Mit DVD

iX DEVELOPER

1/2014

Auf der Heft-DVD:

Über 5 GByte für Entwickler

Visual Studio Express 2013 für Webentwickler, Wakanda, Frameworks wie Ember.js und AngularJS, Bibliotheken und andere nützliche Tools, Codebeispiele, Leseproben aktueller JavaScript-Publikationen, 40 Episoden des SoftwareArchitekTOUR-Podcasts

JavaScript heute

Grundlagen:

Sprachparadigmen, Laufzeitumgebungen und Entwurfsmuster

Frameworks:

Einsatz in professionellen Projekten

Ember.js 1.1, AngularJS und Co.

Sprachen und Standards:

Was ECMAScript 6 bringt

Schneller mit asm.js

CoffeeScript und andere Alternativen

Server-side JavaScript:

Node.js und sein Ökosystem

Qualitätssicherung:

Mocha, Grunt und Headless Browser Testing

Tipps für sichere Anwendungen

präsentiert von: heise Developer www.heise-developer.de

Managed Hosting

IT-Sicherheit zertifiziert nach ISO 27001

Hosting made in Germany

Individuelles Hosting am Standort Frankfurt/Main mit persönlichem und kompetentem Support.

Professionelle Hostinglösungen vom Server bis zum Cluster-Cloudsystem, mit Beratung, Planung und Service 24/7.

Wir bieten über 10 Jahre Erfahrung in Hosting und Systemadministration.

Für mehr Performance, Sicherheit und Verfügbarkeit.

hostserver.de/hosting

HOSTSERVER
Berlin ▪ Marburg ▪ Frankfurt am Main

Beratung unter:
0 30 / 47 37 55 50

Hosting made in Germany

Totgesagte leben länger

Douglas Crockford, Autor von „JavaScript: The Good Parts“ und Schöpfer des Codeanalyse-Werkzeugs JSLint, hat JavaScript einmal als die wohl am meisten missverstandene Programmiersprache der Welt bezeichnet. Nun denkt man beim Wort „missverstanden“ nicht unbedingt an so etwas Technisches wie eine Skriptsprache. Genies – ja, Künstler – okay, aber JavaScript?

Den niederländischen Künstler Vincent van Gogh kann man ohne Zweifel als missverstanden bezeichnen: Nach einigen Jahren im Kunsthandel steckte er zehn Jahre lang alle Energie in seine Werke. Das schien die restliche Welt allerdings nur marginal zu interessieren, verkaufte er doch Zeit seines Lebens nur wenige Bilder und war finanziell bis zuletzt von seinem jüngeren Bruder Theo abhängig. Erst einige Zeit nach seinem Tod nahm das Geschäft mit seiner Kunst richtig Fahrt auf. Heute kennt hingegen fast jeder zumindest seine Sonnenblumen, die einem von Einkaufsbeuteln und Regenschirmen entgegenlachen.

JavaScript war im Gegensatz zu van Gogh nie wirklich tot und konnte sich immer einer nicht zu vernachlässigenden Anhängerschaft erfreuen. Wie jede Sprache hat auch diese Elemente, mit denen man als Entwickler mehr oder weniger gut zurechtkommt – und wem die Namensähnlichkeit zu Java nach all den Jahren tatsächlich noch ein Dorn im Auge ist, sollte sich einmal eine halbe Stunde mit einem C-Programmierer über C++ unterhalten.

Der neuerliche Aufschwung, den JavaScript erlebt, ist also weder die Stimme aus dem Grab noch eine späte Würdigung, wie es bei anderen Missverstandenen der Fall ist. Vielmehr ist er dem technischen Fortschritt und den vielen Entwicklern zu verdanken, die sich von den allzu häufig in den Vordergrund gestellten Makeln nicht abschrecken ließen und sich stattdessen an die Verbesserung des Vorhandenen gewagt haben. Durch sie ist das Ökosystem der Sprache heute bunter und vielfältiger denn je, und langsam, aber sicher ist JavaScript auch in großen Unternehmen angekommen.

Gegen die letzten Reste des Unverständnisses kann, wie so häufig, nur eins helfen: Information. Ähnlich wie es Theo van Goghs Frau übernahm, die Öffentlichkeitsarbeit für den toten Maler zu leisten, soll dieses Heft ein wenig Licht ins Dunkle bringen, wenn es um den heutigen Stand von JavaScript geht. Quereinsteiger können durch die Lektüre einiger Artikel zu Grundlagen der Sprache eine gute Basis für erste Projekte schaffen, während sich fortgeschrittene Entwickler durch Hinweise zu qualitativ hochwertigen und sicheren Webanwendungen eventuell in die Lage versetzt sehen, das eine oder andere in ihrem Workflow zu verbessern.

In separaten Artikeln beleuchten einige Autoren momentan interessante Frameworks und Bibliotheken, und auch serverseitiges JavaScript kommt mit Ausführungen zum Thema Node.js nicht zu kurz. Erscheint dann immer noch vieles wirr und umständlich, können „Sprachaufsätze“ wie CoffeeScript oder TypeScript Alternativen bieten oder Module zur Integration von Code aus anderen Sprachen helfen.

Um nicht zu sehr in der Theorie zu verharren, runden Projektberichte aus der Praxis und einige Programmierbeispiele das Heft ab. Letztere liegen in großen Teilen gemeinsam mit nützlichen Werkzeugen auf der Heft-DVD bei, womit es mehr als genug Möglichkeiten gibt, sich sofort in die Entwicklung zu stürzen oder einfach mal wieder etwas auszuprobieren. Und sollte es dann nicht auf Anhieb funktionieren, kann man sich wieder des missverstandenen Niederländers erinnern, der einmal sagte: „Und gerade dadurch, dass man sich irrt, findet man manchmal den Weg.“

JULIA SCHMIDT

Grundlagen

Bis vor einigen Jahren standen die Chancen ziemlich gut, in Entwicklerkreisen mit einem Lächeln bedacht zu werden, wenn man von seinen JavaScript-Projekten erzählte. Die Konzepte sind die alten, allerdings haben sich die Zeiten geändert und man kann selbst im Unternehmensalltag auf die Sprache treffen.

ab Seite 7

Sprachen und Standards

Um zu gewährleisten, dass alle Browser trotz unterschiedlicher Laufzeitumgebungen JavaScript verstehen, bedarf es einiger Grundregeln. Der Kern der Sprache ist deshalb im ECMAScript-Standard festgeschrieben und Neuerungen sind immer häufiger auch von „JavaScript-Alternativen“ inspiriert.

ab Seite 33

Grundlagen

Einführung
JavaScript verstehen und es hassen und lieben lernen 8

Paradigmen
JavaScript als Multiparadigmen-Sprache 14

Code-Struktur
Übersicht: Entwurfsmuster in JavaScript 19

Brendan Eich
Im Interview mit dem JavaScript-Erfinder 24

Literatur
JavaScript – eine Buchauswahl 28

Einsatz
Laufzeitumgebungen in JavaScript 30

Sprachen und Standards

Der Standard
Die Zukunft von ECMAScript 34

asm.js
JavaScript nähert sich nativer Geschwindigkeit 40

Alternativen zu JavaScript
TypeScript – Hoffnung für JavaScript-Hasser 43
Hello CoffeeScript – eine Einführung 50
Dart – es geht auch anders 55

Frameworks

MVC und MVVM
Model View ViewModel mit Knockout.js 60
Model View Controller mit Backbone.js 64
Webanwendungen mit AngularJS 70
Ember.js 1.1 im Einsatz 76

Web-Apps
Einstieg in die Entwicklung mit Meteor 80
Anwendungen mit jQuery Mobile entwickeln 85

PhoneGap
Cross-Plattform-Apps erstellen 90

Server-side JavaScript

Node.js
Einsatz im Enterprise-Umfeld 98
Das Node.js-Ökosystem im Überblick 102
Projektbericht: Adaptation von Webinhalten mit Node.js 107
Methodik: Webanwendungen mit Node.js 111

