

Verteilte Systeme 1

Technologien des World Wide Web

christian.zirpins@hs-karlsruhe.de

Browser Interaktion mit JavaScript



Hochschule Karlsruhe
Technik und Wirtschaft

UNIVERSITY OF APPLIED SCIENCES



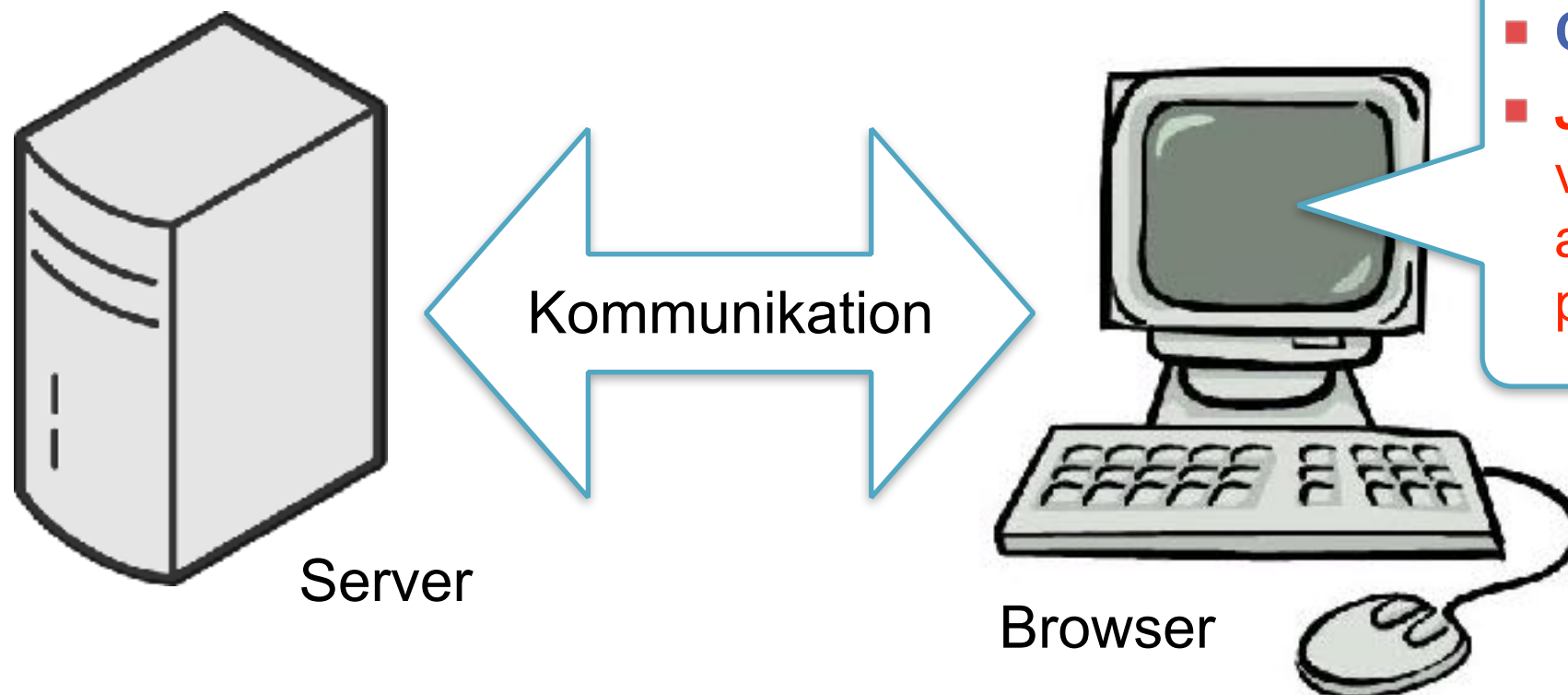
VS1 Termine im Sommer 2017

Termin (ca.)	Thema	Vorbereitung (Begleitbuch)	Raum
KW11: 13.03, 15.03	HTTP , die Sprache des Web		E 301/304
KW12: 20.03, 22.03	Web Apps mit HTML5	Web App Development Kapitel 2	E 301/304
KW13: 27.03, 29.03	Gestaltung von Web Apps mit CSS3	Web App Development Kapitel 3	E 301/304
KW14: 03.04, 05.04	<i>Übung: Webseite mit HTML5/CSS3 erstellen</i>		LI 137
KW15: 10.04, 12.04	Browser Interaktion mit JavaScript	Web App Development Kapitel 4	E 301/304
KW16	Ostern		
KW17: 24.04, 26.04	<i>Übung: Formulare mit JavaScript / HTML5 APIs</i>		LI 137
KW18	Maifeiertag		
KW19: 08.05, 10.05	JavaScript auf dem Server mit Node.js	Web App Development Kapitel 6	E 301/304
KW20: 15.05, 17.05	<i>Übung: Node.js / Express Web App erstellen</i>		LI 137
