



EntwicklerCamp 2014

Track 3, Session 8:

JavaScript für Fortgeschrittene

Gelsenkirchen, 19. März 2014

Innovative Software-Lösungen

www.assono.de



Thomas Bahn

- Diplom-Mathematiker, Universität Hannover
- seit 1997 entwickle ich mit Java und relationalen Datenbanken
- seit 1999 mit Notes/Domino zu tun: Entwicklung, Administration, Beratung und Schulungen
- regelmäßiger Sprecher auf nationalen und internationalen Fachkonferenzen zu IBM Lotus Notes/Domino und Autor für THE VIEW



tbahn@assono.de



www.assono.de/blog



04307/900-401



assono

IT-Consulting & Solutions



Werkzeug

- Mozilla Firefox mit
 - Firebug:
<http://getfirebug.com/>
 - Venkman (JavaScript-Debugger):
<https://developer.mozilla.org/de/docs/Venkman>
- andere Browser zum Testen
 - Firebug Lite:
<http://getfirebug.com/firebuglite>
 - Fiddler:
<http://www.telerik.com/download/fiddler>



Variablendeklaration

- Variablen-Deklaration:
`var name;`
- Variablen-Deklaration kombiniert mit Wertzuweisung:
`var name = wert;`
- SSJS-Spezialität: typisierte Variablen-Deklaration:
`var name : typ = wert;`



Vorsicht Falle: Prüfung, ob Variable deklariert ist

- häufig, aber vereinfacht:

```
if (a) {  
    ...  
}
```
- Vorsicht: der Block wird auch ausgeführt, wenn a gleich `false`, `null`, `""` oder `0` ist
- besser:

```
if (typeof a !== "undefined") {  
    ...  
}
```



Variablentypen

- „loose typing“ heißt nicht untypisiert!
- Es gibt folgende 6 Typen:
 - String
 - Number
 - Boolean
 - Object
 - null
 - undefined



Spezielle Strings

- `\` Maskierungszeichen
- `\\` ein Rückwärtsstrich `\`
- `\'` ein einfacher Anführungsstrich
- `\"` ein doppelter Anführungsstrich
- `\n` neue Zeile (newline)
- `\r` "Wagenrücklauf" (carriage return)
- `\t` Tabulator
- `\u` Unicode-Zeichen, z. B. `\u042F` \Rightarrow я



String-Objekt

- nützliche Methoden des String-Objekts
 - `toUpperCase()`, `toLowerCase()`: wie LotusScript
 - `charAt(position)`: Zeichen an der Position
 - `indexOf(suchstring)`: Position des Suchstrings
 - `indexOf(suchstring, startPos)`: Position des Suchstrings ab `startPos`
 - `lastIndexOf(suchstring)`: dito, aber von hinten
 - `substring(anfang, ende)`: Teilstring
 - `slice(anfang, ende)`: wie `substring`, aber bei negativem ende wird `length` hinzuaddiert (zählt also von hinten)
 - `split(trenner)`: zerlegt String in ein Array



Zahlen

- alle Zahlen sind 64-bit Gleitkommazahlen („double“)
- Rundungsdifferenzen durch Binär-Dezimal-Konvertierung
- Bit-Shifting (<<, >>) möglich, aber kontraproduktiv
- `Number.MAX_VALUE` \Rightarrow `1.7976931348623157e+308`
`Number.MIN_VALUE` \Rightarrow `5e-324`
- `Number.toString(basis)`, z. B.
`var a = 100;`
`a.toString(16)` \Rightarrow `"64"`
`a.toString(2)` \Rightarrow `"1100100"`



Math-Objekt

- nützliche Konstanten und Methoden des Math-Objekts:
 - `Math.PI` = 3.141592653589793
 - `Math.E` = 2.718281828459045
 - `Math.SQRT2` = 1.4142135623730951
 - `Math.LN2` = 0.6931471805599453
 - `Math.LN10` = 2.302585092994046
 - `Math.random()`: $0 \leq \text{Zufallszahl} < 1$
 - `Math.round()`, `Math.floor()`, `Math.ceil()`,
`Math.sin()`, `Math.cos()`, `Math.atan()`,
`Math.min()`, `Math.max()`,
`Math.pow()`, `Math.sqrt()`



Zahlen (forts.)

- `0x` Präfix für Hexadezimalzahl-Literale, z. B. `0xFF`
`0` Präfix für Oktalzahl-Literale, z. B. `014` (=12)
- Achtung bei `parseInt()`,
z. B. `parseInt("08")` \Rightarrow 0,
also abgeschnitten beim ersten ungültigen Zeichen
und `parseInt(08)` \Rightarrow Fehler:
08 is not a legal ECMA-262 octal constant
- insbesondere bei Benutzereingaben, z. B. beim Monat
- daher besser immer bei `parseInt()` die Basis (radix)
als zweiten Parameter mit angeben:
`parseInt("08", 10)`



Zahlen (forts.)

- NaN = Not a Number
 - „ansteckend“ bei Berechnungen
 - ungleich zu allem anderen (inklusive NaN)
 - testen mit `isNaN()`
 - `typeof NaN` \Rightarrow "number" 😊



Zahlen (forts.)