Qualitätssicherung

Sicherheit
Worauf bei JavaScript-Anwendungen zu achten ist 116

Headless Browser Testing
Webseiten rendern und testen ohne Browser 120

Server-side JavaScript

Die weite Verbreitung von Node.js hat einen nicht unerheblichen Einfluss auf JavaScripts Beliebtheit. Die serverseitige Plattform hat dazu beigetragen, dass man Anwendungen nun durchgängig von der Nutzeroberfläche über den Server bis zur Datenbank in nur einer Sprache entwickeln kann.

ab Seite 97

Qualitätssicherung

Trotz IDE-Unterstützung unterlaufen selbst erfahrenen Entwicklern hin und wieder Fehler, die sich manchmal schon mit Tests oder durch Automatisierung vermeiden lassen. Um Sicherheitslücken nicht zuzulassen, ist zudem fundierte Kenntnis etwaiger Schwachstellen vonnöten.

ab Seite 115

Automatisierung
Mit Taskrunner Grunt arbeiten lassen **125**

Code überprüfen
Behavior-Driven Development mit Jasmine **128**
Unit-Tests mit Node.js **130**

Praxis

App-Entwicklung
Cross-Plattform-Entwicklung hybrider Web-Apps mit HTML5 **136**
Offline-fähige Browser-Anwendungen mit JavaScript **144**

Integration
Edge.js verbindet Node.js und .NET Framework in einem Prozess **149**

Anwendungsbeispiel
Aktien im Blick mit Yeoman, AngularJS und YQL **154**

JavaScript und Big Data
Visualisierung mit D3.js **162**

Projektbericht
Echtzeitergebnisanzeige für Motorsportrennen mit JavaScript **166**

Sonstiges

Editorial **3**
Inserentenverzeichnis **161**
Impressum **161**

Auf der Heft-DVD

Visual Studio Express 2013 fürs Web: Die kostenlose Express-Version von Microsofts Entwicklungsumgebung zum Erstellen von Webanwendungen und -diensten.

Wakanda: Das Open-Source-JavaScript-Framework soll dem gezielten Erstellen von Business-Webanwendungen dienen.

Tools: Aktuelle Versionen der im Heft vorgestellten Frameworks und Bibliotheken AngularJS, Backbone.js, Ember.js, jQuery, jQuery Mobile, Knockout, Meteor sowie der Node.js-Plattform.

Außerdem: Sprachalternativen zu JavaScript, Codebeispiele aus dem Heft, kleine Helfer, Leseproben aus aktuellen JavaScript-Publikationen und 40 Episoden des SoftwareArchitekTOUR-Podcasts.

Hinweis für Käufer der digitalen Ausgaben

- PDF- und iPad-Version: In der *iX*-App finden Sie einen Button zum Download des DVD-Images.
- PDF-E-Book: Folgen Sie im Browser der unter „Alle Links“ angegebenen URL zum DVD-Image.

Alle Links: www.ix.de/ix1317004

Artikel mit Verweisen ins Web enthalten am Ende einen Hinweis darauf, dass diese Webadressen auf dem Server der *iX* abrufbar sind. Dazu gibt man den *iX*-Link in der URL-Zeile des Browsers ein. Dann kann man auch die längsten Links bequem mit einem Klick ansteuern. Alternativ steht oben rechts auf der *iX*-Homepage ein Eingabefeld zur Verfügung.

Alle Links: www.ix.de/ix1317SSS

Grundlagen

„Lern' doch lieber richtig programmieren." Solche und andere Nettigkeiten bekam man noch bis vor einigen Jahren zu hören, wenn man in Entwicklerkreisen von seinen neuesten JavaScript-Projekten erzählte. Die Zeiten haben sich geändert und die Sprache hat ihren Weg bis in den Unternehmensalltag gefunden. Die Konzepte, die hinter ihr stecken, sind allerdings die alten.

JavaScript verstehen und es hassen und lieben lernen	**8**
JavaScript als Multiparadigmen-Sprache	**14**
Übersicht: Entwurfsmuster in JavaScript	**19**
JavaScript-Erfinder Brendan Eich im Interview	**24**
Lesetipps: Aktuelle Bücher	**28**
Laufzeitumgebungen in JavaScript	**30**

JavaScript verstehen und es hassen und lieben lernen

Grundstoff

Marcus Ross

Jeder kann doch JavaScript! Sicherlich ist das Nutzen der Sprache in Browsern für viele kein Problem. Aber JavaScript ist mehr als nur der Dreizeiler zum Bildertausch beim Mouseover. Die Sprache auf einen Blick – mit ihren Vor- und Nachteilen.

JavaScript ist nicht neu: 1995 benötigte Netscape eine Skriptsprache für den hauseigenen Browser beziehungsweise Server. Brendan Eich lieferte nach nur circa 12 Tagen Entwicklungszeit JavaScript ab. Der Name wechselte von Mocha zu LiveScript bis man schließlich bei JavaScript angelangte. Als Vorbilder im Sprach-Design dienten Self (Smalltalk) und Scheme (Lisp). Der „Java"-Anteil im finalen Namen wurde gewählt, um die Syntax der Sprache zu erklären, die an Java angelehnt war. Auch das geschah damals nicht unbewusst: Es sollte als Marketingmaßnahme an die bekannte Sprache erinnern. 1996 wollte dann auch Microsoft auf den Zug aufspringen und lieferte ein kompatibles JavaScript mit dem Namen JScript aus. Kurz darauf kam es zur Standardisierung durch die ECMA (European Computer Manufacturers Association). Seit dem heißt JavaScript eigentlich ECMAScript, was sich im normalen Sprachgebrauch nicht durchgesetzt hat.

Auf dem Weg nach oben

Außerhalb des Browsers war JavaScript nicht häufig zu finden. Der zweite Frühling der Sprache kam durch den massiven Einsatz in mobilen Endgeräten. Auch Erweiterungen im HTML-Standard, die es erlauben mit JavaScript auf GPS oder Touchscreens zuzugreifen, trugen dazu bei. Mittlerweile hat JavaScript den Browser verlassen und ist auch in Webservern wie Node.js oder NoSQL-Datenbanken wie MongoDB zu Hause. Hardware wie das Tessel-Board nutzen die Sprache ebenfalls zur Programmierung. Es ist also eine gute Idee, sich mit dem Thema zu beschäftigen. Aber genug zur Geschichte. Wie funktioniert JavaScript? (Natürlich bietet dieser Artikel nur einen Einblick in die Thematik. Für eine ausführliche Einführung lassen sich das „JavaScript"-Buch [1] von Stefan Koch und „JavaScript effektiv" [2] von David Herman heranziehen.)

Die üblichen grundlegenden Datentypen in JavaScript sind String, Integer/Double und Boolean. Die Deklaration von Variablen erfolgt durch das Schlüsselwort *var* und die Wertezuweisung geht von rechts nach links. Tabelle 1 enthält eine Übersicht der wichtigsten Befehle.

Hochkommas um den Wert signalisieren, dass es sich um einen String handelt, ohne entsteht ein einfacher Integer, der sich durch eine neue Zuweisung einfach überschreiben lässt.

Tabelle 1: Deklarationen und erste Befehle

Bedeutung	Syntax
Kommentar	*//Einzeiler* */*Kommentar über mehrere Zeilen*/*
Variablendeklaration	*var vorname;*
Variablendeklaration mit Wertzuweisung	*var vorname = 'Marcus';* *//mit Hochkomma für String* *var sinnDesLebens = 42;* *//ohne Hochkomma für Integer* *var binIchRichtig = true; //für Boolsche Werte*
Wertezuweisung für bestehende Variable	*vorname = 'Marcus';*
Konstantendeklaration	*const sinnDesLebens = 42;*
Arraydeklaration	*var meineFarben = ['rot','grün','blau'];*
Abfrage der Arraylänge	*meineFarben.length();*
Zugriff auf Arrayelemente via Index	*meineFarben[1];*
Array nachträglich erweitern	*meineFarben.push('gelb');*
Element aus Array löschen	*meineFarben.pop();*

Wenn dieses Verhalten nicht gewünscht ist, sind Konstanten zu nutzen. Versucht man sie zu verändern, wird keine Fehlermeldung erstellt, aber der Wert bleibt weiter bestehen.

