OOP mit JavaScript

1. Ein wichtiger Hinweis

Bezeichnungen

Nachdem ich an der einen oder anderen Stelle im Internet ziemlich laute Kritik an diesem Tutorial mitbekommen habe, muss ich vorerst etwas klarstellen. Ich arbeite hier mit Ausdrücken aus anderen Sprachen wie zum Beispiel PHP oder Java.

Ich rede in diesem Tutorial also des Öfteren von Methoden. Früher habe ich sie als Funktionen bezeichnet, aber das hat bei vielen für Verwirrung gesorgt, weil sie an die sonst üblichen Bezeichnungen gewöhnt sind. Darum verwende ich auch weiterhin diesen Begriff, auch wenn es in JavaScript genau genommen keine Methoden gibt. Die heißen wohl Funktionsobjekte, aber so sicher bin ich mir da nicht. Dasselbe gilt natürlich auch für die Begriffe "privat" und "öffentlich", die in diesem Zusammenhang ebenfalls nicht immer hundertprozentig stimmen.

Also

... gehe ich bewusst diesen Weg, da er nach meinen bisherigen Erfahrungen die verständlichste Vorgehensweise ist. Denkt aber immer daran, dass die hier verwendeten Ausdrücke genau genommen **FALSCH** sind. Ich mache das nur, damit die Umsteiger von anderen Sprachen es leichter haben, das teilweise seltsam anmutende Verhalten von JavaScript zu verstehen.

2. Merket auf und lauschet wohl

... ihr Leute, die ihr aus den "klassischen" Programmiersprachen kommt und mit bestimmten Prinzipien der OOP vertraut seid. Denn was ich euch hier zeige, hat eigentlich gar nichts mit dem zu tun, was ihr so kennen gelernt habt. Denn JavaScript geht hier einen völlig anderen Weg. Das liegt (auch) daran, dass sie zu den so genannten Prototypenbasierten Sprachen gehört.

Eben wegen dieser zum Teil großen Unterschiede gibt es seit Jahren einen fürchterlichen Streit darüber, ob JavaScript nun objektorientiert oder nur objektbasiert ist. Ich persönlich tendiere eher zu Letzterem, da es im Vergleich zu anderen Sprachen etliche Dinge nicht gibt. Auch ist die Art der Implementierung doch reichlich gewöhnungsbedürftig und wirkt hier und da auch recht krude.

Wichtig

Was das folgende Tutorial angeht, so sei euch eines gesagt. Den Krempel zu verstehen ist eine Sache. Den Mist auch noch vernünftig zu erklären eine ganz andere. Ich hoffe aber trotzdem, dass ihr die folgenden Erläuterungen begreifen werdet.

GANZ WICHTIG!

Da die Objektorientierte Programmierung unter JavaScript eine Geschichte für sich ist, solltet ihr euch dieses Tutorial wirklich erst **KOMPLETT** durchlesen, bevor ihr euch daran wagt, die erworbenen Kenntnisse praktisch umzusetzen. Und wenn ihr euch an meine Praxistutorials herantraut, so steht denen ein wenig skeptisch gegenüber. Denn da zeige ich euch nicht alles was möglich ist, sondern konzentriere mich nur auf bestimmte Dinge.

3. Objektorientiert und prozedural

In PHP gibt es da eine saubere Trennung. In JavaScript dagegen existiert zwischen beiden Universen eine Art von Hyperraum, wo sie sich treffen, um sich zu einer neuer Dimension zu vereinigen. Klingt ziemlich nach Star Trek, aber anders kann ich das leider nicht erklären. Außerdem wurde darauf in Star Wars nicht eingegangen.

4. Vorkenntnisse

Hier gibt es zwei Möglichkeiten. Erstens, ihr habt null Plan von OOP. Dann werdet ihr das Konzept recht schnell begreifen. Seid dann aber für anderen OOP-basierte Sprachen ein für alle Mal verdorben. Zweitens, ihr kennt die OOP schon aus anderen Sprachen. Dann werdet ihr erst mal gucken wie eine Kuh wenns donnert. Aber vielleicht begreift ihr es doch und könnt euch somit in zwei Welten bewegen.

Für den ersten Einstieg

Wer noch absolut keine Ahnung von OOP hat, sollte sich vielleicht mal meine mittlerweile recht bekannte Einführung in OOP mit PHP5 ansehen. Allerdings reichen da die ersten vier Unterpunkte der Grundlagen völlig. Es geht eh erst mal nur darum, dass ihr mit den verwendeten Begriffen etwas anfangen könnt.

5. Wichtig

Objekte

Im Gegensatz zu PHP ist in JavaScript eigentlich (fast) alles ein Objekt. Eine Funktion ist ein Objekt, eine Funktion in einer Funktion auch. Und das macht den meisten Leuten Angst, wenn sie zum ersten Mal mit OOP unter JavaScript konfrontiert werden. Zumindest denen aus der PHP-Ecke. Java-Entwickler ticken da wohl ein wenig anders. Dachte ich zumindest, bis ich ein paar Dankesmails von einigen bekam, die mit JavaScript auch ihre liebe Not und Mühe hatten.

OOP mit JavaScript - Grundlagen

1. Der feine Unterschied

Im Gegensatz zu PHP (und auch Java!) benötigt man bei JavaScript keine Klasse, um daraus ein Objekt zu erzeugen. Da gelten ein paar andere Regeln, die ich zuerst vorstellen möchte.

Konzept	Klassisches OOP-Modell	OOP-Modell JavaScript
Entwurfsmuster	Klasse	Konstruktorfunktion (*)
Objekt wird erzeugt aus	Klasse	Konstruktorfunktion (*) oder als eigenes Objekt
Objekt benötigt Klasse	Ja	Nein
Vererbung	Über Klassen	Über Prototypen/Aggregation
Methoden	Ja	Ja
Eigenschaften	Ja	Ja
Sichtbarkeitsstufen für Methoden und Eigenschaften	private protected public	private public (mit Einschränkungen)
Statische, finale, abstrakte Klassen/Methoden/Eigenschaften, Interfaces	Ja	Nö, können teilweise nachgebildet werden

(*) Der Ausdruck Konstruktorfunktion ist auf den ersten Blick ziemlich seltsam. Und auf den zweiten auch. Denn dabei handelt es sich dabei nicht immer um einen Konstruktor im eigentlichen Sinn, aber das scheint bei JavaScript egal zu sein. Oder auch nicht, ist mir mittlerweile ziemlich wurscht. Egal, dieser Ausdruck scheint üblich zu sein, darum benutze ich ihn hier auch konsequent, ob er nun korrekt ist oder nicht. Persönlich bevorzuge ich die Bezeichnung "Pseudoklasse". Aber ich lass es mal lieber, sonst nörgelt wieder jemand rum.

2. Einzelobjekte erzeugen

Das geschieht mit dem bekannten Wörtchen `new`. Im Gegensatz zu PHP benötigt man dafür keine Klasse, so lange man mit `Object` arbeitet. Im folgenden Beispiel erzeugt man ein leeres Objekt mit der Referenzvariablen `obj`.

```
// Einzelnes neues Objekt erzeugen
var obj = new Object();
```

Eigenschaften

Will man nun dem Objekt eine Eigenschaft verpassen, so geschieht das ganz einfach damit.

```
obj.eigenschaft = 'blubber';
```

Methoden

Ist auch kein Problem, dazu reicht der folgende Code aus.

```
obj.methode =  function() {
  alert ('hallöle');
}
```

3. Der Zugriff

Will man nun auf eine Eigenschaft oder Methode zugreifen, so geschieht das wie gehabt über die Referenz-/Instanzvariable plus dem Namen. Beide sind also komplett öffentlich zugänglich.

```
alert (obj.eigenschaft);
obj.methode();
```

Zugriff auf Eigenschaften in Methoden

Hier gibt es zwei Möglichkeiten. Entweder greift man auf die Referenz zu oder man benutzt `this`. Die Wirkungsweise ist identisch, nur das Prinzip anders.

```
var obj          = new Object;
obj.eigenschaft = 'blubb';
obj.methode      =  function()
{
  // entweder
  alert (obj.eigenschaft);
  // oder
  alert (this.eigenschaft);
}
obj.methode();
```

Erläuterung

Da bei dieser Vorgehensweise alle Eigenschaften öffentlich zugänglich sind, greift in der ersten Variante `alert (obj.eigenschaft);` die Methode einfach über die Instanz/Referenz darauf zu. Bei der zweiten Möglichkeit wird dagegen `this` benutzt. Das ist genau wie in PHP ein Verweis auf das eigene(!) Objekt. Allerdings gibt es dabei noch ein paar Überraschungen für euch, wenn man mit `this` in einem anderen Zusammenhang arbeitet. Aber dazu komme ich später.

4. Objektgebundenheit

Bei dieser Vorgehensweise ist sowohl die Eigenschaft als auch die Methode an das erzeugte Objekt gebunden. Erzeugt man ein neues Objekt unter demselben Namen, so werden alle bereits existierenden Eigenschaften und Methoden ins Nirwana befördert und euch haut es eine saftige Fehlermeldung um die Ohren.

```
var obj          = new Object ();
obj.eigenschaft = 'blubb';
obj.methode      = function ()
{
  alert (obj.eigenschaft);
};
// Funktioniert nicht, da ein neues Objekt erzeugt wurde
var obj = new Object ();
obj.methode();
alert (obj.methode);
```

Aber

Andere Objekte haben trotzdem Zugriff auf die Eigenschaften und Methoden und zwar über die Referenz- beziehungsweise Instanzvariable.

```
var obj_1          = new Object ();
obj_1.eigenschaft = 'blubb';
obj_1.methode      = function ()
{
  alert (obj_1.eigenschaft);
};
var obj_2      = new Object();
obj_2.methode = function ()
{
  alert (obj_1.eigenschaft);
}
obj_2.methode();
```

OOP mit JavaScript - Objektliterale

1. Schreibweisen

Im vorherigen Abschnitt habe ich euch gezeigt, wie man Objekte auch ohne Klasse in JavaScript erzeugen kann. Und was jetzt kommt, ist eigentlich nur eine andere Schreibweise. Anstelle von

```
var obj          = new Object;
obj.eigenschaft = 'blubb';
obj.methode      = function()
{
  alert (this.eigenschaft);
}
```

kann man das auch so schreiben.

```
var obj =
{
  eigenschaft : 'blubb',
  methode     : function()
  {
    alert (this.eigenschaft);
  }
}
```

Erläuterung

Bei dieser Schreibweise erzeugt man ein Objekt durch `var obj = { ... }`. Damit wird es sozusagen "eingezäunt". Die Eigenschaften und Methoden werden ganz normal durch einen Namen deklariert. Danach folgt ein Doppelpunkt.

Wichtig

Bei den Eigenschaften muss immer ein Wert angegeben oder ein neues Objekt erzeugt werden. Notfalls arbeitet man mit einem Leerstring.

```
eigenschaft_1 : 'blubber',
eigenschaft_2 : '',
eigenschaft_3 : new Array(),
eigenschaft_4 : new Date(),
```

2. Ein beliebter Fehler

Hinter jeder Eigenschaft und Methode muss ein Komma stehen. Das entspricht bei dieser Schreibweise dem sonst üblichen Semikolon. Letzteres kann man teilweise weglassen, aber hier führt ein fehlendes Komma unweigerlich zu einer Fehlermeldung.

