WBE: BROWSER-TECHNOLOGIEN

JAVASCRIPT IM BROWSER (TEIL 2)

ÜBERSICHT

- Event Handling
- Kleiner Exkurs: jQuery
- Bilder und Grafiken
- Weitere Browser-APIs

ÜBERSICHT

- Event Handling
- Kleiner Exkurs: jQuery
- Bilder und Grafiken
- Weitere Browser-APIs

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.

EVENT REGISTRIEREN

- 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.")
  })

  /* oder: button.onclick = () => {...} */
</script>
```

Speaker notes

Es ist auch möglich, Events als HTML- oder DOM-Attribute mit vorangestelltem "on" anzugeben, also zum Beispiel als `onclick`-Attribut.

Im Sinne der Trennung von Struktur und Verhalten ist es aber keine so gute Idee, die Events im HTML-Code zu notieren. Besser ist es, die Events wie im Beispiel via JavaScript an den DOM-Elementen zu registrieren, entweder mit `addEventListener` oder über die Event-Attribute `onclick` etc.

Ein Nachteil der Event-Attribute ist dabei, dass einem Element nur ein Handler eines bestimmten Typs hinzugefügt werden kann.

HANDLER ENTFERNEN

- Entfernen von Handlern mit `removeEventListener`
- Beispiel – Ereignis nur einmal behandeln:

```
<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

EVENT-OBJEKT

```
window.addEventListener("mousedown", console.log)
```

```
altKey: false
bubbles: true
button: 0
buttons: 1
cancelBubble: false
cancelable: true
clientX: 314
clientY: 389
composed: true
ctrlKey: false
currentTarget: null
defaultPrevented: false
detail: 1
eventPhase: 0
▸ explicitOriginalTarget: <html lang="en">
isTrusted: true
layerX: 314
layerY: 389
metaKey: false
movementX: 0
movementY: 0
mozInputSource: 1
mozPressure: 0
offsetX: 0
offsetY: 0
▸ originalTarget: <html lang="en">
pageX: 314
pageY: 389
rangeOffset: 0
rangeParent: null
region: ""
relatedTarget: null
returnValue: true
screenX: 1156
screenY: 499
shiftKey: false
▸ srcElement: <html lang="en">
▸ target: <html lang="en">
timeStamp: 5249
type: "mousedown"
▸ view: Window file:///Users/demo.html
which: 1
x: 314
y: 389
```

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>
  document.querySelector("p").addEventListener("mousedown", () => {
    console.log("Handler for paragraph.")
  })
  document.querySelector("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 Ereignis ausgelöst hat.

Ein solches Vorgehen kann sehr nützlich sein. Angenommen wir haben auf der Seite eine Liste mit einigen Listenelementen. Beim Klick auf ein Listenelement soll eine bestimmte Funktion ausgeführt werden. Das kann man erreichen, indem an jedem Listenelement ein Ereignis registriert wird. Wenn die Liste nun dynamisch um weitere Einträge erweitert wird, muss für jeden der neuen Einträge ebenfalls die Ereignisbehandlung eingerichtet werden. Das ist nicht der Fall, wenn die Ereignisbehandlung von Anfang an für einen übergeordneten Knoten eingerichtet wird.

jQuery unterstützt die Weiterleitung übrigens über 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 vorher 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

BEISPIEL: BILD LADEN

- Funktion `loadImage` soll Bild laden
- Callbacks für Erfolg und Fehler

```
1  loadImage('zhaw.png',
2
3    function onsuccess (img) {
4      document.body.appendChild(img)
5    },
6
7    function onerror (e) {
8      console.log('Error occured while loading image')
9      console.log(e)
10   }
11
12 )
```

BEISPIEL: BILD LADEN

```
1  function loadImage (url, success, error) {
2    var img = new Image()
3    img.src = url
4
5    img.onload = function () {
6      success(img)
7    }
8
9    img.onerror = function (e) {
10     error(e)
11   }
12 }
```