- `Infinity` = unendlich,
 - z. B. `1/0` \Rightarrow `Infinity`
 - größer als die größte gültige Zahl:
`Infinity` > `Number.MAX_VALUE`
 - „ansteckend“ bei Berechnungen, aber
 - `1/Infinity` \Rightarrow `0`
 - `Infinity/Infinity` \Rightarrow `NaN`



Boolean

- ! logisches Nicht
- && logisches Und
- || logisches Oder
- „lazy evaluation“: nur berechnen, wenn (noch) nötig
- Beispiel:
var a = 1; true || (a = 2); a \Rightarrow a ist noch 1



„Falsche“ Werte

- Folgende Werte sind falsch (falsy, false):
 - false
 - null („leer“, bewusst gesetzt)
 - undefined (uninitialisiert oder nicht vorhanden)
 - "" (leere Zeichenkette)
 - 0
 - NaN
- Alle anderen Werte sind wahr (true), auch
 - "0"
 - "false"



Vergleiche

- `==` Gleichheit ggf. nach Typumwandlung
- `!=` Ungleichheit ggf. nach Typumwandlung
- `===` Gleichheit ohne Typumwandlung
- `!==` Ungleichheit ohne Typumwandlung
- Beispiele
 - `"1" == 1` \Rightarrow wahr
 - `"1" === 1` \Rightarrow falsch



Vergleiche (forts.)

- `switch`
 - Vergleich ohne Typumwandlung
- `a ? b : c`
 - ternärer Operator
 - wenn `a` wahr ist, ist das Ergebnis `b`, sonst `c`
 - Beispiel:
`(typeof a !== "undefined") ? a : "Vorgabe"`



Default-Operator: ||

- || ist logisches Oder
- Beispiel

```
var value = arg || "vorgabewert";
```
- wenn arg wahr ist (nicht falsy), wird value arg zugewiesen, sonst "vorgabewert"
- Vorsicht, wenn false, null, "", 0 gültige Werte sein können!
- nützlich bei optionalen Parametern einer Funktion



Guard-Operator: &&

- && ist logisches Und
- Beispiel:
return obj && obj.member;
- gibt obj.member zurück, wenn obj wahr ist (nicht falsy), sonst obj
- verhindert Fehler bei Zugriff auf undefined.member
- Vorsicht, wenn false, null, "", 0 gültige Werte sein können!



Noch mehr Operator-Magie

- + als unärer Operator wandelt Strings um in Zahlen:
`+"1" === 1`
- !! wandelt um in Boolean:
`!!"1" === true`



Arrays

- Arrays sind Objekte (erben von Object)
- haben `length`-Attribut, immer eins größer, als größter (Ganzzahl-)Index
- `length` kann auch gesetzt werden, um das Array zu vergrößern – es wird mit `undefined` aufgefüllt – oder zu verkleinern – es wird dann abgeschnitten.
- Array-Literale: `["eins", "zwei", "drei"]`
- Strings lassen sich nutzen wie Zeichen-Arrays:
`var text = "abcd"; text[2] ⇒ "c"`



Arrays (forts.)

- nützliche Methoden des Array-Objekts:
 - `push` und `pop`, um Array als Stack zu nutzen
 - `sort`: sortiert Array
 - `join(string)`: verbindet Elemente zu einem String
 - `slice(anfang, ende)`: ergibt Teil-Array, lässt Ursprungs-Array unverändert
 - `splice(anfang, ende, weitere Parameter...)`: ergibt Teil-Array, ersetzt Bereich im Ursprungs-Array durch die weiteren Parameter oder entfernt sie so, dass keine Lücke bleibt



Vorsicht Falle: delete

- `delete arr[2]`
 - löscht das 3. Element,
genauer: setzt es auf `undefined`, es bleibt ein Loch:

```
var arr = ["eins", "zwei", "drei", "vier"];  
delete arr[2];  
arr ⇒ ["eins", "zwei", undefined, "vier"]
```
 - `arr.splice(2, 1)`
entfernt das 3. Element und nachfolgende Elemente
rücken auf

```
var arr = ["eins", "zwei", "drei", "vier"];  
arr.splice(2, 1);  
arr ⇒ ["eins", "zwei", "vier"]
```



typeof-Operator

- Die Ergebnisse von `typeof` helfen nicht immer weiter

Typ	Ergebnis von <code>typeof</code>
<code>object</code>	<code>"object"</code>
<code>function</code>	<code>"function"</code>
<code>array</code>	<code>"object"</code>
<code>number</code>	<code>"number"</code>
<code>string</code>	<code>"string"</code>
<code>boolean</code>	<code>"boolean"</code>
<code>null</code>	<code>"object"</code>
<code>undefined</code>	<code>"undefined"</code>



for-Schleifen

- `for (init; vergleich; inkrement) {`
 `...`
}
- alle drei Elemente können durch Komma (!) getrennt mehrere Anweisungen enthalten, z. B.
`for (`
 `var i = 0, result = "";`
 `i < 100;`
 `i++, result += i + "\n"`
 `) {...}`



Vorsicht Falle: with

- `with (obj) {
 member = "wert";
}`
- wenn obj das Attribut member enthält, wird dieses gesetzt
- sonst eine globale Variable namens member!



