FH Bielefeld
University of
Applied Sciences

Campus Minden

Webbasierte Anwendungen

SS 2018

JavaScript

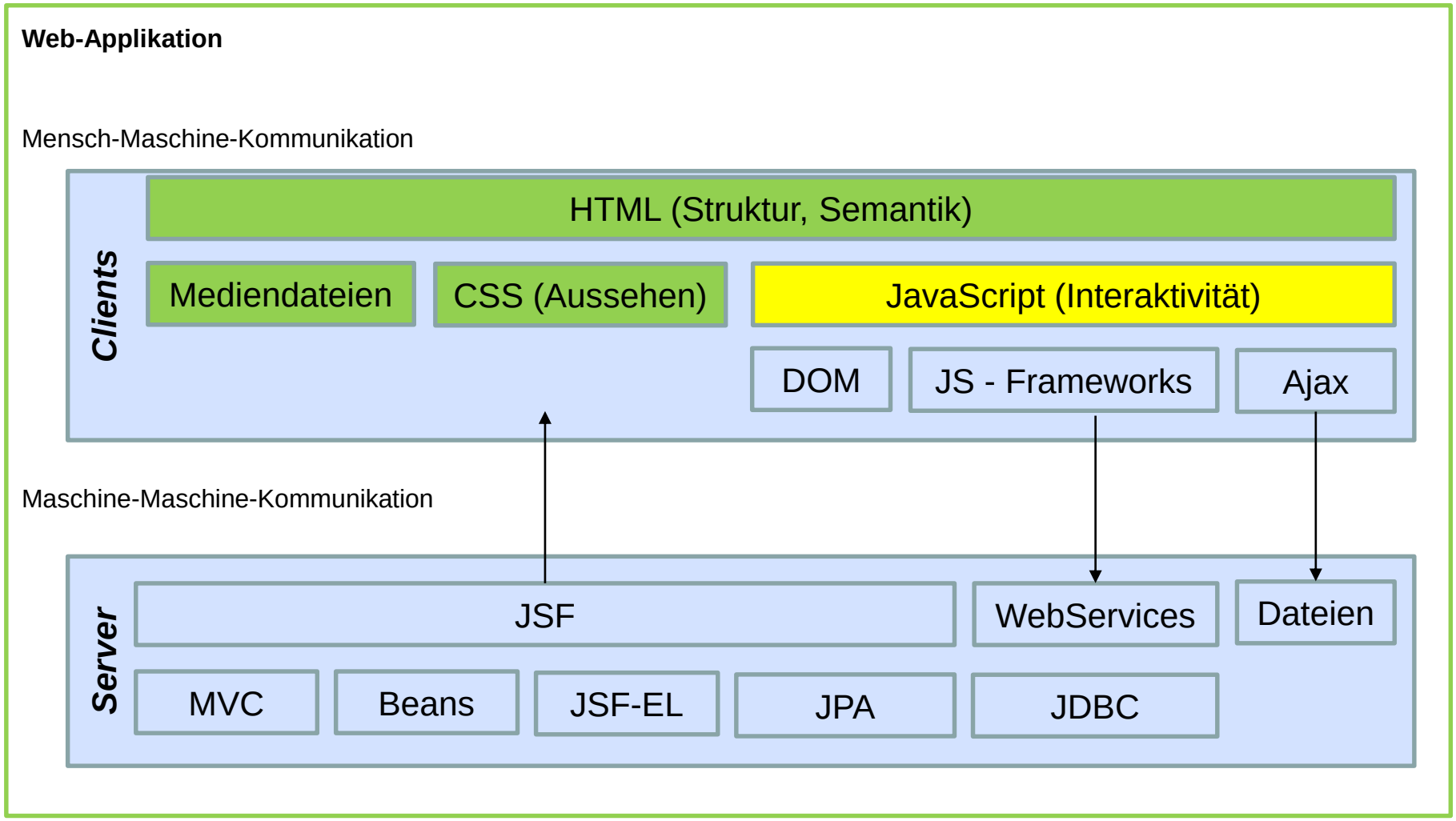
Dozent: B. Sc. Florian Fehring
mailto: florian.fehring@fh-bielefeld.de

Studiengang Informatik

JavaScript

1. **Kontext und Motivation**
2. Eigenschaften und Einbindung
3. Prozedurale Sprachelemente
4. Objektorientierte Sprachelemente
5. (a)synchrone Funktionen
6. Browser-Objekte
7. Browser APIs
8. Darüber hinaus
9. Projekt

Problemfelder



Anforderungen

Welche Anforderungen werden als nächstes bearbeitet?

TODO

- Mehrsprachen-Fähigkeit
- (lokales) Speichern von Artikeln
- Client-Position anzeigen
- Offline-Verwendung ermöglichen
- Inhaltsverzeichnisse
- Medien bearbeiten
- Formlareingaben in Seite einfügen
- Navigation über Tastaturkürzel
- Externe Inhalte einbinden
- Medien hochladen / runterladen
- Kommentare hochladen / runterladen
- Kommentare speichern
- Kommunikation untereinander

DONE

- Technologische Grundlagen erarbeiten
- Was ist eine Web-Anwendung?
- News darstellen
- Projekte vorstellen
- Aufgaben darstellen
- Formular für Kommentare
- Schickes Design für die Seite
- Mediendatein einbinden
- Animationen

JavaScript

1. Kontext und Motivation
2. **Eigenschaften und Einbindung**
3. Prozedurale Sprachelemente
4. Objektorientierte Sprachelemente
5. (a)synchrone Funktionen
6. Browser-Objekte
7. Browser APIs
8. Darüber hinaus
9. Projekt

Eigenschaften und Einbindung I

Definition: **JavaScript ist eine höhere, dynamisch-typisierte, und prototyp-basierte Skriptsprache. Sie unterstützt multi-paradigmen- und modulare Implementierung.**

Skriptsprache:

- Automatische Speicherverwaltung
- Mächtige Datenstrukturen

Höhere Sprache:

- Spracheelemente höherer Ebene wie Schleifen
- Umfangreiche integrierte Bibliotheken

Dynamisch-Typisiert:

- Variablen haben keinen festgelegten Datentyp, können zur Laufzeit unterschiedliche Inhalte annehmen

Prototyp-basiert:

- Zur Laufzeit erstellte Objekte können als Schablone für neue Objekte dienen

Multi-Paradigmen:

- JavaScripte können prozedural, funktional oder objektorientiert sein

Modulare Implementierung:

- Skripte können in andere Skripte eingebunden werden

Eigenschaften und Einbindung I

Weitere Eigenschaften:

- Plattformübergreifend (in der Vergangenheit größere Unterschiede zwischen verschiedenen Browsern)
- Beeinflusst von anderen Programmiersprachen (z.B. Phyton, Java,…)

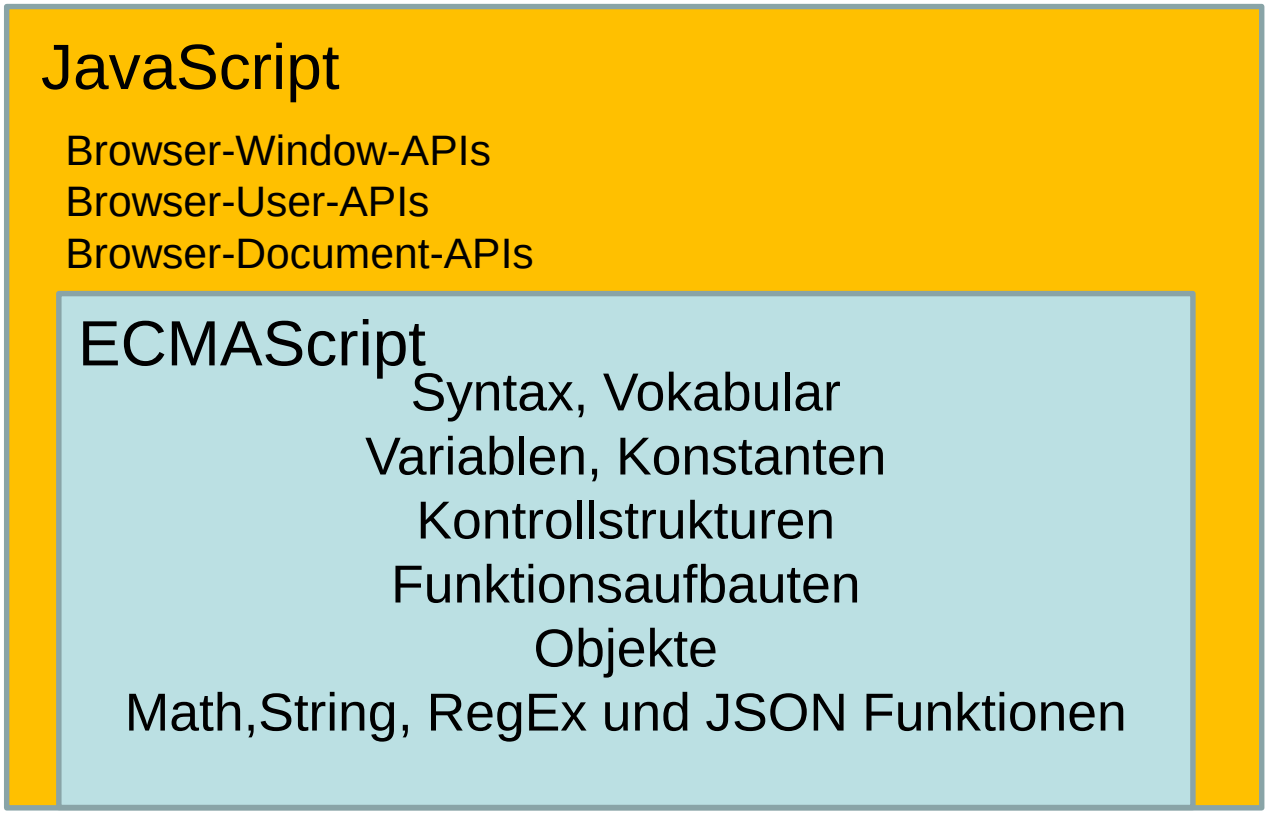
Anwendungsgebiete:

- Client-seitige Anwendungen, hauptsächlich im Webumfeld (HTML-Manipulation, Browser-Schnittstellen)
- Server-seitige Anwendungen (node.js)

ECMAScript

Definition: **ECMAScript ist der W3C standardisierte Kern von JavaScript. Er definiert die Sprachelemente.**

- Kann auch Serverseitig verwendet werden (z.B. node.js)
- Von modernen Browsern implementiert



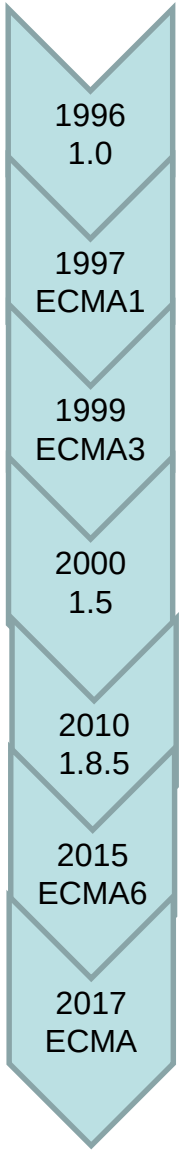
Eigenschaften und Einbindung III

Browser:

1. Implementieren einen JavaScript Interpreter (oder Compiler)
2. Bieten Zugriffsfunktionen / Objekte auf Browser-APIs

Geschichte von JavaScript:

1.0	Implementierung im NetscapeNavigator2
ECMA1	JavaScript-Kern Standardisierung
ECMA3	RegulärеAusdrücke, Fehlerbehandlung
1.5	Vollständige Implementierung ECMA3,
ECMA2015/ES6	Seit 2015 jährliche Aktualisierung. Jetzt ECMA2015
ECMA2017	import(), Rest-Parameter, class-Konstrukte



- BIS HIER 30.04.2018

Eigenschaften und Einbindung IV

```
<head>
  <script src="datei.js"></script>
  <script>…</script>
</head>
<body>
  <script src="datei.js"></script>
  <script>…</script>
  <a href="#" onclick="alert('Ausgabe')">Link</a>
</body>
```

Script-Einbindungen:

- External: Aus einer anderen Datei, im Header oder Body
- Internal: Im Script-Tag innerhalb Header oder Body
- InAttribute: In einem onclick Attribut (veraltet!)