BEISPIEL: MIT PROMISE

```
1  function loadImage (url) {
2    var promise = new Promise(
3      function (resolve, reject) {
4        var img = new Image()
5        img.src = url
6
7        img.onload = function () {
8          resolve(img)
9        }
10
11       img.onerror = function (e) {
12         reject(e)
13       }
14     }
15   )
16   return promise
17 }
```

BEISPIEL: MIT PROMISE

```
1 loadImage('zhaw.png')
2   .then(function (img) {
3     document.body.appendChild(img)
4   })
5   .catch(function (e) {
6     console.log('Error occured while loading image')
7     console.log(e)
8   })
```

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
})
```

In diesem Beispiel wird beim ersten Eintreten des Ereignisses der Timer gestartet. Weitere Ereignise innerhalb der nächsten Sekunde werden ignoriert (könnten aber verarbeitet werden). Nach Ablauf des Timers erfolgt die Ausgabe entsprechend dem Event-Objekt beim letzten Eintreten des Ereignisses und `scheduled` wird wieder auf `null` gesetzt. Beim nächsten Ereignis beginnt der Ablauf von vorn.

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

- Event Handling
- Kleiner Exkurs: jQuery
- Bilder und Grafiken
- Weitere Browser-APIs

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)

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)

JQUERY: BEDEUTUNG ABNEHMEND

- DOM-Element mit CSS-Selektor auswählen
  → `querySelector`, `querySelectorAll`

- Einfache Anpassungen am DOM
  → mit Framworks wie React.js weniger nötig

- Asynchrone Serverzugriffe (Ajax)
  → Fetch API (spätere Lektion)

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.

`querySelector` und `querySelectorAll` sind DOM-Methoden, die lange Zeit nicht zur Verfügung standen. Die Auswahl von Elementen basierend auf einem CSS-Selektor war eine der Stärken von jQuery.

Zum zweiten Punkt: mit Frameworks wie React.js vermeidet man direkte DOM-Manipulationen (bzw. man überlässt diese dem Framework). Wir werden ein solches Framework in den letzten Wochen des Semesters entwickeln.

Zum asynchronen Zugriff auf Server wird jQuery ebenfalls kaum noch benötigt, da mit der Fetch-API eine leistungsfähige Alternative verfügbar ist. In der nächsten Lektion wird auf asynchrone Client-Server-Interaktion eingegangen.

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

- Event Handling
- Kleiner Exkurs: jQuery
- Bilder und Grafiken
- Weitere Browser-APIs

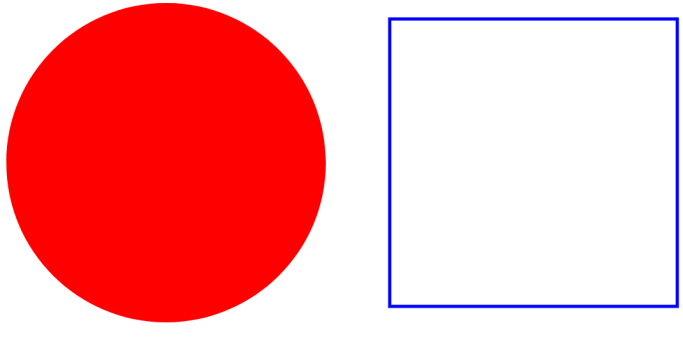
WEB-GRAFIKEN

- 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>
```