Vorsicht Falle: Semikolon-Einfügung

- Wenn ein Fehler auftritt, versucht der Interpreter es noch einmal, nachdem er vor dem nächsten Zeilenumbruch ein Semikolon eingefügt hat.

- Beispiel:

```
return { status: "ok };
```

gegenüber

```
return // hier wird ein Semikolon eingefügt  
{ // wird als Block interpretiert  
  status: "ok" // status ist ein Label und  
              // Semikolon wird eingefügt  
}; // Block wird nie ausgeführt
```

- Das kann auch Fehler maskieren.



Fehlerbehandlung try – catch – finally

- Versuch macht klug... Und bei einem Fehler kann man immer noch reagieren...

- Beispiel:

```
try {  
  // hier könnte ein Fehler auftreten  
} catch (e) { // e ist ein Error-Objekt  
  // irgendwie auf Fehler reagieren  
  alert(e.name + ": " + e.message);  
} finally {  
  // wird immer ausgeführt,  
  // z. B. Ressourcen freigeben  
}
```



Fehlerbehandlung (forts.)

- Error hat Unterklassen, z. B. RangeError
- throw erzeugt (feuert) Fehler
- Beispiel:

```
throw new Error("Nachricht");  
throw {  
  name: "Name",  
  message: "Nachricht"  
}
```
- `window.onerror = myErrorHandler`
setzt Fehlerbehandlungsfunktion im Browser, die dann bei jedem Fehler aufgerufen wird



Referenzen

- Variablen enthalten Referenzen
- Funktionsparameter immer Referenzen, nicht Werte



Funktionen sind Objekte

- Funktionen sind Daten.
 - Funktionen können in Variablen gespeichert werden.
 - Funktionen können als Argumente übergeben werden.
- Funktionen sind Objekte und können Attribute und Methoden (also innere Funktionen) haben.



Definition von Funktionen

- `function f (arg) { ... }`
ist Abkürzung für
`var f = function (arg) { ... }`
- anonyme Funktion
`function (arg) { ... }`
- innere Funktion
`function f() {
 function g() { ... }
}`
- `new Function(args, body)`
`f = new Function("a", "b", "return a + b");`
`f.toString()`



Source-Code einer Funktion

- `f.toString()` gibt die Definition, also den Source-Code der Funktion `f` aus
- nicht bei vordefinierten Funktionen und Methoden
- Beispiel:
`Object.toString.toString()`
⇒ `"function toString() { [native code] }"`

```
function f() {  
  alert(new Date());  
};
```

```
f.toString()  
⇒ "function f() { alert(new Date); }"
```



Funktionsparameter

- Deklaration enthält Liste der erwarteten Parameter
- Parameter sind lokale Variablen der Funktion
- Zugriff auf alle Parameter mit `arguments`, einem Array-ähnlichen Konstrukt

- Beispiel:

```
function sum() {  
  var i, result = 0;  
  for (i = 0; i < arguments.length; i++) {  
    result += arguments[i];  
  }  
  return result;  
}  
sum(1, 2, 3, 4, 5) ⇒ 15
```



Funktionsparameter (forts.)

- arguments hat Attribut length, aber nicht die anderen Array-Methoden.
- Ausweg: `Function.prototype.apply()`

```
function removeFirstArgument() {  
  var slice = Array.prototype.slice;  
  return slice.apply(arguments, [1]);  
} ;  
removeFirstArgument(1, 2, 3, 4)  
⇒ [2, 3, 4]
```
- `arguments.caller` ist Referenz auf aufgerufene Funktion, nützlich bei anonymen Funktionen



Funktionsparameter (forts.)

- Tipp: Objekt als einziger Parameter, dass die benötigten Informationen enthält.
- Beispiel:

```
function showHide(argumentsList) {  
  var node = argumentsList.node || document;  
  var state = argumentsList.state || "show";  
  ...  
};  
showHide({  
  node: document.getElementById("blubb"),  
  state: "hide";  
})
```
- Vorteil: neue Parameter hinzufügen, ohne dass sich die Signatur und damit der aufrufende Code ändert



Funktionsrückgabe

- Funktionen ohne return geben undefined zurück.
- Funktionen mit return x geben x zurück.
- Konstruktoren (später mehr) geben ohne return einen Verweis auf das neue Objekt zurück.
- Konstruktoren mit return x geben x zurück, wenn x ein Objekt ist, sonst das neue Objekt.



Funktionsaufruf

- Funktionen laufen immer in einem Kontext.
- `this` zeigt immer auf den aktuellen Kontext.
- innere Funktionen können nicht auf `this` der übergeordneten Funktion zugreifen
- deshalb Konvention: `var that = this;`
- 4 Arten, Funktionen aufzurufen:



Funktionsaufruf (forts.)

- Funktionsform `f(args)`:
f wird ausgeführt im globalen Kontext.
- Methodenform `obj.f(args)` und `obj["f"](args)`:
f wird ausgeführt mit `obj` als Kontext.
- Vorsicht Falle:
vergisst man das Objekt anzugeben, läuft f im globalen Kontext u. U. fehlerfrei durch



Funktionsaufruf (forts.)

