DIWSH DIGITALE WIRTSCHAFT SCHLESWIG-HOLSTEIN CLUSTERMANAGEMENT

assono

Nordic Coding:

„JavaScript - The World's Most Misunderstood Programming Language“

(Douglas Crockford, 2001)

Kiel, 19. August 2011

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
http://www.assono.de/blog
04307/900-401

assono
IT-Consulting & Solutions

Variablentypen

- JavaScript ist „loosely typed“, nicht untypisiert!
- Es gibt folgende 6 Typen:
  - String
  - Number
  - Boolean
  - Object
  - `null`
  - `undefined`

„Falsche“ Werte

- Folgende Werte sind falsch („falsy“):
  - `false`
  - `null` („leer“, bewusst gesetzt)
  - `undefined` (uninitialisiert oder nicht vorhanden)
  - `""` (leere Zeichenkette)
  - `0`
  - `NaN` (Not-a-Number)
- Alle anderen Werte sind wahr (`true`), auch
  - `"0"`
  - `"false"`

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") {
    ...
}
```

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!

Vergleiche

- `==` Gleichheit ggf. nach Typumwandlung
- `!=` Ungleichheit ggf. nach Typumwandlung
- `===` Gleichheit **ohne** Typumwandlung
- `!==` Ungleichheit **ohne** Typumwandlung
- Beispiele
  - `"1" == 1` ⇒ `true`
  - `"1" === 1` ⇒ `false`
- `switch`
  - Vergleich **ohne** Typumwandlung

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() ⇒ Ausgabe: "Vorname: Lydia"
- 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 (**jede** Funktion!):

```
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 (ohne `return`).
- Mit `return x` kann man gezielt ein **anderes** Objekt zurück geben (Beispiel: siehe parasitäre Vererbung).

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(anzahlMA) {
    this.anzahlMA = anzahlMA;
};
var f1 = new Fabrik(100);
var f2 = new f1.constructor(200);
f2 ⇒ Objekt { anzahlMA = 200 }
```

`typeof`-Operator

- `typeof` hilft nicht immer, den „richtigen“ Typ zu bestimmen:

Typ	Ergebnis von `typeof`
`object`	`"object"`
`function`	`"function"`
`array`	**`"object"`**
`number`	`"number"`
`string`	`"string"`
`boolean`	`"boolean"`
`null`	**`"object"`** (**!?!**)
`undefined`	`"undefined"`

instanceof-Operator

- instanceof-Operator vergleicht mit Konstruktor:
  `f1 instanceof Fabrik` ⇒ `true`
  `f1 instanceof Object` ⇒ `true`
  `f1 instanceof String` ⇒ `false`
- `instanceof` ist wahr auch für übergeordnete Objekte

Konstruktoren (forts.)

- **Konvention**: Konstruktoren werden groß geschrieben.
- **Vorsicht Falle**:
  vergisst man new, wird die Funktion einfach im globalen Kontext ausgeführt (kein Fehler)
- Aber es gibt eine "Abwehrtechnik":

```
function Fabrik() {
    if (!(this instanceof Fabrik)) {
        return new Fabrik();
    }
    …
}
```

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 selbst gehören, wie `Function`, `Object` und `String`!
- Beispiel:

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

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, "");
    }
}
```

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", 2, ["drei"]]`
- Elemente können unterschiedliche Typen haben!
- Strings lassen sich nutzen wie Zeichen-Arrays:
  `var text = "abcd";`
  `text[2]` ⇒ `"c"`

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"]
```

Vorsicht Falle: `with`

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

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
}
```

- genau **eine** catch-Anweisung

Fehlerbehandlung (forts.)

- `throw object` wirft Fehler
- `object` beliebig, muss nur 2 Eigenschaften besitzen: `name` und `message`
- Beispiel:

```
throw new Error("Nachricht");
throw {
    name: "Name",
    message: "Nachricht"
}
```

- `window.onerror = myErrorHandler` setzt Fehlerbehandlungsfunktion im Browser, die dann bei jedem Fehler aufgerufen wird

Funktionen sind Objekte

- Funktionen sind **Daten**, sie können in Variablen gespeichert und 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
```

Funktionsaufruf

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

Funktionsaufruf (forts.)

- 4 Arten, Funktionen aufzurufen:
- 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 die Methode `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.onunload = 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, gleich mehr dazu)

Selbstmodifizierende Funktionen

- Beispiel:

```
f = function() {
    alert("First time");
    return function() {
        alert("Next time");
    };
}(); // Funktion wird sofort ausgeführt!
f();
f.toString()
⇒ "function() { alert("Next time"); }"
```

- 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
```

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):

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

Scopes (forts.)

