Prof. Dr. Friedrich Steimann

63618
Objekt-funktionale Programmierung

Version 2♂

Leseprobe

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks, verbleiben bei der FernUniversität. Das Werk darf gemäß § 53 UrhG im privaten Rahmen zum Beispiel durch Kopien oder digitale Speicherung vervielfältigt werden.

Inhaltsverzeichnis

Vorwort ..... **i**

Didaktisches Konzept i

Verwendete Programmiersprachen ii

Zur Form iii

Zur Sprache iv

Zur Prüfung v

Lektion 1: Voraussetzungen **1**

1 Programmierung **1**

1.1 Computer 1

1.2 Programme 2

1.3 Daten 2

1.4 Programmiersprachen 3

1.5 Compiler und Interpreter 4

1.6 Definition von Programmiersprachen 5

1.6.1 Syntax 5

1.6.2 Semantik 5

1.6.3 Warum konkrete Syntax keine Nebensache ist 6

1.6.4 Die Rolle von Bibliotheken 7

1.7 Programmierparadigmen 7

2 Imperative Programmierung **9**

2.1 Die Elemente imperativer Programmierung 9

2.1.1 Kontrollflussanweisungen 9

2.1.2 Werte, Variablen und Wertzuweisung 10

2.1.3 Unterprogramme, lokale Variablen und Geltungsbereiche 14

2.1.4 Prozeduren und Funktionen 16

2.1.5 Ausdrücke 17

2.1.6 Strukturierte Daten 18

2.1.7 Verweise oder Zeiger 21

2.1.8 Variante Datenstrukturen 24

2.1.9 Parameterübergabe: Call-by-value und Call-by-reference 25

2.1.10 Aliasing und Sharing 26
2.2 Speicherverwaltung 27
2.2.1 Stack 27
2.2.2 Heap 28
2.3 Das Wesen imperativer Programmierung 29

3 Funktionale Programmierung 30
3.1 Funktionen in der Mathematik 31
3.2 Funktionen in der Programmierung 33
3.3 Der Lambda-Kalkül 35
3.3.1 Funktionen als Abstraktionen 36
3.3.2 Funktionsanwendung als Substitution 38
3.3.3 Funktionen als Werte und Funktionen höherer Ordnung 38
3.3.4 Syntax des Lambda-Kalküls 40
3.3.5 Semantik des Lambda-Kalküls: β-Reduktion 42
3.3.6 Freie Vorkommen von Variablen und offene Ausdrücke 45
3.3.7 Das Phänomen der Divergenz 46
3.3.8 Das Problem mit den Variablennamen 46
3.3.9 Funktionen mit mehreren Argumenten: Currying 48
3.3.10 Church-Codierungen 49
3.3.11 Rekursion 54
3.4 Vom Lambda-Kalkül zur Programmiersprache 56
3.4.1 Ausdrücke als Programme 56
3.4.2 Operationale Semantik 57
3.4.3 Laufzeitumgebungen anstelle von Substitution 61
3.4.4 Natürliche Semantik 65
3.4.5 Variablen, Let-Ausdrücke und benannte Funktionen 68
3.4.6 Rekursion 69
3.4.7 Weitere Sprachelemente als abgeleitete Formen 70
3.4.8 Speicher und Adressen 71
3.5 Das Wesen funktionaler Programmierung 73

Literaturhinweise 75

Lektion 2: Objekte 77

4 Objekte als Strukturen ... 78

4.1 Eigenschaften von Objekten ... 78

4.2 Beziehungen zwischen Objekten ... 79

4.3 Oberfläche und interne Repräsentation von Objekten ... 80

5 Metaphysik der Objekte ... 81

5.1 Werte vs. Objekte ... 81

5.2 Veränderlichkeit von Objekten ... 84

5.3 Gleichheit und Identität ... 85

6 Imperative Basis objektorientierter Sprachen ... 85

6.1 Anweisungen ... 86

6.2 Primitive Werte und ihre Datentypen ... 87

6.2.1 Literale Repräsentationen primitiver Werte ... 88

6.2.2 Ausdrücke über primitive Werte ... 89

6.2.3 Gleichheit von primitiven Werten ... 90

6.3 Variablen und Konstanten ... 91

6.4 Arrays ... 93

6.5 Funktionen und Prozeduren ... 94

7 Programmieren mit Objekten ... 96

7.1 Objektliterale ... 96

7.2 Felder ... 99

7.3 Methoden ... 102

7.3.1 Methodendefinitionen ... 102

7.3.2 Methodenaufruf und dynamisches Binden ... 103

7.3.3 Datenkapselung mit Queries, Commands und Zugriffsmethoden ... 105

7.4 Attribute ... 107

7.5 Objekte als Namensräume ... 107

7.6 Wert- und Referenzsemantik sowie Aliasing bei Objekten ... 109

7.7 Gleichheit und Identität von Objekten ... 112

7.8 Lebensdauer von Objekten ... 115

8 Objektkonstruktion ... 115

8.1 Fabrikfunktionen ... 116

8.2 Konstruktoren ... 116

8.3 Konstruktoren als Gemeinsamkeiten definierende Elemente 118
8.4 Standardkonstruktoren 119
8.5 Boxing und Autoboxing 120
8.6 Dualer Nutzen von Funktionen in JavaScript 121
8.7 Kopieren von Objekten 121

9 Prototypen und Vererbung 122
9.1 Prototypen 123
9.2 Vererbung über Konstruktoren 123
9.3 Vererbung ohne Konstruktoren 126
9.4 Überschreiben 127
9.5 Die Prototypenkette 128
9.6 Vererbung und dynamisches Binden 129
9.7 Offene Rekursion 130
9.8 Delegation 131
9.9 Typische Einsatzgebiete prototypenbasierter Vererbung 132

10 Funktionen als Objekte 133
10.1 Funktionsliterale 133
10.2 Funktionsanwendung 135
10.3 Funktionsabschlüsse 136
10.4 Currying 138
10.5 Funktionen als Parameter: Kontrollabstraktion 139
10.6 Beförderung von Funktionen zu Methoden 140

11 Objekte als Funktionen 141
11.1 Datenkapselung durch Funktionsabschlüsse 142
11.2 Funktionale Objekte 143

12 Reflektion 147
12.1 Serialisierung von Objekten 147
12.2 Selbstmodifikation 148

13 Das Wesen von Objekten 149

Lektion 3: Klassen 151

14 Objekte und Klassen 152

14.1 Metaphysik der Klassen 153
14.2 Klassifikation und Instanziierung 154

15 Ausdrücke in RUBY 156
15.1 Literale 156
15.2 Variablen 159
15.3 Instanziierung 160
15.4 Methodenaufrufe 160
15.5 Operatoren 160
15.6 Sequenz 161
15.7 Weitere Kontrollstrukturen 162
15.8 Funktionsobjekte 163
15.9 Kontrollabstraktionen in RUBY 165
15.9.1 Nicht-lokales Return 167
15.9.2 Blöcke als Objekte 168

16 Klassendefinitionen 170
16.1 Definition zur Laufzeit 171
16.2 Konstruktoren 172
16.3 Methoden 173
16.3.1 Datenkapselung 173
16.3.2 Dynamisches Binden 175
16.3.3 Methoden als Objekte 177
16.4 Klassenvariablen und -methoden 179

17 Metaklassen 182
17.1 Individuelle Metaklassen in RUBY 184
17.2 Metaprogrammierung 186
17.3 Erzeugung von Objekten 187
17.4 Metaklassen jenseits der Programmierung 188

18 Superklassen 188
18.1 Generalisierung und Spezialisierung 188
18.1.1 Generalisierung 189
18.1.2 Spezialisierung 192
18.2 Vererbung und Überschreiben 194

18.2.1 Vererbung ... 194
18.2.2 Redefinition und Überschreiben von Methoden ... 197
18.2.3 Löschen von Eigenschaften ... 199
18.3 Abstrakte Klassen und offene Rekursion ... 199
18.3.1 Whitebox-Frameworks ... 201
18.3.2 Das Fragile-base-class-Problem und die Vererbungsschnittstelle ... 203
18.4 Vererbung und Metaklassen ... 205

19 Andere Formen der Wiederverwendung von Klassen ... 206
19.1 Komposition ... 206
19.2 Module ... 208
19.3 Mixins ... 210

20 Wertklassen ... 213
20.1 Wertklassen mit Referenzsemantik ... 216
20.2 Wertklassen mit Wertsemantik ... 220
20.3 Probleme der Koexistenz von Objekten und Werten ... 221
20.3.1 Simulierte Werte ... 221
20.3.2 Echte Werte ... 223

21 Collection-Klassen ... 224
21.1 Algebraische Grundlage von Collections ... 224
21.2 Operationen auf Collections ... 226
21.3 Dictionaries ... 229
21.4 Funktionale Collections ... 229
21.5 Unendliche Collections ... 232

22 Relationen ... 233
22.1 Zu-eins-Relationen ... 233
22.2 Zu-n-Relationen ... 237
22.3 Rollen ... 239
22.4 Generalisierung und Spezialisierung von Relationen ... 241
22.5 Anbindung an relationale Datenbanken ... 242

23 Verhalten für alle Objekte ... 243

24 Klassen in JavaScript ... 244

25 Das Wesen von Klassen 247

Lektion 4: Typen 249

26 Ursprung der Typentheorie 250

26.1 Russells Typentheorie 250

26.2 Churchs einfach getypter Lambda-Kalkül 251

26.2.1 Erweiterung des Lambda-Kalküls um Typen 253

26.2.2 Typrelation und Typableitung 257

26.2.3 Normalisierungseigenschaft 261

26.2.4 Type erasure 262

26.2.5 Übertragung auf die Programmierung 262

27 Datentypen 264

27.1 Primitive Datentypen 265

27.2 Literal types 267

27.3 Zusammengesetzte Datentypen 267

27.3.1 Objekttypen 268

27.3.2 Array-Typen 271

27.3.3 Rekursive Datentypen 271

27.3.4 Algebraische Datentypen 272

28 Abstrakte Datentypen 275

28.1 Algebraische Definition abstrakter Datentypen 275

28.2 Axiomatische Definition abstrakter Datentypen 277

28.3 Objektorientierte Fassung von abstrakten Datentypen: Interfaces 278

29 Funktionstypen 279

29.1 Überladen von Funktionen 282

29.2 Typen von Konstruktoren 284

30 Klassen als Typen 285

30.1 Zugriffsmodifikatoren in Klassen 286

30.2 Abstrakte Klassen als Typen 288

31 Typen von Klassen 289

32 Typkompatibilität und Subtyping 290

32.1 Arten der Typkompatibilität ... 291
32.2 Typkompatibilität bei Funktionsaufrufen ... 294
32.3 Varianz ... 296
32.3.1 Varianz bei Funktionstypen ... 296
32.3.2 Varianz bei Objekttypen ... 298
32.4 Vereinigung und Durchschnitt von Typen ... 305
32.4.1 Union-Typen ... 306
32.4.2 Intersection-Typen ... 310
32.4.3 Typverband ... 312
32.4.4 Nutzen eines leeren Typs ... 313
32.5 Kompatibilität von Klassen- und Interfacetypen ... 314
32.5.1 Interfaceimplementierung ... 315
32.5.2 Schnittstellentrennung ... 317
32.5.3 Subclassing ... 318
32.5.4 Subtyping und Zugriffsmodifizierer ... 319
32.5.5 Interfaceerweiterung ... 321
32.5.6 Verhältnis zu algebraischen Datentypen ... 321
32.5.7 Verhältnis zu Union- und Intersection-Typen ... 323
32.5.8 Kritik der Kontravarianz ... 324
32.6 Typzusicherungen und Type casts ... 327
32.7 Implizite Typumwandlungen ... 328
32.8 Der Typ `any` ... 328

33 Null-Sicherheit und sichere Initialisierung ... 329

34 Typabstraktion ... 331
34.1 Ausgangslage ... 334
34.2 Polymorphe Funktionen ... 335
34.2.1 Polymorpher Lambda-Kalkül ... 335
34.2.2 Polymorphe Funktionen in TypeScript ... 340
34.3 Typkonstruktoren ... 343
34.3.1 Ein Lambda-Kalkül mit Typkonstruktoren ... 343
34.3.2 Typkonstruktoren in TypeScript ... 346
34.3.3 Instanziierung parametrisierter Klassen ... 350
34.3.4 Weitere Typoperatoren in TypeScript ... 352

34.4 Polymorphe Funktionen vs. Typkonstruktoren 354
34.4.1 Unterschiede 354
34.4.2 Gemeinsame Verwendung 355
34.5 Reifizierung von Typargumenten 357
34.6 Typkonstruktoren und Varianz 358
34.6.1 Definition der Varianz 360
34.6.2 Einstellen der Varianz von Typkonstruktoren 362
34.6.3 Varianz eingebauter Typkonstruktoren 364
34.7 Beschränkte Typabstraktion 366
34.7.1 Polymorpher Lambda-Kalkül mit Typbeschränkungen 367
34.7.2 Typkonstruktoren mit Typbeschränkungen 369
34.7.3 Rekursive Beschränkung 370

35 Typinferenz 373
35.1 Typinferenz in TypeScript 373
35.2 Bedeutung der Typinferenz für die Programmierung 377

36 Graduelle Typsysteme 379

37 Das Wesen von Typsystemen 381

Lektion 5: Scala 383

38 Scalas Syntax 384

39 Ontologie Scalas 387
39.1 Objekte und Klassen 387
39.1.1 Klassen 387
39.1.2 Traits 391
39.1.3 Innere Klassen 397
39.1.4 Singleton-Objekte 398
39.1.5 Companion-Objekte 398
39.1.6 Anonyme Klassen 401
39.1.7 Tupel und Records 401
39.2 Objekte und Werte 402
39.2.1 Einordnung von Werten in Scalas Klassenhierarchie 403
39.2.2 Gleichheit und Identität 405