- Konstruktor-Form `new f(args)`:
f läuft im Kontext eines neu erstellten Objekts, das auch Rückgabewert der Funktion ist.
- `f.apply(obj, args)` und `f.call(obj, arg1, ...)`:
f läuft im Kontext von obj.
f braucht dazu nicht Methode von obj zu sein!



Anonyme Funktionen

- als Parameter für andere Funktionen
 - wenn sie nur dafür gebraucht werden
 - innerhalb der aufgerufenen Funktion haben sie dann einen Namen (den des Arguments)
- Beispiel: Call-Back-Funktionen bei Ajax-Aufrufen, bei `setTimeout()` oder `setInterval()`:

```
setTimeout(  
  function() {  
    alert(new Date());  
  },  
  1000  
)
```



Anonyme Funktionen (forts.)

- zum sofortigen, einmaligen Ausführen, z. B. für Initialisierung
- Beispiel: (nächste Seite)



Anonyme Funktionen (forts.)

- mit anonymer Funktion kann man Code „verstecken“
- Beispiel:

```
(function () {  
  var nachricht = "Und Tschüß!";  
  window.onload = function () {  
    alert(nachricht);  
  };  
})(); // Funktion definieren & ausführen
```
- globales Objekt bleibt sauber
- Variable `nachricht` bleibt für die anonyme Funktion erreichbar, ist aber ansonsten unsichtbar und nicht erreichbar (Closure!)



Selbstmodifizierende Funktionen

- Beispiel:

```
f = function() {  
  alert("First time");  
  return function() {  
    alert("Next time");  
  };  
}();  
f();  
f.toString()  
⇒ "function() { alert("Next time"); }"
```
- vorher 2 Alerts: „First time“ und „Next time“
- typische Nutzung: einmalige Initialisierung und dann Code ersetzen mit einer passenden Variante, z. B. für einen bestimmten Browser



Currying

- Man erzeugt zu einer Funktion mit mindestens einem Parameter eine neue Funktion mit mindestens einem Parameter weniger.

- Beispiel:

```
function addGenerator(a) {  
  return function(b) {  
    return a + b;  
  };  
};  
addOne = addGenerator(1);  
addOne(47) ⇒ 48
```



Function-Objekt

- nützliche Attribute und Methoden des Function-Objekts:
 - `length`: Anzahl erwarteter Parameter
 - `caller`: aufrufende Funktion (nicht im Standard)
 - `call(objekt, parameter1, parameter2, ...)`: führt Funktion im Kontext von Objekt aus
 - `apply(objekt, parameterArray)`: wie `call`, nur alle Parameter in einem Array



„eval is evil“

- Performance
- Sicherheit:
nur auf vertrauenswürdige Argumente anwenden
- Das gilt auch für `new Function(args, body)`



alert()

- „Hilfsdebugger“
- blockiert Programmausführung
- Vorsicht bei asynchronen Ajax-Aufrufen



Scopes

- nur zwei Arten von Scopes:
 - global
 - Funktion
- insbesondere kein „Block-Scope“, wie z. B. in Java
- Zugriff auf undeklarierte Variable erzeugt neue globale Variable (z. B. bei Schreibfehler), aber auch function

```
() {  
  a = 1; // ohne var ist a global!  
}
```



Scopes (forts.)

- ein unerwarteter Effekt:

```
var a = 1;
function f() {
  alert(a);
  var a = 2;
  alert(a);
}
f()
```

- Ergebnis: Alerts mit undefined (!) und 2
- 2 Phasen:
 - lokalen Variablen finden und reservieren, dann
 - Block ausführen



Scopes (forts.)

- JavaScript hat lexikalischen Scope.
- Kontext der Funktionen wird bei der Definition gebildet, nicht wenn die Funktion ausgeführt wird.



Kontext

- jedes Objekt hat eigenen Kontext
- globaler Kontext
- Funktionen haben Zugriff auf Attribute und Methoden des Kontexts, in dem sie definiert wurden...
- und weiter „nach oben“ bis zum globalen Kontext.



Kontext (forts.)

- Bei Browsern ist das `window`-Objekt das globale Objekt!
- Beispiel:
`a = 1;`
`alert(window.a);` \Rightarrow 1
- globale Funktionen wie z. B. `parseInt` sind in Wirklichkeit Methoden des globalen Objekts:
`parseInt === window.parseInt`



Kontext (forts.)

- lexikalischer Kontext, also „so wie es im Source-Code steht“, nicht nach Aufrufreihenfolge
- Beispiel:

```
function outer() {  
  var o = 1;  
  function inner() {  
    alert(o);  
  }  
}
```
- Wenn `inner` definiert wird, ist `o` schon bekannt.



Kontext (forts.)

- noch ein Beispiel:

```
function outer() {  
  var o = 1;  
  return (function() {  
    alert(o);  
  })  
};  
var f = outer();  
f() ⇒ 1  
delete outer;  
f() ⇒ 1 (!)
```
- o ist auch noch bekannt, nachdem outer komplett abgearbeitet wurde – sogar nachdem es gelöscht wurde, kann f noch auf o zugreifen!



