#p_TzaMQEZthV

DARSTELLUNG

- Attribut `style` (HTML und DOM)
- Wert ist ein String (HTML) bzw. ein Objekt (DOM)
- HTML: CSS-Eigenschaften mit Bindestrich: `font-family`
- DOM: CSS-Eigenschaften in „Camel Case”: `fontFamily`

```
<p id="para" style="color: purple">Nice text</p>

<script>
  let para = document.getElementById("para")
  console.log(para.style.color)
  para.style.color = "magenta"
</script>
```

ANIMATION

- Anpassen zum Beispiel der `position`-Eigenschaft
- Synchronisieren mit der Browser-Anzeige: `requestAnimationFrame`

```
function animate(time, lastTime) {
  // calculate new position
  // ...
  requestAnimationFrame(newTime => animate(newTime, time));
}
requestAnimationFrame(animate);
```

Speaker notes

Das Callback von `requestAnimationFrame` wird jeweils aufgerufen bevor der Browser die Anzeige aktualisiert. Es erhält die aktuelle Zeit als Argument, so dass die Positionsänderung von der verstrichenen Zeit abhängig gemacht werden kann, was zu einer gleichmässigen Animation führt.

Mit `cancelAnimationFrame` kann die für das nächste Update eingeplante Funktion wieder entfernt werden (falls nicht bereits ausgeführt).

Beispiel:

https://eloquentjavascript.net/14_dom.html#h_MAsyozbjjZ

ÜBERSICHT

- JavaScript im Browser
- Document Object Model
- Event Handling im Browser
- Kleiner Exkurs: jQuery
- SVG, Canvas und Web-Storage

EVENT HANDLING IM BROWSER

- Im Browser können Event Handler registriert werden
- Methode: `addEventListener`
- Erstes Argument: Ereignistyp
- Zweites Argument: Funktion, die beim Eintreten des Events aufgerufen werden soll

```
<p>Click this document to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?")
  })
</script>
```

Speaker notes

Das Ereignis wurde hier am `window`-Objekt registriert. Das ist das globale Objekt der JavaScript-Umgebung im Browser und repräsentiert das geöffnete Browser-Fenster.

EVENTS UND DOM-KNOTEN

- Events können auch an DOM-Elementen registriert werden
- Nur Ereignisse, die im Kontext dieses Elements auftreten, werden dann berücksichtigt

```
<button>Click me</button>
<p>No handler here.</p>
<script>
  let button = document.querySelector("button")
  button.addEventListener("click", () => {
    console.log("Button clicked.")
  })
</script>
```

Speaker notes

Es ist auch möglich, Events als HTML- oder DOM-Attribute mit vorangestelltem "on" anzugeben, also zum Beispiel als `onclick`-Attribut. Nachteil: Auf diese Weise kann einem Element nur ein Handler eines bestimmten Typs hinzugefügt werden.

Auch gibt es die Möglichkeit, Events mit `removeEventListener` wieder zu entfernen:

```
<button>Act-once button</button>
<script>
  let button = document.querySelector("button")
  function once () {
    console.log("Done.")
    button.removeEventListener("click", once)
  }
  button.addEventListener("click", once)
</script>
```

EVENT-OBJEKT

- Nähere Informationen über das eingetretene Ereignis
- Wird dem Event Handler automatisch übergeben
- Je nach Ereignistyp verschiedene Attribute
- Bei Mouse Events z.B. x und y (Koordinaten)

```
<script>
  let button = document.querySelector("button")
  button.addEventListener("click", (e) => {
    console.log("x="+e.x+", y="+e.y)
  })
  // z.B.: x=57, y=14
</script>
```

https://developer.mozilla.org/en-US/docs/Web/API/Event

Speaker notes

Weiteres Beispiel:

```
<button>Click me any way you want</button>
<script>
  let button = document.querySelector("button")
  button.addEventListener("mousedown", event => {
    if (event.button == 0) {
      console.log("Left button")
    } else if (event.button == 1) {
      console.log("Middle button")
    } else if (event.button == 2) {
      console.log("Right button")
    }
  })
</script>
```