Außer Ganzzahlen sind Fließkommazahlen möglich. Mit *var pi = 3.141592653589793;* erstellt man etwa *pi* als solche. Die Nachkommastellen werden per Punkt festgelegt. Wenn der Typ der Variablen zur Laufzeit zu bestimmen ist, lässt sich das Kommando *typeof* verwenden. Es liefert den Typ einer Variable zurück. Möglich sind dabei die Werte "number", "string", "boolean", "undefined", "function" und "object".

Verwendet man nun *typeof* für Zahlen, kommt bei Ganz- und Fließkommazahlen immer der Datentyp „number" zurück. Da Zahlen in JavaScript intern ein Länge von 64 Bit haben und sie nach dem IEEE Standard 754 implementiert ist, sind sie auch als „doubles" bekannt. Mit ihnen lassen sich sowohl Integer als auch Fließkommazahlen im Bereich von -9.007.199.254.740.992 bis 9.007.199.254.740.992 repräsentieren. JavaScript verzichtet daher auf den expliziten Integer-Typ. Auch eine gemischte Verwendung ist möglich, was aber zu unerwünschten Ergebnissen führen kann, wie folgendes Beispiel zeigt. (Eine Liste der möglichen Operatoren findet sich in Tabelle 2.)

```
var x = 0.1;
var y = 0.2;
var z = x+y;
```

Ein eventuell erwartetes Ergebnis von *z = 0.3* kommt nicht zustande. Da es sich um eine Fließkommazahl handelt, ergibt die Rechnung 0.30000000000000004. Das lässt sich durch einen kleinen Trick umgehen. Wenn zum Beispiel mit einer Währung wie Euro gerechnet wird, nutzt man als Basis Cent statt Euro.

Auch JavaScript kennt Arrays, deren Deklaration in Tabelle 1 zu finden ist. Beim Abfragen eines einzelnen Elements ist zu beachten, das der Index mit 0 beginnt. Im in der Tabelle aufgeführten Beispiel zum Zugriff wird folglich der Wert „grün" zurückgegeben. Möchte man die Werte des folgenden Zahlen-Arrays sortieren, bietet sich die Methode *n.sort();* an.

```
var n = [4, 8, 15, 16, 23, 42];
```

Nach dem Aufruf gibt sie das Ergebnis n ist 15, 16, 23, 4, 42, 8 zurück. JavaScript hat das Array folglich nicht nach Zahlen sortiert, sondern von A bis Z. Da *sort* auch Sortierfunktionen entgegennimmt, lässt sich das schnell ändern. Definiert man

```
function sortfunction(a, b){
  return (a - b)
};
```

und verwendet die Funktionen mit:

```
n.sort(sortfunction);
```

ist das Ergebnis des Sortiervorgangs 4, 8, 15, 16, 23, 42 und das Problem gelöst.

Eine andere Schwierigkeit bei Arrays ist häufig das Löschen von Elementen. Ist beispielsweise der Array

```
var myArray = ['a', 'b', 'c', 'd'];
```

gegeben und man entfernt das zweite Element per *delete*-Anweisung (*delete myArray[1];*) liegt der Inhalt wie folgt vor

```
['a', undefined, 'c', 'd'];
```

Der Wert wurde zwar gelöscht, nicht aber das Element. Daher steht nun an der zweiten Stelle „undefined". Die Alternative dazu stellt der Befehl *splice* dar. Die Anweisung

```
myArray.splice(1,1);
```

Tabelle 2: Operationen in JavaScript

Operation	Operator
Stringkonkatenation	+
Zuweisung	=
Addieren und zuweisen	+=
Subtrahieren und zuweisen	-=
Dividieren und zuweisen	/=
Multiplizieren und zuweisen	*=
Addition	+
Subtraktion	-
Multiplikation	*
Division	/
Inkrementieren	++
Dekrementieren	–
Negation	-
gleich	==
nicht gleich	!=
strenges Gleich	===
größer als	<
kleiner als	>
größer oder gleich	< =
kleiner oder gleich	> =
NOT	!
logisches OR	||
bitweises OR	|
AND	&&
Bit nach links	<<
Bit nach rechts	>>
nach rechts und mit Nullen füllen	>>>
bitweises Komplement	~
Typabfrage	typeof
ist Instanz von	instanceof
Löschen	delete
Test	in
Erzeugen	new
Evaluieren	void
aktuelles Objekt	this

spaltet das Array an der Stelle Index =1 auf und entfernt das Element, sodass danach nur noch ['a','c','d']; im Array steht.

Von Schleifen und Fällen

Kontrollstrukturen sind sicherlich die meist benutzen Elemente einer Sprache. Wenn etwas einer Kondition (die Wahr oder Falsch ergeben muss) entspricht, wird ein Block mit Code ausgeführt, oder eben ein anderer. Einer der prominentesten Vertreter ist sicherlich die Fallunterscheidung per *if*.

```
var stunde = 8;
if (stunde === 12) {
   alert("Es ist Mittag");
} else {
   alert("Es ist kein Mittag")
}
```

Hier wird geprüft, ob die Variable *stunde* den Wert 12 besitzt und mit wahr evaluiert. Da dies nicht der Fall ist, führt das Skript den *else*-Zweig aus. Beim Vergleich ist zu beachten, dass kein einfaches Gleichheitszeichen zum Einsatz kommt, da es sonst eine Zuweisung wäre. Der Code ließe sich weiter ausbauen, um auch andere Fälle abzudecken. Dafür ist der *else if*-Block zu wiederholen.

```
var stunde = 8;
if (stunde===12) {
   alert("Es ist Mittag");
} else if (stunde===18){
   alert("Es ist Abend")
} else {
   alert("Es ist nicht Mittag")
}
```

Alternativ dazu gibt JavaScript die Möglichkeit, einen *switch* zu nutzen:

```
var stunde = 18;
switch (stunde) {
   case 8:
      alert("Es ist Morgen");
   case 12:
      alert("Es ist Mittag");
   case 18:
      alert("Es ist Abend");
   default:
      alert("Kein Treffer");
}
```

Sicherlich ist der so geschriebene Code kompakter und besser lesbar. Der Switch bekommt die betreffende Bedingung übergeben, im Beispiel die Variable *stunde*. Die jeweiligen Case-Abschnitte prüfen auf den Inhalt (8,12,18). Der *default*-Zweig wird ausgeführt, wenn kein Case zutrifft. Weitere Schleifen in JavaScript sind *do/while* und *while/do*.

```
var i=0;
do {
   i += 1;
   alert(i);
} while (i < 5);
```

Die Schleife erhöht *i* in jeder Iteration um eins und gibt den aktuellen Wert per *print* aus. Dabei läuft die Schleife so lange *i* kleiner fünf ist. Wichtig ist, das sie auf jeden Fall einmal zu durchlaufen ist, da die Bedingung erst am Ende geprüft wird. Alternativ gibt es die *while*-Schleife ohne *do*. Sie durchläuft das Programm so lang, bis die angegebene Bedingung erfüllt ist.

```
var n = 0;
while (n < 3) {
  n++;
  alert(n);
};
```

Fragezeichen und Funktionen

Nachdem die grundlegenden Datentypen und Operatoren sowie die Herangehensweise zu deren Deklaration erklärt ist, ist noch der ternäre Operator *?* zu beachten. Es handelt sich dabei um eine Art kurze Schreibweise für ein *if*-Statement. Das folgende Codefragment

```
var result;
if (a > b) {
  result = x;
}
else {
  result = y;
}
```

lässt sich mit dem Operator auf ein simples

```
var result = a > b ? x : y;
```

Listing 1: Nutzung eines Callbacks

```
function calculate1To1000(callback) {
  var result = 0;
  for (var i = 1; i <= 1000; i++){
      result += i;
  }
  callback(result);
}

var cb = function(data){
   print(data);
};

calculate1To1000(cb);
```

verkürzen. Zu beachten ist aber, das hier keine *else-if*-Zweige möglich sind.

