

1 Einleitung

In den nächsten Abschnitten finden Sie einige Hinweise zu den in Javascript verfügbaren Datentypen, insbesondere Funktionen und Objekte, und zur objektorientierten Programmierung mit den Sprachmitteln von Javascript.

Die Zusammenstellung basiert vor allem auf dem Buch von Nicholas Zakas, das in deutscher Übersetzung 2014 unter dem Titel "JavaScript objektorientiert" beim d.punkt-Verlag erschienen ist.

Ausführung der Beispiele

Die Beispiele können Sie mit dem Javascript-Interpreter **nodejs** ausführen. Erläuterungen zu nodejs und zur Ausführung der Beispiel in einem Webbrowser finden Sie im Abschnitt [Werkzeuge](#).

2 Datentypen

In Javascript stehen eine Reihe von Datentypen zur Verfügung. In den folgenden Abschnitten finden Sie einen kurzen Überblick. Die speziellen Eigenschaften von Funktionen und Objekten werden im Kapitel [Objektorientierte Programmierung](#) ausführlicher behandelt.

Wie bei anderen Script-Sprachen (z.B. Python) verwendet Javascript eine dynamische Typbindung. Bezeichner werden durch Zuweisung oder Verwendung in einem entsprechenden Kontext an einen Typ gebunden, die Bindung kann jederzeit geändert werden (Hinweis auf neuere Sprachmittel ...).

2.1 Primitive Datentypen

Javascript kennt folgende primitive Datentypen:

Datentyp	Wert	Beispiele
Zahlen	Ganzzahl oder Gleitkommazahl	123 oder 14.15
Zeichenketten	Zeichen oder Zeichenfolge	"a" oder 'xyz'
null	null	
nicht definiert	undefined	
boolscher Wert	true oder false	

Bei den primitiven Datentypen werden die Werte direkt zugewiesen, d.h. es gibt einen direkten Zusammenhang zwischen Bezeichner und Wert. Jeder Bezeichner mit einem primitiven Datentyp hat einen eigenen Wert.

2.2 Referenztypen

Referenztypen speichern Referenzwerte. Bei der Zuweisung von Referenzwerten werden nicht die referenzierten Daten, sondern nur die Referenzen verwendet bzw. kopiert.

Referenzwerte werden solange vorgehalten, solange es noch Referenzen auf sie gibt. Der automatisch ablaufende Garbage-Collector entfernt einen Referenzwert erst, wenn es keine Referenzen mehr darauf gibt.

Beispiel:

```
var referenzTyp1 = referenzwert;  
var referenzTyp2 = referenzTyp1;
```

Durch die zweite Zuweisung wird **keine** Kopie des Referenzwertes `referenzwert` erstellt, sondern nur die Referenz kopiert. Anschließend verweisen **beide** Variablen auf denselben Referenzwert. Änderungen des Referenzwertes sind bei beiden Variablen sichtbar.

Eine Referenz kann in Javascript durch Zuweisung von `null` aufgehoben werden:

```
referenzTyp1 = null;
```

Nach der Zuweisung enthält nur noch `referenzTyp2` die Referenz auf den Referenzwert.

In Javascript repräsentieren Referenztypen (ausschließlich) Objekte.

2.3 Objekte

Ein Objekt ist in Javascript eine nicht geordnete Liste von Eigenschaften, die aus einem Namen (Zeichenkette [*string*]) und einem Wert (primitiver Wert oder Referenzwert) bestehen.

Objekte werden entweder direkt mit der Literalform oder mit Hilfe des `new`-Operators erzeugt. Verwendet man den `new`-Operator, muss eine Funktion angegeben werden, die beim Konstruktionsvorgang ausgeführt wird:

```
var newPerson_o = new Person_c1();
```

`Person_c1` ist dabei eine Javascript-Funktion. Jede Funktion kann mit Hilfe des `new`-Operators zur Erzeugung von Objekten verwendet werden.

Die Variable `newPerson_o` repräsentiert einen Referenztyp, da Objekte nicht direkt bei der Variablen abgespeichert werden, sondern auf die Objekte nur verwiesen werden kann.