EVENT-WEITERLEITUNG

- Ereignisse werden für Knoten im DOM-Baum registriert
- Reagieren auch, wenn Ereignis an untergeordnetem Knoten auftritt
- Alle Handler nach oben bis zur Wurzel des Dokuments ausgeführt
- Bis ein Handler `stopPropagation()` auf dem Event-Objekt aufruft

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  let para = document.querySelector("p")
  let button = document.querySelector("button")
  para.addEventListener("mousedown", () => {
    console.log("Handler for paragraph.")
  })
  button.addEventListener("mousedown", event => {
    console.log("Handler for button.")
    if (event.button == 2) event.stopPropagation()
  })
</script>
```

Speaker notes

Im Beispiel wird `stopPropagation` aufgerufen, wenn das Klick-Ereignis mit der rechten Maustaste ausgelöst wurde. Dann wird nur der Handler am Button aufgerufen. Ansonsten werden beide, der am Button und der im Absatz aufgerufen.

EVENT-WEITERLEITUNG

- Element, bei welchem das Ereignis ausgelöst wurde: `event.target`
- Element, bei welchem das Ereignis registriert wurde: `event.currentTarget`

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", event => {
    if (event.target.nodeName == "BUTTON") {
      console.log("Clicked", event.target.textContent)
    }
  })
</script>
```

Speaker notes

Auf diese Weise kann ein Event Handler an einem zentralen Knoten angehängt werden, obwohl die Events eigentlich bei untergeordneten Knoten interessieren. Mittels `target` kann man dann feststellen, welches Element das Event ausgelöst hat.

jQuery kenn zu diesem Zweck übrigens so genannte "Delegated Events": Es wird ebenfalls ein zentraler Event Handler verwendet, aber nur an bestimmten untergeordneten Elementen (angegeben durch einen CSS-Selektor) auftretende Ereignisse werden tatsächlich berücksichtigt.

DEFAULT-VERHALTEN

- Viele Ereignisse haben ein Default-Verhalten
- Beispiel: auf einen Link klicken
- Eigene Handler werden vor Default-Verhalten ausgeführt
- Aufruf von `preventDefault()` auf Event-Objekt verhindert Default-Verhalten

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a")
  link.addEventListener("click", event => {
    console.log("Nope.")
    event.preventDefault()
  })
</script>
```

Speaker notes

Das Default-Verhalten abzuschalten sollt gut überlegt werden. Gute *User Experience* heisst auch, entsprechend den Erwartungen der Anwender auf Ereignisse zu reagieren.

Rückgabe von `false` vom Event Handler bewirkt das Verhindern sowohl der Weiterleitung (`stopPropagation`) als auch des Default-Verhaltens (`preventDefault`).

TASTATUR-EREIGNISSE

```
<p>Press Control-Space to continue.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == " " && event.ctrlKey) {
      console.log("Continuing!")
    }
  })
</script>
```

- Ereignisse `keydown` und `keyup`
- Modifier-Tasten als Attribute des Event-Objekts
- Achtung: `keydown` kann bei längerem Drücken mehrfach auslösen

Speaker notes

Auslöser der Tastatur-Ereignisse ist das DOM-Element, das aktuell den Fokus hat (Formularelemente oder Elemente mit `tabindex`-Attribut), oder `document.body`.

ZEIGER-EREIGNISSE

- Mausklicks:
  `mousedown`, `mouseup`, `click`, `dblclick`
- Mausbewegung:
  `mousemove`
- Berührung (Touch-Display):
  `touchstart`, `touchmove`, `touchend`

Speaker notes

Die Positionen können den Attributen `clientX` und `clientY` entnommen werden, die die Koordinaten in Pixel relativ zur linken oberen Fensterecke enthalten. Oder `pageX` und `pageY` relativ zur linken oberen Ecke des Dokuments.

Bei einem Touch-Screen können mehrere Finger gleichzeitig auf dem Display sein. In diesem Fall hat das Event-Objekt eine Attribut `touches`, das ein Array-ähnliches Objekt von Punkten mit den entsprechenden Koordinaten enthält.

Bei den Touch Events ist es häufig wichtig, `preventDefault` aufzurufen, da fast alle diese Ereignisse ein Default-Verhalten haben.

Demos:
https://eloquentjavascript.net/15_event.html#h_cF46QKpzec

SCROLL-EREIGNISSE

- Ereignis-Typ: `scroll`
- Attribute des Event-Objekts: `pageYOffset`, `pageXOffset`