Das Salz in der JavaScript-Suppe sind zweifellos Funktionen. In diesem Zusammenhang spricht man auch oft von „First Class"- oder „First Order"-Funktionen.

```
function addition(x,y){
  return x+y;
};
```

Durch das Schlüsselwort *function* wird eine neue Funktion erstellt. Sie erwartet die Parameter *x* und *y* beim Aufruf, die der Funktion dann als lokale Variable zur Verfügung stehen. Die Return-Anweisung gibt einen Wert zurück. Im Beispiel also die Summe der zwei Werte *a* und *b*. Die Funktion lässt sich wie folgt verwenden.

```
var wert1 = 3;
var wert2 = 7;
addition(wert1, wert2);
```

Wie erwartet ist das Ergebnis 10. Allerdings sollte man beim Erstellen von Funktionen eher den neuen Deklarationsstil

```
var addition = function(x,y){
  return x+y;
};
```

verwenden. Der Aufruf ist immer noch derselbe. Sicherlich verwundert es etwas, dass eine Variable, hier *addition*, auch eine Funktion aufnehmen kann. Aber genau das ist der Vorteil von JavaScript. Ein Nachteil ist hingegen der *return*-Wert, da er nur eine Angabe zulässt. Sollten mehrere Werte benötigt werden, ist ein Objekt oder Array zu nutzen.

Die Besonderheit von Callbacks

Die sogenannten Callbacks gehören zu den Besonderheiten von JavaScript. Da die Sprache asynchron arbeiten soll, muss es eine Möglichkeit geben, blockierende Zugriffe aufzulösen. Dabei springt man aus der aktuellen Funktion heraus.

In Listing 1 ist zu sehen, wie die Funktion *calculate1To100* einen Callback nutzt. Sie soll nur von 1 bis 100 zählen ohne zu wissen, wie das Ergebnis weiter zu verarbeiten ist. Genau dafür lässt sich ein Callback nutzen. Die Funktion *calculate1To100* erledigt ihre Aufgabe und ruft dabei immer wieder die Funktion *cb* auf, die als Callback übergeben wurde. Diese nimmt Daten entgegen und gibt sie aus. Damit sind Callbacks eine Art „Mach Du das mal und sag Bescheid, wenn Du fertig bist".

In Listing 2 sieht man den Aufbau der Funktion *eineFunktion*, die als Parameter eine URL und einen Callback entgegennimmt. Als Erstes erstellt man nun ein normales *XMLHttpRequest*-Objekt. Für den Internet-Explorer weicht die Herangehensweise dafür leider ab, weshalb sie per *if* zu berücksichtigen ist. In *onreadystatechange* wird vor dem Callback sichergestellt, dass er nur bei einer erfolgreichen Verbindung ausgeführt wird. Dafür lassen sich *readystate* und die *status*-Eigenschaft verwenden. Man ruft eine noch nicht benannte Funktion auf und übergibt ihr den Inhalt des *httpRequest* als XML. Es ist nicht definiert wann der Aufruf passiert. Erst nach dem erfolgreichen Verbinden und Abrufen der Daten löst der Callback aus. Es sollte dabei zu keinerlei Blockierungen kommen, sodass der Benutzer normal weiterarbeiten kann. Beim Aufruf der Funktion in den folgenden drei Zeilen übergibt man zunächst eine URL (*text.xml*) und eine anonyme Funktion als Callback. Letztere tut nichts weiter, als eine Logausgabe auf der Konsole mit den übergebenen Daten (*this*) zu

erstellen. Dies geschieht asynchron, sobald die Daten vorhanden sind.

Keine Sprache ohne Objekte!

Sind komplexe Datentypen gewünscht, gelangt man schnell zu Objekten. In JavaScript handelt es sich hierbei intern nur um eine Liste aus Schlüsseln und Werten, die sich aber von außen wie Objekte verwenden lassen:

```
var zaphod = {
  name:'Beeblebrox',
  age:42,
  numberOfHeads: 2;
};
```

Die erzeugte Variable *zaphod* enthält nun Eigenschaften wie *name*, die den Wert *Beeblebrox* hat. Der Zugriff auf das Objekt und seine Eigenschaften erfolgt nach dem Muster Objekt.Eigenschaft:

```
zaphod.name;
zaphod.age;
```

oder alternativ via

```
zaphod['name'];
zaphod['age'];
```

Dabei ist das Objekt nicht explizit von einem „Root-Objekt“ abgeleitet. Das ist nicht nötig – einer der Vorteile von JavaScript.

Allerdings kann das Objekt nicht nur Eigenschaften haben, auch Methoden sind möglich:

```
var zaphod = {
  name:'Beeblebrox',
  age:42,
  numberOfHeads: 2,
  greetings: function() {
    return "Hallo, ich heisse " + this.name;
  }
};
```

Jetzt führt ein Aufruf von *zaphod.greetings();* zum Ergebnis: „Hallo, ich heisse Beeblebrox“.

Ist mehr als ein Objekt mit denselben Eigenschaften zu erstellen, kommen Konstruktoren zum Einsatz.

```
function User(name, age, numberOfHeads) {
  this.name = name;
  this.age = age;
  this.numberOfHeads = numberOfHeads;
  this.greetings = function() {
    return "Hallo, ich heisse " + this.name;
  }
}
```

Der gezeigte Codeausschnitt erstellt eine Funktion mit dem Namen *User*. Die Großschreibung des Funktionsnamen ist nicht willkürlich – Objektnamen sind in JS-Kreisen der Konvention nach groß zu schreiben. Der Konstruktorteil ist abgeschlossen und lässt sich nun per *new* instanziieren. Möchte man die Instanz *arthurdent* erstellen, ist der folgende Aufruf vonnöten:

```
var arthurdent = new User("Arthur", "33", 1);
```

Ein Zugriff ist ohne Weiteres via *arthurdent.name;* möglich und auch die Methode *greetings()* lässt sich nutzen. Zum Erweitern des *User*-Objekts benötigt man sogenannte Prototypen.

Das Nutzen von *prototype*

Normalerweise würde eine neue Eigenschaft beziehungsweise Methode für die Konstruktor-Funktion *User* direkt in dieser hinterlegt werden. Um allerdings ein Objekt abzuleiten oder ihm zur Laufzeit neue Eigenschaften hinzuzufügen, ist *prototype* zu nutzen. Mit dem Befehl

Listing 2: Callback als Parameter

```
function eineFunktion(url, callback) {
  var httpRequest; // erstelle das XMLHttpRequest Object
  if (window.XMLHttpRequest) {
    httpRequest = new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    httpRequest = new
      ActiveXObject("Microsoft.XMLHTTP");
  }

  httpRequest.onreadystatechange = function() {
    if (httpRequest.readyState === 4 &&
        httpRequest.status === 200) {
      callback.call(httpRequest.responseXML);
    }
  };
  httpRequest.open('GET', url);
  httpRequest.send();
}

eineFunktion("text.xml", function() {
  console.log(this);
});
```

```
User.prototype.dimension = 3;
```

erhält das *User*-Objekt beispielsweise die Eigenschaft *dimension* mit dem Wert 3, die alle zugehörigen Instanzen erben. So kann man auch vorhandene Objekte wie *Objekt* einfach erweitern. Eine Methode, um das Datum von Ostern zu berechnen, ließe sich wie folgt realisieren:

```
Date.prototype.Easter = function(Y) {
    var C = Math.floor(Y/100);
    var N = Y - 19*Math.floor(Y/19);
    var K = Math.floor((C - 17)/25);
    var I = C - Math.floor(C/4) - Math.floor((C - K)/3) + 19*N + 15;
    I = I - 30*Math.floor((I/30));
    I = I - Math.floor(I/28)*(1 - Math.floor(I/28)*Math.floor(29/(I + 7
                                          1))*Math.floor((21 - N)/11));
    var J = Y + Math.floor(Y/4) + I + 2 - C + Math.floor(C/4);
    J = J - 7*Math.floor(J/7);
    var L = I - J;
    var M = 3 + Math.floor((L + 40)/44);
    var D = L + 28 - 31*Math.floor(M/4);
  return padout(D) +  .' + padout(M); }
function padout(number) { return (number < 10) ?  0' + number : 7
                                                          number; }
```