Objekte können jederzeit erweitert werden, d.h. man kann einem bereits erzeugten Objekt zur Laufzeit weitere Eigenschaften hinzufügen.

Dazu finden Sie weitere Erläuterungen im Kapitel [Objektorientierte Programmierung](#).

2.3.1 Vordefinierte Objekte

Einige Objekte sind in Javascript vordefiniert und können als in die Sprache eingebaute Referenztypen mit dem `new`-Operator zur Erzeugung entsprechender neuer Objekte verwendet werden:

Referenztyp	Bedeutung	Erläuterung
Object	generisches Objekt	leeres Objekt, i.d.R. zur Erweiterung vorgesehen

Referenztyp	Bedeutung	Erläuterung
Array	Liste mit numerischem Index (0..n)	nicht kompakte Speicherung, Werte jeder Art
Date	Datum und Uhrzeit	stellt Methoden zur Bearbeitung bereit
Error	Informationen zu Laufzeitfehler	wird bei der Ausnahmebehandlung verwendet
Function	Funktion	alle Javascript-Funktionen sind von diesem Typ
Regex	regulärer Ausdruck	stellt Methoden zur Anwendung bereit

2.3.2 Literalformen

Für primitive Werte und Referenztypen, also Objekte, gibt es literale Formen für die Werte. Die Schreibweisen für primitive Werte finden Sie im Abschnitt [Primitive Datentypen](#).

Objekte können mit der literalen Schreibweise einfach eingerichtet werden:

- die Liste der Eigenschaften (Key-Value-Paare) wird in geschweifte Klammern gesetzt
- es ist möglich, eine leere Liste anzugeben, d.h. das Objekt hat dann zunächst keine speziellen Eigenschaften.

Beispiel:

```
var secondPerson_o = {  
  name: "Meier",  
  vorname: "Egon"  
}
```

Die Eigenschaftsnamen können auch als Zeichenkettenkonstante notiert werden (mit begrenzenden ' oder "). Dann sind auch Namen mit Leerzeichen und anderen Sonderzeichen möglich.

Ein über die literale Schreibweise erzeugtes Objekt unterscheidet sich von einem mit `new` erzeugten Objekt dadurch, dass keine konstruierende Funktion aufgerufen wird. Die über die literale Schreibweise erzeugten Objekte basieren direkt auf dem allgemeinen Objekt `Object`.

Auch für Funktionen gibt es eine literale Schreibweise. Diese wird in den kommenden Abschnitten behandelt.

Reguläre Ausdrücke können mit einer speziellen literalen Schreibweise definiert werden. Darauf wird hier nicht weiter eingegangen.

2.3.3 Zugriff auf die Eigenschaften eines Objekts

Auf die Eigenschaften eines Objekts kann über die Punktschreibweise (wie in C/C++: Elementzugriffsoperator) oder den Index-Operator zugegriffen werden. Aus syntaktischen Gründen kann man die Punktschreibweise nur bei Elementnamen verwenden, die zu keinen syntaktischen Fehlinterpretationen führen können.

Die beiden folgenden Zugriffe sind gleichbedeutend:

```
console.log(secondPerson_o.name);  
console.log(secondPerson_o["name"]);
```

2.3.4 Wrapper für primitive Typen

Für die primitiven Typen Zahlen und Zeichenketten gibt *Wrapper*-Objekte, die temporär erzeugt werden, wenn man mit diesen Typen arbeitet. Dadurch können diese primitiven Typen wie Referenztypen verwendet werden.

2.4 Funktionen

Aus der weiter vorne präsentierten Aufstellung ist ersichtlich, dass auch Funktionen Objekte sind. Mit Hilfe des vordefinierten Objekts `Function` kann man Funktionen erzeugen: dazu gibt man die Parameterliste (ggf. keine) und den Funktionsrumpf (Quellcode als Zeichenkette) an; als Name der Funktion dient der Name der Variablen, die die Referenz auf die neu erzeugte Funktion erhält:

```
// Erzeugung  
var x = new Function ("v", "return v;");  
  
// Aufruf  
var y = x("abcde"); // welchen Wert erhält y?
```

2.4.1 Literalformen für Funktionen

Es gibt **zwei** Literalformen für Funktionen:

- **function declaration**
- **function expression.**

Bei einer `function declaration` wird das Schlüsselwort `function` zusammen mit einem Namen verwendet, um die Funktion zu deklarieren:

```
function x (v) {  
    return v;  
}
```

Diese Schreibweise entspricht etwa den Notationen in C oder C++.