```
window.addEventListener("scroll", () => {
  let max = document.body.scrollHeight - innerHeight
  bar.style.width = `${(pageYOffset / max) * 100}%`
})
```

https://eloquentjavascript.net/15_event.html#h_xGSp7W5DAZ

Speaker notes

Das Beispiel zeigt die vertikale Scroll-Position in einer horizontalen Leiste an.

Das Scrollen kann übrigens nicht mit `preventDefault` verhindert werden.

FOKUS- UND LADE-EREIGNISSE

- Fokus erhalten/verlieren: `focus`, `blur`
- Seite wurde geladen: `load`
  - Ausgelöst auf `window` und `document.body`
  - Elemente mit externen Ressourcen (`img`) unterstützen ebenfalls `load`-Events
  - Bevor Seite verlassen wird: `beforeunload`
- Diese Ereignisse werden nicht propagiert

WEB WORKERS

- Laufen parallel zum Haupt-Script
- Ziel: aufwändige Berechnungen blockieren nicht die Event Loop

```
// squareworker.js
addEventListener("message", event => {
  postMessage(event.data * event.data)
})

// main script
let squareWorker = new Worker("code/squareworker.js")
squareWorker.addEventListener("message", event => {
  console.log("The worker responded:", event.data)
})
squareWorker.postMessage(10)
squareWorker.postMessage(24)
```

VERZÖGERTES BEARBEITEN

- Bestimmte Ereignisse in schneller Folge ausgelöst
- Zum Beispiel: `mousemove`, `scroll`
- Ereignisbearbeitung auf Wesentliches reduzieren
- Oder jeweils mehrere Ereignisse zusammenfassen

```
<textarea>Type something here...</textarea>
<script>
  let textarea = document.querySelector("textarea")
  let timeout
  textarea.addEventListener("input", () => {
    clearTimeout(timeout)
    timeout = setTimeout(() => console.log("Typed: " + textarea.value), 500)
  })
</script>
```

Speaker notes

Beim ersten Aufruf von `clearTimeout` ist `timeout` noch undefined. Das ist unproblematisch, die Funktion macht in diesem Fall einfach nichts.

Im Beispiel werden die Tastendrücke erst verarbeitet, wenn eine kleine Pause (hier 0.5s) auftritt. In manchen Situationen möchte man aber die Ereignisse kontinuierlich verarbeiten, aber nicht jedes einzelne Ereignis, sondern mit bestimmten Mindestabständen. Dann ist eine etwas andere Vorgehensweise sinnvoll:

```
let scheduled = null
window.addEventListener("mousemove", event => {
  if (!scheduled) {
    setTimeout(() => {
      document.body.textContent =
        `Mouse at ${scheduled.pageX}, ${scheduled.pageY}`
      scheduled = null
    }, 1000)
  }
  scheduled = event
})
```

ÜBERSICHT

- JavaScript im Browser
- Document Object Model
- Event Handling im Browser
- Kleiner Exkurs: jQuery
- SVG, Canvas und Web-Storage

JQUERY

- DOM-Scripting ist oft mühsam
- Grund: unübersichtliche, inkonsistente API
- Abhilfe für lange Zeit: jQuery
  - DOM-Element mit CSS-Selektor auswählen
  - Einfache Anpassungen am DOM
  - Asynchrone Serverzugriffe (Ajax)
- Bedeutung von jQuery heute abnehmend

Speaker notes

jQuery war lange Zeit eine fast unerlässliche Bibliothek in der Web-Entwicklung. In letzter Zeit hat die Bedeutung mit zunehmendem Fortschritt nativer JavaScript-APIs aber abgenommen.

DOM-Element mit CSS-Selektor auswählen

Das geht in JavaScript mittlerweile mit `querySelector` und `querySelectorAll` (s.o.). Diese Möglichkeit stand aber lange Zeit nicht zur Verfügung.

Einfache Anpassungen am DOM

Mit Frameworks wie React.js sind direkte DOM-Manipulationen weniger nötig.

Asynchrone Serverzugriffe (Ajax)

Auch dafür ist jQuery mittlerweile nicht mehr nötig, da mit der Fetch-API (in einer späteren Lektion behandelt) eine Alternative zur Verfügung steht.

JQUERY: DOM UND EVENTS

```
$("button.continue").html("Next Step...")

var hiddenBox = $("#banner-message")
$("#button-container button").on("click", function(event) {
  hiddenBox.show()
})
```

- `$(<selector>)` erzeugt jQuery Objekt, das eine Sammlung von DOM-Elementen enthält
- Darauf sind zahlreiche Methoden anwendbar
- DOM-Traversal und -Manipulation sehr einfach

https://api.jquery.com
http://jqapi.com

JQUERY: ÜBERBLICK

Aufruf	Bedeutung	Beispiel
$(*Funktion*)	DOM ready	$(function() { .... });
$("*CSS Selektor*") .aktion(arg1, .,..) .aktion(...)	Wrapped Set - Knoten, die Sel. erfüllen - eingepackt in jQuery Obj.	$(".toggleButton").attr("title") $(".toggleButton").attr("title", "click here") $(".toggleButton").attr({title : "click here", ...}) $(".toggleButton").attr("title", function(){...}) .css(...) .text(...) .on("click", function(event) { ...})
$("*HTML-Code*")	Wrapped Set - neuer Knoten - eingepackt in jQuery Obj. - noch nicht im DOM	$(" ... ").addClass(...) .appendTo("*Selektor*") $(" ... ").length $(" ... ")[0]
$(DOM-Knoten)	Wrapped Set - dieser Knoten - eingepackt in jQuery Obj.	$(document.body) $(this)

Speaker notes

Fazit: Wenn Code geschrieben werden soll, mit dem in grösserem Umfang direkt mit dem DOM gearbeitet werden soll, kann jQuery noch sehr nützlich sein. Zumal der Overhead dieser Bibliothek sich in Grenzen hält, zumindest seit mit dem *Slim Build* auch eine schlanke Variante verfügbar ist, ohne Ajax- und Effect-Funktionen.

- http://youmightnotneedjquery.com
- https://github.com/nefe/You-Dont-Need-jQuery

ÜBERSICHT

- JavaScript im Browser
- Document Object Model
- Event Handling im Browser
- Kleiner Exkurs: jQuery
- SVG, Canvas und Web-Storage

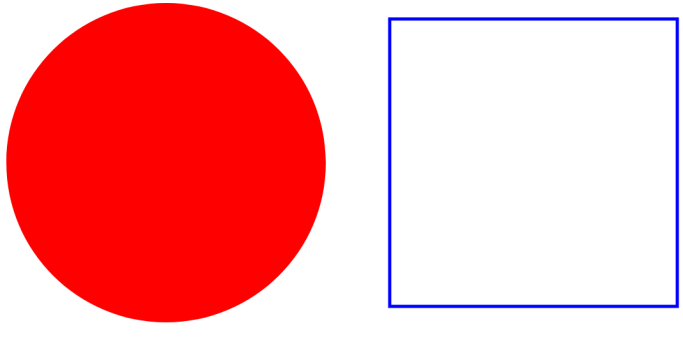
WEB-GRAFIKEN

- Einfache Grafiken mit HTML und CSS möglich
- Zum Beispiel: Balkendiagramme
- Alternative für Vektorgrafiken: SVG
- Alternative für Pixelgrafiken: Canvas

SVG