Die nun bekannte Berechnung kann der Entwickler beispielsweise so nutzen:

```
d = new Date(); // Fri Nov 01 2013 05:09:02 GMT-0700 (UTC)
d.Easter(2012)  // "08.04" für das Jahr 2012
d.Easter(2013)  // "31.03" für das Jahr 2013
d.Easter(2014)  // "20.04" für das Jahr 2014
```

Jedoch ist Vorsicht geboten: Wird eine vorhandene Eigenschaft mit dem gleichen Namen über *prototype* hinzugefügt, ist sie nicht geschützt, sondern wird überschrieben! Daher ist vorher immer zu prüfen, ob bereits eine Eigenschaft mit dem gewählten Namen existiert.

Als Beispiel für die Vererbung soll Marvin der Roboter dienen. Er hat zusätzlich zu den Eigenschaften des *User*-Objekts noch die Eigenschaft „Hersteller“.

```
function Robot(manufacturer) {
     this.manufacturer = manufacturer;
};
// Robot extends User
Robot.prototype = new User();
marvin = new Robot(); // Instanz marvin aus Robot erzeugen
```

Damit sind nach einer Instanziierung von *Robot* die Eigenschaften *manufacturer*, *name*, *age*, *numberOfHeads*, *greetings: function* aus *User* und *Robot* enthalten.

Nach dem Erstellen von Variablen, Funktionen und Objekten ist deren Gültigkeitsbereich noch nicht geklärt. Ein *Private* oder *Public* fehlt bei der Deklaration. Das heißt, einmal dekla-

riert ist die Variable von überall erreichbar und nicht vor Zugriff geschützt. Das gilt auch beim Laden mehrerer JavaScript-Dateien in denen derselbe Variablenname genutzt wird. Es gibt allerdings Ausnahmen von dieser Regel.

Aus dem Kontext genommen

Eine Unterscheidung zwischen lokalem und globalem Kontext liegt bei Funktionen vor. Ein Parameter/Argument einer Funktion lebt privat für sich im Funktionsaufruf und wird danach zerstört. Selbiges gilt für dort deklarierte Variablen. Legt man außerhalb der Funktion eine Variable mit demselben Namen fest, benutzt die Funktion immer die lokale Variante aus ihrem Funktionskörper:

```
var meineVariable1 = 20;
var meineVariable2 = 50;
function meinTest() {
  var meineVariable1 = 100;
  var lokal = 2000;
  meineVariable2 = 15;
 }
```

Im obigen Beispiel sind die Variablen *meineVariable1* und *meineVariable2* global erstellt und haben so im gesamten Skript Gültigkeit. Es folgt die Funktion *meinTest()*, in der man die Variable *meineVariable1* per *var* erzeugt. Damit hat sie zwar denselben Namen wie die globale, ist aber lokal vorhanden. Das Erstellen der Variable *lokal* erzeugt ebenfalls eine einfache Variable, die nur innerhalb der Funktion existiert. Zum Schluss weist man der globalen Variable *meineVariable2* den Wert 15 zu und überschreibt damit den alten Wert 50.

Es ist also entscheidend zu wissen, dass globale Variablen entstehen, wenn sie nicht durch *var* deklariert wurden, bei einer Deklaration außerhalb aller Funktionen sowie bei einer Wertzuweisung innerhalb einer Funktion, ohne vorherige Deklaration der Variable. Durch das Vergessen des Schlüsselworts vor einem Variablennamen erstellt man also ungewollt eine globale Variable, die auch das Verlassen der Funktion überlebt.

Wenn die Anwendungen größer werden, wächst häufig der Wunsch nach „Modulen“. Um zu verhindern, dass der gewählte Name der Variablen oder Funktion bereits anderweitig existiert, empfiehlt das jQuery-Projekt beispielsweise das im Folgenden beschriebene Konzept der Self Invoking oder Immediate Functions (IFFY) für das Entwickeln von Plug-ins:

```
(function(){
  //hier steht dann was...
})();
```

Das sieht etwas fremd aus, ist aber schnell erklärt. Man erstellt zunächst eine anonyme Funktion. Sie enthält den normalen JavaScript-Code und ruft sich am Ende per *()* selbst auf.

Der Trick besteht darin, dass der Interpreter den kompletten Inhalt genau einmal parst und er sich danach einfach aufrufen lässt. Fehler würden so auch schon beim Einbinden entdeckt. Kombiniert man das Vorgehen mit einem sogenannten Namespace, ist der Weg der sauberen Entwicklung mit JavaScript gegeben.

Erstellen eines Namensraums

Um Namensräume zu erzeugen, kommen leere Objekte zum Einsatz, die sich nach dem Muster *var myapp = {};* erstellen lassen und gleichzeitig den Root-Namespace angeben. Durch Nachstellen eines Postfixes (zum Beispiel *myapp.math = {};*) lässt er sich erweitern. Der Namespace für die eigentliche Funktion lässt sich wie folgt deklarieren:

```
myapp.math.add = function(a,b){
  return a+b;
};
```

Sie lässt sich später mit

```
myapp.math.add(10,20); // 30
```

aufrufen. Listing 3 und 4 zeigen ein Beispiel, in dem Doppelungen von Funktionsnamen ausgeschlossen sind, da man sie innerhalb des Objektes (Namensraum) kapselt.

Fazit

JavaScript ist vergleichsweise einsteigerfreundlich – zumindest am Anfang. Durch den geringen Umfang der Menge der Schlüsselwörter wirkt es erst einmal sehr einfach.

Dass die Sprache nicht statisch typisiert ist und auch nicht vorkompiliert wird, birgt sicherlich Potenzial für Fehler. Dafür ist sie universell einsetzbar und bringt durch die ständige Weiterentwicklung immer mehr Verbesserungen mit sich. Damit ist sie einen Blick für alle Entwickler wert – und wie bei jeder Sprache hilft zum Erfolg nur üben, üben, üben! (jul)

Listing 3: index.html

```
<html>
  <head>
    <title>Namespace</title>
    <script src='mathMod.js'>
    <script>
    var erg = myapp.math.add(10,20)
    alert(erg);
  </script>
  </head>
<body> </body>
</html>
```

Listing 4: mathMod.js

```
(function(){

  myapp.math = {};

  myapp.math.add = function(a,b){
    return a+b;
  };

  myapp.math.multi = function(a,b){
    return a*b;
  };

})();
```

Literatur

[1] Stefan Koch; JavaScript: Einführung, Programmierung und Referenz; dpunkt.verlag; 2011

[2] David Herman; JavaScript effektiv; dpunkt.verlag; 2013

Marcus Ross
ist Dipl.-Informatik-Betriebswirt und arbeitet als freier IT-Berater und Trainer mit den Schwerpunkten Datenbanken und Webportale. Seit 2010 widmet er sich dem Thema der mobilen Cross-Plattform-Entwicklung und ist zudem PhoneGap-Comitter.

Find your connection to us!

TNG TECHNOLOGY CONSULTING

JavaScript als Multiparadigmen-Sprache

Alles drin

Manfred Steyer

Prozedural, funktional, objektorientiert und modular – JavaScript kann alles unter einen Hut bringen. Wer sich mit den unterschiedlichen Aspekten der Sprache auskennt, erleichtert sich die Arbeit mit ihr ungemein.