KW21: 22.05, 24.05	Web Entwicklung mit Ajax & Co	Web App Development Kapitel 5	E 301/304
KW22: 29.05, 31.05	<i>Übung: Web App mit REST und AJAX erweitern</i>		LI 137
KW23	Pfingsten		
KW24: 12.06, 14.06	Web Apps Personalisieren	Web App Development Kapitel 9	E 301/304
KW25: 19.06, 21.06	Web App Sicherheit		E 301/304
KW26: 26.06, 28.06	Klausurvorbereitung, Q&A		E 301/304

Heutige Lernziele

Nach dieser Vorlesung können Sie...

- ...**OO-Prinzipien** in JavaScript **verwenden**
- ...das Prinzip der **Callbacks** **erklären**
- ...interaktive Web-Anwendungen mit **Ereignissen** **schreiben**
- ...jQuery-basierten Code in **jQuery-losen Code** **übersetzen**



- **HTML**: zeichnet Inhalte aus
- **CSS**: steuert Erscheinungsbild
- **JavaScript**: manipuliert Inhalte von HTML Dokumenten, reagiert auf Benutzerinteraktion, programmiert Plattform APIs

Learning Web App Development

LWAD Kapitel 4

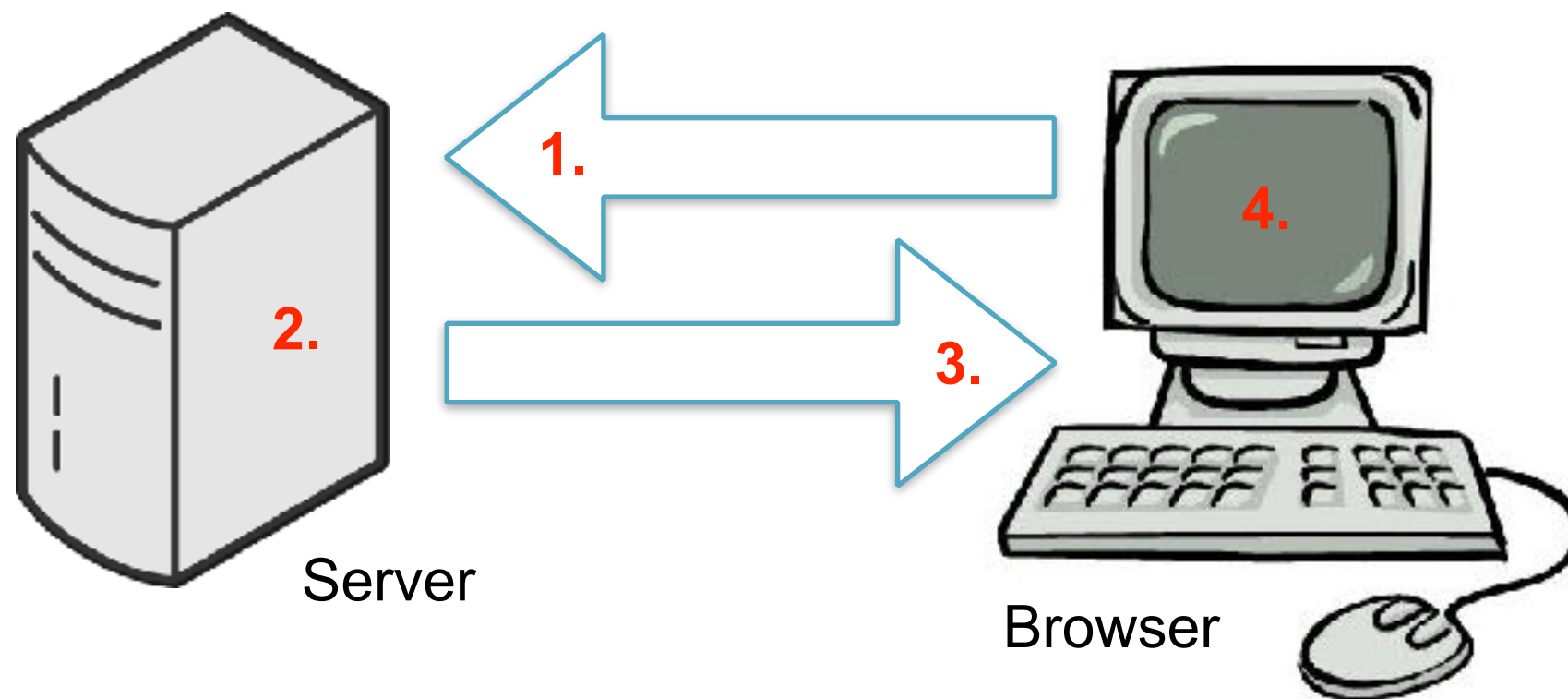
Umfasst JavaScript-Grundlagen ...

