Prof. Dr. Friedrich Steimann

Objekt-funktionale Programmierung

Lehrveranstaltung 63618 Version 1♂

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Inhaltsverzeichnis

Lektion 2: Objekte

Ein gängiger Leitspruch der objektorientierten Programmierung lautet:

> Alles ist ein Objekt.

Wie das bei so einfachen Leitsprüchen häufig der Fall ist, lässt auch dieser ein weites Interpretationsspektrum zu. An dessen einem Ende kann er bedeuten, dass mit dem Begriff des Objekts nicht besonders viel verbunden ist. In diesem Fall bedeutet es nicht viel, wenn alles darunter fällt. Am anderen Ende kann es bedeuten, dass alle Elemente eines Programms auf derselben Stufe stehen, also alle gleich behandelt werden können. In diesem Fall wären nicht nur Daten, sondern auch Kontrollstrukturen, Variablen und Funktionen Objekte, die durch ein Programm, selbst ein Objekt, manipuliert werden können. Eine derartige Homogenität ist jedoch kaum zu erreichen: So ist auch im Lambda-Kalkül nicht alles eine Funktion — es gibt auch Variablen und Funktionsanwendungen. Entsprechend verhält es sich in der objektorientierten Programmierung.

Gleichwohl sind Objekte für die objektorientierte Programmierung nicht nur namensgebend, sondern zentral: Sie stellen die dominierende, wenn nicht sogar einzige, *Datenstruktur* dar und kapseln zudem *Verhalten*, also das, was man mit Funktionen verbindet. Tatsächlich ist die Struktur eines objektorientierten Programms üblicherweise durch die Struktur seiner Objekte bestimmt.

Diese Lektion befasst sich mit den definierenden Charakteristika der objektorientierten Programmierung. Es sind dies:

- Objekte haben *Identität* und *Struktur*.
- Objekte sind mit anderen verbunden und bilden somit einen *Objektgraphen* (ein *Objektgeflecht*).
- Objekte zeigen *Verhalten*.
- Objekte können Verhalten und Struktur *erben*.

Für die objekt-funktionale Programmierung kommt definierend hinzu:

- Funktionen sind Objekte.

Das ist eine ganze Menge. Wir beginnen mit der Struktur von Objekten.

4 Objekte als Strukturen

Wir nähern uns dem Begriff des Objekts zunächst über Beispiele und nutzen dafür die Programmiersprache JAVASCRIPT. Sie erlaubt uns, den Begriff des Objekts unabhängig von dem der *Klasse* einzuführen, was der Didaktik dieses Lehrtextes entgegenkommt. Einige andere Besonderheiten (im Vergleich zu den anderen in diesem Text verwendeten Programmiersprachen) nehmen wir dabei zur Kenntnis.


WIKIPEDIA

Als Sprachstandard für JAVASCRIPT ziehe ich hier ECMAScript heran. Die Beispiele in dieser Lektion sind in der Read-Evaluate-Print-Loop (REPL) von NODE.JS entstanden; NODE.JS läuft, anders als das ursprüngliche JAVASCRIPT, nicht in der Umgebung eines Browsers[48], sondern bring seine eigene Umgebung mit Standardein- und -ausgabe mit sich. Doch Achtung: Es gibt zwischen der REPL und dem Interpreter leichte Unterschiede in der Syntax von Ausdrücken (insbesondere Objektliteralen), die möglicherweise erklären, warum das eine oder andere hier angeführte Beispiel nur in der REPL, nicht aber in einer Entwicklungsumgebung funktioniert.

Objekte als identitätsbehaftete Container von Name/Wert-Paaren

In JAVASCRIPT ist ein Objekt ein identitätsbehafteter Container für eine Menge von Eigenschaften (engl. properties), die das Objekt beschreiben. Dabei wird jede Eigenschaft durch ein Name/Wert-Paar repräsentiert. So steht beispielsweise

```
12 {                                                                    JS
13     vorname: 'Hein',
14     nachname: 'Blöd',
15     alter: 42,
16     hallo: function() { console.log('Käpt'n, Käpt´n!') },
17     name: function() { return this.vorname + ' ' + this.nachname }
18 }
```

in JAVASCRIPT für ein Objekt mit den aufgelisteten (und durch Kommata getrennten) Eigenschaften. Objekte entsprechen damit zunächst der Datenstruktur des Records (s. Abschnitt 2.1.6) mit der Besonderheit, dass hier auch Funktionen als Objekteigenschaften auftreten können.[49] Funktionen sind in JAVASCRIPT nämlich (wie im Lambda-Kalkül) Werte; so stehen die den Namen `hallo` und `name` zugeordneten Funktionen gleichberechtigt neben den Zeichenketten (Strings), die den Namen `vorname` und `nachname` zugeordnet sind, wie auch neben der Zahl, die `alter` zugeordnet ist. Auch handelt es sich bei beiden Funktionen um *anonyme Funktionen* (wiederum wie im Lambda-Kalkül); Namen erhalten sie (genau wie die Strings) lediglich über ihre Rolle als Eigenschaften eines Objekts (hier die Eigenschaften `hallo` und `name`).[50] Zu-

[48] Gleichwohl verwendet es die JAVASCRIPT-Engine V8 des Chromium-Projekts, auf dem viele Web-Browser basieren.

[49] Den einen oder anderen Leser werden JAVASCRIPT-Objekte eher an Dictionaries oder Hash maps erinnern und tatsächlich sind sie das ihrem Wesen nach auch; allerdings sind Dictionaries oder Hash maps in der Regel selbst Objekte, so dass dieser Vergleich hier nicht weiterhilft.

[50] Im allgemeinen Sprachgebrauch wird oft der Name einer Eigenschaft eines Objekts (also die erste Komponente eines Name/Wert-Paars, hier beispielsweise `vorname`) als Eigenschaft betrachtet. Man

dem sind beide Funktionen parameterlos (erkennbar an den leeren Klammern hinter dem Schlüsselwort `function`); sie verfügen jedoch über einen für die objektorientierte Programmierung wesentlichen impliziten Parameter `this`, von dem die zur Eigenschaft `name` gehörende Funktion auch Gebrauch macht (`this` qualifiziert dort den Zugriff auf die Eigenschaften `vorname` und `nachname` im Rumpf der Funktion). Der Parameter `this` (in JAVASCRIPT ein Schlüsselwort) steht dabei für das Objekt, dem die Funktion als Eigenschaft zugeordnet ist.

Identität von Objekten

Damit ein Container von Name/Wert-Paaren ein Objekt *ist* und nicht nur (wie ein Record) eines beschreibt, wird ihm eine **Identität** zugeordnet, die es von anderen Objekten, selbst solchen mit gleichen (oder identischen) Eigenschaften, unterscheidet. So hat auch ein eigenschaftsloses, oder leeres, Objekt `{}` in JAVASCRIPT eine Identität, die es von anderen leeren Objekten unterscheidet:

```
19 > {} == {}                                                    JS
   false
```

Hierbei steht `==` für den Test auf Identität und nicht auf Gleichheit (wie es beispielsweise bei `2 == 2` der Fall wäre). Dass Objekte als strukturierte Daten eine Identität haben, die sich von Gleichheit unterscheidet, ist eines der zentralen Wesensmerkmale der objektorientierten Programmierung (das andere ist die *Vererbung*). Ich werde in Kapitel 6 noch ausführlich darauf zurückkommen.

Objekte als Werte von Eigenschaften

Genau wie Records können Objekte (als Datenstrukturen) geschachtelt werden. So definiert auch

```
20 {                                                             JS
21     name: 'Hein'
22     chef: {
23         name: 'Blaubär'
24     }
25 }
```

ein Objekt. Wie wir noch sehen werden, ist durch die Schachtelung aber, anders als bei Records, *keine Teil-Ganzes-Beziehung* begründet (vgl. dazu die Abschnitte 2.1.6 und 2.1.7).

Arten von Eigenschaften: Felder und Methoden

Wie die obigen Beispiele andeuten, können die Werte der Eigenschaften von Objekten in JAVASCRIPT primitive Werte wie Zahlen oder Strings, Objekte oder Funktionen sein. Wenn man der in der imperativen Programmierung üblichen Einteilung in Daten und Funktionen folgen will, dann fallen die ersten beiden zusammen unter Daten (wobei primitive Werte ein Objekt beschreiben und Objekte Beziehungen des Objekts zu anderen ausdrücken). In der funktionalen Programmierung sind jedoch auch Funktionen Daten. Entsprechend werden in der objekt-funktionalen Sprache JAVASCRIPT Eigen-

spricht beispielsweise von „Farbe“ oder „Alter“ als Eigenschaften von Objekten („das Auto hat eine Farbe“), ohne dass man den Wert der Eigenschaft (also beispielsweise blau) dazusagen würde (wobei ja nach obiger Definition ja das Paar „Farbe/blau“ erst die Eigenschaft wäre). Ich mache das hier auch. Später (in Lektion 3) werden wir sehen, dass diese doppelte Bedeutung des Wortes Eigenschaft (Existenz und Ausprägung) sich auch in den Konstrukten der objektorientierten Programmierung materialisiert.

schaften nicht grundsätzlich in solche, die einem Objekt Daten zuordnen, und solche, die ihm Funktionen zuordnen, unterteilt. In einer klassisch objektorientierten Diktion nennt man jedoch die Eigenschaften, die Daten mit einem Objekt verbinden, seine *Felder* (oder seine *Instanzvariablen*) und die, die Funktionen mit ihm verbinden, seine *Methoden*. Methoden unterscheiden sich von den Funktionen der nicht objektorientierten, imperativen und funktionalen Programmierung dadurch, dass ihnen stets ein impliziter Parameter zur Verfügung steht, der je nach Sprache `this` oder `self` heißt (`this` in JAVASCRIPT) und der stets an das Objekt gebunden ist, dem die Methode zugeordnet ist. Wie wir noch sehen werden, können dies bei verschiedenen Aufrufen derselben Methode durchaus verschiedene Objekte sein.

Bevor wir uns mit der Erzeugung von Objekten und ihrer Verwendung in einem Programm (dem typisch Objektorientierten) befassen, schauen wir uns die imperative Basis objektorientierter Sprachen am Beispiel von JAVASCRIPT an.

5 Imperative Basis objektorientierter Sprachen

Obwohl objektorientierte Sprachen in der Regel starke funktionale Anteile haben (und somit objekt-funktional genannt werden müssen), haben die meisten einen imperativen Kern. Dies äußert sich u. a. darin, dass Programme als eine Folge von Anweisungen definiert sind und dass Variableninhalte überschrieben werden können. Zudem verfügen sie neben Objekten (und der Behauptung, in ihnen sei alles ein Objekt, zum Trotz) zumeist auch über *primitive Werte* (das sind Werte, die eben keine Objekte sind).

Wir nähern uns der imperativen Basis objektorientierter und objekt-funktionaler Sprachen wieder am Beispiel von JAVASCRIPT.

5.1 Anweisungen

Wie imperative Sprachen auch verfügt JAVASCRIPT über *Deklarations-* und *Kontrollflussanweisungen*. Außerdem können Ausdrücke zu Anweisungen befördert werden, indem man sie mit einem Semikolon abschließt; es entsteht so eine *Ausdrucksanweisung* (engl. expression statement; vgl. Abschnitt 2.1.5). So werden insbesondere Wertzuweisungen, die in JAVASCRIPT Ausdrücke sind, zu Anweisungen:

```
26 x = 1;                                                                    JS
```

beispielsweise ist eine solche Ausdrucksanweisung. Auch Deklarationsanweisungen, von denen wir Beispiele in den Abschnitten 5.3 (Variablen) und 5.4 (Funktionen) sehen werden, werden, sofern sie nicht mit einer geschweiften Klammer („}“) enden, durch ein Semikolon abgeschlossen; allerdings können in JAVASCRIPT Semikolons auch häufig weggelassen werden, wobei sie dann durch den Compiler automatisch (und für den Programmierer unsichtbar) ergänzt werden. Dies kann aber auch zu Fehlinterpretation führen und ist daher mit Vorsicht zu genießen.

Gleichwohl lasse ich sie in den nachfolgenden Beispielen meistens weg, um die Darstellung von „syntaktischem Rauschen“ zu befreien.

Blöcke und Geltungsbereiche

Ein mittels eines Paars geschweifter Klammern[51] gebildeter Block gilt ebenfalls als eine Anweisung. Blöcke fassen somit syntaktisch mehrere Anweisungen zu einer zusammen; es dient dies (im Zusammenspiel mit Kontrollflussanweisungen), der *strukturierten Programmierung*. Zugleich bilden Blöcke Geltungsbereiche für Deklarationen.

Kontrollflussanweisungen

Die Kontrollflussanweisungen von JAVASCRIPT sind im Wesentlichen dieselben, die man auch von anderen imperativen und objektorientierten Programmiersprachen her kennt, also `if` und `if...else`, `switch...case...default`, `for`, `while`, `do...while`, `break`, `continue`, `throw` sowie `try...catch` und `try...catch...finally`. Sie können über die Ausführung einzelner Anweisungen oder ganzer Blöcke bestimmen. Hinzu kommt der *Unterprogrammaufruf* in Form eines zu einer Anweisung *beförderten* Funktionsaufrufs (der imperativen Form der *Funktionsanwendung*; s. Abschnitt 2.1.5 sowie weiterführend Abschnitt 5.4).

Ein alleinstehendes Semikolon bedeutet die leere Anweisung.

5.2 Primitive Werte

Es gibt in JAVASCRIPT derzeit sechs Arten von primitiven Werten, von denen jede durch einen String, im JAVASCRIPT-Kontext *Typ*, hier aber *Datentyp* genannt, repräsentiert wird:

- Zahlen wie `1` oder `2.4`, repräsentiert durch `'number'`
- beliebig große (nur durch den Speicher begrenzt) Ganzzahlen wie `0n`, repräsentiert durch `'bigint'`
- Zeichenketten wie `"abc"` oder `'string'` (einfache und doppelte Anführungsstriche haben dieselbe Bedeutung), repräsentiert durch `'string'`
- Symbole, repräsentiert durch `'symbol'`
- die Wahrheitswerte `true` und `false`, repräsentiert durch `'boolean'`,
- der Wert `undefined`, repräsentiert durch `'undefined'`.

Große Zahlen und Symbole sind erst nachträglich hinzugekommen und nicht wirklich gut integriert; so würden beispielsweise die Datentypen (wie `'number'`, `'string'` etc.) besser durch Symbole als durch Strings repräsentiert. Auch gibt es zwar einen Datentyp für `null`, der aber offenbar

[51] Die Verwendung geschweifter Klammern sowohl für Objektliterale als auch für Anweisungsblöcke ist etwas unglücklich, zumal in JAVASCRIPT-Code Objektliterale häufig vorkommen und diese Bedeutungsüberladung gelegentlich zu Fehlinterpretationen führen kann. In solchen Fällen kann ein Objektliteral durch zusätzliche runde Klammern als solches kenntlich gemacht werden, also beispielsweise wie in `({})`.

(noch) nicht verwendet wird; stattdessen ist der Datentyp von `null` `'object'`, der Datentyp von Objekten. Objekte sind aber keine primitiven Werte.

Der Datentyp eines Werts kann in JAVASCRIPT mit dem Operator `typeof` abgefragt werden:

```
27 > typeof null                                                    JS
   'object'
```

dynamisch typisierte Sprache

Datentypen haben in JAVASCRIPT keine tiefergehende Bedeutung. Insbesondere werden sie nur sehr bedingt, und dann auch nur zur Laufzeit, zu einer *Typprüfung* herangezogen. Sprachen, die das tun, werden manchmal auch als **dynamisch typisierte Sprachen** bezeichnet. In diesen Sprachen werden die Typen von Werten — als **Laufzeittypinformation** — vornehmlich zur Steuerung des Kontrollflusses verwendet, wobei hier sowohl die Umsetzung von Werten in einen anderen, passenden Datentyp als auch Programmabbrüche mit Typfehlermeldungen eingeschlossen sind. Da die meisten Sprachen ohne statisches Typsystem aber ohnehin Laufzeittypinformation zur Speicherverwaltung benötigen, ist der Schritt von einer (scheinbar) untypisierten zu einer dynamischen typisierten Sprache nur ein kleiner. Dagegen ist die Einführung von „echten“ (gemeint: statischen) Typen in TYPESCRIPT (Lektion 4) eine ganz andere Sache.

Strings sind Werte

Anders als beispielsweise in JAVA und anderen objektorientierten und objekt-funktionalen Sprachen zählen Strings in JAVASCRIPT zu den primitiven Werten. Da sie aber (genau wie die Bigints und Symbole) beliebige Größen annehmen können, werden Strings intern anders behandelt als beispielsweise (normale) Zahlen.

kein eigener Datentyp für Integer

Bei den Zahlen des Datentyps `'number'` wird nicht grundsätzlich zwischen Ganz- und Fließkommazahlen unterschieden; dennoch sind an manchen Stellen (so bei der Indizierung von Arrays) nur ganze Zahlen zulässig. Das ist spätestens dann ein Problem, wenn sich bei der Berechnung eines Indexes die für Gleitkommazahlrechnungen typischen Rundungsfehler einschleichen. Da JAVASCRIPT aber ganze Zahlen bis zur Größe von $2^{53} - 1$ als Ganzzahlen repräsentiert, können in diesem Bereich Rundungsfehler vermieden werden, solange man nicht teilt.

Primitive Werte (einschließlich Strings) sind nicht änderbar. Was das genau bedeutet, darauf kommen wir noch zu sprechen.

5.2.1 Literale Repräsentationen primitiver Werte

Primitive Werte mit Ausnahme von Symbolen haben in JAVASCRIPT vorgegebene Notationen, mit deren Hilfe sie buchstäblich („literal“) in einen Programmtext hineingeschrieben werden können. Man nennt Ausdrücke in diesen Notationen auch *Literale* (vgl. Abschnitt 2.1.5).

Zahlliterale

Zahlliterale, also Darstellungen von Zahlen im Programmtext, sind die üblichen vorzeichenbehafteten Ziffernfolgen einschließlich Dezimalbrüchen (mit dem Punkt als Komma) sowie der Exponentialschreibweise (z. B. `1.2e3` statt `1200`). Zusätzliche Zahlliterale sind `NaN` (für not a number) und `Infinity`.

Stringliterale

Stringliterale sind alle in einfachen oder doppelten Anführungszeichen eingeschlossenen Zeichenfolgen einschließlich der leeren Zeichenfolge. Strings dürfen die üblichen Escape-Sequenzen enthalten. Zusätzlich bietet JAVASCRIPT mit seinen sog. *String templates* auch Stringliterale mit eingebetteten Ausdrücken, die bei jeder ihrer Verwendungen neu ausgewertet werden (die sog. *Stringinterpolation*), weswegen sie strenggenommen keine Literale sind.

Jeder String hat eine Länge; die maximale Länge ist $2^{53} - 1$ und der leere String hat die Länge 0. Jedes Zeichen eines Strings wird durch einen UTF-16-Character dargestellt.

weitere Literale

Die literalen Darstellungen der beiden Wahrheitswerte sind `true` und `false`; weitere Literale sind `undefined` und `null`. Dabei ist „undefined“ auch der Name einer globalen Eigenschaft (einer Eigenschaft des *globalen Objekts*; s. Abschnitt 7.6), die aber in neueren Versionen von JAVASCRIPT zumindest nicht mehr schreibbar (`writable`; s. Abschnitt 7.5) ist.

5.2.2 Ausdrücke über primitive Werte

Wie in anderen Sprachen auch können Literale primitiver Werte in Ausdrücken vorkommen, also beispielsweise durch Operatoren wie `+`, `-`, `*` und durch `/` verbunden werden. Anders als viele andere Sprachen versucht JAVASCRIPT dabei stets, „das Beste“ aus einem Ausdruck zu machen, also selbst solche Ausdrücke auszuwerten, die von anderen Sprachen als fehlerhaft zurückgewiesen würden. So haben wir beispielsweise

```
28 > '1' + 0                                                    JS
   '10'
```

(da `+` auch auf Strings definiert ist und dort als Konkatenation verstanden wird) oder auch

```
29 > undefined + 1                                              JS
   NaN
```

und

```
30 > null + 1                                                   JS
   1
```

was jeweils auf ein *Zwingen* (engl. *coercion*) des „falschen“ Operanden in einen passenden Wert zurückzuführen ist. Wie man leicht erkennt, liegt hier eine gewisse Beliebigkeit vor, was leicht zu unerwarteten Ergebnissen und aufwendigen Fehlersuchen führen kann.