Um JavaScript, aber vor allem populäre JavaScript-Frameworks, effektiv nutzen zu können, muss man sich vor Augen halten, dass es sich hierbei nicht – wie vielerorts vermutet – um eine Teilmenge des objektorientierten Java, sondern vielmehr um eine eigenständige Sprache handelt, die mehrere Paradigmen in sich vereint.

Die prozedurale Seite von JavaScript

Das wohl bekannteste Paradigma, das sich in JavaScript wiederfindet, ist der prozedurale Ansatz: Befehle lassen sich zu Funktionen kombinieren und bei Bedarf zur Ausführung bringen. Der Name Funktion wird dabei unabhängig von der Tatsache verwendet, ob ein Wert zurückgeliefert wird oder nicht.

Listing 1: Einfache JavaScript-Funktion

```
<html>
<head>
    <title></title>

    <script language="javascript">

        function calcZins(k, p, t) {
            var result = k * p * t / 36000;
            return result;
        }

        var result = calcZins(200,2,360);
        alert("Ergebnis: " + result);

    </script>
</head>

</html>
```

Listing 1 zeigt ein Beispiel dafür. In ihm ist die Funktion *calcZins* definiert. Da es sich bei JavaScript um eine dynamische Sprache handelt, werden weder die Typen der verwendeten Parameter beziehungsweise Variablen noch der Datentyp des Rückgabewerts festgelegt. Allerdings gehört es zum guten Ton, Variablen mit dem Schlüsselwort *var* zu deklarieren und ihren Gültigkeitsbereich so auf die aktuelle Funktion (und nicht, wie man denken könnte, auf den aktuellen Block) zu beschränken. Das führt im betrachteten Fall dazu, dass *result* lediglich innerhalb der Funktion *calcZins* bekannt ist. Verwendet der Entwickler eine Variable, ohne sie vorher zu deklarieren, sieht JavaScript sie als global an. Aus Gründen der Wartbarkeit ist davon allerdings abzuraten.

Anders als bei Sprachen wie Java oder C#, muss der Entwickler nicht sämtliche Parameter beim Aufruf einer Funktion angeben. Parameter, die nicht übergeben wurden, weisen innerhalb der Methode den Wert *undefined* auf. Da JavaScript-Interpreter die Werte *undefined*, *0* und *null* bei logischen Vergleichen

Listing 2: Optionale Parameter

```
function calcZins(k, p, t) {

    if (!t) t = 360;
    // if (typeof t == 'undefined') t = 360;


    var result = k * p * t / 36000;
    return result;
}
```

wie *false* sowie alle anderen Werte wie *true* behandeln, kann der Entwickler, wie in Listing 2 gezeigt, sehr einfach prüfen, ob ein Wert (der von den drei genannten abweicht) für einen Parameter vorliegt. Um direkt auf den Wert *undefined* zu prüfen, lässt sich, wie mit dem Kommentar in Listing 2 angedeutet, unter anderem der Operator *typeof* heranziehen.

Die funktionale Seite von JavaScript

Die Tatsache, dass es sich bei JavaScript auch um eine funktionale Sprache handelt, dürfte weniger bekannt sein. In Listing 3 erwartet die Funktion *forEach* zwei Parameter: einen Array und eine Funktion, die für jeden Array-Eintrag aufzurufen ist. Das ist möglich, da unter JavaScript Funktionen gleichzeitig auch Objekte sind. Als solche lassen sie sich sowohl Variablen zuweisen, als auch an andere Funktionen weiterreichen.

Bei *showItem* handelt es sich um eine weitere Funktion, die den übergebenen Wert darstellt und *myInts* ist ein Array mit vier vordefinierten Werten, wobei auffällt, dass JavaScript-Arrays eckige Klammern verwenden. Interessant wird es, wenn man den Aufruf von *forEach* betrachtet: Als erstes Argument übergibt das betrachtete Beispiel das Array, als zweites die Funktion *showItem*. Hätte der Entwickler an der Stelle stattdessen *showItem()* (man achte auf die runden Klammern) geschrieben, würde der JavaScript-Interpreter die Funktion an dieser Stelle ausführen und ihren Rückgabewert an *forEach* übergeben.

Der Entwickler kann Funktionen, die an andere übergeben werden, auch an Ort und Stelle definieren ohne einen Namen zu vergeben. Zudem besteht die Option, Variablen Funktionen zuzuweisen. Wird die Variable wie eine Funktion behandelt, indem ihr runde Klammern nachgestellt werden, führt der JavaScript-Interpreter die Funktion aus, auf die die Variable verweist.

Ähnlich wie mit Lambda-Ausdrücken in C# bildet sich bei Einsatz des Schlüsselwortes *function* ein sogenannter Closure (Funktionsabschluss, siehe dazu auch Seite 19). Das bedeutet, dass Funktionen Zugriff auf alle Variablen haben, die sich in sämtlichen übergeordneten Gültigkeitsbereichen (Scopes) befinden.

Die Funktion *createLamp* in Listing 4 erzeugt zum Beispiel eine weitere, die sie der Variablen *controller* zuweist. Die neue Funktion hat nun Zugriff auf die Variable *dimFactor*, zumal sie in einem übergeordneten Gültigkeitsbereich definiert wurde. Deswegen und aufgrund der Tatsache, dass man die Funktion zurückliefert, zerstört der Interpreter die lokale Variable nach der Ausführung von *createLamp* nicht. Vielmehr findet sie bei jedem Aufruf der zurückgelieferten Funktion Verwendung. Somit lässt sich eine „private Variable“ schaffen, die im betrachteten Fall nur dieser Funktion zur Verfügung steht und von außerhalb nicht direkt einzusehen oder zu verändern ist.

Die objektorientierte Seite von JavaScript

Eine vergleichsweise einfache Möglichkeit zum Definieren von Objekten stellt der Einsatz der Java Script Object Notation (JSON) dar (Listing 5). Sie beschreibt Objekte durch geschweifte Klammern und die zugehörigen Eigenschaften in Form von Name/Wert-Paaren, die durch einen Doppelpunkt zu trennen sind. Bei den Werten kann es sich um atomare Werte handeln (etwa Strings, die in Anführungszeichen zu halten sind), um weitere Objekte (in geschweiftem Klammern) oder um Arrays (in eckigen Klammern). Dasselbe gilt für Werte, die sich innerhalb von Arrays wiederfinden.

Listing 3: Die Funktion *forEach* erwartet für den Parameter *action* eine weitere Funktion

```
function forEach(ary, action) {
    for (var i = 0; i < ary.length; i++) {
        action(ary[i]);
    }
}

function showItem(item) {
    alert(item);
}

var myInts = [1, 2, 3, 4];

forEach(myInts, showItem);
forEach(myInts, function (item) {
    alert(item);
});

var fn = function (item) {
    alert(item);
}

forEach(myInts, fn);
fn("Hallo Welt");
```

Listing 4: Einsatz von Closures

```
function createLamp(lampId) {
    var dimFactor = 0;

    var controller = function (command) {
        switch (command) {
            case 'dimUp':
                if (dimFactor < 100) {
                    dimFactor = dimFactor + 10;
                }
                alert(lampId + ": " + dimFactor);
                break;

            case 'dimDown':
                if (dimFactor > 0) {
                    dimFactor = dimFactor - 10;
                }
                alert(lampId + ": " + dimFactor);
                break;

            case 'dimFactor':
            default:
                return dimFactor;

        }
    }

    return controller;
}

var lamp1 = createLamp("Lamp #1");
var lamp2 = createLamp("Lamp #2");

lamp1("dimUp");
lamp1("dimUp");
lamp1("dimUp");
lamp1("dimDown");

lamp2("dimUp");
```