- Wie man JavaScript in einer Webanwendung integrieren kann
- Wichtige JavaScript-integrierte Typen
- Wie benutzt man JavaScript-Kontrollstrukturen (if, for, while)
- Wie deklariert man Variablen und Funktionen
- Zweck von `console.log()`
- Arbeiten mit Arrays
- Wie man grundlegende jQuery-Funktionen verwendet

Scripting Überblick

Anfordern/Bearbeiten einer Webseite in 4 Schritten

1. Browser sendet GET Request für Webseite an Webserver
2. Webserver führt serverseitige Skripts aus (PHP, ASP, node.js, etc.)
3. Web-Server sendet (generierte) HTML-Datei zum Browser
4. Browser führt Skripte (JavaScript, etc.) aus und *rendert* die Seite



JavaScript macht Web-Apps **interaktiv** und **reaktiv** auf Benutzeraktionen

Serverseitiges Scripting

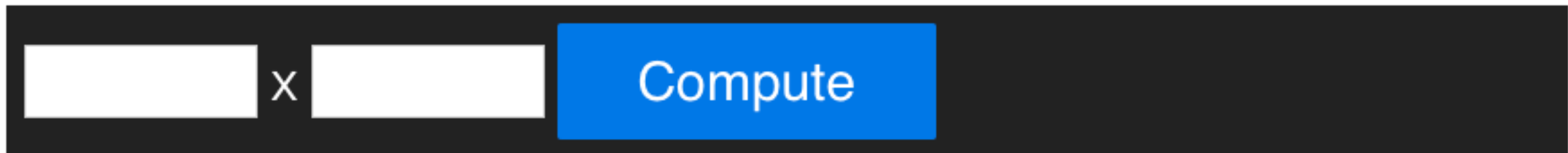
- Quellcode ist privat, Ergebnis der Skriptausführung wird zurückgegeben (in HTML), nicht das Skript selbst
- HTML kann von jedem Browser gerendert werden
- Serverseitige Skripte können auf zusätzliche **Ressourcen** (einschließlich Datenbanken) zugreifen
- Serverseitige Skripte können **nicht-standardmäßige Sprachfunktionen** verwenden (Software des Servers ist bekannt)

Clientseitiges Scripting

- Quellcode ist für alle sichtbar ("View Page Source")
- Skriptausführung durch den Browser reduziert Last des Webserver
- Alle nötigen **Rohdaten** (z.B. für Visualisierungen) müssen vom Client heruntergeladen und verarbeitet werden
- JavaScript ist **ereignisgesteuert**: Codebausteine werden oft als Reaktion auf Benutzeraktionen ausgeführt (Klick, "Hover", Verschieben usw.)

JavaScript ist

- Immer noch meist eine clientseitige Technologie (dies ändert sich)
- Ganz anders als die Programmiersprache Java
- Nicht in allen Browsern gleich unterstützt (ähnlich CSS3, HTML5)
- Wird verwendet, um das Web interaktiver zu gestalten
 - Seitenlayout basierend auf Ereignissen ändern (z.B. Knopfdruck)
 - Ändern der Interaktion anhand von Browsertyp, Sprache, Cookies
 - Berechnungen werden auf dem Client durchgeführt.



A dark gray rectangular container holds a form. On the left is a white rectangular input field. To its right is a small 'x' symbol. To the right of the 'x' is another white rectangular input field. Further right is a blue rectangular button with the word 'Compute' written in white text.