Closures

- Zugriff auf Attribute und Methoden der übergeordneten Funktion bleibt auch nach deren Ende erhalten.
- Das nennt man Closure.
- für LotusScript- (und Java-)Entwickler wahrscheinlich das ungewöhnlichste Konzept
- Closures sind eines der wichtigsten Sprachmittel von JavaScript!



Vorsicht Falle: Referenz, nicht Wert-Kopie

- f hat eine Referenz auf o , keine Kopie des Werts von o zum Zeitpunkt der Definition.
- Wenn o nachträglich geändert wird, wirkt sich das auch auf f aus.
- Entkoppeln über Vermittler: Funktion, die definiert und gleich ausgeführt wird.



Vorsicht Falle: Referenz, nicht Wert-Kopie (forts.)

- Beispiel mit Fehler:

```
function f() {  
  var a = [];  
  for (var i = 0; i < 5; i++) {  
    a[i] = function() {  
      return i;  
    }  
  }  
  return a;  
};  
var a = f();  
a[0]() ⇒ 5 (sollte 0 sein)
```



Vorsicht Falle: Referenz, nicht Wert-Kopie (forts.)

- verbessertes Beispiel:

```
function f() {  
  var a = [];  
  for (var i = 0; i < 5; i++) {  
    a[i] = (function(x) {  
      return function() {  
        return x;  
      }  
    })(i);  
  }  
  return a;  
};  
var a = f();  
a[0]() ⇒ 0
```



Objekte

- Objekte sind Mengen von Name-Wert-Paaren
- Namen sind Strings, Werte von beliebigen Typen
- entsprechen „assoziative Arrays“ oder „Hash Maps“



Erzeugen von Objekten

- Objekt-Literale wie `{}`
`var object = {};`
- `new`
`var object = new Constructor();`
- `Object.create(...)`
braucht man eigentlich nie



Objekt-Literale und Zugriff auf Mitglieder

- ```
var obj = {
 vorname: "Thomas",
 "Ort der Geburt": "Gehrden",
 "123": 123,
 "@$&%": 1,
 tueWas: function() {
 alert(this.vorname);
 }
};
obj.tueWas()
```
- Dann ist `obj.vorname` gleich "Thomas" und `obj["Ort der Geburt"]` gleich "Gehrden" usw.
- zweite Form notwendig bei reservierten Wörtern



## Objekte ändern

- ```
var obj = {  
  vorname: "Thomas",  
  tueWas: function() {  
    alert(this.vorname);  
  }  
};  
obj.vorname = "Lydia";  
obj.tueWas = function() {  
  alert("Vorname: " + this.vorname);  
};  
obj.tueWas()
```
- Attribute und Methoden können verändert werden.
- Damit ist eine sehr hohe Dynamik möglich – geht nicht bei klassenbasierten Programmiersprachen.



Konstruktoren

- Funktionen können in Kombination mit `new` als Konstruktor verwendet werden.

```
function Fabrik(standort) {  
  this.standort = standort;  
};  
var f = new Fabrik("Hamburg")
```
- Es wird ein neues, leeres Objekt erzeugt und die Funktion in dessen Kontext ausgeführt.
- Vorgabe-Rückgabewert eines Konstruktors ist das neu erzeugte Objekt.
- Wenn ein Konstruktor mit `return x` verlassen wird, wird `x` zurückgegeben, wenn `x` ein Objekt ist, sonst das neue Objekt



Konstruktoren (forts.)

- Objekte haben das `constructor`-Attribut, das auf ihren Konstruktor zum Zeitpunkt der Erzeugung des Objekts zeigt.
- Beispiel erstellt neues, gleichartiges Objekt:

```
function Fabrik(anzahlMitarbeiter) {  
  this.anzahlMitarbeiter = anzahlMitarbeiter;  
};  
var f1 = new Fabrik(100);  
var f2 = new f1.constructor(200);  
f2 ⇒ Object { anzahlMitarbeiter = 200 }
```



Konstruktoren (forts.)

- Konvention: Konstruktoren werden groß geschrieben.
- Vorsicht Falle:
vergisst man `new`, wird die Funktion einfach im globalen Kontext ausgeführt (kein Fehler)
- `instanceof`-Operator vergleicht mit Konstruktor:
`f1 instanceof Fabrik ⇒ true`
`f1 instanceof Object ⇒ true`
`f1 instanceof String ⇒ false`
- `instanceof` ist wahr auch für übergeordnete Objekte



for ... in-Schleife

- Schleife über alle Attribute und Methoden eines Objekts und seiner „Vorfahren“(!)
- Beispiel:

```
for (prop in obj) {  
  alert(prop.toString());  
}
```
- `obj.hasOwnProperty("name")` zeigt, ob `obj` selbst das Attribut `name` hat.



Erweitern von vorhandenen Objekten

- Man kann Objekte auch nach deren Erzeugung noch erweitern...
- auch Objekte, die zur Sprache gehören, wie Function, Object und String!

- Beispiel:

```
String.prototype.trim = function() {  
  return this.replace(/^s+|\s+$/g, "");  
}  
alert("'" + " test ".trim() + "'');
```



Erweitern von vorhandenen Objekten (forts.)