Allgemein führt das automatische Zwingen von Operanden in einen von einer Operation erwartet Datentyp wie in den obigen Beispielen dazu, dass bei der Ausführung der Operation

kein Laufzeitfehler auftritt, was in Sprachen mit statischer Typprüfung durch einen entsprechenden Typfehler zur Übersetzungszeit ebenfalls verhindert wird. Das automatische Zwingen basiert jedoch auf einer impliziten Annahme, die im Einzelfall auch falsch sein kann, nämlich dass das Zwingen gewollt war und nicht einen Programmierfehler wie die Angabe eines falschen Operanden, beispielsweise durch Verwechselung von Variablennamen, kaschiert. Was also in einem Fall als ungemein praktisch erscheint, kann in einem anderen eine schmerzhafte Fehlersuche nach sich ziehen.

unechte Wahrheitswerte

Geradezu auf die Spitze getrieben wird das Zwingen von JAVASCRIPT bzgl. der Wahrheitswerte `true` und `false`, an deren Stelle es auch nahezu beliebige andere Werte als „truthy“ und „falsy“ akzeptiert: Objekte, alle Zahlen außer `0` und alle Strings außer dem leeren als truthy; `0`, `NaN`, `""`, `undefined` und `null` als falsy. Damit aber nicht genug: Logische Operatoren wie `&&` (für logisch und) und `||` (für logisch oder, jeweils mit einer sog. Short-circuit-Auswertung) akzeptieren somit nicht nur beliebige Werte als Operanden, sondern geben auch noch einen der beiden Operanden unverändert als Ergebnis zurück. So ergibt sich beispielsweise

```
31 > 0 && ''                                                    JS
   0
32 > '' && 0
   ''
```

sowie umgekehrt

```
33 > 0 || ''                                                    JS
   ''
34 > '' || 0
   0
```

Was eigentlich ein Boolescher Ausdruck sein sollte, ist in Wahrheit eine konditionale Identitätsfunktion. Auch dies mag manchmal praktisch erscheinen (weil es zu knapperen Ausdrücken führt), ist aber höchst fehleranfällig. Es ist somit ein typisches Beispiel für ein ewiges Dilemma beim Entwurf von Programmiersprachen: Während Ausdrucksmöglichkeiten wie die obigen cool sind und wesentlich zur Popularität einer Sprache beitragen, kann der Spaß auch schnell mal sehr teuer werden. In technischen Disziplinen wie der Programmierung sind hölzerne Ausdrucksweisen nicht selten die besseren.

5.2.3 Gleichheit von primitiven Werten

Eine wichtige Operation jeder Programmiersprache ist die *Gleichheitsoperation*, mit der zwei Werte auf Gleichheit geprüft werden können und die damit die *Gleichheitsrelation* implementiert. Während `=` in JAVASCRIPT (wie in vielen auf der C-Syntax basierenden Sprachen, so auch JAVA und C#) für die *Wertzuweisung* steht, gibt es in JAVASCRIPT zwei Gleichheitsoperatoren: `==` und `===`. Im Kontext von Werten unterscheiden sie sich darin, dass `==` die Operanden vor ihrem Vergleich bzgl. ihres Datentyps angleicht, so dass also etwa

```
35 > 1 == "1"                                                   JS
```

```
true
```

ergibt. Im Gegensatz dazu steht

```
> 1 === "1"                                                    JS
false
```

In Abschnitt 7.8 werden wir sehen, was die Unterscheidung von zwei Arten von Gleichheit für Objekte bedeutet.

5.3 Variablen und Konstanten

Wie praktisch alle anderen Programmiersprachen auch hat JAVASCRIPT Variablen, die im Programm für Werte stehen können und die somit zu den *Ausdrücken* zählen. Dabei handelt es sich bei den Variablen JAVASCRIPTs um die Variablen einer *imperativen Programmiersprache*, also um Variablen, deren Wert nach der ersten Zuweisung durch weitere Zuweisungen überschrieben werden kann (s. Abschnitt 2.1.2). Sie sind also zu unterscheiden von den Variablen deklarativer Sprachen (und den Variablen des Lambda-Kalküls; vgl. Abschnitt 3.4.5).

Anders als in vielen anderen Programmiersprachen müssen Variablen in JAVASCRIPT vor ihrer Verwendung nicht *deklariert* (*vereinbart*) werden. Sie sind dann dennoch keine *freien Variablen* (s. Abschnitt 3.3.6), sondern werden bei ihrer ersten Verwendung implizit angelegt, und zwar als *globale Variablen*.[52] Da solche Formen der *impliziten Vereinbarung* gern auch mal versehentlich geschehen (durch simples Vertippen) und globale Variablen generell verpönt sind, sollten Sie sie vermeiden. Wie ernst zu nehmen dieser Ratschlag ist, kann man nicht zuletzt daran ablesen, dass JAVASCRIPT einen sog. Strict mode (per Direktive `'use strict'`) vorsieht, der die Verwendung nicht zuvor deklarierter Variablen zurückweist.

Variablendeklarationen

In JAVASCRIPT gibt es nämlich auch die Möglichkeit, Variablen zu deklarieren, und zwar entweder per Var- oder per Let-Anweisung, wobei in beiden Fällen der Variable bei ihrer Deklaration gleich ein Wert zugewiesen werden kann:

```
var x = 1                                                      JS
let y = "a"
```

Bleibt dies aus, erhält sie den Wert `undefined` (was ein definierter Zustand ist; der Wert der Variable ist also durchaus gegeben und nicht undefiniert im eigentlichen Sinne). Ein Unterschied zwischen let- und var-deklarierten Variablen besteht darin, dass let-deklarierte Variablen erst initialisiert und damit zugreifbar werden, nachdem die Let-Anweisung ausgeführt wurde (vorher ist ihr Wert tatsächlich undefiniert), wohingegen var-deklarierte Variablen schon vorher initialisiert sind (und zwar mit `undefined`):

```
> var x = x                                                    JS
```

[52] Je nach verwendeter Version von JAVASCRIPT handelt es sich dabei tatsächlich um Felder eines *globalen Objekts* (s. Abschnitt 7.6).

```
40 > x
   undefined
41 > let y = y
   Uncaught ReferenceError: Cannot access 'y' before initialization
```

Die Deklaration mit `var` entspricht also in gewisser Weise dem Letrec aus Abschnitt 3.4.6.

Geltungsbereich von Variablen

Allen Variablen von JAVASCRIPT ist ein *Geltungsbereich* (engl. scope) zugeordnet, der sich aus dem Programmaufbau (seiner statischen Struktur; das sog. *Static* oder *Lexical scoping*; s. Abschnitt 2.1.3), dem Ort, an dem die Variable deklariert wird, und dem dabei verwendeten Schlüsselwort ableitet (der globale Geltungsbereich, wenn die Deklaration implizit erfolgt). So ist beispielsweise der Geltungsbereich von `x` in

```
42 { let x = 1; … }                                                    JS
```

auf den die Deklaration enthaltenden Block beschränkt; stünde dort stattdessen

```
43 { x = 1; … }                                                        JS
```

(und ist `x` nicht außerhalb des Blocks explizit deklariert), dann wäre der Geltungsbereich von `x` global. Variablen, die mit `var` deklariert werden, sind in ihrer Gültigkeit auf die umgebende Funktion beschränkt (unabhängig von Blöcken innerhalb der Funktion; s. Abschnitt 5.4) und können darin auch vor ihrer Deklaration verwendet werden (mit `var` deklarierte Variablen, jedoch nicht auch ihre Initialisierung, werden an den Anfang ihres Geltungsbereich hochgezogen; das. sog. *Hoisting*); sie dürfen innerhalb einer Funktion auch mehrfach (oder re-) deklariert werden.

Konstanten

Wenn einer Variable nur einmal ein Wert zugewiesen werden soll, dann kann anstelle von `let` auch `const` verwendet werden. Die Gültigkeitsregeln sind dieselben wie bei `let`. Die Wertzuweisung der somit Konstanten[53] muss, als Initialisierung, innerhalb der Deklaration erfolgen; alle anderen (weiteren) Zuweisungen an die Konstante sind ausgeschlossen:

```
44 > const y = 1                                                       JS
45 > y = 2
   Uncaught TypeError: Assignment to constant variable.
```

Konstanten sollte in der objekt-funktionalen Programmierung gegenüber Variablen grundsätzlich der Vorzug gegeben werden, auch wenn der Konstanz hier enge Grenzen gesetzt sind: Dass eine Variable konstant ist, heißt nur, dass ihr kein (weiterer) Wert zugewiesen werden kann, nicht, dass sich der Wert selbst nicht ändert. Letzteres wird nämlich bei Objekten als Werten von Konstanten durch die Verwendung von `const` nicht ausgeschlossen. Somit kann bei Verwendung von `const` die Variablenname vom Compiler (im Rahmen der Optimierung) nur durch eine Referenz (das objektorientierte Äquivalent von Zeigern) substituiert werden und nicht durch das Ziel der Referenz.

[53] Grundsätzlich verwende ich das Wort „Konstante“ wie „Variable“ (vgl. Fußnote 5); dies hier ist eine Ausnahme.

Lebensdauer von Variablen

Die *Lebensdauer* einer Variable hängt nicht strikt an ihrem Geltungsbereich, also dem Block bzw. der Funktion, innerhalb derer sie deklariert wurde (vgl. dazu Abschnitt 2.2.1); sie kann durch *Funktionsabschlüsse*, oder *Closures* (Abschnitt 3.4.4), verlängert werden. Mehr dazu in Abschnitt 8.4.

5.4 Funktionen

Wie wir in Kapitel 4 gesehen hatten, sind Funktionen in JAVASCRIPT zunächst Werte. Als solche haben sie eine literale Repräsentation. Dabei müssen Funktionen nicht anonym bleiben — sie können auch benannt werden. So ist die rechte Seite der Zuweisung

```
46 > f = function quadrat(x) { return x * x }                    JS
   [Function: quadrat]
```

die literale Repräsentation einer mit „quadrat“ benannten Funktion. Allerdings ist dieser Name nur innerhalb der Funktion gültig; er kann also nur zu rekursiven Aufrufen eingesetzt werden (dient aber außerdem noch dem Debuggen mit Werkzeugen, da der Name bei der Ausgabe einer Funktion wie oben zu sehen als String verwendet wird). *Funktionsliterale* wie das obige, die in der objekt-funktionalen Programmierung eines zentrale Rolle spielen, werden in Abschnitt 8.1 ausführlich behandelt.

Funktionsdeklarationen

Steht ein Funktionsliteral wie das obige für sich, wird daraus eine **Funktionsdeklaration**;[54] diese beinhaltet neben der Vereinbarung des Namens auch die *Definition* der Funktion. Die Funktion wird damit im umgebenden *Geltungsbereich* sicht- und unter ihrem Namen aufrufbar. Außerhalb der REPL bedeutet dies, dass die Funktion an Stellen im Programm aufgerufen werden kann, die vor ihrer Deklaration liegen; man nennt das dazugehörige Verfahren, das ein Durchsuchen des Codes vor seiner eigentlichen Interpretation verlangt, auch *Hoisting*. Funktionsdeklarationen können auch ineinander geschachtelt werden, wie etwa in

```
47 function äußere() {                                           JS
48     function innere() { ... }
49     ...
50 }
```

Die Gültigkeit einer solchen **inneren Funktion** ist dann auf den Rumpf der äußeren beschränkt.

lokale Variablen einer Funktion

Funktionen können neben ihren Parametern eigene, lokale Variablen einführen. In JAVASCRIPT sollte man diese vor der Verwendung mit `var` oder `let` deklarieren (für den Unterschied s. Abschnitt 5.3); vergisst man dies, dann bindet entweder die Verwendung der Variablennamen an Deklarationen von Variablen gleichen Namens aus der Umgebung oder, falls nicht vorhanden, es werden implizit neue *globale Variablen* angelegt. Im ersten Fall haben Zuweisungen innerhalb der Funktion unmittelbare Auswirkungen auch au-

[54] Genaugenommen wird das Literal, ein Ausdruck, dabei zu einer *Deklarationsanweisung befördert*.

ßerhalb der Funktion (ein sog. *Nebeneffekt*); im zweiten behalten die Variablen zwischen zwei Funktionsaufrufen ihren Wert. Für den Fall, dass dies gewünscht ist, gibt es allerdings eine bessere Möglichkeit, die wir in Abschnitt 8.5 näher beleuchten werden.

Prozeduren

Zwar gibt es in JAVASCRIPT strenggenommen keine Prozeduren, aber wenn eine Funktion keine Return-Anweisung enthält oder `return` keinen Wert angibt, wird `undefined` zurückgegeben. Solche Funktionen kann man als Prozeduren interpretieren. Allerdings gibt es in JAVASCRIPT kein Call-by-reference, so dass Prozeduren nur über Nebeneffekte eine Wirkung auf den Aufrufkontext haben können. Dazu zählt die Wertzuweisung an globale Variablen.

Funktionsaufrufe

Funktionsaufrufe bestehen aus dem Namen der Funktion gefolgt von einer geklammerten Argumentliste, die auch leer sein kann. Funktionsaufrufe sind Ausdrücke und können als solche Bestandteile von anderen Ausdrücken sein. So ist

```
51 10 + quadrat(4) * 2                                                    JS
```

ein Ausdruck, der den Aufruf der Funktion `quadrat` enthält und zu `42` auswertet. Stünde der Aufruf hingegen für sich wie etwa in

```
52 quadrat(4);                                                            JS
```

dann handelt es sich um eine Anweisung (eine *Ausdrucksanweisung*)[55], die einem Unterprogrammaufruf (Abschnitt 2.1.1) entspricht: Die Funktion wird aufgerufen und somit ausgeführt, das Ergebnis wird verworfen. Insbesondere Prozeduren werden so aufgerufen.

Variabilität der Argumentzahl

Anders als in vielen anderen Programmiersprachen ist in JAVASCRIPT die Zahl der Argumente eines Funktionsaufrufs nicht durch Funktionsdefinition bestimmt. Zwar gibt die Funktionsdefinition Zahl (und Namen) der Funktionsparameter vor, aber ein Aufruf der Funktion kann auch jede andere Zahl an Argumenten angeben. Parameter, für die beim Aufruf keine Argumente vorgesehen werden, werden mit `undefined` belegt, überzählige Argumente ignoriert. Dies bedeutet, dass es in JAVASCRIPT kein *Überladen* von Funktionen (also Funktionen, die sich in ihren Namen gleichen, sich aber anhand ihrer Parameterzahlen unterscheiden) gibt. Die Liste aller Argumente, mit denen eine Funktion aufgerufen wurde (inkl. der überzähligen), ist im Rumpf über die *Pseudovariable* `arguments` zugreifbar.[56]

Default-Argumente

Es ist in JAVASCRIPT möglich, für Parameter in einer Funktionsdefinition Standardwerte vorzugeben, die verwendet werden, wenn bei einem Funktionsaufruf keine Argumente an der Stelle dieser Parameter angegeben werden:

```
53 function inkrement(x, schritt = 1) {                                   JS
```

[55] Auch hier wird ein Ausdruck zu einer Anweisung befördert.

[56] Pseudovariable, obwohl im Kontext von JAVASCRIPT kein gebräuchlicher Term, steht hier für einen Bezeichner (oder Namen), dessen Inhalt variabel ist, wobei der Inhalt vom Laufzeitsystem festgelegt wird und nicht durch explizite Zuweisungen. Im Grunde handelt es sich auch bei `this` um eine Pseudovariable.

```
54   return x + schritt
55 }
```

`inkrement(0, 2)` liefert somit `2`, `inkrement(0)` hingegen `1`. Die Standardwerte dienen somit als **Default-Argumente**.

6 Metaphysik der Objekte

Sowohl in der imperativen als auch in der funktionalen Programmierung nehmen Werte eine zentrale Stellung ein: Ausdrücke, sofern nicht bereits Werte, werten zu Werten aus und die Ergebnisse dieser Auswertungen, Werte, können Variablen zugewiesen bzw. an sie gebunden werden. Werte stellen die Ein- und Ausgaben eines Programms sowie allgemein die Daten, die es verarbeitet. Eine gewisse Sonderrolle unter den Werten nehmen lediglich die Referenzen ein; zwar sind auch diese strenggenommen Werte, jedoch besteht ihr einziger Zweck darin, auf andere Werte zu verweisen. Sie dienen damit dem indirekten Zugriff auf Werte (s. Abschnitt 2.1.7).

6.1 Werte vs. Objekte

Was aber sind Werte? Zahlen beispielsweise sind zunächst mathematische Objekte oder Entitäten und als solche Gegenstand der *Zahlentheorie*. In allen anderen Disziplinen sind sie jedoch nicht selbst Gegenstand der Betrachtung: Hier beschreiben Zahlen — oder vielmehr *Zahlenwerte* — andere Objekte hinsichtlich ihrer quantitativen Eigenschaften wie Anzahl, Größe etc. Ähnliches gilt für andere Werte: Farben wie Blau sind Objekte der Farbenlehre, aber in allen anderen Disziplinen beschreiben sie als Werte andere Objekte („blau“ ist schließlich ein Adjektiv und wird damit einem Objekt zugeschrieben). Dasselbe gilt auch für die Wahrheitswerte: Sie sind Objekte der mathematischen Logik, bewerten aber ansonsten Zuschreibungen von Eigenschaften an Objekte wie etwa „ist frei“ als wahr oder falsch. Wenn in einem Programm die Werte Zwei oder Blau oder Wahr vorkommen (als Eingabe, Ausgabe, oder Wert eines Ausdrucks), dann hat man es in den allermeisten Fällen mit Attributen[57] von Objekten zu tun und nicht mit Objekten selbst: Die meisten Programme beschäftigen sich weder mit Zahlentheorie noch mit Farbenlehre noch mit Logik — sie verwenden deren Objekte ausschließlich als Werte, die andere Objekte beschreiben.

Was aber sind dann Objekte? Zur Beantwortung dieser für die objektorientierte und die objektfunktionale Programmierung zentralen Frage möchte ich weiter ausholen.

Individuen und Universalien

Aus der Philosophie bekannt ist die Unterscheidung zwischen Individuen und Universalien. Wie man die beiden systematisch unterscheiden kann,

[57] Ich verwende hier „Attribut“ nicht im Sinne JavaScripts (Kapitel 4), sondern als Synonym für „Eigenschaft“ in einem modellierenden, oder mathematisch-philosophischen, Kontext. Ein anderes Wort wäre „Merkmal“.

wurde u. a. von dem Philosophen, Mathematiker und Logiker *Bertrand Russell* (dem mit der nach ihm benannten Antinomie) sehr anschaulich dargelegt: Ein Individuum kann zu einem Zeitpunkt nur an genau einem Ort sein, wohingegen ein Universalium zu einem Zeitpunkt an vielen Orten sein kann. Eine bestimmte Person etwa oder ein bestimmter Gegenstand weilt zu einem bestimmten Zeitpunkt an einem bestimmten Ort, eine bestimmte Zahl oder Farbigkeit aber kann man zu einem bestimmten Zeitpunkt an vielen Orten antreffen, nämlich, um beim Beispiel zu bleiben, überall dort, wo eine Eigenschaft eines Individuums mit Zwei oder Blau bemessen wird.[58] Weilt ein Individuum zu einem Zeitpunkt an einem Ort, dann ist zudem dieser Ort zu diesem Zeitpunkt durch das Individuum eingenommen — es können also zwei Individuen nicht zum selben Zeitpunkt am selben Ort sein. Universalien hingegen schon: So sind Blau und Wahr zusammen und gleichzeitig an dem Ort, an dem etwas blau und eine Zuschreibung wahr ist.

Zeit und Raum in der Programmierung

Nun kann (und muss) man in der Philosophie streiten, was ein Zeitpunkt und ein Ort bezeichnen, oder was überhaupt Zeit und Raum sind, aber in der Programmierung müssen wir das zu unserem Glück nicht: Ein Zeitpunkt bezeichnet eine bestimmten Stelle in der Ausführung eines Programms, gedacht als Folge von Zuständen, und ein Ort bezeichnet eine bestimmte Stelle im Speicher, gedacht als eindeutig identifizierbare — adressierbare — Speicherzelle (s. Abschnitt 1.3). Und da bemerken wir Folgendes: Werte (wie Zwei oder Blau oder Wahr) können zu einer Zeit an vielen Stellen im Speicher stehen, nämlich überall da, wo sie Individuen beschreiben. Werte sind also, nach obiger Definition, Universalien.

Objekte als Orte

Wenn die Werte, die ein Programm verarbeitet, Universalien entsprechen, was in einem Programm entspricht dann Individuen? Der Darlegung Russells folgend kann sich etwas, das einem Individuen entsprechen soll, zu jedem Zeitpunkt der Ausführung eines Programms nur in genau einer Speicherzelle befinden, aber dieses Kriterium kann nicht hinreichend sein, weil es ja auch sein kann, dass ein Wert, und damit ein Universalium, zu einem bestimmten Zeitpunkt gerade nur einmal im Speicher vorkommt, dies den Wert aber nicht automatisch zu einem Individuum machen kann, denn die Anzahl seiner Vorkommen kann sich ja jederzeit ändern. Stattdessen können wir ausnutzen, dass ein Ort (eine Speicherzelle) zu jedem Zeitpunkt nur von einem Individuum eingenommen werden kann, so dass der Ort stellvertretend für das Individuum steht, das sich zu diesem Zeitpunkt an ihm befindet. Wenn wir nun noch annehmen, dass sich Individuen nicht bewegen, also Zeit ihrer Existenz immer am selben Ort verweilen (dieselbe Speicherzelle besetzen), dann wird durch die Adresse der Speicherzelle nicht nur ein Ort, sondern auch ein Individuum lebenslang referenziert („adressiert“). Die Annahme der Ortsunveränderlichkeit, wie schräg sie auch erscheinen mag, ist aber tatsächlich kaum eine Einschränkung, denn aufgrund der niedrigen Dimensionalität des Speichers (eindimensional, wenn man Adressen als geordnet annimmt, ansonsten nulldimensional) und der damit einhergehenden eingeschränkten oder gar mangelnden Nachbarschaft gibt es keinen

[58] Das gilt letztlich auch für einen Allgemeinbegriff wie Person: Person (als Allgemeinbegriff) ist überall anzutreffen, wo eine konkrete Person (als Individuum) ist. Mit Allgemeinbegriffen beschäftigen wir uns aber erst ab Lektion 3.

Grund für Individuen, sich zu bewegen.[59] Wir verbinden daher ein Individuum mit einem Ort und nennen diese Einheit **Objekt**.