39.2.3 Case classes 406
39.2.4 Enumerationen 410
39.3 Objekte und Funktionen 412
39.3.1 Funktionen als Objekte 412
39.3.2 Objekte als Funktionen 416
39.3.3 Anonyme Zugriffsmethoden 416
39.3.4 Erweiterungsmethoden 418

40 SCALAs Typsystem 419
40.1 Typaliase 420
40.2 Typkonstruktoren 421
40.2.1 Klassen- und andere Produkttypen 421
40.2.2 Funktionstypen 422
40.2.3 Union- und Intersection-Typen 424
40.2.4 Generische Typen 424
40.3 Metatypen 427
40.4 Singleton-Typen 429
40.5 Type member 431
40.6 Abstrakte Typen 434
40.7 Pfadabhängige Typen 436

41 Kontextabstraktion 439

42 *Kontrollabstraktionen* in SCALA 443
42.1 Verzögerte Auswertung 444
42.2 Sequenzabstraktion 447
42.3 Pattern matching 453

43 Das Wesen SCALAs 460

Lektion 6: F# 463

44 Die Syntax von F# 463

45 Der Lambda-Kalkül in F# 465
45.1 Funktionen 465
45.1.1 Anonyme Funktionen 465
45.1.2 Benannte Funktionen 466

45.1.3 Funktionsanwendung ... 467
45.1.4 Rekursive Funktionen ... 469
45.1.5 Muster als Parameter ... 469
45.2 Let-Ausdrücke ... 471
45.3 Sequenz ... 473
45.4 Pipelining und Funktionskomposition ... 474

46 Das Typsystem von F# ... 476
46.1 Zeigertypen ... 477
46.2 Algebraische Datentypen ... 478
46.2.1 Produkttypen ... 478
46.2.2 Summentypen ... 482
46.2.3 Algebraische Datentypen als echte Werttypen ... 483
46.2.4 Der Options- und der Listentyp ... 484
46.2.5 Companion-Module ... 486
46.3 Collections ... 487
46.3.1 Arrays ... 487
46.3.2 Sequenzen ... 488
46.4 Objektorientierte Datentypen ... 491
46.4.1 Member ... 491
46.4.2 Klassen ... 493
46.4.3 Vererbung, Überschreiben und abstrakte Klassen ... 495
46.4.4 Interfaces und ihre Implementierung ... 497
46.4.5 Objektausdrücke ... 499
46.5 Typconstraints ... 499

47 Kontrollabstraktionen in F# ... 500
47.1 Pattern matching ... 500
47.2 Computation expressions und monadische Typen ... 503

48 Metaprogrammierung in F# ... 508
48.1 Type provider ... 508
48.2 Quotations ... 509

Lektion 7: Objekt-funktionale Perlen ... 513

49 Fluent Interfaces ... 513

50 Verarbeitungspipelines ... 515

50.1 Trägheit gewinnt ... 517

50.2 Stream fusion ... 519

51 Anfragesprachen ... 519

51.1 Language integrated query (LINQ) ... 520

51.2 MapReduce ... 524

52 Type-driven development ... 525

53 Implementierung von Parsern ... 531

53.1 Parsen per rekursivem Abstieg ... 532

53.2 Parserkombinatoren ... 535

54 Memoization ... 539

Nachwort ... 543

Literaturverzeichnis ... 545

Verzeichnis der Weblinks im Rand ... 547

Index ... 551

Lektion 2: Objekte

Das Mantra der objektorientierten Programmierung lautet:

> Alles ist ein Objekt.

Wie das bei Mantras häufig der Fall ist, lässt auch dieses ein weites Interpretationsspektrum zu. An dessen einem Ende kann es bedeuten, dass mit dem Begriff des Objekts nicht besonders viel verbunden ist. In diesem Fall bedeutet es nicht viel, wenn alles darunterfällt. 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 und Funktionen Objekte, die durch ein Programm, selbst ein Objekt, manipuliert werden können. Damit wäre die objektorientierte Programmierung ein direkter Gegenentwurf zur funktionalen Programmierung, in der alles eine Funktion ist. Eine solche Homogenität ist in ihrer Absolutheit jedoch kaum zu erreichen: So ist auch im Lambda-Kalkül nicht alles eine Funktion — es gibt auch Variablen und Funktionsanwendungen. Entsprechend verhält es sich mit Objekten in der objektorientierten Programmierung.

Es sind somit 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. Insofern ist das Mantra also durchaus zutreffend.

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 gerichteten *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 entspricht. 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, sondern bringt seine eigene Umgebung mit Standardein- und -ausgabe mit sich. Dabei gibt es 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 oder einem Web-Browser funktioniert.

4.1 Eigenschaften von Objekten

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

```
31 {                                                                    JS
32   vorname: 'Hein',
33   nachname: 'Blöd',
34   alter: 42,
35   hallo: function() { console.log('Käpt´n, Käpt´n!') },
36   name: function() { return this.vorname + ' ' + this.nachname }
37 }
```

in JAVASCRIPT für ein Objekt mit den fünf aufgelisteten (und durch Kommata getrennten) Eigenschaften. Objekte entsprechen damit zunächst der Datenstruktur des Records (s. Abschnitt 2.1.6[55]) mit der Besonderheit, dass in JAVASCRIPT auch Funktionen als Werte von Objekteigenschaften auftreten können: So stehen die den Namen `hallo` und `name` zugeordneten Funktionen gleichberechtigt neben den Werten der anderen Name/Wert-Paare des Objekts. Dabei sind beide Funktionen *anonym*; benannt werden sie, genau wie die Werte der anderen Name/Wert-Paare, lediglich über ihre Rolle als Eigenschaften des Objekts (hier die Eigenschaf-

[55] Den einen oder anderen Leser werden JAVASCRIPT-Objekte eher an Dictionaries oder Hash maps erinnern und tatsächlich sind sie das ihrem Wesen nach auch (mehr dazu in Abschnitt 7.2); allerdings sind Dictionaries oder Hash maps in der Regel auch Objekte, so dass dieser Vergleich hier nicht weiterhilft.

ten `hallo` und `name`).[56] Zudem 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 im Rumpf der Funktion den Zugriff auf die Eigenschaften `vorname` und `nachname`). Der implizite Parameter `this` (in JAVASCRIPT ein Schlüsselwort, seinem Wesen nach aber eher eine *Pseudovariable*) steht dabei für das Objekt, dem die Funktion als Eigenschaft zugeordnet ist. `this` kann bei Zugriffen auf die Eigenschaften eines Objekts aus dem Objekt selbst heraus (wie hier in der Funktionen `name` der Fall) anders als beispielsweise in JAVA nicht weggelassen werden.

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, darunter selbst solchen mit gleichen (oder identischen) Eigenschaften, unterscheidet. So hat denn auch ein eigenschaftsloses, oder leeres, Objekt `{}` in JAVASCRIPT eine Identität, die es von anderen leeren Objekten unterscheidet:

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

Hierbei steht `==` für den Test auf Identität und nicht auf Gleichheit (wie es beispielsweise bei `1 == 1` der Fall wäre). *Dass Objekte eine Identität haben, die sie unabhängig von Gleichheit unterscheidet, ist eines der zentralen Wesensmerkmale der objektorientierten Programmierung.* Ich werde in Kapitel 5 und Abschnitt 7.7 wieder darauf zurückkommen.

4.2 Beziehungen zwischen Objekten

Objekte als Werte von Eigenschaften

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

```
39 {                                                             JS
40   nachname: 'Blöd'
41   chef: {
42     nachname: 'Blaubär'
43   }
44 }
```

ein Objekt. Genaugenommen definiert es sogar zwei Objekte, denn anders als dies bei Records der Fall ist, ist durch eine solche Schachtelung von Objekten grundsätzlich keine *Teil-Ganzes-*

[56] Im allgemeinen Sprachgebrauch ist mit Eigenschaft eines Objekts häufig nicht ein Name/Wert-Paar gemeint, sondern nur der Umstand, dass ein Objekt die Eigenschaft überhaupt besitzt (wie z. B. in „das Objekt hat die Eigenschaft Name“), ohne dass man dabei den Wert der Eigenschaft im Sinn hätte, wobei ja nach obiger Definition erst das Name/Wert-Paar die Eigenschaft wäre. Das habe ich hier auch getan und werde es auch weiter tun. In Lektion 3 werden wir sehen, dass sich diese doppelte Bedeutung des Wortes Eigenschaft (Anwendbarkeit auf ein Objekt und *Ausprägung* durch ein Objekt) auch in den höheren Konstrukten der objektorientierten Programmierung wiederfindet, und in Lektion 4, dass auf der Ebene der Anwendbarkeit Typen die Werte von Eigenschaften ersetzen.

Beziehung begründet (vgl. dazu die Abschnitte 2.1.6 und 2.1.7) — vielmehr steht das äußere Objekt mit dem inneren nur in loser Beziehung. Insbesondere gehört das innere Objekt nicht zum äußeren und kann deswegen nicht nur fortbestehen, wenn das äußere nicht mehr ist, sondern auch gleichzeitig in derselben oder anderen Beziehungen mit mehreren anderen Objekten stehen (wobei das im obigen Beispiel zunächst nicht der Fall ist — wie solche Beziehungen entstehen, erfahren wir in Kapitel 7).

Beziehungen zu mehreren Objekten

Nicht selten steht ein Objekt nicht mit einem, sondern mit mehreren anderen Objekten in derselben Beziehung. Dafür mehrere Name/Wert-Paare zu verwenden wird der Sache schon deswegen nicht gerecht, weil jedes Paar jeweils eine, seine eigene, Beziehung herstellt, und funktioniert spätestens dann nicht mehr, wenn die Zahl der so zu einem Objekt in einer Beziehung stehenden Objekte variabel und dabei potentiell unbegrenzt ist.

Arrays als Hilfsobjekte

In diesen Fällen kann ein *Array* als Wert einer Eigenschaft helfen, dessen Elemente dann die Objekte sind, mit denen ein Objekt eigentlich in Beziehung steht. Eigentlich, weil das Objekt tatsächlich mit dem Array in Beziehung steht, das hier aber nur die Rolle eines weiteren Containers spielt, der stellvertretend für die vielen Objekte steht, die seine Elemente sind. So stellt

```JS
{
  nachname: 'Blaubär',
  enkel: [ { name: 'Gelb' }, { name: 'Grün' }, { name: 'Rosa' } ]
}
```

die Enkelbeziehung zwischen dem äußeren und den drei inneren (innersten) Objekten her. Man bemerke die Parallelität zur Darstellung von Relationen mittels Abbildungen auf Mengen in Abschnitt 3.2.

4.3 Oberfläche und interne Repräsentation von Objekten

Die Eigenschaften der bisher verwendeten Objekte repräsentieren sie in einem gegebenen (hier nur angenommenen) Kontext: Abgesehen von der Identität sind Name, Alter, Chef und Enkel das, was ein Objekt auszeichnet. Sie charakterisieren also gewissermaßen Objekte über ihre sichtbare **Oberfläche**.

interne Repräsentation

Die Implementierung eines Objekts, insbesondere die seines Verhaltens (von dem wir hier noch wenig gesehen haben), verlangt aber in aller Regel noch weitere Eigenschaften des Objekts. Sofern diese für seine Verwendung in irgendwelchen Kontexten nicht direkt relevant sind, können sie nach außen unsichtbar bleiben. Im gegebenen Beispiel könnten das die Eigenschaften `vorname` und `nachname` sein, nämlich dann, wenn in allen Verwendungen des Objekts die Eigenschaft `name` ausreicht. Man spricht von diesen Eigenschaften dann als der **internen Repräsentation** eines Objekts. Je nach Programmiersprache können Oberfläche und interne Repräsentation, die sich ja beide gleichermaßen in Eigenschaften (Daten und Funktionen) eines Objekts niederschlagen, mehr oder weniger gut vonei-

nander abgegrenzt werden; im Fall von JAVASCRIPT gelingt das tatsächlich weniger gut mit der unmittelbaren Folge, dass dort auch die (eigentlich) interne Repräsentation zur Oberfläche eines Objekts gehört.

Repräsentationsobjekte

Man spricht von Objekten, die Teil der internen Repräsentation eines Objekts sind, auch von **Repräsentationsobjekten**. Es entspricht ihrer Funktion, exklusiver Teil der Repräsentation genau eines Objekts zu sein. Insofern ist die Beziehung von einem Objekt zu seinen Repräsentationsobjekten ihrem Wesen nach die Beziehung eines Ganzen zu seinen Teilen. Da diese aber selbst durch die Schachtelung von Objekten nicht sicher hergestellt wird, sind zusätzliche Mechanismen erforderlich, die die interne Repräsentation vor Zugriffen von außen schützen.

5 Metaphysik der Objekte

Sowohl in der imperativen als auch in der funktionalen Programmierung nehmen Werte eine zentrale Stellung ein: Ausdrücke werten zu Werten aus und die Ergebnisse der Auswertungen, die Werte, können Variablen zugewiesen werden (in der Sprechweise der funktionalen Programmierung auch an sie gebunden; aber s. Fußnote 32 in Abschnitt 3.3.1). Werte stellen die Ein- und Ausgaben eines Programms dar 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).

5.1 Werte vs. Objekte

Was aber sind Werte? Zahlen beispielsweise sind zunächst einmal mathematische Objekte 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 auch 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 Merkmalen 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 die Objekte, und was sind Objekte überhaupt? Zur Beantwortung dieser für die objektorientierte und die objekt-funktionale Programmierung zentralen Frage möchte ich weiter ausholen.