- Vorsicht: Was, wenn es die neue Property in der nächsten JavaScript-Version auch gibt?
- Oder mehrere Entwickler auf die gleiche Idee kommen?
- wenigstens vorher testen
- Beispiel:

```
if (!String.prototype.trim) {  
  String.prototype.trim = function() {  
    return this.replace(/^\s+|\s+$/g, "");  
  }  
}
```



Öffentliche Attribute und Methoden (public)

- Grundsätzlich sind alle Properties, also Attribute und Methoden eines Objekts öffentlich sichtbar.

- Beispiel:

```
function Fabrik() {  
  this.anzahlMitarbeiter = 0;  
  this.produziere = function() {  
    alert(this.anzahlMitarbeiter +  
Mitarbeiten arbeiten hart.");  
  };  
};  
var f = new Fabrik();  
f.anzahlMitarbeiter = 1000;  
f.produziere();
```



Private Attribute und Methoden

- Im Konstruktor definierte lokale Variablen und Funktionen werden zu privaten Attributen und Methoden aller damit erzeugter Objekte.
- Argumente des Konstruktors werden zu privaten Attributen.
- Nur innere Funktionen des Konstruktors können auf private Attribute zugreifen.



Private Attribute und Methoden (forts.)

- Beispiel:

```
function Fabrik(anzahlMitarbeiter) {  
  var anzahlFertigerWaren = 0;  
  var produziere = function() {  
    anzahlFertigerWaren++;  
  };  
};  
var f = new Fabrik();  
f.produziere() ⇒ Fehler
```



Priviligierte Methoden

- Priviligierte Methoden sind öffentlich aufrufbar und haben gleichzeitig auch Zugriff auf private Attribute und Methoden.
- Definition:
`this.privilegedFunction = function() {...}`
im Konstruktor.
- können nicht nachträglich hinzugefügt werden
- funktionieren über Closures!



Privilegierte Methoden (forts.)

- Beispiel:

```
function Fabrik(anzahlMitarbeiter) {
  var anzahlFertigerWaren = 0;
  this.produziere = function(anzahl) {
    anzahlFertigerWaren += anzahl;
  };
  this.getAnzahlFertigerWaren = function() {
    return anzahlFertigerWaren;
  };
}
var f = new Fabrik();
f.produziere(5);
f.getAnzahlFertigerWaren() ⇒ 5
```



„Klassenattribute“

- In Java gibt es den `static`-Modifier für Klassen-Attribute und -Methoden.
- In JavaScript gibt es keine Klassen, aber...
- man kann den Konstruktoren selbst (also den `Function`-Objekten) Attribute hinzufügen und hat damit fast so etwas wie Klassenattribute.



„Klassenattribute“ (forts.)

- Beispiel:

```
var Fabrik = function() {
  Fabrik.anzahlFabriken++;
  var anzahlFertigerWaren = 0;
  this.produziere = function(anzahl) {
    anzahlFertigerWaren += anzahl;
  };
  this.getAnzahlFertigerWaren = function() {
    return anzahlFertigerWaren;
  };
};
Fabrik.anzahlFabriken = 0;
var f1 = new Fabrik();
var f2 = new Fabrik();
Fabrik.anzahlFabriken ⇒ 2
```



Vererbung a la JavaScript

- jedes Objekt hat ein prototype-Attribut, normalerweise `{}`: nutzlos in dieser Form
- damit haben auch Funktionen und insbesondere Konstruktoren dieses Attribut (sind ja auch Objekte)
- Man kann auch neue Properties zum prototype-Objekt hinzufügen oder es komplett ersetzen.



Vererbung a la JavaScript (forts.)

- Suchreihenfolge für Attribute und Methoden:
 - aktuelles Objekt a
 - `a.constructor.prototype`
 - `a.constructor.prototype.constructor.prototype`
 - usw.
 - bis zuletzt Object
- Prototype-Kette
- live, d. h. Änderungen an einem prototype-Objekt wirken sich sofort auf alle Objekte „darunter“ aus



Vererbung a la JavaScript (forts.)

- Properties in einem Objekt „überdecken“ gleichnamige Properties weiter oben in der Kette.
- So können Methoden und Attribute überschrieben (overridden) werden.
- Methoden im Konstruktor werden in jedes erzeugte Objekt kopiert und verbrauchen Speicherplatz.
- Stattdessen kann man sie auch in das prototype-Objekt des Konstruktors schreiben.
- Das spart Speicherplatz, aber kostet Performance (erst Objekt durchsuchen, dann das prototype-Objekt).



Prototypische Vererbung

- Attribute und Methoden, die man dem Prototype-Objekt hinzufügt, werden vererbt an alle damit erstellten Objekte.

- Beispiel:

```
var Fabrik = function() {  
  this.anzahlWaren = 0;  
};  
var f = new Fabrik();  
Fabrik.prototype.produziere =  
  function(anzahl) {  
    this.anzahlWaren += anzahl;  
  };  
f.produziere(10);  
f.anzahlWaren ⇒ 10
```



Prototypische Vererbung (forts.)

