


Hacking the Museum

JavaScript-Tutorial

Dozentinnen

Melanie Krauth
Antje Goldau

 JS

EUROPÄISCHE UNION
Europäischer Fonds für
regionale Entwicklung

Die Dozentinnen

Melanie Krauth

Studium Medientechnik (Dipl.–Ing.)
Studium Medieninformatik (M.Sc.)

SHK Fraunhofer (Java, C++)
Zalando Logistik (Java, Postgre SQL)
Reactive Core (Scala, Javascript, C++)
Eidu (C#)
Museum für Naturkunde Berlin (JavaScript)

Mag: Algorithmen und Wald

Mag nicht: Konfigurieren

Antje Goldau

Studium Informations und Medientechnik (1.-3.Sem)
Studiengangswechsel zu Informatik (3.Sem-heute)

FU Berlin im Mentoring tätig gewesen
Museum für Naturkunde Berlin(SHK)

Mag: theoretische Informatik und kreativ sein

Mag nicht: Elektrotechnik

Kennenlernrunde



Tag 1

Geschichte & Entwicklung von JS

DOM

Übungen im Browser

Basistypen, var, let, scope

Vergleichsoperatoren

Kontrollstrukturen

Semikolon & ASI

Strict Mode

Funktionen



Mittagpause (12:30-14:00)

Praktische Arbeit am Memory-Spiel (Kapitel 1 und 2)

Tag 2

Derzeitiger Stand
Fragerunde
Fetch & Promise
Übersicht IIIF
IIIF in einem JSFiddle ausprobieren

Mittagpause (12:30-14:00)

Weiterarbeit am Memory-Spiel (Kapitel 3 und 4)
Evaluationsrunde (ca. letzte halbe Stunde)

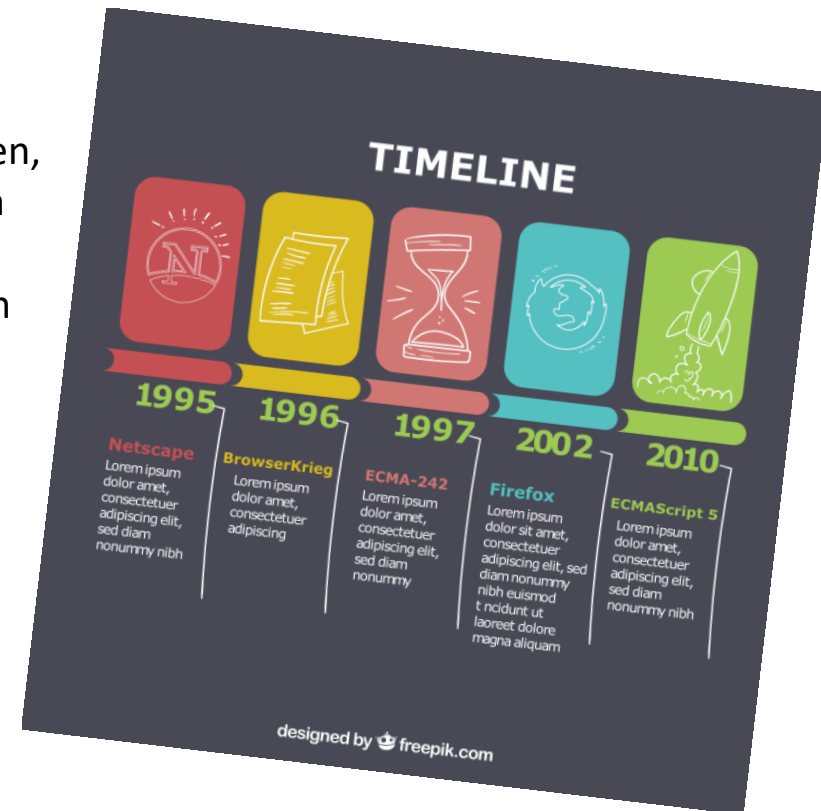


Einführung in JavaScript

A large, bold, black 'JS' logo is centered within a bright yellow rectangular area. The 'J' and 'S' are stylized with thick strokes and rounded terminals. The yellow area is set against a solid green background.