Types:

- Typangabe kann in HTML5 entfallen

```
<script src="dat.js" type="application/javascript"></script>
```

Ausführungsreihenfolge

Definition: **JavaScripte werden ausgeführt, sobald der Browser sie vollständig geladen hat.**

Auswirkungen:

1. Skripte können ausgeführt werden, ehe das Dokument vollständig geladen ist
2. Funktionen können „zu früh“ ausgeführt werden
3. Eventuell sind noch nicht alle Abhängigkeiten geladen, wenn ein Skript startet

```
<head>
  <script>
   alert(script2.funktion());  // Ergebnis aus script2.js
  </script>
  <script src="script2.js"></script>
</head>
<body>
          …
</body>
```

JavaScript

1. Kontext und Motivation
2. Eigenschaften und Einbindung
3. **Prozedurale Sprachelemente**
4. Objektorientierte Sprachelemente
5. (a)synchrone Funktionen
6. Browser-Objekte
7. Browser APIs
8. Darüber hinaus
9. Projekt

Prozedurale Elemente I – Variablen

```
var [variablenname]
let [variablenname]                    // Seit ES2015
const [konstantenname]
```

var	Automatisch globale Variable (auch in eingebundenen Skripten)
let	Lokale Variable (Sichtbarkeit auf einen Block beschränkt)
const	Nicht veränderbarer Wert (Sichtbarkeit wie bei let)

Wertzuweisung:
Variable = Wert

Prozedurale Elemente I – Variablen

script1.js

```
var myVariable = "Text";
console.log(myVariable);  // Gibt Text aus
…                         // Irgendwelcher Code
var myVariable = "Text2"; // Neu-Deklaration
console.log(myVariable);  // Gibt Text2 aus
```

script2.js

```
console.log(myVariable);

// Gibt Text2 aus oder führt zum Fehler, //
je nachdem ob script1.js vor oder
// nach script2.js geladen wurde
```

script3.js

```
let myVariable = "Text1";
console.log(myVariable);  // Gibt Text1 aus
{
  let myVariable = "Text2";        // Block-Variable
  console.log(myVariable);         // Gibt Text2 aus
}
console.log(myVairable);  // Gibt Text1 aus
let myVariable = "Text3"  // Fehler!
```

Prozedurale Elemente I – Variablen

Regeln für Variablennamen in JavaScript:

- Beginn mit Buchstabe oder Unterstrich (Achtung : case sensitive)
- Rest kann Buchstabe, Ziffer oder das Sonderzeichen „_“ (underscore) sein
- (keine Leerzeichen, keine anderen Sonderzeichen)
- Maximallänge 32 Zeichen
- es gibt reservierte Wörter wie : var, case, for, …

```
let _2_Test_Wert                          // gültig
let 2_Test_Wert                           // ungültig
let Email_Adresse                         // gültig
let @_Adresse                             // ungültig
let Langer_Variablenname                  // gültig
let Das_ist_noch_laengerer_Variablenname  // ungültig
let Test 1                                // ungültig
let Test_1                                // gültig
let test_1                                // gültig, andere Variable als zuvor
```

Prozedurale Elemente II – Typisierung

Definition: **Eine Variable hat den Datentyp des Wertes, der ihr zugewiesen wurde. Der Datentyp kann sich zur Laufzeit ändern. (dynamische Typisierung)**

Primitive Datentypen:

`boolean`	Wahrheitswerte (true/false)
`string`	Zeichen oder Zeichenketten; 16bit je Zeichen
`number`	Zahlenwerte (Umfasst, int, long, double,…); 64bit
`null`	Variable ist definiert aber ohne Wert
`undefined`	Variable wurde nicht definiert
`symbol`	Eindeutiger, unveränderbarer Symbolwert

Objekt-Datentyp:

`object`	Für alle verschiedenen Objekte

Typeof-Operator:

- Der typeof-Operator dient zum Auslesen des Datentyps

```
let myVar = 8;
console.log(typeof myVar);          // number
myVar = "Test";                     // Alternativ auch myVar = 'Test‘;
console.log(typeof myVar);          // string
```

Prozedurale Elemente III – Operatoren

Definition: **Operatoren führen einfache Berechnungen oder Zuweisungen auf Datentypen aus.**

Arten:

- Arithmetische Operatoren
- Konkatenations-Operatoren
- Logische Operatoren
- Vergleichs-Operatoren
- Zuweisungsoperatoren
- Bit-Operatoren
- Intropersepktive-Operatoren (typeof)

Operationen und dynamische Typisierung:

- JavaScript ermittelt den passendsten Datentyp für das Ergebnis einer Operation

```
let nummer = 1;
let zeichen = '1';
var ergebnis = nummer+zeichen;
console.log(ergebnis + ' has type: ' + typeof ergebnis);    // 11 has type:
string
```

Prozedurale Elemente III – Operationen

Operatoren	**Beispiele**	**Datentyp**
Vergleichsoperatoren	==, !=, <>, <, >, <=, >= Ergebnis: boolescher Wert	Zahlen, Strings
Berechnungsoperatoren	+, -, *, /, % (Modulo), ++ (Inkrement), --(Dekrement)	Zahlen
Konkatenationsoperator	‘1 ‘+ ‘1 ‘= ‘11 ‘;	Strings
Logische Operatoren	&& - UND, || - ODER, ! . NICHT	Boolesche Werte
Bit-Operatoren	& - (1010&0110)= 0010; |- ODER, ~ - NICHT, << Linksverschiebung	Zahlen, boolesche Werte
Zuweisungsoperatoren	a=a+5; a+=5; //beide gleich	alle

Prozedurale Elemente IV - Kontrollstrukturen

```
if(bedingung){dann}
if(bedingung){dann} else{sonst}
(bedingung) ? dann : sonst
switch(bedingung) {case:dann; default sonst;}
```

Kontrollstrukturen (wie in Java):

`if(b){d}`	Wenn-Dann-Anweisung
`if(b){d} else{s}`	Wenn-Dann-Sonst-Anweisung
`(b) ? d : s`	Wenn-Dann-Sonst-Anweisung verkürzt
`switch(b) {case:d; default s;}`	Switch-Case-Anweisung

```
switch (Variable) {
   case “a“ : console.log(‘You pressed a‘);
         break;
   case “b“ : console.log(‘You pressed b‘);
         break;
   default : console.log(‘You pressed something else‘);
}
```

Weitere Informationen unter: https://www.w3schools.com/Js/js_if_else.asp
https://www.w3schools.com/Js/js_switch.asp

Prozedurale Elemente IV - Kontrollstrukturen

```
for(i=0;i<x;i++){anweisungen}
while(bedingung){anweisungen}
do{anweisungen} while(bedingung)
```

Kontrollstrukturen (wie in Java):

for	For-Schleife mit Zählvariable
while	Kopf-gesteuerte while-Schleife
do-while	Fuß-gesteuerte while-Schleife

```
for (i=0; i<10;i++) {
   console.log(i);
}
```

Weitere Informationen unter: https://www.w3schools.com/Js/js_loop_for.asp
https://www.w3schools.com/Js/js_loop_while.asp

Prozedurale Elemente IV - Funktionen

Definition: **Eine Funktion kapselt Anweisungen in einen, von anderer Stelle aufrufbaren Block. JavaScript Funktionen sind referenzierbar.**

```
function Funktionsname(Parameter 1, Parameter 2, …) {
   … // JavaScript - Anweisungen
}
```

```
// Deklariere Funktion
function function1(param) {
   return param * param;
}
// Benutze Funktion
function1(2);
```

```
// Deklariere Funktion und speichere Referenz auf Funktion in Variable
let functionVar1 = function(param) {
   return param * param;
}
console.log(typeof functionVar1); // Ausgabe: function
// Benutze Funktion
functionVar1(2);                  // Ausgabe: 4
```

Prozedurale Elemente IV - Funktionen

Möglichkeiten von Funktionen:

- können als Parameter verwendet werden
- können Standard-Werte für Parameter besitzen
- können einen Rest-Parameter besitzen
- haben als Standard-Rückgabewert „undefined“

```
// Deklariere Funktion und speichere Referenz auf Funktion in Variable
let functionVar1 = function(param) {
   return param * param;
}

// Checks if an function always returns the same with the same param
function isDeterministic(func, param) {
         let res1 = func(param);
         let res2 = func(param);
         if(res1==res2) {
                  return true;
         }
         return false;
}
console.log(isDeterministic(functionVar1,4));
```

Prozedurale Elemente IV - Funktionen

Möglichkeiten von Funktionen:

- können als Parameter verwendet werden
- **können Standard-Werte für Parameter besitzen**
- **können einen Rest-Parameter besitzen**
- haben als Standard-Rückgabewert „undefined“

```
// Function with default value
function defaultValue(param=2) {
        return param * param;
}

console.log(defaultValue());                // Ausgabe: 4
console.log(defaultValue(4));               // Ausgabe: 16
```

```
function addAll(value1, ...valuesN) {
        let val = value1;
        for(var i=0; i < valuesN.length;i++) {
                val += valuesN[i];
        }
        return val;
}

console.log(addAll(1,2));           // Ausgabe: 3
console.log(addAll(1,2,3,4,5,6));  // Ausgabe: 21
```

JavaScript

1. Kontext und Motivation
2. Eigenschaften und Einbindung
3. Prozedurale Sprachelemente
4. **Objektorientierte Sprachelemente**
5. (a)synchrone Funktionen
6. Browser-Objekte
7. Browser APIs
8. Darüber hinaus
9. Projekt

Objektorientierung I – JavaScriptOO

Definition: **In JavaScript sind Objekte eine strukturierte Sammlung von Attributen. Attribute können beliebige Typen an Daten halten. JavaScript Objekte werden nicht (notwendigerweise) aus Klassen erzeugt.**

Möglichkeiten von Objekten:

- können eine Menge primitiver Attribute halten
- können eine Menge von Objekten halten
- können eine Menge von Funktionsreferenzen halten
- können als Vorlage für andere Objekte dienen (templateing)

Eigenschaften:

- Attribute sind immer public
- Zugriff auf Attribute mit this oder der Objektreferenz

```
// Objekt deklarieren
let obj = {
   // Objekt mit einer Eigenschaft ausstatten
   eigenschaft : 'grün',
   // Objekt mit einer Methode ausstatten
   methode : function() {
         console.log(this.eigenschaft);     // Ausgabe: grün
         console.log(obj.eigenschaft);                // Ausgabe: grün
   }
}
obj.methode();
```

Objektorientierung I – JavaScriptOO

Definition: **JavaScript Objekte können zur Laufzeit Attribute und Funktionen zugewiesen bekommen.**

```
// Objekt deklarieren
let obj = new Object();
// Objekt mit einer Eigenschaft ausstatten
obj.eigenschaft = 'grün';
// Objekt mit einer Methode ausstatten
obj.methode = function() {
   console.log(this.eigenschaft);          // Ausgabe: grün
   console.log(obj.eigenschaft);           // Ausgabe: grün
}

obj.methode();
```

```
// Objekt deklarieren (hier alternative Schreibweise)
let obj = {}
// Objekt mit einer Eigenschaft ausstatten
obj.eigenschaft = 'grün';
// Objekt mit einer Methode ausstatten
obj.methode = function() {
   console.log(this.eigenschaft);          // Ausgabe: grün
   console.log(obj.eigenschaft);           // Ausgabe: grün
}

obj.methode();
```

Objektorientierung III – Klassen

Definition: **JavaScript Klassen bilden Prototyp-Objekte.**

```
class Klassenname {
   constructor(parameter) { }
   methodenName(parameter) { }
}
```

Eigenschaften:

- JavaScript-Klassen sind Objekte
- Methoden werden ohne Schlüsselwort definiert
- constructor() ist die Methode, die als Konstruktor-Methode verwendet wird
- Attribute werden im Konstruktor deklariert
- Zugriff auf Attribute in Methoden ausschließlich mit this.
- Objekte aus Klassen sind zur Laufzeit normale Objekte mit allen Fähigkeiten
- Klassen unterstützen einfache Vererbung
- Klassen können erst nach ihrer Deklaration verwendet werden (kein hoisting)
- Eingeführt in ES6. Einfacherer Syntax als das zuvor gebräuchliche prototyping

Weitere Informationen:
https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Klassen

Objektorientierung III – Klassen

```
class MeineKlasse {
   //Hier gibt es keine Attribut-Deklarationen

   constructor(param1) {
         // Hier kommen Attributdeklarationen
         this.attribut1 = param1;
         this.attribut2 = undefined;
   }

   methode1() {
         return this.attribut1;
   }
}
meineVar = new MeineKlasse(20);

console.log(meineVar.attribut1);   // Ausgabe: 20
console.log(meineVar.methode1());  // Ausgabe: 20

meineVar.attribut3 = 42;
console.log(meineVar.attribut3);   // Ausgabe: 42
```

```
class MeineKlasse {
   let attribut1;          // Syntax-Fehler!

   …
}
```

Objektorientierung IV – getter / setter

Definition: **Mit den Schlüsselwörtern get und set können Getter und Setter für Attribute definiert werden.**

```
class Klassenname {
   set attributname(parameter) { … }
   get attributname() { … }
}
```

Eigenschaften:

- Getter und Setter müssen immer zusammen deklariert werden
- Werden beim Zugriff auf ein Attribut angesprochen
- Können nicht als Funktion angesprochen werden
- Das Attribut muss einen anderen Namen haben, als die Getter und Setter

Objektorientierung IV – getter / setter

```
// Klasse anlegen
class MeineKlasse {

   constructor(param1) {
         // Hier kommen Attributdeklarationen
         this.attribut1 = param1;          // Ruft den Setter auf
         this.attribut2 = 10;              // Legt das Attribut an
   }

   get attribut1() {
         console.log('get attribut1');
         return this.a1;
   }

   set attribut1(param1) {
         console.log('set attribut1');
         this.a1 = param1;
   }
}
meineVar = new MeineKlasse(20);           // Ausgabe: set attribute1

console.log(meineVar.attribut1);          // Ausgaben: get attribut1, 20
console.log(meineVar.attribut1());        // attribut1 is not a function
console.log(meineVar.a1);                 // Ausgabe: 20
console.log(meineVar.attribut2);          // Ausgabe: 10
```

Objektorientierung V - Vererbung

```
class Klassenname extends KlassennameElternklasse { … }
```

Eigenschaften:

- Einfache-Vererbung, keine Mehrfachvererbung
- Es werden alle Methoden der Elternklasse geerbt
- Attribute werden über den Eltern-Konstruktor geerbt
- Eltern-Konstruktor muss verwendet werden, sobald this in Elternklasse verwendet wurde und ein Konstruktor in der abgeleiteten Klasse genutzt wird
- Eltern-Konstruktor wird automatisch verwendet, wenn kein abgeleiteter erstellt wird.
- Methoden können Methoden der Elternklasse überdecken
- Wird ein Getter überdeckt, muss auch der Setter überdeckt werden

Objektorientierung V – Vererbung

```
// Klasse anlegen
class MeineKlasse {
        … // Wie zuvor
}

class MeineKlasse2 extends MeineKlasse {

   constructor(param1) {
        super(param1);                              //Zwingend
        this.attribut1 = 55;
   }

   get attribut1() {
        return this.a1;
   }

   set attribut1(param1) {
        this.a1 = param1 * 2;
   }
}

meineVar = new MeineKlasse2(20);

console.log(meineVar.attribut1);                    // Ausgabe: 110
console.log(meineVar.attribut2);                    // Ausgabe: 10
```

Objektorientierung VII - Attributiteration

```
for(var i in object) { … }
```

Attributiteration:

- Iteriert über alle Attribute eines Objekts
- liefert den Attributnamen

```
// Objekt deklarieren
let obj = {
   // Objekt mit einer Eigenschaft ausstatten
   eigenschaft : 'grün',
   // Objekt mit einer Methode ausstatten
   methode : function() {
         console.log(this.eigenschaft);
   }
}
for(var i in obj) {
         console.log(i + ' = ' + obj[i]);
}

Ausgabe:
eigenschaft = grün
methode = function() { console.log(this.eigenschaft); }
```

Weitere Informationen:
https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/for...in

Objektorientierung VIII - Arrays

Definition: **JavaScript Arrays sind eine Sammlung von Eigenschaften.**

```
let arrayname = [];
arrayname[index] = Wert;
Wert = arrayname[index];
```

Eigenschaften:

- haben keine fest vorgegebene Größe
- können beliebige Werte enthalten (auch gemischt)
- Array**Elemente** können über einen laufenden Index angesprochen werden
- sind Integer-indiziert
- sind Objekte
  - haben Arrray**Attribute** und haben Funktionen

```
let myArray = ['w0','w1'];          // Neues Array mit 2 Elementen
myArray[3] = ‚w3‘,                   // Hinzufügen eines Elements
console.log(myArray.length);        // Auslesen des Attributs
„length“
let myArray2 = [];                  // Neues Array ohne Elemente
let myArray3 = new Array(2);        // Neues Array mit 2 „undefined“
Elementen
```

Weitere Informationen:
https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Array

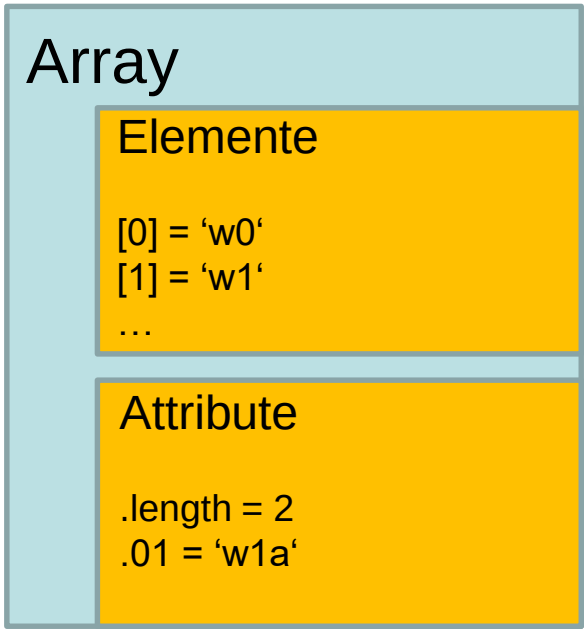
Objektorientierung VIII - Arrays

JavaScript versucht Indizes automatisch in integer zu konvertieren.

```
let myArray = ['w0','w1'];          // Neues Array mit 2 Elementen
myArray[3] = ‚w3‘,                   // Hinzufügen eines Elements
console.log(myArray.length);        // Auslesen des Attributs „length“

console.log(myArray);     // Ausgabe: Array ["w0","w1",<1 empty slot>,"w3"]

myArray['3'] = 'w3a';     // Automatische Konvertierung zu integer
console.log(myArray.length);        // Ausgabe: Array [ "w0", "w1", <1 empty
slot>, "w3a" ]

myArray['01'] = 'w1a';    // Konvertierung zu integer klappt nicht
console.log(myArray);     // Ausgabe: Array["w0","w1",<1 empty slot>,"w3a"]
```

Objektorientierung VIII - Arrays

- **ArrayElemente**
  - Primitive, Objekte, Funktionen,…
  - Ansprechen über numerischen Index
  - Nicht assoziativ
- **ArrayAttribute**
  - Primitive, Objekte, Funktionen,…
  - Ansprechen über String-“Index“
  - Assoziativ



=> Wenn ein assoziativer Speicher gebraucht wird Object() benutzen kein Array!

```
let myArray = ['w0','w1'];          // Neues Array mit 2 Elementen
myArray[3] = ,w3‘,                  // Hinzufügen eines Elements

myArray['test'] = 'TestWert';       // Schreiben des Attributs
console.log(myArray['test']);       // Zugriff auf Attribut nicht Element
```

Objektorientierung IX - ArrayIteration

```
for(var i=0; i < array.length; i++) { … }
for(i in array) { … }
for(elem of array) { … }
myArray.forEach(function(elem) { … });
```

Schleifen:

for	Läuft über den Index
for…in	Läuft über alle Elemente (ohne leere) und Attribute
for…of	Läuft über alle Elemente (mit leere) //ES6
forEach(func)	Übergabe aller Elemente (ohne leere) an eine Funktion.

```
for(var elem of myArray) {
        console.log(elem);
}

Ausgabe:

w0
w1
undefined
w3a
```

```
for(var i in myArray) {
   console.log(i + ' = ' +
myArray[i]);
}

Ausgabe:

0 = w0
1 = w1
3 = w3a
01 = w1a
test = TestWert
```

Weitere Informationen:
https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/for...of

Objektorientierung X – Weitere Konzepte

Weitere nicht im Detail behandelte Konzepte:

Iteratoren	Iteratoren erzeugen (genutzt in der for…of)
Generator-Funktionen	Generieren von Iterations-Ergebnissen
Arrow-Funktionen	Alternative Funktions-Deklaration
Spreading	Verteilen von Array- und Objekt-Inhalten
Typisierte Arrays	Typsichere Arrays
Sets und Maps	Set und Map Konstrukte
Destructuring	Aufteilung von Strukturen auf Variablen

