

JavaScript

Objekte, Events und Asynchronität

Informatik – 3. Semester

Fehler

- `<script>` elements sind nicht standardmäßig async

Realität

- Eingefügte `<script>` Elemente sind standardmäßig async
 - ... in Firefox
 - Das Verhalten variiert von Browser zu browser

JavaScript

Objekte, Events und Asynchronität

Informatik – 3. Semester

1 **Objekte**

2 Events

3 Asynchronität

Objekte, Funktionen, Prototypen, Klassen und die DOM

- Java-Script Objekte sind ungeordnete Sammlungen von Werten, hier genannt „Properties“.
 - Objekte werden mit dem new-Operator erzeugt.
 - Properties können anschließend hinzugefügt werden.

```
// obj ist ein leeres Objekt
var obj = new Object();
// obj erhält neue Properties
obj.p = 2; // p mit Wert 2
obj.q = "x"; // q mit Wert "x"
// alternative Schreibweise
obj["p"] = 2;
obj["q"] = "x";
```

- Bei der Zuweisung werden nur Referenzen auf die Objekte übergeben.
- Es wird keine Kopie angelegt.

```
var a = new Object();  
var b = new Object();  
a.p = 2;  
b.q = 3;  
a = b;  
b.p = 4;  
document.write("a.p: " + a.p + " / a.q: " + a.q);
```

Ausgabe: a.p: 4 / a.q: 3

Funktionen werden mit `function` deklariert und können Parameter, lokale Variablen und Rückgabewerte haben. Innerhalb einer Funktion können auch andere Funktionen aufgerufen werden.

```
function abstand( x1, y1, x2, y2 ) {  
    var dx = x1 - x2;  
    var dy = y1 - y2;  
    return Math.sqrt(dx*dx + dy*dy);  
}  
var d = abstand(1,4,2,3); // d hat den Wert 1.4142...
```

Auch rekursive Funktionen sind erlaubt.

```
function fakultaet(x) {  
    if( x <= 1 ) {  
        return 1;  
    } else {  
        return x * fakultaet(x-1);  
    }  
}  
var d = fakultaet(5); // d hat den Wert 120
```

Als Parameter können auch Objekte übergeben werden.

```
function abstand( p1, p2 ) {  
    var dx = p1.x - p2.x;  
    var dy = p1.y - p2.y;  
    return Math.sqrt(dx*dx + dy*dy);  
}  
var p1 = new Object();  
p1.x = 1;  
p1.y = 4;  
var p2 = new Object();  
p2.x = 2;  
p2.y = 3;  
document.write( abstand(p1, p2) );
```

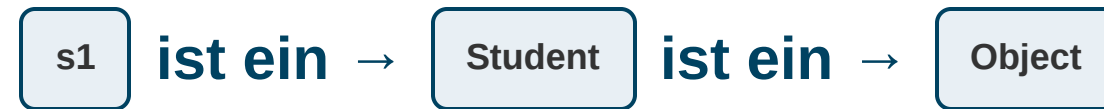
Mit Hilfe von Funktionen können Objekt-Konstruktoren definiert werden.

```
Punkt = function( x, y ) {  
    this.x = x; // "this" bezieht sich auf das neu  
    this.y = y; // erzeugte Objekt  
}  
var p1 = new Punkt(1, 4); // p1.x = 1 und p1.y = 4  
var p2 = new Punkt(2, 3); // p2.x = 2 und p2.y = 3
```

Ein Konstruktor kann auch Funktionen definieren:

```
Punkt = function ( x, y ) {  
  this.x = x; this.y = y;  
  this.abstand = function(p) {  
    var dx = x - p.x;  
    var dy = y - p.y;  
    return Math.sqrt(dx*dx + dy*dy);  
  }  
}  
  
var p1 = new Punkt(1, 4);  
var p2 = new Punkt(2, 3);  
var d = p1.abstand(p2); // d = 1.414213...
```