Inhalt eines Objekts

Wenn aber nun das Objekt unauflösbar mit einer Speicherzelle verbunden ist, was steht dann in der Speicherzelle? Die Antwort mag überraschen, ist aber denkbar einfach: Dort stehen all die Werte (Universalien), die das Objekt beschreiben und die deswegen am Ort des Objekts (des Individuums, das es repräsentiert) anzutreffen sind. Programmiersprachlich betrachtet stehen sie dort nicht ungeordnet oder überlagern sich gar, sondern in Form eines *Records* (s. Abschnitt 2.1.6) oder seines objektorientierten Äquivalents (Kapitel 4): als eine Menge benannter Felder, von denen jedes einzelne einen Wert für die Eigenschaft des Objekts hält, die durch das Feld benannt wird.

Referenzen als Beziehungen

Nun sind auch Referenzen technisch Werte und so bleibt die Frage, wie man Referenzen in unserer Welt von Universalien und Individuen, in der sie technisch Universalien sein müssten, interpretieren soll. Wie schon in Abschnitt 2.1.7 dargelegt, sind Referenzen als Zeiger (Speicher-)Adressen. Sofern sich unter einer Adresse ein Individuum (Objekt) befindet, zeigt die Referenz auf das Individuum (Objekt), dass unter der Adresse aufgefunden wird. Wenn eine Referenz also selbst ein Attribut eines Objekts ist (also in der damit verbundenen Speicherzelle steht), dann verbindet eine Referenz zwei Objekte so, dass man vom einen (dem Inhaber der Referenz) zum anderen (dem Ziel der Referenz) gelangen kann. Man nennt diese Reise auch **Navigation**; sie ist elementarer Bestandteil der objektorientierten Programmierung. Ein Referenzfeld eines Objekts implementiert eine **gerichtete Beziehung** (oder *Relation*) zwischen zwei Objekten, nämlich dem Objekt, zu dem die Referenz gehört, und dem Objekt, auf das die Referenz verweist.

Referenzen als endliche Abbildungen

Eine Referenz, die einem Objekt zugeordnet ist, kann mathematisch als eine *endliche Abbildung* oder *Funktion* betrachtet werden, da sie ein Objekt auf (höchstens) ein anderes abbildet und die Anzahl der abgebildeten Objekte immer endlich ist, die Abbildung also beispielsweise in Form einer Tabelle dargestellt werden kann. Die Funktion ist zudem *benannt*; ihr Name ist der Name des Feldes. Jedes Objekt, das ein Feld mit dem gleichen Namen hat, trägt somit einen, nämlich seinen, Teil zur Definition der Funktion, die diesen Namen trägt, bei. In Lektion 4 werden wir sehen, wie die Definitions- und Wertebereiche von solchen Funktionen systematisch eingeschränkt werden können.

Werte ohne Objekte

Es bleibt die Frage, ob alle Speicherzellen für Objekte stehen oder ob es auch außerhalb von Objekten Werte (einschließlich Referenzwerten), die ja schließlich auch im Speicher stehen und deswegen Speicherzellen belegen müssen, geben kann. Solche wären dann die Werte von Variablen, die keinem Objekt zugeordnet sind, die also keine Felder eines Objekts sind. In JAVASCRIPT müssen wir uns die Frage nicht stellen, da globale Variablen als Felder dem globalen Objekt zugeordnet sind. In anderen Sprachen kann man ein oder mehrere solche Ob-

[59] Es gibt tatsächlich technische Gründe für eine solche Bewegung, nämlich z. B. eine *Speicherbereinigung*. Wenn eine Programmiersprache so etwas vorsieht, dann werden Speicherzellen indirekt adressiert, also eine virtuelle Adresse durch eine Tabelle auf eine reale umgesetzt. Die virtuelle Adresse eines „Orts“ (in unserem Modell) bleibt somit stets gleich.

jekte einfach als implizit annehmen: Es sind die impliziten Objekte, denen die Eingaben und Ausgaben eines Programms als Attribute zugeordnet sind.

terminologische Ungenauigkeit

Wir unterscheiden also grundsätzlich Werte (Universalien) und Objekte (Individuen). Diese Unterscheidung machen wir jedoch nicht, wenn wir davon sprechen, dass Variablen Objekte *zum Wert* haben oder dass Ausdrücke, die Objekte liefern, aus*werten*. Dies ist strenggenommen kein Widerspruch, sofern Objekte stets durch Referenzen vertreten werden, denn Referenzen auf Objekte sind ja technisch gesehen Werte (*Referenzwerte*). Wenn dann auch noch Referenzwerte stets als Objekte interpretiert werden (die *Referenzsemantik* objektorientierter Programmiersprachen; s. Abschnitt 7.7), ist die Sprache von „einem Objekt als Wert“ oder von der „Auswertung zu einem Objekt“ entschuldbar.

6.2 Gleichheit und Identität

Um noch ein wenig bei der Metaphysik zu bleiben: Mit Leibniz verbunden ist der Gedanke, dass zwei Dinge, die sich in allen Aspekten gleichen, die also nichts an sich haben, anhand dessen man sie unterscheiden könnte, identisch (also nicht zwei, sondern eins) sein müssen.[60] Dieses als **Identitas indiscernibilium** bekannte Prinzip kann man in einem mathematischen Kontext, in dem Zeit keine besondere Rolle spielt, durchaus so stehen lassen — schließlich ist es die Grundlage vom *Gesetz der Substituierbarkeit*[61]. In der realen Welt (so wie auch in Programmen; s. dazu das entsprechende Beispiel aus Abschnitt 7.8) können sich zwei Dinge aber auch nur vorübergehend vollkommen gleichen, ohne dass sie deswegen zu einem Individuum verschmelzen würden, denn sonst könnten sie sich hernach nicht mehr unterscheiden (eines ist immer von sich selbst ununterscheidbar und ein Individuum per Definition nicht in zwei teilbar).[62] Von der vollkommenen Gleichheit zweier Individuen ausgenommen ist nämlich immer die Tatsache, dass zwei Individuen zu jeder Zeit jeweils verschiedene Orte einnehmen müssen, ihr Ort sie also immer unterscheidet. Also gilt zumindest für unsere Objekte, dass ihre Gleichheit (manchmal auch als **qualitative Identität** bezeichnet) nicht bedeutet, dass sich auch ihre Referenzen gleichen, es sich deswegen nur um eines handelt (das wäre dann die **quantitative Identität**).

[60] Dies liest sich wie ein Widerspruch, der darin besteht, dass in der Prämisse zwei Dinge vorausgesetzt werden, die Konklusion aber besagt, dass es tatsächlich nur eines ist. Gemeint ist, dass es keine zwei Dinge geben kann, die genau gleich („vollkommen identisch“) sind.

[61] vgl. dazu Fußnote 22!

[62] Daran hat Leibniz natürlich auch gedacht. Die Antwort liegt in seiner *Monadologie*, die hier aber ins Abseits führen würde (obwohl ein Vergleich mit den in der modernen funktionalen Programmierung Anwendung findenden Theorie der *Monaden* durchaus lohnend erscheint).

7 Programmieren mit Objekten

Wir hatten in Kapitel 4 schon Darstellungen von Objekten in JAVASCRIPT gesehen — tatsächlich handelte es sich hierbei um Objektliterale, also die buchstäbliche (literale) Repräsentation von Objekten in einem Programm. Ihr Datentyp ist, wie der von allen Objekten, `'object'`.

7.1 Objektliterale

Ähnlich wie die Literale von primitiven Werten stellt ein **Objektliteral** ein Objekt in einem Programmtext dar. Anders als Werte sind Objekte jedoch änderbar, was bedeutet, dass das Literal das Objekt nur zu einem bestimmten Zeitpunkt, nämlich dem Zeitpunkt, zu dem das Literal als Teil des Programm ausgeführt wird, repräsentiert. Dies passt eigentlich nicht ganz zum Begriff des Literals (das ja im Programmtext, also buchstäblich und somit zeitunabhängig, also statisch, für etwas steht), aber im Grunde verhält es sich hier ähnlich wie bei einem *String template*: Das Literal wird in JAVASCRIPT, wie in anderen dynamischen Sprachen auch, (erst) zum Zeitpunkt seiner Verwendung im Programmablauf zu einem Objekt „ausgewertet“ und erzeugt erst dann ein neues Objekt. Dabei kann das gleiche Literal auch mehrere Objekte erzeugen, nämlich wenn es wiederholt ausgewertet wird; mehr dazu in Kapitel 9.

Der Mechanismus der Auswertung von Literalen erlaubt es, in literalen Definitionen von Objekten Variablen zu verwenden. So lässt sich anstelle von

```
56 {                                                                    JS
57     name: 'Blöd',
58     alter: 42
59 }
```

auch

```
60 let name = 'Blöd'                                                    JS
61 let alter = 42
62 {
63     name: name,
64     alter: alter
65 }
```

schreiben, wobei für letzteres sogar die abkürzende Schreibweise (der „syntaktische Zucker“)

```
66 { name, alter }                                                      JS
```

vorgesehen ist. Anstelle von Variablen (an der Stelle von Werten) können auch beliebige Ausdrücke stehen, die (wie bei der *Stringinterpolation*) erst bei der Auswertung des Literals ausgewertet werden (zur Verwendung von Variablen an der Stelle der Namen von Eigenschaften gleich mehr).

Mithilfe von Variablen lassen sich auch Schachtelungen von Objekten in ihren literalen Definitionen auflösen (linearisieren) und (wieder) herstellen:

```JS
let adresse = {
    straße = 'Universitätsstraße 1',
    ort = 'Hagen'
}
{
    name: 'Blöd',
    adresse: adresse
}
```

ist gleichbedeutend mit

```JS
{
    name: 'Blöd',
    adresse: {
        straße:'Universitätsstraße 1',
        ort: 'Hagen'
    }
}
```

Jedoch lassen sich so keine Objekte definieren, die sich selbst enthalten:

```JS
var selbst = {f: selbst}
```

initialisiert `selbst` mit einem Objekt, dessen Feld `f` den Wert `undefined` hat (da `selbst` zum Zeitpunkt der Auswertung des Literals das resultierende Objekt noch nicht zugewiesen wurde). Die Definition zirkulärer Objektstrukturen verlangt mehrere Schritte (s. dazu Abschnitt 7.2).

Ausdrücke als Namen von Eigenschaften

Etwas anderes geht dafür aber schon: Variablen dürfen in Objektliteralen auch anstelle der Namen von Eigenschaften stehen:

```JS
let eigenschaft = 'name'
{
    [eigenschaft]: 'Blöd',
    alter: 42
}
```

ist auch gültiges JAVASCRIPT (wobei hier anstelle von `eigenschaft` auch jeder andere Ausdruck stehen kann, der sich in einen String zwingen lässt). Die eckigen Klammern zeigen hierbei an, dass der enthaltene Name nicht literal als Feldname verwendet werden soll, sondern ein Ausdruck ist, dessen Auswertung erst den Feldnamen ergibt. Die eckigen Klammern sehen wir gleich noch einmal wieder.

Die Auswertung von Objektliteralen zur Laufzeit und insbesondere der Umstand, dass die Namen der Felder selbst durch Ausdrücke vertreten werden dürfen, erklären, dass Namen in einem Objektliteral mehrfach vorkommen dürfen. Dabei bildet nur der letzte angeführte Wert das Name/Wert-Paar, das das Objekt beschreibt — vorherige Zuordnungen werden einfach überschrieben (durch das jeweils nachfolgende Paar ersetzt). Jedoch werden alle Ausdrücke, die in einem Objektliteral vorkommen, ausgewertet, was sich ggf. an *Nebeneffekten* der Auswertung bemerkbar macht.

7.2 Felder

Wie bereits erwähnt bezeichnet man Eigenschaften eines Objekts, die primitive Werte oder Objekte zum Wert haben, im Allgemeinen (jedoch nicht unbedingt in der JavaScript-Parlance) als **Felder** (engl. fields).

Feldnamen sind Strings

Die obigen Beispiele haben bereits Verwendung von Feldern gemacht, die, wie alle Eigenschaften eines Objekts, Name/Wert-Paare sind. Als Namen kommen zunächst beliebige *Bezeichner* (engl. names oder identifiers; das sind in JavaScript Folgen von Buchstaben, Unterstrichen und Ziffern, die mit einem Buchstaben anfangen) und selbst Schlüsselwörter (die ansonsten keine gültigen Bezeichner sind) infrage; letztere müssen dafür jedoch als Strings angegeben und in eckige Klammern gesetzt werden, also etwa `{ ['new']: true }`. Tatsächlich werden Bezeichner als Namen von Feldern in Strings konvertiert, weil die Namen von Eigenschaften in JavaScript immer entweder Strings oder Symbole sind. Darin u. a. unterscheiden sich die Objekte von JavaScript von Records (Abschnitt 2.1.6), aber auch von Objekten vieler anderer objektorientierter Programmiersprachen, wo die Feldnamen Variablennamen und keine Werte (Strings), sondern Bezeichner sind.

Feldzugriff

Um ein Feld `f` auszuwählen oder darauf zuzugreifen, gibt es in JavaScript zwei Schreibweisen: die **Punktschreibweise**

```
88 > {f: 1}.f                                                    JS
   1
```

und die **Klammerschreibweise**

```
89 > {f: 1}['f']                                                 JS
   1
```

wobei in der Klammerschreibweise der Feldname als String angegeben werden muss. Beide Zugriffsformen können auch verkettet und dabei gemischt werden: so liefern

```
90 { f1: { f2: 1 } }.f1.f2                                       JS
91 { f1: { f2: 1 } }['f1']['f2']
92 { f1: { f2: 1 } }.f1['f2']
93 { f1: { f2: 1 } }['f1'].f2
```

jeweilige Vorteile der verschiedenen Schreibweisen

allesamt den Wert `1`. Der Vorteil der Punktschreibweise `objekt.f` ist neben ihrer weiten Verbreitung (sie wird in vielen Programmiersprachen nicht nur für Objekte, sondern auch für Records und Structs verwendet) ihre Kompaktheit; der Vorteil der Klammerschreibweise `objekt['f']`, die in anderen Sprachen häufig für Array-Zugriffe verwendet wird, ist, dass sie auch die Verwendung von Feldnamen zulässt, die keine Bezeichner sind, und — wichtiger — dass das zugegriffene Feld nicht buchstäblich (literal) im Programm stehen muss, sondern selbst durch einen Ausdruck angegeben werden kann. So geht beispielsweise

```
94 > let feld = 'f'                                              JS
```

```
> { f: 1 }[feld]
1
```

Objekte als Dictionaries

Eine solchermaßen flexible Möglichkeit zum Zugriff auf ein Feld bieten die meisten anderen objektorientierten Programmiersprachen nicht. In JavaScript ist es aber nichts Besonderes, denn Objekte sind in JavaScript eine Datenstruktur nach der Art der *Dictionaries*, oder *Maps*, wie Sie sie vielleicht von anderen Programmiersprachen her kennen: In solchen Dictionaries werden unter *Schlüsseln*, häufig *Keys* genannt, Werte gespeichert. Bei den Objekten JavaScripts sind die Schlüssel von Schlüssel/Wert-Paaren lediglich auf Strings (oder Symbole) beschränkt — ansonsten sind die Objekte JavaScripts im Wesentlichen Dictionaries.

Felder als Variablen; Wertzuweisungen an Felder

Wie die Felder eines Records lassen sich die Felder eines Objekts als Variablen auffassen, die einem Objekt zugeordnet sind. So lassen sich obige Feldzugriffe als lesende Zugriffe auf die entsprechenden Variablen interpretieren. Genauso gut kann man den Feldern eines Objekts auch Werte zuweisen, und zwar sowohl mittels der Punktschreibweise

```
objekt.f = 2
```
JS

als auch mittels der Klammerschreibweise

```
objekt['f'] = 2
```
JS

Dabei sind beide Zuweisungen Ausdrücke, liefern also bei ihrer Auswertung ein Ergebnis zurück (hier `2`).

Mithilfe von Zuweisungen an Felder lassen sich nun auch zirkuläre Objektstrukturen herstellen:

```
> let selbst = {f: undefined}
> selbst.f = selbst
<ref *1> { f: [Circular *1] }
```
JS

Zugriffe auf nicht vorhandene Felder

Besitzt ein Objekt ein Feld nicht, dann führt der lesende Zugriff auf diese Feld nicht zu einem Laufzeitfehler, sondern liefert den Wert `undefined`:

```
> {}.f
undefined
```
JS

Beim schreibenden Zugriff hingegen wird ein (bis dato nicht vorhandenes) Feld dem Objekt hinzufügt. Damit ließe sich obiges Beispiel auch wie folgt hinschreiben:

```
> let selbst = {}
> selbst.f = selbst
<ref *1> { f: [Circular *1] }
```
JS

Felder werden nicht deklariert

Hier zeigt sich, dass Felder eines Objekts in JavaScript nicht deklariert werden müssen. Anders als Variablen (die in JavaScript ja auch nicht deklariert

werden müssen; s. Abschnitt 5.3) *können* sie aber auch nicht deklariert werden; man hat also in JavaScript keinen besonderen Schutz vor dem Verwenden nicht vorhandener Felder. Dies ist nicht gut und in vielen anderen Programmiersprachen anders.

alle Objekte haben eine variable Struktur

Sieht man einmal von dem fehlenden Schutz vor Programmierfehlern ab, so bleibt die Erkenntnis, dass nicht nur die Werte von Feldern eines Objekts änderbar sind, sondern auch die Struktur des Objekts! Es können sich also zwei strukturell gleiche Objekte jederzeit strukturell auseinanderentwickeln oder umgekehrt vormals strukturell verschiedene Objekte strukturell angleichen. Dies ist bei klassenbasierten objektorientierten und objekt-funktionalen Programmiersprachen typischerweise nicht der Fall; es resultiert erneut aus dem Umstand, dass in JavaScript Objekte den Dictionaries (genauer: Instanzen einer solchen Klasse) anderer Sprachen entsprechen. Insofern haben dann doch alle Objekte die gleiche Struktur, nämlich die eines Dictionaries.

fehlerhafte Feldzugriffe

Während der Zugriff auf nicht vorhandene Felder eines Objekts weder lesend noch schreibend einen Fehler darstellt, löst jedoch jeder Feldzugriff auf `undefined` oder `null` anstelle eines Objekts einen Laufzeitfehler aus:

```
103 > let o                                                                    JS
104 > o.f
    Uncaught TypeError: Cannot read properties of undefined (reading 'f')
```

Schutz vor `null` und `undefined`

Diesem Laufzeitfehler kann durch den sog. *optionalen Zugriff*, der Anstelle des Punktoperators `.` den Operator `?.` verwendet, abgeholfen werden:

```
105 > o?.f                                                                     JS
    undefined
```

Hierbei scheint `null` bzw. `undefined` durch einen Feldzugriff „hindurchpropagiert“ zu werden[63]; tatsächlich wird der Feldzugriff aber abgebrochen und der Wert von `o` zurückgegeben. Soll hingegen der lesende Zugriff auf ein nicht vorhandenes Feld eines Objekts `o` nicht `undefined`, sondern einen Ersatzwert `w` liefern, dann kann man

```
106 o?.f ?? w                                                                  JS
```

verwenden, was den Wert `w` ergibt, wenn `o.f` zu `undefined` oder `null` auswertet. Man findet in älterem Code anstelle von `o?.f` und `o.f ?? w` auch `o && o.f` und `o && o.f || w`, die die oben gescholtene Interpretation von `null` und `undefined` als falsy und die dazu passenden Definitionen der Operatoren `&&` und `||` ausnutzen. Wie man aber an der (vor diesem Hintergrund weitgehend redundanten und deswegen nicht unbedingt benötigten) Einführung von `?.` und `??` sieht, ist es sinnvoll (und besser), für verschiedene Aufgaben verschiedene Operatoren vorzusehen.

[63] vgl. dazu Fußnote 16

7.3 Viele Objekte: Arrays

Ein Feld, das ein Objekt zum Wert hat, stellt eine Beziehung von einem Objekt, dem Besitzer des Feldes, zu einem anderen, dem Wert des Feldes, her. Oftmals stehen jedoch mit einem Objekt mehrere Objekte in ein und derselben Beziehung. Dafür mehrere Felder zu verwenden wird der Sache schon deswegen nicht gerecht, weil jedes Feld jeweils eine, seine eigene, Beziehung herstellt, und funktioniert auch dann nicht, wenn die Zahl der so in einer Beziehung stehenden Objekte variabel und potentiell unbegrenzt ist.

In diesen Fällen kann ein Feld ein *Array* als Wert haben, dessen Elemente die Objekte sind, mit denen der Besitzer des Feldes eigentlich in Beziehung steht. Eigentlich, weil das Objekt tatsächlich mit dem Array in Beziehung steht, das hier aber nur die Rolle eines Containers spielt, der stellvertretend für die vielen Objekte steht, die seine Elemente sind.

Arrays haben in JAVASCRIPT eine einfache literale Repräsentation, mit deren Hilfe solche Beziehungen zu vielen leicht ausgedrückt werden können:

```
107 const blaubär = {                                                    JS
108     grad: 'Käpt´n',
109     enkel: [ { name: 'Gelb' }, { name: 'Grün' }, { name: 'Rosa' } ]
110 }
```

Das Objekt, dass durch die Konstante `blaubär` benannt wird, hat somit, qua Indirektion über ein Array, den eigentlichen Wert von `enkel`, drei Enkel; man vergleiche dazu die Darstellung von Relationen mittels Abbildungen auf Mengen in Abschnitt 3.2.