- Beispiel:

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

- Was passiert?

Scopes (forts.)

- Beispiel:

```
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
- und es gibt einen globalen 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.)

- **lexikalischer** Kontext, also „so wie es im Source-Code steht“
- 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 Java-Entwickler wahrscheinlich das ungewöhnlichste Konzept
- Closures sind eines der **wichtigsten Sprachmittel** von JavaScript!

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.
- Sie können nicht nachträglich hinzugefügt werden.
- Priviligierte Methoden funktionieren über Closures!

Priviligierte 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 Eigenschaften zum `prototype`-Objekt hinzufügen oder es komplett ersetzen.

Vererbung a la JavaScript (forts.)

- Suchreihenfolge für Attribute und Methoden:
  - lokale Variablen
  - aktueller Kontext (Objekts 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 werden (override).
- 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");
"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.construtor = Child;
    Child.uber = Parent.prototype;
}
```

- Verwendungsbeispiel:

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

Klassische Vererbung nachgebaut

- verschiedene Ansätze, die klassische = klassen-basierte Vererbung in JavaScript nachzubauen
  - nach Douglas Crockford
  - nach Markus Nix
  - mit Prototype (JavaScript-Bibliothek)
  - mit dojo (JavaScript-Bibliothek)
  - ...

Klassische Vererbung nach Crockford

- 2 Erweiterungen von Function

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

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.passwort;
});
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 Konstruktor1(a) {
    this.x = 1;
};
function Konstruktor2(a, b) {
    var that = new Konstruktor1(a);
    that.m = function (b) { alert(b) };
    return that; // statt implizit this
}
var obj = new Konstruktor2();
obj ⇒ Konstruktor1 { x=1, m=function()}
```

Parasitäre Vererbung (forts.)

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

```
var obj = new Konstruktor2();
Konstruktor2.prototype.a = 1;
obj.a ⇒ undefined
Konstruktor1.prototype.b = 2;
obj.b ⇒ 2
```

Binding

- Mit `function.call` und `function.apply` kann mann Methoden von anderen Objekten "borgen" (wie beim `arguments`-Objekt mit `function.slice`).
- Mit folgender Funktion kann man eine permanente Bindung erzeugen:

```
function bind(o, m) {
    return function () {
        return m.apply(o,
                       [].slice.call(arguments));
    };
}
```

- Verwendung:

```
var newObj = bind(oldObj, anotherObj.method);
newObj.method();
```

Attribute und Methoden kopieren

- Statt Eigenschaften zu (ver)erben, kann man sie auch einfach aus einem anderen Objekt **kopieren**:

```
function extend(parent, child) {
    var prop;
    child = child || {};
    for (prop in parent) {
        if (parent.hasOwnProperty(prop)) {
            child[prop] = parent[prop];
        }
    }
    return child;
}
```

- Verwendung:

```
var Benutzer = extend(Person);
```

Mix-Ins

- Das geht natürlich auch mit **mehreren** Quellobjekten, womit man eine **Mehrfachvererbung** realisieren kann:

```
function mix() {
  var arg, prop, child = {};
  for (arg=0; arg<arguments.length; arg+=1) {
    for (prop in arguments[arg]) {
      if (arguments[arg].
          hasOwnProperty(prop)) {
        child[prop] = arguments[arg][prop];
      }
    }
  }
  return child;
}
```

Namespaces

- Es gibt keine Pakete (package), aber Namensräume helfen, den globalen Kontext sauber zu halten.
- Beispiel:

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

Fragen?

jetzt stellen – oder später:

tbahn@assono.de
http://www.assono.de/blog
04307/900-401

assono
IT-Consulting & Solutions

Folien unter:
www.assono.de/blog/d6plinks/NordicCoding-JavaScript

viel ausführlichere Version (125 Seiten!) unter:
www.assono.de/blog/d6plinks/
UKLUG-2011-JavaScript-Blast