Funktionen können aber auch als auswertbarer Ausdruck literal angegeben werden. Zur Laufzeit wird der Ausdruck ausgewertet, Ergebnis ist ein Funktionsobjekt, das entweder einer Variablen zugewiesen wird oder als Aktualparameter für eine Funktion verwendet wird:

```
var x = function (v) {  
    return v;  
}
```

Beim Funktionsausdruck wird der Name, da weiter nicht sinnvoll verwendbar, weggelassen.

In der Regel werden Funktionen mit Hilfe einer der beiden literalen Formen definiert. Die Erzeugung mit Hilfe des vordefinierten Objekts `Function` ist eher die Ausnahme.

Beachten Sie: die erzeugten Funktionen sind Objekte, die Namen (direkt bei der `function declaration` oder als Variablennamen) stehen für die Referenzen auf diese Funktionsobjekte. Funktionen können außerdem weitere Eigenschaften zugewiesen werden.

2.4.2 Methoden: Funktionen als Eigenschaften von Objekten

Wie oben ausgeführt, weisen Objekte Eigenschaften auf, denen ein Wert - primitiver Wert oder Referenzwert - zugewiesen wird. Damit sind also auch Referenzen auf Funktionen als Werte möglich, die ggf. sogar als `function expression` direkt notiert werden. Funktionen, die in Objekten als Werte von Eigenschaften genutzt werden, nennt man **Methoden**.

Beispiel:

```
var person_o = {  
  name: "Meier",  
  vorname: "Egon",  
  gibName: function () {  
    return this.name + ', ' + this.vorname;  
  }  
}  
  
console.log(person.gibName());
```

Im Beispiel werden die beiden Eigenschaften zu einer neuen Zeichenkette zusammengesetzt (konkateniert) und als Ergebnis der Methode zurückgegeben. Auf die Bedeutung von `this` wird in den nächsten Abschnitten genauer eingegangen.

3 Objektorientierte Programmierung

3.1 Überblick

Objektorientierte Programmierung wird in Javascript nicht mit Hilfe von Klassen und Instanzen vorgenommen, sondern anhand von Objekten und deren Prototypen (die, strenggenommen, wiederum Objekte sind).

In Javascript gibt es wie zuvor erläutert nur Objekte.

Objekte können über individuelle Eigenschaften verfügen, die nur sie besitzen, und allgemeine Eigenschaften, die allen Objekten eines Typs gemeinsam sind. Die allgemeinen Eigenschaften werden in den sog. Prototype-Objekten definiert; jedes Objekt erhält einen impliziten Verweis auf sein Prototyp-Objekt.

Hinweis:

Dieser implizite Verweis kann bei Firefox als Eigenschaft `__proto__` angesprochen werden. Die Mozilla-Dokumentation weist darauf hin, dass in den aktuellen Javascript-Versionen anstelle des direkten Zugriffs auf diese Eigenschaft die Methode `Object.getPrototypeOf` verwendet werden soll. `__proto__` ist ggf. in anderen Webbrowsern (z.B. IE) nicht verfügbar!

Prototyp-Objekte werden nur für die vordefinierten Objekte und die erzeugten Funktionen bereitgestellt. Mit `new` oder der Literalform für Objekte erzeugte Objekte haben kein eigenes Prototype-Objekt, sondern beziehen sich auf das Prototype-Objekt der erzeugenden Funktion (bei Verwendung von `new`) oder das Prototype-Objekt von `Object`.

Wird auf eine Eigenschaft eines Objekts zugegriffen, wird zunächst untersucht, ob es eine individuelle Eigenschaft gibt. Wird diese nicht gefunden, wird der implizite Prototyp-Verweis ausgewertet: es wird geprüft, ob im Prototyp-Objekt, das über den impliziten Verweis erreichbar ist, die gesuchte Eigenschaft vorhanden ist. Wenn das Prototyp-Objekt selber wieder einen von null verschiedenen Verweis auf ein weiteres Prototyp-Objekt enthält, wird ggf. dem Verweis gefolgt und dort nach der Eigenschaft gesucht.