Einzige Ausnahme

Bei der letzten Anweisung in Form einer Eigenschaft oder Methode kann man es weglassen. Aber wirklich nur bei der Allerallerletzten! Und wenn man den Internet Explorer 6 noch unterstützt, so muss man es weglassen! Denn sonst zickt der rum.

3. Zugriffsrechte

Innerhalb des Objektes gibt es wie schon gesagt zwei Zugriffsmöglichkeiten. Entweder über den internen Zeiger `this` oder über den Referenz-/Instanznamen. Mein Tipp dazu, nehmt `this`. Da ist der Code sauberer, da man sofort erkennt, dass es sich um die Eigenschaften und Methoden des jeweiligen Objektes handelt.

```
var obj =
{
  eigenschaft : 'blubb',
  methode_1   : function()
  {
    alert (this.eigenschaft);
  },
  methode_2   : function()
  {
    this.methode_1();
  }
}
obj.methode_2();
```

Objektgebundenheit

Hier gilt dasselbe, was ich schon an dieser Stelle unter Punkt 4 gesagt habe. Die Eigenschaften und Methoden gelten nur für ein spezielles Objekt. Allerdings können andere Objekte ebenfalls darauf zugreifen.

```
var obj_1 =
{
  eigenschaft : 'blubb',
  methode     : function()
  {
    alert (this.eigenschaft);
  }
}
var obj_2 =
{
  methode     : function()
  {
    obj_1.methode();
  }
}
obj_2.methode();
```

OOP mit JavaScript - Objektliterale - Verschachtelungen

1. Eigenschaften

Sie können beliebig tief aufgebaut sein. Man muss sich das dann als eine Art von Array vorstellen, auch wenn die Verarbeitung ein wenig anders ist. Auf die Besonderheiten gehe ich im nächsten Abschnitt ein.

```
var daten = {
  person : {
    vorname    : 'Peter',
    name       : 'Kropff',
    geschlecht : 'männlich'
  }
}
```

Der Zugriff

... erfolgt ganz normal über die Punktnotation. Innerhalb des Objektes sollte man mit `this` arbeiten, außerhalb davon muss man die Objektreferenz-/instanz angeben.

```
var daten = {
  ...
  methode : function() {
    alert(this.person.name);
  }
}
alert(daten.person.name);
```

2. Methoden

Innerhalb von Methoden kann man zusätzliche Objektliterale erzeugen.

```
var obj_1 = {
  eigenschaft_1 : 'bla',
  methode_1     : function () {
    var obj_2 = {
      eigenschaft_2 : 'jodelblah',
      methode_2     : function () {
        alert (obj_1.eigenschaft_1);
        alert (this.eigenschaft_2);
      }
    }
    obj_2.methode_2();
  }
}
obj_1.methode_1();
```

Erläuterung

Das Literalobjekt `obj 2` ist Bestandteil der Methode `methode 1`. Das heißt, es wurde "gekapselt" und ist nun in der Methode eingeschlossen. Das bedeutet, dass man außerhalb des umschließenden Literals `obj 1` nicht darauf zugreifen kann. Das geht nur innerhalb und auch nur über den Namen des verschachtelten Literals (`obj_2.methode_2();`).

Dasselbe gilt auch anders herum. Will eine Methode von `var obj 2` zum Beispiel auf Eigenschaften von `var obj 1` zugreifen, so geschieht dies ebenfalls über den entsprechenden Namen (`alert (obj_1.eigenschaft_1);`).

Ein Hinweis

Anstelle von `var obj 2` kann man auch mit `this.obj 2` arbeiten. Trotzdem ist der Zugriff nur innerhalb von `obj_1` möglich. Mein Tipp: lasst es bleiben.

3. Kapselung in Eigenschaften

Will man sein Objekt hierarchisch tiefer staffeln, so geht das über Eigenschaften, die zum Beispiel aus weiteren Methoden bestehen. Mein Tipp: übertreibt es nicht. Ich habe es mal gemacht und das Ergebnis war nicht so prickelnd.

```
var obj_1 =
{
  eigenschaft_1 : 'bla',
  eigenschaft_2 :
  {
    eigenschaft : 'jodelblah',
    methode     : function ()
    {
      alert (obj_1.eigenschaft_1);
      alert (this.eigenschaft);
    }
  }
}
obj_1.eigenschaft_2.methode();
```

OOP mit JavaScript - Objektliterale - Besonderheiten

1. Ei verbibbsch

Bekanntlicherweise dürfen in fast allen Programmiersprachen Variablen (Eigenschaften) und Funktionen (Methoden) nicht mit einer Zahl beginnen. Und da macht selbst JavaScript keine Ausnahme.

Denkste!

Eine Ausnahme gibt es doch. Wie könnte es bei dieser Sprache auch anders sein. Nämlich bei den Eigenschaften und Methoden eines Objektliterals. Das sieht zwar ziemlich seltsam aus, funktioniert jedoch einwandfrei.

2. Nummerierung

Analog zu numerischen Indizes bei Arrays kann man das auch auf Literale übertragen. Dabei setzt man als "Bezeichner/Index" einfach eine Zahl, also zum Beispiel das.

```
var obj =
{
  0 : 'blubb',
  1 : 'blubber'
}
```

Nur unterscheidet sich der Zugriff von dem sonst üblichen. Folgender Code wirft natürlich einen Fehler aus,

```
alert (obj.0);
```

während der hier reibungslos funktioniert.

```
alert (obj[0]);
```

Also!

Immer daran denken, bei numerischen Bezeichnungen immer die eckigen Klammern verwenden, sonst knallt es!

3. Anzahl der Einträge

Bei Arrays mit numerischen Indizes kann man in JavaScript jederzeit die Menge ermitteln. Das geschieht über ein einfaches

```
var level = new Array('a','b','c');
alert (level.length);
```

Bei Literalen nicht!

Denn in dem folgenden Fall haben wir eine Eigenschaft, die wiederum zwei weitere beinhaltet. Es handelt sich hier also **NICHT** um eine abnorme Art von Array.

```
var bla =
{
  laber :
  {
    0: 'blubb',
    1: 'blubber'
  }
}
// Ergibt undefined
alert (bla.laber.length);
```

Etwas Grundsätzliches

Mal abgesehen von diesen Seltsamkeiten kann man sich Literale doch als eine Art von Array vorstellen. Und zwar ein assoziatives. Und genau wie bei richtigen Arrays dieser Art durchläuft man ein Literal mit einer `for ... in` -Schleife. Allerdings muss man dann unterscheiden zwischen Eigenschaften und Methoden. Auf die Feinheiten werde ich in einem kommenden Tutorial eingehen.

4. Methoden

Was bei den Eigenschaften schon komisch aussieht, wirkt bei den Methoden völlig absurd. Aber auch hier kann man einfach eine Zahl als Namen angeben. Man muss nur beim Aufruf darauf achten, dass man die eckigen Klammern nicht vergisst.

```
var obj =
{
  0 : function()
  {
    alert ('hallo')
  }
}
obj[0]();
```

5. Und wozu sollte man damit arbeiten?

Gute Frage. Normalerweise überhaupt nicht. Ich sitze gerade aber an einer echt kranken Konzeptprogrammierung, wo ich fast nur damit arbeite. Ist sehr elegant aufgebaut, hochgradig flexibel, lässt sich wunderbar steuern und ist nach spätestens zwei Wochen nicht mehr lesbar. Vielleicht stelle ich euch das mal vor. Aber nur wenn ihr versprecht, nicht die Leute mit den Jacken zu holen, die man hinten zumacht. Die Jacken, nicht die Leute.

OOP mit JavaScript - Konstruktorfunktionen

1. Und nochmal

... der Hinweis

Ich benutze hier sehr oft die Begriffe "Methoden" und "privat". Das mache ich aber nur wegen des Verständnisses. Die Gründe dazu habe ich ja schon hier erläutert.

2. Grundsätzliches

Wie ich zu Beginn schon sagte, kennt JavaScript keine Klassen. Wenn man aber Daten, so wie man es aus anderen Sprachen kennt, sauber kapseln möchte, so ist dies die einzige Möglichkeit, das auf eine gewohnte Art und Weise zu tun. Ich habe mir daher auch angewöhnt, hier von "Pseudoklassen" zu reden.

3. Der Aufbau

Eine Konstruktorfunktion ist zunächst mal nur eine pisselige und stinknormale Funktion, die erst durch die Art des Aufrufs zu diesem besonderen Konstrukt wird.

```
function Konstruktor()
{
  ...
}
var obj = new Konstruktor;
```

Erläuterung

Da das doch ziemlich seltsam aussieht, versuche ich den Code mal so zu erklären. Die Funktion `Konstruktor` stellt eine Art von Container dar, der Methoden und Eigenschaften für ein Objekt bereitstellt. Das ist also vergleichbar mit den sonst üblichen Klassen. Mit `var obj` erstellen wir nun ein Instanz/Referenz des Konstruktors. Wir haben also ein Objekt erzeugt, das Zugriff auf die Eigenschaften und Methoden der Konstruktorfunktionen hat.

4. Der Unterschied

In Sprachen wie PHP gibt es ja die magische Methode __construct. Sie wird automatisch aufgerufen, wenn man beim Erzeugen eines Objektes Parameter an die Klasse übergibt. In JavaScript ist das ein wenig anders. Hier kann man Parameter übergeben, muss es aber nicht. Es ist trotzdem eine Konstruktorfunktion. Auch wenn der Ausdruck doch ziemlich unglücklich ist.

Parameter

Wenn man denn nun welche übergeben möchte, so muss die Konstruktorfunktion sie "bereitstellen". Also im Gegensatz zu PHP so.

```
function Konstruktor(val)
{
  ...
}
var obj = new Konstruktor('blubb');
```

5. Eigenschaften und Methoden

So Leute, jetzt wird es echt kompliziert. Und das ist wohl auch einer der Gründe, warum viele JavaScript nie so ganz begreifen werden. Meine Wenigkeit eingeschlossen. Also! In "normalen" Sprachen gibt es Methoden und Eigenschaften, die über die Sichtbarkeitsstufen `private`, `protected`, `public` und `static` verfügen.

In JavaScript gibt es privilegierte, öffentliche, globale oder anonyme Funktionen, Closures, öffentliche und pseudo-private Eigenschaften sowie eingeschlossene Literalobjekte und was

weiß ich nicht noch alles. Dazu kann man unterschiedlichste Methoden auch noch beliebig tief verschachteln und die Sache mit den Zugriffsrechten ist dementsprechend kompliziert.

6. **`this`**

   Hinzu kommt auch noch, dass die Verwendung von `this` abhängig vom Kontext ist, in der es verwendet wird. Allerdings ist dieses Tohuwabohu das Ergebnis einer völlig idiotischen Spezifikation von ECMA-Script, auf die JavaScript (bekanntlicherweise) aufbaut. Entweder hat da jemand gepennt, sich vorher das Hirn mit zu viel Alkohol weggeballert oder der Typ war ein Sadist. Egal, ich versuche jetzt, euch mit den Prinzipien vertraut zu machen.