Individuen und Universalien

Aus der Metaphysik wohlbekannt ist die Unterscheidung zwischen *Individuen* (auch Partikularien oder Einzeldingen) und *Universalien* (allgemeine Eigenschaften oder Arten). Wie man die beiden systematisch unterscheiden kann, 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 eine Universalie 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 oder Wahr bemessen wird.[57] 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 ein Individuum blau und eine es betreffende 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 bestimmte Stelle in der als Folge von Zuständen gedachten Ausführung eines Programms und ein Ort bezeichnet eine bestimmte Stelle im Speicher, gedacht als eindeutig identifizierbare, oder 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 als Eigenschaften etwas 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 eine Universalie, 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 das Umgekehrte ausnutzen, nämlich 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

[57] Das gilt auch für einen Allgemeinbegriff (oder eine Art) 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, wo sie uns als Klassen begegnen werden.

ihrer Existenz immer am selben Ort bleiben (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 unrealistisch sie in der Realität auch erscheinen mag, ist in einem Computer aber tatsächlich kaum eine Einschränkung, denn aufgrund der niedrigen Dimensionalität des Speichers (eindimensional, wenn man Adressen als geordnet annimmt, sonst sogar dimensionslos) und der damit einhergehenden eingeschränkten oder gar mangelnden Nachbarschaft gibt es keinen Grund für Individuen, sich zu bewegen.[58] 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: 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; Navigation

Nun sind auch Referenzen technisch Werte und so bleibt die Frage, wie man Referenzen in unserer Welt von Universalien und Individuen (in der sie als Werte Universalien sein müssten) interpretieren soll. Wie schon in Abschnitt 2.1.7 dargelegt, sind Referenzen als Zeiger Adressen, benennen also Orte. Sofern sich an einem Ort (unter einer Adresse) ein Individuum (Objekt) befindet, zeigt die Referenz auf das Individuum (Objekt), das mit dem Ort (der Adresse) verbunden ist. 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. Ein Referenzfeld eines Objekts implementiert somit eine **gerichtete Beziehung** zwischen zwei Objekten, nämlich dem Objekt, zu dem die Referenz gehört, und dem Objekt, auf das die Referenz verweist. Man nennt das Verfolgen einer solchen Beziehung, also ihre *Dereferenzierung*, auch **Navigation**; sie ist elementarer Bestandteil des „Rechnens" mit Objekten. In der Metaphysik sind Beziehungen allerdings keine Attribute von Individuen, sondern *Relationen* (oder *Prädikate*), die mehrere Individuen in einen Zusammenhang stellen.

Referenzen als endliche Abbildungen

Tatsächlich kann eine Referenz, die einem Objekt als Eigenschaft beigeordnet ist, mathematisch als eine *endliche Abbildung* oder *Funktion* betrachtet werden, da sie einem Objekt (höchstens) ein anderes zuordnet und die Anzahl der abgebildeten Objekte schon aufgrund der Begrenztheit von Speicher immer endlich ist. Die Abbildung kann also durch eine Aufzählung ihrer Paare, beispielsweise in Form einer Tabelle oder eben im Spei-

[58] 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.

cher, dargestellt werden. Die Abbildung ist zudem *benannt*; ihr Name ist der Name der Eigenschaft, deren Wert die Referenz ist. Jedes Objekt, das eine Eigenschaft mit dem gleichen Namen hat, trägt somit einen, nämlich seinen, Teil zur Definition einer Abbildung mit dem Namen der Eigenschaft bei. In Lektion 4 werden wir sehen, wie die *Definitions-* und *Wertebereiche* von solchen Funktionen zudem 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) geben kann — diese müssen ja schließlich auch irgendwo im Speicher stehen und deswegen Speicherzellen belegen. Solche Werte wären dann die Werte von Variablen, die keinem Objekt als Eigenschaft zugeordnet sind. In JAVASCRIPT müssen wir uns die Frage nicht stellen, da globale Variablen einem *globalen Objekt* als Eigenschaften zugeordnet und Funktionsparameter sowie lokale Variablen Funktionen sind, die in JAVASCRIPT auch zu den Objekten zählen. In anderen Sprachen kann man ein oder mehrere Objekte einfach als implizit gegeben annehmen und diesen die freischwebenden Variablen eines Programms als Eigenschaften zurechnen. Dazu zählen insbesondere auch die Variablen, die die Ein- und Ausgaben eines Programms darstellen, sofern diese nicht selbst Objekte 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, *auswerten*. 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 gar nicht als Werte in Erscheinung treten, sondern ausnahmslos als Objekte interpretiert werden (die *Referenzsemantik* objektorientierter Programmiersprachen; s. Abschnitt 7.6), ist die Sprache von einem Objekt als Wert oder von der Auswertung zu einem Objekt entschuldbar.

5.2 Veränderlichkeit von Objekten

Universalien sind ihrer Natur nach unveränderlich: Die Farbe Blau ist immer dieselbe, alle ihre Eigenschaften (sofern man ihr denn welche zuschreiben möchte) sind unveränderlich und machen sie aus (wenn sich eine Eigenschaft änderte, wäre sie nicht mehr dieselbe Farbe). Dasselbe hatten wir ja in Abschnitt 2.1 auch schon für Werte konstatiert: Aus einer Zwei kann keine Drei werden. Für Individuen gilt das naturgemäß nicht: Eine Blüte beispielsweise ist nicht dauernd blau, aber der Wechsel der Farbe macht aus der Blüte kein neues Individuum — vielmehr wechselt die Blüte ihr Erscheinungsbild oder, technischer, ihren Zustand. Da in der objektorientierten Programmierung Objekte Individuen entsprechen, gilt dasselbe auch für Objekte.

Mutables und Immutables

Objekte unterscheiden sich also grundsätzlich auch darin von Werten, dass sie **veränderlich** sind (vgl. dazu insbesondere die Ausführungen zur *Unveränderlichkeit von Records* in Abschnitt 2.1.6). Das wird in der objekt-funktionalen Programmierung jedoch genauso grundsätzlich in Abrede gestellt (wir erinnern uns, dass in der funktionalen Programmierung ja schon Variablen unveränderlich sind) und so unterscheidet man auch

in der objekt-funktionalen Programmierung zwischen **veränderlichen Objekten** (engl. „mutable objects“ oder kurz **Mutables**) und **unveränderlichen Objekten** (engl. „immutable objects“ oder kurz **Immutables**).

5.3 Gleichheit und Identität

Um noch ein wenig bei der Metaphysik zu bleiben: Der Gedanke, dass identische Dinge sich in allen Eigenschaften gleichen (die Grundlage der Substitution; s. Abschnitt 3.3.2), leitet sich (als Umkehrung) daraus ab, dass Dinge, die sich in allen Eingeschalten gleichen, die also nichts an sich haben, anhand dessen man sie unterscheiden könnte, identisch (also nicht mehrere, sondern eins) sein müssen. Dieses von LEIBNIZ formuliert und als *Identitas indiscernibilium* bekannte Prinzip, das Identität definiert, kann man in einem mathematischen Kontext, in dem Zeit keine besondere Rolle spielt, durchaus so stehen lassen. Nicht nur in der realen Welt, sondern auch in Programmen, die sie abbilden, können sich zwei Individuen oder Objekte aber auch nur vorübergehend vollkommen gleichen, ohne dass sie deswegen zu einem 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).[59] 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 bzgl. ihrer Eigenschaften (auch als *qualitative Identität* bezeichnet) nicht bedeutet, dass sich auch ihre Referenzen (Identitäten) gleichen, es sich deswegen nur um eines handelt (das wäre dann *quantitative Identität*). Trivialerweise sind aber identische Objekte (die somit nicht mehrere, sondern nur eines sind), immer gleich (jedes Objekt gleicht sich selbst vollkommen).

Veränderlichkeit braucht Identität

Es ist übrigens die Veränderlichkeit von Dingen unauflösbar an die Identität gebunden: Wenn etwas keine Identität hat, kann es sich auch nicht ändern, denn woran sollte man sonst festmachen, dass es sich bei dem Vorher und dem Nachher — der Veränderung — um verschiedene Erscheinungen desselben Dings handeln sollte?

6 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* (die

[59] Daran hat LEIBNIZ natürlich auch gedacht. Die Antwort liegt in seiner *Monadologie*, die hier aber ins Abseits führen würde, zumal sie mit den in der funktionalen Programmierung Anwendung findenden *Monaden* nichts zu tun hat.

eben keine Objekte sind), manche darüber hinaus auch über zusammengesetzte Werte (wie z. B. die Structs von C#, die auch keine Objekte sind).

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

6.1 Anweisungen

Wie imperative Sprachen auch verfügt JavaScript über *Deklarations-* und *Kontrollflussanweisungen*.[60] 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). Dadurch werden insbesondere Wertzuweisungen, die in JavaScript Ausdrücke sind, zu Anweisungen: So ist z. B.

```
49 x = 1;                                                                    JS
```

eine solche Ausdrucksanweisung. Auch Deklarationsanweisungen, von denen wir Beispiele in den Abschnitten 6.3 (Variablen) und 6.5 (Funktionen) sehen werden, werden durch ein Semikolon abgeschlossen, sofern sie nicht mit einer geschweiften Klammer („}") enden; allerdings können in JavaScript Semikolons in der Regel 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[61] 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* (Abschnitt 2.1.1). 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

[60] Dass Deklarationen zu den Anweisungen gehören gilt nicht für alle Sprachen; in manchen Sprachen sind sie getrennte syntaktische Kategorien.

[61] Die Verwendung geschweifter Klammern sowohl für *Objektliterale* wie das aus Kapitel 4 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, wie beispielsweise in `({})`.

eines zu einer Anweisung *beförderten* Funktionsaufrufs (der imperativen Form der *Funktionsanwendung*; s. Abschnitt 2.1.5 sowie weiterführend Abschnitt 6.5).

Ein alleinstehendes Semikolon bedeutet die leere Anweisung.

6.2 Primitive Werte und ihre Datentypen

Es gibt in JAVASCRIPT derzeit sechs Arten von primitiven Werten, von denen jede durch einen String repräsentiert wird, der in der JAVASCRIPT-Dokumentation *Typ* genannt wird, den ich hier aber *Datentyp* nenne:[62]

- Zahlen wie `1` oder `2.4`, repräsentiert durch `'number'`
- beliebig große (nur durch den Speicher begrenzte) Ganzzahlen wie `0n`, repräsentiert durch `'bigint'`
- Strings wie `"abc"` oder `'string'` (paarige einfache und doppelte Anführungsstriche haben dieselbe Bedeutung und erlauben dadurch Anführungsstriche der jeweils anderen Art in Anführungsstrichen), repräsentiert durch `'string'`
- Symbole, repräsentiert durch `'symbol'`
- die Wahrheitswerte `true` und `false`, repräsentiert durch `'boolean'`,
- der Wert `undefined`, repräsentiert durch `'undefined'`.

Man beachte, dass in JAVASCRIPT (anders als beispielsweise in JAVA, SCALA und F#) Strings zu den primitiven Werten zählen. Große Zahlen und Symbole sind erst nachträglich hinzugekommen und nicht wirklich gut integriert; so würden beispielsweise die Datentypen als Werte besser durch Symbole als durch Strings repräsentiert. Auch gibt es zwar einen Datentyp für `null`, der aber offenbar (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:

```
50 > typeof null                                                    JS
   'object'
```

Laufzeittyp-information und -fehler

Datentypen haben in JAVASCRIPT keine tiefere Bedeutung. Sie werden vornehmlich — als **Laufzeittypinformation** — im Programmtext für Fallunterscheidungen mittels `typeof` herangezogen und vom Laufzeitsystem genutzt, Werte von einem vorliegenden automatisch in einen anderen, in einem gegebenen Kontext benötigten Datentyp umzuformen. Auch werden Datentypen verwendet, um undefinierte

[62] Die Typen JAVASCRIPTs sind eigentlichen *Basistypen*, wie wir sie in Kapitel 26 kennenlernen werden; in Programmiersprachen nennt man solche Typen auch *primitive Typen*.

Operationen zur Laufzeit zu entdecken, wie beispielsweise den Aufruf einer Zahl als Funktion wie in

```
51 > 42()                                                                    JS
Uncaught TypeError: 42 is not a function
```

(wobei hier die Klammern als *Aufrufoperator* dienen): Hier meldet JAVASCRIPT einen **Laufzeittypfehler**. Würden solche Prüfungen nicht durchgeführt, wäre unklar, was ein Programm in solchen Fällen machen soll; so bricht es einfach ab. Sprachen ohne Laufzeittypinformation (wie beispielsweise der Lambda-Kalkül) kennen ebenfalls Situationen, die Laufzeittypfehlern entsprechen: So ist beispielsweise $x\ y$ zwar eine Funktionsanwendung, x aber keine Funktion, so dass die Reduktion hier stehenbleibt, bevor ein Wert berechnet werden konnte (s. Abschnitt 3.4.2).

dynamisch getypte Sprache

Sprachen, die wie JAVASCRIPT Laufzeittypinformation nutzen, um Typfehler während der Programmausführung zu korrigieren oder zu melden, werden manchmal auch als **dynamisch getypte Sprachen** bezeichnet. Allerdings sind die Datentypen JAVASCRIPTs zu schwach, um alle dynamischen Typprüfungen zu begründen, die JAVASCRIPT tatsächlich vornimmt (wie obiges Beispiel vom Aufruf einer Zahl als Funktion bereits zeigt). In diesem Text wird uns dynamische Typisierung nicht weiter interessieren; dafür werden wir uns mit *statischer Typisierung* ausführlich befassen (in Lektion 4).

kein eigener Datentyp für Integer

JAVASCRIPT unterscheidet bei den Zahlen des Datentyps `'number'` nicht grundsätzlich zwischen ganzen und Fließkommazahlen; 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 Fließkommarechnungen typischen Rundungsfehler einschleichen. Da JAVASCRIPT aber ganze Zahlen bis zur Größe von $2^{53} - 1$ auch als solche 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.