Auf die Elemente eines Arrays kann durch Angabe eines Indexes, der in eckigen Klammern angegeben wird, einzeln zugegriffen werden (JAVASCRIPT-Arrays sind 0-basiert!):

```
111 > blaubär.enkel[0]                                                   JS
    { name: 'Gelb' }
```

Für die Iteration über alle Elemente eines Arrays steht neben einer For-Schleife, die den Index hochzählt, auch eine Version zur Verfügung, die direkt über die Elemente iteriert:

```
112 for (let e of blaubär.enkel) {                                       JS
113     console.log(e.name)
114 }
```

Neben der *externen Iteration* über die Elemente eines Arrays stehen Methoden zur Verfügung, die eine *interne Iteration* über ein Array durchführen, so z. B.

```
115 > [1, 2, 3].map(x=>x+1)                                              JS
    [ 2, 3, 4 ]
```

Wie ein Array zu einer solchen Methode kommt, können Sie in Abschnitt 10.1 lesen; über die Bedeutung und Funktionsweise von `map` und anderen Funktionen, die auf Arrays und anderen sog. *Collections* definiert sind, erfahren Sie genaueres in Kapitel 19.

Arrays als spezielle Objekte

Übrigens: In JAVASCRIPT sind Arrays als Objekte implementiert, also als Menge von Name/Wert-Paaren, deren Namen allerdings keine Strings oder Symbole, sondern natürliche Zahlen sind. Das array-artige Verhalten wird nur darübergelegt. Man kann einem Array in JAVASCRIPT wie einem normalen Objekt weitere Eigenschaften hinzufügen, die dann aber nicht zu seinen Elementen zählen. Als Namen dieser weiteren Eigenschaften dürfen auch negative Zahlen verwendet werden. Ein Vorteil der Repräsentation von Arrays als Objekte mit Indizes als Feldnamen ist, dass dünn besetze Arrays nur vergleichsweise wenig Speicher verbrauchen; der Nachteil ist, dass sie im Vergleich zu „richtigen" Arrays langsam sind.

7.4 Methoden

Eigenschaften eines Objekts können nicht nur primitive Werte oder Objekte zum Wert haben, sondern auch Funktionen.[64] Man nennt die derart Objekten zugeordneten Funktionen **Methoden**. Die Zusammenfassung von Daten (Feldern) und Funktionen (Methoden) in einer Struktur, dem Objekt (bzw. der Klasse; s. Lektion 3), ist ein Charakteristikum der objektorientierten und damit auch der der objekt-funktionalen Programmierung.

Die Methoden eines Objekts bestimmen sein **Verhalten**. Es ist in der Regel abhängig vom *Zustand* des Objekts, repräsentiert durch die Belegung seiner Felder. Mit ihren Methoden (dem Verhalten) wird den Objekten gewissermaßen Leben eingehaucht: Während Daten sonst durch davon getrennte Funktionen verarbeitet werden, verarbeiten Objekte mit ihren Methoden ihre Daten selbst. Dies führt zu einer ganz anderen als der herkömmlichen Programmorganisation, die der objektorientierte Programmierung ihren Ruf als eigenständiges *Paradigma* eingebracht hat.

7.4.1 Methodendefinitionen

In JAVASCRIPT sind Methoden Eigenschaften von Objekten, die mit Feldern auf einer Stufe stehen (und tatsächlich auch Felder im Sinne von Abschnitt 7.2 sind), die jedoch Funktionen zum Wert haben. Diese Parallelität manifestiert sich in der Schreibweise

```
116 {                                                                    JS
117     nachname: 'Blöd',
118     vorname: 'Hein',
119     name: function() { return this.vorname + ' ' + this.nachname }
120 }
```

mit der das Objekt eine Eigenschaft erhält, die den Namen `'name'` trägt und die eine namen- und parameterlose Methode zum Wert hat. Die (anonyme) Funktion lässt sich auch benennen:

```
121 > { m: function f() {} }.f()                                         JS
```

[64] „Nicht nur Objekte sondern auch Funktionen" ist im Kontext der objekt-funktionalen Programmierung nicht ganz richtig, da dort Funktionen in der Regel ebenfalls Objekte sind. Mehr dazu in Kapitel 8.

```
Uncaught TypeError: {(intermediate value)}.f is not a function
```

Der Name ist dann aber nur innerhalb der Funktion gültig und dient dort beispielsweise der Rekursion:

```
> { m: function f() { f() } }.m()                                    JS
Uncaught RangeError: Maximum call stack size exceeded
```

Alternativ lässt sich in JavaScript zur Definition obiger Eigenschaft `name` auch die sog. Methodensyntax verwenden:

```
{   …                                                                 JS
    name() { return this.vorname + ' ' + this.nachname }
}
```

Dies kommt der Methodendefinition in vielen anderen objektorientierten Programmiersprachen syntaktisch näher, ändert aber nichts am Status der Methode als Eigenschaft (oder besonderes Feld). Auch bleibt die Funktion damit (wie oben) anonym:

```
> { m() { m() } }.m()                                                JS
Uncaught ReferenceError: m is not defined
    at Object.m
```

Für einen rekursiven Aufruf muss hier die Eigenschaft (Methode) `m` des Objekts verwendet werden, was die Qualifizierung mit `this` verlangt:

```
> { m() { this.m() } }.m()                                           JS
Uncaught RangeError: Maximum call stack size exceeded
```

Wie das Beispiel zeigt, kann zum Aufrufen von Methoden aus einer Methode desselben Objekts `this` nicht wegelassen werden, wie das beispielsweise in Java der Fall ist. Dasselbe gilt für den Zugriff auf Felder des Objekts (wie oben auf `vorname` und `nachname` aus der Methode `name`); auch hier kann `this` nicht wegelassen werden.

Stringausdrücke als Methodennamen

Da Methoden Eigenschaften wie Felder auch sind, gelten bzgl. ihrer Benennung dieselben Bedingungen. Insbesondere ist ein Methodenname ein String (oder ein Symbol) und bei der Definition einer Methode kann anstelle ihres Namens auch ein Ausdruck in eckigen Klammern stehen:

```
{   …                                                                 JS
    ['name']() { return this.vorname + ' ' + this.nachname }
}
```

beispielsweise ist zu obigem Beispiel äquivalent.

Parameterzahl und Überladen

Analog zu Funktionen (Abschnitt 5.4) kann eine Methode beliebig viele Parameter haben, wobei die Zahl der Argumente an der Aufrufstelle davon

abweichen kann. Methoden können jedoch wie Funktionen nicht *überladen* werden: Ein Objekt kann keine zwei Methoden mit demselben Namen als Eigenschaft haben.[65]

this in inneren Funktionen

Innerhalb einer Methode können in JAVASCRIPT zwar keine weiteren Methoden, aber immerhin Funktionen definiert werden. Verwenden diese *inneren Funktionen* `this`, dann verweist es auf das *globale Objekt* und nicht auf das Empfängerobjekt der umgebenden Methode. Das bedeutet, dass man aus diesen inneren Funktionen nicht direkt auf die Eigenschaften des Objekts, zu deren Methode sie gehören, zugreifen kann; sie sind eben Funktionen und keine Methoden. Dass ihnen `this` dennoch zur Verfügung steht und auf das globale Objekt verweist, ist konsistent mit der Bedeutung von `this` in den global deklarierten Funktionen aus Abschnitt 5.4, denn diese können `this` ebenfalls für den Zugriff auf das globale Objekt verwenden; das liegt daran, dass globale Funktionen als Methoden eines globalen Objekts (Containers) betrachtet werden können, das (der) alles enthält. Dasselbe gilt übrigens auch für *globale Variablen*, die nicht mit `let` oder `var` deklariert worden sind. JAVASCRIPT verlässt also mit seinen globalen Funktionen und Variablen nicht die objektorientierte Welt.

7.4.2 Methodenaufruf: dynamisches Binden

Wie bereits oben erwähnt sind in JAVASCRIPT Methoden genau wie Felder Objekten zugeordnet. Man kann entsprechend auf Methoden eines Objekts genau wie auf seine Felder zugreifen, und zwar sowohl mittels der Punkt- als auch mittels der Klammerschreibweise: `o.m` und `o['m']` liefern beide die mit `'m'` benannte Funktion des Objekts `o` (so vorhanden). Um die Methode auch aufzurufen, müssen dem Namen allerdings noch runde Klammern hintangestellt werden, die dann auch die Argumente des Methodenaufrufs aufnehmen (bezüglich der Angabe der Argumente gilt das gleiche wie für Funktionsaufrufe; s. Abschnitt 5.4). In Fortsetzung des obigen Beispiels kann man hier beispielsweise per

```
131 {                                                                    JS
132     …
133     name: function() { … }
134 }.name
```

auf die Methode `name` des durch das voranstehenden Objektliteral repräsentierten Objekts lesend zugreifen und mittels

```
135 { … }.name()                                                         JS
```

diese auch noch aufrufen, d. h., zur Ausführung bringen. Solche Aufrufe hatten wir oben schon verwendet, um die (mangelnde) Verwendbarkeit von Methoden und Funktionen in bestimmten Fällen zu verdeutlichen.

dynamische Bindung

Vergleicht man den Methodenaufruf mit dem Funktionsaufruf, dann fällt auf, dass der Name der Methode mit einem Objekt qualifiziert ist. Tatsächlich hängt beim Me-

[65] Das stimmt nicht ganz: Im Kontext von Überschreibungen gibt es auch das. Mehr dazu im Kontext der Vererbung in Abschnitt 10.3.

thodenaufruf, anders als bei einem Funktionsaufruf, die Wahl der aufgerufenen Methode nicht nur an ihrem Namen, sondern auch an dem Objekt, auf dem sie aufgerufen wird. Es kann daher mit einem Aufruf `o.m(...)` im Programmtext bei jeder seiner Ausführungen eine andere Methode `m` aufgerufen werden, nämlich wenn sich der Wert der Variable `o` zwischen den Aufrufen ändert (sie also zu verschiedenen Zeiten auf verschiedene Objekte verweist). Das ist bei Funktionsaufrufen nicht so: Hier wird mit `f(...)` jedes Mal dieselbe Funktion `f` aufgerufen, und zwar auch dann, wenn man `f` das Argument `o` mitgeben würde. Man nennt Methodenaufrufe daher auch **dynamisch gebunden** (wobei hier mit *Bindung* die Bindung des Methodennamens an der Aufrufstelle an die Definition der Methode gemeint ist).

Empfänger und die implizite Bindung von this

Die Wahl der aufgerufenen Methode hängt also an dem Objekt, auf dem sie aufgerufen wird. Man nennt dieses Objekt auch den **Empfänger**, wobei diese Sprechweise durch die Programmiersprache SMALLTALK geprägt wurde, in deren Parlance der Methodenaufruf **Nachrichtenversand** heißt (und der Versand natürlich einen Empfänger braucht). Innerhalb der aufgerufenen Methode wird der Empfänger in JAVASCRIPT (wie in vielen anderen Sprachen auch) durch `this` repräsentiert, so dass die Verwendung von `this` der Methode den Zugriff auf andere Eigenschaften des Empfängerobjekts gestattet. So verweist bei der Ausführung der Methode `name` aus dem obigen Beispiel mittels `o.name()` der Ausdruck `this.vorname` auf das Feld des Objekts `o`, auf dem die Methode aufgerufen wurde. `this` kann wie ein impliziter zusätzlicher Parameter jeder Methode angesehen werden, der beim Aufruf der Methode automatisch an den Empfänger gebunden wird; in manchen Sprachen, wie z. B. PYTHON, ist dieser Parameter noch nicht einmal implizit, sondern muss explizit deklariert werden (wobei er dann auch anders heißen darf).

grundsätzliche Bedeutung in der objektorientierten Programmierung

Während die dynamische Bindung für dynamische Sprachen wie JAVASCRIPT eine Voraussetzung ist, ohne die sie überhaupt nicht funktionieren würden (jeder Zugriff auf eine dynamische Struktur wie ein Objekt, das nicht durch ein Literal repräsentiert wird, muss dynamisch, also während des Zugriffs, gebunden werden), ist in statischen Sprachen, in denen die Strukturen von Objekten und anderen Daten bereits zur Übersetzungszeit feststehen, statische Bindung der Regelfall. Soweit diese Sprachen objektorientiert sind, werden in ihnen Methodenaufrufe dennoch dynamisch gebunden, was hier jedoch ausschließlich daran liegt, dass ein Objekt eine Methode auch erben und überschreiben kann. Erben und Überschreiben, die es beide auch in JAVASCRIPT gibt und denen wir uns in Kapitel 10 zuwenden werden, sind für objektorientierte Programmiersprachen wesentlich; das dynamische Binden ist nur eine technische Voraussetzung dafür, dass sie die beabsichtigte Wirkung entfalten können.

Anlass zu Kritik

Die dynamische Bindung ist nicht nur unauflösbar mit der objektorientierten Programmierung verbunden, sie gibt auch Anlass zu Kritik daran. So sind dynamisch gebundene Aufrufe teurer (langsamer und damit ressourcenintensiver) als statisch gebundene. Wenn man dazu bedenkt, dass Methoden häufig kurz sind und somit Methodenaufrufe zahlreich (die Programme werden durch kurze Methoden nämlich nicht selbst kürzer), dann kommt in der Praxis einiges an Aufwand für das dynamische Binden zusammen. Außerdem sieht man beim dynamischen Binden an der Aufrufstelle nicht, welche Methode genau aufgerufen wird; das macht das Lesen und Debuggen eines Programms schwieriger. Deswegen verlangen man-

che objektorientierten Programmiersprachen, dass Methoden, die dynamisch gebunden werden können sollen, in einem Programm explizit als solche ausgewiesen werden müssen (so z. B. C# und C++); alle anderen werden dann statisch gebunden. Das gilt naturgemäß nur für solche Sprachen, in denen statische Bindung überhaupt möglich ist; in JavaScript ist das nicht der Fall.

7.4.3 Beförderung von Funktionen zu Methoden

Wie in anderen objektorientierten Programmiersprachen auch sind in JavaScript Methoden Funktionen, die einem Objekt untergeordnet sind. Mittels `this` haben Methoden Zugriff auf die (anderen) Eigenschaften des Objekts. Eine Besonderheit JavaScripts ist aber, dass auch Funktionen, die nicht einem Objekt untergeordnet sind (die also keine Methoden sind), `this` verwenden können. Außer im Strict mode verweist `this` dann auf das globale Objekt (als dessen Methoden man die Funktionen dann ansehen kann).

Diese Besonderheit JavaScripts erlaubt, Funktionen zu Methoden zu befördern, indem man sie Objekten als Eigenschaften zuweist. So ergibt sich

```
136 > function wer() { console.log(this.ich) }                                JS
137 > const o1 = { ich: 'o1', hallo: wer }
138 > const o2 = { ich: 'o2', hallo: wer }
139 > o1.hallo()
    o1
140 > o2.hallo()
    o2
```

Hierbei wird ausgenutzt, dass Funktionen Werte sind (genaugenommen sind es Objekte; s. Kapitel 8), die durch ihren Namen (hier `wer`) bezeichnet werden.

explizite Bindung an Empfänger

Funktionen können in JavaScript aber auch ohne Zuweisung an eine Eigenschaft als Methoden ausgeführt werden, nämlich indem man sie an ein Empfängerobjekt bindet (und es ihnen damit gewissermaßen unterjubelt). Dies gelingt mit der Methode `bind`, über die alle Funktionen (als Objekte) verfügen:

```
141 > const o1 = { ich: 'o1' }                                               JS
142 > const o2 = { ich: 'o2' }
143 > function hallo() { console.log(this.ich) }
144 > hallo()
    undefined
145 > const hallo1 = hallo.bind(o1)
146 > const hallo2 = hallo.bind(o2)
147 > hallo1()
    o1
148 > hallo2()
    o2
```

Wie das Beispiel zeigt, wird es damit möglich, Methoden auf Objekten auszuführen, die diese Methoden gar nicht besitzen (weder selbst noch geerbt). Man mag darüber streiten, ob dies mit der *Datenkapselung*, die die Objektorientierung eigentlich bedeuten soll, bricht — in keinem Fall

steht es jedoch im Einklang mit der Bündelung von Daten und Funktionen in einer Einheit, dem Objekt, für die die Objektorientierung ursprünglich einmal stand.

7.4.4 Datenkapselung mit Queries, Commands und Zugriffsmethoden

Eine der Grundideen der Objektorientierung ist, dass die Daten und damit der innere Aufbau von Objekten vor der Außenwelt, also anderen Objekten, verborgen bleiben sollen: Ein Objekt soll ausschließlich über seine Methoden, aufgeteilt in *Queries* und *Commands*, in Erscheinung treten. Dabei geben Queries Auskunft über den Zustand des Objekts, während Commands in ändern. Man nennt dies auch **Datenkapselung**. Der Gedanke dahinter ist, dass sich die interne Repräsentation eines Objekts, wie sie durch seine Felder bestimmt wird, im Laufe der Entwicklung eines Programms ändern kann (die berühmt-berüchtigte Änderung einer Entwurfsentscheidung, vor denen niemand gefeit ist). Die Methoden eines Objekts nimmt man hingegen als relativ stabil an.

Gleichwohl ist es oft so, dass das Lesen und Setzen der Werte einzelner Felder eines Objekts durchaus problemadäquaten (und nicht nur durch eine konkrete Implementierung determinierten) Queries und Commands entspricht. Dies ist insbesondere dann so, wenn durch Felder Eigenschaften oder Beziehungen zu anderen Objekten umgesetzt werden, die sich direkt aus einer Anwendungsdomäne ergeben. Um der Idee der Datenkapselung, also dem Verbergen von Feldern hinter Methoden, dennoch Rechnung zu tragen, unterstützen viele objektorientierte Programmiersprachen, so auch JAVASCRIPT, sog. **Zugriffsmethoden** (engl. *Accessor methods*), konkret die Definition von **Gettern** (als speziellen Queries für den lesenden Zugriff auf Felder) und **Settern** (als speziellen Commands für den schreibenden Zugriff). Anstatt für ein Objekt mit dem Feld `name` etwa

```
let o = {
    name: '',
    holeName() { return this.name },
    setzeName(name) { this.name = name }
}
```

zu schreiben und dann den Wert dieser Eigenschaft mit dem Command `o.setzeName('a')` zu setzen oder mit der Query `o.holeName()` zu lesen, kann man in JAVASCRIPT stattdessen

```
let o = {
    _name: '',
    get name() {return this._name},
    set name(name) {this._name = name}
}
```

schreiben, was einem ermöglicht, per `o.name` lesend und per `o.name = 'abc'` schreibend auf das Feld `_name` des Objekts `o` zuzugreifen. Der Unterstrich in `_name` ist hierbei Konvention, die zum Ausdruck bringt, dass das Feld `_name` nicht von außen verwendet werden soll.[66] Dies geht aber trotzdem: `o._name` hat dasselbe Ergebnis wie `o.name` und `o._name = 'abc'` denselben Effekt wir

[66] Diese Konvention kann in TYPESCRIPT durch Zwang ersetzt werden; s. dazu Kapitel 26.

`o.name = 'abc'`. Nicht zuletzt deswegen müssen sich der Feldname und die Namen seines Getters und seines Setters unterscheiden.

Nutzen von Zugriffsmethoden

Stellt man für alle Felder eines Objekts beide Zugriffsmethoden zur Verfügung, scheint zunächst wenig gewonnen. In der Praxis wird man sie daher vor allem dann verwenden, wenn bei Feldzugriffen noch mehr passieren soll als nur das Lesen bzw. Schreiben, oder wenn ein Feld virtuell ist, also in Wirklichkeit gar nicht existiert, sondern durch andere Felder emuliert wird. Außerdem kann für ein Feld auch nur eine der beiden Zugriffsmethoden angeboten werden, wodurch der jeweils andere Zugriff als unerwünscht gekennzeichnet wird.

funktionale Sicht auf Felder

Die Verwendung von Gettern und Settern für den Zugriff auf Felder eines Objekts beschreibt auf ganz praktische Art eine funktionale Interpretation von Feldern: Der lesende Zugriff auf ein Feld entspricht der Anwendung einer Funktion auf das Empfängerobjekt, die den Wert des Feldes (als Ausgabe) liefert, wohingegen der schreibende Zugriff der Anwendung mit einem zusätzlichen Argument (als Eingabe) entspricht, die einen neuen Zustand des Objekts zum Ergebnis hat, was in der funktionalen Programmierung einem *Nebeneffekt* entspricht, in der objektorientierten Programmierung aber der Zweck („Haupteffekt“) des Setters ist; als Wert der Funktion wird hier, je nach Implementierung, entweder das Empfängerobjekt oder der zugewiesene Wert oder nichts zurückgegeben.

Die funktionale Sicht auf Felder wird im Kontext von Typen und deren Varianz in Lektion 4 noch eine Rolle spielen (in Kapitel 28).

7.5 Attribute

In JAVASCRIPT kann jede Eigenschaft neben ihrem Namen und Wert auch noch eine Reihe von sog. **Attributen** besitzen, die die Verwendbarkeit der Eigenschaft bestimmen.[67] So gibt es beispielsweise das Attribut `writable`, das die Änderbarkeit einer Eigenschaft festlegt, oder das Attribut `enumerable`, das festlegt, ob eine Eigenschaft eines Objekts bei einer Aufzählung seiner Eigenschaften dabei ist oder nicht gelistet werden soll. Attribute sind in JAVASCRIPT wie die Eigenschaften eines Objekts selbst, die sie beschreiben, dynamisch, d. h., sie können zur Laufzeit eines Programms hinzugefügt und auch wieder entfernt werden. Attribute kennt man auch von anderen Sprachen, so z. B. von C# oder auch von JAVA, wo sie *Annotationen* heißen; in diesen beiden Sprachen, die zu den statischen zählen, werden sie u. a. dazu verwendet, während der Übersetzungszeit Änderungen oder Ergänzungen an einem Programm vorzunehmen.