- funktioniert auch nach der Erzeugung der Objekte!
im Beispiel: Objekt `f` wird erzeugt, dann erst `produziere` definiert und dem Fabrik-Prototypen zugewiesen
- `produziere` wird nicht direkt in `f`, sondern über die Prototype-Kette gefunden und dann ausgeführt.
- statt
`Fabrik.prototype.produziere = ...`
geht auch
`f.constructor.prototype.produziere = ...`



Prototypische Vererbung (forts.)

- noch ein Beispiel:

```
var Person = function (name) {
  this.name = name;
};
Person.prototype.getName = function () {
  return this.name;
};
var Benutzer = function(name, passwort) {
  this.name = name;
  this.password = passwort;
};
Benutzer.prototype = new Person();
var ich = new Benutzer("Thomas", "geheim");
alert("Benutzer "+ich.getName()+" erstellt");
```



Vorsicht Falle: prototype-Objekt ersetzen

- Wird das prototype-Objekt ersetzt statt ergänzt, wirkt das nur auf danach erzeugte Objekte.
- Intern haben Objekte Zeiger auf das ursprüngliche, echte prototype-Objekt, der nach Erzeugung nicht mehr geändert wird.
- Dieses interne Attribut heißt manchmal `__proto__`.
- Außerdem zeigt das `constructor`-Attribut nach dem Ersetzen des prototype-Objekts manchmal auf das falsche Objekt. Deshalb nach dem Ersetzen des prototype-Objekts am besten immer den `constructor` auch neu setzen.



Vorsicht Falle: prototype-Objekt ersetzen (forts.)

- Beispiel:

```
var Person = function (name) {
  this.name = name;
};
Person.prototype.getName = function () {
  return this.name;
};
var Benutzer = function(name, passwort) {
  this.name = name;
  this.password = passwort;
};
Benutzer.prototype = new Person();
Benutzer.prototype.constructor = Benutzer;
```



Zugriff auf „Superklasse“ mit uber

- kein direkter Zugriff auf „Superklasse“
- Konvention: Attribut uber auf prototype der „Superklasse“ setzen
- Beispiel:
`Benutzer.prototype = new Person();`
`Benutzer.prototype.constructor = Benutzer;`
`Benutzer.uber = Person.prototype;`



Vereinfachung

- Vereinfachung für das ganze Prozedere:

```
function extend(Child, Parent) {  
  var F = function() {}; // leere Funktion  
  F.prototype = Parent.prototype;  
  Child.prototype = new F();  
  Child.prototype.constructor = Child;  
  Child.uber = Parent.prototype;  
}
```
- Verwendungsbeispiel:

```
extend(Benutzer, Person);
```



Klassische Vererbung nachgebaut

- verschiedene Ansätze, die klassische = klassenbasierte Vererbung in JavaScript nachzubauen
 - nach Douglas Crockford
 - nach Markus Nix
 - mit Prototype
 - ...



Klassische Vererbung nach Crockford

- 3 Erweiterungen von Function

```
Function.prototype.method = function (name, func)
{
  this.prototype[name] = func;
  return this; // nützlich für's Verketteten
};
Function.method("inherits", function(parent) {
  this.prototype = new parent();
  return this;
}); // vereinfachte Version ohne uber();
```



Klassische Vererbung nach Crockford (forts.)

- 3 Erweiterungen von Function (forts.)

```
Function.method("swiss", function (parent) {  
  for (var i = 1; i < arguments.length; i += 1) {  
    var name = arguments[i];  
    this.prototype[name] = parent.prototype[name];  
  };  
  return this;  
});
```



Klassische Vererbung nach Crockford (forts.)

- Anwendungsbeispiel

```
var Person = function(name){this.name = name};
Person.method("getName", function() {
  return this.name;
});
var Benutzer = function(name, passwort) {
  this.name = name;
  this.password = passwort;
};
Benutzer.inherits(Person);
Benutzer.method("getPasswort", function() {
  return this.password;
});
var b = new Benutzer("Thomas", "geheim");
b.getName() ⇒ "Thomas"
```



Parasitäre Vererbung

- Rufe im Konstruktor einen anderen Konstruktor auf und gebe das von ihm erzeugte Objekt nach Erweiterung zurück (statt des neu erzeugten Objekts).

- Beispiel:

```
function vorhandenerKonstruktor(a) {...};  
function NeuerKonstruktor(a, b) {  
  var that = new vorhandenerKonstruktor(a);  
  that.weitereMethode = function (b) {...};  
  return that; // statt implizit this  
}  
var obj = new NeuerKonstruktor();  
obj
```



Parasitäre Vererbung

- Nachteil: Änderungen und Erweiterungen des NeuerKonstruktor.prototype werden nicht vererbt
- Beispiel:

```
var obj = new NeuerKonstruktor();  
NeuerKonstruktor.prototype.a = 1;  
obj.a ⇒ undefined
```



Globale Variablen

- Globale Variablen sind böse 😊 und zu vermeiden.
- Probleme, wenn z. B. mehrere Bibliotheken die gleichen globalen Variablen benutzen.
- unvorhersehbare Ergebnisse



Namespaces

- Namensräume helfen, den globalen Kontext sauber zu halten.