6.2.1 Literale Repräsentationen primitiver Werte

Primitive Werte sind der Anfang aller Daten und müssen daher direkt in einem Programmtext stehen können. Mit Ausnahme von Symbolen haben sie in JAVASCRIPT jeweils eine eigene Syntax, unter Verwendung derer sie als Zeichenfolge (also „buchstäblich“ oder literal) aufgeschrieben werden können. Man nennt solche Zeichenfolgen 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. *Template-Literalen* auch Stringliterale mit eingebetteten Ausdrücken, die zum Zeitpunkt ihrer Verwendung im Programmablauf jedes Mal neu ausgewertet werden (ein Vorgang, der auch als *Stringinterpolation* bekannt ist) und damit jedes Mal unterschiedliche Wert liefern können. Insofern stehen Template-Literale im Gegensatz zu normalen (oder echten) Literalen, die immer für denselben Wert stehen (eine andere Ausnahme sind die Objektliterale; s. Abschnitt 7.1).

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; Strings gleicher Länge können daher unterschiedlich viel Speicherplatz belegen. Dies kann beim Debuggen gut zu wissen sein (z. B. wenn Strings an Byte arrays mit fester Länge übergeben werden).

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 7.5), die aber in neueren Versionen von JAVASCRIPT zumindest nicht mehr schreibbar (`writable`; s. Abschnitt 7.4) ist.

6.2.2 Ausdrücke über primitive Werte

Wie in anderen Sprachen auch können über Literale Ausdrücke gebildet werden, so z. B. durch Operatoren wie `+`, `-`, `*` und durch `/`. 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

```
52 > '1' + 0                                                    JS
   '10'
```

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

```
53 > undefined + 1                                              JS
   NaN
```

und

```
54 > null + 1                                                   JS
   1
```

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

Bequemlichkeit vor Sicherheit

Allgemein führt die implizite Umwandlung von Argumenten in einen von einer Operation erwartet Datentyp wie in den obigen Beispielen dazu, dass bei der Ausführung der Operation kein Laufzeitfehler auftritt. Ein solcher Laufzeitfehler wird

in Sprachen mit statischer Typprüfung durch einen entsprechenden *Typfehler* zur Übersetzungszeit ebenfalls verhindert. Das implizite Umwandeln, das konzilianter und damit dem Programmierer genehmer erscheinen mag, basiert jedoch auf einer impliziten Annahme, die im Einzelfall auch falsch sein kann: nämlich, dass das Umwandeln 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 die implizite Umwandlung 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

```
55 > 0 && ''                                                    JS
   0
56 > '' && 0
   ''
```

sowie umgekehrt

```
57 > 0 || ''                                                    JS
   ''
58 > '' || 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.

6.2.3 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. In JAVASCRIPT gibt es gleich zwei Gleichheitsoperatoren: `==` und `===` (`=` steht in JAVASCRIPT, wie in vielen auf der C-Syntax basierenden Sprachen, so auch JAVA und C#, für die *Wertzuweisung*). Damit haben wir

```
59 > 1 == 1                                                     JS
   true
```

und

```
60 > 1 === 1                                                                    JS
   true
```

Im Kontext von Werten unterscheiden sich `==` und `===` darin, dass `==` die Operanden vor ihrem Vergleich bzgl. ihres Datentyps angleicht, so dass also etwa

```
61 > 1 == "1"                                                                   JS
   true
```

ergibt. Im Gegensatz dazu steht

```
62 > 1 === "1"                                                                  JS
   false
```

weil hier der Datentyp der linken und der rechten Seite nicht übereinstimmen. In Abschnitt 7.7 werden wir sehen, wie die Gleichheit von Objekten geregelt ist.

6.3 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, der Mathematik und des Lambda-Kalküls.

Wie in Scriptsprachen nicht unüblich 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*.[63] 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, in dem die Verwendung nicht zuvor deklarierter Variablen zurückgewiesen wird. Dies ist in der REPL von NODE.JS Standard und so erhält man dort

```
63 > x                                                                          JS
   Uncaught ReferenceError: x is not defined
```

[63] Je nach verwendeter Version von JAVASCRIPT handelt es sich dabei tatsächlich um Felder eines *globalen Objekts* (s. Abschnitt 7.5).

Variablen-deklarationen

Explizite Variablendeklarationen erfolgen in JAVASCRIPT per Var- oder per Let-Anweisung, wobei in beiden Fällen der Variable bei ihrer Deklaration gleich ein Wert zugewiesen werden kann:

```
64 var x = 1                                                    JS
65 let y = 'a'
```

Bleibt die Initialisierung aus, erhält die Variable automatisch 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`):

```
66 > var x = x                                                  JS
67 > x
   undefined
68 > let y = y
   Uncaught ReferenceError: Cannot access 'y' before initialization
```

Die Deklaration mit `var` verhält sich also in gewisser Weise wie letrec aus dem Lambda-Kalkül (Abschnitt 3.4.6), allerdings aus einem anderen Grund, wie gleich ersichtlich wird.

Geltungsbereich von Variablen

Allen Variablen von JAVASCRIPT ist ein *Geltungsbereich* 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

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

auf den die Deklaration enthaltenden Block beschränkt; stünde dort stattdessen

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

und ist `x` nicht in einem umgebenden Block oder einer umgebenden Funktion 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 6.5) und können darin auch vor ihrer Deklaration verwendet werden: Var-Deklarationen (jedoch nicht damit verbundene explizite Initialisierungen!) werden nämlich an den Anfang ihres Geltungsbereichs hochgezogen. Dieses sog. *Hoisting* ermöglicht auch die obige selbstbezügliche explizite Initialisierung, bei der `x` aufgrund der hochgezogenen Deklaration schon den Wert `undefined` hat. Mit `var` deklarierte Variablen dürfen innerhalb einer Funktion auch mehrfach (oder re-) deklariert werden, weswegen sie sich zum Experimentieren in einer REPL besonders eignen.

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 muss (als Initialisierung) innerhalb der Deklaration erfolgen; alle anderen (weiteren) Zuweisungen an die *Konstante* sind ausgeschlossen:

```
71 > const c                                                              JS
   Uncaught SyntaxError: Missing initializer in const declaration
72 > const c = 1
73 > c = 2
   Uncaught TypeError: Assignment to constant variable.
```

`const` bevorzugt; Grenzen der Konstanz

Bei Pflege eines funktionalen Stils sollte Konstanten 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. Sind Objekte die Werte von Konstanten, dann kann sich der Wert der Variable insofern ändern, als das Objekt seinen Zustand ändern kann – das wird durch die Verwendung von `const` nämlich nicht ausgeschlossen.

Lebensdauer von Variablen

Die *Lebensdauer* einer Variable hängt in JAVASCRIPT nicht strikt an ihrem Geltungsbereich, also dem Block bzw. der Funktion, innerhalb derer sie deklariert wurde (vgl. dazu Abschnitt 2.2.1): Sie kann wie im Lambda-Kalkül durch *Funktionsabschlüsse*, oder *Closures* (Abschnitt 3.4.4), verlängert werden. Mehr dazu in Abschnitt 10.3.

6.4 Arrays

Wir hatten in Abschnitt 4.2 schon gesehen, dass mehrere Werte oder Objekte zu einem Array zusammengefasst werden können:

```
74 var blaubär = {                                                        JS
75   enkel: [ { name: 'Gelb' }, { name: 'Grün' }, { name: 'Rosa' } ]
76 }
```

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

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

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

```
78 for (const e of blaubär.enkel) {                                       JS
79   console.log(e.name)
80 }
```

Neben dieser sog. *externen Iteration* über die Elemente eines Arrays bietet JAVASCRIPT auch *interne Iterationen*. Diese basieren jedoch auf den funktionalen Anteilen von JAVASCRIPT und werden deswegen erst in Abschnitt 10.5 vorgestellt.

In JavaScript sind Arrays als Objekte implementiert; genaueres dazu in Abschnitt 7.2.

6.5 Funktionen und Prozeduren

Funktionsdeklarationen

Auch wenn, wie wir bereits in Abschnitt 4.1 gesehen haben, Funktionen in JavaScript Werte von Eigenschaften sein können, so lassen sie sich doch wie in imperativen Sprachen als eigenständige Programmelemente (*Unterprogramme*; s. Abschnitt 2.1.3) deklarieren:

```
81 function id(x) { return x }                                                JS
```

Eine solche **Funktionsdeklaration** beinhaltet neben der Vereinbarung ihres Namens und ihrer Parameter auch die *Definition* der Funktion in Form einer Berechnungsvorschrift (ihr Rumpf). Die Funktion wird mit ihrer Deklaration im umgebenden *Geltungsbereich* sicht- und unter ihrem Namen aufrufbar. Außerhalb der REPL bedeutet dies, dass die Funktion auch an Stellen im Programm aufgerufen werden kann, die vor ihrer Deklaration liegen; dies liegt daran, dass Funktionsdeklarationen, genau wie mit `var` deklarierte Variablen, an den Anfang ihres Gültigkeitsbereichs hochgezogen werden (das sog. *Hoisting*).

geschachtelte Funktionen

Funktionsdeklarationen können auch ineinander geschachtelt werden, wobei aber die Gültigkeit einer **inneren Funktion** auf den Rumpf der äußeren beschränkt ist:

```
82 > function äußere() {                                                      JS
83 .   innere()
84 .   function innere() {
85 .     console.log("Hallo")
86 .   }
87 . }
88 > äußere()
   Hallo
89 > innere()
   Uncaught ReferenceError: innere is not defined
```

Namenszwang

Da es nicht sinnvoll ist, anonyme Funktionen zu deklarieren, müssen Funktionsdeklarationen einen Namen angeben:

```
90 > function() {}                                                            JS
   Uncaught SyntaxError: Function statements require a function name
```

Tatsächlich wird mit der Deklaration einer Funktionen, analog zu einer Var-Deklaration, eine Variable eingeführt, deren Wert die Funktion ist. Mehr dazu in Abschnitt 10.1; hier noch kurz der Nachweis, dass es sich bei einer Deklaration in JavaScript tatsächlich um eine Anweisung handelt:

```
91 if (Math.random() > 0.5) {                                                 JS
92   function f() { return 1 }
93 } else {
```

```
94   function f() { return 2 }
95 }
```

ist gültiges JavaScript. Eine höchst dynamische Sprache!

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 6.3); 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. In beiden Fällen haben Zuweisungen innerhalb der Funktion unmittelbare Auswirkungen auch außerhalb der Funktion (ein sog. *Nebeneffekt*). Zudem behalten die Variablen, anders als die lokalen Variablen der Funktion, zwischen zwei Funktionsaufrufen ihren Wert; für den Fall, dass dies gewünscht sein sollte (es gibt durchaus gute Anwendungen dafür), existiert allerdings eine bessere Möglichkeit, die wir in Abschnitt 11.1 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 nicht nur die Wertzuweisung an globale Variablen, sondern auch die Änderung an Objekten, die als Argumente übergeben wurden (s. Abschnitt 7.6).

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

```
96 > id(0) + 1                                                    JS
   1
```

ein Ausdruck, der den Aufruf der Funktion `id` enthält. Stünde der Aufruf hingegen für sich wie etwa in

```
97 id(4);                                                         JS
```

dann handelt es sich um eine Anweisung (eine *Ausdrucksanweisung*)[64], 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,

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

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.[65]

nur beschränkte partielle Anwendung

Man beachte, dass JAVASCRIPT, anders als beispielsweise SCALA, keine allgemeine *partielle Anwendung* von Funktionen ermöglicht. Will man jedoch eine mehrstellige Funktion nur mit den ersten einer Reihe von Argumenten aufrufen und damit eine auf die angegebenen Argumente festgelegte Funktion zum Ergebnis erhalten, kann man die Funktion entweder curryen (s. Abschnitte 3.3.9 und 10.4) oder die ersten Parameter durch den Aufruf von `bind` auf der Funktion mit Argumenten versorgen, wodurch man dann eine neue Funktion zurückerhält, in der die ersten Parameter auf die übergebenen Argumente festgelegt sind.

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:

```JS
function inkrement(x, schritt = 1) {
  return x + schritt
}
```

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

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 `'object'`.

7.1 Objektliterale

Ähnlich wie die Literale von primitiven Werten stellen **Objektliterale** Objekte in einem Programmtext dar. Anders als Werte sind Objekte jedoch änderbar, was zusammen mit dem Umstand, dass Objekte eine Identität haben, bedeutet, dass ein Literal ein Objekt nur zu einem bestimmten Zeitpunkt repräsentiert. Genau wie bei den *Template-Literalen* für Strings ist dies der Zeitpunkt im Programmablauf, zu dem das Objektliteral verwendet und dabei (wie ein Template-Literal) ausgewertet wird. Dies passt eigentlich nicht zur Idee eines Literals, das ja

[65] 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.

im Programmtext, also buchstäblich und somit zeitunabhängig oder statisch, für etwas steht, aber so ist es möglich, dass das gleiche Literal mehrere Objekte erzeugt, nämlich wenn es wiederholt ausgewertet wird (mehr dazu in Kapitel 8).

Variablen in Literalen

Der Umstand, dass Objektliterale erst zur Laufzeit ausgewertet werden, erlaubt es, in literalen Definitionen von Objekten Variablen zu verwenden. So lässt sich statt

```JS
{
  name: 'Blöd',
  alter: 42
}
```

auch

```JS
var name = 'Blöd'
var alter = 42
{
  name: name,
  alter: alter
}
```

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

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

vorgesehen ist. Anstelle von Variablen als Werte eines Name/Wert-Paars 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 in Name/Wert-Paaren gleich mehr).

Linearisierung von geschachtelten Objekten

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

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

ist gleichbedeutend mit

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

So gelingt dann auch das bereits in Abschnitt 4.2 in Aussicht gestellte Herstellen von Beziehungen mehrerer Objekte zu einem:

```
127 var kbb = { name: 'Blaubär' }                                    JS
128 var hein = { chef: kbb }
129 var blöd = { chef: kbb }
```

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

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

initialisiert `selbst` mit einem Objekt, dessen Eigenschaft `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, wie in Abschnitt 7.2 dargelegt werden wird.

berechnete Namen von Eigenschaften

Variablen dürfen in Objektliteralen auch anstelle der Namen von Eigenschaften stehen:

```
131 var eigenschaft = 'name'                                         JS
132 {
133   [eigenschaft]: 'Blöd',
134   alter: 42
135 }
```

ist auch gültiges JavaScript (wobei hier anstelle von `eigenschaft` im Objektliteral auch jeder andere Ausdruck stehen kann, dessen Ergebnis sich in einen String umwandeln lässt). Die eckigen Klammern zeigen hierbei an, dass `eigenschaft` nicht literal als Eigenschaftsname verwendet werden soll, sondern ein Ausdruck ist, dessen Auswertung erst den Namen ergibt. Die eckigen Klammern sehen wir gleich noch einmal wieder.

Eigenschaftsnamen sind Strings

Als Namen von Eigenschaften 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 wie oben in eckige Klammern gesetzt werden. Tatsächlich werden auch Bezeichner als Namen von Eigenschaften von JavaScript in Strings (mittlerweile: in Symbole) konvertiert, denn anders als Variablen oder in statischen Sprachen die Felder eines Records werden Eigenschaftsnamen erst zur Laufzeit eines Programms mit Speicherzellen verbunden, in der Bezeichner nicht mehr existieren.

mehrere gleichnamige Eigenschaften eines Objekts

Aus der Auswertung von Objektliteralen zur Laufzeit und insbesondere aus dem Umstand, dass die Namen der Eigenschaften selbst durch Ausdrücke vertreten werden dürfen, folgt, dass bei der Definition eines Objekts mehrere gleichnamige Eigenschaften vorkommen dürfen. Dabei bildet nur der letzte angeführte Wert das Name/Wert-Paar, das das Objekt beschreibt — vorherige Zuordnungen werden 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

In der objektorientierten Programmierung 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“).

Feldzugriff

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

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

und die **Klammerschreibweise**

```
137 > {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

```
138 { f1: { f2: 1 } }.f1.f2                                       JS
139 { f1: { f2: 1 } }['f1']['f2']
140 { f1: { f2: 1 } }.f1['f2']
141 { 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

```
142 > var feld = 'f'                                              JS
143 > { 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 solchen Schlüssel/Wert-Paaren allerdings auf Strings und Symbole beschränkt (deswegen „*Name*/Wert-Paare“).

Arrays als spezielle Objekte

In JAVASCRIPT sind auch 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 (mit Zahlen als Namen) wird nur darübergelegt.[66] 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 Objekten 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.

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

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

als auch mittels der Klammerschreibweise

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

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

Änderung von Objekten

Man beachte, dass die Zuweisung an ein Feld eines Objekts, anders als bei den Records als Werten der imperativen Programmierung (Abschnitt 2.1.6), kein neues Objekt erzeugt — hier wird tatsächlich das bestehende Objekt, zu dem das Feld gehört, verändert (man spricht dann auch von einem *Zustandswechsel* eines Objekts; s. Abschnitt 5.2). Dieser für die objektorientierte Programmierung zentrale Unterschied wird in Abschnitt 7.7 noch näher beleuchtet. Hier halten wir fest, dass sich mithilfe von Zuweisungen an Felder nun auch zirkuläre Objektstrukturen herstellen lassen:

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

`selbst` steht so über das Feld `f` mit sich selbst in Beziehung. Würde durch diese Zuweisung ein neues Objekt entstehen, würde dieses per `f` nicht auf sich selbst verweisen.

[66] Damit ist die klassische Unterscheidung von Arrays und Records zur Strukturierung von Daten (Abschnitt 2.1.6) zugunsten einer universellen Datenstruktur, dem Objekt, praktisch aufgelöst.

Zugriffe auf nicht vorhandene Felder

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

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

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

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

Felder werden nicht deklariert

Felder eines Objekts müssen also in JAVASCRIPT nicht deklariert werden. Anders als Variablen (die in JAVASCRIPT ja auch nicht deklariert werden müssen; s. Abschnitt 6.3) *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 deswegen anders.

alle Objekte haben eine variable Struktur

Sieht man einmal vom 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 insbesondere 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. Nur insofern haben alle Objekte die gleiche Struktur, nämlich die eines Dictionaries.

ungültige Feldzugriffe

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

```
151 > var o                                                                   JS
152 > 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:

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

Hierbei scheint `null` bzw. `undefined` durch einen Feldzugriff „hindurchpropagiert“ zu werden[67]; tatsächlich wird der Feldzugriff aber abgebrochen und der Wert von `o` zurückgegeben. Soll hin-

[67] vgl. dazu Fußnote 18

gegen der lesende Zugriff auf ein nicht vorhandenes Feld eines Objekts `o` nicht `undefined`, sondern einen Ersatzwert `w` liefern, dann kann man

```
154 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`, was die in Abschnitt 6.2.2 gescholtene Interpretation von `null` und `undefined` als falsy und die dazu passenden Definitionen der Operatoren `&&` und `||` ausnutzt. Wie man aber an der (vor diesem Hintergrund weitgehend redundanten und deswegen technisch nicht nötigen) Einführung von `?.` und `??` sieht, ist es sinnvoll (und besser), für verschiedene Aufgaben verschiedene Operatoren vorzusehen (vgl. dazu die Anmerkungen in Abschnitt 3.3.10).

7.3 Methoden

Wie bereits in Abschnitt 4.1 angedeutet können nicht nur primitive Werte oder Objekte als Eigenschaften eines Objekts auftreten, sondern auch Funktionen.[68] Man nennt die auf diese Weise 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 objektorientierten Programmierung ihren Ruf als eigenständiges *Paradigma* eingebracht hat.

7.3.1 Methodendefinitionen

Alternativ zu der in Abschnitt 4.1 verwendeten Darstellung von Methoden, die die Parallelität zu Feldern auch syntaktisch unterstreicht, lässt sich in JavaScript auch die sog. *Methodensyntax* verwenden:

```
155 {                                                            JS
156   vorname: 'Hein', nachname: 'Blöd', alter: 42,
157   hallo() { console.log('Käpt´n, Käpt´n!') },
158   name() { return this.vorname + ' ' + this.nachname }
159 }
```

[68] „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 10.

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 selbst so anonym; zugreifbar ist sie nur als Eigenschaft des Objekts, so dass ihr Aufruf aus dem Rumpf derselben oder einer anderen Methode desselben Objekts heraus (wie auch der ihrer Felder) mit `this` qualifiziert werden muss. Sollte die Funktion sich selbst und nur unter Nennung ihres Namens rekursiv aufrufen können sollen, dann muss dafür die Feldsyntax verwendet und die Funktion zusätzlich benannt werden (s. dazu Abschnitt 10.1).

Stringausdrücke als Methodennamen

Für die Benennung von Methoden gelten dieselben Bedingungen wie für Felder. 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:

```
160 { …                                                                    JS
161   ['name']() { return this.vorname + ' ' + this.nachname }
162 }
```

beispielsweise ist zu obigem Beispiel äquivalent.

Parameterzahl und Überladen

Analog zu Funktionen (Abschnitt 6.5) 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 besitzen.

`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 global deklarierten Funktionen, 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. Da dasselbe auch für *globale Variablen*, die nicht mit `let` oder `var` deklariert worden sind, gilt, verlässt JAVASCRIPT selbst mit seinen globalen Funktionen und Variablen nicht die objektorientierte Welt. Das ist eine durchaus bemerkenswerte Eigenheit der Sprache.

7.3.2 Methodenaufruf und dynamisches Binden

Die Methoden eines Objekts werden auf die gleiche Weise wie seine Felder selektiert: mittels der Punkt- oder mittels der Klammerschreibweise. So liefern `o.m` und `o['m']` beide die mit `'m'` benannte Funktion des Objekts `o` (so vorhanden). Um die Methode auch aufzurufen, müssen dem Namen runde Klammern hintangestellt werden, die dann auch die Argumente des Metho-

denaufrufs aufnehmen, also etwa `o.m()` bzw. `o['m']()`. Bezüglich der Angabe der Argumente gilt das gleiche wie für Funktionsaufrufe; s. hierfür Abschnitt 6.5.

dynamische Bindung

Vergleicht man den Methodenaufruf `o.m(...)` mit dem Funktionsaufruf `f(...)`, dann fällt auf, dass der Name der Methode mit `o` qualifiziert ist. Tatsächlich hängt beim Methodenaufruf, 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 (oder des Ausdrucks) `o` zwischen den Aufrufen ändert (sie bzw. er 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).

verdeckte Fallunterscheidung

Bei der dynamischen Bindung handelt es sich um eine *verdeckte Fallunterscheidung*: Der Programmablauf verzweigt abhängig vom Wert von `o` zu verschiedenen Implementierungen einer Funktion `f`. Dass diese Form der Fallunterscheidung sogar die einzige ist, die man in der objektorientierten Programmierung braucht, hat die Sprache SMALLTALK eindrucksvoll gezeigt: Hier ist sogar If-then-else keine native Kontrollstruktur, sondern über dynamisches Binden von Funktionen höherer Ordnung auf den Objekten `true` und `false` realisiert (wir werden in Abschnitt 16.3.2 am Beispiel von RUBY nachvollziehen, wie das geht). Dass man die Fallunterscheidung nicht unmittelbar als solche erkennt und auch nicht sieht, wohin genau verzweigt wird (man sieht ja `o` das Objekt, für das es steht, nicht an), hat der objektorientierten Programmierung in manchen Kreisen einen schlechten Ruf eingebracht (mehr dazu gleich) — auf der anderen Seite ist sie die Basis zahlreicher nützlicher Entwurfsmuster, die der objektorientierten Programmierung zu ihrer weiten Verbreitung verholfen haben.

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 Beispiel aus Abschnitt 7.3.1 der Ausdruck `this.vorname` auf das Feld des Objekts `o`, auf dem die Methode mittels `o.name()` 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 veränderliche Struktur, wie sie die Objekte JAVASCRIPTs darstellen, muss dynamisch, also zum Zeitpunkt 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 vor allem 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 9 zuwenden werden, sind für objektorientierte Programmiersprachen wesentlich; das dynamische Binden ist eine technische Voraussetzung dafür, dass sie die beabsichtigte Wirkung entfalten können. *Die dynamische Bindung ist somit ein weiteres Wesensmerkmal der objektorientierten Programmierung.*

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 oder zumindest sein sollten 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 (die oben bereits erwähnte verdeckte Fallunterscheidung); das macht das Lesen und Debuggen eines Programms schwieriger. Deswegen verlangen manche 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++, mittels Schlüsselwörtern wie `virtual` oder `dynamic`); 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.

7.3.3 Datenkapselung mit Queries, Commands und Zugriffsmethoden

Eine der ursprünglichen Ideen der Objektorientierung ist, dass sämtliche Daten, die einem Objekt über seine Felder als Eigenschaften zugeordnet sind, 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 ihn ändern. Man nennt dies auch **Datenkapselung**. Der Gedanke dahinter ist, dass sich die *interne Repräsentation* eines Objekts, soweit sie durch seine Felder bestimmt wird, im Laufe der Entwicklung eines Programms ändern kann (die berühmt-berüchtigte Änderung einer Entwurfsentscheidung, vor der niemand gefeit ist). Die Methoden eines Objekts nimmt man hingegen als relativ stabil an.

Zugriffsmethoden

Nun hatten wir ja in Abschnitt 4.3 gesehen, dass durchaus auch Daten (wie Name oder Alter) zur Oberfläche eines Objekts gehören, es also nach außen repräsentieren können. Dies ist insbesondere dann der Fall, wenn durch Felder Eigenschaften oder Beziehun-

gen 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

```
163 var o = {                                                    JS
164   name: '',
165   holeName() { return this.name },
166   setzeName(name) { this.name = name }
167 }
```

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

```
168 var o = {                                                    JS
169   _name: '',
170   get name() {return this._name},
171   set name(name) {this._name = name}
172 }
```

schreiben, was 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. Dies geht aber trotzdem: `o._name` hat dasselbe Ergebnis wie `o.name` und `o._name = 'abc'` denselben Effekt wir `o.name = 'abc'`.[69] Damit Getter und Setter sich nicht rekursiv aufrufen, 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. Bleibt jedoch beides aus, liefern also die Zugriffsmethoden genau den lesenden und schreibenden Zugriff auf die Felder, die sie kapseln sollen, dann bleibt insbesondere bei solchen Feldern, die man eigentlich zur Oberfläche eines Objekts rechnen würde, immer der fade Beigeschmack einer bloßen Pflichterfüllung ohne tieferen Sinn. Noch fragwürdiger ist es, wenn die interne Repräsentation eines Objekts über Zugriffsmethoden nach außen sichtbar gemacht wird, nur weil dies für bestimmte Funktionen eines System benötigt wird (wie z. B. das Prüfen auf Gleichheit von zwei Objekten, wie es in Abschnitt 21.4 thematisiert werden wird). Insofern ist es auch keine wirkliche Errungenschaft, wenn Programmierwerkzeuge oder

[69] Während andere Sprachen zur besseren Abschirmung von Feldern *Zugriffsmodifikatoren* wie `private` bieten, gibt es das in JAVASCRIPT nicht (wohl aber in TYPESCRIPT; s. Kapitel 30).

gar Programmiersprachen selbst die automatische Bereitstellung von Zugriffsmethoden anbieten — man sollte sich vor einer Nutzung dieses Dienstes immer fragen, ob man nicht genauso gut die Felder direkt zugreifbar machen kann.