7.6 Objekte als Namensräume

In der Programmierung spielen Namen eine wesentliche Rolle: Variablen und Funktionen haben Namen und werden über diese angesprochen, Objekte bekommen Namen, wenn sie Vari-

[67] Genaugenommen sind Attribute Eigenschaften von Eigenschaften und somit *Metaeigenschaften*.

ablen zugewiesen werden, und ihre Eigenschaften sind, als Name/Wert-Paare, mit Namen versehen und werden über diese ausgewählt. Beides, das Ansprechen und Auswählen (die man auch als den gleichen Vorgang ansehen kann), verlangt, dass Namen eindeutig sind.

Nun hat ein Programm schnell so viele Namen, dass diese knapp werden. Daraus ergibt sich eine Namenskonkurrenz: Zwei Dinge sollen gleich benannt werden, können es aber nicht, wenn dem die Eindeutigkeit entgegensteht. Dies ist insbesondere so, weil Namen häufig kurz gewählt werden und manche Namen zudem ziemlich populär sind („x“, „i“ und „name“ z. B.).

Aus diesen Gründen ist es sinnvoll, ein Programm in sog. **Namensräume** (engl. namespaces) aufzuteilen, die kleiner sind als das ganze Programm und in denen demzufolge nicht so ein Gedränge von Namen herrscht. Dabei sind Namensräume von Modulen zu unterscheiden: Namensräume regeln nicht zuvörderst die Zugreifbarkeit von benannten Elementen von außen (die sog. *Isolation*), sondern dienen vornehmlich der Eindeutigkeit von Namen. Diese müssen nämlich nur noch innerhalb eines Teils des Programms, eines Namensraumes, eindeutig sein.

Qualifizierung von Namen

Nicht zuletzt, weil durch Namensräume allein nichts verborgen werden soll, stellt sich die Frage, wie man auf Elemente unterschiedlicher Namensräume zugreifen kann. Dazu bekommen Namensräume selbst Namen, die die Namen ihrer Bewohner **qualifizieren**. Heißt also beispielsweise ein Namensraum „a“ und einer seiner Bewohner „b“, dann wird „b“ mittels des qualifizierten Namens „a.b“ angesprochen. Namensräume können auch geschachtelt sein, wodurch längere Qualifikationspfade entstehen. Sie kennen das im Grunde von den Verzeichnissen und Pfaden eines Dateisystems.

Objekte als Namensräume

Nun bildet in JAVASCRIPT jedes Objekt selbst eine Art Namensraum: Seine Eigenschaften müssen alle unterschiedliche Namen haben.[68] Da Objekte zudem geschachtelt werden können, sind auch die entsprechenden Namensräume geschachtelt. Um auf die Eigenschaft eines Objekts zuzugreifen, muss der Name der Eigenschaft mit dem Namen des Objekts qualifiziert werden: Wenn ein Objekt, das eine Eigenschaft `b` besitzt, `a` heißt (d. h., eine Variable `a` das Objekt zum Wert hat), dann bezeichnet `a.b` diese Eigenschaft (die Pseudovariable `this` im Rumpf einer Methode kann hier als generischer Name für ein Objekt verstanden werden). Aber welches ist der Namensraum, in dem `a` ein Name ist? Das Ganze braucht einen Anfang.

das globale Objekt

In früheren Versionen von JAVASCRIPT befand sich sämtlicher Code im Kontext eines Objekts, des sog. **globalen Objekts**. Grob gesprochen bildete es den Namensraum für alle globale Variablen und Funktionen, also solchen, die nicht im Namensraum einer Funktion oder eines Objekts eingeführt wurden (tatsächlich entsprechen globale Variablen und

[68] Man beachte, dass sie keine echten Namensräume im Sinne eines statischen, oder lexikalischen, Scopings bieten, da die Schachtelung von Objekten zur Laufzeit veränderlich, also dynamisch, ist und in JAVASCRIPT keine Einschränkungen bzgl. der Schachtelung von Objekten bestehen, also insbesondere nicht garantiert werden kann, welche Eigenschaften ein Objekt hat, das ein anderes in einer Schachtelung ersetzt. Das ist in statischen Sprachen anders.

Funktionen Eigenschaften des globalen Objekts[69]). Dieses globale Objekt, das mittels seiner Eigenschaft `globalThis` auf sich selbst verweist, heißt im klassischen Einsatz von JAVASCRIPT auch `window` und steht dort für das Browserfenster, in dem der JAVASCRIPT-Code ausgeführt wird. Es bietet Zugriff beispielsweise auf das im Fenster angezeigte Dokument (über seine Eigenschaft `document`). In NODE.JS ist hingegen `global` der Alias für `globalThis`; hier bietet er Zugriff auf Eigenschaften, die für serverseitige Programme nützlich sind. Man beachte, dass alle drei globalen Variablen entsprechen, die nicht mit `var` oder `let` deklariert wurden und daher selbst Eigenschaften des globalen Objekts sind. Daneben sind noch andere Standardelemente Eigenschaften des globalen Objekts, so die Ihnen schon bekannten `Infinity`, `NaN` und `undefined` sowie die in Abschnitt 9.4 vorgestellten *Konstruktoren* (die ja in JAVASCRIPT Funktionen sind) `Object`, `String` etc.

Module als Namensräume

In aktuelleren Versionen von JAVASCRIPT wird die Verwendung des globalen Objekts als globaler Namensraum zugunsten der Einführung von *Modulen*, die Zugleich die *Isolation* der in ihnen enthaltenen Programmelemente bieten, abgelöst. Module sind jedoch kein spezifisch objektorientiertes oder funktionales Konstrukt und werden daher in diesem Text — mit Ausnahme der Abschnitte 17.1 und 36.1.2, wo es um die Fassung von Modulen in objektorientierten Termini geht — nicht weiter behandelt.

7.7 Wert- und Referenzsemantik sowie Aliasing bei Objekten

Referenzen und Referenzwerte

Auch wenn in der Objektorientierung eigentlich alles ein Objekt sein soll, unterscheiden die meisten objektorientierten Programmiersprachen doch zwei Arten von Werten, nämlich *primitive Werte* (wie Zahlen, Zeichen und Wahrheitswerte) und *Objekte*. Variablen, die für Werte stehen, enthalten diese Werte, oder haben sie zum Wert; Variablen, die für Objekte stehen, enthalten hingegen nur Zeiger auf Objekte, haben also Zeiger zum Wert. Diese Zeiger, die im Kontext der Objektorientierung üblicherweise **Referenzen** heißen[70], werden zur Abgrenzung von (primitiven) Werten gelegentlich auch **Referenzwerte** (engl. reference values) genannt. Da Referenzwert umgangssprachlich jedoch etwas anderes bedeutet, spricht man in der Regel besser von Referenzen.

In Programmen stehen nicht nur Variablen, sondern allgemein Ausdrücke (zu denen Variablen ja zählen; s. Abschnitt 5.3) für Werte. Entsprechend unterscheiden wir Ausdrücke, die zu einem primitiven Wert auswerten (wie beispielsweise die Addition von zwei Zahlen) und die genau

[69] In Abschnitt 5.3 hatten wir gesehen, dass der lesende Zugriff auf nicht deklarierte Variablen in JAVASCRIPT `undefined` ergibt und der schreibende Zugriff auf solche Variablen diese einfach anlegt. Genau das ist aber auch das Verhalten JAVASCRIPTs bei der Verwendung nicht vorhandener Felder eines Objekts: Lesender Zugriff liefert `undefined`, schreibender legt sie als neue Felder an (Abschnitt 7.2). Mit `let` und `var` deklarierte Variablen (sowie Konstanten) werden hingegen keine Felder des globalen Objekts; man kann dies durchaus als Abkehr von JAVASCRIPTs selbstbezüglichem, minimalen Sprachdesign auffassen, das sich für größere Aufgaben so nicht bewährt hat.

[70] Zeiger werden gelegentlich als das technische Konstrukt (eine Speicheradresse) gesehen und Referenzen als ihr logisches Pendant. Verweise sind je nach Autor näher an dem einen oder dem anderen.

diesen Wert liefern, und Ausdrücke, die zu einem Objekt auswerten und eine Referenz darauf liefern. So liefert auch ein *Objektliteral* kein Objekt, sondern eine Referenz auf das Objekt, das durch das Literal definiert (und durch seine Auswertung erzeugt) wird. (Wir erinnern uns, dass in JAVASCRIPT Objektliterale Ausdrücke sind, die zur Laufzeit ausgewertet werden; s. Abschnitt 7.1).

Wert- und Referenzsemantik

Allerdings treten Referenzen in den meisten objektorientierten Programmiersprachen, so auch in JAVASCRIPT, aus Sicht des Programmierers nie als Werte in Erscheinung. Insbesondere erfolgt die *Dereferenzierung*, also der Zugriff über die Referenz auf das Objekt (genauer: auf seine Eigenschaften), auf das sie verweist, immer implizit. Wenn man sich also den Inhalt einer Variable oder den Wert eines Ausdrucks anschaut (z. B. per Ausgabe auf der Konsole oder in einem Source-level-Debugger) und es sich dabei um eine Referenz handelt, dann sieht man das Objekt (oder vielmehr eine literale Repräsentation des Objekts) und nicht die Referenz. Man nennt die Repräsentation von Objekten in einem Programm durch Referenzen und die implizite Dereferenzierung der Referenzen beim Zugriff auf die Eigenschaften der Objekte, die durch die Referenzen repräsentiert werden, auch **Referenzsemantik**. Referenzsemantik steht im Gegensatz zur **Wertsemantik**, die der direkten Repräsentation primitiver Werte (sowie in Programmiersprachen, die solche anbieten, also beispielsweise C# mit seinen Structs, auch strukturierter Werte), zugrunde liegt. Per Referenzsemantik stehen aus Sicht des Programmierers Ausdrücke (einschließlich Variablen), die zu Referenzen auswerten, für Objekte, und zwar genauso, wie die Ausdrücke, die zu primitiven Werten auswerten, für diese Werte stehen. Die Referenzen sind für den Programmierer also transparent (er sieht durch sie hindurch), zumindest zunächst.

Referenz- und Wertsemantik unter Wertzuweisungen

Die implizite Indirektion, die mit der Verwendung von Referenzen einhergeht, hat nämlich durchaus Konsequenzen, die bisweilen überraschen können. Beim Zuweisen einer Variable an eine andere wird nämlich immer deren Inhalt, also ihr Wert, kopiert. Das bedeutet aber im Fall von Referenzen, dass nur die Referenz, nicht aber das Objekt, auf das die Referenz verweist, kopiert wird! Wenn man also dem Anschein verfallen ist, dass Variablen Objekte genauso wie primitive Werte enthalten, dann hat die Zuweisung von Objekten eine fundamental andere Bedeutung als die von primitiven Werten, obwohl sich beide syntaktisch nicht unterscheiden: Nach

```
159 let x1 = 1                                                    JS
160 let x2 = x1
```

enthalten `x1` und `x2` jeweils eine Kopie (der internen Repräsentation) von `1`, nach

```
161 let o1 = {}                                                   JS
162 let o2 = o1
```

`o1` und `o2` jedoch (scheinbar) dasselbe Objekt (tatsächlich jeweils eine Kopie einer Referenz auf das bei der ersten Zuweisung angelegte Objekt). Dies macht sich spätestens dann bemerkbar, wenn man eines der beiden Objekte ändert:

```
163 > let o1 = {}                                                 JS
164 > let o2 = o1
```

```
165 > o1.f = 1
166 > o2
    { f: 1 }
```

Aliasing

Die Änderung sieht man auch am durch die Zuweisung vermeintlich kopierten Objekt! Tatsächlich wurde bei der Zuweisung aber eben nur die Referenz kopiert, weswegen man hier auch von *Aliasing* spricht, weil nämlich nach der Zuweisung in Zeile 164 `o2` nur ein weiterer Name für das Objekt ist, das durch die Zuweisung aus Zeile 163 schon durch `o1` benannt wurde.

kein merklicher Unterschied zur Wertsemantik von primitiven Werten

Nun mag man argumentieren, dass dies überhaupt keine Anomalie ist, denn woran sollte man feststellen, dass `x2` im vorangegangenen Beispiel nicht auch nur ein weiterer Name für den Wert ist, den auch schon `x1` benennt? Mit dem Experiment der Änderung eines Werts gelingt das jedenfalls nicht, denn primitive Werte sind ja auch in JAVASCRIPT nicht änderbar! Und ob man `x1` oder `o1` einen neuen Wert zuweist, macht sowieso keinen Unterschied — keines von beiden hat eine Auswirkung auf den Wert der jeweils anderen Variable, also `x2` bzw. `o2`.[71]

Unterschied bei änderbaren Werten

Dieses Gleichverhalten ändert sich allerdings schlagartig, wenn man zusammengesetzte (strukturierte) Werte wie Records oder Arrays einführt (wie in Abschnitt 2.1.6), die keine Objekte sind, die also direkt in Variablen enthalten sein können (in Sprachen wie C++ und Eiffel gibt es neben Objekten mit Referenzsemantik zusätzlich auch Objekte mit Wertsemantik). Dann würde man erwarten, dass sich Änderungen an einem Wert, der in einer Variable enthalten ist, *nicht* auch auf seine Kopien auswirken, die in anderen Variablen enthalten sind. Wenn man also so etwas wie

```
167 > let a1 = ['a', 'b']                                    JS
168 > let a2 = a1
169 > a1[0] = 'c'
170 > a1
    ['c', 'b']
```

schreibt (wobei `['a', 'b']` ein Array-Literal ist; vgl. Abschnitt 7.3) und unterstellt, dass Arrays Wertsemantik haben, dann würde man erwarten, dass `a2` von der Änderung von `a1` unberührt bleibt[72], sofern `a2` eine Kopie des Werts von `a1` enthält. Nun sind in JAVASCRIPT *Arrays* aber Objekte, mit Referenzsemantik so dass `a2` nur ein Alias für `a1` ist, die Änderung des Objekts mit Namen `a1` also auch über `a2` sichtbar wird:

```
171 > a2                                                     JS
    ['c', 'b']
```

[71] Tatsächlich werden in dynamischen Sprachen wie JAVASCRIPT, RUBY oder PYTHON auch primitive Werte von Variablen in der Regel durch Zeiger vertreten, da bei der Vereinbarung der Variable, und damit der Allokierung des von ihr benötigten Speicherbereichs, nicht klar ist, wie groß ihre Werte sein werden. Eine Ausnahme hiervon bilden lediglich die sog. *Immediates*.

[72] Strenggenommen sind Werte ja nicht änderbar, sondern werden durch andere ersetzt. Vgl. dazu die Diskussion in Abschnitt 2.1.6.

Dies mag überraschen, ist aber nur die Konsequenz dessen, dass in JAVASCRIPT sämtliche strukturierte Daten, also auch Arrays, Objekte sind und somit Referenzsemantik haben.

Bedeutung der Referenzsemantik für Parameter

Auch Programmierer, die dies im Wesentlichen verinnerlicht haben, tappen mitunter in eine böse Falle, wenn sie Objekte als Argumente an Funktionen übergeben und diese von den Funktionen geändert werden, nicht um dieses Zweckes willen, sondern um den Rückgabewert der Funktion durch Manipulation daraus abzuleiten. Die Änderungen sind dann nämlich auch an der Aufrufstelle sichtbar, obwohl es sich beim Aufruf um einen *Call-by-value* handelt, das Funktionsargument also in den Funktionsparameter kopiert wird (s. Abschnitt 2.1.8). Da hierbei aber nur die Referenz kopiert wird, die im Rumpf der Funktion bei jeder Verwendung automatisch dereferenziert wird, ist es das Argument, das geändert wird, so dass man den Eindruck bekommen kann, es habe sich beim Funktionsaufruf um einen *Call-by-reference* gehandelt. Dies ist besonders fatal bei rekursiven Aufrufen, bei denen man ausnutzen möchte, dass jeder Aufruf seinen eigenen Satz lokaler Variablen hat, die von den lokalen Variablen der vorherigen und nachfolgenden Aufrufe unabhängig sind. Was nützt es da, wenn diese Variablen alle auf dasselbe Objekt verweisen? Man bräuchte den Parameter gar nicht!

Unveränderlichkeit als Weg zur vollkommenen Angleichung

Diese Verwirrung lässt sich freilich beseitigen, indem man nicht nur primitive Werte, sondern auch Objekte unveränderlich macht, wie es in der *reinen funktionalen Programmierung* der Fall ist[73]. Dann ist die Wahl zwischen Wert- und Referenzsemantik nur noch eine Frage der (effizienten) Implementierung einer Sprache.

die große Illusion der Objektorientierung

Unter Referenzsemantik verschwimmt also die Unterscheidung von Objekten und Werten zumindest insofern, als Ausdrücke (inkl. Variablen) für Werte und Objekte gleichermaßen unmittelbar (also ohne explizite Dereferenzierung im Fall von Objekten) stehen können. Dies erleichtert die objektorientierte Programmierung in vielen Fällen ungemein und verhilft zudem zur Illusion, nach der in der objektorientierten Programmierung alles (also auch jeder primitive Wert) ein Objekt ist. Diese Illusion ist jedoch gefährlich, da sie zu Programmierfehlern führt, die ohne genaue Kenntnis des Unterschieds praktisch nicht zu finden sind. Anschauliche Beispiele hierfür finden Sie in nebenstehender Literatur.

7.8 Gleichheit und Identität von Objekten

Wir hatten uns in Abschnitt 5.2.3 bereits mit der Gleichheit von primitiven Werten befasst und festgestellt, dass zwei Werte gleich sind, wenn sie die gleiche Repräsentation haben. (Zusätzlich definiert JAVASCRIPT Gleichheit mittels `==` so, dass zwei Ausdrücke gleich sind, wenn ihre Werte in gleiche des gleichen Datentyps gezwungen werden können.) Die Frage ist nun, wie Gleichheit für Objekte definiert ist. Wir nähern uns der Antwort experimentell an. Dem schicken wir voraus, dass es beim Objektvergleich in JAVASCRIPT keinen Unterschied zwischen `==`

[73] rein im Sinne von engl. pure, also ohne überschreibbare Variablen, was dann, wenn man die Felder von Objekten als Variablen auffasst (Abschnitt 7.2), bedeutet, dass auch Objekte unveränderlich sind

und `===` gibt. Da es in anderen Sprachen wiederum `===` nicht gibt und dort `==` der Bedeutung von `===` entspricht, verwenden wir hier `==`.

Zunächst vergleichen wir noch einmal ein leeres Objekt mit einem leeren Objekt:

```
> {} == {}                                                                    JS
false
```

Hätten wir das Ergebnis nicht schon in Kapitel 4 zur Kenntnis genommen, dann wären wir vielleicht jetzt überrascht: Da das Objektliteral auf der linken und der rechten Seite des Operators (also beide Operanden) syntaktisch absolut gleich sind, wäre nicht unbedingt zu erwarten, dass sie als ungleich getestet werden. Das liegt daran, dass die beiden Objektliterale bei ihrer Auswertung jeweils ein neues Objekt erzeugen und eine Referenz darauf zurückgeben. Da hier zwei Objekte erzeugt werden, werden auch zwei Referenzen zurückgeliefert, die als Werte nicht gleich sind. Insofern werden also Referenzen beim Test auf Gleichheit (mittels `==`) nicht anders behandelt als primitive Werte. Dass die Objekte, für die die Referenzen stehen, sich in Struktur und Inhalt nicht unterscheiden, spielt dabei keine Rolle. Trotzdem muss man konstatieren, dass die beiden Objekte *gleich* im herkömmlichen Sinn von Gleichheit sind. Sie sind nur nicht *identisch*.

Was aber ist der Unterschied zwischen Gleichheit und Identität? Während dies eine der zentralen Fragen der klassischen Philosophie ist mit Auswirkungen bis in die Grundlagen der Mathematik und mathematischen Logik (s. Kapitel 6), ist sie in der objektorientierten Programmierung leicht zu beantworten: Zwei Objekte sind gleich, wenn sie sich in ihren Eigenschaften nicht unterscheiden. Zwei Objekte können hingegen nicht identisch sein, denn sonst wären es nicht zwei, sondern nur eines — zwei Ausdrücke werten aber zum identischen Objekt aus, wenn sie zu den gleichen Referenzen auswerten.

Wenn aber mittels `==` in JAVASCRIPT nur die Identität geprüft werden kann, wie lässt sich dann die Gleichheit prüfen? Dazu kann man sich eine Funktion

```
function gleich(obj1, obj2) {                                                 JS
    if (Object.keys(obj1).length !== Object.keys(obj2).length) {
        return false
    }
    for (let name in namen1) {
        if (obj1[name] !== obj2[name]) {
            return false
        }
    }
    return true
}
```

definieren, die über alle Eigenschafen iteriert und deren Werte vergleicht. Wir prüfen

```
> gleich({}, {})                                                              JS
true
```

und sind zufrieden. Ebenso mit

```
185 > gleich({ f: 1 }, { f: 1} )                                    JS
    true
```

und mit

```
186 > gleich({ f: 1 }, { f: 2 })                                    JS
    false
```

jedoch weniger mit

```
187 > gleich({ f: {} }, { f: {} })                                  JS
    false
```

rekursive Definition von Gleichheit

Er ergibt sich hier nämlich das rekursive Problem der Gleichheit von Werten von Eigenschaften, wenn diese selbst Objekte sind: In der obigen Funktion wird hier Identität verlangt. Die Gleichheit ist somit eine **flache**; eine **tiefe** würde hingegen verlangen, dass auch die Werte der Eigenschaften lediglich gleich und nicht identisch sind. Wie tief diese Gleichheit allerdings gehen soll, dürfte von Fall zu Fall unterschiedlich sein (und bereitet spätestens bei Objekten, die sich selbst als Wert einer Eigenschaft angeben, auch ein algorithmisches Problem); insofern ist es JAVASCRIPT nicht anzulasten, dass es keine Definition der Gleichheit von Objekten vorgibt, sondern es bei der Identität belässt. Gleichheit von Objekten muss fallweise definiert werden.