JavaScript kann viel mehr als wir hier behandeln
Beispiele bei **Chrome Experiments**

JavaScript vs. OO Sprachen

JavaScript	OO [Java]
interpretiert (JIT)	kompiliert
Programm lesbar (<u>mehr oder weniger</u>)	byte code
weniger Datentypen	--
Variablen müssen nicht deklariert werden	Variablen müssen deklariert werden
"Silent Errors" WebConsole/Firebug helfen	Errors und Exceptions
Fokus auf Funktionen	Fokus auf Klassen & Objekten
Meist in Verbindung mit CSS & HTML	--

JavaScript Entwurfsmuster

LWAD Kapitel 4

- Fokussiert darauf welches JavaScript wo zu platzieren ist
- Reduziert Redundanz im Code auf **Funktionsebene**
- Betont nicht OO Prinzipien

OO für JavaScript

- Kleine Programme brauchen sie oft nicht
- JavaScript hat **Funktionen als "First Class Citizens"** (nicht Klassen wie in Java)
- Große Projekte profitieren von OO
- OO **gruppiert** Daten und Verhalten
- **Eingebaute Objekte**: *Strings, Arrays, HTML/DOM-Knoten*
- Objekte können auf unterschiedliche Weise **erstellt** werden (am besten man beschränkt sich auf eine Weise)

Objekte in JavaScript

- `new Object()` erzeugt ein leeres Objekt, das Name/Wert-Paare aufnehmen kann
 - Name: beliebiger String
 - Wert: irgendetwas (String, Array, Zahl, etc.) außer `undefined`
- Member (Attribute) werden zugegriffen durch
 - `.name` (Punktnotation)
 - `[Name]` (Klammernotation)

```
var note1 = new Object();  
note1["type"] = 1;  
note1["note"] = "Math homework due";  
console.log(note1["type"]); /* schreibt: 1 */  
console.log(note1.note); /* schreibt: "Math..." */
```


Anderer Weg: Objektliterale

```
var note2 = {  
  type: 2,  
  message: "Math homework due"  
  /* kein Komma am Ende */  
};
```

Eine Methode hinzufügen

```
var note1 = new Object();
note1["type"] = 1;
note1["note"] = "Math homework due";
note1["toString"] = function() {
    /* 'this' bezieht sich auf aktuelles Objekt */
    return "Note:" + this.note + ",type:" + this.type;
};
```

```
var note2 = {
    type: 2,
    message: "Math homework due",
    toString: function() {
        /* 'this' bezieht sich auf aktuelles Objekt */
        return "Note:" + this.note + ",type:" + this.type;
    }
};
```

Ist das genug?

```
var note = {  
  type: 1,  
  message: "Math homework due",  
  toString: function() {  
    return "Note:" + this.note + ",type:" + this.type;  
  }  
};
```

- Was passiert, wenn wir 1000 Objekte dieser Art brauchen?
- Was passiert in einem großen Projekt, wenn eine Methode allen `note` Objekten hinzugefügt werden muss?

1. Entwurfsmuster

Basic Constructor

Basic Constructor

In JavaScript sind Funktionen "first-class citizens"

```
function Note( note, type ) {  
    this.note = note; /* 'this': Referenz auf aktuelles Objekt */  
    this.type = type;  
  
    this.setType = function(t) {this.type = t;};  
  
    this.getType = function() {return this.type;};  
  
    this.getNote = function() {return this.note;};  
  
    this.toString = function () {  
        return "Note: "+this.note+", type: "+this.type;  
    };  
}
```

```
var note1 = new Note("Maths homework assignment", 1);  
note1.setType(2);  
note1.toString();  
var note2 = new Note("English homework due"); /* was passiert  
mit type? */  
var note3 = Note("Music homework due", 3); /* was nun? */
```

Basic Constructor

- Ein **Objektkonstruktor** ist nur eine normale Funktion
- Was macht JavaScript mit **new**?
 - Ein neues anonymes leeres Objekt wird erstellt und als **this** verwendet
 - Am Ende der Funktion wird das neue Objekt **zurückgegeben**

Basic Constructor

```
/* Erinnerung: JavaScript ist dynamisch typisiert */
var note1 = new Note("Maths homework", "IMPORTANT");
note1.type; /* "IMPORTANT" */
var note2 = new Note("English homework", 1);
note2.type; /* 1 */
note2.dueDate = "1-1-2015"; /* neues Attribut spontan ergänzt */
note1.toString(); /* "Note: Maths homework, type: IMPORTANT" */
note1.toString = function(){return this.type;};
note1.toString(); /* "IMPORTANT" */

/* Es gibt auch einige zusätzliche Funktionen */
note1.hasOwnProperty("dueDate"); /* false */
note1.hasOwnProperty("type"); /* true */
```