OOP mit JavaScript - Konstruktorfunktionen - Eigenschaften

1. Öffentliche Eigenschaften (`public`)

Hier wird jeder Eigenschaft einfach ein `this.` vorangestellt. Das entspricht in etwa dem `public` in PHP. Es können also aus der Konstruktorfunktion erzeugte Objekte darauf zugreifen und sie ändern. Bei den Methoden gibt es ein paar Einschränkungen, aber dazu komme ich später.

```
function Konstruktor(val)
{
  this.eigenschaft_1;
  this.eigenschaft_2 = 'blubb';
  this.eigenschaft_3 = val;
}
var obj = new Konstruktor('blubber');
alert (obj.eigenschaft_2);
```

2. "Private" Eigenschaften (`private`)

Um das zu erreichen, arbeitet man genau so wie bei "normalen" Funktionen. Man klatscht einfach ein `var` davor und der Zugriff außerhalb der Konstruktorfunktion ist nicht mehr möglich. Auch hier gibt es (leider) ein paar Besonderheiten aber dazu äußere ich mich später.

```
function Konstruktor()
{
  var eigenschaft = 'blubb';
}
var obj = new Konstruktor;
// Funzt net
alert (obj.eigenschaft);
```

3. `protected` und `static`

Gibet in JavaScript net. Isso.

4. Objektliterale

... können jederzeit eingebaut werden. Gut, das hat jetzt mit Eigenschaften im eigentlichen Sinne nichts zu tun. Darum weise ich hier auch nur kurz darauf hin und zeige euch die Details hier.

```
function Konstruktor()
{
  var literal =
  {
    ...
  }
}
var obj = new Konstruktor;
```

OOP mit JavaScript - Konstruktorfunktionen - Methoden

1. Tohuwabohu

Im Gegensatz zu anderen Sprachen bietet uns JavaScript eine Fülle von verschiedenen Methoden, die man in einer Konstruktorfunktion einsetzen kann. Genauer gesagt sind es deren fünf. Als da wären folgende.

2. Methoden in Objektliteralen

Dazu sage ich jetzt nichts mehr. Das Prinzip ist dasselbe wie bei Eigenschaften aus dem vorherigen Abschnitt.

3. Privilegierte öffentliche Methoden

Sie werden genau so deklariert wie die öffentlichen Eigenschaften, also mit dem `this` davor. Auf sie kann auch ein Objekt zugreifen, das aus der Konstruktorfunktion erzeugt worden ist. Es gibt auch noch "normale" öffentliche Methoden, aber das erfahrt ihr beim Thema Prototypen.

```
function Konstruktor()
{
  this.methode = function()
  {
    ...
  }
}
```

4. "Private" Methoden

Sie werden wie ganz normale Funktionen erzeugt.

```
function Konstruktor()
{
  function methode()
  {
    ...
  }
}
```

5. "Globale" Methoden

Auch wenn viele Leute sagen, dass es Ausdruck in diesem Zusammenhang garnicht gibt, so verwende ich ihn trotzdem. Dabei dreht man die Notation der privaten Methode einfach um. Auf was hier aber unbedingt achten müsst, zeige ich euch gleich unter dem Abschnitt Besonderheiten.

```
function Konstruktor()
{
  methode = function() {
    ...
  }
}
```

6. "Variablenfunktion"

Ehrlich gesagt habe ich keine Ahnung, wie das wirklich heißt. Aber ich bin im Moment zu faul, nach der richtigen Bezeichnung zu suchen. Zumal ich mich dann wieder durch zigtausend Seiten im Internet wühlen muss, um die Spreu vom Weizen zu trennen. Egal, hier wird einfach eine "globale" Methode genommen und ein `var` vorangestellt.

```
function Konstruktor()
{
  var methode = function()
  {
    ...
  }
}
```

Ein Hinweis

Bei dieser "Funktion/Methode" wird soweit ich das verstanden habe, die Referenz einer anonymen Funktion/Methode in einer Variable abgespeichert. Wenn ich mich irre, so sagt mir einfach Bescheid.

Feinheiten

Normalerweise werden Funktionen/Methoden sofort vom Parser erfasst, auch wenn der Aufruf vorher stattfindet.

```
// ergibt "function"
alert (typeof bla);
function bla() { }
```

Bei einer "Variablenfunktion" kann dies allerdings immer erst dann geschehen, wenn sie vorher(!) "initialisiert" wurde.

```
// ergibt "undefined
alert (typeof bla);
var bla = function() { }
// ergibt „function"
alert (typeof bla);
```

OOP mit JavaScript - Konstruktorfunktionen - Zugriffsrechte

1. Ein Wort zuvor

Hier geht es erst mal nur um die einfachen Dinge, die ihr wohl schon aus anderen Sprachen kennt. Alles andere wie Closures, anonyme und globale Funktionen oder die Seltsamkeiten von `this` kommen noch peu à peu auf euch zu.

2. Zugriffsrechte eines Objekts

Wenn man ein Objekt aus einer Konstruktorfunktion erzeugt, so hat das nur Zugriff auf die öffentlichen Eigenschaften und Methoden. Alles andere ist verboten.

```
function Konstruktor()
{
  this.eigenschaft = 'bla';
  this.methode     = function()
  {
    alert ('hallo');
  }
}
var obj = new Konstruktor;
alert (obj.eigenschaft);
obj.methode();
```

3. Privilegierte öffentliche Methoden

Die dürfen so ziemlich alles. Sie können sowohl auf öffentliche und "private" Eigenschaften oder Methoden zugreifen, auf Objektliterale, "Variablenfunktion" oder "globale" Methoden/Funktionen. Bei öffentlichen Methoden und Eigenschaften muss man darauf achten, das `this` voranzustellen. Ein kleines Beispiel dazu.

```
function Konstruktor()
{
  var privat       = 'blubb';
  this.oeffentlich = 'blubber';
  this.methode     = function()
  {
    alert (this.oeffentlich);
    alert (privat);
    privateMethode();
  }
  function privateMethode()
  {
    alert ('hallo');
  }
}
var obj = new Konstruktor;
obj.methode();
```

4. "Private", "globale" und anonyme Methoden/Funktionen

... können erst mal nur auf Methoden zugreifen, die von ihresgleichen sind. Und ein direkter Zugriff auf Eigenschaften ist nur dann möglich, wenn die ebenfalls privat sind. Ach ja, das folgenden Beispiel kann so nicht funktionieren, da es keine öffentliche Methode gibt, die all die privaten aufruft. Die müsst ihr aber hier selber einbauen (s.o.).

```
function Konstruktor()
{
  var privat = 'blubb';
  function privateMethode()
  {
    alert (privat);
  }
  var anonymeMethode = function()
  {
    globaleMethode();
  }
  globaleMethode = function()
  {
    privateMethode()
  }
}
```

Es gibt zwar einen Weg, auch auf öffentliche Eigenschaften und Methoden zuzugreifen, aber leider hält da JavaScript eine Überraschung für uns auf Lager. Und zu der komme ich jetzt.

OOP mit JavaScript - Konstruktorfunktionen - Besonderheiten

1. Die Sache mit dem `this`

Normalerweise handelt es sich dabei ja um einen Verweis auf das eigene Objekt. Und bei öffentlichen Methoden oder Eigenschaften trifft das sogar zu. Aber irgendein Vollhonk kam wohl im Vollrausch auf die Idee, bei JavaScript (respektive ECMA-Script) eine Ausnahme einzubauen. Der muss wohl genauso zugedröhnt gewesen sein wie der Typ, der das Boxmodell entworfen hat, oder die zweifelhafte Person, die für die Collapsing margins verantwortlich ist.

2. Der Zusammenhang

Wenn man innerhalb einer "privaten"/"globalen"/anonymen Methode/Funktion der entsprechenden Konstruktorfunktion mit `this` arbeitet, so verweist das nicht auf das eigene sondern auf das `window`-Objekt! Wenn ich das genauer ausbaldowert habe, so werde ich natürlich darüber etwas schreiben. Es hat auf jeden Fall damit zu tun, wem die Funktion/Methode gehört.

3. Der Zugriff

Will denn nun eine der oben aufgeführten Methoden auf eine öffentliche Eigenschaft oder Methode zugreifen, so funktioniert das natürlich nicht über `this`. Stattdessen speichert man vorher eine Referenz auf das eigene Objekt in einer Variablen ab. Über die klappt dann auch der Zugriff.

```
function Konstruktor()
{
  var that          = this;
  this.eigenschaft  = 'blubber';
  this.methode      = function()
  {
    privateMethode();
  }
  function privateMethode()
  {
    alert (that.eigenschaft);
  }
}
var obj = new Konstruktor;
obj.methode();
```

Erläuterung

Mit `var that` wird gleich zu Beginn eine Referenz auf das eigene Objekt erzeugt. Die Details dazu kommen später. Viele Leute nehmen anstelle des `that` auch gerne das `self`. Das verweist zwar ebenfalls auf das `window`-Objekt, aber da man das außer bei Frames eh nie braucht, kann man es auch getrost überschreiben. Obercoole "Brothaz from da Hood" nehmen auch gerne ein `thiz` dafür.

Eine Alternative

```
function Konstruktor()
{
  this.eigenschaft = 'blubber';
  this.methode     = function()
  {
    privateMethode.call(this);
  }
  function privateMethode()
  {
    alert (this.eigenschaft);
  }
}
var obj = new Konstruktor;
obj.methode();
```

Erläuterung

Da ich selber damit noch nicht gearbeitet habe, halte ich mich mit meinen Aussagen ein wenig zurück. Mit `call` ruft man eine Funktion/Methode auf. Dabei muss der erste Parameter ein Verweis auf das eigene Objekt sein. Gibt es weitere, so werden die einfach angehängt. In den Parametern der Funktion selber wird er nicht aufgelistet. Die Details dazu liefer ich irgendwann nach.

4. "Globale" Methoden

Jetzt zeige ich euch mal, warum einige die so nennen. Gut andere sagen, dass es so was nicht gibt, aber mir gefällt die Bezeichnung, da sie meiner Meinung nach recht einleuchtend ist. Also, wenn man in einer Konstruktorfunktion eine "globale" Methode erzeugt, so hängt die automatisch im `window`-Objekt drin, sobald eine Referenz/Instanz von der Konstruktorfunktion erzeugt worden ist. Ob so eine Vorgehensweise nun besonders sinnvoll ist, sei mal dahingestellt.

```
function Konstruktor()
{
  var that         = this;
  this.eigenschaft = 'blubb';
  globaleMethode   = function()
  {
    alert (that.eigenschaft);
  }
}
var obj = new Konstruktor;
window.globaleMethode();
```

Erläuterung

Ich hoffe, ich rede jetzt keinen Blödsinn. Aber ich denke, dass es in etwa so abläuft. Zuerst erzeugen wir eine Konstruktorfunktion. Dann wird per `var obj = new Konstruktor;` ein Objekt erzeugt. Und dann hängt wegen des schon erwähnten Vollhonks die Funktion `globaleMethode` im `window`-Objekt.

Trotzdem kann sie über die Referenzvariable `that` noch auf die öffentliche Eigenschaft `eigenschaft` zugreifen. Ach ja, das `window` beim Aufruf kann man sich getrost sparen, es reicht auch ein

```
var obj = new Konstruktor;
globaleMethode();
```

völlig aus. Wichtig ist nur, dass **ZUERST** ein Objekt aus der Konstruktorfunktion erzeugt worden ist. Denn ohne das `var obj = new Konstruktor;` haut es euch eine Fehlermeldung um die Ohren.