- Object ist der Grundprototyp in JavaScript



Alle JavaScript Objekte erben von einem Prototyp:

```
Student = function( firstName, lastName ) {  
    this.firstName = firstName;  
    this.lastName  = lastName;  
}  
  
var s1 = new Student("Lena", "Müller");  
Student.prototype.identificationNumber = 1234;  
  
console.log(s1.identificationNumber);
```

Über den Zugriff auf den Prototyp kann Vererbung realisiert werden.

```
ExchangeStudent = function(firstName, lastName, homeInstitution) {  
    Student.call(this, firstName, lastName);  
    this.homeInstitution = homeInstitution;  
}  
  
ExchangeStudent.prototype = new Student();  
var eS1 = new ExchangeStudent("Lin", "Vu", "East China University");  
  
console.log(eS1.firstName);
```

- Mit der Methode call kann man einen Konstruktor aufrufen.
- Hier den Konstruktor von Student

- Seit ES6 gibt es in JavaScript das Schlüsselwort `class`. Damit können Sie objektorientiert programmieren, wie es z.B. aus Java gewöhnt sind. Das neue Klassenmodell ist allerdings nur syntactic sugar, es wird keine neue Funktionalität ermöglicht. Implementiert ist weiterhin das prototypenbasierte Modell.
- Vererbung mit `extends` und Methodenaufrufe mit `super()` sind seit ES6 auch möglich.

```
class Tier {  
    constructor(name) {  
        this.name = name;  
    }  
    sprich() {  
        console.log(this.name + ' macht ein Geräusch.');    }  
}
```

Für Konstruktoren benutzt man das Schlüsselwort constructor.

```
class Katze extends Tier {  
    sprich() {  
        super.sprich();  
        console.log(this.name + ' miaut.');    }  
}
```

Für Vererbung benutzt man das Schlüsselwort extends.

- Knoten (nodes) sind Objekte im DOM-Baum und bieten eine bestimmte Schnittstelle, z.B.
 - parentNode
 - firstChild
 - nextSibling
- Einige Knotentypen können Kinder haben, andere sind immer Blätter.

Konstante	Wert	Beispiel
Node.ELEMENT_NODE	1	An Element node like <code><p></code> or <code><div></code> .
Node.TEXT_NODE	3	The actual Text inside an Element or Attr.
Node.PROCESSING_INSTRUCTION_NODE	7	A ProcessingInstruction of an XML document, such as <code><?xml-stylesheet ... ?></code> .
Node.COMMENT_NODE	8	A Comment node, such as <code><!-- ... --></code> .
Node.DOCUMENT_NODE	9	A Document node.
Node.DOCUMENT_TYPE_NODE	10	A DocumentType node, such as <code><!doctype html></code> .
Node.DOCUMENT_FRAGMENT_NODE	11	A DocumentFragment node.

Die DOM lässt sich wie alle Bäume traversieren

```
// DOM Traversal
function walkTheDOM(node, func) {
  func(node);
  node = node.firstChild;
  while (node) {
    walkTheDOM(node, func);
    node = node.nextElementSibling;
  }
}
```

Zu DOM gehören (u.a.) die folgenden Funktionen:

`document.getElementById( value )`

Liefert eine Referenz auf den Elementknoten, der ein id-Attribut mit dem angegebenen Wert besitzt. Dadurch ist es möglich, direkt auf einen Elementknoten zuzugreifen, ohne eine Kette von Referenzen angeben zu müssen. Da die Werte von id-Attributen innerhalb eines Dokuments eindeutig sein müssen, kann es immer nur einen passenden Elementknoten geben.

`document.getElementsByTagName( tagName )`

Liefert ein Array mit Referenzen auf alle Elementknoten mit dem angegebenen Tag-Namen (z.B. „form“, „input“, „h1“, „p“, usw.) in der Reihenfolge einer Pre-Order-Traversierung des Objektbaums.