- Neue Variablen und Methoden können sofort hinzugefügt werden
- Objekte haben Standardmethoden (Prototypverkettung)

Zusammenfassung: Basic Constructor

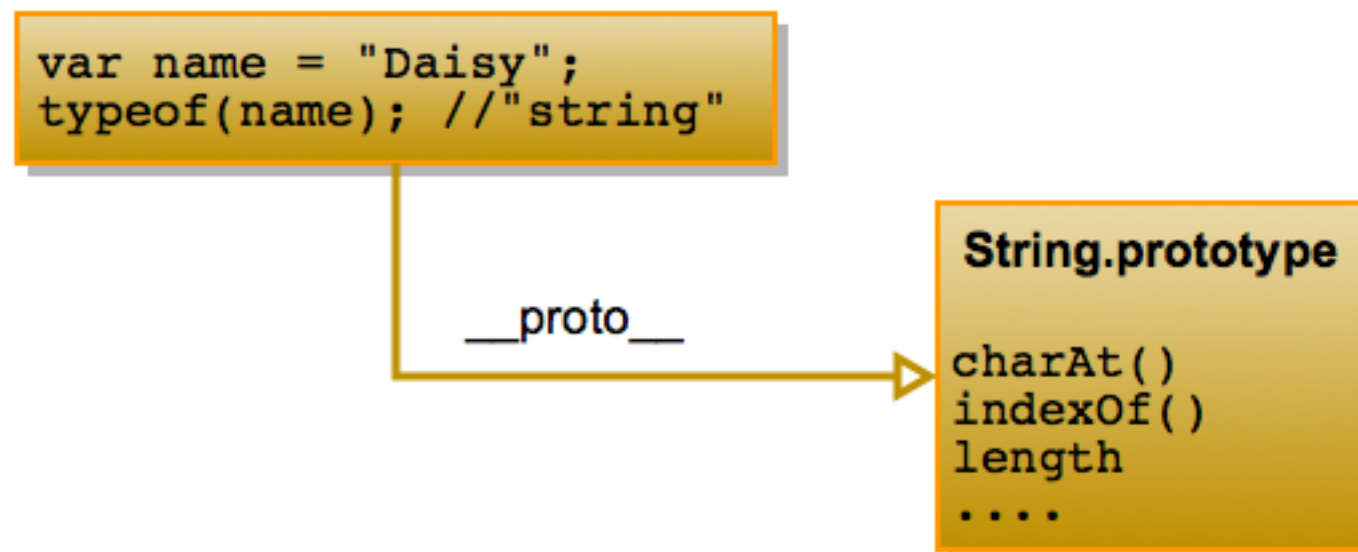
- Vorteil: sehr einfach zu bedienen
- Probleme:
 - Nicht offensichtlich wie **Vererbung** funktioniert (z.B. `NoteWithDueDate`)
 - Objekte teilen Funktionen **nicht**
 - Funktion `toString()` wird nicht zwischen `note1` und `note2` geteilt
 - Alle Mitglieder sind öffentlich
 - Jedes Stück Code kann die Attribute `type` und `note` abrufen/ändern/löschen (!)

2. Entwurfsmuster

Prototype-based Constructor

Prototyp Verkettung erklärt

- Objekte haben einen **geheimen Zeiger** auf ein anderes Objekt: den **Prototyp** des Objekts
 - Eigenschaften des Konstruktor-Prototyps sind auch im neuen Objekt zugänglich
 - Wenn ein Member nicht im Objekt definiert ist, wird der **Prototypkette** gefolgt, bis das Element gefunden wird



Prototype-based Constructor

```
function Note( note, type ) {  
    this.note = note; /* this: Referenz auf aktuelles Objekt */  
    this.type = type;  
}  
  
/* Member Methoden werden einmal im Prototyp definiert */  
Note.prototype.setType = function(t) {this.type = t;};  
Note.prototype.getType = function() {return this.type;};  
Note.prototype.getNote = function() {return this.note;};  
Note.prototype.toString = function () {  
    return "Note: "+this.note+", type: "+this.type;  
};  
  
// Using it:  
var note1 = new Note("Maths homework due", "IMPORTANT");  
var note2 = new Note("English homework", 2);  
note1.getType(); /* "IMPORTANT" */  
note2.getNote(); /* "Maths homework due" */
```