Geschichte und Entwicklung

- 1995 von Netscape entwickelte Skriptsprache um die Möglichkeiten von HTML und CSS zu erweitern
- ermöglicht die Auswertung, von Benutzerinteraktionen, Veränderung, nachladen und generieren von Inhalten
- Wird heute auch außerhalb des Browsers, auf Servern und Microcontrollern eingesetzt
- JavaScript ist:
 - ✓ dynamisch Typisiert
 - ✓ Objektorientiert
 - ✓ Klassenlos
 - Nicht Java!



DOM – Document Object Model

Das Document Object Model(DOM) repräsentiert eine Seite, die vom Browser dargestellt wird.

Es stellt eine Schnittstelle für HTML und XML Dokumente dar, worüber Programme Inhalt, Struktur und Aussehen des Dokuments ändern können.

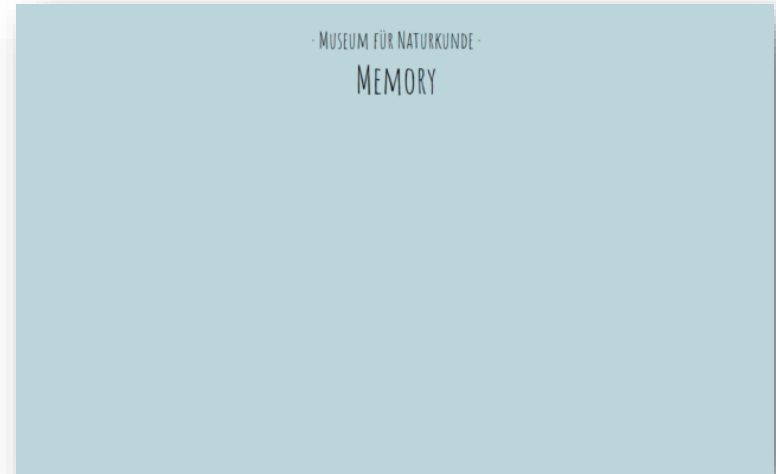
```
<!doctype html>
<html lang="de">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">

  <title>Memory</title>
  <link rel="stylesheet" href="style.css">
</head>

<body>
  <div>
    <p class='p3'> &#183; Museum für Naturkunde &#183;</style></p>
    <p class='p5'>Memory</p>
  </div>
</body>
</html>
```

HTML-Code



Umsetzung des Browsers

DOM – Document Object Model

APIs die wir einsetzen werden:

<code>document.getElementById(id)</code>	→ element
<code>document.getElementsByClassName(className)</code>	→ NodeList[element]
<code>document.createElement(name)</code>	→ element
<code>document.querySelectorAll(className)</code>	→ NodeList[element]
<code>element.innerHTML = text</code>	→ text
<code>element.classList.add(className)</code>	→ undefined
<code>element.classList.remove(className)</code>	→ undefined
<code>element.style.background = color</code>	→ color
<code>element.appendChild(element)</code>	→ element



Ausprobieren im Browser

Datentypen

Primitive:

- *number* inkl. NaN und Infinity
- *string*
- *boolean*
- *undefined*
- (Symbol)

(13)

(true)

Nicht primitive:

- function
- object (*Anmerkung*: null == object)
- wrapper-objekte
 - String(*string*)
 - Number(*number*)
 - Boolean()

(name : "Tina",
surname: "Schneider")

Datentypen

Bei Bedarf werden Werte der primitiven Typen **Number**, **Boolean** und **String** automatisch in Objekte der entsprechenden Konstruktorfunktion umgewandelt.

Beispiel:

```
let s = „hallo“  
s.length  <- wird in Objekt konvertiert bevor length  
aufgerufen wird
```

Wenn eine Variable nicht deklariert wird, bekommt diese automatisch den Wert **undefined** zugeschrieben. Methoden oder Anweisungen, die eine oder mehrere undefinierten Variablen auswerten oder deren Ausgabewert nicht deklariert wird, geben **undefined** als Ausgabe zurück.

Funktionen können **undefined** zurückgeben, wenn sie keinen Wert zurückgeben (z.B. kein `return` besitzen).

Var, let und Scope

`var` ist nur bis zum nächsten Funktionsblock gültig.

`let` bis zum nächsten schließenden Block.