```
<p>Normal HTML here.</p>
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
        stroke="blue" fill="none"/>
</svg>
```

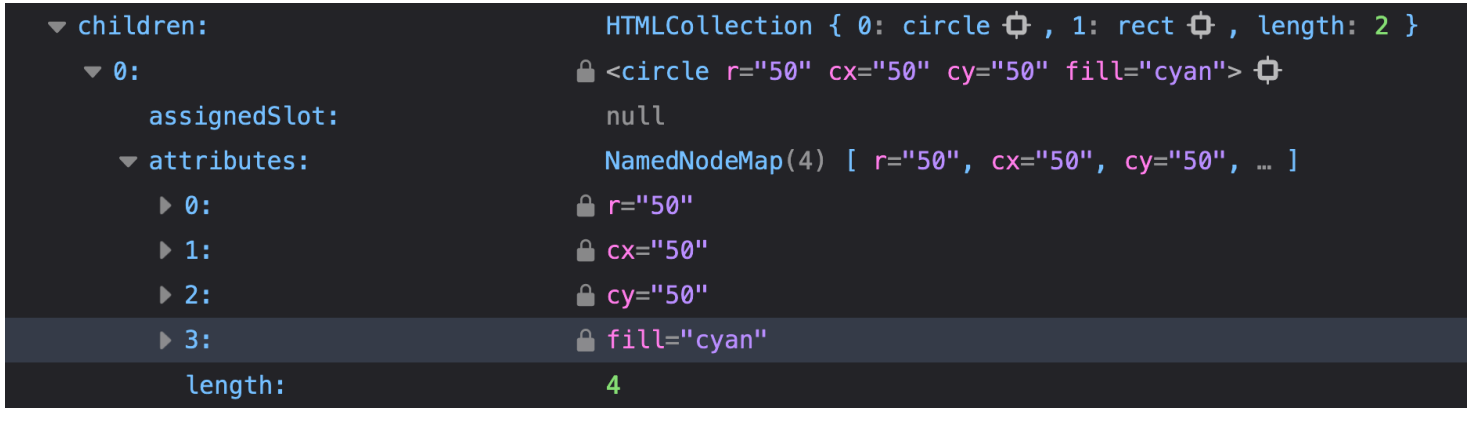
Normal HTML here.



SVG

- Basiert wie HTML auf XML
- Elemente repräsentieren grafische Formen
- Ins DOM integriert und durch Scripts anpassbar

```
let circle = document.querySelector("circle")
circle.setAttribute("fill", "cyan")
```



CANVAS

- Element `canvas` als Zeichenbereich im Dokument
- API zum Zeichnen auf dem Canvas

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
  let canvas = document.querySelector("canvas")
  let context = canvas.getContext("2d")
  context.fillStyle = "red"
  context.fillRect(10, 10, 100, 50)
</script>
```

Speaker notes

Statt dem "2d"-Kontext gibt es als Alternative noch "webgl" für 3D-Grafiken, welche die OpenGL-Schnittstelle verwendet.

Im DOM ist nicht mehr sichtbar, als dass hier ein `canvas`-Element ist. Dort ist auch nur eine Menge von Pixeln gespeichert. Dass es einmal als Rechteck angelegt wurde, ist hier nicht mehr zu erkennen.

Die Parameter von `fillRect` sind x und y für die linke obere Ecke, sowie die Breite und Höhe des Rechtecks. Soll statt einem ausgefüllten Rechteck nur die Linie gezeichnet werden, kommt `strokeRect` zum Einsatz. Stil und Dicke der Linie werden als Attribute des Kontext-Objekts festgelegt:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d")
  cx.strokeStyle = "blue"
  cx.strokeRect(5, 5, 50, 50)
  cx.lineWidth = 5
  cx.strokeRect(135, 5, 50, 50)
</script>
```

CANVAS: PFADE

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d")
  cx.beginPath()
  cx.moveTo(50, 10)
  cx.lineTo(10, 70)
  cx.lineTo(90, 70)
  cx.fill()
</script>
```

Speaker notes

Hier werden nur zwei Linien explizit gezeichnet, das `fill` füllt aber das damit festgelegte Dreieck. `cx.stroke()` würde aber trotzdem nur zwei Linien zeichnen.