Die zuvor besprochenen Selektoren werden wie folgt verwendet:

document.querySelector(Selector)

Liefert eine Referenz auf den ersten Elementknoten, der dem Selektor entspricht

document.querySelectorAll(Selector)

Liefert ein Array mit Referenzen auf alle Elementknoten, die dem Selektor entsprechen.

document.createElement(tagName)

Liefert eine Referenz auf einen neuen Elementknoten vom angegebenen Typ (z.B. „form“, „input“, „h1“, „p“, usw.). Dieser Elementknoten gehört jedoch noch nicht zum Objektbaum. Dazu muss er erst eingefügt werden.

node.appendChild(newChild)

Fügt den Elementknoten newChild als neuen Kindknoten des Knotens node ein (ggf. hinter den letzten bereits bestehenden).

node.removeChild(oldChild)

Entfernt den Elementknoten oldChild des Knotens node.

node.setAttribute(name, wert)

Haben Sie einen Node als Variable können Sie mit setAttribute Werte setzen, wie z.B. die Id.

node.innerHTML = text

Sie können den Text auch direkt mit dem Befehl innerHTML setzen.

node.getAttribute(name)

Analog zu setAttribute können Sie mit getAttribute den Wert eines Attributes auslesen.

Eine Reihe nützlicher, browserunabhängiger Objekte sind in ECMAScript bereits vordefiniert. Unter anderem gibt es:

Array – Erzeugung

```
new Array()  
new Array(n)  
new Array(Element1, ..., Elementn)
```

Anzahl der Elemente: `a.length`

Array – Zugriff (auf ein Array a)

<code>a.push(Element)</code>	Fügt ein neues Element am Ende ein
<code>a.pop()</code>	Löscht das letzte Element
<code>a.unshift(Element)</code>	Fügt ein neues Element am Anfang ein
<code>a.shift()</code>	Löscht das erste Element
<code>a[i]</code>	Zugriff auf das i-te Element (beginnend bei 0)

Vordefinierte Objekte: (Fortsetzung)

Math – Braucht nicht instanziiert zu werden. Funktionen (u.a.):

<code>random()</code>	Liefert eine Zufallszahl r , wobei $0 \leq r < 1$
<code>round(r)</code>	Rundet die Zahl r auf eine ganze Zahl
<code>ceil(r)</code>	Liefert die kleinste ganze Zahl, die nicht kleiner als r ist
<code>floor(r)</code>	Liefert die größte ganze Zahl, die nicht größer als r ist
<code>sqrt(r)</code>	Berechnet die Quadratwurzel von r
<code>exp(r)</code>	Berechnet e^r
<code>log(r)</code>	Berechnet den natürlichen Logarithmus von r

Vordefinierte Objekte: (Fortsetzung)

Date – Erzeugung:

- `new Date()`
Erzeugt ein Datumsobjekt mit dem aktuellen Client-Datum und der aktuellen Client-Uhrzeit
- `new Date( 2006, 11, 15 )`
Erzeugt ein Datumsobjekt, das als Datum den 15. Dezember 2006 und als Zeit 0:00:00 Uhr enthält (die Monatszählung beginnt bei 0!)
- `new Date( 2006, 11, 12, 15, 30, 10 )`
Erzeugt ein Datumsobjekt, das als Datum den 12. Dezember 2006 und als Zeit 15:30:10 Uhr enthält

Intern enthält ein Datumsobjekt lediglich eine Zahl, welche die seit dem 1.1.1970 vergangenen Millisekunden angibt (bzgl. UTC).