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. Letzteres entspricht in der funktionalen Programmierung einem *Nebeneffekt*, ist in der objektorientierten Programmierung aber der Zweck (oder „Haupteffekt“) des Setters; als Wert der Funktion wird hier, je nach Implementierung, entweder das Empfängerobjekt oder der zugewiesene Wert oder nichts (`undefined`) zurückgegeben.

Die funktionale Sicht auf Felder wird im Kontext von Typen und deren Varianz in Lektion 4 noch eine erklärende Rolle spielen (in Kapitel 32).

7.4 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.[70] 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, 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.

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

Nun hat ein Programm schnell so viele Namen, dass gut lesbare knapp werden. Daraus ergibt sich eine Namenskonkurrenz: Zwei Dinge sollen gleich benannt werden, können es aber nicht,

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

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“ beispielsweise).

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, in dem „a“ der Qualifizierer ist. 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.[71] 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, die ebenfalls als Qualifizierer der Namen von Eigenschaften dient, 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 Funktionen Eigenschaften des globalen Objekts[72]). Dieses globale Objekt, das mittels seiner

[71] 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.

[72] In Abschnitt 6.3 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;

Eigenschaft `globalThis` auf sich selbst verweist, heißt in der Browser-Version von JAVASCRIPT per Alias 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` und bietet 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 8.4 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 19.2 und 39.1.4, wo es um die Fassung von Modulen in objektorientierten Termini geht — nicht weiter behandelt.

7.6 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*. In erster Näherung enthalten Variablen, die für Werte stehen, diese Werte oder haben sie zum Wert (doch s. Abschnitt 2.1.2, warum dies in dynamischen Sprachen nicht gilt); 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[73], werden zur Abgrenzung von (primitiven) Werten gelegentlich auch **Referenzwerte** (engl. „reference values“) genannt. Da das Wort „Referenzwert“ umgangssprachlich jedoch etwas anderes bedeutet, spricht man in der Regel besser von Referenzen.

Ausdrücke, die für Objekte stehen, liefern Referenzen

In Programmen stehen nicht nur Variablen, sondern allgemein Ausdrücke (zu denen Variablen ja zählen; s. Abschnitt 6.3) für Werte. Entsprechend unterscheiden wir Ausdrücke, die zu einem primitiven Wert auswerten (wie beispielsweise die Addition von zwei Zahlen) und die genau 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

man kann dies durchaus als Abkehr von JAVASCRIPTs selbstbezüglichem, minimalen Sprachdesign auffassen, das sich für größere Aufgaben so nicht bewährt hat.

[73] Zeiger werden gelegentlich als das technische Konstrukt (eine Speicheradresse) gesehen und Referenzen als ihr logisches Pendant.

(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, 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.

Bedeutung der Referenzsemantik für Wertzuweisungen: Aliasing

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. So enthalten nach

```
173 var x1 = 1                                                    JS
174 var x2 = x1
```

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

```
175 var o1 = {}                                                   JS
176 var o2 = o1
```

`o1` und `o2` jedoch scheinbar ein und dasselbe Objekt (tatsächlich jeweils eine Kopie einer Referenz auf das bei der ersten Zuweisung angelegte Objekt, aber das ist ja wegen der Referenzsemantik nicht sichtbar). Die Variable `o2` ist also nur ein weiterer Zugang (Name) für das Objekt, das bereits durch `o1` zugänglich (benannt) ist — es handelt sich bei `o2` um einen *Alias*.

Sichtbarkeit von Aliasing bei Änderungen

Dass es sich bei `o1` und `o2` tatsächlich nur um verschiedene Namen (Variablen; vgl. dazu die Anmerkungen zu Benennung in Abschnitt 2.1.2) für das-

selbe Objekt handelt, macht sich immer dann bemerkbar, wenn man über einen der beiden Namen das Objekt, für das er steht, ändert:

```
177 > o1.f = 1                                                    JS
178 > o2
    { f: 1 }
```

Die Änderung sieht man auch am durch die Zuweisung vermeintlich kopierten Objekt! Tatsächlich wurde bei der Zuweisung in Zeile 176 aber eben nur die Referenz kopiert.

Unterschied nur bei veränderlichen Objekten

Voraussetzung für diese manchmal als übel (weil ungewollt) erachtete Auswirkung des Aliasing ist allerdings, dass das, wofür ein Alias steht, überhaupt veränderlichen ist. Ist es das nämlich nicht (so wie das bei Werten der Fall ist), dann wird das Aliasing in einem Programm niemals sichtbar und kann ignoriert werden (sieht man einmal von Effizienzüberlegungen ab — es ist für gewöhnlich billiger, einen Zeiger zu kopieren als eine ganze Datenstruktur). Deswegen ist Aliasing in der *reinen funktionalen Programmierung*[74] auch eher kein Thema und deswegen wird auch in der objekt-funktional Programmierung immer wieder empfohlen, mit unveränderlichen Objekten (den sog. *Immutables*) zu arbeiten. Man verliert damit aber ein Charakteristikum der objektorientierten Programmierung, nämlich dass Objekte einen Zustand haben.

Bedeutung der Referenzsemantik für Parameter

Der Unterschied zwischen Wertsemantik bei Werten und Referenzsemantik bei Objekten erfährt im Kontext von Methodenaufrufen (Funktionsaufrufen, Prozeduraufrufen) eine besondere Bedeutung: Wenn der Aufruf per *Call-by-value* erfolgt, enthalten die Parameter in beiden Fällen Kopien der Argumente (Abschnitt 2.1.9). Wenn nun die aufgerufene Methode den Wert eines Parameters ändert, erhält der Parameter damit einen anderen Wert (s. Abschnitt 2.1.6, warum das so ist). Auf das Argument wirkt sich das nicht aus — das ist ja gerade das Wesen von Call-by-value. Verändert man aber ein Objekt, das in einem Parameter vorliegt, dann ändert man damit auch das Argument, obwohl das nur bei *Call-by-reference* und nicht bei Call-by-value möglich sein sollte! Dies ist besonders fatal bei rekursiven Aufrufen, bei denen man ausnutzen möchte, dass jeder Aufruf seinen eigenen Satz Parameter hat, die von den Parametern der vorherigen und nachfolgenden Aufrufe unabhängig sind. Was nützt es da, wenn diese Parameter alle auf dasselbe Objekt verweisen? Man bräuchte die Parameter gar nicht, sondern könnte einfacher (oder zumindest billiger) eine globale Variable nehmen!

Unveränderlichkeit als Weg zur vollkommenen Angleichung

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

[74] 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

die große Illusion der Objektorientierung

Unter der mit der Referenzsemantik einhergehenden impliziten Dereferenzierung verschwimmt 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. Wir werden das Thema in Kapitel 20 noch einmal ausführlicher behandeln und da auch konkrete Beispiele sehen.

7.7 Gleichheit und Identität von Objekten

Wie wir gesehen hatten, ergeben in JAVASCRIPT die Vergleiche `1 === 1` und sogar `1 == "1"` den Wert `true` (Abschnitt 6.2.3), `{} == {}` hingegen den Wert `false` (Abschnitt 4.1; genauso übrigens wie `{} === {}`, denn beim Objektvergleich gibt es in JAVASCRIPT keinen Unterschied zwischen `==` und `===`). Hätten wir das Ergebnis nicht schon in Kapitel 4 zur Kenntnis genommen, dann wären wir jetzt vielleicht überrascht: Da das Objektliteral auf der linken und der rechten Seite des Vergleichs syntaktisch absolut gleich sind, wäre nicht unbedingt zu erwarten, dass die dazugehörigen Objekte als ungleich gewertet werden. Das liegt zum einen daran, dass die beiden Objektliterale bei ihrer Auswertung jeweils ein neues Objekt erzeugen und eine Referenz darauf zurückgeben, und zum anderen daran, dass mit `==` und `===` die Referenzen verglichen werden und nicht die Objekte, die dahinter stehen. Mit der Erzeugung zweier Objekte werden auch zwei verschiedene Referenzen verglichen, die als Werte eben nicht gleich sind. Referenzen werden beim Test auf Gleichheit 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*.

Unterschied zwischen Gleichheit und Identität; Identität eines Objekts

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 5), 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. Wenn allerdings zwei Ausdrücke zu denselben Referenzen auswerten, werten sie zum identischen Objekt aus. Man spricht deswegen auch von der **Identität eines Objekts** und meint damit die Manifestation der Einzigkeit des Objekts — es gibt keine zwei Objekte mit gleicher Identität. Da aber die Identität eines Objekts eine abstrakte Sache ist, muss ersatzweise eine Referenz auf das Objekt als Identität eines Objekts herhalten; die Referenz reifiziert die Identität gewissermaßen, macht sie also zu einer Sache (hier zu einem

Wert). Wir haben in Kapitel 5 gesehen, warum das mehr als nur ein informatischer Kunstgriff ist.[75]

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

```
179 function gleich(obj1, obj2) {                                    JS
180   const namen1 = Object.keys(obj1)
181   const namen2 = Object.keys(obj2)
182   if (namen1.length !== namen2.length)
183     return false
184   for (const name of namen1)
185     if (obj1[name] !== obj2[name])
186       return false
187   return true
188 }
```

definieren, die über Eigenschafen iteriert und deren Werte vergleicht (`Object.keys` ist hier eine Funktion, die die Namen der Name/Wert-Paare, die ein Objekt beschreiben, liefert). Wir prüfen

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

und sind zufrieden, ebenso mit

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

und mit

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

jedoch weniger mit

```
192 > 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 definierte Gleichheit ist somit eine **flache**; eine **tiefe** würde hingegen verlangen, dass auch die Objekte, die als Eigenschaften dienen, 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 Eigen-

[75] Hier sei noch hinzugefügt, dass in relationalen Datenbanken Primärschlüssel, die ausschließlich der eindeutigen Identifikation von Datensätzen (Objekten oder Entitäten) dienen und keine aus der Domäne stammende Bedeutung haben, auch *Surrogate keys* genannt werden. Ein solcher Schlüssel ist hier, genau wie eine Referenz, ein Stellvertreter für ein Objekt (wobei sich „surrogate“ wohl eher darauf bezieht, dass der Schlüssel andere, domänenspezifische ersetzt).

schaft 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.[76]

Gleichheit und Identität unter Veränderlichkeit

Zur Verdeutlichung der Betrachtungen aus Abschnitt 5.3 zu Gleichheit und Identität unter Veränderlichkeit sehen wir uns hier noch ein konkretes Beispiel an, warum es richtig ist, bei Objekten zwischen Gleichheit und Identität strikt zu unterscheiden. Solange Objekte veränderbar sind, können sie sich vorübergehend gleichen:

```
193 > var o1 = { f: 1 }                                                    JS
194 > var o2 = { f: 1 }
195 > gleich(o1, o2)
    true
196 > o1 == o2
    false
197 > o2.f = 2
198 > o1 == o2
    false
199 > 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 das Ergebnis des obigen ersten Tests auf Identität `true` statt `false` wäre), 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 bei den nachfolgenden Tests auf Identität und Gleichheit auch `true` stünde). Wollte man das nicht, dann müsste durch jede Änderung eines Objekts ein neues entstehen (ganz so, wie das bei Werten und auch bei Objekten in der *reinen funktionalen Programmierung* der Fall wäre), das vom alten zu trennen ist (so dass der zweite Test auf Identität selbst dann `false` ergäbe, wenn der erste noch `true` ergab). Dies würde auch die Probleme mit der vorübergehenden Gleichheit von Objekten als Elementen von Mengen beheben, die Gegenstand von Abschnitt 20.3.1 sind. Man beachte jedoch, wie sehr die Begründung des Unterschieds von Gleichheit und Identität an der *Veränderlichkeit* von Objekten hängt: Wäre diese nicht gegeben, dann wäre auch die Unterscheidung von Gleichheit und Identität nicht sinnvoll (oder höchstens ein Implementierungsdetail ohne Auswirkung auf die Semantik von Programmen).

WIKIPEDIA

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 den leeren String).

[76] Wenn man beherzigt, dass primitive Werte als Werte von Eigenschaften Objekte beschreiben, Objekte als Werte von Eigenschaften hingegen Beziehungen zwischen Objekten repräsentieren, dann kann man sich auf den Standpunkt stellen, dass letztere gänzlich unberücksichtigt bleiben können, genügt doch für die Gleichheit von Objekten i. A. die Gleichheit ihrer Beschreibungen.

7.8 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 Kopie des Wertes (genauer: eine Kopie seiner Repräsentation als *Bitstring*). Wenn eine Variable eine Referenz auf ein Objekt enthält, dann gilt für die Referenz dasselbe: Sie 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 für neue Objekte 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[77], 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 die automatische *Speicherbereinigung*. Sie müssen sich also, wenn Sie JAVASCRIPT verwenden, zumindest in dieser Hinsicht um nichts kümmern.

8 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

```
200 > var objekte = []                                                    JS
201 > for (const i of [1, 2]) objekte[i] = {}
202 > objekte[1] == objekte[2]
    false
```

[77] Solche Registraturen verwenden allerdings in der Regel sog. *Weak references*, die der Beseitigung eines Objekts nicht im Wege stehen.

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 neue Objekte durch Funktionen erzeugen zu lassen.

8.1 Fabrikfunktionen

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

```
203 function neuePerson(name, alter) {                                    JS
204   return { name, alter }
205 }
```

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

```
206 > neuePerson('Hein', 35)                                              JS
    { name: 'Hein', alter: 35 }
```

sowie

```
207 > neuePerson('Hein', 35) == neuePerson('Hein', 35)                    JS
    false
```

überzeugen kann.

Man beachte, dass die durch den Aufruf von `neuePerson` 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“). 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 Zweck (der hier durch das Präfix „neue“ in ihrem Namen angedeutet ist; statt „neue“ findet man hier häufig auch „make“ oder schlicht „mk“).