Listing 5: Objekte mit JSON beschreiben

```
var flugBuchung = {
    Von: "Graz",
    Nach: "Mallorca",
    Passagiere: [
        {
            Vorname: "Max",
            Nachname: "Muster"
        },
        {
            Vorname: "Susi",
            Nachname: "Sorglos"
        }
    ],
    Bezahlung: {
        Art: "Kreditkarte",
        Betrag: 250,
        Bezahlt: true
    }
};

alert(flugBuchung.Von + " - " + flugBuchung.Nach);
alert("Bezahlt: " + flugBuchung.Bezahlung.Bezahlt);
alert("Anzahl Passagiere: " + flugBuchung.Passagiere.length);
alert("Nachname 1. Passagier: " + flugBuchung.PassagiereEO].
                                                    Nachname);
```

Listing 6: Konstruktorfunktion

```
function Person(vorname, nachname) {
    var that = this;
    that.vorname = vorname;
    that.nachname = nachname;

    that.vollerName = function () {
        return that.vorname + " " + that.nachname;
    }
}

var rudi = new Person("Rudolf", "Rentier");
alert(rudi.vorname);
alert(rudi.nachname);
alert(rudi.vollerName());
```

Anders als die meisten objektorientierten Sprachen, kennt JavaScript das Konzept der Klassen (noch) nicht (siehe Seite 34). Als Alternative kommen sogenannte Konstrukturfunktionen zum Einsatz (Listing 6). Eine Konstruktorfunktion hat die Aufgabe, ein Objekt zu erzeugen. Dazu weist sie jene Eigenschaften, die das zu erzeugende Objekt aufweisen soll, der Pseudovariablen *this* zu. Eigenschaften können dabei auf Funktionen (Methoden) verweisen, die das zu erzeugende Objekt anbieten soll. Da sich der Wert von *this* im Zuge des Ausführens der Konstruktormethode ändern kann, hat es sich eingebürgert, ihn vorm Durchlaufen der Funktion in einer anderen Variablen abzulegen und fortan diese zu verwenden.

Listing 6 definiert zur Demonstration eine Konstruktormethode *Person*, die die Variable *that* als Ersatz für *this* verwendet und dem zu erzeugenden Objekt die Eigenschaften *vorname*, *nachname* sowie *vollerName* zuweist, wobei letztere auf eine Funktion verweist.

Um ein Objekt mit einer Konstruktormethode zu erzeugen, ist die Methode, wie am Ende des Listings demonstriert, mit dem Operator *new* aufzurufen. Das Ergebnis weist das Beispiel der Variablen *rudi* zu. Über den Punktoperator greift es anschließend auf die Eigenschaften des Objektes zu und führt die Funktion *vollerName* aus.

Die Eigenschaften, die innerhalb der Konstruktormethode via *this* zugewiesen werden, lassen sich mit öffentlichen Membern in Java oder C# vergleichen – jeder hat darauf Zugriff. Um private Member, auf die in JavaScript lediglich das Objekt selbst zugreifen darf, zu definieren, sollte man sich daran erinnern, dass das Schlüsselwort *function* eine Closure erzeugt, die ihren Kontext einschließt. Eine Variable, die der Entwickler innerhalb der Konstruktormethode definiert, lässt sich folglich von sämtlichen Funktionen des erzeugten Objektes verwenden. Bedingung ist allerdings, dass sie, wie im betrachteten Beispiel, innerhalb der Konstruktormethode hinterlegt sind, was in Listing 7 zu sehen ist. Während die Eigenschaft *inhaber* dort über *this* zugewiesen wird und somit öffentlich ist, handelt es sich bei *guthaben* und *protokoll* um Variablen der Konstruktormethode. Somit sind sie außerhalb dieser nicht sichtbar, lassen sich jedoch als private Variablen von den einzelnen Methoden verwenden.

Ein weiteres wissenswertes Detail zu Objekten unter JavaScript ist die Tatsache, dass sie streng genommen assoziative Arrays („Dictionaries“) sind, die die Namen der Eigenschaften auf deren Werte abbilden. Demonstrieren lässt sich das unter anderem durch Listing 8, das auf die Eigenschaften eines Kontos sowohl über den Punktoperator als auch über die von Arrays bekannte Schreibweise zugreift. Da es sich jedoch um ein assoziatives Array handelt, kommen für die Indizes Zeichenfolgen zum Einsatz.

Das bedeutet aber auch, dass man lediglich ein Objekt anlegen muss, um ein assoziatives Array zu erhalten. Um sämtliche Schlüssel zu ermitteln, lässt sich die im betrachteten Listing gezeigte *for…in*-Schleife heranziehen. Im Gegensatz zur *forEach*-Schleife in Java oder C# iteriert sie nicht die Werte sondern tatsächlich die Menge der Schlüssel. Deswegen sind im betrachteten Fall die Werte durch erneuten Zugriff auf das Array via *punkte [key]* in Erfahrung zu bringen.

Vererbung in JavaScript

Zum Aufbau von Vererbungshierarchien bietet sich in JavaScript unter anderem die Verwendung von Prototypen an. Ein Prototyp ist ein Objekt, das einer Konstruktorfunktion zugewiesen ist. Kann der JavaScript-Interpreter eine Eigenschaft in einem Objekt nicht finden, durchsucht er dessen Prototyp (genauer: den Prototyp der Konstruktormethode). Wird er dort nicht fündig, setzt er die Suche beim Prototyp des Prototyps fort – dieses Spielchen wiederholt sich, bis er am Ende der Kette angekommen ist.

Listing 9 spendiert auf dem Weg der von JavaScript vorgegebenen Konstruktorfunktion *Array* eine zusätzliche Funktion *each*. Instanziiert der Entwickler nun ein neues *Array*-Objekt und ruft *each* auf, wird der JavaScript-Interpreter innerhalb des Objekts nicht fündig. Aus diesem Grund durchsucht er anschließen den Prototyp und stößt dort auf die dort eingerichtete Funk-

Listing 7: Einsatz einer Konstruktorfunktion

```
function Konto(inhaber, startGuthaben) {
    var that = this;
    that.inhaber = inhaber;

    var guthaben = startGuthaben;
    var protokoll = [];

    var addToProtocol = function (action, betrag, text) {

        // push fügt ein neues Element am Ende des Arrays ein

        protokoll.push({
            aktion: "ein",
            betrag: betrag,
            text: text
        });
    }

    that.einzahlen = function (betrag, text) {
        addToProtocol("EIN", betrag, text);
        guthaben += betrag;
    }

    that.abheben = function (betrag, text) {
        addToProtocol("AUS", betrag, text);
        guthaben -= betrag;
    };

    that.getGuthaben = function () {
        return guthaben;
    }

    that.getProtokoll = function () {
        return protokoll;
    }

}

var myKonto = new Konto("Susi Sorglos", 200);

myKonto.einzahlen(50);
myKonto.abheben(30);

var guthaben = myKonto.getGuthaben();
var protokoll = myKonto.getProtokoll();

// Sinnlos, denn das vom Konto verwendete Guthaben ist
// in einer anderen Variablen 'guthaben' gespeichert ...
// myKonto.guthaben = 33;

alert(guthaben);
```

Listing 9:
Den Prototyp einer bestehenden Klasse erweitern

```
Array.prototype.each = function (fn) {
    var ary = this;
    for (var i = 0; i < ary.length; i++) {
        fn(ary[i]);
    }
}

var myArray = new Array(3, 2, 1);

myArray.each(function (item) { alert(item); });

var ok = myArray instanceof Array;
```

tion, die er daraufhin ausführt. Um herauszufinden, ob ein Objekt einen bestimmten Typ aufweist, kann der Entwickler den Operator *instanceof* verwenden. Im betrachteten Listing lässt sich damit prüfen, ob es sich bei der Variable *myArray* um ein Objekt des Typs *Array* handelt. Das ist der Fall, wenn *myArray* direkt vom Typ *Array* ist oder wenn sich *Array* in der Kette der Prototypen von *myArray* befindet.