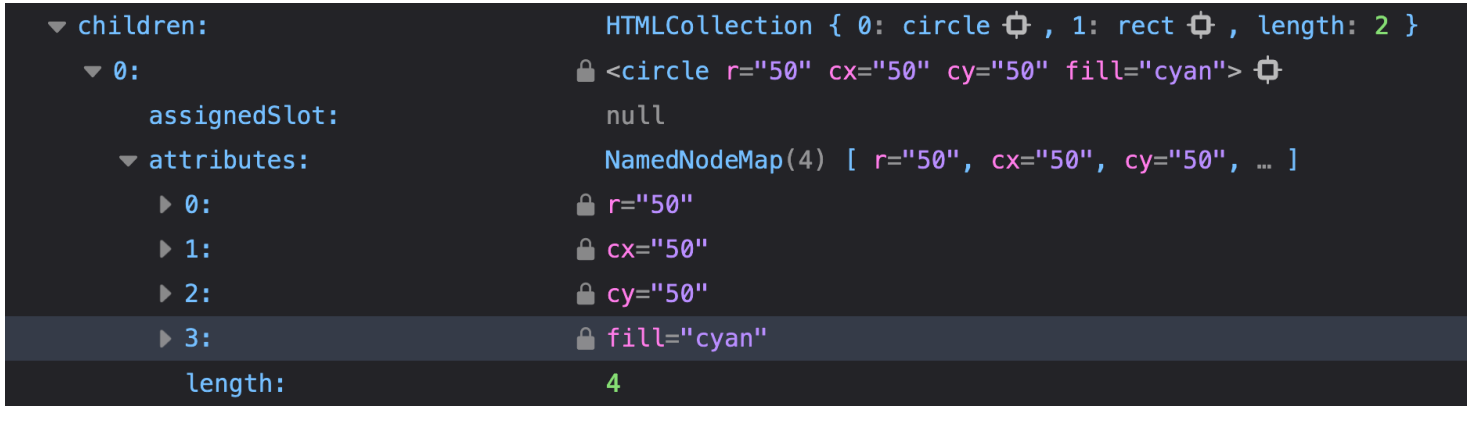
Normal HTML here.



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 nur sichtbar, 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.

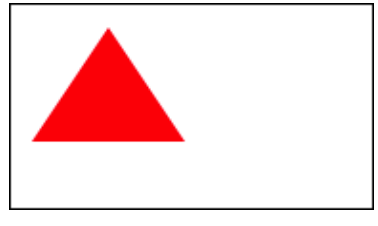
Die Parameter von `fillRect` sind x und y für die linke obere Ecke, sowie die Breite und Höhe des Rechtecks. Soll statt einem ausgefüllten Rechteck nur die Linie gezeichnet werden, kommt `strokeRect` zum Einsatz. Stil und Dicke der Linie werden als Attribute des Kontext-Objekts festgelegt:

```
<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 Canvas-API ist etwas gewöhnungsbedürftig.

CANVAS: WEITERE MÖGLICHKEITEN (1)

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

CANVAS: BILD EINFÜGEN

```
<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>
```

Speaker notes

In diesem 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.

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.

What are the best JavaScript drawing libraries?
https://www.slant.co/topics/28/~best-javascript-drawing-libraries

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

ÜBERSICHT

- Event Handling
- Kleiner Exkurs: jQuery
- Bilder und Grafiken
- Weitere Browser-APIs

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))
```

HISTORY

- Zugriff auf den Verlauf des aktuellen Fensters/Tabs
- `length`: Anzahl Einträge inkl. aktueller Seite
- `back()`: zurück zur letzten Seite
- Seit HTML5 mehr Kontrolle über den Verlauf: `pushState()`, `replaceState()`, `popstate`-Event

```
function goBack () {
  window.history.back()
}
```

MDN: Working with the History API

GEOLOCATION

```
var options = { enableHighAccuracy: true, timeout: 5000, maximumAge: 0 }

function success(pos) {
  var crd = pos.coords
  console.log(`Latitude : ${crd.latitude}`)
  console.log(`Longitude: ${crd.longitude}`)
  console.log(`More or less ${crd.accuracy} meters.`)
}

function error(err) { ... }

navigator.geolocation.getCurrentPosition(success, error, options)
```

MDN: Geolocation API

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)
```

QUELLEN

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

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

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