Prototype-based Constructor

Prototypänderungen spiegeln sich auch in bestehenden Objekten wider!

```
function Note( note, type ) {  
    this.note = note; /* this: Referenz auf aktuelles Objekt */  
    this.type = type;  
}  
  
/* Member Methode setType() definiert */  
Note.prototype.setType = function(t) {this.type = t;};  
  
var note1 = new Note("Maths homework due", "IMPORTANT");  
note1.setType(2); /* OK */  
note1.getType(); /* TypeError: note1.getType keine Funktion*/  
  
/* Definition der Methode können wir nachholen */  
Note.prototype.getType = function() {return this.type;}  
  
note1.getType(); /* 2 */
```

Prototype-based Constructor

Vererbung durch Prototyping

```
function Note( note, type ) {
    this.note = note;
    this.type = type;
}

Note.prototype.setType = function(t) {this.type = t;};
Note.prototype.getType = function() {return this.type;};
Note.prototype.getNote = function() {return this.note;};
Note.prototype.toString = function () {
    return "Note: "+this.note+", type: "+this.type;
};

/* Konstruktor */
function NoteWithDeadline(note,type,dueDate) {
    this.note = note;
    this.type = type;
    this.dueDate = dueDate;
};

/* Umleiten des Prototype */
NoteWithDeadline.prototype = new Note();
/* Umleiten des Konstruktors */
NoteWithDeadline.prototype.constructor = NoteWithDeadline;

/* using it */
var nw = new NoteWithDeadline("Maths homework",1,"1-1-2015");
nw.setType(2); /* funktioniert, setType(t) ist in Note definiert */
```

Zusammenfassung: Prototype-based Constructor

- Vorteile:

- **Vererbung** ist leicht zu erreichen
- Objekte teilen Funktionen

- Problem:

- Alle Mitglieder sind **öffentlich**
 - Jedes Stück Code kann die Attribute `type` und `note` abrufen/ändern/löschen (!)

3. Entwurfsmuster

Module

JavaScript Scoping (Sichtbarkeitsbereiche)

- Aller JavaScript-Code kommt in denselben **Namensraum**
- JavaScript hat begrenztes Scoping
 - **var** in Funktion: Sichtbarkeit lokal begrenzt
 - **var** außerhalb einer Funktion: globale Sichtbarkeit
 - Kein **var**: globale Sichtbarkeit (gilt auch für Funktionsnamen)

JavaScript Scoping

- Aller JavaScript-Code kommt in denselben Namensraum
- JavaScript hat begrenztes Scoping
 - `var` in Funktion: Sichtbarkeit lokal begrenzt
 - `var` außerhalb einer Funktion: globale Sichtbarkeit
 - Kein `var`: globale Sichtbarkeit (gilt auch für Funktionsnamen)

```
var note1 = new Note("Maths", 1); /* global */
var note2 = new Note("English", 3); /* global */
function calcMinType(n1,n2) { /* global */
    var t1 = Number(n1.type); /* local */
    t2 = Number(n2.type);      /* global */
    return Math.min(t1,t2);
}
t1; /* ReferenceError: t1 is not defined */
t2; /* ReferenceError: t2 is not defined */
calcMinType(note1,note2);
t1; /* ReferenceError: t1 is not defined */
t2; /* 3 */
```

- Was ist, wenn eine andere im Projekt verwendete JavaScript-Bibliothek `note1` oder `calcMinType(a, b, c)` definiert?

Module

■ Ziele:

- Keine globalen Variablen/Funktionen deklarieren, sofern nicht nötig
- Private/öffentliche Member emulieren
- Nur nötige Member der Öffentlichkeit zeigen (als API)

■ Ergebnisse:

- Weniger mögliche Konflikte mit anderen JavaScript-Bibliotheken
- Public API minimiert unbeabsichtigte Seiteneffekte bei falscher Verwendung

Module

```
var notesModule = (function () {  
  
    /* 'private' Member */  
    var noteCounter = 0;  
    ...  
  
    /* 'public' Member; gibt verwendbares Objekt zurück */  
    return {  
        incrNoteCounter : function () {  
            noteCounter++;  
        },  
        ...  
    };  
})(); /* Funktion wird aufgerufen */
```

```
notesModule.incrNoteCounter(); /* OK */  
notesModule.noteCounter; /* undefined */
```

Zusammenfassung: Module

■ Vorteile:

- Kapselung funktioniert
- Objekt Member sind entweder öffentlich oder privat

■ Probleme:

- Auf "öffentliche" und "private" Member wird unterschiedlich zugegriffen
- Ändern von Members zwischen öffentlich/privat kostet Zeit
- Methoden, die später hinzugefügt werden, können nicht auf "private" Member zugreifen