Man nennt dies die "prototype-chain" (Prototyp-Kette). Der Suchvorgang wird solange durchgeführt, bis entweder die Eigenschaft gefunden wurde (dann wird sie verwendet, im Falle einer Funktion heißt das ggf. Aufruf der Methode) oder das Ende der Kette erreicht ist (was dann zu einer Fehlermeldung führt).

Wann und wie wird der implizite Verweis auf ein Prototyp-Objekt erzeugt?

Zur Erzeugung eines Objekts benötigt man eine "Constructor-Funktion" (Konstruktor-Funktion) (CF). CFs sind spezielle Objekte: sie werden erzeugt anhand des vordefinierten `Function`-Objekts, das wiederum eine Spezialisierung des allgemeineren `Object`-Objekts ist.

Die CF wird zusammen mit der Anweisung `new` verwendet:

```
neues-objekt = new CF;
```

(Man kann dabei auch eine Parameterliste verwenden.)

Was passiert dabei:

- es wird ein leeres Objekt erzeugt
- der `this`-Bezeichner verweist auf dieses leere Objekt
- der Funktionsrumpf der CF wird ausgeführt, z.B. werden dort über `this.eigenschaft` neue Eigenschaften im (zunächst leeren) Objekt eingetragen: das sind individuelle Eigenschaften(!)
- der implizite Prototyp-Verweis des neuen Objekts wird dem EXPLIZITEN Prototypen der CF zugeordnet.

Die CF verwendet also einen expliziten Prototyp-Verweis für den Prototypen, der für die mit Anwendung der CF zu erzeugenden Objekte verwendet wird. Dieser Verweis wird in der Eigenschaft `prototype` gespeichert und ist NICHT der implizite Verweis der CF.

Die CF hat einen impliziten Prototyp-Verweis auf den Prototyp des `Function`-Objekts, das `Function`-Object einen impliziten Verweis auf den Prototyp des `Object`-Objekts. D.h., sowohl bei `Function`-Objekten als auch bei `Object`-Objekten steht ein korrespondierendes Prototyp-Objekt zur Verfügung, das die allgemeinen Eigenschaften von Funktionen und Objekten aufnimmt. Ergänzt man die Eigenschaften dieser Objekte, werden allen Funktionen bzw. allen Objekten (und damit auch allen Funktionen) diese (neuen) Eigenschaften zur Verfügung gestellt.

Im expliziten Prototyp einer CF wird der Rückverweis auf die CF in der Eigenschaft `constructor` gespeichert und zur Verfügung gestellt.

Wenn Sie also bei allen erzeugten Objekten eines Typs (d.h. mit derselben CF erzeugt!) eine Eigenschaft ergänzen wollen, weisen Sie diese der `prototype`-Eigenschaft des CF zu (siehe z.B. Zeile 56).

Man kann Objekte auch mit Hilfe einer Objekt-Initialisierung in einer literalen Schreibweise erzeugen (siehe z.B. Zeilen 15 bis 18). Worin besteht der Unterschied zur Verwendung einer CF?

Bei der Initialisierung wird zunächst ein leeres Objekt erzeugt (d.h. intern wird `new Object()` aufgerufen), dann werden die in der Initialisierung angegebenen Eigenschaften als individuelle Eigenschaften in das neue Objekt