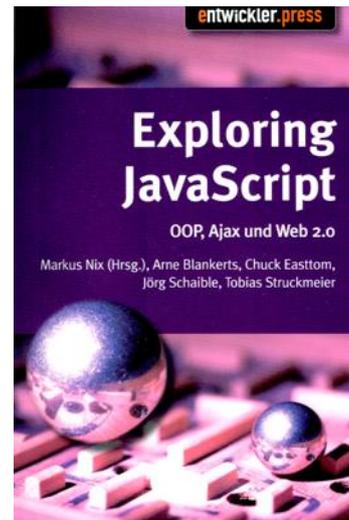
- Beispiel:

```
var de = {};  
de.assono = {};  
de.assono.HtmlHelper = {  
  appendDiv: function(child, parent) {...};  
  ...  
}
```



Quellen

- Markus Nix (Hrsg.), et. al.
"Exploring JavaScript – OOP, Ajax und Web 2.0"
entwickler.press, ISBN 978-3-939084-28-0, 165 S.



- seine Web-Seite: <http://www.markusnix.com/>

Quellen (forts.)

- Douglas Crockford: „JavaScript: The Good Parts“
O'Reilly, ISBN 978-0-596-51774-8, 155 S.

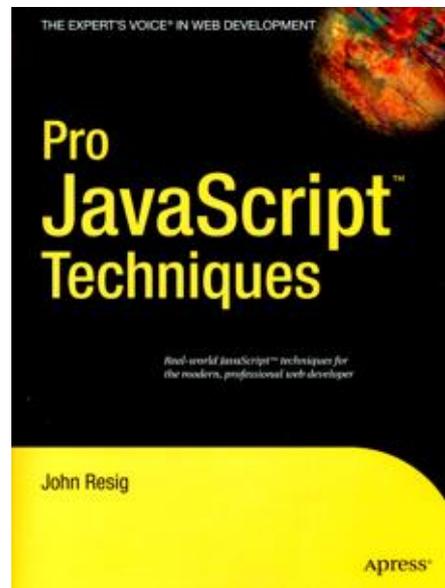


- seine Web-Seite: <http://www.crockford.com/>



Quellen (forts.)

- John Resig: „Pro JavaScript Techniques“
Apress, ISBN 1-59059-727-3, 359 S.

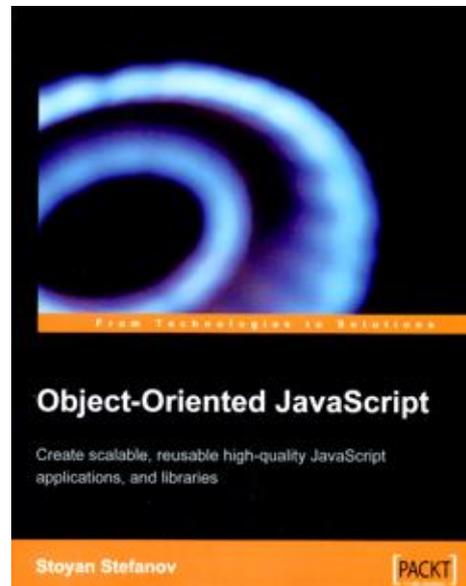


- seine Web-Seite: <http://ejohn.org/>



Quellen (forts.)

- Stoyan Stefanov: "Object-Oriented JavaScript"
Packt Publishing, ISBN 978-1-847194-14-5, 337 S.

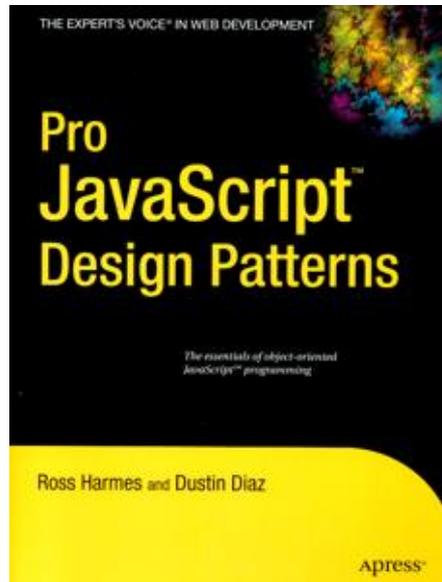


- seine Web-Seite: <http://www.phpied.com/>



Quellen (forts.)

- Ross Harmes, Dustin Diaz:
„Pro JavaScript Design Patterns“
Apress, ISBN 978-1-59059-908-2, 269 S.



- Ross' Web-Seite: <http://techfoolery.com/>
Dustins Web-Seite: <http://www.dustindiaz.com/>



Quellen (forts.)

- Standard ECMA-262
ECMAScript Language Specification
[http://www.ecma-international.org/
publications/standards/Ecma-262.htm](http://www.ecma-international.org/publications/standards/Ecma-262.htm)



Fragen?

- jetzt stellen – oder später:

 tbahn@assono.de

 www.assono.de/blog

 04307/900-401



- Folien unter:
www.assono.de/blog/d6plinks/EC-2014-JavaScript-fuer-Fortgeschrittene