Ereignisse & das DOM

Nochmal LWAD Kapitel 4

```
var main = function () {  
    "use strict";  
    $(".comment-input button").on("click", function (event) {  
        var $new_comment = $("<p>"),  
        comment_text = $(".comment-input input").val();  
        $new_comment.text(comment_text);  
        $(".comments").append($new_comment);  
    });  
};  
$(document).ready(main);
```

- Verwendet jQuery extensiv (eine große Zeitersparnis)
- Aber: es ist wichtig zu verstehen, was jQuery "versteckt"

Nochmal LWAD Kapitel 4

```
/* jQuery's Weg, um DOM Elemente zuzugreifen */  
$(".comment-input button").on("click", function (event) {  
    ...  
});
```

- Mit jQuery: egal ob Klasse oder id oder ..., das Zugriffsmuster ist das gleiche, d.h. `$()`
- **Callback** Prinzip: wir definieren, was passieren soll, wenn ein Ereignis ausgelöst wird

Schritt für Schritt: eine reaktive Benutzeroberfläche erstellen

1. Eine Steuerung wählen (z. B. eine Schaltfläche)
2. Ein Ereignis wählen (z.B. einen Klick auf die Schaltfläche)
3. Eine JavaScript-Funktion schreiben: was sollte passieren, wenn das Ereignis auftritt? (z.B. ein Popup-Fenster erscheint)
4. Die Funktion an das Ereignis der Steuerung hängen

Document Object Model: **wie man Elemente in einer Seite "findet"**

DOM: was geht?

- Extrahiere den Zustand eines Elements
 - Ist das Kontrollkästchen aktiviert?
 - Ist die Schaltfläche deaktiviert?
 - Ist ein `<h1>` auf der Seite?
- Ändern des Zustandes eines Elements
 - Aktiviere ein Kontrollkästchen
 - Deaktiviere eine Schaltfläche
 - Erstelle ein `<h1>`-Element auf einer Seite, wenn keins da ist
- Ändere den Stil eines Elements
 - Ändere die Farbe einer Schaltfläche
 - Ändere die Größe eines Absatzes
 - Ändere die Hintergrundfarbe der Seite

Auswahl von Gruppen von DOM-Elementen

`document` (das Webseitenobjekt) enthält einige Methoden:

- `document.getElementById`
- `document.getElementsByTagName`
- `document.getElementsByClassName`
- `document.getElementsByName`
- `document.querySelector`
- `document.querySelectorAll`

document.getElementById

window ist das Browserfensterobjekt

```
/* JavaScript file, importiert mit <script> */

/* Wir hängen eine Funktion an ein Button-Click Ereignis
 * nachdem das DOM fertiggeladen ist (d.h. alle HTML Elemente sind da)
 */
window.onload = function() {
    document.getElementById("myButton").onclick = sayHello;
};

/* Definition der Funktion */
function sayHello() {
    var tb = document.getElementById("out");
    tb.value = 'Hello World';
};
```

Say Hello World!

Hello World!

DOM Objekteigenschaften

```
<!-- HTML -->
<div id="main" class="main_class">
  <p>Hello <em>World!</em></p>
  
</div>
```

```
//JavaScript
var m = document.getElementById( "main" );
var w = document.getElementById( "worldImage" );
```

Eigenschaft	Beispiel
tagName	m.tagName ist div
className	m.className ist main_class
innerHTML	m.innerHTML ist <p>Hello....
src	w.src ist images/1.jpg

DOM Objekteigenschaften (Formulare)

```
<!-- HTML -->
<input id="firstName" type="text" />
<input id="agreed" type="checkbox" checked="checked" /> Did you
read the terms and conditions?
```

```
//JavaScript
var f = document.getElementById("firstName");
var a = document.getElementById("agreed");
```

Eigenschaft	Beispiel
value (Text in input Steuerung)	f.value ist vielleicht "Tom"
checked (checkbox)	a.checked ist true
disabled (ob Steuerung deaktiviert ist)	a.disabled ist false
readOnly (read-only text box)	f.readOnly ist false

Neue Knoten erstellen

HTML-Tags und Inhalte können dynamisch in zwei Schritten hinzugefügt werden:

1. Erstelle einen DOM-Knoten
2. Neuer Knoten wird der Seite als **Kind eines vorhandenen Knotens** hinzugefügt

Einfügen von Knoten in den DOM-Baum

Jedes DOM-Element-Objekt verfügt über diese Methoden

Name	Beschreibung
<code>appendChild(node)</code>	Hängt den Knoten am Ende der Child-Liste des Knotens an
<code>insertBefore(new, old)</code>	Legt den gegebenen neuen Knoten in die Child-Liste dieses Knotens kurz vor dem alten Child
<code>removeChild(node)</code>	Entfernt den gegebenen Knoten aus der Child-Liste dieses Knotens
<code>replaceChild(new, old)</code>	Ersetzt das angegebene Child durch einen neuen Knoten

Einfügen von Knoten in den DOM-Baum

- A list element
- List element 2

Add list element

```
/* JavaScript */
window.onload = function() {
    document.getElementById("myButton").onclick = addElement;
};

function addElement() {
    var ul = document.getElementById('ul_domtree');
    var li = document.createElement('li');
    li.innerHTML = 'List element ' + (ul.childElementCount+1) + ' ';
    ul.appendChild(li);
};
```

Löschen von Knoten aus dem DOM-Baum

- List element 4
- List element 5
- List element 6

Remove last element

Remove first element

```
/* JavaScript */
window.onload = function() {
    document.getElementById("myButton1").onclick = removeLastChild;
    document.getElementById("myButton2").onclick = removeFirstChild;
};

function removeLastChild() {
    var ul = document.getElementById('ul_domtree2');
    ul.removeChild(ul.lastChild);
};

function removeFirstChild() {
    var ul = document.getElementById('ul_domtree2');
    ul.removeChild(ul.firstChild);
}
```

this : das aktuelle Objekt

- Event-Handler an Objekte des angehängten Elements gebunden
- Handler-Funktion "weiß", auf welches Element es hört (**this**)
- Vereinfacht Programmierung, eine Funktion für verschiedene Objekte



```
/* Eine Funktion pro Button (redundant) */
window.onload = function() {
  document.getElementById("button10").onclick = computeTimes10;
  document.getElementById("button23").onclick = computeTimes23;
  document.getElementById("button76").onclick = computeTimes76;
}

/* Funktioniert nicht: Event-Handlers haben keine Parameter */
window.onload = function() {
  document.getElementById("button10").onclick = computeTimes(10);
  document.getElementById("button23").onclick = computeTimes(23);
  document.getElementById("button76").onclick = computeTimes(76);
}
```

this : das aktuelle Objekt

- Event-Handler an Objekte des angehängten Elements gebunden
- Handler-Funktion "weiß", auf welches Element es hört (**this**)
- Vereinfacht Programmierung, eine Funktion für verschiedene Objekte

10 times

23 times

76 times

```
/* beste Option: nutze 'this' */
window.onload = function() {
    document.getElementById("button10").onclick = computeTimes;
    document.getElementById("button23").onclick = computeTimes;
    document.getElementById("button76").onclick = computeTimes;
}

function computeTimes() {
    var times = parseInt(this.innerHTML);
    var input = parseFloat(document.getElementById("input"));
    var res = times * input;
    alert('The result is ' + res);
}
```

Beispiele für Ereignisse

Maus-Ereignisse

- Ein Klick auf die Maustaste ist eine Reihe von Ereignissen ("Klick"):
 1. `mousedown`
 2. `mouseup`
 3. `click`
- Ein Mausklick bei gedrückter Maustaste ("Ziehen"):
 4. `mousedown`
 5. `mousemove`
 6. . . .
 7. `mousemove`
 8. `mouseup`
- Andere Maus-Effekte: `dblclick`, `mouseover`, `mouseout`

onblur & onchange

- **Onblur**: feuert, wenn ein Element nicht mehr im Fokus ist
 - Wird häufig in der **Eingabevalidierung** verwendet
- **Onchange**: Wird ausgelöst, wenn sich der Wert eines Elements ändert

Keyboard- und Text-Ereignisse

Ereignis	Beschreibung
blur	Element verliert Tastaturfokus
focus	Element bekommt Tastaturfokus
keydown	Benutzer drückt Taste während Element Tastaturfokus hat
keypress	Benutzer drückt Taste und lässt los, während Element Tastaturfokus hat (problematisch)
keyup	Benutzer lässt Taste los, während das Element den Tastaturfokus hat
select	Benutzer selektiert Text in einem Element

Literatur

- **Learning Web App Development, Kapitel 4**
- Empfehlung: Marijn Haverbeke, "*Eloquent JavaScript*", No Starch Press, 2014 (Online: <http://eloquentjavascript.net>)