Mit `closePath` kann ein Pfad explizit geschlossen werden.

Stimmt, die Cancas-API ist etwas gewöhnungsbedürftig.

CANVAS: WEITERE MÖGLICHKEITEN (1)

- Quadratische Kurven: `quadraticCurveTo`
- Bezier-Kurven: `bezierCurveTo`
- Kreisabschnitte: `arc`
- Text: `fillText`, `strokeText` (und: `font`-Attribut)
- Bild: `drawImage`

Speaker notes

In Eloquent JavaScript Kapitel 17 Abschnitt "Drawing a pie chart" (https://eloquentjavascript.net/17_canvas.html#h_9yOdkmATfT) ist beschrieben, wie man ein Tortendiagramm zeichnen kann. Normalerweise verwendet man für solche Zwecke aber eine Bibliothek, die entsprechende Abstraktionen zur Verfügung stellt.

Im folgenden Beispiel wird eine Bilddatei (hat.png) mehrfach in ein Canvas gezeichnet. Zunächst wird es in ein `img`-Element geladen, das nicht ins DOM eingehängt ist. Erst nach dem vollständigen Laden des Bilds (`load`-Event) wird es ins Canvas übernommen.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d")
  let img = document.createElement("img")
  img.src = "img/hat.png"
  img.addEventListener("load", () => {
    for (let x = 10; x < 200; x += 30) {
      cx.drawImage(img, x, 10)
    }
  })
</script>
```

CANVAS: WEITERE MÖGLICHKEITEN (2)

- Skalieren: `scale`
- Koordinatensystem verschieben: `translate`
- Koordinatensystem rotieren: `rotate`
- Transformationen auf Stack speichern: `save`
- Letzten Zustand wiederherstellen: `restore`

HTML, SVG, CANVAS

HTML

- einfach
- Textblöcke mit Umbruch, Ausrichtung etc.

SVG

- beliebig skalierbar
- Struktur im DOM abgebildet
- Events auf einzelnen Elementen

Canvas

- einfache Datenstruktur: Ebene mit Pixeln
- Pixelbilder verarbeiten

WEB STORAGE

- Speichern von Daten clientseitig
- Einfache Variante: Cookies (s. spätere Lektion)
- Einfache Alternative: LocalStorage
- Mehr Features: IndexedDB (nicht Stoff von WBE)

LOCALSTORAGE

```
localStorage.setItem("username", "bkrt")
console.log(localStorage.getItem("username"))    // → bkrt
localStorage.removeItem("username")
```

- Bleibt nach Schliessen des Browsers erhalten
- In Developer Tools einsehbar und änderbar
- Alternative solange Browser/Tab geöffnet: `sessionStorage`

Speaker notes

localStorage ist also eine einfache Attribut-Wert-Datenbank. Tatsächlich werden die Daten auch direkt über das `localStorage`-Objekt zugänglich:

```
localStorage["username"] = "bkrt"
localStorage.username = "bkrt"
```

LOCALSTORAGE

- Gespeichert nach Domains
- Limit für verfügbaren Speicherplatz pro Website (~5MB)
- Attributwerte als Strings gespeichert
- Konsequenz: Objekte mit JSON codieren

```
let user = {name: "Hans", highscore: 234}
localStorage.setItem(JSON.stringify(user))
```

QUELLEN

- Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
  https://eloquentjavascript.net/
- Ältere Slides aus WEB2 und WEB3

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

- Kapitel 14, 15 und 17 von:
  Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
  https://eloquentjavascript.net/