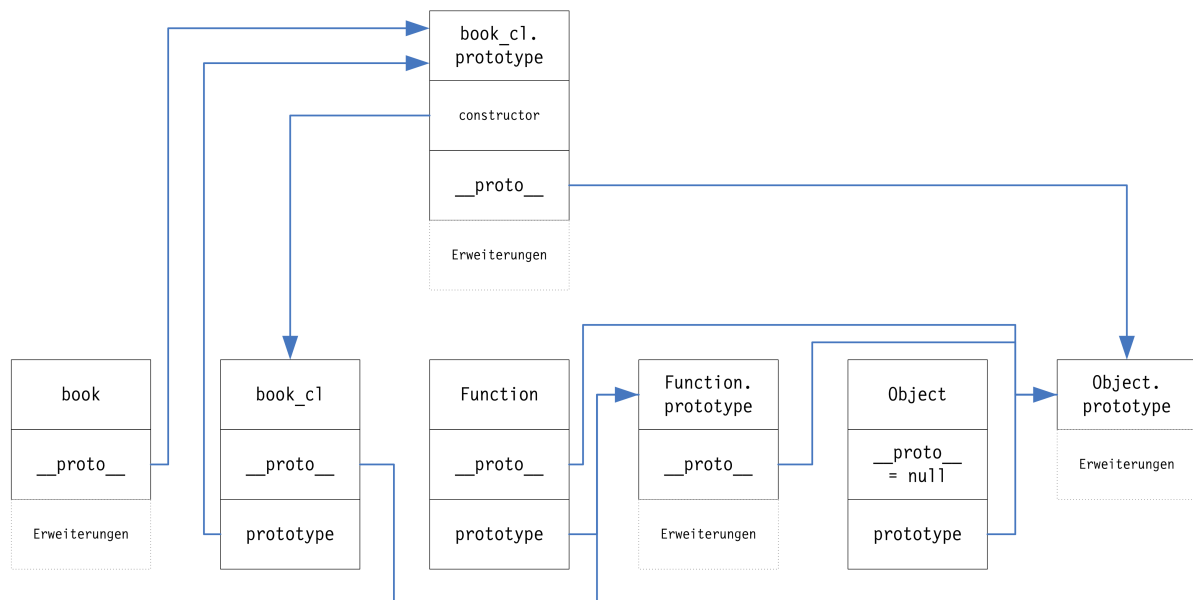
übernommen. Da keine explizite CF angegeben wird, sondern Object als CF benutzt wird, ist der Prototyp des neuen Objekts das durch `Object.prototype` angegebene Prototyp-Objekt.

Beispiel Anwendung des Prototype-Konzepts

```
1 function book_cl (title_spl, publisher_spl) { // constructor-function (CF)
2     this.title = title_spl;
3     this.publisher = publisher_spl;
4 }
5 console.log("- 01 - ", "prototype" in book_cl); // in: wertet prototype-chain aus
6 console.log("- 02 - ", "constructor" in book_cl);
7 console.log("- 03 - ", book_cl.hasOwnProperty("prototype")); // hasOwnProperty: untersucht nur das
8 console.log("- 04 - ", book_cl.hasOwnProperty("constructor")); // einzelne Objekt
9
10 book_cl.prototype.toString = function() {
11     return this.title + " / " + this.publisher;
12 }
13
14 function DoIt() {
15     var book = { // Objekt-Initialisierer, d.h.
16         title: "Titel", // es wird intern nur new Object ausgeführt
17         publisher: "MySelf" // und nicht die CF "book_cl" !
18     };
19     console.log("- 05 - ", book.toString());
20     console.log("- 06 - ", book.hasOwnProperty("publisher"));
21     console.log("- 07 - ", book.hasOwnProperty("toString"));
22     console.log("- 08 - ", book.hasOwnProperty("__proto__"));
23     console.log("- 09 - ", book.hasOwnProperty("prototype"));
24     console.log("- 09 - ", "publisher" in book);
25     console.log("- 10 - ", "toString" in book);
26     console.log("- 11 - ", "__proto__" in book);
27
28     console.log("- 12 - ", "mit Konstruktor-Funktion");
29
30     book = new book_cl("t", "p");
31     console.log("- 13 - ", book.toString());
32     console.log("- 14 - ", book.hasOwnProperty("prototype"));
33     console.log("- 15 - ", "prototype" in book);
34     console.log("- 16 - ", "prototype" in book_cl);
35
36     // Erweiterungen der vordefinierten (built-in) Javascript-Objekte
37
38     Function.prototype.test = function () { console.log("- 17 - ", "Function-test"); }
39     Object.prototype.otest = function () { console.log("- 18 - ", "Object-test"); }
40
41     book_cl.test();
42
43     book_cl.prototype.otest();
44
45     book_cl.prototype.constructor.test();
46
47     book.__proto__.constructor.test();
48     try {
49         book.test(); // klappt nicht! Warum? Machen Sie sich den Unterschied
50                     // zwischen book_cl und book klar!
51     }
52     catch(e) {
```

```
53     console.log("- 19 - ", "klappt nicht!");  
54 }  
55  
56 book.otest();  
57  
58 book_cl.prototype.test = function () { console.log("- 20 - ", "book_cl.prototype.test!"); }  
59  
60 book_cl.test();  
61  
62 book.test(); // Jetzt geht's! Warum? Was wird angezeigt?  
63 }  
64  
65 DoIt();
```