`var` wird dem globalen `window`-Objekt angehängt:

Beispiel:

```
var varVariabel = „Ich bin eine var Variabel“;  
let letVariabel = „Ich bin eine let Variabel“;
```

```
console.log(window.varVariabel);  
  > Ich bin eine var Variabel  
console.log(window.letVariabel);  
  > undefined
```

Hoisting

Eine Variable, die mit `var` Deklariert wird, wird an den Anfang des Scopes gehoben. Die Definition allerdings erfolgt an der Stelle, wo sie auch im Code steht. Bei `let` ist das nicht der Fall.

Var, let und Scope

Der scope legt die Sichtbarkeit von Variablen fest.

local:

Eine Variable, die innerhalb einer Funktion definiert wurde, ist nicht außerhalb der Funktion bekannt. Sie befindet sich lokal in der Funktion.

Beispiel:

```
function beispiel(){  
    var x = 2;  
    // in dieser Funktion kann x benutzt werden  
}  
// x hier nicht mehr nutzbar
```

global:

Wird eine Variable außerhalb einer Funktion deklariert, ist sie global. Sie kann von allen Skripten und Funktionen eines Projektes erreicht werden.

Beispiel:

```
//x überall nutzbar  
var x = 2;  
function beispiel(){  
    x++;    //x ist hier bekannt  
}
```

typeof und instanceof

`typeof`-Operator gibt den Typ einer Variable oder eines Ausdrucks als Zeichenfolge (String) zurück:

`undefined, null, boolean, string, symbol, number, object`

Beispiel 1:

```
> typeof 1  
"number"
```

Beispiel 2:

```
> typeof "hello"  
"string"
```

`instanceof` prüft, ob ein Objekt von einem bestimmten Typ ist, gibt `true` zurück, wenn das angegebene Objekt vom angegebenen Typ ist.

Beispiel:

```
> [1,2] instanceof Array  
true
```

Vergleichsoperatoren

== und **===** sind Vergleichsoperatoren.

== (Equals Operator)

Dieser Operator konvertiert die Operanden, wenn sie nicht vom gleichen Typ sind und prüft dann die strikte Gleichheit dieser.

Beispiel:

```
true == 1; //true  
"2" == 2; //true
```

=== (Strict Equals Operator)

Dieser Operator führt im Gegensatz zum Equals Operator keine Typkonvertierung durch. Hier wird nur true zurückgegeben, wenn die Operanden strikt gleich sind (also auch denselben Typ haben).

Beispiel:

```
true === 1; //false  
"2" === 2; //false
```

Kontrollstrukturen

If-else

```
if (bedingung) {  
    anweisung  
} else if (bedingung) {  
    anweisung  
}  
else {  
    anweisung  
}  
  
variable = bedingung? wertWennWahr  
: wertWennFalsch;
```

switch

```
switch (variable) {  
    case wert1 :  
        anweisungen;  
        break;  
    case wert2 :  
        anweisungen;  
        break;  
    default :  
        anweisungen;  
}
```


Kontrollstrukturen-Schleifen

While

```
while (bedingung) {  
    anweisungen;  
}  
  
do {  
    anweisungen;  
} while (bedingung);
```

for

```
for (startausdruck; bedingung;  
iterationsausdruck) {  
    anweisungen;  
}  
  
for (var eigenschaftsname in  
array/objekt) {  
    anweisungen;  
}  
  
for (var wert of array) {  
    anweisungen;  
}
```

Automatic Semicolon Insertion (ASI)

Das Semikolon am Ende eines Ausdrucks ist Optional

Es gibt drei Regeln, bei denen von ASI ein Semikolon automatisch eingefügt wird.

1. Regel: Neue Zeile und unzulässiges Zeichen

Wenn eine neue Zeile begonnen wird und gleichzeitig ein Zeichen folgt, dass im jeweiligen Zusammenhang unerlaubt ist, fügt ASI automatisch ein Semikolon ein.