Vordefinierte Objekte: (Fortsetzung)

Date – Zugriff auf die Komponenten:

<code>getDate(), setDate()</code>	Liefert bzw. setzt den Tag
<code>getMonth(), setMonth()</code>	Liefert bzw. setzt den Monat
<code>getFullYear(), setFullYear()</code>	Liefert bzw. setzt das Jahr
<code>getHours(), setHours()</code>	Liefert bzw. setzt die Stunde
<code>getMinutes(), setMinutes()</code>	Liefert bzw. setzt die Minute
<code>getSeconds(), setSeconds()</code>	Liefert bzw. setzt die Sekunde

Ausgabe:

<code>d.toString()</code>	Liefert das Datum von d als Zeichenfolge
<code>d.toLocaleString()</code>	Liefert die Uhrzeit von d als Zeichenfolge
<code>d.toISOString()</code>	Liefert Datum und Uhrzeit von d als Zeichenfolge, z.B. „Dienstag, 5. Dezember 2006 15:20:00“

1 Objekte

3 Asynchronität

2 **Events**

Auf Interaktionen reagieren

- Sie benutzen JavaScript im Client besonders, um Ihre Webanwendung interaktiv zu gestalten. Dazu müssen Sie auf Events reagieren. Es gibt folgende Eventtypen in JavaScript:
- UI events (Klick, Touch, Mausbewegung, ...)
- UI logical events (Änderung des Fokus, ...)
- Mutation events / mutation observers (Einfügen oder Entfernen von Knoten, Ändern von Attributen,...)
- Custom events (selbstdefinierte Events)

- Typischerweise wird ein Event vom User Agent (Browser) geworfen
- Alle DOM Events implementieren das Interface Event
- Man kann listener auf Events anmelden
- Das Event bewegt sich durch das DOM (capturing bzw. bubbling) und benachrichtigt auf diesem Weg alle listener

- Um auf Events zu reagieren, meldet man einen EventListener an. Die Methoden sind im Interface EventTarget definiert
- Zum Beispiel:

```
element.addEventListener("someEventType",  
    function(e) {  
        //do something  
    })
```

Optionen möglich

- Unter DOM Mutations versteht man das Ändern von Attributen der DOM-Elemente. Dazu zählt das Hinzufügen oder Entfernen von Kindern. Die Methoden `document.createElement( tagName )`, `node.appendChild( newChild )` und `node.removeChild( oldChild )` haben wir bereits besprochen.
- Neuere Browser unterstützen den `MutationObserver`. Damit können Sie z.B. mit `obj.addEventListener('DOMNodeInserted', callback, false);` arbeiten.

- JSON (JavaScript Object Notation) ist ein menschenlesbares Format zum Austausch von Daten. Es ist an JavaScript orientiert. Sie können JSON als Alternative zu XML verwenden, um z.B. Daten vom Server zu laden.
- Sie kennen folgenden Weg, um in JavaScript Objekte anzulegen:

```
let p1 = new Object();  
p1.x = 1;
```

- Sie können ein Objekt aber auch als Literal erzeugen.

```
let p1 = {x: 1, y: 42, c: {}};
```

```
{
  "Titel": "JavaScript: The Good Parts: Working with...",
  "Autor": "Douglas Crockford",
  "ISBN-10": "0596517742",
  "Verlag": "O'Reilly and Associates",
  "Rezensionen": [
    {
      "Autor": "Algorithmenfreund",
      "Text": "Der Autor..."
    },
    {
      "Autor": "Moaaz Ashour",
      "Text": "One of my teachers..."
    }
  ]
}
```

- Eine dem Literal ähnliche Form können Sie auch in einem JSON-Dokument verwenden
- Mit den Funktionen `JSON.parse()` und `JSON.stringify()` können Sie Strings in JSON-Objekte umwandeln (und umgekehrt).