Destructureing:

```
let myArray = ['w0','w1', 'w3'];

let a,b,c;
[a,b,c] = myArray;

console.log('a: '+a+' b: '+b+' c:
'+c);

Ausgabe:

a: w0 b: w1 c: w3
```

Spreading:

```
let myArray = ['w0','w1', 'w3'];

function spreadingTest(a,b,c) {
   console.log('a: '+a+' b: '+b+' c:
'+c);
}
spreadingTest(...myArray);

Ausgabe:

a: w0 b: w1 c: w3
```

Weitere Informationen: https://developer.mozilla.org/de/docs/Web/JavaScript

Objektorientierung XI – Standard-Objekte

ECMA-Skript definiert ein paar Standard-Objekte mit Funktionalitäten:

`Math`	Mathematische Operationen
`Date`	Datums Angaben und Operationen darauf
`Error`	Für Fehlerbehandlung mit try…catch
`JSON`	
…	

Weitere Informationen: https://developer.mozilla.org/de/docs/Web/JavaScript

JSON

Definition: **JSON (JavaScriptObjectNotation) ist ein Format für den Datenaustausch zwischen Anwendungen. Es basiert auf einer JavaScript konformen Darstellung der Daten.**

Eigenschaften:

- JSON Dokumente können in JavaScript Objekte transformiert werden
- kann mit der JavaScript Funktion eval() interpretiert werden
- ist kompakter als XML
- Es gibt Interpreter für viele Sprachen

Datentypen:

- Alle JavaScript Datentypen, aber keine Referenzen und undefined

Unterschiede in der Notation:

- Namen von Eigenschaften in doppelte Anführungszeichen
- Strings immer in doppelte Anführungszeichen
- Keine führenden Nullen

JSON

ECMA-Skript bietet mit dem Objekt JSON Methoden um aus Strings Objekte und aus Objekten JSON-Strings zu machen.

```
{
   "attribut1" : 42,
   "array1" : ["w1","w2","w3","w4"],
   „object1" : {"at1" : 1,"at2" : 2}
}
```

```
let jsonStr="{\"attribut1\":42, \"array1\":[\"w1\",\"w2\",\"w3\",\"w4\"]}";

jsonObj = JSON.parse(jsonStr);
for(var attr in jsonObj) {
   console.log(attr);
   if(typeof jsonObj[attr] == "object") {
         for(var i in jsonObj[attr]) {
                  console.log("> " + jsonObj[attr][i]);
         }
   }
}
Ausgabe:
attribut1
array1
> w1
> w2
> w3
> w4
```

JSON

```
let obj = {
   attr1 : 42,
   array1 : [1,2,3,4],
   object1 : {sub1 : "sub1", sub2 : "sub2"}
}

let newJsonStr = JSON.stringify(obj);
console.log(newJsonStr);

Ausgabe:
{"attr1":42,"array1":[1,2,3,4],"object1":{"sub1":"sub1","sub2":"sub2"}}
```

JavaScript

1. Kontext und Motivation
2. Eigenschaften und Einbindung
3. Prozedurale Sprachelemente
4. Objektorientierte Sprachelemente
5. **(a)synchrone Funktionen**
6. Browser-Objekte
7. Browser APIs
8. Darüber hinaus
9. Projekt

(a)synchrone Funktionen

Definition: **Synchron ausgeführte Funktionen, werden strikt in der Reihenfolge ihres Aufrufs ausgeführt. Asynchrone Funktionen lassen die Ausführung anderer Funktionen während ihrer eigene Ausführung zu.**

Wichtige Eigenschaften von JavaScript zu Synchronizität:

- JavaScript wird üblicherweise als single-thread ausgeführt
- Asynchron != Nebenläufig (Javascript-Asynchron ~ Queue)
- Viele eingebaute Funktionen arbeiten asynchron

Synchron:

- Funktionen werden nacheinander ausgeführt
- Ergebnis ist bei Rückkehr aus der Funktion bekannt

Asynchron:

- Funktionen werden nebeneinander ausgeführt
- Ergebnis ist bei Rückkehr aus der Funktion wahrscheinlich nicht bekannt

(a)synchrone Funktionen

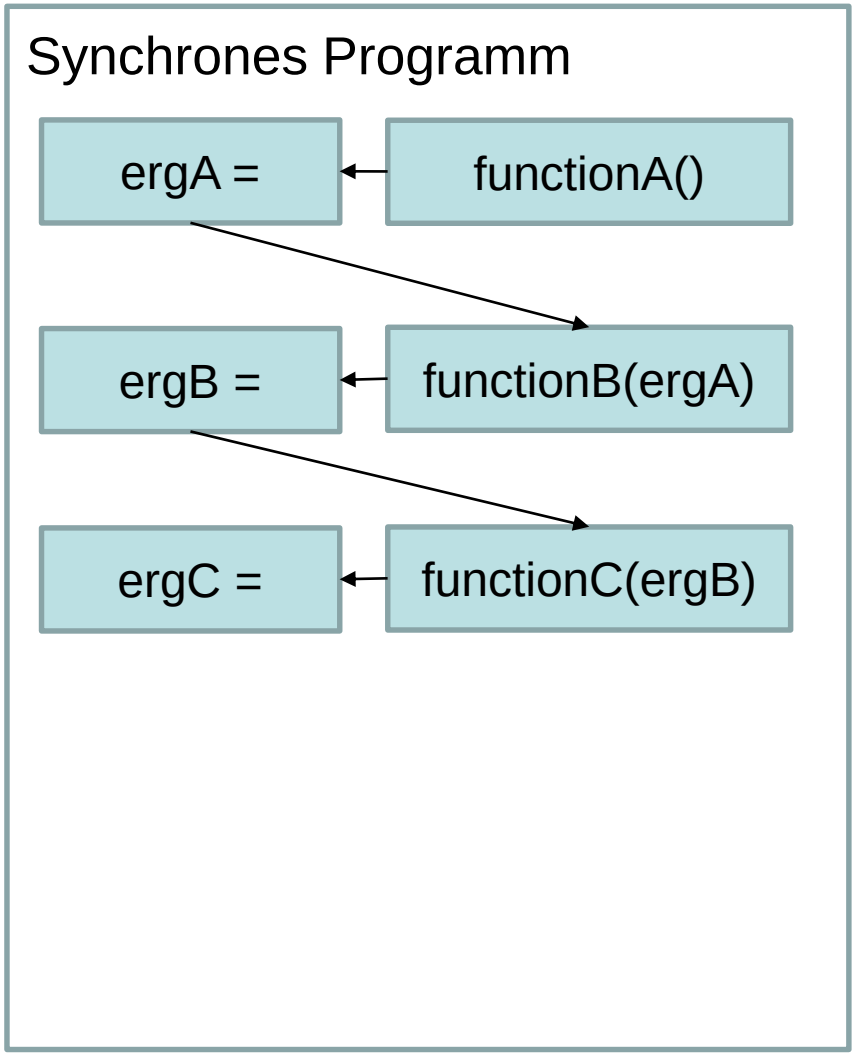
Vorteile:

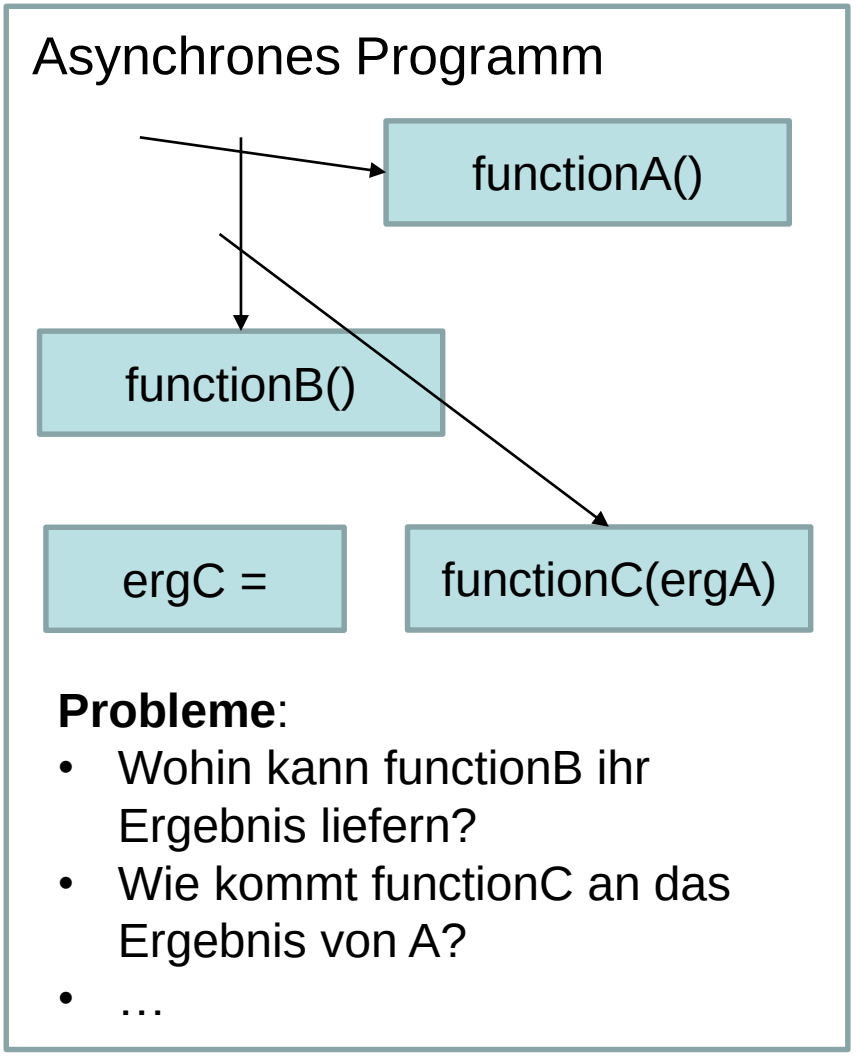
- Vermeiden eventuell langer Wartezeiten (laden von Ressourcen)
- Effizientes abarbeiten der Funktionen
- Bessere UserExperience, kein „einfrieren“ der Seite
- Ergebnisse werden geliefert, sobald sie verfügbar sind
- Zeitgesteuerte Funktionsaufrufe sind nur asynchron möglich

Nachteile:

- Programmablauf ist schwerer nachzuvollziehen
- Programmablauf ist nicht immer gleich
- Unübersichtlicherer Programmcode
- Wie könne asynchrone Funktionen Ergebnisse liefern?
- Gleichzeitiger Zugriff auf gemeinsame Ressourcen problematisch

(a)synchrone Funktionen I - Allgemein





(a)synchrone Funktionen II - Callbacks

Definition: **Der Callback-Mechanismus für asynchrone Funktionen besteht darin, der asynchron ausgeführten Funktion eine Callback-Methode als Parameter mitzugeben, diese wird durch die asynchrone Funktion aufgerufen.**

```
function asyncFunc(callbackFunc);
callbackFunc = function(result) { … };
```