<pre>if (a<0) a = 0 console.log(a)</pre>	<pre>if(a<0)a=0; console.log(a);</pre>
---	---

Automatic Semicolon Insertion (ASI)

2. Regel: Verbot von Line Terminatoren an bestimmten Positionen

An bestimmten Positionen ist eine neue Zeile unzulässig. Diese Fälle sind:

- PostfixExpression: -> **x++** und **x--**
- ContinueStatement: -> **continue label;**
- BreakStatement: -> **break label ;**
- ReturnStatement: -> **return expression;**
- ThrowStatement: -> **throw expression ;**

Setzt der User dort trotzdem eine neue Zeile, fügt ASI ein Semikolon ein.

<code>return a + b</code>	<code>return; a + b;</code>
-------------------------------	---------------------------------

Automatic Semicolon Insertion (ASI)

3. Regel: Letztes Statement

Fehlt im letzten Statement in einem Block vor der schließenden Klammer oder ganzen Programm ein Semikolon, wird es von ASI hinzugefügt.

<code>function add(a,b) { return a+b }</code>	<code>function add(a,b) { return a+b; }</code>
---	--

Empfehlung: Semikolon manuell setzen!

Strict Mode

Der **Strict Mode** (strenger Modus) verhindert Fehler in der Setzung von globalen Variablen.

Beispiel 1:

```
a = 2; //unbedachtes Setzen einer globalen Variable
```

Beispiel 2:

```
var Lieblingszahl = 9;  
function zahlenAusgabe(){  
    Lieblingszahl = prompt("Das ist meine Lieblingszahl", "")  
    // durch falsche Schreibweise der vorher deklarierten globalen Variable  
    // Lieblingszahl wird eine neue globale Variable angelegt  
}
```

Strict Mode

Globale Variablen können Schwierigkeiten in einem Programm hervorrufen, wenn sie z.B. in anderen Funktionen überschrieben werden. Die Verwendung von **'use strict'** verhindert dies. Außerdem können durch das Ausgeben des **Reference-Errors** Fehler früher entdeckt werden und Programme leichter debuggt werden.

Beispiel:

```
'use strict';  
a = 2;
```

Mit `a = 2` wird normalerweise eine globale Variante deklariert. Wird jedoch **,use strict'** am Anfang deklariert, gibt der Browser einen Reference Error zurück. **'use strict'** kann auch innerhalb einer Funktion verwendet werden, so wird nur diese Funktion in den strengen Modus gesetzt.

Funktionen

In JavaScript sind Funktionen Objekte.
Funktionen können erstellt und überschrieben, als Argumente an andere Funktionen übergeben und von ihnen erzeugt sowie zurückgegeben werden.

1: Funktionsdeklaration

```
function a (parameter1, parameter2) {  
    anweisungen;  
    return ausdruck;  
}
```

Funktionen

2: Funktionsausdruck ('function expression')

2.1: Normalfall, b ist eine anonyme Funktion

```
var b = function (parameter1, parameter2){  
    anweisungen;  
    return ausdruck;  
};
```

Anonyme Funktionen können auch direkt ausgeführt werden. Dies wird häufig zur Kapselung des Gültigkeitsbereichs von Variablen verwendet.

```
( function (parameter1, parameter2){  
    anweisungen;  
    return ausdruck;  
} )(par1, par2 );
```

2.2: benannter Funktionsausdruck ('named function expression')

```
var c = function d (parameter1, parameter2) {  
    anweisungen;  
    return ausdruck;  
} ;
```


Funktionen

3: Function-Konstruktor

Ein Konstruktor ist dafür da, um Objekte zu erzeugen. Da Funktionen in JS Objekte sind, können diese auch über Konstruktoren folgendermaßen erzeugt werden:

```
var e = new Function('arg1', 'arg2', 'return arg1 + arg2;');
```

4: 'expression closure' aus JavaScript 1.8

Diese Art ist dem Lambda-Kalkül ähnlich, kommt ohne geschweifte Klammern sowie `return` aus und gibt das Ergebnis von `ausdruck` zurück.

```
function f (...) ausdruck;
```

Funktionen

5: 'arrow functions'

Benutzbar ab ECMAScript 2015.

```
x => ausdruck
x => {ausdruck
    return
}
(parameter1, parameter2) => ausdruck
(parameter1, parameter2) => {ausdruck
    return
}
```

Eine so erzeugte Funktion kann natürlich auch einer Variablen zugewiesen werden:

```
var g = (parameter1, parameter2) => ausdruck;
```

Noch Fragen?