1 Objekte

2 Events

3 Asynchronität

Daten nachladen mit Callbacks, Promises und fetch()

- In JavaScript arbeiten Sie oft asynchron
- Sie möchten z.B. Daten im JSON Format vom Server nachladen, um die Seite zu aktualisieren (Client-Side Rendering, AJAX)
- Beispiel: Single-Page Applications
 - Es wird nur ein HTML-Dokument initial geladen
 - alle Inhalte werden nachgeladen
- Damit Website reaktiv bleibt, sollte Nachladen im Hintergrund passieren
 - nach der Fertigstellung werden bestimmte Funktionen ausgelöst (asynchroner Callback)
- Sie können Daten vom Server laden und dann Events und Callbacks benutzen, um asynchron zu arbeiten.

- Die Fetch-API ermöglicht es, Daten vom Server zu laden.
- `fetch()` bekommt eine URL übergeben und gibt ein Promise zurück (folgt). Diese API wird momentan oft verwendet.
- Alternativ kann die XMLHttpRequest API verwendet werden. Diese findet man eher in älterem Code und werden häufig mit den bekannten EventListeners verwendet.

```
const request = new XMLHttpRequest();
try {
  request.open("GET", "https://.../products.json");
  request.responseType = "json";
  request.addEventListener(
    "load", () => initialize(request.response)
  );
  request.addEventListener(
    "error", () => console.error("XHR error")
  );
  request.send();
} catch (error) {
  console.error(`XHR error ${request.status}`);
}
```

- Um asynchrone Programmierung zu erleichtern, gibt es Mechanismen, z.B. Promises
- Eine asynchrone Funktion gibt ein Objekt vom Typ Promise zurück. Dies repräsentiert den Status der Operation
- Die `then()` Methode eines Promises wird aufgerufen, wenn eine erfolgreiche Antwort der asynchronen Funktion kommt. Fehler können Sie mit `catch()` abfangen
- Der Einfachheit halber arbeiten Sie oft mit `async`, um asynchrone Funktionen zu kennzeichnen. In der Funktion nutzen Sie `await` um eine Funktion aufzurufen, die ein Promise zurück gibt.

fetch() lädt Ressourcen vom Netzwerk und gibt ein Promise zurück

```
sync function fetchProducts() {  
  try {  
    const response = await fetch(  
      "https://.../products.json"  
    );  
    if (!response.ok) {  
      throw new Error(`HTTP error: ${response.status}`);  
    } const data = await response.json(); return data;  
  } catch (error) {  
    console.error(`Could not get products: ${error}`);  
  }  
}  
const promise = fetchProducts();  
promise.then((data) => console.log(data[0].name));
```

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise.then((response) => {
  const jsonPromise = response.json();
  jsonPromise.then((data) => {
    console.log(data[0].name);
  });
});
```

- Sie können Promises verketteten (chaining), wenn Sie mit dem Ergebnis einer asynchronen Funktion die nächste aufrufen wollen. Das funktioniert mit Verschachtelung (nesting).

```
const fetchPromise = fetch(  
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",  
);  
  
fetchPromise  
  .then((response) => response.json())  
  .then((data) => {  
    console.log(data[0].name);  
  });
```

- ...oder dadurch das Promise einfach weiter zu geben.

Probleme mit Promises

Es gibt verschiedene Probleme, die bei der Benutzung von Promises auftreten können, z.B. sind einfache Loops manchmal problematisch. Die Benutzung von Promise.all() ist vorzuziehen.

```
// I want to remove() all docs
db.allDocs({include_docs: true}).then(function (result) {
  result.rows.forEach(function (row) {
    db.remove(row.doc);
  });
}).then(function () {
  // I naively believe all docs have been removed() now!
});
```

Returned undefined, es wird nicht gewartet

Probleme mit Promises

Die Fehlerbehandlung ist teilweise schwierig.

```
const handleSave = rawData => {  
  saveUser(rawUserData)  
    .then(user => createToast(`User ${displayName(user)} has been created`))  
    .catch(err => createToast(`User could not be saved`));  
};
```

Catch bezieht sich auch auf den Erfolgsfall. Wenn es z.B. bei displayName einen Fehler gibt, springt er dann in den catch() Block