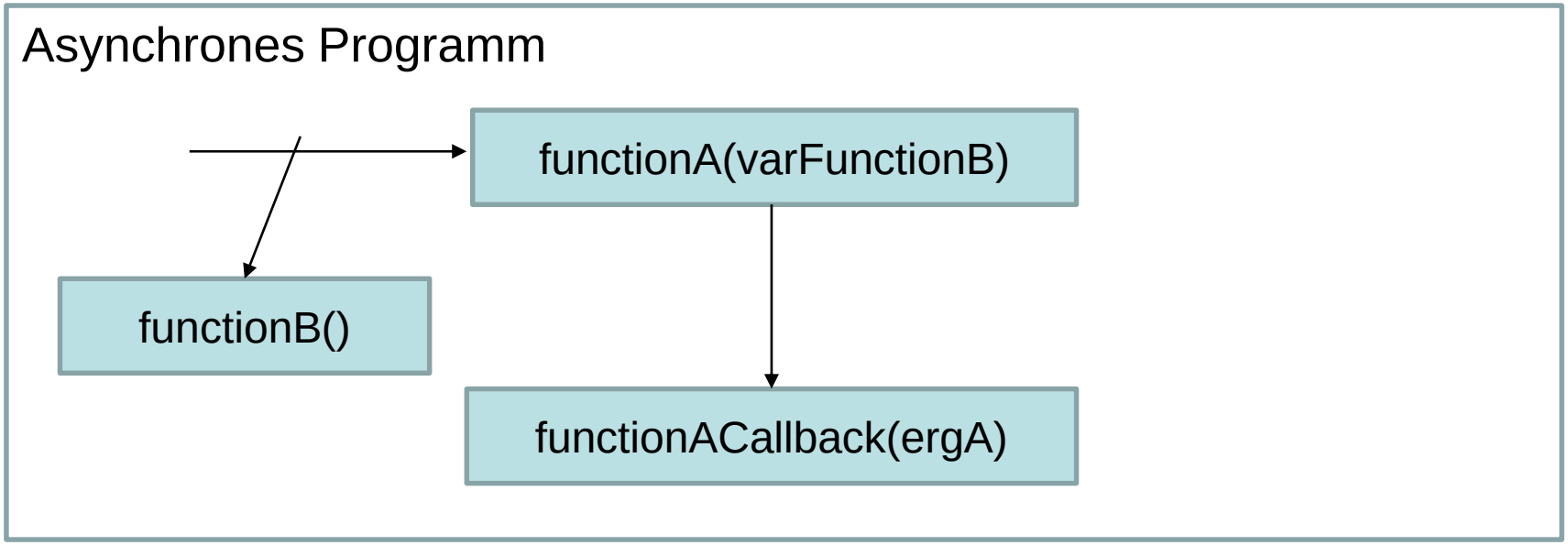
Eigenschaften:

- Die asynchrone Funktion hat mindestens den Parameter der Callback-Funktion
- Einfaches Konzept

Nachteile:

- Fehlerbehandlung schwierig (kein try-catch)

(a)synchrone Funktionen II – Callbacks




```
function printAtTime() {
   console.log("Time is over!");
}

setTimeout(printAtTime,100);

for(var i=0; i < 10000; i++) {
   console.log("I'am dooing hard work");
}
```

Frage: Wann wird „Time is over!“ ausgegeben?

(a)synchrone Funktionen III - Promises

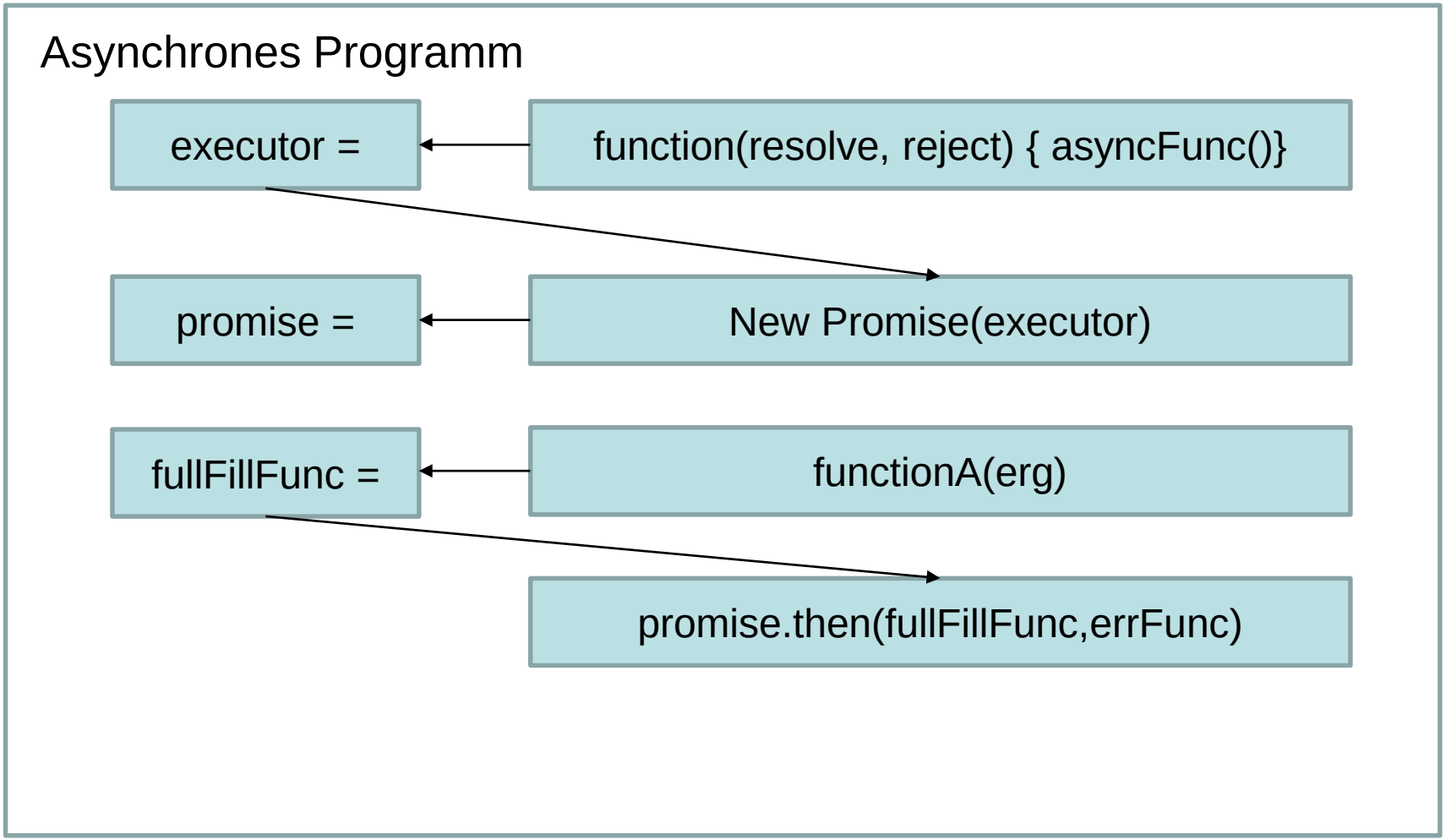
Definition: **Der Prommise-Mechanismus für asynchrone Funktionen liefert ein Promise-Objekt als sofortige Antwort, welches nach Beendigung eine Funktion aufruft.**

```
let executor = function(resolve,reject) { asyncFunc()}
let promise = new Promise(executor);
let fullFillFunc = function(result);
promise.then(fullFillFunc).catch(errorFunc);
```

Eigenschaften:

- Promise-Objekt kennt Ausführungsstatus
- Executor-Funktion wird sofort ausgeführt
- Executor-Funktion ruft eine asynchrone Funktion auf
- Asynchrone Funktion ruft resolve oder reject-Funktion auf
- Resolve-Funktion wird mit then() registriert (1 Parameter)
- Reject-Funktion wird mit then() registriert (2 Parameter)
- Werden Funktionen erst registriert, nachdem die asynchrone Funktion fertig ist, werden sie direkt ausgeführt.

(a)synchrone Funktionen III – Promises



Weitre Informationen: http://wiki.selfhtml.org/wiki/JavaScript/Promise

(a)synchrone Funktionen III – Promises

```
// Promise - alternative
let executor = function(resolve,reject){
        setTimeout(function() {
                console.log("I do something that needs time...");
                resolve("done!");
        },2000);
}

let fullFillFunc = function(erg) { console.log(erg)};

var promise = new Promise(executor);
promise.then(fullFillfunc);
console.log("This should be printed before done!");
```

```
var promise = new Promise(
        function(resolve,reject){
                setTimeout(function() {
                        console.log("I do something that needs
time...");
                        resolve("done!");
                },2000);
        }
);

promise.then(function(erg) { console.log(erg)});
console.log("This should be printed before done!");
```

JavaScript

1. Kontext und Motivation
2. Eigenschaften und Einbindung
3. Prozedurale Sprachelemente
4. Objektorientierte Sprachelemente
5. (a)synchrone Funktionen
6. **Browser-Objekte**
7. Browser APIs
8. Darüber hinaus
9. Projekt

Browser-Objekte

JavaScript nimmt die Browserumgebung über Objekte war und tritt über diese Objekte in Wechselwirkung mit seiner Umgebung

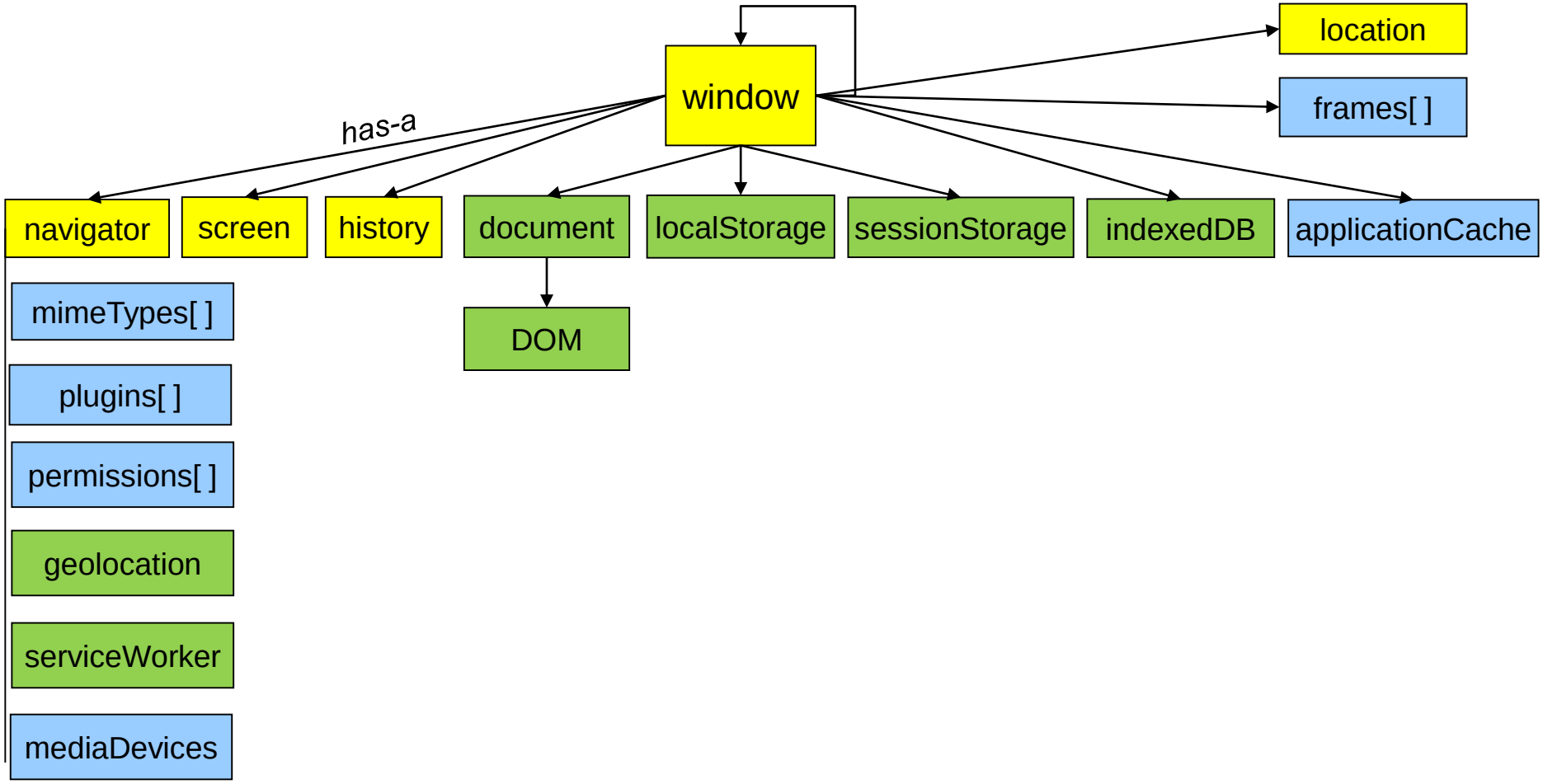
Browser-Objekte:

- beschreiben ***Eigenschaften und Funktionalitäten***
- sind ***hierarchisch*** angeordnet
- sind nicht durchgehend standardisiert (aber weit verbreitet)

Beispiele:

- *window* – aktuelles Browserfenster ist Wurzelobjekt
- *window* **enthält (has-a-Beziehung)** :
  - *navigator* (Browsereigenschaften)
  - *history* (Aufzeichnungspfad)
  - *document* (Repräsentation das Dokuments)

Browser-Objekte I - Objektmodell



Weitere Informationen screen: https://wiki.selfhtml.org/wiki/JavaScript/Screen

Browser-Objekte II - window

Das Objekt window steht in der Objekthierarchie an oberster Ebene. Seine Eigenschaften sind Objekte, die das Fenster und das Dokument im Fenster beschreiben und festlegen. Außerdem gibt es hier Zugriff auf Speicher-APIs.

Attribute (Auswahl):

`outerHeight`	Höhe des Fensters (inklusive Menübar)
`outerWidth`	Breite des Fensters (inklusive Menüs)
`opener`	Referenz auf Fenster, welches das aktuelle Fenster geöffnet hat
`pageXOffset`	Anzahl der Pixel, welche die Seite gescrollt wurde

Methoden (Auswahl):

`alert()`	Meldungsfenster wird angezeigt
`blur()`	Browserfenster wird deaktiviert; in den Hintergrund verschoben
`close()`	Schließen des Browserfensters
`scrollTo()`	Scrollen auf der Seite
`setTimeout()`	Eine Funktion nach einer bestimmten Zeit ausführen
`setIntervall()`	Setz eine Funktion, die in einem Intervall ausgeführt wird

Weitere Informationen: https://www.w3schools.com/jsref/obj_window.asp

Browser-Objekte II - window

```
window.open(URL,NAME,SPECS);
window.setTimeout(FUNCTION,MILLISECONDS);
```

window.open():

URL	URL der Seite die anzeigt werden soll
NAME	Bezeichner für das Fenster
SPECS	Einstellungen zur Anzeige

window.setTimeout():

FUNCTION	Funktion die ausgeführt werden soll
MILLISECONDS	Millisekunden, nach der die Funktion aufgerufen wird

Die Methode window.setTimeout() arbeitet **asynchron**.

```
let fenster1 = window.open("", "popup1", "width=200,height=100");
let fenster2 = window.open("","popup", "width=200,height=200");
fenster2.setTimeout("close()", 2000); // Zeitverzögerung 2000 ms
fenster1.setTimeout("close()", 5000); // Zeitverzögerung 5000 ms
```

Weitere Informationen: https://www.w3schools.com/jsref/met_win_open.asp

Browser-Objekte III - location

Das Objekt *location* erlaubt es, die URL des aktuellen Fensters auszulesen und zu bearbeiten.

Attribute (Auswahl):

`protocol`	Ausgabe des Internet-Protokolls, mit dem die aktuelle Webseite aufgerufen wurde (z.B. http, ftp, …)
`hostname`	Auslesen des Hostnamens, Domainnamens oder der IP-Adresse
`port`	Ausgabe der Portnummer des Servers
`pathname`	Auslesen der Pfadangabe des Dokuments
`search`	Ausgabe des Parameterstrings
`hash`	Ausgabe des Verweisankers
`href`	Auslesen der kompletten URL
`host`	Ausgabe von *hostname* und *port*

Funktionen (Auswahl):

`reload()`	Neu Laden der Seite

Weitere Informationen: https://www.w3schools.com/jsref/obj_location.asp

Browser-Objekte IV - navigator

Das Objekt *navigator* erlaubt es, die Eigenschaften des Browsers auszulesen.

Attribute (Auswahl):

`userAgent`	vollständige standardisierte Browserbezeichnung
`platform`	verwendete Computerplattform
`language`	Spracheinstellung des Clientcomputers
`onLine`	Gibt an, ob der Browser eine Internetverbindung hat
`cookieEnabled`	Gibt an, ob Cookies aktiviert sind
`plugins`	liefert Feld mit allen installierten Plugins (nur Mozilla)
`mimeTypes`	Array aller akzeptierter MIME-Typen (nur Mozilla)

Funktionen (Auswahl):

`javaEnabled()`	liefert true, wenn Java-Unterstützung vorhanden, sonst false

Weitere Informationen: https://www.w3schools.com/jsref/obj_navigator.asp

Browser-Objekte IV - navigator

```
navigator.onLine;
navigator.language;
```

```
if(navigator.onLine) {
        console.log("Sie sind mit dem Internet verbunden.");
} else {
        console.log("Sie sind nicht mit dem Internet verbunden.");
}
```

```
if(navigator.language=='de') {
        console.log("Hallo!");
} else if(navigator.language=='en') {
        console.log("Hello!");
}
```

Browser-Objekte V - history

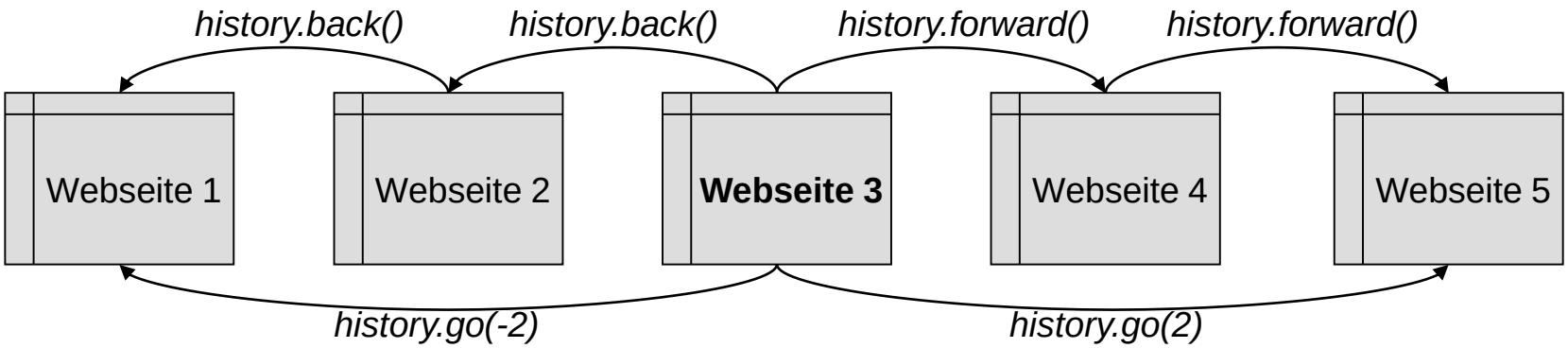
Das Objekt *history* ermöglicht vorwärts und rückwärts einen Zugriff auf die im aktuellen Browserfenster bereits aufgerufenen URLs.

Attribute (Auswahl):

`length`	Anzahl der History-Einträge im Browser

Methoden (Auswahl):

`back()`	Zugriff rückwärts: Laden des zuvor besuchten Dokuments
`forward()`	Zugriff vorwärts: Laden des nächsten Dokuments
`go(steps)`	Zugriff vorwärts (steps=positive Zahl) und rückwärts (steps=negative Ziel)



Browser-Objekte VI - document

Das Objekt document ist das Zugriffsobjekt für Inhalte eines Dokuments. Das document wird durch das DocumentObjectModel (DOM) beschrieben.

Methoden (Auswahl):

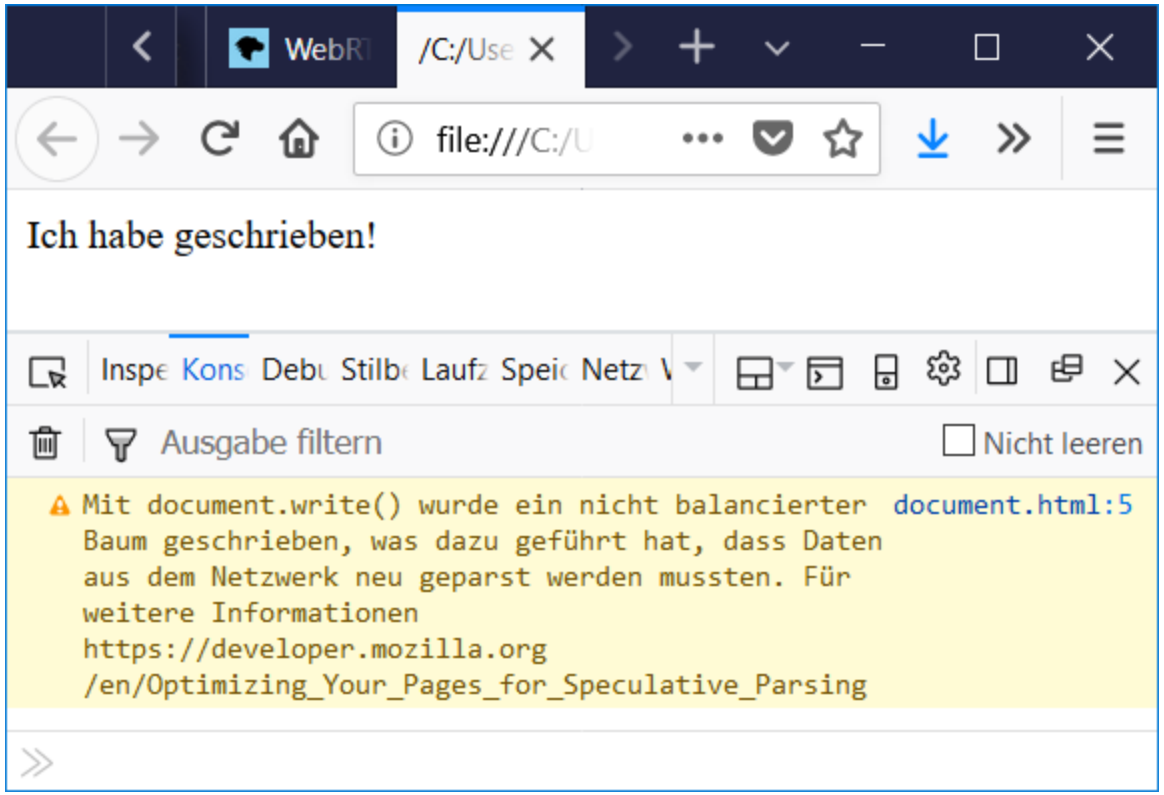
`open()`	öffnet einen output-Stream in das Dokument
`write(Text)`	schreibt Text in das HTML-Dokument
`close()`	schließt den output-Stream des Dokumentes