5. Ein Tipp

Wer von euch das mit dem `this` nicht glaubt, der kann das ganz einfach mit Firefox (oder anderen modernen Browsern) mittels einem `alert(this)` testen. Denn kann bekommt ihr eine Meldung à la `object Window` ausgegeben. Nur der Internet Explorer 6/7 ist da ein wenig schweigsam und haut euch nur das ach so aussagekräftige `object` um die Ohren.

OOP mit JavaScript - Anonyme Funktionen

1. Vorwort

Einen ersten Einstieg zu dem Thema habe ich euch ja hier schon gezeigt. Kommen wir nun zu den Besonderheiten und Varianten. Ach ja, auf die Sache mit den Event-Handlern bin ich hier schon eingegangen, darum überspringe ich das. Bis auf einen Hinweis, der sehr wichtig ist.

Eine Besonderheit

... stellen bei Event-Handlern die Schleifen da. Denn die werden durchlaufen, bevor man auf ein Element klickt. Und das wirkt sehr irritierend. Eine genauere Beschreibung des Problems inklusive Lösung findet ihr in diesem Praxistutorial.

2. In vorgegebenen Methoden

Hier ist besonders eine Variante interessant, die mich früher immer in den Wahnsinn getrieben hat. Und zwar die Methoden setTimeout und setIntervall des window-Objektes. Denn da muss man den Funktions-/Methodenaufruf in Anführungszeichen oder Hochkommata setzen. Und bei dynamischen Parametern wurde man wahnsinnig ob der Zeichenverknüpfungsproblematik. Aber mit anonymen Funktionen ist das überhaupt kein Problem mehr.

```
function funktion(str)
{
  alert (str);
}
window.setTimeout(
  function()
  {
    funktion('jodelblah');
  }
  , 2000
);
```

Wichtig

Bei `setTimeout` gibt es mal wieder eine dieser meiner Meinung nach völlig idiotischen Regelungen, die einen bei dieser Sprache in den Wahnsinn treiben. Da ist es doch tatsächlich ein großer Unterschied ob man mit `window.setTimeout("funktion()",1000);` (Stichwort Anführungszeichen) oder mit `window.setTimeout(funktion(),1000);` arbeitet. Bei letzterer verweist `this` in der aufgerufenen Methode (mal wieder) auf das `window`-Objekt, während es bei der ersten Variante auf das eigene Objekt zeigt. Sobald ich Zeit dazu habe, werde ich euch das genauer erläutern. Bis dahin muss der Hinweis leider erst mal genügen.

3. Selbstzünder

Will man eine anonyme Funktion automatisch auszuführen ohne sie aufzurufen, so benötigt man dafür eine spezielle Syntax. Fragt mich nicht nach dem Sinn dieser Schreibweise, der interessiert mich nicht. Hauptsache es funktioniert.

```
(
  function () {
    alert('hy');
  }
)();
```

Wenn man so was in eine Konstruktorfunktion packt, so wird dieser Selbstzünder erst dann hochgehen, wenn man Erstere aufruft. Oder genauer gesagt ein Objekt daraus erzeugt.

```
function Konstruktor()
{
  (
    function ()
    {
      alert('hy');
    }
  )();
}
var obj = new Konstruktor;
```

4. Anonyme "Variablenfunktion"

Keine Ahnung, ob die Bezeichnung stimmt, aber ich bin jetzt zu faul, danach zu suchen. Man kann also anonyme Funktionen auch in Variablen abspeichern. Der Aufruf erfolgt dann über den Namen plus die normalen Klammern. Klingt ein wenig unlogisch, aber dieses Gefühl hat man bei JavaScript ja des Öfteren. Ich zumindest. Ach ja, manche bezeichnen so was als "Benannte Funktionsausdrücke". Das aber nur am Rande.

```
var funktion = function()
{
  alert('hallöle');
}
funktion();
```

Innerhalb eines Konstruktors

... gibt es nur eine Möglichkeit. Man kapselt die anonyme Funktion in einer Variablen mittels `var`. Dann ist sie nur innerhalb der Konstruktorfunktion zugänglich.

```
function Konstruktor()
{
  var funktion = function()
  {
    alert('hallöle');
  }
  funktion();
}
var obj = new Konstruktor;
```

OOP mit JavaScript - Prototypen

1. Vererbung mal anders

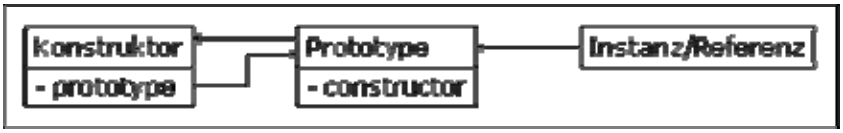
Da JavaScript keine Klassen sondern nur Konstruktorfunktionen kennt, läuft das mit der Vererbung auch ein wenig anders ab. Einfach ausgedrückt bedeutet es, dass man ein Objekt hat und dem dann Stück für Stück Eigenschaften und Methoden hinzufügt oder sie ändert. Dabei gibt es drei Möglichkeiten.

- Erweiterung JavaScript-eigener Objekte
- Erweiterung einzelner Objekte
- Aufbau einer Vererbungshierarchie

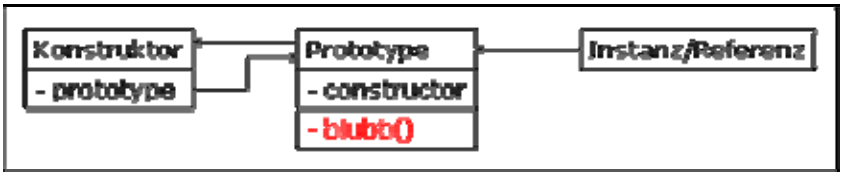
2. Funktionsweise

Jede Konstruktorfunktion wird automatisch mit einer `prototype`-Eigenschaft ausgestattet. Sie beinhaltet einen Verweis auf ein, wie soll ich sagen, virtuelles Objekt mit einer so genannten `constructor`-Eigenschaft, das dann auf den Konstruktor verweist. Und dieses Konstrukt ist dann der eigentliche Prototyp. Wenn man also ein Objekt aus einem Konstruktor erzeugt, so verweist die Referenz-/Instanzvariable nicht direkt auf ihn, sondern genau genommen auf dessen Prototype-Objekt.

Illustration



Wird jetzt nun zum Beispiel von einer Instanz/Referenz aus eine Eigenschaft oder Methode per Prototyping hinzugefügt, so landet die nicht im Konstruktor sondern im Prototype-Objekt. Also so.



Das hier erklärte ist allerdings eher theoretischer Natur, da die Referenz/Instanz eh immer auf den Konstruktor zugreift, wie ihr an den beiden Bildern klar sehen könnt.

3. Und jetzt in einem verständlichen Deutsch

Stellt euch das in etwa so vor, wie man in vielen Industriezweigen ein Produkt entwickelt. Als Beispiel nehme ich mal etwas Martialisches. Einen Panzer! In der Rüstungsindustrie ist es ja gang und gäbe, ein Basismodell zu entwickeln und ausgehend von dem dann verschiedene Arten von Panzern zu bauen. So gab es zum Beispiel bei Y-Tours einen Pionier- und einen Brückenpanzer, die beide auf den Chassis des Leopard 1 basierten. Also geht man in etwa so vor.

- Man nehme einen Basispanzer, der aus einer Wanne besteht, inklusive Kettenantrieb und Motor. Das ist dann der Prototyp.
  - Und aus dem entwickelt man dann verschiedene Varianten
    - einen Kampfpanzer mit Turm und Kanone
    - einen Pionierpanzer, der ein Schäufelchen zum Buddeln hat
    - einen Brückenpanzer mit so einem ausklappbaren Dingsbums

Wie ich schon sagte, ist das Beispiel ziemlich martialisch. Aber hey, ich bin noch in einer Zeit groß geworden, wo sich keiner darüber aufregte, wenn man mit so was gespielt hat. Darum basieren so weit wie möglich auch alle folgenden Erläuterungen darauf. Wer damit ein ethisches Problem hat, soll sich bei Claudia der Weinerlichen ausheulen.

4. Zugriffsrechte

Hier gibt es (ausnahmsweise) mal eine ganz einfache Regel. Man hat nur Zugriff auf die öffentlichen Methoden und Eigenschaften des Prototyps. Alles was privat ist, bleibt verboten.

OOP mit JavaScript - Prototypen - Originäre Objekte

1. Bereits existierende Objekte erweitern

Leider bieten die schon vorgegebenen Methoden von JavaScript nicht die Möglichkeiten, die andere Sprachen haben. Da muss man dann schon mal ein wenig Eigeninitiative zeigen. Greifen wir dazu mal auf die schöne Funktion `easter_date` von PHP zurück. Die liefert das Datum des Ostersonntags eines x-beliebigen Jahres zurück. Leider bietet uns JavaScript das nicht, also muss man selbst Hand anlegen. Wie es bei Singles so üblich ist.

2. Die Methode `easterDate`

... muss man sich von Prinzip her so vorstellen.

```
Date.prototype.easterDate = function()
{
  var easter_date;
 /**
   * Den folgenden Code programmiert mal schön selber!
   * Ich habe keine Lust, da jetzt groß im Kalender rumzuwühlen
   * und das händisch nachzuprogrammieren
  **/
  return easter_date;
}
var datum = new Date;
alert (datum.easterDate());
```

Erläuterung

`Date` ist ein bereits existierendes JavaScript-Objekt, das schon über diverse Eigenschaften und Methoden verfügt. Über `prototype` ergänzen wir es nun um die Methode `easterDate`. Die hängt zwar nicht im eigentlichen Objekt, aber in dessen Prototype-Referenz. Und darauf greifen Instanz-/Referenzvariablen zu. Das Prinzip kann man auch auf Eigenschaften anwenden.

3. Obacht!

Wenn man ein Objekt um eigene Methoden ergänzt, so geschieht Folgendes. Sie werden zusätzlich in das entsprechende Objekt eingehängt. Das funktioniert eigentlich auch ganz wunderbar, nur in einem Fall kann das zu echten Problemen führen. Nämlich beim `Array`-Objekt.

Das Problem

... kann man wunderbar sehen, wenn man mit `prototype.js` arbeitet. Denn in dem Framework geht man so vor. Da werden tatsächlich Eigenschaften und Methoden über `prototype` in das bereits existierende Objekt eingehängt. Und sobald man ein Array mit einer `for...in`-Schleife durchlaufen möchte, erlebt man eine gaaaanz böse Überraschung. Das Problem dazu erläutere ich an dieser Stelle. Es geht zwar nicht um `prototype.js`, aber das Prinzip ist dasselbe.

4. Mein Tipp

Da uns einige vorgegebene Objekte in JavaScript nur sehr wenige Eigenschaften und Methoden bereitstellen, ist es durchaus sinnvoll, sie ein wenig zu ergänzen. Man muss aber höllisch aufpassen! So kann man, wenn man nicht aufpasst, per `prototype` sehr leicht etwas überschreiben oder aber man erlebt unter Umständen eine böse Überraschung.

Darum mein eindringlicher Rat. Wenn ihr das macht, so überlegt euch genau, was ihr tut. Denn diese Vorgehensweise ist genau so, als ob ihr in PHP direkt am Sprachkern rumwerkelt. Und glaubt mir. Die lassen an solche Dinge keine Dödel ran.