Beim Verwenden von Prototypen ist darauf zu achten, dass sämtliche Objekte eines Typs auf ein und dasselbe Prototyp-Objekt verweisen. Weist der Prototyp einen Zustand auf, teilt er ihn mit sämtlichen Objekten, die ihn nutzen. Dies entspricht statischen Variablen in Sprachen wie Java oder C#. Attribute, die nicht statisch sein sollen, sind somit erneut im erbenden Objekt zu definieren. Für vom Prototyp „geerbte" Funktionen gilt das jedoch nicht, da JavaScript den Wert *this* beim Ausführen einer Funktion im Prototyp nach wie vor auf das eigentliche Objekt zeigen lässt. Somit verwenden Funktionen, die sich im Prototyp befinden, den Zustand des gerade verwendeten Objektes.

Alternative Herangehensweise

Ein weiteres Beispiel für die Umsetzung von Vererbung findet sich in Listing 10. Hier wird sie zunächst ohne die Verwendung von Prototypen realisiert. Stattdessen ruft die Konstruktor-Funktion *GiroKonto* die Konstruktor-Funktion *Konto* auf. Das geschieht jedoch nicht auf dem herkömmlichen Weg, sondern durch Verwendung der Funktion *call*. An sie übergibt der Entwickler als ersten Parameter jenen Wert, der im Rahmen der Abarbeitung von Konto über *this* erreichbar sein soll. Da das betrachtete Beispiel das *this* von *GiroKonto* übergibt, richtet die gerufene Funktion *Konto* im Objekt, das *GiroKonto* konstruierten wird, jene Eigenschaften ein, die auch ein *Konto* aufweist. Somit existieren sämtliche Eigenschaften des *Kontos* im *GiroKonto*. Das wiederum bewirkt, dass jedes *GiroKonto*-Objekt seinen eigenen Kontostand hat, statt sich gemeinsam mit allen anderen den Kontostand des Prototypens teilen zu müssen. Der Aufruf der Konstruktor-Methode *Konto* lässt sich mit dem des Super-Konstruktors in Sprachen wie Java oder C# vergleichen, zumal er das Initialisieren der von Konto geerbten Aspekte bewirkt.

Zusätzlich definiert das betrachtete Beispiel für Girokonten eine eigene Methode *abheben* und überschreibt somit die geerbte. Da die Methode jedoch jene, die sie überschrieben hat, aufruft, sichert es zuvor deren Wert, indem *abheben* der Variablen *base_abheben* zugewiesen wird.

Der Vollständigkeit halber folgt im Code das Festlegen einer neuen Instanz von *Konto* als Prototyp von *GiroKonto*. Da die Konstruktorfunktion *GiroKonto* danach *Konto* aufruft, und somit *GiroKonto*-Objekte sämtliche Eigenschaften eines Kontos besitzen, wäre dies nicht zwingend notwendig. Allerdings erlaubt es dem Entwickler, zur Laufzeit über *instanceof* heraus-

zufinden, dass ein *GiroKonto* ein *Konto* ist. Außerdem erhält so jedes GiroKonto die Funktionen, die er – analog zu Listing 9 – nachträglich dem Prototyp von Konto hinzufügt. Dabei muss der Entwickler jedoch beachten, dass solche Funktionen keinen Zugriff auf über Closures realisierte private Variablen, wie den Kontostand, haben.

Die modulare Seite von JavaScript

Sprachen, wie Java oder C#, verwenden Namespaces zum Organisieren von Klassen. Ein direktes Gegenstück dazu existiert in JavaScript zwar nicht, allerdings lässt sich das Konzept durch Objekte nachbilden (siehe auch Seite 19). In Listing 11 wird auf diese Weise ein Namespace *sample.oop.konto* erstellt. Eine weitere Herangehensweise zum Definieren von Namensräumen folgt etwas später. Da Funktionen ihren eigenen Gültigkeitsbereich (Scope) besitzen, lassen sie sich zum Definieren von Modulen, die voneinander abgeschottet sind, heranziehen. In der Regel soll jedes Modul jedoch nur einmal existieren. Deswegen hat sich hierfür der Einsatz von anonymen Funktionen, die sofort nach ihrer Deklaration ausgeführt werden, eingebürgert. Das ist mit der Schreibweise

```
(function() { ... })();
```

zu erreichen. Das erste äußere Klammernpaar umschließt die Funktion und das abschließende bringt es zur Ausführung, wie der folgende Codeausschnitt zeigt:

```
(function () {
    var info = "Hallo Welt";
    function sum(a, b) { return a + b; }
    function alertInfo() { alert(info); }
})();
```

Sowohl die Variable *info* als auch die beiden Methoden *sum* und *alertInfo* sind nur innerhalb des Moduls sichtbar – außerhalb existieren sie nicht.

Da ein Modul, das lediglich private Inhalte hat, nicht sonderlich nützlich ist, muss ein Weg gefunden werden, um ausgewählte Teile des Moduls öffentlich zugänglich zu gestalten. Eine Lösung stellt das Verwenden eines Übergabeparameters dar. Übergibt man ihm ein Objekt, lässt sich der Code innerhalb des zugehörigen Moduls um alle Funktionen und Eigenschaften anreichern, die öffentlich verfügbar sein sollen. Ein Beispiel dafür ist in Listing 12 abgebildet.

Listing 10: Vererbung über Prototypen

```
function GiroKonto(inhaber, startGuthaben, limit) {
    var that = this;
    Konto.call(this, inhaber, startGuthaben);

    var base_abheben = that.abheben;

    that.abheben = function (betrag, text) {
        base_abheben(betrag, "*" + text + "*");
    }
}

GiroKonto.prototype = new Konto();

var gk = new GiroKonto("susi", 100, 300);
gk.abheben(99, "test");
alert(gk.getGuthaben());
```

Listing 11: Namespaces über Objekte nachbilden

```
// Definition
sample = {};
sample.oop = {};
sample.oop.konto = {};
sample.oop.konto.GiroKonto = GiroKonto;
sample.oop.konto.Konto = Konto;
sample.oop.konto.Dauerauftrag = function (von, zu, betrag) {

    var that = this;

    that.von = von;
    that.zu = zu;
    that.betrag = betrag;

    that.ausfuehre = function () {
        // what ever ...
    }
}

// Verwendung

var kto = sample.oop.konto; // Alias

var gk = new kto.GiroKonto("Susi Sorglos", 200, 200);
gk.einzahlen(50, "Taschengeld von der Oma ...");
```

Listing 12: Ausgewählte Elemente veröffentlichen

```
var tools = {};

(function (root) {
    var info = "Hallo Welt";

    root.sum = function(a, b) { return a + b; }
    root.sayHello = function() { alert(info); }
})(tools);

var sum = tools.sum(1,2);
alert(sum);
tools.sayHello();
```

Fazit

JavaScript vereint das prozedurale und modulare Paradigma mit der funktionalen und objektorientierten Programmierung. Um den eigenen Code strukturieren zu können und bestehende Frameworks besser zu verstehen, ist der Entwickler angehalten, sich damit zu beschäftigen. Aufgrund semantischer Schwächen von JavaScript werden Sprachelemente, die der Entwickler aus anderen Sprachen kennt – darunter Module, Namensräume, Lambdaausdrücke und Klassen –, mit Funktionen unter Verwendung etablierter Muster nachgebildet. Diese gilt es zu beherrschen und in bestehenden Code-Strecken zu erkennen. (jul)

Manfred Steyer
ist freiberuflicher Trainer und Berater bei IT-Visions sowie verantwortlich für den Fachbereich „Software Engineering“ der Studienrichtung „IT und Wirtschaftsinformatik“ an der FH CAMPUS 02 in Graz. In seinem aktuellen Buch „Moderne Webanwendungen mit ASP.NET MVC und JavaScript-APIs“ beschreibt er unter anderem den Einsatz von offlinefähigen Webanwendungen.

Quelle

Der vorliegende Artikel ist ein überarbeiteter Auszug aus dem Buch:

Manfred Steyer, Dr. Holger Schwichtenberg

Moderne Webanwendungen mit ASP.NET MVC

Microsoft Press Deutschland, Mai 2013