WIKIPEDIA

8.2 Konstruktoren

Auch wenn die Verwendung von Objektliteralen in Fabrikfunktionen eine populäre Möglichkeit ist, systematisch neue Objekte zu erzeugen, ist in JAVASCRIPT ein anderer Weg, Funktionen zur Objekterzeugung heranzuziehen, prominent. Dazu werden Funktionen über das Schlüsselwort `new` aufgerufen, also beispielsweise per

```
208 new Person('Hein', 35)                                               JS
```

(wobei `Person` hier eine Funktion ist). 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 9).
3. Die Funktion `Person` wird ausgeführt und dabei `this` temporär 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 der Gesamtausdruck `this`, also das neue Objekt zurück, andernfalls den Wert des Return-Statements.

Nun liefert obige Definition von `neuePerson` (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 neuePerson()` kein anderes Ergebnis als der ohne `new`; das durch `new` erzeugte leere Objekt wird verworfen. Um hingegen das von `new` erzeugte Objekt zurückzuerhalten, muss eine Funktion wie folgt definiert werden:

```
209 function Person(name, alter) {                                    JS
210   this.name = name
211   this.alter = alter
212 }
```

Damit erhalten wir dann

```
213 > new Person('Hein', 35)                                          JS
    Person { name: 'Hein', alter: 35 }
```

Hierbei unterscheidet sich die Ausgabe der REPL gegenüber der bei Verwendung der Fabrikfunktion 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**. Dabei schreibt man die Namen von Konstruktoren wie im obigen Beispiel von `Person` per Konvention groß. Konstruktoren grenzen sich von *Fabrikfunktionen* 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 mit ihren jeweiligen Konstruktoren verbinden. Dabei ersetzt die Verbindung mit Konstruktoren die Verbindung von Objekten mit Klassen, wie Sie sie vielleicht von klassenbasierten objektorientierten Programmiersprachen schon kennen. JAVASCRIPT kennt aber — wie alle prototypenbasierten objektorientierten Programmiersprachen — keine wirklichen Klassen.

Parameterlose Konstruktoren können in JAVASCRIPT übrigens ohne die leeren Klammern aufgerufen werden. Dies unterscheidet sie von normalen Funktionsaufrufen (also ohne `new`), bei denen die Klammern zwingend erforderlich sind (als *Aufrufoperator*).

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 zum 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 Direktive `'use strict'`) unterbunden — `this` hat dann stattdessen den Wert `undefined`, dessen Dereferenzierung (beim Zugriff auf eine Eigenschaft über `this`) zu einem Laufzeitfehler führt.

8.3 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 7.3 gesehen, Methoden genau wie Felder behandeln, sie also insbesondere dynamisch Objekten hinzufügen. Es wäre also sowohl im Fall der Fabrikfunktionen als auch im Fall der Konstruktoren leicht 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, jedoch nicht anhand ihrer Struktur und nicht anhand ihrer Methoden — alle haben den gleichen Satz an Eigenschaften oder, genauer, *prägen die gleichen Eigenschaften aus* und zeigen das gleiche Verhalten. 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

```
214 function Person(name, alter) {
215   return { name, alter, hallo: function() { console.log(this.name) } }
216 }
```

manifestiert, wird im Fall der Konstruktorfunktion durch

```
217 function Person(name, alter) {                                    JS
218   this.name = name
219   this.alter = alter
220   this.hallo = function() { console.log(this.name) }
221 }
```

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 vollkommen generisch ist, indem sie 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 Anzweifelung 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 9.

8.4 Standardkonstruktoren

In JAVASCRIPT gibt eine ganze Reihe von Standardkonstruktoren, wie z. B. `Object`, `Array` und `Function`. So ist beispielsweise der Aufruf `new Object()` gleichbedeutend mit der Auswertung von `{}` (erzeugt also ein neues, leeres Objekt) und `new Array('a', 'b', 'c')` 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 erlaubt insbesondere die Verwendung von Konstruktoren als Träger von **Fabrikmethoden** als objektorientierte Variante von *Fabrikfunktionen*, die eine Alternative zu den jeweiligen Konstruktoren darstellen. Ein gutes Beispiel hierfür ist der Konstruktor (die Funktion) `Object`: er bietet neben der Objekterzeugung mit `new Object()` den alternativen Aufruf einer Fabrikmethode über `Object.create(o)` an, wobei diese Methode gleich auch noch den Prototyp des neuen Objekts auf `o` festlegt (was das bedeutet, lernen Sie in Kapitel 9). 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 179 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.

8.5 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:

```
222 > typeof 'abc'                                                    JS
    'string'
223 > typeof new String('abc')
    'object'
```

Gerade bei Strings kann das sinnvoll sein, da lange Strings als Werte viel Speicher belegen und Stringoperationen, die jedes Mal neue Strings erzeugen anstatt einen vorhandenen String zu ändern, insbesondere in Schleifen oder bei rekursiven Aufrufen viel Speicher verbraten. Auch wird mit diesen Konstruktoren das sog. **Boxing** vollzogen, das einen primitiven Wert in ein entsprechendes Objekt packt, das dann an die Stelle des Werts tritt, so dass darauf Methoden, die mit dem jeweiligen Konstruktor definiert wurden, ausgeführt werden können:

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

oder

```
225 > 1['toFixed'](2)                                                 JS
    '1.00'
```

(wobei hier die Punktschreibweise für den Aufruf von `toFixed` wegen der Interpretation des Punkts als Dezimalkomma zu einem Syntaxfehler führen würde). Da das Boxing hier automatisch erfolgt (ohne einen expliziten Aufruf des Konstruktors `String` bzw. `Number`), spricht man auch von **Autoboxing**; es entspricht in etwa der *impliziten Umwandlung* unpassender Operanden von Operationen, wie wir sie Abschnitt 6.2.2 kennengelernt haben. Ein *Autounboxing* findet hierbei übrigens nicht statt, auch wenn

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

dies nahelegen würde; wir haben nämlich auch

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

(man beachte, dass der New-Operator stärker an seinen Operanden bindet als der Punkt als Selektionsoperator). Die Methode `trim` liefert einen Wert zurück und nicht das Objekt, auf dem es aufgerufen wurde — 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 9.2.

8.6 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

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

also ein Objekt, wohingegen

```
229 > 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:

```
230 function String(wert) {                                                JS
231   if (this instanceof String)
232     this.wert = wert.toString()
233   else
234     return wert.toString()
235 }
```

Hier ist `instanceof` ein eingebauter Operator, der testet, ob ein Objekt durch einen Konstruktoraufruf (`new` in Verbindung mit einem Funktionsnamen, hier `String`) erzeugt wurde. Dabei wird ausgenutzt, dass durch Konstruktoraufrufe erzeugte Objekte mit ihrem Konstruktor verbunden sind.

8.7 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:

```
236 > var original = new Person('Hein', 35)                                JS
237 > var kopie = Object.assign(new Person(), original)
238 > kopie
    Person { name: 'Hein', alter: 35 }
239 > 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.

Zusammenführen von mehreren Objekten in eines

Wenn man bei der Verwendung von `assign` weitere Objekte als Argumente angibt, werden auch deren Eigenschaften in das erste kopiert. Es handelt sich dabei um eine besonders einfache Methode, um bestimmte Features in ein Objekt „einzumischen“. So mischt beispielsweise

```
240 Object.assign(original, {laune: 'gut'}, {pläne: 'keine'})                JS
```

zwei Objekte, die jeweils nur ein Feld umfassen, in die oben erstellte Person `original` ein. Es handelt sich dabei um eine (recht primitive, da auf Kopieren basierende) Form, ein Objekt mithilfe von *Mixins* um Funktionalität zu erweitern. Auf Vererbung basierende Formen werden wir in Abschnitt 9.8 sowie später noch einmal in den Abschnitten 19.3 und 39.1.2 kennenlernen.

funktionales Update

Das Kopieren eines Objekts kommt auch zum Einsatz, wenn ein „funktionales Update“ eines Objekts durchgeführt, also anstatt den Zustand eines Objekts zu ändern, ein neues mit den Änderungen erzeugt werden soll. Dies gelingt in JavaScript beispielsweise mit der sog. **Spread-Syntax**:

```
241 > var original = new Person('Hein', 35)                                  JS
242 > var kopie = { ... original, alter: 42) }
243 > kopie == original
    false
244 > kopie
    Person { name: 'Hein', alter: 42 }
```

Hier wird das auf den Spread-Operator `...` folgende Objekt gewissermaßen ausgerollt, also seine Name/Wert-Paare, soweit sie `enumerable` sind, in das Objektliteral eingespleißt, wobei die nachfolgenden Eigenschaften (hier nur `alter: 42`) alle vorausgehenden mit gleichen Namen überschreiben. Dies Art der updatenden Kopie ist u. a. bei der Simulation von Wertobjekten, wie sie in Abschnitt 20.3.1 behandelt werden, interessant; sie entspricht im Wesentlichen dem, was wir auch schon beim Update von Laufzeitumgebungen in Abschnitt 3.4.3 gesehen haben.

Das Kopieren von Objekten stellt keinen Zusammenhang zwischen dem Original und der Kopie her. Insbesondere entsteht durch Kopieren in JavaScript (anders als beispielsweise durch das Klonen in der Sprache Self) keine Vererbungsbeziehung. Was das ist, dazu kommen wir als nächstes.

9 Prototypen und Vererbung

Wie bereits in Abschnitt 8.3 erwähnt ist das Muster, nach dem jedem Objekt, das von einer *Fabrikmethode* oder einem *Konstruktor* erzeugt wird, die gleichen Methoden beigefügt werden, nicht besonders ressourcenschonend. Besser wäre es in solchen Fällen, wenn sich die Objekte die Methoden teilen. Das Verhalten, dass durch diese Methoden abgebildet wird, hinge damit immer noch (über die Verwendung von `this` für den Zugriff auf Felder) vom Zustand der einzelnen Objekte ab.

9.1 Prototypen

Während in der klassenbasierten objektorientierten Programmierung Klassen dafür sorgen, dass sich mehrere Objekte ihre Methoden teilen, 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, was so viel heißt wie dass die Objekte alle Eigenschaften ihres Prototyps erben. Man spricht hier von **objekt**- oder **prototypenbasierter Vererbung**. Sie steht im Gegensatz zur *klassenbasierten Vererbung* der meisten anderen objektorientierten Programmiersprachen (Gegenstand von Kapitel 18).

Damit in der prototypenbasierten Programmierung ein Objekt von einem anderen erbt, muss es mit diesem als seinem Prototyp verbunden sein. Dies geschieht in JAVASCRIPT automatisch bei der Erzeugung von Objekten mittels Konstruktoraufrufen. Das schauen wir uns zunächst genauer an.

9.2 Vererbung über Konstruktoren

Prototyp-Eigenschaft von Funktionen

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

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

haben wir beispielsweise

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

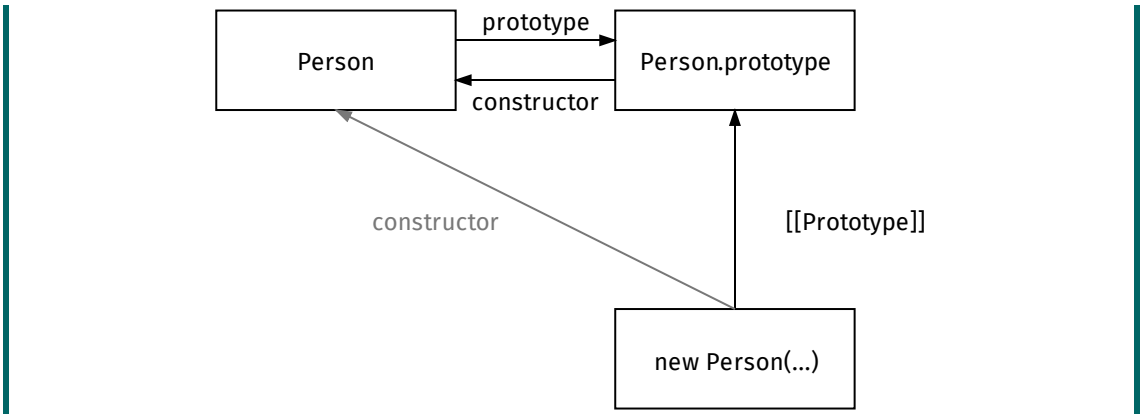
sowie

```
247 > new Person('Hein').constructor                               JS
[Function: Person]
```

Das neue Objekt hat selbst kein Feld `prototype`:

```
248 > new Person('Hein').prototype                                 JS
undefined
```

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), das dadurch geerbte Feld `constructor` ist gegraut.

erben vom Prototyp Nicht nur das Feld `constructor`, sondern alle Eigenschaften, 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 Erweiterung des mit `Person.prototype` benannten Prototypobjekts lassen sich so beispielsweise alle Objekte, die mit `new` und `Person` erzeugt wurden, um die Methode `hallo` ergänzen:

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

Für Felder gilt entsprechendes:

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

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:

```
253 > var p = new Person('Rosa')
254 > p.frei
    true
255 > Person.prototype.frei = false
256 > new Person('Gelb').frei
    false
257 > p.frei
    false
```
JS

Das Erben von Feldern ist damit aber nur bedingt sinnvoll — in der Regel sind es Methoden, die in einem Prototyp definiert und somit vererbt werden. Aber wie funktioniert das Erben von einem Prototyp?

Prototyp als verborgene Eigenschaft

In JAVASCRIPT hat jedes Objekt ein verborgenes Feld, das auf den Prototyp des Objekts verweist (in obiger Abbildung mit [[Prototype]] bezeichnet). Dieses Feld 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 des (nicht verborgenen) Felds `prototype` der Funktion ist.[78] Das verborgene Feld eines Objekts, das auf seinen Prototyp verweist, kann per

```
258 > Object.getPrototypeOf(p)                                    JS
    { hallo: [Function (anonymous)], frei: false }
```

abgefragt werden.[79] Tatsächlich gilt im (obigen) Fall von `p` als einem Objekt, das über `new Person()` erzeugt wurde,

```
259 > Object.getPrototypeOf(p) == 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` der Standardkonstruktor für Objekte ist; s. Abschnitt 8.4):

```
260 > 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 `p` selbst definiert ist, so dass `p.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**.