OOP mit JavaScript - Prototypen - Einfache Erweiterung

1. Wir bauen einen Panzer

Damit ihr ein Bild davon bekommt, wie Prototypen funktionieren gehen wir folgendermaßen vor. Zunächst mal definieren wir einen Panzerkonstruktor mit einer öffentlichen Eigenschaft und Methode.

```
function Panzer()
{
  this.ps     = 400;
  this.fahren = function()
  {
    alert ('Brumm brumm mit ' + this.ps + ' PS');
  }
}
```

2. Der Prototyp

Da wir bis jetzt noch keinen richtigen Kampfpanzer haben, erweitern wir unser Modell ein wenig. Und das geht so.

```
Panzer.prototype.turm      = true;
Panzer.prototype.kanone    = 120;
Panzer.prototype.schiessen = function()
{
  if (this.kanone && true == this.turm) {
    alert ('Bumm bumm mit ' + this.kanone + 'mm-Kanone');
  }
}
```

Unser Panzer-Prototyp verfügt also nun über die Eigenschaften `ps`, `turm` und `kanone` sowie über die beiden Methoden `fahren` und `schiessen`. Jetzt kann man daraus ein Objekt erzeugen und ein paar Dinge damit machen.

```
var kampfpanzer = new Panzer;
kampfpanzer.fahren();
kampfpanzer.schiessen();
```

Ein Hinweis

Die Methode `schiessen` bezeichnet man nur als öffentliche Methode. Warum? Ganz einfach, weil sie keinen Zugriff auf die privaten Eigenschaften und Methoden von `Panzer` hat. Wenn es sie denn in diesem Beispiel gäbe. Das unterscheidet sie von den privilegierten öffentlichen Methoden. Das aber nur am Rande.

3. Und was soll das jetzt?

Werden sich jetzt viele fragen. Und damit haben sie auch völlig Recht. In diesem Beispiel hätte man getrost alle Eigenschaften und Methoden im Konstruktor `Panzer` bündeln können. Das Ergebnis wäre dasselbe gewesen. Außerdem kann man keine anderen Panzer daraus bauen, die keine Kanone haben, also zum Beispiel einen Pionierpanzer.

Fazit

Ihr seht also selber schon, dass diese einfache Form des Prototypings überflüssig wie ein Kropf(f) ist, weil völlig sinnlos. Ich habe euch das hier auch nur gezeigt, um euch das Prinzip zu erklären. Wirklich sinnvoll wird das erst, wenn man Methoden und Eigenschaften erweitern oder anpassen möchte.

OOP mit JavaScript - Prototypen - Weitervererbung

1. Sinnvoll mit Prototyping arbeiten

Durch das Protyping kann man wunderbar Hierarchien aufbauen, wo man etwas von oben nach unten vererbt. Dazu bauen wir uns zunächst mal verschiedene Panzertypen.

```
function Panzer()
{
  this.ps     = 400;
  this.fahren = function()
  {
    alert ('Brumm, brumm mit ' + this.ps + ' PS');
  }
}
function Kampfpanzer()
{
  this.kanone    = 120;
  this.schiessen = function()
  {
    alert ('Bumm bumm mit ' + this.kanone + 'mm-Kanone');
  }
}
function SpezialPanzer()
{
  this.kanone = 220;
}
function Pionierpanzer()
{
  this.schaeufelchen = true;
  this.buddeln       = function()
  {
    alert ('Buddel, buddel');
  }
}
```

Im Moment ist das noch völlig sinnfrei, da weder der `Kampfpanzer`, der `Pionierpanzer` noch der `Spezialpanzer` fahren können. Das kann nur der `Panzer`, der beherrscht aber sonst nichts. Und das zusammen ist im alltäglichen Geschäft der Militärs doch eher hinderlich.

2. Die Vererbung

... läuft nun folgendermaßen ab.

```
Kampfpanzer.prototype   = new Panzer;
Pionierpanzer.prototype = new Panzer;
SpezialPanzer.prototype = new Kampfpanzer;
```

Erläuterung

Damit vor allem die PHPler begreifen, was wir soeben gemacht haben, müsst ihr euch das in etwa so vorstellen.

```
class Kampfpanzer   extends Panzer {...}
class Pionierpanzer extends Panzer {...}
class SpezialPanzer extends Kampfpanzer {...}
```

`Panzer` vererbt also alle öffentlichen Eigenschaften und Methoden auf `Kampfpanzer` und den `Pionierpanzer`. Und der `Spezialpanzer` erbt alle Eigenschaften und Methoden von `Kampfpanzer` und `Panzer`.

3. Nutzung

```
var supipanzer = new SpezialPanzer;
supipanzer.fahren();
supipanzer.schiessen();

var piopanzer = new Pionierpanzer;
piopanzer.buddeln();
```

Erläuterung

Konzentrieren wir uns zunächst mal auf das Objekt `supipanzer`. Das kann auf alle öffentlichen Eigenschaften und Methoden von `Panzer` und `Kampfpanzer` zugreifen. Es hat also ordentlich PS unter der Haube (gut der Leo 2 hat 1500, aber unser neuer Panzer ist extrem umweltfreundlich und hat einen geringen Spritverbrauch, damit Claudia nicht wieder rumflennt), verfügt über einen Turm und eine ordentliche Wumme, vulgo Kanone.

ABER das Teil hat ein noch größeres Kaliber als der `Kampfpanzer`. Denn der `Spezialpanzer` besitzt sozusagen eine Viagrakanone. Einfach ausgedrückt heißt das, dass wir in diesem Beispiel die Eigenschaft `kanone` von `Kampfpanzer` beim(!) `Spezialpanzer` überschrieben haben.

Der `Pionierpanzer`

... ist auch sehr interessant. Der leitet sich direkt vom `Panzer` ab. Das heißt, dass der keine `kanone` hat. Der besitzt lediglich ein `schaeufelchen` und kann `buddeln`. Wenn ihr versucht, mit dem Gerät auch noch rumzuballern, so führt das zu einem Fehler.

```
// Fehler
pionierpanzer.schiessen();
```

Ein Tipp

Wer von euch zum Beispiel mit Firefox und Firebug arbeitet, kann das Prinzip sehr schön erkennen, wenn man zum Beispiel ein `console.log(supipanzer.prototype)` in den Code einfügt und sich das Ergebnis an der Konsole anschaut.

OOP mit JavaScript - Prototypen - this und that

1. Private Methoden und Eigenschaften

Hier gelten zunächst mal dieselben Regeln wie für alle Konstruktorfunktionen. Nämlich, dass man außerhalb derselben nicht auf die privaten Eigenschaften und Methoden zugreifen kann. Gut, das geht bei Klassen in PHP ja auch nicht. In JavaScript kommt halt noch diese idiotische Sache mit dem `this` und `window` hinzu.

2. `this` und `that`

Innerhalb einer Konstruktorfunktion kann man ja mit einer internen Referenz à la `var that = this;` arbeiten, wenn private Methoden auf öffentliche Eigenschaften oder Methoden zugreifen sollen. Clevere Kerlchen könnten nun auf die Idee kommen und das auch bei Prototypen einsetzen. Anstelle eines `var` wird dagegen ein `this` genommen. Also so.

```
function Kampfpanzer()
{
  this.that     = this;
  var besatzung = 4;
}
function SpezialPanzer()
{
  this.kanone  = 220;
  this.machWas = function()
  {
    alert (this.that.besatzung);
  }
}
SpezialPanzer.prototype = new Kampfpanzer;

var supipanzer          = new SpezialPanzer;
supipanzer.machWas();
```

Denkste!

Leider bekommt man hier das übliche `undefined` um die Ohren genauen. Bei einem `console.log(this.that)` anstelle des `alert` könnt ihr im Firebug an der Konsole auch wunderbar sehen, dass da ein `besatzung` nicht auftaucht.

3. Noch etwas Wichtiges

Wenn man innerhalb(!) einer Konstruktorfunktion mit `var that = this;` arbeitet, so gilt es noch eine Feinheit zu beachten. Dazu nehmen wir den folgenden Code. Ach ja, lasst euch von der zweiten Zeile nicht irritieren, die kann tatsächlich vor den Konstruktorfunktionen stehen. Das hat aber gar nichts mit dem eigentlichen Problem zu tun.

```
// so was funktioniert übrigens auch
SpezialPanzer.prototype = new Kampfpanzer;
function Kampfpanzer()
{
  var that     = this;
  this.kanone  = 120;
  this.ballern = function()
  {
    schiessen();
  }
  function schiessen()
  {
    alert ('Bumm bumm mit ' + that.kanone + 'mm-Kanone');
  }
}
function SpezialPanzer()
{
  this.kanone  = 220;
}

var kampfpanzer = new SpezialPanzer;
kampfpanzer.ballern();
```

Hä!

Ja genau. Hä? Denn im Gegensatz zu dem Beispiel hier liegt das Kaliber nicht bei `220` sondern nur bei `120`. Und auch dieses angebliche Mysterium lässt sich ganz einfach erklären. Denn `this` verweist ja immer auf das eigene(!) Objekt. In diesem Fall also `Kampfpanzer`. Und der hat halt nur so eine kleine Wumme.

OOP mit JavaScript - Prototypen - Zugriffsrechte Teil 1

1. Rauf und runter, hin und her

In JavaScript kann (fast) alles auf alle öffentlichen Eigenschaften und Methoden zugreifen. Das ist genau so, wie man es zum Beispiel aus Sprachen wie PHP kennt. Die Richtung ist also bidirektional.

Ein Beispiel

Hier geht es diesmal nur ums Rumballern. Und das muss jeder Panzer können, ganz gleich um welchen Typ es sich handelt. Darum bauen wir mal eine Methode ein, wo man mit einem MG schießen kann.

```
function Panzer()
{
  this.mgSchiessen = function()
  {
    if (this.mg) {
      alert ('Ratatat mit ' + this.mg + 'mm-MG');
    }
  }
}
```

2. Die Eigenschaft

... `mg` definieren wir aber in den einzelnen Panzertypen, da die MGs dort ein unterschiedliches Kaliber haben sollen. Der `Kampfpanzer` bekommt einen dicken Ballermann, während der `Pionierpanzer` doch eher schwach bestückt ist.

```
function Kampfpanzer()
{
  this.mg = 7.62;
}
function SpezialPanzer()
{

}
function Pionierpanzer()
{
  this.mg = 5.56;
}
```

3. Der Zugriff

Hier setzen wir zunächst wie gehabt das Prototyping ein

```
Kampfpanzer.prototype   = new Panzer;
Pionierpanzer.prototype = new Panzer;
SpezialPanzer.prototype = new Kampfpanzer;
```

... und rotzen dann richtig los mit

```
var supipanzer = new SpezialPanzer;
supipanzer.mgSchiessen();
var piopanzer  = new Pionierpanzer;
piopanzer.mgSchiessen();
```

Erläuterung

Die Objekte, die aus `SpezialPanzer` und `Pionierpanzer` erzeugt wurden, können ohne Probleme auf die Methode `mgSchiessen` des Konstruktors `Panzer` zugreifen. Und der holt sich die Eigenschaft `mg` aus seinen "Kind"-Konstruktoren. Gut, letztere Bezeichnung ist nicht ganz so glücklich, ihr solltet aber wissen, was gemeint ist.