In der nachfolgenden Abbildung finden eine Darstellung der Zusammenhänge.



```
book = new book_cl();
```

Erweiterungen:
- bei „book“ : individuell
- bei „book_cl.prototype“: für alle aus book_cl erzeugten Objekte
- bei „Function.prototype“: für alle Funktionen
- bei „Object.prototype“: für alle Objekte

(Es sind bei Function und Object nicht alle Verweise dargestellt)

Abbildung 1: Prototyp-Konzept: Vererbung in Javascript

3.2 Javascript-Functions und der verflixte this-Verweis

In Javascript gibt es den vordefinierten `this`-Verweis, der im Zusammenhang mit Funktionen und deren direkter Verwendung oder Verwendung als Constructor-Function von unterschiedlicher Bedeutung ist.

Bei direkter Verwendung muss zudem unterschieden werden zwischen

- function expression und
- function declaration.

Bei einer `function expression` wird besonders deutlich, dass Funktionen in Javascript *Werte (values)* sind, d.h. sie können wie (andere) Werte verwendet und ausgewertet werden. Das Ergebnis der Auswertung einer `function expression` ist das Funktionsobjekt:

```
1 var f = function () {};
```

Die Variable `f` hat als Wert ein Funktionsobjekt, das sich durch die Auswertung des Ausdrucks auf der rechten Seite der Zuweisung ergibt - in diesem Fall mit einem leeren Funktionsrumpf.

Die Auswertung der `function expression` darf nicht mit dem **Aufruf** der Funktion verwechselt werden! Nach der Zuweisung an die Variable `f` kann der Funktionsaufrufoperator `()` verwendet werden (wie z.B. auch in C oder Python), um durch die Auswertung des **Funktionsrumpfs** ein Ergebnis zu berechnen:

```
1 var fwert = f();
```

Man kann eine `function expression` auch direkt auswerten und aufrufen, wenn man Klammerung verwendet: das Funktionsobjekt wird geklammert und damit direkt ausgewertet, auf das Ergebnis - eben die Funktion - wird der Funktionsaufrufoperator angewendet:

```
1 var fwert = (function () {} )();
```

Als Ergebnis der Ausführung einer Funktion sind zwei verschiedene Arten von Ergebnissen möglich:

- `undefined`, z.B. wenn keine `return`-Anweisung vorliegt oder die `return`-Anweisung ohne Ausdruck verwendet wird
- oder
- die Auswertung des Ausdrucks bei der `return`-Anweisung.

Beispiele

```
1 var fwert1 = (function () { alert('Meldung'); return; })();  
2 // fwert1 hat den Wert undefined  
3  
4 var fwert2 = (function (v1, v2) { return v1+v2; })(1,2);  
5 // fwert2 hat den Wert 3  
6  
7 var fwert3 = (function (v1, v2) { return v1+v2; })( (function (p1, p2) { return p1*p2; })(2,3), 4);  
8 // fwert3 hat den Wert 10!
```

Wie in C oder Python werden beim Funktionsaufruf zunächst die Aktualparameter ausgewertet. Skalare Werte als Kopie weitergegeben (call by value), ansonsten werden Referenzen übergeben. Als Aktualparameter kann auch eine `function expression` verwendet werden (siehe `fwert3`):

- zunächst wird die `function expression` ausgewertet und ergibt die Funktion (hier im Beispiel: zur Multiplikation der beiden Parameter)

- dann wird die erzeugte (anonyme) Funktion aufgerufen durch Anwendung des Funktionsaufrufoperators mit den Aktualparametern 2 und 3
- das Ergebnis des Funktionsaufrufs ($2*3 \Rightarrow 6$) wird als erster Aktualparameter verwendet, d.h. v1 erhält dann den Wert 6, v2 den Wert 4, woraus sich dann das Ergebnis 10 durch den zweiten Funktionsaufruf ergibt.