Kontext Veränderlichkeit

Hier noch ein kurzer Exkurs, warum es richtig ist, bei Objekten zwischen Gleichheit und Identität strikt zu unterscheiden: Solange Objekte veränderbar sind, können sie sich vorübergehend gleichen:

```
188 > let o1 = { f: 1 }                                             JS
189 > let o2 = { f: 1 }
190 > gleich(o1, o2)
    true
191 > o1 == o2
    false
192 > o2.f = 2
193 > o1 == o2
    false
194 > gleich(o1, o2)
    false
```

Würde die vorübergehende Gleichheit dazu führen, dass die beiden Objekte als identisch betrachtet werden und somit zu einem verschmelzen (so dass in Zeile 0 `true` statt `false` stünde), könnten sie sich in der Folge auch nicht mehr auseinanderentwickeln: Jede Änderung an dem einen Objekt über den Namen `o1` würde auch über `o2` sichtbar und umgekehrt (so dass in Zeile 0 `true` stünde). Wollte man das nicht, dann müsste durch jede Änderung eines Objekts ein neues entstehen (ganz so, wie das in der *reinen funktionalen Programmierung* der Fall wäre), das vom alten zu trennen ist (so dass in Zeile 0 selbst dann `false` stünde, wenn in Zeile 0 noch `true` stand). Man beachte jedoch, wie sehr die Begründung des Unterschieds von Gleichheit und Identität an der *Veränderlichkeit* von Objekten hängt: Wäre diese nicht gegeben (wie etwa in reinen funktionalen Sprachen), dann wäre auch die Unterscheidung von Gleichheit und Identität nicht

sinnvoll (oder höchstens ein Implementierungsdetail ohne Auswirkung auf die Semantik von Programmen).

Übrigens: Dadurch, dass in JAVASCRIPT Objekten Eigenschaften hinzugefügt werden können, ist auch ein leeres Objekt veränderlich. Es ist also auch nicht sinnvoll, {} zu *internen*, also jedes Vorkommen von {} durch dasselbe Objekt (dieselbe Referenz) zu ersetzen, so wie das für unveränderliche Objekte gemacht werden kann (z. B. für Stringliterale in JAVASCRIPT).

7.9 Lebensdauer von Objekten

Wenn eine Variable, die einen primitiven Wert enthält, einen neuen Wert bekommt oder abgeräumt wird (z. B. weil die Ausführung der Funktion, im Rahmen derer sie angelegt wurde, beendet ist), dann ist der alte Wert weg. Er ist natürlich nicht wirklich weg, denn die Variable enthält ja nur eine Repräsentation des Wertes (in Form eine Bitstrings in der von ihr adressierten Speicherzelle), von der bei jeder Zuweisung eine Kopie angefertigt wird. Wenn eine Variable eine Referenz auf ein Objekt enthält, dann gilt für die Referenz dasselbe: Er ist nur eine Kopie und kann deswegen weg. Was aber passiert mit dem Objekt, auf das die Referenz verweist?

Jetzt wird es schwierig. Aufgrund des *Aliasing* kann es nämlich sein, dass die Referenz nicht die einzige ist, die auf das Objekt verweist (sie ist es sogar wahrscheinlich nicht, wenn die Variable ein Funktionsparameter ist, der nach Beendigung der Funktion wieder vom Stack genommen wird). In diesem Fall darf das Objekt nicht abgeräumt werden, weil die anderen Referenzen (Aliase) sonst ins Leere zeigen würden. Wenn die Referenz aber die letzte ist und verschwindet, dann ist das Objekt nicht mehr zugreifbar und belegt nur noch unnütz Speicher. Es *kann* also nicht nur weg, sondern *soll* auch, um Platz zu machen.

Speicherbereinigung

Nun kann es aber sein, dass Objekte über ihre Felder gegenseitig auf sich verweisen oder dass eine Methode eines Objekts in einem eigenen Thread ausgeführt wird, ohne dass sonst noch irgendeine Variable darauf verweist, oder dass es irgendwo eine Registratur aller erzeugten Objekte gibt, oder, oder, oder. Wie man leicht einsieht, ist es schwierig, all diese möglichen Sachverhalte im Auge zu behalten. Deswegen wird es in den meisten objektorientierten Programmiersprachen, so auch in JAVASCRIPT, nicht dem Programmierer überlassen, zu entscheiden, welche Objekte aus dem Speicher entfernt werden können. Stattdessen übernimmt dies ein automatischer Prozess, **Speicherbereinigung** genannt. Sie müssen sich also, wenn Sie JAVASCRIPT verwenden, zumindest in dieser Hinsicht um nichts kümmern.

8 Funktionen als Objekte

Wir hatten in Kapitel 3 gesehen, dass im Lambda-Kalkül Funktionen Werte sind. In JAVASCRIPT sind Funktionen hingegen Objekte. Im Vergleich zu anderen Objekten verfügen sie über ein paar zusätzliche interne Eigenschaften, darunter maßgeblich

1. die Ausführbarkeit: Eine Funktion kann der JavaScript-Engine zur Ausführung übergeben werden. (Objekte werden hingegen nur gespeichert.)
2. die *Umgebung*: Eine Funktion wird immer in dem Kontext ausgeführt, in dem sie definiert wurde, selbst wenn sie zur Ausführung in einen anderen Kontext verbracht wird (z. B. als Wert eines Parameters einer anderen Funktion). Ausnahmen hiervon bilden `this` und `super`, die sich auf den Aufrufkontext beziehen.

Dabei ist die Umgebung im Wesentlichen der *Funktionsabschluss*, oder *Abschluss* (engl. closure) der Funktion, über alle freien Variablen, die in ihr vorkommen (das sind alle verwendeten Variablen, die nicht lokal in der Funktion deklariert sind). Dies war uns in Abschnitt 3.4.4 schon einmal im Rahmen des Lambda-Kalküls begegnet, bedarf hier aber noch einmal einer genaueren Beleuchtung.

8.1 Funktionsliterale

Während Funktionen in imperativen Programmiersprachen (einschließlich vieler objektorientierter Programmiersprachen) Unterprogramme mit Rückgabewert sind und in funktionalen Programmiersprachen Werte, sind sie in JavaScript Objekte. Als solche haben sie eine literale Repräsentation, die allerdings nicht wie die eines Objekts, sondern wie die einer Funktionsdeklaration in einer anderen Programmiersprache aussieht (vgl. Abschnitt 5.4). So ist z. B. der Ausdruck

```
195 function quadrat(x) { return x * x }                                    JS
```

ein **Funktionsliteral**, das für ein Funktionsobjekt steht, das eine Funktion mit Namen `quadrat` repräsentiert, die das Quadrat ihres Arguments `x` zurückgibt. Anders als bei einer Funktionsdeklaration, die es auch in JavaScript gibt und die sich von einer Funktionsdefinition syntaktisch nicht unterscheidet (die Definition wird zu einer Deklaration *befördert*; s. Abschnitt 5.4), gilt der mit dem Funktionsliteral für die Funktion vergebene Name nur im Rumpf der Funktion, wo er einem rekursiven Aufruf zu dienen vermag; außerhalb des Rumpfes ist er nicht verwendbar.

anonyme Funktionen Tatsächlich wird der Name einer Funktion in einem Funktionsliteral (außer für rekursive Aufrufe oder andere „Erwähnungen“ der Funktion in ihrem Rumpf) gar nicht gebraucht: So stehen

```
196 function (x) { return x * x }                                           JS
```

sowie

```
197 x => x * x                                                             JS
```

für die gleiche, nur anonyme Funktion. Beide Ausdrücke entsprechen dem Lambda-Ausdruck $\lambda x.\, x * x$, der ja auch eine literale Repräsentation einer Funktion, die zugleich ein Wert ist, darstellt (s. Abschnitt 3.3). Dass es sich bei den Ausdrücken aus den Zeilen 195–197 tatsächlich um

literale Repräsentationen von Objekten handelt, kann man zum einen daran erkennen, dass sie einer Variable zugewiesen werden können, etwa durch

```
198 let quadrat = x => x * x                                                    JS
```

zum anderen aber auch daran, dass Funktionen wie Objekte Eigenschaften haben können. So hat jede Funktion eine Eigenschaft `name`, der im Fall einer anonymen Funktion bei ihrer erstmaligen Zuweisung an eine Variable intern der Name der Variable als Wert zugewiesen wird:

```
199 > (function f() {}).name                                                   JS
    'f'
200 > (function () {}).name
    ''
201 > let g = function () {}
202 > g.name
    'g'
```

auch Funktionsdeklarationen erzeugen Funktionsobjekte

Man beachte, dass die Tatsache, dass ein Funktionsliteral (als Ausdruck) zu einer Deklarationsanweisung befördert werden kann (Abschnitt 5.4), JAVASCRIPT lediglich ein imperatives (prozedurales) Flair verleiht, ohne am Status der Funktion als Objekt etwas zu verändern. Es wird also auch mit der Deklarationsanweisung

```
203 function inkrement(x) { return x + 1 };                                     JS
```

ein Objekt angelegt, wovon man sich mittels

```
204 > typeof inkrement                                                         JS
    'function'
```

leicht überzeugen kann (der Datentyp `'function'` ist hier als Spezialfall des Datentyps `'object'` zu verstehen).

8.2 Funktionsanwendung

indirekter Funktionsaufruf

Wie wir gesehen haben, sind Funktionen in JAVASCRIPT Bürgerinnen erster Klasse, können also auch Variablen zugewiesen werden. Tatsächlich ist dies die Voraussetzung dafür, dass anonyme Funktionen überhaupt einen Nutzen entfalten können, da sie sonst nach ihrer Erzeugung gleich wieder verlorengingen (s. Abschnitt 7.9). Um in Variablen gespeicherte (von Variablen benannte) Funktionen auszuführen, wendet man sie auf Argumente an. Dazu stellt man den Variablen die geklammerte Liste der Argumente hintan:

```
205 > let quadrat = x => x * x                                                 JS
206 > quadrat(2)
    4
```

Dies entspricht im Wesentlichen dem Ausdruck

$$\text{let } \mathit{quadrat} \;=\; \lambda x.\, x * x \text{ in } \mathit{quadrat}\ 2$$

des Lambda-Kalküls. Man beachte, dass es sich bei einem Aufruf wie `quadrat(2)` oben nicht um einen klassischen, statisch gebundenen Funktionsaufruf, wie man ihn von der imperativen Programmierung her kennt, handeln kann, da der Wert einer Variable wie `quadrat` im allgemeinen Fall erst zur Laufzeit bekannt ist.

Anstelle eines Funktionsnamens kann in JAVASCRIPT nicht nur eine Variable, sondern genau wie im Lambda-Kalkül jeder beliebige Ausdruck, der zu einem Funktionsobjekt auswertet, zu einem Funktionsaufruf gemacht werden, indem man ihn syntaktisch um eine Argumentliste ergänzt. Insbesondere kann auch eine Funktionsanwendung selbst für einen solchen Ausdruck stehen. Genau das ist ja die Grundlage des *Currying* (s. a. Abschnitt 3.3.9).

8.3 Currying

Da Funktionen Objekte sind, können sie auch von anderen Funktionen zurückgegeben werden. Dies erlaubt die **curryfizierte Darstellung** mehrstelliger Funktionen, also beispielsweise der zweistelligen Addition als eine einstellige Funktion höherer Ordnung, wie durch

```
207 function addiere(a) {                                                JS
208     return function (b) {
209         return a + b
210     }
211 }
```

gegeben. Diese Funktion kann dann beispielsweise per

```
212 > addiere(1)(2)                                                       JS
    3
```

angewendet oder aufgerufen werden, wobei natürlich der zweite Aufruf (mit Argument `2`) nicht unmittelbar nach dem ersten erfolgen muss.[74] Der Aufruf

```
213 > addiere(1, 2)                                                       JS
    [Function (anonymous)]
```

funktioniert jedoch nicht, wie vielleicht erhofft — das zweite Argument wird hier als überzählig gewertet, wie

```
214 > addiere(1, 2)(2)                                                   JS
    3
```

[74] Tatsächlich hat ein isolierter Aufruf wie `addiere(1)` durchaus seine Existenzberechtigung: Er liefert eine Funktion zurück, die, auf eine Zahl angewendet, eine um 1 größere zurückliefert. Man spricht dann auch von *partieller Evaluation* der (eigentlich zweistelligen) Funktion `addiere`. Partielle Evaluation wird eigentlich zur Optimierung der Ausführung von Programmen verwendet; davon kann aber hier nicht die Rede sein, da ja noch keine Berechnungen durchgeführt werden.

zeigt. Um gekehrt wird eine mehrstellige Funktion in JAVASCRIPT auch nicht implizit curryfiziert: Eine als `addiere(a, b)` deklarierte Funktion kann nicht per `addiere(1)(2)` aufgerufen werden.

8.4 Funktionsabschlüsse

Die Parameter einer Funktion wie etwa `a` von `addiere` aus Zeile 207 sind lokale Variablen derselben. Sie können im Rumpf der Funktion gelesen und geschrieben werden. Außerdem können im Rumpf auch die Variablen gelesen und geschrieben werden, die dort (in dem Geltungsbereich), wo die Funktion definiert ist (wo das Funktionsliteral steht), gelesen und geschrieben werden können. Im obigen Beispiel von `addiere` wäre dies `a` in Zeile 209, das in der anonymen Funktion, die mittels `return` (in Zeile 208) zurückgegeben wird, verwendet wird und keine lokale Variable dieser Funktion ist. Dies hat in dem Moment, indem eine Funktion (als Objekt) aus ihrem Kontext entfernt und an anderer Stelle ausgeführt werden soll, weitreichende Implikationen.

Funktionsabschluss als lebensverlängernde Maßnahme für Variablen

Parameter und (andere) lokale Variablen von Funktionen werden bei ihrem Aufruf auf dem Stack angelegt und nach Beendigung der Funktion wieder abgeräumt. Da Funktionen in JAVASCRIPT aber Objekte sind (und somit überhaupt erst, wie im obigen Beispiel, von einer Funktion zurückgegeben werden können), können sie die Lebensdauer der Variablen, auf die sie in ihrem Rumpf zugreifen und die nicht ihre eigenen lokalen Variablen sind (sondern solche aus dem umgebenden Geltungsbereich), überleben. Genau das ist aber beim Aufruf von `addiere(1)` der Fall: Die anonyme Funktion, die hier zurückgegeben wird, verwendet `a`. Die Funktion muss also die Variable `a` über deren Lebensdauer hinaus behalten. Dies geschieht in JAVASCRIPT mithilfe eines Funktionsabschlusses, ganz so, wie wir ihn in Abschnitt 3.4.4 schon kennengelernt haben.

8.5 Datenkapselung durch Funktionsabschlüsse

Funktionsabschlüsse und das damit verbundene „Retten“ von lokalen (und damit aus weiten Teilen des Programms unsichtbaren) Variablen über ihre eigentliche Lebensdauer hinaus kann dazu verwendet werden, Daten zu kapseln. Dies zeigt das folgende Beispiel von einem Zähler:

```
215 let zähler = function () {                                    JS
216     let x = 0
217     return function () { x += 1; return x }}
```

Hier ist `x` lokal zur (anonymen) Funktion, die der Variable `zähler` zugewiesen wird, aber global zu der (anonymen) Funktion, die von einen Aufruf von `zähler()` als Ergebnis zurückgeben wird. Somit erhält man mittels

```
218 > let z = zähler()                                            JS
```

einen Funktionsabschluss als Wert von `z`, der sich wie folgt verhält:

```
> z()
1
> z()
2
```
JS

Die Variable `x`, die das Gedächtnis dieser Funktion darstellt (wobei Funktionen ja eigentlich selbst gar kein Gedächtnis haben, sondern zu diesem Zweck auf globale Variablen angewiesen sind), ist vollkommen versteckt: Sie wurde in den (unsichtbaren) Funktionsabschluss verschoben, der bei jedem Aufruf von `zähler()` neu (mit neuer Variable `x`) erzeugt wird. Es erfolgt hier also eine **Datenkapselung durch Funktionsabschluss**.

Die Datenkapselung durch Funktionsabschlüsse ist ein originärer Beitrag der funktionalen Programmierung. Sie steht in Konkurrenz zu dem, was man in der Objektorientierung gemeinhin über hinter Methoden verborgenen Feldern realisiert: So könnte man für einen Zähler `z` auch

```
let z = { x: 0, inkrement() { this.x = this.x + 1; return this.x } }
```
JS

definieren und hernach `z.inkrement()` schreiben, um den Wert der Variable `x` zu erhöhen und zurückzubekommen. Allerdings ist hier `x` auch noch über `z` (mittels des Ausdrucks `z.x`) zugreifbar. Während andere Sprachen *Zugriffsmodifikatoren* wie `public` und `private` bieten, gibt es das in JAVASCRIPT nicht (wohl aber in TYPESCRIPT; s. Kapitel 26). Hier ist Datenkapselung durch Funktionsabschlüsse eine sichere Alternative.

Funktionsabschlüsse lassen sich auch für die unvollständige *Datenkapselung* von Objekten mit *Zugriffsmethoden* aus Abschnitt 7.4.4 einsetzen:

```
function neuePerson() {
    var _name;
    return {
        get name () { return _name },
        set name (name) { _name = name }
    }
}
```
JS

ist eine Fabrikmethode, die ein Objekt mit Getter und Setter, aber ohne dazugehöriges Feld liefert: Dieses wird von der Variable `_name` aus den Funktionsabschlüssen des Getters und des Setters vertreten.

9 Objektkonstruktion

Wir hatten in Abschnitt 7.1 bereits bemerkt, dass Objektliterale im Gegensatz zu den Literalen primitiver Werte (mit Ausnahme interpolierender Stringliterale) nicht buchstäblich für Werte (Objekte) stehen, sondern Ausdrücke sind, die zur Laufzeit eines Programms ausgewertet werden und als Ergebnis eine Referenz auf ein neues Objekt liefern. Besonders deutlich wird das an dem Beispiel

```
> let array = []
```
JS

```
> for (let i of [1, 2]) array[i] = {}
> array[1] == array[2]
false
```

in dem das Objektliteral `{}` nur einmal vorkommt, es aber zweimal ausgewertet wird und somit zwei Objekte liefert (die natürlich nicht identisch sind). Dies lässt sich ausnutzen, um Objekte durch Funktionen erzeugen zu lassen.

9.1 Fabrikfunktionen

Objektliterale können auch in Funktionen auftreten, die somit eine einfache Form der Objektkonstruktion anbieten. So liefert beispielsweise die Funktion

```JS
function Person(name, alter) {
    return { name, alter }
}
```

bei jedem Aufruf ein neues Objekt, wie man sich anhand von

```JS
> Person('Hein', 42)
{ name: 'Hein', alter: 42 }
```

sowie

```JS
> Person('Hein', 42) == Person('Hein', 42)
false
```

überzeugen kann.

Man beachte, dass die durch den Aufruf von `Person` erzeugten Objekte keinerlei Bezug zu der Funktion oder ihrem Namen haben. Dass also durch das zurückgelieferte Objekt eine Person repräsentiert werden soll, verbleibt allein im Auge des Betrachters. Tatsächlich kann ein und dieselbe Funktion, per Fallunterscheidung in ihrem Rumpf, Objekte beliebiger Struktur liefern. Man nennt solche Funktionen deswegen auch **Fabrikfunktionen** (engl. factory functions). Dabei ist es Fabrikfunktionen in JAVASCRIPT sogar freigestellt, gar kein Objekt, sondern einen beliebigen Wert zu liefern; Fabrikfunktionen sind — nicht nur im Prinzip — ganz normale Funktionen, namensgebend ist allein ihr Verwendungszweck.

9.2 Konstruktoren

Jedoch gibt es in JAVASCRIPT noch eine andere Möglichkeit, Funktionen zur Erzeugung neuer Objekte heranzuziehen. Dazu werden Funktionen über das Schlüsselwort `new` aufgerufen, also beispielsweise per

```JS
new Person('Hein', 42)
```

Dabei passiert hinter den Kulissen folgendes:

1. Es wird ein neues leeres Objekt erzeugt.
2. Das Objekt wird mit der Funktion `Person` verbunden (wie das geht folgt in Kapitel 10).
3. Die Funktion `Person` wird ausgeführt und dabei `this` an das neue Objekt gebunden, so dass die Funktion den Aufbau des Objekts unter Verwendung von `this` festlegen kann.
4. Wird die Funktion nicht mit einem expliziten Return-Statement beendet, gibt sie implizit `this`, also das neue Objekt zurück.

Nun liefert obige Definition von `Person` (als eine Fabrikfunktion) von sich aus, also auch ohne `new`, ein neues Objekt. Da dieses neue Objekt zudem per `return` zurückgegeben wird, liefert der Aufruf mit `new` kein anderes Ergebnis als der ohne `new`; das durch `new` erzeugte leere Objekt wird verworfen. Um hingegen das von `new` erzeugte Objekt zurückzuerhalten, muss `Person` wie folgt definiert werden:

```
238 function Person(name, alter) {                                    JS
239     this.name = name
240     this.alter = alter
241 }
```

Damit erhalten wir dann

```
242 > new Person('Hein', 42)                                          JS
    Person { name: 'Hein', alter: 42 }
```

Hierbei unterscheidet sich die Ausgabe der REPL gegenüber der von Zeile 0 darin, dass der literalen Repräsentation des neuen Objekts der Name der Funktion, „Person“, vorangestellt ist. Das neu erzeugte Objekt hat also eine Verbindung zu der Funktion, aus deren Aufruf über `new` es hervorgegangen ist.