Weitere Informationen: In der nächsten Vorlesung

oder hier: https://www.w3schools.com/jsref/dom_obj_document.asp

Browser-Objekte VI - document

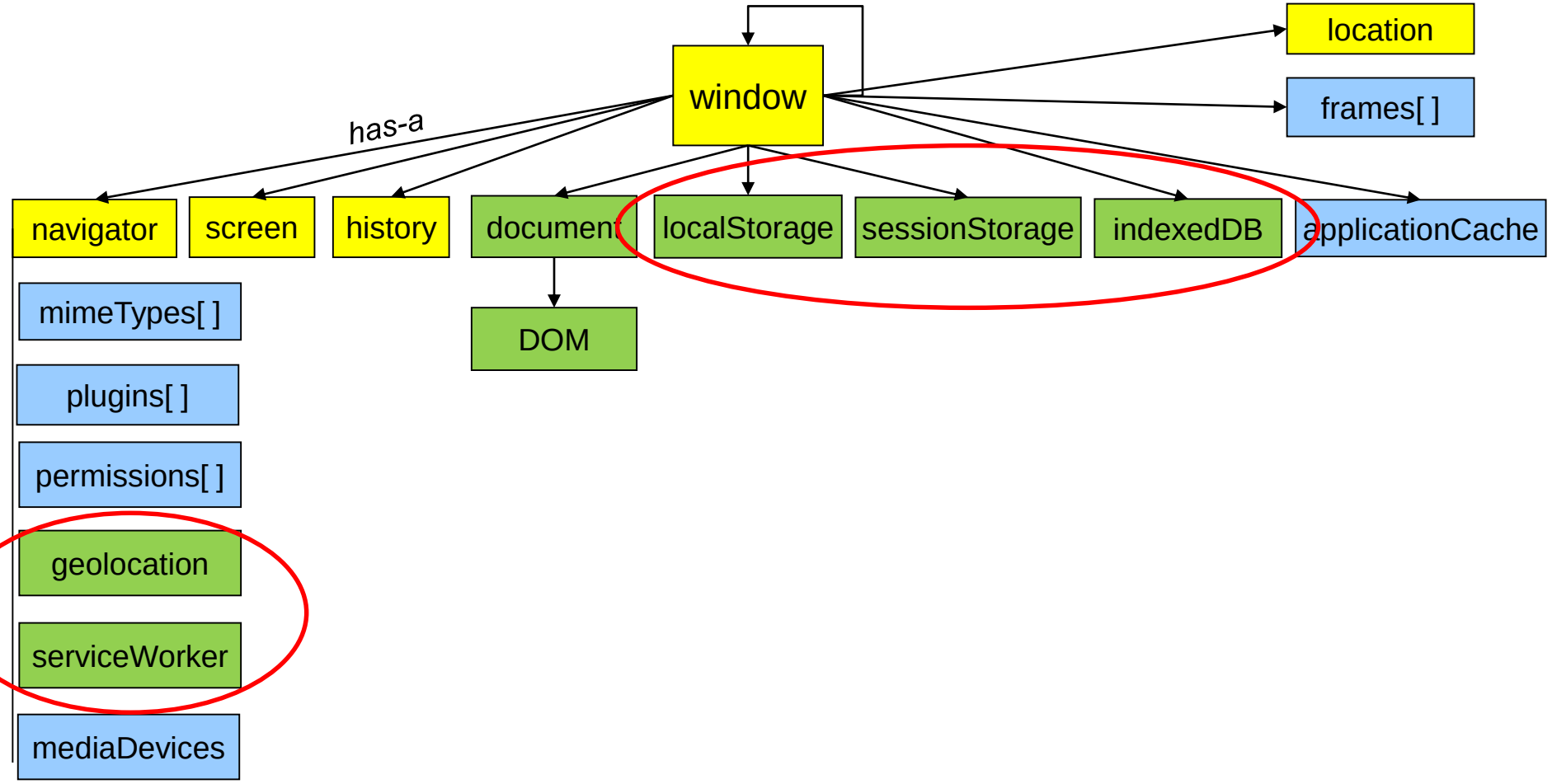
```
document.open();
document.write("Ich habe geschrieben!");
document.close();
```



JavaScript

1. Kontext und Motivation
2. Eigenschaften und Einbindung
3. Prozedurale Sprachelemente
4. Objektorientierte Sprachelemente
5. (a)synchrone Funktionen
6. Browser-Objekte
7. **Browser APIs**
8. Darüber hinaus
9. Projekt

Browser-APIs



Browser-APIs I – WebStorage

Definition: **WebStorage bezeichnet zwei standardisierte Schnittstellen für clientseitigen assoziativen Speicher.**

Eigenschaften:

- Speichern von umfangreicheren Datenmengen im Browser (mind. 5MB)
- Vom W3C standardisiert in den meisten aktuellen Browsern implementiert
- Zugriff per JavaScript
- Speicherung von Key-Value Paaren
- Key und Value sind Strings
- Tipp: Umfangreichere Werte als JSON speichern
- PerOrigin (für jede Weburl und das dabei verwendete Protokoll)
- Zwei Schnittstellen: localStorage und sessionStorage
  - APIs der Schnittstellen identisch
  - localStorage: Daten bleiben Dauerhaft erhalten, über Browser-Neustarts hinweg
  - sessionStorage: Daten werden beim Schließen des Tabs gelöscht

Weitere Informationen: https://wiki.selfhtml.org/wiki/JavaScript/Web_Storage

Browser-APIs I – WebStorage

```
localStorage.setItem(KEY,VALUE);
localStorage.getItem(KEY);
localStorage.removeItem(KEY);
localStorage.clear();
```

Methoden:

`setItem()`	Fügt ein neues Key-Value-Paar in den Speicher ein
`getItem()`	Holt den Wert zum angegebenen Key aus dem Speicher
`removeItem()`	Löscht den Wert zum angegebenen Key aus dem Speicher
`clear()`	Löscht den gesamten Speicher

Browser-APIs I – WebStorage

```
// Test if Storage is supported
if (typeof(Storage) !== "undefined") {
   let visits = localStorage.getItem("visits");
   if(visits) {                              // Test if visits is not undefined
      visitsNo = parseInt(visits);           // parse because storage is string only
      visitsNo++;
      localStorage.setItem("visits",visitsNo);

      if(visitsNo>5) {
         localStorage.removeItem("visits");
      }
   } else {
      localStorage.setItem("visits",1);
   }
   console.log("This is your " + localStorage.getItem("visits") + " visit");
} else {
    console.log("Sorry! No Web Storage support..");
}
```

Browser-APIs II – IndexedDB

Definition: **Die IndexedDB API bietet Zugriff auf eine standardisierte clientseitige Datenbank.**

Eigenschaften:

- Speichern von umfangreicheren Datenmengen im Browser
- Vom W3C standardisiert in den meisten aktuellen Browsern implementiert
- Zugriff per JavaScript
- Objektorientierte Datenbank
- Transaktionsbasiert
- Durchsuchbar und filterbar
- Speichern von Dateien möglich
- PerOrigin (für jede Weburl und das dabei verwendete Protokoll)
- Asynchrone Nutzung

Weitere Informationen: https://developer.mozilla.org/de/docs/IndexedDB
https://developer.mozilla.org/de/docs/IndexedDB/IndexedDB_verwenden

Browser-APIs III – geolocation

Definition: **Die geolocation-API ermöglicht den Zugriff auf Standortdaten des Benutzers.**

Eigenschaften:

- Liefert die aktuelle Position des Benutzers als GPS-Koordinaten
- Verwendet, wenn vorhanden, die lokalisierungs-Hardware des Geräts
- Wenn keine Hardware vorhanden ist, wird ein webbasierter location-Service genutzt
- Liefert Informationen zur Genauigkeit
- Funktioniert nur über gesicherte Verbindungen (https)
- Der Benutzer kann der Verwendung seiner Lokalisierung wiedersprechen
- Vom W3C standardisiert in den meisten aktuellen Browsern implementiert
- Zugriff per JavaScript
- Asynchrone API

Weitere Informationen: https://www.w3schools.com/html/html5_geolocation.asp

Browser-APIs III – geolocation

```
navigator.getCurrentPosition(func, errorfunc);
```

navigator.geoloccation:

`getCurrentPosition(func)`	Aktuelle Position holen und func aufrufen
`watchPosition(func)`	Aufruf von func bei Positionsänderungen
`clearWatch()`	Positionsüberwachung beenden

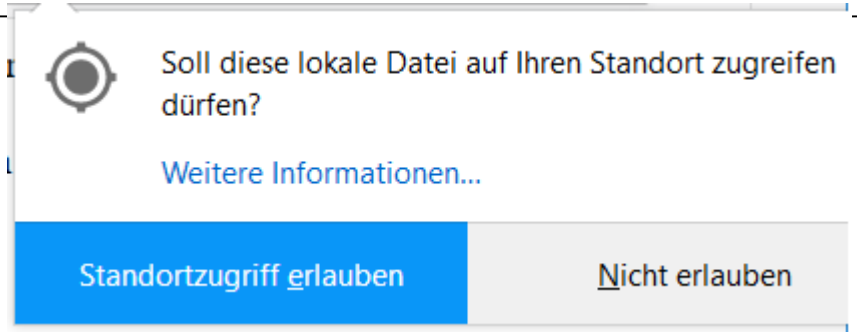
getCurrentPosition() ruft die Funktion func auf, sobald Koordinateninformationen vorliegen (oder ein Fehler auftritt) als Parameter wird der aufgerufenen Funktion ein geoposition-Objekt übergeben.