OOP mit JavaScript - Prototypen - Zugriffsrechte Teil 2

1. Ein "Phänomen"

Bei Methoden gilt praktisch dasselbe wie bei den Eigenschaften. Es gibt nur eine klitzekleine Feinheit. Die mich dafür aber einen ganzen Abend beschäftigt hat. Und zwar muss der Aufruf "oben" erfolgen. Um euch das genauer zu erklären, schauen wir uns mal folgendes Beispiel an.

```
function Panzer()
{
  ...
  this.mgSchiessen();
}
function Kampfpanzer()
{
  this.mg           = 7.62;
  this.mgSchiessen = function()
  {
    alert ('Ratatat mit ' + this.mg + 'mm-MG');
  }
}
Kampfpanzer.prototype = new Panzer;
var kampfpanzer       = new Kampfpanzer;
```

Erläuterung

Hier haut es euch eine Fehlermeldung um die Ohren. Denn `Panzer` hat zu diesem Zeitpunkt keine Zugriffsmöglichkeit auf die Methode `mgSchiessen` von `Kampfpanzer`. Denn es erfolgt kein Aufruf von oberhalb. Gut, man sollte in Konstruktorfunktionen meiner Meinung nach eh nicht automatisch Methoden aufrufen, aber das nur am Rande.

Der Grund

... für dieses seltsam anmutende Verhalten erklärt sich aus dem Prototypen-Modell. Denkt mal genau darüber nach und fragt euch auch, was zuerst passiert. Vielleicht kommt ihr selber auf die Lösung.

2. So macht man es richtig

Dazu gibt es zwei Möglichkeiten. Man kann zum Beispiel den Aufruf der Methode `mgSchiessen` in `Panzer` ebenfalls in eine eigene Methode kapseln und die dann über die Referenz aufrufen.

```
function Panzer()
{
  this.rumballern = function()
  {
    this.mgSchiessen();
  }
}
```

```
function Kampfpanzer()
{
  this.mg           = 7.62;
  this.mgSchiessen = function()
  {
    alert ('Ratatat mit ' + this.mg + 'mm-MG');
  }
}
Kampfpanzer.prototype = new Panzer;
var kampfpanzer       = new Kampfpanzer;
kampfpanzer.rumballern();
```

Oder wenn man schon beim Erzeugen eines "Panzer"-Objektes sofort losrotzen will, so ruft man zum Beispiel die Methode in einer "höheren" Konstruktorfunktion auf.

```
function Panzer()
{
  this.ballern = function()
  {
    this.mgSchiessen();
  }
}
function Kampfpanzer()
{
  this.mg = 7.62;
  this.mgSchiessen = function()
  {
    alert ('Ratatat mit ' + this.mg + 'mm-MG');
  }
}
function SpezialPanzer()
{
  this.ballern()
}

Kampfpanzer.prototype   = new Panzer;
SpezialPanzer.prototype = new Kampfpanzer;
var supipanzer          = new SpezialPanzer;
```

3. Ein Hinweis

Manche von euch werden sich jetzt sicher fragen, was der Blödsinn soll. Nicht das "Phänomen", sondern mein Geschreibsel. So programmiert doch keine Sau (oder sollte es zumindest nicht). Außerdem ist es doch ein wenig unwahrscheinlich, dass man in eine solche Situation kommt.

Wohl wahr, aber

... es sei denn, ihr programmiert ziemlich schlampig. Ich selber bin nur durch Zufall beim Ausprobieren darauf gestoßen. Und da ich meinen Auftritt auch als internes Nachschlagewerk missbrauche, habe ich das hier mal zu Papier (HTML) gebracht.

OOP mit JavaScript - Aggregation

1. Uijuijui

Ehrlich gesagt bin ich mir noch nicht mal sicher, ob man das so nennt. Ich habe das an zwei oder drei Stellen im Internet so gefunden und es dann nach Absprache mit einem Kollegen übernommen. Falls jemand weiß, wie das Folgende tatsächlich genannt wird, so möge er/sie mir doch bitte eine Mail schicken. Aber wie sage ich immer so schön.

> *"Namen sind Schall und Rauch"*
> [Quelle: Der olle Goethe (Faust)]

Hauptsache ihr versteht, worum es geht. Denn zu wissen, wie etwas funktioniert, ist meiner Meinung nach wichtiger als zu wissen, wie es heißt. Ach ja, meine "Freundin" aus meinem Lieblingsforum nennt das "enge Kopplung".

2. Die Ausgangssituation

Also, wir haben drei Panzertypen. Den normalen zum durchs Gelände gurken und Felder umpflügen, den Kampfpanzer zum Rumballern und unsere Viagraversion, die vor Kraft kaum laufen kann.

```
function Panzer()
{
  this.ps     = 400;
  this.fahren = function()
  {
    alert ('brumm, brumm mit ' + this.ps + ' PS');
  }
}

function Kampfpanzer()
{
  this.kanone    = 120;
  this.schiessen = function()
  {
    alert ('Bumm bumm mit ' + this.kanone + 'mm-Kanone');
  }
}

function SpezialPanzer()
{
  ...
}

var supipanzer = new SpezialPanzer;
```

3. Das Ziel

Jetzt wollen wir dem hormongestörten `Spezialpanzer` die (öffentlichen) Eigenschaften und Methoden von `Panzer` und `Kampfpanzer` zu Gute kommen lassen und ihm noch ein paar andere Dinge mit auf dem Weg geben.

OOP mit JavaScript - Aggregation - Referenz

1. Die Sache mit den Variablen

Wenn man nun die öffentlichen(!) Eigenschaften von `Panzer` und `Kampfpanzer` innerhalb von `Spezialpanzer` nutzen möchte, so kann man neben dem Prototyping auch mit einer Instanz-/Referenzvariablen arbeiten.

```
function SpezialPanzer()
{
  var panzer      = new Panzer;
  var kampfpanzer = new Kampfpanzer;
}
```

Wichtig!

Mit einem `var panzer = new Panzer;` erzeugt man ein neues Objekt(!) innerhalb der Konstruktorfunktion. Alle Änderungen, die man hier später am `var panzer` vornimmt, beziehen sich nur auf das Objekt und nicht auf das Original.

2. Zugriffsrechte

Über diese Referenz hat nun die Konstruktorfunktion die oben erwähnten Zugriffsmöglichkeiten.

```
function SpezialPanzer()
{
  var panzer      = new Panzer;
  var kampfpanzer = new Kampfpanzer;
  alert(panzer.ps);
  kampfpanzer.schiessen();
}
var supipanzer = new SpezialPanzer;
```

3. Aber

Das gilt aber nicht für das daraus erzeugte Objekt `supipanzer`, da wir die Instanz in einer privaten Eigenschaft gespeichert haben. Um das nun zu erreichen muss man so vorgehen.

```
function SpezialPanzer()
{
  this.panzer      = new Panzer;
  this.kampfpanzer = new Kampfpanzer;
}
```

Dann kann auch das Objekt auf die öffentlichen Eigenschaften und Methoden von `Panzer` und `Kampfpanzer` zugreifen.

```
var supipanzer = new SpezialPanzer;
alert (supipanzer.kampfpanzer.kanone);
supipanzer.panzer.fahren();
```

Wichtig

Nun muss natürlich auch der Zugriff innerhalb der Konstruktorfunktion angepasst werden, da es sonst knallt.

```
function SpezialPanzer()
{
  this.panzer      = new Panzer;
  this.kampfpanzer = new Kampfpanzer;
  alert(this.panzer.ps);
  this.kampfpanzer.schiessen();
}
var supipanzer = new SpezialPanzer;
```

4. Ein Hinweis

Der direkte Aufruf von Eigenschaften und Methoden innerhalb der Konstruktorfunktion ist eigentlich nicht die feine englische Art. Ich habe es hier nur aus Anschauungsgründen gemacht und weil ich von Natur aus tippfaul bin. Also bitte übernehmt diesen Stil nicht.

OOP mit JavaScript - Aggregation - Vererbungsprinzip

1. Ein paar Modifikationen

Zunächst mal wollen wir unseren `Spezialpanzer` ein wenig aufmotzen. Dazu verpassen wir ihm ordentlich PS unter der Haube und geben ihm auch noch was zum Rumballern mit.

```
function SpezialPanzer()
{
  var panzer        = new Panzer;
  panzer.ps         = 1500;
  panzer.mg         = 7.62;
  panzer.schiessen  = function()
  {
    alert ('Ratatat mit ' + this.mg + 'mm-Maschinengewehr');
  }
}
```

Erläuterung

Das Objekt `panzer` erbt hier die Methode `fahren` aus dem Konstruktor `Panzer`. Gleichzeitig wird ihm die Methode `schiessen` und die Eigenschaft `mg` hinzugefügt. Die bereits existierende Eigenschaft `ps` dagegen wird mit einem anderen Wert belegt (dazu gleich mehr).

Und noch mal der Hinweis

Alles, was hier gemacht wurde, bezieht sich ausschließlich auf das Objekt `panzer` innerhalb der Konstruktorfunktion `SpezialPanzer`. Unser Originalpanzer hat noch immer kein MG und nur schlappe 400 PS unter der Haube.

2. Und noch ein paar Modifikationen

```
function SpezialPanzer()
{
  var panzer        = new Panzer;
  panzer.ps         = 1500;
  panzer.mg         = 7.62;
  panzer.schiessen  = function()
  {
    alert ('Ratatat mit ' + this.mg + 'mm-Maschinengewehr');
  }
  var kampfpanzer    = new Kampfpanzer;
  kampfpanzer.kanone = 220;
  this.kaempfen      = function()
  {
    panzer.fahren();
    panzer.schiessen();
    kampfpanzer.schiessen()
  }
}
```

```
var supipanzer = new SpezialPanzer;
supipanzer.kaempfen();
```

Erläuterung

Ich denke, hierzu brauche ich nicht mehr viel zu erzählen. Zunächst mal bekommt die Objektreferenz `kampfpanzer` eine größere Wumme und zum Schluss werden alle Methoden aufgerufen, die man in einem Gemetzel so braucht.

OOP mit JavaScript - Aggregation - Zugriffsrechte

1. Der feine Unterschied

Im Gegensatz zum Prototyping gibt es bei der Aggregation nur eine Richtung, nämlich von oben nach unten. Und das liegt daran, dass man hier innerhalb von Konstruktorfunktionen mit eigenen Objekten arbeitet, die aus anderen Konstruktorfunktionen erzeugt worden sind.

Ein Beispiel

```
function Panzer()
{
  this.mgSchiessen = function()
  {
    alert ('Ratatat mit ' + this.mg + 'mm-MG');
  }
}
function Kampfpanzer()
{
  var pz           = new Panzer;
  this.mg          = 7.62;
  this.rumballern = function()
  {
    pz.mgSchiessen();
  }
}
var kampfpanzer = new Kampfpanzer;
kampfpanzer.rumballern();
```

Erläuterung

Der Zugriff auf die Methode `mgSchiessen` des Konstruktors `Panzer` kann hier nur über die Referenz `pz` innerhalb(!) von `Kampfpanzer` erfolgen. Wenn das Objekt `kampfpanzer` darauf zugreifen möchte, so muss die Referenz öffentlich zugreifbar sein. Also mit `this.pz` und dann einem `kampfpanzer.pz.mgSchiessen`.