In JavaScript nennt man Funktionen, die mit `new` aufgerufen werden, um ein neues Objekt zurückzuliefern, **Konstruktoren** und die dazugehörigen Aufrufe **Konstruktoraufrufe**. Konstruktoren grenzen sich von Fabrikmethoden dadurch ab, dass sie selbst keine neuen Objekte erzeugen, sondern sie lediglich initialisieren; Konstruktoraufrufe grenzen sich von den Aufrufen von Fabrikmethoden dadurch ab, dass sie die neuen Objekte ihren jeweiligen Konstruktoren zuordnen. Dabei ersetzt die Zuordnung zu Konstruktoren die Zuordnung von Objekten zu Klassen, wie Sie sie vielleicht von klassenbasierten objektorientierten Programmiersprachen schon kennen. JavaScript kennt aber — wie alle prototypenbasierten objektorientierten Programmiersprachen — keine Klassen.

Gefahr des falschen Aufrufs von Konstruktoren in JavaScript

Obige Funktion `Person` sieht zwar aus wie ein Konstruktor in manchen klassenbasierten objektorientierten Programmiersprachen (wie beispielsweise Java; bis hin nicht vorhandenen Return-Statement), es ist aber eine ganz normale Funktion, die auch ohne die Verwendung von `new` aufgerufen werden kann. Dabei wird dann ausgenutzt, dass `this` auch in solchen Funktionen ein gültiger Name ist, der sich in dem Fall aber auf das *globale Objekt* bezieht. Da eine solche Verwendung einer Funktion, die eigentlich als Konstruktor gedacht ist, leicht ein Programmierfehler sein kann,

wird die Verwendung von `this` für das globale Objekt bei Verwendung des Strict mode (per `'use strict'`) unterbunden — `this` hat dann stattdessen den Wert `undefined`, dessen Dereferenzierung (beim Zugriff auf eine Eigenschaft über `this`) zu einem Laufzeitfehler führt.

9.3 Konstruktoren als Gemeinsamkeiten definierende Elemente

In den obigen Beispielen zur Objektkonstruktion wurden nur Felder verwendet — Methoden als Eigenschaften der erzeugten Objekte habe ich bewusst weggelassen. Dabei kann man, wie in Abschnitt 7.4 gesehen, Methoden genau wie Felder behandeln, sie also insbesondere dynamisch Objekten hinzufügen. Es wäre also sowohl im Fall der Fabrikmethoden als auch im Fall der Konstruktoren möglich gewesen, neben den Feldwerten auch Methoden als Argumente zu übergeben, die dann zur Konstruktion der Objekte herangezogen werden.

ungleiche Objekte mit gleichen Methoden

Dass man das häufig nicht macht (und die obigen Beispiele es auch nicht zeigen) hat einen einfachen Grund: Der Zweck von Konstruktoren ist in der Regel, viele Objekte herzustellen, die sich nur anhand der Werte ihrer Felder (ihrer Daten) unterscheiden, aber nicht anhand ihrer Struktur (alle haben den gleichen Satz an Eigenschaften) und nicht anhand ihrer Methoden (ihres Verhaltens). Deswegen hat der Konstruktor `Person` oben zwei Parameter, die es erlauben, für jede damit erzeugte neue Person unterschiedliche Werte für die Felder `name` und `alter` festzulegen. Die Anzahl und Namen der Felder sind aber für alle durch den Konstruktoraufruf erzeugten Objekte gleich. Wenn dazu auch noch das Verhalten aller Personen gleich sein soll (modulo ihrer jeweiligen Zustände, von denen ihr Verhalten ja auch abhängt), dann würde der Konstruktor jedem von ihm erzeugten Objekt die gleichen Methodendefinitionen hinzufügen. Was sich bei einer entsprechenden Fabrikfunktion beispielsweise als

```
243 function Person(name, alter) {
244     return { name, alter, hallo: function() { console.log(this.name) } }
245 }
```

manifestiert, wird im Fall der Konstruktorfunktion durch

```
246 function Person(name, alter) {                                    JS
247     this.name = name
248     this.alter = alter
249     this.hallo = function() { console.log(this.name) }
250 }
```

ausgedrückt. Sofern sich die Definition der Methode `hallo` auch zukünftig (nach der Erstellung von Objekten) nicht einzeln ausdifferenziert, ist das eine ziemliche Ressourcenverschwendung: die gleiche Methode (Funktion) wird x-fach repliziert, obwohl sie generisch ist, also auf die Eigenschaften des Objekts, zu dem sie gehört, ausschließlich über `this` zugreift.

Konstruktoren als Vorboten von Klassen

Die Gleichförmigkeit vieler Objekte bezüglich ihrer Struktur und die Gleichheit ihres Verhaltens ist die Grundannahme der *klassenbasierten objektorientierten Programmierung*, wie sie Gegenstand der nächsten Lektion sein wird. Die Anzweife-

lung dieser Annahme ist hingegen die Grundlage der *prototypenbasierten objektorientierten Programmierung*, wie sie u. a. durch JAVASCRIPT repräsentiert wird: Es ist gerade gewollt, dass jedes Objekt seine eigene Struktur und sein eigenes Verhalten aufweisen kann. Gleichwohl müssen auch die Verfechter des Prototypenansatzes anerkennen, dass der Fall gleichförmiger und sich gleich verhaltender Objekte in der Praxis der Programmierung häufig vorkommt, weswegen auch die prototypenbasierte Programmierung damit ökonomisch umgehen können muss. Sie setzt dazu auf *Vererbung zwischen Objekten* und die namensgebenden *Prototypen*; diese sind Gegenstand von Kapitel 10.

9.4 Standardkonstruktoren

In JAVASCRIPT gibt eine ganze Reihe von Standardkonstruktoren, so z. B. `Object`, `Array` und `Function`. So ist beispielsweise der Aufruf `new Object()` gleichbedeutend mit der Auswertung von `{}` (erzeugt also ein neues, leeres Objekt), `new Array('a', 'b', 'c')` hingegen mit der des Array-Literals `['a', 'b', 'c']`. Mit `new Array(1000)` hingegen lässt sich ein Array mit eintausend (leeren) Plätzen anlegen. Dies ist von Vorteil, wenn man ein Array nach und nach mit Werten belegt, da das Array dann (im Rahmen der reservierten Plätze) nicht dynamisch wachsen muss.

Standardkonstruktoren als Sammlungen von Nützlichkeiten

Da Konstruktoren Funktionen sind und Funktionen Objekte, können auch Konstruktoren Eigenschaften, also Felder und Methoden haben. Dies scheint zunächst etwas widersinnig, erlaubt aber die Verwendung von Konstruktoren als Träger von Fabrikmethoden, die ja als Alternative von Konstruktoren angesehen werden und die somit zusammen mit den Konstruktoren definiert werden können. Ein gutes Beispiel hierfür ist der Konstruktor (die Funktion) `Object`: so gibt es zur Objekterzeugung mit `new Object()` die Alternative `Object.create(o)`, die gleich auch noch den Prototyp des neuen Objekts auf `o` festlegt (was das bedeutet, lernen Sie in Kapitel 10). Neben solchen Fabrikmethoden können auch noch andere kleine Nützlichkeiten angeboten werden: Eine, nämlich `Object.keys()`, hatten wir schon verwendet (in der Definition von `gleich`, Zeilen 173 ff.). Auch hier offenbart sich wieder eine gewisse konzeptionelle Anlehnung an klassenbasierte objektorientierte Programmiersprachen wie JAVA, wobei dort allerdings die Klasse an die Stelle des Konstruktors tritt und sowohl Konstruktoren als auch Fabrikmethoden den Klassen zugeordnet sind.

9.5 Boxing und Autoboxing

Zu den Standardkonstruktoren zählen auch `Number`, `Boolean` und `String`, deren Zweck es ist, Pendants zu primitiven Werten in Objektform zu erzeugen, also etwa einen String als Objekt. Gerade bei Strings kann das sinnvoll sein, da lange Strings viel Speicher belegen und Stringoperationen, die jedes Mal neue Strings erzeugen anstatt einen vorhandenen String zu ändern, insbesondere in Schleifen oder bei rekursiven Aufrufen den Speicher verbraten. Auch wird mit diesen Konstruktoren das sog. *Autoboxing* vollzogen, das einen primitiven Wert implizit in ein entsprechendes Objekt konvertiert und dann darauf Methoden, die mit dem jeweiligen Konstruktor definiert wurden, auszuführen erlaubt:

```
> ' abc '.trim()                                                    JS
'abc'
```

oder

```
> 1['toFixed'](2)                                                   JS
'1.00'
```

(wobei hier die Punktschreibweise für den Aufruf von `toFixed` zu einem Syntaxfehler führen, da der Punkt als Dezimalkomma interpretiert würde). Ein Autounboxing findet hierbei übrigens nicht statt, auch wenn das Beispiel mit `trim` dies nahelegen würde: Zwar haben wir

```
> typeof ' ab '.trim()                                              JS
'string'
```

aber eben auch

```
> typeof new String(' abc ').trim()                                 JS
'string'
```

Die Methode `trim` liefert einen Wert zurück — die Box, die durch `new String(' abc ')` erzeugt wird, überlebt den Methodenaufruf auf ihr nicht. Wie die Objekte, die durch das Boxing aus Werten entstehen, zu ihren Methoden kommen, ist Gegenstand von Abschnitt 10.1. (`new` bindet übrigens stärker an den nachfolgenden Funktionsaufruf als der Zugriff auf die Methode `trim`, weshalb im obigen Beispiel keine Klammern um `new String(' abc ')` erforderlich sind.)

9.6 Dualer Nutzen von Funktionen in JavaScript

An den Standardkonstruktoren kann man auch sehen, dass die Dualität der Verwendung von Funktionen (normal oder als Konstruktor) interessante Angebote erlaubt: So liefert

```
> new String('abc')                                                 JS
[String: 'abc']
```

also ein Objekt, wohingegen

```
> String(new String('abc'))                                         JS
'abc'
```

einen primitiven Wert liefert. Es bietet also dieselbe Funktion `String` einmal die Gewinnung eines Objekts aus einem primitiven Wert und einmal die Umkehrung in Form der Gewinnung eines primitiven Werts aus einem Objekt. Ein solches Verhalten lässt sich durch die nachfolgende beispielhafte Implementierung der Funktion `String` erklären:

```
function String(wert) {                                             JS
    if (this instanceof String) {
        this.wert = wert.toString()
    } else {
        return wert.toString()
```

```
262     }
263 }
```

Hierbei ist `instanceof` ein eingebauter Operator, der testet, ob ein Objekt durch einen Konstruktoraufruf (`new` in Verbindung mit einem Funktionsnamen, hier `String`) erzeugt wurde. Von welchem Konstruktor ein Objekt konstruiert wurde war ja schon oben Teil der Ausgabe eines Konstruktoraufrufs (wie das geht ist Gegenstand des Kapitels 10).

9.7 Kopieren von Objekten

Der Vollständigkeit halber soll nicht unerwähnt bleiben, dass auch das Kopieren vorhandener Objekte eine Möglichkeit ist, Objekte zu konstruieren. Dies kann in JAVASCRIPT mit der Methode `assign` bewerkstelligt werden:

```
264 > const original = new Person('Hein', 35)                    JS
265 > const kopie = Object.assign(new Person, original)
266 > kopie
    Person { name: 'Hein', alter: 35 }
267 > kopie == original
    false
```

Sie kopiert alle Eigenschaften (inkl. Methoden) des zweiten Arguments, die mit `enumerable` attributiert sind, in das erste Argument und gibt das so modifizierte erste Argument zurück. Die Kopie ist eine *flache*, d. h., etwaige Objekte als Eigenschaften werden nicht selbst kopiert (sondern nur die Referenzen auf sie).

Um hingegen ein „funktionales Update“ eines Objekts zu erreichen, also anstatt den Zustand eines Objekts zu ändern, ein neues mit den Änderungen zu erzeugen, kann man in JAVASCRIPT die sog. Spread-Syntax verwenden:

```
268 > const original = new Person('Hein', 35)                    JS
269 > const kopie = { ... original, alter: 42) }
270 > kopie == original
    false
271 > kopie
    Person { name: 'Hein', alter: 43 }
```

Dabei wird durch `...` das nachfolgende Objekt gewissermaßen ausgerollt, hier also seine Name/Wert-Paare, soweit sie `enumerable` sind, in das Objektliteral eingespleißt. Dies Art der updatenden Kopie ist u. a. bei der Simulation von Wertobjekten (Abschnitt 14.3.1) interessant.

Das Kopieren von Objekten stellt keinen Zusammenhang zwischen dem Original und der Kopie her.

10 Prototypen und Vererbung

Das Muster aus Abschnitt 9.3, nach dem jedem Objekt, das von einer Fabrikmethode oder einem Konstruktor erzeugt wird, die gleichen Methoden beigefügt werden, ist nicht besonders ressourcenschonend. Dies gilt auch für den Fall, dass mehrere Objekte alle denselben Wert für eines ihrer Felder vorgesehen haben. Auch solche Fälle kann es nämlich durchaus geben; sie werden in klassenbasierten objektorientierten Programmiersprachen durch Felder, die Klassen und nicht Objekten zugeordnet sind (auch *Klassenvariablen* genannt) abgedeckt. In prototypenbasierten Sprachen wie JAVASCRIPT gibt es die naturgemäß nicht — stattdessen hat jedes Objekt seine eigene Version.

Während in der klassenbasierten objektorientierten Programmierung Klassen verwendet werden, um die Gleichartigkeit verschiedener Objekte bezüglich ihrer Eigenschaften darzustellen, kommen in der prototypenbasierten objektorientierten Programmierung hierfür — wer hätte es gedacht — **Prototypen** zum Einsatz. Dabei sind Prototypen Objekte wie alle anderen auch; ihre Besonderheit besteht allein darin, dass sie anderen Objekten als Prototypen dienen, also Eigenschaften aufweisen, die die Objekte, von denen sie die Prototypen sind, von ihnen erben. Man spricht hier von **prototypenbasierter Vererbung**. Sie steht im Gegensatz zur *klassenbasierten Vererbung* der meisten anderen objektorientierten Programmiersprachen (Gegenstand von Kapitel 16).

Damit in der prototypenbasierten Programmierung ein Objekt von einem anderen erbt, muss es diesen als Prototypen angeben. Dies geschieht bei prototypenbasierten Programmiersprachen wie SELF automatisch, wenn man ein Objekt als Klon eines anderen erzeugt. In JAVASCRIPT spielt das Klonen jedoch keine zentrale Rolle — stattdessen setzt es, wie wir in Kapitel 9 gesehen haben, auf die Erzeugung mittels Konstruktoren. Und tatsächlich erhalten mit Konstruktoren erzeugte Objekte in JAVASCRIPT automatisch ein Prototypobjekt zugewiesen, von dem sie erben.

10.1 Vererbung über Konstruktoren

Prototyp-Eigenschaft von Funktionen

In JAVASCRIPT haben alle Funktionen und damit auch alle Konstruktoren eine Eigenschaft namens `prototype`. Diese Eigenschaft verweist auf ein Objekt, das als Prototyp aller Objekte dient, die über einen Konstruktoraufruf mit `new` erzeugt werden. Dieses Prototypobjekt hat selbst eine Eigenschaft namens `constructor`, die auf den Konstruktor zurückverweist und die von allen Objekten, für die es Prototyp ist (also alle die, die mit dem dazugehörigen Konstruktor erzeugt wurden), geerbt wird. Im Fall des Konstruktors

```
272 function Person(name) { this.name = name }                    JS
```

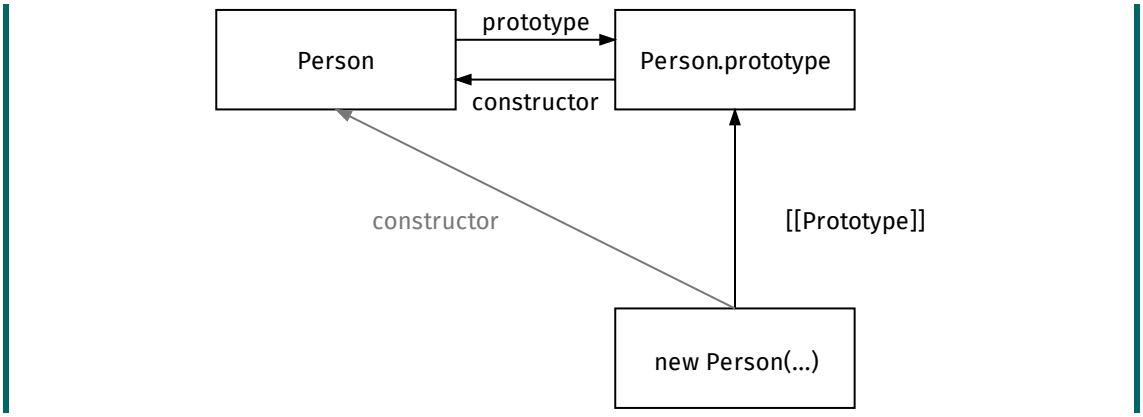
haben wir beispielsweise

```
273 > Person.prototype.constructor                                 JS
    [Function: Person]
```

sowie

```
> new Person('Hein').constructor                                    JS
Person
```

Die Situation ist in der nachfolgenden Abbildung dargestellt:



Die Vererbungsbeziehung ist hier durch den mit „[[Prototype]]“ beschrifteten Pfeil dargestellt (das durch `new Person(...)` erhaltene neue Objekt erbt von dem Objekt, das mit `Person.prototype` benannt ist), die dadurch geerbte Eigenschaft `constructor` ist gegraut.

erben vom Prototyp Nicht nur das Feld `constructor`, sondern alle Felder und Methoden, die für das Prototypobjekt einer Funktion (hier also `Person.prototype`) angelegt werden, werden an alle Objekte, die mit dieser Funktion als Konstruktor, also hier per `new Person(...)`, erzeugt wurden, vererbt (wobei wir zunächst einmal dahingestellt lassen, was Vererbung heißt). Durch eine entsprechende Modifikation des mit `Person.prototype` benannten Prototypobjekts lassen sich so beispielsweise alle Objekte, die mit `new` und `Person` erzeugt wurden, um die Methode `hallo` ergänzen:

```
> Person.prototype.hallo = function() { console.log(this.name) }    JS
> new Person('Hein').hallo()
'Hein'
```

Für Felder gilt entsprechendes:

```
> Person.prototype.frei = true                                      JS
> new Person('Hein').frei
true
```

Jedes per `new Person(...)` erzeugte Objekt kommt so in den Genuss der Eigenschaften, die für den Prototyp von `Person` definiert wurden. Dabei zeigt folgendes kleine Experiment, dass die Eigenschaften trotzdem nur einmal definiert sind:

```
> let p = new Person('Gelb')                                        JS
> p.frei
true
> Person.prototype.frei = false
> new Person('Rot').frei
false
> p.frei
false
```

Aber wie funktioniert das Erben von einem Prototyp?

Objekte haben Prototypen

In JAVASCRIPT hat jedes Objekt eine verborgene Eigenschaft, die auf den Prototyp des Objekts verweist. Diese Eigenschaft wird bei der Erzeugung des Objekts automatisch gesetzt, im Fall der Erzeugung mittels `new` und einer somit als Konstruktor genutzten Funktion auf das Objekt, das Wert der (nicht verborgenen) Eigenschaft `prototype` der Funktion ist.[75] Die verborgende Eigenschaft eines Objekts, die auf seinen Prototyp verweist, kann per

```
284 > Object.getPrototypeOf(o)                                                JS
    { frei: false }
```

abgefragt werden.[76] Tatsächlich gilt im (obigen) Fall von `o` als einem Objekt, das über `new Person()` erzeugt wurde,

```
285 > Object.getPrototypeOf(o) == Person.prototype                            JS
    true
```

Objekte, die per Objektliteral oder per `new Object()` erzeugt wurden, haben einen Standardprototyp, der durch den Ausdruck `Object.prototype` gegeben ist (wobei hier `Object` die Standardkonstruktorfunktion für Objekte ist und `prototype` eine Eigenschaft davon; s. Abschnitt 9.4):

```
286 > Object.getPrototypeOf({}).constructor                                   JS
    [Function: Object]
```

prototypenbasierte Vererbung

Der Prototyp eines Objekts wird nun von JAVASCRIPT genutzt, um Zugriffe auf Eigenschaften, die das Objekt nicht selbst definiert, beim Prototyp nachzuschlagen, also im gegebenen Beispiel die Eigenschaft `frei` des durch `Person.prototype` bezeichneten Objekts zu verwenden, sofern `frei` eben nicht für `o` selbst definiert ist, so dass `o.frei` eigentlich `undefined` liefern würde. Dieses ersatzweise Nachschlagen einer Eigenschaft beim Prototyp eines Objekts erzeugt den Anschein, das Objekt habe die Eigenschaft selbst; man spricht hier deswegen auch von **Vererbung**, genauer von **prototypenbasierter Vererbung**.

Prototypenbasierte Vererbung in Verbindung mit Konstruktoren ist die Erklärung, wie beispielsweise Arrays an Methoden wie `map` (Abschnitt 7.3) kommen oder Funktion an Methoden wie `bind` (Abschnitt 7.4.3) oder Strings an Methoden wie `trim` (Abschnitt 9.5): Alle Arrays werden über den Konstruktor `Array` erzeugt, dessen Eigenschaft `Array.prototype` das Objekt bezeichnet,

---

[75] Man beachte, dass das Objekt, das Wert der Eigenschaft `prototype` einer Funktion ist, nicht der Prototyp der Funktion ist, die, als Objekt, wie alle anderen Objekte auch einen Prototyp haben kann. Der Prototyp eines Objekts `o` ist nicht über `o.prototype` zugänglich; tatsächlich haben Objekte, die keine Funktionen sind, keine solche Eigenschaft (obwohl sie jeweils einen Prototyp haben). Dies ist leider sehr verwirrend.