Attribute des geoposition-Objekts:

`timestamp`	Zeitstempel zu dem die Positionsangaben gelten
`coords.latitude`	Geographische Breite
`coords.longitude`	Geographische Länge
`coords.accuracy`	Genauig von Länge und Breite in Metern +/-
`coords.altitude`	Höhe über Normal-Null
`coords.heading`	Blickrichtung im Uhrzeigersinn von Norden
`coords.speed`	Bewegungsgeschwindigkeit

Browser-APIs III – geolocation

```
// Check if geolocation-api is available
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(positionRecivedHandler, errorHandler);
} else {
   console.log("Geolocation not supported");
}
function positionRecivedHandler(position) {
   console.log("Ihre Positionsinformationen:");
   for(let i in position.coords) {
        console.log(i + " = " + position.coords[i]);
   }
}
function errorHandler(error) {
   console.log("Keine geolocation Daten vorhanden.");
}
```




```
Ihre Positionsinformationen:
latitude = 53.548038
longitude = 10.0176987
altitude = 0
accuracy = 84198
altitudeAccuracy = 0
heading = NaN
speed = NaN
```

Browser-APIs IV – WebWorker

Definition: **Die WebWorker API ermöglicht es JavaScripte im Hintergrund auszuführen, selbst dann, wenn eine Webapplikation nicht angezeigt wird.**

Eigenschaften:

- Skripte können im Hintergrund ausgeführt werden (Nebenläufige Ausführung)
- Zwei Varianten: Dedicated Workers und Shared Workers
- WebWorker Skripte haben eingeschränkten Zugriff auf Objekte außerhalb
- Nur Zugriff auf: navigator-Objekt, location-Objekt (lesend), Anwendungscache
- Ausführung komplexer Berechnungen ohne die Seite zu beeinflussen
- Datenaustausch zwischen Seite und Worker über Methoden
- Datenaustausch immer mit Datenkopie
- Standardisiert vom W3C
- PerOrigin-Policy WebWorker werden nur von der gleichen Adresse ausgeführt
- Keine generelle Ausführung von WebWorkern aus dem fileSystem

Weitere Informationen: https://www.html5rocks.com/de/tutorials/workers/basics/
https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

Browser-APIs IV – WebWorker

```
worker = new Worker(SCRIPT-URL);
```

Worker-Objekt (im aufrufenden Skript):

`Worker(SCRIPT-URL)`	Ausführen des angegebenen Skripts in neuem Thread
`postMeassage(obj)`	Senden einer Nachricht an den Worker (Kopie)
`onmessage`	Attribut das eine Funktion hält, die ausgeführt wird, wen der **Worker** eine Nachricht sendet
`onerror`	Attribut für eine Funktion im Fehlerfall
`terminate()`	Abbrechen des Worker-Skripts

Worker-Objekt (im Worker):

`Worker(SCRIPT-URL)`	Unter-Worker erstellen
`postMessage(obj)`	Nachricht an den Aufrufer senden
`onmessage`	Attribut das eine Funktion hält, die ausgeführt wird, wen der **Aufrufer** eine Nachricht sendet.

`importScripts(…URLS)` Hinzuladen von (beliebig vielen) Script-Dateien

Weitere Informationen zu verfügbaren Objekten und Methoden:
https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Functions_and_classes_available_to_workers

Browser-APIs IV – WebWorker

worker.js (Seiten-Datei / Aufrufer)

```
// Call worker script
var worker = new Worker("webworker-
worker.js");

// Called when reciving message from
worker
worker.onmessage = function(e) {
   console.log(e.data);
};
// Called on error
worker.onerror = function(e) {
   console.log(e.message);
}

// Send messages to worker
worker.postMessage("message 1");
worker.postMessage("message 2");
//worker.terminate();
worker.postMessage("message 3");
worker.postMessage("message 4");
```

webworker-worker.js (WebWorker-Datei)

```
var mCounter=0;

onmessage = function(e) {
   mCounter++;
   if(messageCounter<=3) {
         console.log(e.data);
         // Reply to caller
         postMessage("reply
"+mCounter);
   } else {
         postMessage("no more
replies");
         //Close the worker
         close();
   }
}
```

Browser-APIs V – Service Worker

Definition: **Die Service Worker API ermöglicht es JavaScripte zu schreiben, die eine offline-Funktionalität der Webanwendung sicherstellen.**

Eigenschaften:

- Ermöglicht WebAnwendungen eine bessere offline-Funktionalität
- Basiert auf WebWorkern
  - Kein Zugriff auf Seiten-Elemente
- Funktionieren ähnlich wie Proxy-Server
- Ermöglichen das explizite cachen von Inhalten
- Ermöglichen es Netzwerkanfragen zu modifizieren
- Service Worker funktionieren nur in https-Umgebungen
- PerOrigin-Policy WebWorker werden nur von der gleichen Adresse ausgeführt
- Laufen im Hintergrund, auch wenn die Seite nicht angezeigt wird

Weitere Informationen: https://www.html5rocks.com/de/tutorials/workers/basics/
https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

Browser-APIs V – ServiceWorker

serviceworker.js (Seiten-Datei / Aufrufer)

```
if(navigator.serviceWorker) {
   let serviceworker = navigator.serviceWorker.register('/pwa/serviceworker-
worker.js');
   let succsessfullRegisterFunc = function(reg) {
         console.log("Erfolgreich registriert");
   }
   let errorRegisterFunc = function(err) {
         console.log("Fehler beim Registrieren: " + err);
   }
   serviceworker.then(succsessfullRegisterFunc,errorRegisterFunc);
} else {
         console.log("No ServiceWorker support");
}
```

Wichtig:

- Das serviceworker-worker Skript muss sich im root-Verzeichnis befinden
- Pfadangaben müssen absolut sein
- Pfadangaben müssen im ServiceWorker und auf der Seite identisch sein

Browser-APIs V – ServiceWorker

Serviceworker-worker.js (Seiten-Datei / Aufrufer)

```
// Called on install stage
this.addEventListener('install', function(event) {
  // Sicherstellen, das zuerst alle Datein im cache landen
  event.waitUntil(
    caches.open('v1').then(function(cache) {
      return cache.addAll([
        '/pwa/',                              // Root muss mit gelistet sein
        '/pwa/serviceworker.html',
        '/pwa/serviceworker.js',
        '/pwa/serviceworker-tc.jpg'
      ]);
    })
  );
});
```

Browser-APIs V – ServiceWorker

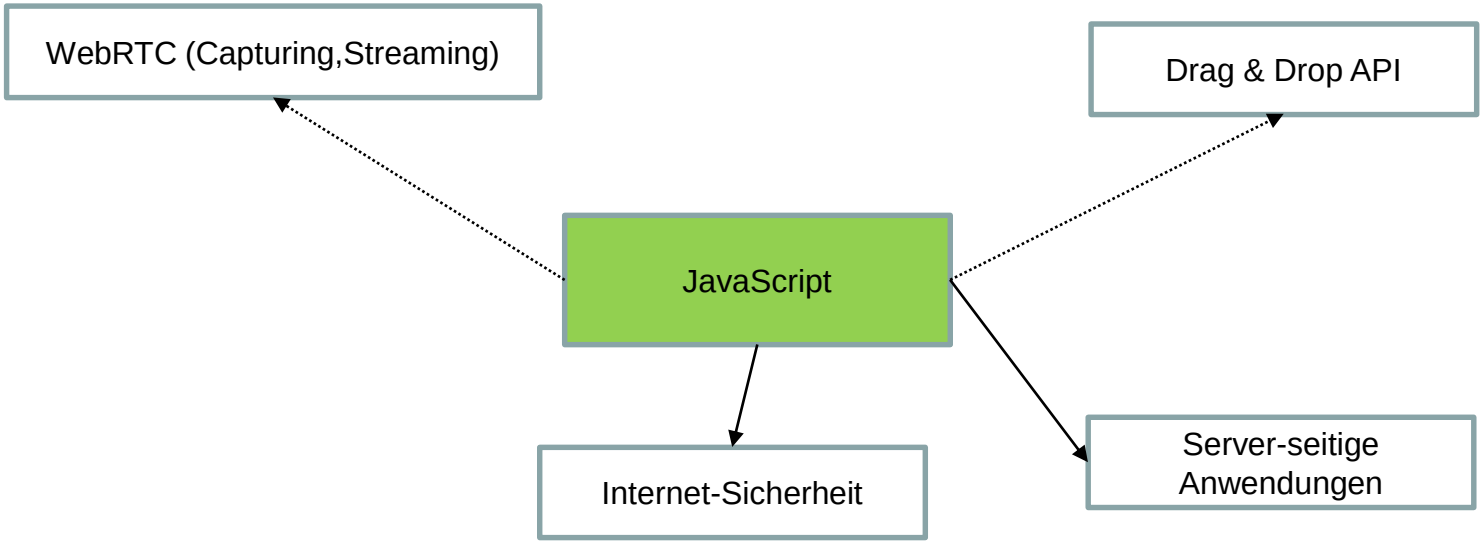
Serviceworker-worker.js (Seiten-Datei / Aufrufer)

```
// Wird aufgerufen wenn Dateien angefragt werden
this.addEventListener('fetch', function(evt) {
   console.log("Hole " + evt.request.url);

   evt.respondWith(                          // Responde with erwartet eine
asynchrone Funktion

         caches.match(evt.request).then(
            function(res) {                          // res ist Ergebnis von
caches.match()
            console.log("Resource >" + res.url + "< aus dem Cache geholt");
            return res;

         }).catch(function(err) {
            console.log("Resource >" + evt.request.url + "< nicht im Cache
gefunden");                  //return fetch(evt.request);
         })
   );
});
```

JavaScript

1. Kontext und Motivation
2. Eigenschaften und Einbindung
3. Prozedurale Sprachelemente
4. Objektorientierte Sprachelemente
5. (a)synchrone Funktionen
6. Browser-Objekte
7. Browser APIs
8. **Darüber hinaus**
9. Projekt

Darüber hinaus



Links:

Node.js - https://nodejs.org/en/
Drag&Drop API - https://www.w3schools.com/html/html5_draganddrop.asp
WebRTC: - https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API

Webanwendungen und Sicherheit

Sicherheitslücken in Browserimplementierungen können durch JavaScript-Programme ausgenutzt werden z.B.:

- unbemerktes Versenden von Emails
- Auslesen des Browserverlaufs
- Live-Verfolgungen von Internetsitzungen
- Erraten von EBAY-Passwörtern

- Anwender deaktivieren daher manchmal das „Ausführen von JavaScript-Code“ im Browser

- JavaScript-Anwendungen laufen im Browser: Sandbox (abgeriegelte Umgebung ohne Zugriff auf Dateien, Benutzerdaten, BS,..)

JavaScript

1. Kontext und Motivation
2. Eigenschaften und Einbindung
3. Prozedurale Sprachelemente
4. Objektorientierte Sprachelemente
5. (a)synchrone Funktionen
6. Browser-Objekte
7. Browser APIs
8. Darüber hinaus
9. **Projekt**

Anforderungen

Welche Anforderungen werden als nächstes bearbeitet?

TODO

- Mehrsprachen-Fähigkeit
- (lokales) Speichern von Artikeln
- Client-Position anzeigen
- Offline-Verwendung ermöglichen
- Inhaltsverzeichnisse
- Medien bearbeiten
- Formlareingaben in Seite einfügen
- Navigation über Tastaturkürzel
- Externe Inhalte einbinden
- Medien hochladen / runterladen
- Kommentare hochladen / runterladen
- Kommentare speichern
- Kommunikation untereinander

DONE

- Technologische Grundlagen erarbeiten
- Was ist eine Web-Anwendung?
- News darstellen
- Projekte vorstellen
- Aufgaben darstellen
- Formular für Kommentare
- Schickes Design für die Seite
- Mediendatein einbinden
- Animationen