Man beachte, dass die mit `function expression` erzeugten Funktionen in der Regel keinen eigenen Namen aufweisen. Die Bindung an einen Namen ergibt sich durch die Zuweisung an eine Variable oder eine Eigenschaft eines Objekts, womit die Funktion dann zur Methode des Objekts wird (siehe weiter oben). Es ist aber auch möglich, einen Namen anzugeben, der dann eine Eigenschaft des Funktionsobjekts ist!

Funktionen können auch in einer `function declaration` auftreten. Dieser Fall liegt vor, wenn eine Funktion außerhalb eines Ausdrucks notiert wird:

```
1 function (a,b,c) {  
2   return a*b*c;  
3 }  
4  
5 function Summe (x,y) {  
6   return x+y;  
7 }
```

Die in den Zeilen 1 bis 3 notierte Funktionsdeklaration weist keinen Namen auf, die damit erzeugte Funktion kann daher nicht angesprochen werden!

Die in den Zeilen 5-7 notierte Funktionsdeklaration stellt den normalen Fall dar: die benannte Funktion `Summe` wird erzeugt. Dabei treten 2 Fragen auf:

1. wann wird die Funktion erzeugt, d.h. wann wird das Funktionsobjekt ausgewertet?
2. wo wird der Name der Funktion gespeichert?

Die Antwort zur Frage 1 lautet: sobald der Javascript-Interpreter bei der Verarbeitung des Quellcodes auf die Funktionsdeklaration trifft.

Zur Beantwortung der Frage 2 muss man die *Umgebung* betrachten, die zum Zeitpunkt der Auswertung der `function declaration` vorliegt:

- der Funktionsname, der bei der `function declaration` angegeben wird, wird als Eigenschaft in die Umgebung übernommen
- wenn die `function declaration` in keine andere Umgebung eingebettet ist, ist das `global object` die Umgebung, d.h. der Funktionsname wird als Eigenschaftswert in das `global object` übernommen (wie immer bei Objekten, siehe oben), die Funktion kann dann über diesen Namen angesprochen werden.

Die gleiche Wirkung wird erreicht, wenn man einer globalen Variablen eine `function expression` zuweist. Eine globale Variable ist eine Eigenschaft des `global object` und wird erzeugt, wenn vor dem Variablennamen **kein** `var`-Hinweis angegeben ist.

```
1 // als function declaration  
2 function mult (a,b,c) {  
3   return a*b*c;
```

```
4 }  
5  
6 // als Zuweisung einer function expression an eine globale Variable  
7 mult2 = function (a,b,c) {  
8     return a*b*c;  
9 }
```

Und was ist nun mit dem verflixten this-Verweis?

Sie kennen den `this`-Verweis aus der Programmierung mit C++. Dort wird mit dem `this`-Verweis in Methoden das aktuelle Objekt als Arbeitsumgebung angesprochen.

In Javascript dient der `this`-Verweis im Prinzip demselben Zweck. Allerdings gibt es zwei wesentliche Unterschiede zu C++:

1. der Verweis kann jederzeit verwendet werden
2. zur Laufzeit kann der Bezug zur Ausführung einer Funktion (Methode) verändert werden.

In Javascript verweist `this` immer auf die *Ausführungsumgebung* einer Funktion, die im Prinzip immer durch ein Objekt - vordefiniert oder speziell erzeugt - dargestellt wird. Damit werden auch die beiden genannten Unterschiede verständlich:

- wenn das `global object` wie im vorhergehenden Beispiel die Umgebung einer Funktion ist, verweist `this` auf `global`
- wenn eine Funktion als Methode eines Objekts auftritt, verweist `this` auf das Objekt.