2. Aber!

Die Methode `mgSchiessen` funktioniert nicht so, wie sie sollte. Denn als Ergebnis bekommt man ein `Ratatat mit undefinedmm-MG`. Denn der Konstruktor `Panzer` weiß gar nichts vom `Kampfpanzer`. Wie auch? Dasselbe Verhalten gibt es natürlich auch bei den Methoden. Jetzt könnte man auf die Idee kommen und im `Panzer` ebenfalls eine Referenz à la `var kpz = new Kampfpanzer` zu erzeugen, um an die Eigenschaft `mg` zu kommen.

Vergesst es!

Das ist Tinnef, Humbug, Blödsinn hoch drei. Das macht man bei vier oder fünf Konstruktorfunktionen und dann geht alles baden, weil man den Überblick verloren hat. Man dreht sich nur noch im Kreis und irgendwann ist einem schwindelig und man rennt gegen die Wand.

3. Ein Tipp

Wie ihr sehen könnt, sind die Zugriffsrechte beim Prototyping viel einfacher zu handhaben. Allerdings besteht hier immer die Gefahr, dass man aus Versehen etwas überschreibt. Trotzdem ist diese Form der Vererbung grundsätzlich erst mal vorzuziehen.

Mit Aggregationen sollte man dagegen nur in ganz speziellen Fällen arbeiten. Also immer dann, wenn man konsequent von "oben" nach "unten" vererben will.

OOP mit JavaScript - Closures

1. Der Sinn

... von Closures ist mir lange Zeit verschlossen geblieben. Dachte ich zumindest, bis ich mich des Themas angenommen habe. Und oh Wunder, ich selber arbeite damit schon seit Jahren, ohne es zu wissen. Denn das Konzept ist normalerweise(!) recht intuitiv.

2. Garbage Collection

Wie schön, dass man endlich mal an die Leute von der Müllabfuhr denkt. Die sind ja wichtig genug. OK, zurück zum Thema. Garbage Collection bedeutet in der Programmierung, dass automatisch alles aus dem Arbeitsspeicher gelöscht wird, was man nicht braucht.

Bei JavaScript hängt das natürlich von den Browserherstellern ab, aber ich denke, dass die sich mittlerweile an die vorgegebenen Standards halten. Zunächst mal gilt die einfache Regel, dass "private" Eigenschaften oder lokale Variablen innerhalb von Konstruktoren und normalen Funktionen nach dem Aufruf aus dem Speicher gelöscht werden und somit nicht zur Verfügung stehen.

3. Das Prinzip

Bei einem Closure werden die Variablen, oder genauer gesagt die Eigenschaften, eingeschlossen, wenn man in einer Methode darauf zugreift. Sie bleiben also erhalten und können später ohne Probleme wieder aufgerufen werden. Dazu ein Beispiel.

```
function Konstruktor ()
{
  var bla    = 'blubb';
  var anonym = function()
  {
    alert(bla);
  }
  this.methode = function()
  {
    anonym();
  }
}
var instanz = new Konstruktor();
document.getElementById('button').onclick = function()
{
  instanz.methode();
}
```

Erläuterung

Zunächst mal haben wir die private Eigenschaft `var bla` mit dem Wert `blubb`. Nun wird deren äußere (Konstruktor-) Funktion mit `var instanz = new Konstruktor();` aufgerufen. Durch die Garbage Collection sollte daher die private Eigenschaft `bla` aus dem Speicher gelöscht werden.

Wird sie aber nicht

Denn durch den Aufruf in der anonymen Funktion bleibt sie erhalten. Das gilt auch für die "Methode" `var anonym = function()`. Denn sie wurde ebenfalls in einer Variablen gespeichert. Die wird jetzt ebenfalls "gekapselt" und zwar durch den Aufruf in der öffentlichen Methode `this.methode`.

Hä? Wat is?

Ihr müsst euch das so vorstellen. Wir haben die Konstruktorfunktion `Konstruktor`. Die wird aufgerufen über `var instanz = new Konstruktor();`. Der Zugriff auf die Methode `this.methode` erfolgt aber erst, wenn man auf ein HTML-Element mit der ID `button` klickt. Also müssten eigentlich die Eigenschaft `bla` und die in der Variablen `anonym` abgespeicherte anonyme Funktion bereits aus dem Speicher entfernt worden sein. Das geschieht aber nicht, da beide durch den Aufruf erhalten bleiben. Und die Funktionen/Methoden, die dafür sorgen, sind die Closures.

4. Und was soll das?

Nun, ohne Closures würde uns hier alles um die Ohren fliegen. Allerdings gibt es ein paar Feinheiten, zu denen ich jetzt komme.

OOP mit JavaScript – Closures - Feinheiten

1. Ein Beispiel

..., das übrigens von besagtem Arno ist. Ich schmücke mich nicht gerne mit fremden Federn. Kommen wir mal zu der hier schon mehrfach erwähnten Problematik (siehe Tutorials), wo man in einer Schleife einen Event-Handler auf einen Link legt. Der Standardcode sieht dann vom Prinzip her meistens so aus.

```
var links = document.getElementsByTagName('a');
for (var i = 0; i < links.length; i++) {
  links[i].onclick = function (e) {
    e.preventDefault();
    alert('Link Nr.: ' + (i + 1));
  }
}
```

Das kann natürlich nicht funktionieren, da vor einem Klick die Schleife bereits durchlaufen wurde und man immer den letzten Wert bekommt. Ach ja, das `preventDefault` verhindert, dass man zu dem angeklickten Link kommt. Nur nicht im IE, da benötigt man wieder eine Extrawurst.

2. Ein Closure

```
var links = document.getElementsByTagName('a');
for (var i = 0; i < links.length; i++) {
  links[i].onclick =
  (
```

```
    function () {
      var link = i;
      return function (e) {
        e.preventDefault();
        alert('Link Nr.: ' + (link + 1));
      }
    }
  )();
}
```

Erläuterung

Ehrlich gesagt habe ich mit so einer Vorgehensweise noch nie gearbeitet. Darum hoffe ich mal, dass meine Erläuterungen hierzu korrekt sind. Zunächst mal bauen wir uns mit `links[i].onclick = (...)()` einen Selbstzünder, der sofort ausgeführt wird. Innerhalb dieses Selbstzünders erzeugen wir nun ein Closure, indem wir den Wert von `i` in der Variable `link` abspeichern (`var link = i;`). Somit steht also immer der richtige Wert zur Verfügung.

Den Rest benötigen wir nur, um zu verhindern, dass man dann zum eigentlichen Link kommt. Die anonyme Funktion `function (e)` verhindert, dass unser Selbstzünder direkt nach dem Laden der Seite ausgelöst wird, sondern nur bei einem Klick auf einen Link läuft. Das `return` ist dabei der Rückgabewert, der benötigt wird um `preventDefault` auszulösen. Das Prinzip ist dasselbe wie bei einem Formular abschicken. Also die Sache mit dem `<form ... onsubmit="return checkForm();">`.

3. Die Sache mit dem `this`

Das Problem habe ich euch ja schon an dieser Stelle gezeigt. Auch wenn ich mich hier nicht um die Details kümmere (interessiert eh keine Sau), so soll es sich hier ebenfalls um ein Closure handeln. Wenn es auch etwas Seltsames.

```
function Konstruktor() {
  var that = this;
  ...
}
```

OOP mit JavaScript - Überschreiben

1. And now for something completely different

Soll heißen, jetzt kommt etwas, das wirklich muy importante ist. Also sauwichtig! Drum spitzt die Ohren, vergesst die Closures und konzentriert euch. Also, in PHP gibt es die Möglichkeit, Eigenschaften und Methoden von Elternklassen zu überschreiben. Und das funktioniert auch so ähnlich in JavaScript. Nur mit einem Unterschied. Es ist viel viel leichter.

2. Der Sinn dahinter

Da ich schon einige Mails zu diesem Thema bekommen habe, wo man mich fragte, was das überhaupt soll, werde ich das jetzt versuchen zu erklären. Also, ein paar Beispiele für das Überschreiben habe ich euch ja schon gezeigt. Und das Konzept ist nun so schwer zu verstehen nicht. Mein Gott, ich rede wie Yoda. Egal, kommen wir nun zu den Details.

Wir haben in den letzten Abschnitten ein paar Panzer gebaut. Und über die Vererbung uns dann die einzelnen Arten zusammengebaut. Jetzt erinnern wir uns an unseren Viagrapanzer, also den mit der großen Wumme. Dort haben wir die Eigenschaft `kanone` überschrieben.

Das hätte man auch anders machen können

Jau, aber wie? Dazu müsste man jedem individuellen `Kampfpanzer`-Objekt das entsprechende Kaliber verpassen und das nur, weil es irgendeinen dummen `Spezialpanzer` gibt. In der Realität sähe das dann so aus, dass Y-Tours 500 `Kampfpanzer` und nur 10 `Spezialpanzer` bestellt. Da wäre es doch ziemlich sinnlos, für jeden einzelnen `Kampfpanzer` eine eigene(!) Kanone zu entwickeln. Denn programmiertechnisch wäre dem so.

3. Ausnahmen

Man überschreibt also Eigenschaften und Methoden in besonderen Fällen, um sich Arbeit zu sparen. Das ist so ähnlich wie bei CSS, wo man das auch ganz gerne macht. Da definiert man sehr häufig bei den Eigenschaften allgemeine Werte und passt sie dann Stück für Stück an.

4. Das Prinzip

Dazu greifen wir mal auf ein ganz einfaches Beispiel zu, wo wir in einem Objekt die Methode und die Eigenschaft ändern.

```
var obj          = new Object();
obj.eigenschaft = 'blubb';
obj.methode      = function ()
{
  alert (this.eigenschaft);
};
obj.methode();
// hier wird "eigenschaft" überschrieben
obj.eigenschaft = 'blubber';
obj.methode();
// hier wird "methode" überschrieben
obj.methode      = function ()
{
  alert ('Kein Bock mehr');
};
obj.methode();
```

Ein Hinweis

Bei diesem Beispiel wird mir persönlich angst und bange. Denn normalerweise arbeite ich damit nur, wenn ich Objekte einmalig(!) hart verdrahte. Und natürlich nehme ich dafür Objektliterale. Das hier ist zwar dasselbe in Grün, aber übersichtlicher.

Wichtig

An diesem Beispiel könnt ihr schon erahnen, dass man nur öffentliche Eigenschaften und Methoden überschreiben kann aber keine privaten. Das sieht zwar manchmal so aus, stimmt aber nicht. Außerdem müsst ihr dabei noch auf ein paar Feinheiten achten, aber das kommt jetzt.

OOP mit JavaScript - Überschreiben - Bei Referenzen

1. Die lieben Ballermänner

Da die vorherigen Beispiele mit den Panzern doch wunderbar waren, machen wir direkt mit ihnen weiter. Konzentrieren wir uns daher wieder mal auf die Kanone. Scheint bei Männern wohl eine Fixierung auszulösen. Darum mache ich die jetzt kleiner!