[76] Dieses Konstrukt ist allerdings wenig objektorientiert, weil hier das Objekt, von dem eine Eigenschaft abgefragt wird, nicht Empfänger, sondern Argument eines Methodenaufrufs ist. Objektorientiert ist vielmehr der Ausdruck `o.__proto__`, der jedoch nicht mehr verwendet werden soll („deprecated“).

von dem alle Arrays erben und das die Methode `map` definiert; entsprechendes gilt für Funktionen und Strings.

10.2 Vererbung ohne Konstruktoren

Konstruktoren als Schema für gleichartige Objekte

Konstruktoren Prototypen zuzuordnen, von denen die mit den Konstruktoren erzeugten Objekte automatisch erben, erfüllt den Zweck, viele gleichartige Objekte zu erzeugen, also Objekte, die die gleichen Felder und Methoden haben und die sich nur in den Werten der Felder unterscheiden. Felder, deren Werte sich alle Objekte teilen sollen, sowie Methoden werden dann nur einmal, in dem den Objekten zugeordneten Prototyp definiert, was die Ressourcen schont. Genau dies ist aber der Zweck von *Klassen* in der *klassenbasierten objektorientierten Programmierung*, von der sich die prototypenbasierte gerade abgrenzen will.

explizites Setzen des Prototyps

Was die prototypenbasierte objektorientierte Programmierung nämlich eigentlich bietet, ist, ein Objekt erzeugen zu können, das *wie* ein anderes, aber nicht *das* andere ist und das sich über die Differenz zu einem anderen Objekt, seinem Prototyp, definieren lässt. Dies lässt sich wie im Beispiel

```
287 > let o1 = { eigen: 1 }                                          JS
288 > let o2 = { vererbt: 2 }
289 > Object.setPrototypeOf(o1, o2)
290 > o1
    { eigen: 1 }
291 > o1.vererbt
    2
```

erreichen: Durch den Aufruf der Methode `setPrototypeOf` wird hier das zweite Argument, `o2`, ein Objekt mit einer Eigenschaft namens `vererbt`, dem ersten Argument, `o1`, ebenfalls ein Objekt, als Prototyp zugeordnet, wodurch das erste Objekt die Eigenschaft `vererbt` vom zweiten Objekt erbt.

Klonen

Nun mag es etwas unnatürlich erscheinen, zunächst ein Objekt nur mit den Differenzen zu erzeugen und ihm erst dann seinen Prototyp zuzuweisen, zu dem die Differenzen formuliert sind. Natürlicher schiene dann, zunächst ein Objekt unter Angabe seines Prototyps zu erzeugen und dann die Differenzen hinzuzufügen. In JavaScript lässt sich das durch Verwendung der Methode `Object.create` erreichen:

```
292 > let o1 = { vererbt: 1 }                                        JS
293 > let o2 = Object.create(o1)
294 > o2.eigen = 2
295 > o2
    { eigen: 2 }
296 > o2.vererbt
    1
```

Man leitete also zuerst (mittels `create`) ein neues Objekt `o2` von einem vorhandenen Objekt `o1` ab, so dass `o2` (über den Mechanismus von `create`) `o1` als Prototyp hat und somit alle Eigen-

schaften von ihm erbt. Erst danach fügt man die Differenzen hinzu. Man nennt diesen Vorgang auch **Klonen**, weil das durch `Object.create` nach dem Vorbild (dem Prototyp) eines Objekts erzeugte neue Objekt dem Vorbild zunächst vollständig gleicht. Bis auf seine Identität, die natürlich verschieden ist.

Tatsächlich ist das Klonen das Standardverfahren der Objekterzeugung in der prototypenbasierten Programmierung, wie es durch die Sprache SELF etabliert wurde. Die Konstruktoren JAVASCRIPTs nehmen sich im Vergleich dazu wie eine Verwirrung aus.

10.3 Überschreiben

Bislang hatten wir uns nur mit dem lesenden Zugriff auf geerbte Eigenschaften befasst — geschrieben hatten wir Eigenschaften stets in den Prototyp. Mit einer Ausnahme: In Zeile 294 hatten wir `o1` ein neues Feld hinzugefügt, was auch tatsächlich in `o1` und nicht etwa in `o2` gelandet ist. Allgemein greift beim Schreiben von Eigenschaften der Mechanismus der Vererbung nicht:

```
297 > o1.vererbt = 3                                                    JS
298 > o2
    { vererbt: 2 }
299 > o1
    { eigen: 1, vererbt: 3 }
```

Das Schreiben (per Zuweisung) in eine geerbte Eigenschaft in Zeile 297 führt also dazu, dass die Eigenschaft im Objekt neu angelegt wird. Da `o1` jetzt selbst ein Feld mit Namen `vererbt` hat, wird das (zuvor geerbte) gleichnamige Feld von `o2` **verdeckt**; man sagt dann auch, es sei unsichtbar geworden, was aber nicht heißt, dass sein Name seine Gültigkeit verloren hat: Das Feld ist nämlich nicht weg, sondern kann immer noch über

```
300 > Object.getPrototypeOf(o1).vererbt                                 JS
    2
```

oder, aus dem Kontext (also beispielsweise dem Rumpf einer Methode) von `o1`, über

```
301 super.vererbt                                                       JS
```

erreicht werden. Auch wird es nach Löschen des verdeckenden Feldes mittels

```
302 > delete o1.vererbt                                                 JS
303 > o1.vererbt
    2
```

wieder geerbt.

Die Verdeckung von geerbten Eigenschaften durch Definition der Eigenschaften im erbenden Objekt wird auch als **Überschreiben** (engl. *overriding*) bezeichnet, wenn die Eigenschaft dynamisch gebunden wird (s. Abschnitt 10.5). Dies ist bei Methoden der Regelfall; in JAVASCRIPT

werden aber auch Feldzugriffe dynamisch gebunden, so dass auch Felder überschrieben werden. Wird hingegen eine nicht dynamisch gebundene Eigenschaft durch eine neue Definition im erbenden Objekt verdeckt, spricht man hingegen von **Hiding**.[77] Mit Überschreiben ist ausdrücklich nicht das Überschreiben einer Eigenschaft mit einem neuen Wert (engl. *overwriting*) im Rahmen einer Zuweisung gemeint — dies hat mit Vererbung nichts zu tun.

Die Möglichkeit des Überschreibens geerbter Eigenschaften ist für die objektorientierte Programmierung genauso wesentlich wie die Vererbung selbst. Wir werden in Abschnitt 10.6 zunächst kurz und dann in Kapitel 16 ausführlicher darauf eingehen.

10.4 Die Prototypenkette

Da Prototypen Objekte sind, können sie selbst Prototypen haben. Dies bedeutet, dass Objekte nicht nur von ihren Prototypen, sondern auch von deren Prototypen usw. erben. Diese Vererbungskette endet mit der Kette der Prototypen, in der Regel bei `Object.prototype`, das der Prototyp von Objekten ist, die nicht per Konstruktor erzeugt wurden (so u. a. von Objektliteralen; s. oben) und das selbst keinen Prototyp hat:

```
304 > Object.getPrototypeOf(Object.prototype)                    JS
    null
```

Per `Object.create(null)` können aber auch andere Objekte erzeugt werden, die keine Prototypen haben, also nichts erben.

Es ist übrigens nicht möglich, eine zirkuläre Prototypenkette herzustellen. So führt beispielsweise der nachfolgende Code zu einem Laufzeitfehler:

```
305 > Object.setPrototypeOf(o1, o1)                              JS
    Uncaught TypeError: Cyclic __proto__ value
        at Function.setPrototypeOf (<anonymous>)
```

10.5 Vererbung und dynamisches Binden

Wir hatten in Abschnitt 7.4.2 ja bereits gesehen, dass Zugriffe auf Eigenschaften eines Objekts, das durch einen variablen Ausdruck (also beispielsweise eine Variable) repräsentiert wird, also beispielsweise

```
306 o.m()                                                        JS
```

dynamisch gebunden werden: Solange der Wert von `o` nicht bekannt ist, steht nicht fest, welche Methode `m` aufgerufen wird. Die Bindung des Aufrufs kann also erst zur Laufzeit des Pro-

[77] Hiding ist von *Shadowing* zu unterscheiden, das Namen ebenfalls unsichtbar macht, aber nicht im Kontext der Vererbung, sondern durch Verwendung desselben Namens in einem geschachtelten, inneren Gültigkeitsbereich.

gramms stattfinden, wenn der Wert von `o` bekannt ist. Dies unterscheidet sich grundsätzlich von Funktionsaufrufen, bei denen die aufgerufene Funktion feststeht (sieht man einmal ab von Fällen, bei denen der Name der Funktion nicht buchstäblich im Programm steht, sondern erst durch einen Ausdruck geliefert wird), und auch von Zugriffen auf Variablen. Man spricht hier auch von **dynamischem Binden**, wobei diese Form Sprachen wie JAVASCRIPT, die über keine statische Typprüfung verfügen, immanent ist.

dynamisches Binden im Kontext der Vererbung

Der Prozess des dynamischen Bindens wie eben skizziert wird durch Vererbung erweitert: Es wird nicht nur beim Empfängerobjekt nach der Eigenschaft gesucht, auf die zugegriffen werden soll, sondern auch bei seinem Prototyp sowie rekursiv bei dessen Prototypen, und zwar so lange, bis die Eigenschaft gefunden wurde oder die Kette der Prototypen vollständig abgelaufen wurde. Während dies in JAVASCRIPT auch für den lesenden Zugriff auf Felder gilt, ergibt sich bei Methodenaufrufen eine Besonderheit: `this` verweist beim Ausführen der gefundenen Methode in deren Rumpf nicht auf das Objekt (den Prototyp), das (der) die Methode definiert, sondern auf das ursprüngliche Empfängerobjekt. Daraus ergeben sich Verhaltensweisen, die für die objektorientierte Programmierung zentral sind.

10.6 Offene Rekursion

Die Vererbung und das damit verbundene dynamische Binden erlaubt ein besonderes Idiom der objektorientierten Programmierung, das gelegentlich auch als **offene Rekursion** bezeichnet wird. Seine Wirkweise wird an folgendem Beispiel deutlich:

```
307 > let o1 = {                                                    JS
308 .   m() { this.n() },
309 .   n() { console.log(1) }
310 . }
311 > let o2 = {
312 .   n() { console.log(2) }
313 . }
314 > Object.setPrototypeOf(o2, o1)
315 > o1.m()
    1
316 > o2.m()
    2
```

Was ist hier los? Im Fall des Aufrufs `o1.m()` aus Zeile 315 ist alles klar: Es wird die Methode `m` des Objekts `o1` aufgerufen, die wiederum auf `o1`, durch `this` repräsentiert, die Methode `n` aufruft, mit dem Ergebnis, dass `1` ausgegeben wird. `o2` hingegen hat keine eigene Methode `m`, erbt eine solche aber von `o1`. Wird nun in Zeile 316 die in `o1` definierte, von `o2` geerbte Methode `m` auf `o2` aufgerufen, so verweist `this` in deren Rumpf in Zeile 308 nicht auf `o1` wie im ersten Fall, sondern auf `o2`. Die Methode `m` ruft hier also in ihrem Rumpf die `n` von `o1` überschreibende Methode `n` von `o2` auf, was zur Ausgabe von `2` führt.

Das gleiche gilt übrigens für einen Feldzugriff:

```
> o1 = {                                                                    JS
.   f: 1,
.   m() { console.log(this.f)}
. }
> o2 = { f: 2 }
> Object.setPrototypeOf(o2, o1)
> o2.m()
2
```

Inversion of control

Der Mechanismus der offenen Rekursion ist für die Bedeutung der objektorientierten Programmierung zentral. Er ermöglicht, dass in sog. *Anwendungsframeworks* (für die *GUI-Frameworks* vielleicht die bekanntesten Beispiele sind) Methoden aufgerufen werden, die nicht selbst Teil des Frameworks sind, sondern dieses erweitern. Im gegebenen Beispiel wäre `o1.m` eine solche Framework-Methode, die `o2.n` als Erweiterung aufrufen kann, obwohl `o1` selbst gar keine Kenntnis von `o2` hat — diese wird ihr über `this` quasi untergeschoben. Man nennt dieses Aufrufprinzip auch **Inversion of control** oder **Hollywood-Prinzip** („don't call us, we call you").

Wir werden in Abschnitt 16.3 wieder auf Anwendungsframeworks zurückkommen.

10.7 Delegation

Forwarding

Die offene Rekursion lässt sich auch verwenden, um in JAVASCRIPT **Delegation** zu implementieren. Dabei ist Delegation vom sog. **Forwarding** zu unterscheiden. Was Forwarding ist, wird an folgendem Beispiel deutlich:

```
const o1 = {                                                                JS
    m() { this.n() }
    n() {}
}
const o2 = {
    o: o1
    m() { o.m() }
    n() {}
}
```

Hier wird mit `o2.m()` über `o.m()` zunächst die Methode `o1.m` und in der Folge dann `o1.n` aufgerufen. Deren Aufruf durch `this.n()` hat gewissermaßen vergessen, dass `m` ursprünglich auf `o2` aufgerufen wurde — `this` bindet hier an `o1`. Das liegt daran, dass der Zugriff auf `o1.m` direkt über `o1` erfolgte und nicht über Vererbung von `m` auf `o2`. Das Objekt `o1` tritt also bei der Erledigung von `m` vollständig an die Stelle von `o2` — `o2` hat `m` an `o1` geforwardet.

Delegation

Sollte das gewünschte Verhalten hingegen sein, dass `this.n()` in `o1.m` die Methode `o2.n` aufruft, dann wird Delegation anstelle von Forwarding benötigt. Dazu müssen wir das Beispiel in zwei Punkten ändern:

1. `o2` muss `o` (und damit `o1`) als Prototypen nennen.
2. Der Aufruf `o.m()` muss durch `super.m()` ersetzt werden.

Damit wäre der Aufruf `this.n()` offen rekursiv und würde `o2.n` zur Ausführung bringen. Bei der Delegation wird also natives Verhalten des Objekts, an das der ursprüngliche Methodenaufruf gerichtet war, in das Verhalten des Objekts, an das delegiert wurde, eingespleißt.

Offensichtlicher Nachteil dieser Methode der Umsetzung von Delegation ist, dass hier immer nur an ein Objekt delegiert werden kann. Soll an mehrere (oder über die Zeit verschiedene) delegiert werden, muss jeweils unmittelbar zuvor der Prototyp gewechselt werden. Da das im Allgemeinen weiterreichende Konsequenzen hat als nur die offene Rekursion bei einem Aufruf, generalisiert dieser Ansatz nicht besonders gut. An dessen Stelle kann man in Zeile 330 aber einfach

```
333 o.m.call(this)                                                    JS
```

schreiben: Dadurch wird wie zuvor `o1.m` aufgerufen, `this` darin aber nicht an `o1` gebunden, sondern an `o2` (was ja die Bedeutung von `this` in Zeile 330 ist). Somit ruft auch `this.n()` in Zeile 325 `o2.n` auf, was hier gewollt war.

Die Methode `call` ist übrigens in `Function.prototype` definiert und wird von allen Funktionen geerbt.

10.8 Typische Einsatzgebiete prototypenbasierter Vererbung

Die Vererbung zwischen Objekten (und nicht zwischen Klassen wie in vielen anderen objektorientierten Programmiersprachen) als Sprachmerkmal ist überall dort sinnvoll, wo sich verschiedene Objekte regelmäßig nicht nur in den Werten ihrer Eigenschaften unterscheiden (wie das bei den Instanzen von Klassen in klassenbasierten objektorientierten Programmiersprachen der Fall ist), sondern auch in deren Zahl und Namen. Zudem erlaubt der Prototypenansatz auf besonders einfache Weise, das Verhalten von einzelnen Objekten abzuändern, indem jedes Objekt seine eigene Implementierung einer bestimmten Funktion vorsehen kann. Dies ist z. B. bei der Implementierung von GUIs nützlich, wenn dasselbe Ereignis (z. B. Klicken) je nach Objekt unterschiedliche Reaktionen auslösen soll (dies ist typischerweise bei unterschiedlichen Buttons der Fall, die sich nicht nur in ihrer Beschriftung, dem Wert eines Feldes, sondern der auch mit ihnen verbundenen Funktion unterscheiden). Es ist wohl eher kein Zufall, dass JavaScript, als Sprache für ausführbaren Inhalt von Webseiten konzipiert, prototypenbasiert ist.

11 Serialisierung und Deserialisierung von Objekten

An den Objektliteralen aus Abschnitt 7.1 haben wir gesehen, dass Objekte, darunter auch geschachtelte, in textueller und damit in linearer Form dargestellt werden können. Eine solche lineare Form der Darstellung von Objekten kann auch außerhalb von Programmtexten nützlich sein, z. B. zur Speicherung von Objekten als Text oder zum Austausch von Objekten zwischen Programmen oder Programmteilen, die sich keinen gemeinsamen Speicher teilen. Tatsächlich

ist die Syntax der Objektliterale von JAVASCRIPT Pate eines der derzeit am weitesten verbreiteten Datenaustauschformate, JSON (für JAVASCRIPT OBJECT NOTATION) genannt.

Damit ein Objekt sich selbst oder andere serialisieren kann, bedarf es der Einsicht in die Struktur (oder den Aufbau) eines Objekts. Die Fähigkeit, diese Einsicht zu erhalten, nennt man in der Programmierung **Introspektion** (gelegentlich, und etwas ungenauer, auch als *Reflexion* bezeichnet). JAVASCRIPT ist reich an Möglichkeiten der Introspektion.

Für die Serialisierung von Objekten kann ausgenutzt werden, dass Objekte in JAVASCRIPT Container von Name/Wert-Paaren sind (Kapitel 4), wobei die Namen Werte sind. So lassen sich mittels

```
334 for (name in o) console.log(name + ': ' + o[name])                    JS
```

alle Eigenschaften (Name/Wert-Paare) eines Objekts `o`, die als `enumerable` attributiert sind, auf der Konsole ausgeben. Dabei wird ausgenutzt, dass nicht nur primitive Werte, sondern auch alle Objekte eine Stringrepräsentation haben, in die sie im oben auf der Konsole ausgegebenen Ausdruck gezwungen werden. Allerdings gibt die Standardimplementierung dieser Konversion nur eine sehr knappe Darstellung eines Objekts:

```
335 > console.log(String({name: 'Hein'}))                                 JS
    [object Object]
```

Um Objekte vollständig zu serialisieren, fehlt die rekursive Serialisierung der Objekte, die Werte von Eigenschaften des zu serialisierenden Objekts sind. Das erledigt die folgende Funktion in einem ersten Ansatz:

```
336 function serialisiere(o) {                                            JS
337     if (typeof o !== 'object' || o === null)
338         return typeof o === 'string' ? `"${o}"` : String(o)
339     let s = '{ '
340     for (const name in o) s += `${name}: ${serialisiere(o[name])}, `
341     if (s.length > 2) s = s.slice(0, -2); // entfernt das letzte ", "
342     s += ' }'
343     return s
344 }
```

Allerdings führt der Aufruf dieser Funktion mit einem Objekt als Argument, das sich selbst enthält (wie in Abschnitt 7.2 konstruiert), in eine Endlosrekursion. Diese muss in einer ernsthaften Serialisierung abgefangen werden.

12 Selbstmodifikation von Programmen

Tatsächlich ist JAVASCRIPT nicht nur introspektiv, sondern reflexiv: Es versetzt seine Programme nicht nur in die Lage, sich selbst zu erkennen, sondern auch, sich selbst zu verändern. So erwirkt die Abwandlung des Codes von Zeile 334,

```JS
for (name in o) p[name] = o[name]
```

die Übertragung aller Name/Wert-Paare von `o` auf `p`, wobei Namen, die `p` noch nicht hat, neu angelegt werden. Auch können einzelne Eigenschaften eines Objekts mittels `delete` gelöscht werden:

```JS
for (name in o) delete o[name]
```

macht aus `o` ein Objekt ohne eigene Eigenschaften (geerbte werden nicht gelöscht!). Auch beim Wechseln des Prototyps eines Objekts handelt es sich strenggenommen um Reflexion, da sich hierdurch die Eigenschaften eines Objekts (nicht deren Werte wie bei einer Zuweisung) ändern.

Während die Darstellung von Objekten als manipulierbare Datenstruktur (Container) in JAVASCRIPT den Unterschied zwischen einer „normalen“ (nicht reflexiven) Programmierung und Reflexion weitgehend auflöst, ist die Möglichkeit, ein Programm Deklarationen sich selbst hinzufügen zu lassen, von einer anderen Qualität. Aber auch das ist in JAVASCRIPT kein Problem:

```JS
> const summe = new Function('a', 'b', 'return a + b;');
> summe(1, 2)
3
```

Hier wird aus einzelnen Strings eine neue Funktion erzeugt, die sich von einer durch ein Funktionsliteral entstandenen nicht unterscheidet. Dabei ist ein Funktionsliteral ein Teil des (durch das Programm unveränderlichen) Programmtexts und kein String (kein Stringliteral).

Es geht in JAVASCRIPT aber sogar noch freier:

```JS
> eval("var x = 1")
> x
1
```

Eine derartig unbeschränkte Reflexion ist aber aus vielerlei Hinsicht problematisch: Sie erschwert nicht nur statische Analysen eines Programms vor seiner Ausführung (inkl. der Syntaxprüfung!), sondern öffnet auch dem Einschleusen von Schadsoftware Tür und Tor. Sie sollte daher vermieden werden.