Im folgenden Beispiel sehen Sie zunächst den unterschiedlichen Bezug (Zeilen 3 bis 18):

```
1 // globale Variable  
2  
3 g = 'globalix';  
4  
5 function f() {  
6     console.log(g);  
7     console.log(this.g);  
8 }  
9  
10 var o = {  
11     'wert': 'EinWert',  
12     'methode': function (par) {  
13         console.log('Methode / ' + par + ' / ' + this.wert);  
14     }  
15 }  
16  
17 f();  
18 o.methode('myPar');  
19  
20 o.methode.call(o, 'myPar');  
21  
22 var o2 = {  
23     'wert': 'EinWert2'  
24 }
```

25	
26	<code>o.methode.call(o2, 'myPar2');</code>

Es ergibt sich bei Anwendung von `node.js` folgende Ausgabe

```
c:>node test2.js
globalix
globalix
Methode / myPar / EinWert
Methode / myPar / EinWert
Methode / myPar2 / EinWert2
```

In der Zeile 20 sehen Sie, wie der Aufruf einer Methode mit Hilfe der im `Function`-Object vordefinierten Methode `call` ausgeführt wird:

- der erste Parameter von `call` ist das Objekt, das die Ausführungsumgebung und damit die Bindung des `this`-Verweises darstellt: in Zeile 20 wird als `this` an das Objekt `o` gebunden
- der zweite (und ggf. weitere) Parameter wird als Aktualparameter an die Methode, die aufgerufen wird, weitergegeben
- die Methode, die aufgerufen wird, ergibt sich aus den Angaben *vor* `call`, in Zeile 20 wird mit `o.methode` die Methode `methode` des Objekts `o` referenziert, das in den Zeilen 10 - 15 angelegt wurde.

Zeile 20 und Zeile 18 sind gleichbedeutend.

In Zeile 26 sehen Sie dann, dass mit `call` auch ein anderer Objektbezug hergestellt werden kann! In Zeile 26 wird die Methode `methode` des Objekts `o` mit der Ausführungsumgebung `o2` aufgerufen, weshalb sich eine andere Ausgabe ergibt. Selbstverständlich ist es erforderlich, dass in der anderen Ausführungsumgebung gleich benannte Bezeichner verwendet werden, ansonsten ergeben sich Laufzeitfehler.

Was muss man also **beachten**? Eine Methode eines Objekts kann in Javascript auch mit Bezug zu einem ganz anderen Objekt aufgerufen werden!

Kommt das vor? Ja! Und zwar immer dann, wenn Sie Methoden als Callback-Funktionen z.B. bei der Ereignisbehandlung angeben. Wenn Sie ein Objekt mit einer Methode zur Ereignisbehandlung erzeugen und diese Methode z.B. bei einem Schalter in der `jquery`-Methode `...on('click', methode)` verwenden, wird diese Methode bei der Betätigung des Schalters mit Bezug zum *Schalterobjekt* aufgerufen und nicht mit dem Bezug zum Objekt, in dem die Methode definiert wurde! In `jquery` können Sie in der Regel dieses Problem durch die vorherige Vereinbarung einer Ausführungsumgebung umgehen.

Weitere Beispiele finden Sie in den Varianten des Anwendungsbeispiels "LIT".

3.3 Werkzeuge

3.3.1 Webbrowser

Mozilla Firefox: Addon *Firebug*

Google Chrome: Entwicklerwerkzeuge

In beiden Fällen u.a.:

- Script-Debugger
- Inspektor für die Dokumentenstruktur
- Inspektor für CSS / Layout
- Monitor Netzwerk

3.3.2 Javascript-Interpreter nodejs

Der Javascript-Interpreter **V8** von Google ist Bestandteil des Webbrowsers Google Chrome. Da dieser Interpreter quelloffen ist und wie alle Javascript-Interpreter in eine Ausführungsumgebung eingebettet werden muss, gibt es auch Einbettungen ohne Webbrowser. Ein populäres Beispiel ist `nodejs`, womit V8 als Konsolanwendung verfügbar wird. Informationen sowie Versionen für die verschiedenen Betriebssysteme finden Sie unter <http://nodejs.org>.

Wenn Sie `nodejs` zur Ausführung der Beispiele verwenden wollen, müssen Sie `alert`-Anweisungen durch die Ausgabe auf der Konsole ersetzen. Informationen zur Ausgabe auf der Konsole (über die Kanäle `stdout` und `stderr`) finden Sie in hier: <http://nodejs.org/api/console.html>.