Der Code

```
function Kampfpanzer()
{
  this.kanone    = 120;
  this.schiessen = function()
  {
    alert ('Bumm bumm mit ' + this.kanone + 'mm-Kanone');
  }
}
var pz_1    = new Kampfpanzer;
pz_1.kanone = 105;
pz_1.schiessen();
```

Die Erläuterung

... könnte ich mir eigentlich sparen. Wenn es da nicht eine Besonderheit gäbe. Denn die Eigenschaft `kanone` wurde nur im Objekt `pz 1` überschrieben. Der Wert im Konstruktor selber bleibt erhalten. Das kann man sehr schön erkennen, wenn man ein neues Objekt erzeugt und nur(!) die Methode `schiessen` ausführt. Denn da hat man wieder ein ordentliches Kaliber.

```
var pz_2 = new Kampfpanzer;
pz_2.schiessen();
```

2. Überschreiben bei Aggregation

Da man hier in einer anderen Konstruktorfunktion ebenfalls nur mit einer Referenz/Instanz arbeitet, gilt hier dasselbe Prinzip. Allerdings müsst ihr euch vor Augen halten, dass Änderungen auch Auswirkung auf alle(!) anderen Objekte hat, die aus diesem Konstruktor erzeugt worden sind.

Der Ausgangscode

```
function Kampfpanzer()
{
  this.kanone    = 120;
  this.schiessen = function()
  {
    alert ('Bumm mit ' + this.kanone + 'mm-Kanone');
  }
}
function nochnPanzer()
{
  var nochEiner    = new Kampfpanzer;
  nochEiner.kanone = 155;
  nochEiner.schiessen();
}
```

```
var pz_1 = new nochnPanzer;
var pz_2 = new nochnPanzer;
```

So, jetzt hab ich die Kanone mal wieder größer gemacht, sonst reagiert noch einer von euch auf diese komischen Mails, wo es um blaue Pillen geht.

Erläuterung

Da sowohl das Objekt `pz 1` als auch `pz 2` aus derselben Konstruktorfunktion erzeugt wurden, haben beide Objekte jetzt deutlich mehr in der Hose als üblich. Objekte, die aus den Original entstanden sind, müssen sich dagegen mit deutlich weniger begnügen.

```
var pz_3 = new Kampfpanzer;
pz_3.schiessen();
```

OOP mit JavaScript - Überschreiben - Beim Prototyping

1. Altbekanntes

Beim Prototyping gelten dieselben Regeln wie bei der Aggregation. Alle Eigenschaften und Methoden die überschrieben werden, gelten nur für die entsprechenden Objekte, die daraus erzeugt wurden.

Der Code

```
function Kampfpanzer()
{
  this.kanone    = 120;
  this.schiessen = function()
  {
    alert ('Bumm mit ' + this.kanone + 'mm-Kanone');
  }
}
function nochnPanzer()
{
  this.kanone    = 155;
  this.schiessen = function()
  {
    alert ('Bumm mit superfetter endgeiler ' + this.kanone + 'mm-Kanone');
  }
}
nochnPanzer.prototype = new Kampfpanzer;
var pz_1              = new nochnPanzer;
pz_1.schiessen();
var pz_2              = new nochnPanzer;
pz_2.schiessen();
```

Erläuterung

Probiert das Beispiel einfach aus. Dann seht ihr was passiert. Und das Prinzip kennt ihr ja schon. Ich glaube auch nicht, dass ich dazu noch viel sagen muss.

2. Obacht!

Wenn ihr beim Prototyping bereits existierende Objekte um Methoden oder Eigenschaften ergänzen wollt, so passt um Gottes Willen höllisch auf, wie ihr die nennt. Denn wenn ihr Bezeichnungen dafür nehmt, die bereits existieren, so werden die knüppelhart überschrieben. Dann fallt ihr dabei so richtig derbe auf die Fresse. In dem folgenden Beispiel überschreibt ihr eine der wenigen Methoden, die das `Array`-Objekt euch bietet. Und das werdet ihr bitter bereuen, sobald ihr sie benötigt. Also die Augen auf!

```
Array.prototype.join = function ()
{
  alert ('Ätsch!');
};
blubb = new Array ('kräh','bla','schwall');
blubb.join (' ');
```

OOP mit JavaScript - Überschreiben - Privates

1. Missverständnisse

... gibt es beim Überschreiben immer wieder. Vor allem, wenn es um **PRIVATE** Eigenschaften und Methoden geht. Keine Angst, ist mir auch so gegangen, darum werde ich an dieser Stelle endlich mal das Licht der Erkenntnis über euch zum Leuchten bringen. Auf dass euer Geist erhellt werde.

Der Code

```
function Kampfpanzer()
{
  var kanone     = 120;
  this.schiessen = function()
  {
    alert ('Bumm mit ' + kanone + 'mm-Kanone');
  }
}
function nochnPanzer()
{
  var kanone = 155;
}
nochnPanzer.prototype = new Kampfpanzer;

var pz_1 = new nochnPanzer;
pz_1.schiessen();
```

Erläuterung

Da ich ja schon vorher sagte, dass man sowohl beim Prototyping als auch der Aggregation nur Zugriff auf die öffentlichen Eigenschaften und Methoden des "Mutter"-Konstruktors hat, kann das hier natürlich nicht funktionieren.

2. Aber

... was passiert, wenn man zum Beispiel eine Eigenschaft im "Kind"-Konstruktor als öffentlich festlegt? Funktioniert dann das Überschrieben? **NÖ!** Man fügt in dem Fall einfach eine weitere Eigenschaft hinzu.

```
function Kampfpanzer()
{
  var kanone     = 120;
  this.schiessen = function()
  {
    alert ('Bumm mit ' + kanone + 'mm-Kanone');
    alert ('Bumm mit ' + this.kanone + 'mm-Kanone');
  }
}
function nochnPanzer()
{
  this.kanone = 155;
}
nochnPanzer.prototype = new Kampfpanzer;
```

```
var pz_1 = new nochnPanzer;
pz_1.schiessen();
```

Denn in JavaScript, man glaubt es kaum, ist auch so was möglich, ohne dass sich beide Eigenschaften in die Quere kommen.

```
function Kampfpanzer()
{
  var kanone     = 120;
  this.kanone    = 155;
  this.schiessen = function()
  {
    alert ('Bumm mit ' + kanone + 'mm-Kanone');
    alert ('Bumm mit ' + this.kanone + 'mm-Kanone');
  }
}
```

Mein Tipp

Lasst um Gottes Willen die Finger davon. Da landet ihr ganz schnell in Teufels Küche.

OOP mit JavaScript - Fehlerbehandlung

1. Ein Hinweis vorab

Das was ich euch jetzt zeige, hat mit OOP eigentlich nichts zu tun. Aber das gilt ja für andere Themen aus diesem Bereich auch. Egal, es ist auf jeden Fall was für fortgeschrittene Anfänger, darum habe ich das hier hinein gepackt.

2. Das Prinzip

... ist ähnlich wie in PHP. Also versuchen, fangen, werfen. Allerdings gibt es bei JavaScript noch etwas, das in PHP immer noch fehlt, nämlich zu guter Letzt.

Ein Bespiel

```
try {
  if (!document.getElementById('bla')) {
    throw ('Is net da');
  }
  else { ... }
}
catch (e) {
  alert (e);
}
finally {
  alert ('hier');
}
```

Erläuterung

Im `try`-Block prüfen wir zunächst, ob ein HTML-Element mit der ID `bla` vorhanden ist. Wenn dem nicht so ist, so wird eine Exception geworfen. Die steht dann im `catch`-Block als Parameter zur Verfügung. In diesem Fall geben wir die im `throw` festgelegte Fehlermeldung aus.

Interessant ist hier das `finally`. Denn was in dem Block steht, wird auf jeden Fall ausgeführt. Noch ein Hinweis. Natürlich kann man im `catch`-Block auch noch mit weiteren Bedingungen arbeiten und die Fehlerbehandlung verfeinern.

3. Vorgegebene Fehlertypen

Neben eigenen definierten Fehlern bietet uns JavaScript auch noch interne an, die natürlich als Objekt angelegt sind. Als da wären.

- `EvalError` greift, wenn bei `eval` ein Fehler auftritt
- `SyntaxError` greift, wenn bei `eval` oder einem regulären Ausdrück ein Syntaxfehler vorhanden ist
- `RangeError` greift, wenn eine numerische Variable oder ein Parameter außerhalb des Gültigkeitsbereichs liegt
- `TypeError` greift, wenn eine Variable oder ein Parameter nicht valide ist
- `ReferenceError` greift, wenn beim Zugriff auf eine Variable oder Funktion selbige nicht vorhanden ist

Eigenschaften

Davon gibt es eigentlich eine ganze Menge, Allerdings ist die Implementierung bei den einzelnen Browsern so unterschiedlich, dass man eigentlich nur zwei davon benutzen kann.

- `name` Name des Fehlers
- `message` die Fehlermeldung

Und diese Fehlertypen kann man nun mit `instanceof` abfangen.

Ein Beispiel

```
try {
  bla.blubb();
}
catch (e) {
  if (e instanceof EvalError) {
    alert(e.name + ':' + e.message);
  }
  else if (e instanceof RangeError) {
    alert(e.name + ':' + e.message);
  }
  else if (e instanceof TypeError) {
    alert(e.name + ':' + e.message);
  }
  else if (e instanceof ReferenceError) {
    alert(e.name + ':' + e.message);
  }
}
```

Erläuterung

In diesem Beispiel wird im `catch`-Block nach Fehlertypen unterschieden und dann der Name und die Meldung ausgegeben. Aber vorsichtig! Normale Browser werfen hier einen `ReferenceError`. Nur der Internet Explorer tanzt mal wieder aus der Reihe und haut uns einen `TypeError` um die Ohren.

3. Eigene Fehler definieren

Auch das ist in JavaScript möglich und zwar über das `Error`-Objekt. Allerdings sind die Möglichkeiten arg beschränkt. Oder genauer gesagt, man kann nur Fehlermeldungen erzeugen.

```
try {
  if (document.getElementById('bla')) {

  }
  else {
    throw (new Error('Riesen Schei***e'));
  }
}
catch(e) {
  alert(e.message);
}
```

Wie ihr schon seht, gibt es zum ersten Beispiel praktisch gesehen keinen Unterschied. Der ist nur theoretischer Natur. Denn hier haben wir ein `Error`-Objekt erzeugt. Mehr nicht. `name` hat als Wert `Error` und `message` unseren String. Interessant wird es dagegen, wenn man selber interne Abstufungen der eigenen Fehler vornehmen möchte.

```
try {
  if (...) {
    ...
  }
  else {
    if (...) {
      e1 = new Error('Nicht so wild');
      throw (e1);
    }
    else {
      e2 = new Error('Riesen Schei***e');
      throw (e2);
    }
  }
}
catch(e1) {
  // Weitermachen
}
catch(e2) {
  // Alle Maschinen voller Stop
}
```

4. Ein kleines Fazit am Ende

Ich hoffe, ihr habt meinen Ausführungen folgen können. Denn OOP mit JavaScript ist ganz schön kompliziert. Vor allem, wenn man sonst mit anderen Sprachen zu tun hat. Und passt höllisch auf, was euch andere so erzählen. Oder was ihr so im Internet zu diesem Thema findet. Denn da geben viele einen ziemlichen Schrott von sich. Ich selber bin da auch schon zweimal darauf reingefallen.

Eine Ergänzung

Auf einige Möglichkeiten bin ich bisher nicht eingegangen, weil mir deren Konzepte nicht so wahnsinnig toll gefallen. Trotzdem werde ich demnächst ein Ergänzungstutorial dazu verfassen, wo ich mich besonders auf Literale und die ominösen anonymem Variablenfunktionen/benannte Funktionsausdrücke konzentrieren werde.