Konstruktor + Prototyp ≈ Klasse

Prototypenbasierte Vererbung ist die Erklärung dafür, wie Strings an Methoden wie `trim` (Abschnitt 8.5) kommen: Alle Strings, die durch einen Konstruktoraufruf von `String` entstanden sind, haben das durch `String.prototype` bezeichnete Objekt als Prototyp, der die Methode `trim` definiert, die alle Strings somit erben. Um die vererbten Eigenschaften, die allen Strings gemeinsam sein sollen, festzulegen, weist man sie einfach `String.prototype` zu, analog dazu, wie wir es oben am Beispiel von `Person` schon gesehen haben. Diese als idiomatisch zu betrachtende Vorgehensweise wird in neueren Versionen von

[78] 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.

[79] 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“).

JavaScript durch die Verwendung von Klassen abgelöst, die in JavaScript jedoch nur syntaktischer Zucker für die Definition von Konstruktoren und Prototypen sind. Da wir Klassen aber erst in Lektion 3 einführen, werden wir uns die Klassen JavaScripts auch erst in Kapitel 24 ansehen. Hier sei aber schon erwähnt, dass Strings mittels Prototypen einzuführen eine etwas seltsame Anmutung hat, sollten sich doch Strings grundsätzlich immer alle gleich verhalten. Nicht zuletzt gibt es ja genau deswegen für Strings als primitive Werte einen eigenen Datentyp (s. Abschnitt 6.2).

9.3 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. Insbesondere Methoden, die sich alle Objekte teilen sollen, 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 Unterschiede zu einem anderen Objekt, seinem Prototyp, definieren lässt. Dies lässt sich wie im Beispiel

```
261 > var o1 = { eigen: 1 }                                          JS
262 > var o2 = { vererbt: 2 }
263 > Object.setPrototypeOf(o1, o2)
264 > o1
    { eigen: 1 }
265 > 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 `o1` die Eigenschaft `vererbt` von `o2` erbt.

Klonen

Nun mag es etwas unnatürlich erscheinen, zunächst ein neues Objekt nur mit den Unterschieden zu erzeugen und ihm erst dann seinen Prototyp zuzuweisen, relativ zu dem die Unterschiede formuliert sind. Natürlicher schiene da, zunächst ein neues Objekt unter Angabe seines Prototyps zu erzeugen und erst dann die Differenzen hinzuzufügen. In JavaScript wird genau diese Vorgehensweise von der Methode `Object.create` unterstützt:

```
266 > var o1 = { vererbt: 1 }                                        JS
267 > var o2 = Object.create(o1)
268 > o2.eigen = 2
269 > o2
    { eigen: 2 }
```

```
270 > 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 damit alle Eigenschaften 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 popularisiert wurde. Die Konstruktoren JAVASCRIPTs nehmen sich im Vergleich dazu eher wie eine Verwirrung aus: Konstruktoren erinnern irgendwie an Klassen, sollen aber auf keinen Fall Klassen sein.

9.4 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 268 hatten wir `o2` ein neues Feld hinzugefügt, was dann aber auch tatsächlich in `o2` und nicht etwa in dessen Prototyp `o1` gelandet ist. Allgemein greift in JAVASCRIPT beim Schreiben von Eigenschaften der Mechanismus der Vererbung nicht:

```
271 > o2.vererbt = 3                                                    JS
272 > o1
    { vererbt: 1 }
273 > o2
    { eigen: 2, vererbt: 3 }
```

Das Schreiben (per Zuweisung) in eine geerbte Eigenschaft in Zeile 271 führt also dazu, dass die (zuvor geerbte) Eigenschaft im Objekt neu angelegt wird. Da `o2` jetzt selbst ein Feld mit Namen `vererbt` hat, wird das zuvor von `o1` geerbte gleichnamige Feld **überschrieben**, was auch **Overriding** genannt wird.[80] Das Feld wird damit vom erbenden Objekt aus zwar *unsichtbar*, ist damit aber nicht weg, sondern kann immer noch über

```
274 > Object.getPrototypeOf(o2).vererbt                                 JS
    2
```

oder, aus dem Kontext von `o2` (also beispielsweise dem Rumpf einer Methode), über

```
275 super.vererbt                                                       JS
```

[80] Mit Überschreiben ist hier ausdrücklich nicht das Überschreiben einer Eigenschaft mit einem neuen Wert (engl. *overwriting*) im Rahmen einer Zuweisung gemeint — dies hat mit Vererbung nichts zu tun. Leider müssen wir im Deutschen mit dieser Mehrdeutigkeit leben.

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

```
> delete o2.vererbt                                                    JS
> o2.vererbt
2
```

wieder geerbt.

Überschreiben von Feldern

Das Überschreiben ist im normalen Sprachgebrauch Methoden vorbehalten — von Feldern sagt man das normalerweise nicht. Da in JAVASCRIPT aber Felder und Methoden gleichermaßen unter die Eigenschaften eines Objekts fallen und Zugriffe auf Felder und Methoden auf die gleiche Weise dynamisch gebunden werden, rede ich auch vom *Überschreiben von Feldern*. Das ist allerdings eher unüblich, schon weil in den meisten Sprachen Felder statisch gebunden werden und Überschreiben an der *dynamischen Bindung* hängt.

Überschreiben vs. Verdecken

Da JAVASCRIPT eine *dynamische Sprache* ist, in der sich die Struktur von Objekten jederzeit ändern kann, müssen Feldzugriffe genau wie Methodenaufrufe *dynamisch gebunden* werden (s. Abschnitt 7.3.2). In *statischen* objektorientierten und objekt-funktionalen Programmiersprachen werden hingegen nur Methodenaufrufe dynamisch gebunden — Feldzugriffe finden ihr Ziel hingegen schon zur Übersetzungszeit. Allerdings können auch in diesen Sprachen gerbte Felder durch eigene mit dem gleichen Namen ersetzt werden, wodurch die geerbten Felder wiederum *unsichtbar* werden. Dies nennt man dann allerdings **verdecken** oder **Hiding**.[81]

Die Möglichkeit des Überschreibens geerbter Eigenschaften ist für die objektorientierte Programmierung genauso wesentlich wie die Vererbung selbst. Wir werden in Abschnitt 9.7 zunächst kurz und dann in Kapitel 18 ausführlicher darauf eingehen.

9.5 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:

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

[81] 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 Geltungsbereich. Shadowing war uns schon im Lambda-Kalkül (Abschnitt 3.3.1) begegnet.

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:

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

9.6 Vererbung und dynamisches Binden

Wir hatten in den Abschnitten 7.3.2 und 9.4 ja bereits gesehen, dass Zugriffe auf Eigenschaften eines Objekts, das durch einen variablen Ausdruck (also beispielsweise eine Variable) repräsentiert wird, also beispielsweise

```
280 o.m()                                                              JS
```

dynamisch gebunden werden: Solange der Wert von `o` nicht bekannt ist, steht nicht fest, welche Methode `m` hier aufgerufen wird. Die Bindung des Zugriffs auf `m` kann also erst zur Laufzeit des Programms stattfinden, wenn der Wert von `o` bekannt ist. Daher kann es auch erst zur Laufzeit zu einer Fehlermeldung kommen, nämlich wenn `o` die Eigenschaft `m` nicht hat. Dies unterscheidet den Zugriff auf Eigenschaften von Objekten grundsätzlich von Funktionsaufrufen, bei denen die aufgerufene Funktion feststeht (was in JAVASCRIPT allerdings auch nicht unbedingt der Fall ist; s. Abschnitt 10.2), und insbesondere von Zugriffen auf Variablen, denn hier können fehlende Bindungen schon vorher (statisch) festgestellt werden.

dynamisches Binden im Kontext der Vererbung

Die Bedeutung des dynamischen Bindens wird nun 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. Im Fall von Methodenaufrufen ergibt sich dabei eine wichtige Besonderheit: `this` verweist beim Ausführen der gefundenen Methode in deren Rumpf nicht auf den Prototyp, der die Methode definiert, sondern auf das ursprüngliche Empfängerobjekt. Das garantiert, dass das Verhalten eines Objekts, das in einer geerbten Methode festgeschrieben ist, vom Zustand des erbenden Objekts abhängt, auf dem die Methode aufgerufen wurde. Zudem ergeben sich bei objektinternen Methodenaufrufen über `this` Verhaltensweisen, die für die objektorientierte Programmierung zentral sind, wie wir gleich sehen werden.

9.7 Offene Rekursion

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

```
281 > var o1 = {                                                    JS
282 .   m() { this.n() },
283 .   n() { console.log(1) }
284 . }
285 > var o2 = {
286 .   n() { console.log(2) }
287 . }
288 > Object.setPrototypeOf(o2, o1)
289 > o1.m()
    1
290 > o2.m()
    2
```

Was ist hier los? Im Fall des Aufrufs `o1.m()` aus Zeile 289 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 290 die in `o1` definierte, von `o2` geerbte Methode `m` auf `o2` aufgerufen, so verweist `this` in im Rumpf von `m` in Zeile 282 nicht auf `o1` wie im ersten Fall, sondern auf `o2`. Die Methode `m` ruft hier also in ihrem Rumpf die `o1.n` überschreibende Methode `o2.n` auf, was zur Ausgabe von `2` führt.

Das gleiche gilt in JavaScript übrigens für einen Feldzugriff:

```
291 > o1 = {                                                        JS
292 .   f: 1,
293 .   m() { console.log(this.f)}
294 . }
295 > o2 = { f: 2 }
296 > Object.setPrototypeOf(o2, o1)
297 > o2.m()
    2
```

Inversion of control

Der Mechanismus der offenen Rekursion ist für die praktische 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 des Frameworks aufrufen kann, obwohl `o1` selbst gar keine Kenntnis von `o2` oder `o2.n` hat — `o2.n` 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 18.3 wieder auf Anwendungsframeworks zurückkommen.

9.8 Delegation

Forwarding

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

```
298 var o1 = {                                                    JS
299   m() { this.n() },
300   n() {}
301 }
302 var o2 = {
303   o: o1,
304   m() { this.o.m() },
305   n() {}
306 }
```

Hier wird mit `o2.m()` über `this.o.m()` zunächst die Methode `o1.m` und in der Folge dann über `this.n()` `o1.n` aufgerufen. Der Aufruf durch `this.n()` kann nicht berücksichtigen, dass `m` ursprünglich auf `o2` aufgerufen wurde, da der Aufruf von `o1.m` von `o2` aus über das Feld `o` erfolgt, das `o1` referenziert, und nicht über `super` (was hier auch nicht funktioniert hätte, da `o2` nicht von `o1` erbt). Das Objekt `o1` tritt also bei der Erledigung von `m` vollständig an die Stelle von `o2` — `o2` hat den Aufruf von `m` an `o1` geforwardet.

Delegation über Vererbung

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

1. `o2` muss `o` (und damit `o1`) als Prototypen nennen.
2. Der Aufruf `this.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, über Aufrufe auf `this` in das Verhalten des Objekts, an das delegiert wurde, eingespleißt. Die Prototypenbasierte Vererbung von JAVASCRIPT realisiert also Delegation.

Delegation unabhängig von Vererbung

Offensichtlicher Nachteil der Umsetzung von Delegation über Vererbung ist, dass hier immer nur an ein Objekt delegiert werden kann, das zudem auch noch als Prototyp herhalten muss, obwohl das delegierende Objekt vielleicht ein anderes Objekt als Prototyp hat. Soll an mehrere (oder über die Zeit verschiedene) Objekte delegiert werden, muss zudem jeweils unmittelbar zuvor der Prototyp gewechselt werden. Da das im Allgemeinen weiterreichende Implikationen hat als nur die offene Rekursion bei einem Aufruf, generalisiert dieser Ansatz nicht besonders gut. An dessen Stelle kann man in JAVASCRIPT in Zeile 304 aber einfach

```
307 this.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 304 ist). Somit ruft auch `this.n()` in Zeile 299 `o2.n` auf, was hier gewollt war. Die Methode `call` ist übrigens in `Function.prototype` definiert und wird von allen Funktionen geerbt.

Mixins

Der offensichtliche Vorteil dieser Vorgehensweise ist, dass ein Objekt so an beliebige andere Objekte delegieren kann, ohne dass sich das Objekt an diese als seine Prototypen binden muss — stattdessen wird `this` in den per `call` aufgerufenen Methoden der anderen Objekte für den jeweiligen Aufruf an das delegierende Objekt gebunden. Auf diese Weise lassen sich in JAVASCRIPT *Mixins* realisieren, also das Einspleißen des Verhaltens verschiedener anderer Objekte in das eigene. Mixins spielen in RUBY und in SCALA herausgehobene Rollen.

Komposition

Obwohl nicht Gegenstand dieses Abschnitts, sei doch an dieser Stelle schon erwähnt, dass Forwarding und, sofern in einer Sprache verfügbar, auch die von Vererbung unabhängige Delegation in der Praxis eine wichtige Alternative zur Vererbung darstellen, die unter dem Begriff der *Komposition* bekannt ist. Wesentlich ist dabei, dass ein Objekt sich ein anderes nicht nur hält, sondern auch dessen Methoden wie seine eigenen zur Verfügung stellt (was oben im Fall von `o2` mit der Methode `m` der Fall ist), so dass die Komposition von außen zunächst nicht von der Vererbung zu unterscheiden ist. Klienten von `o2`, die ja anstatt `o2.m()` auch `o2.o.m()` für den Dienst von `o1.m` verwenden könnten (und damit dasselbe Ergebnis erhalten würden), müssen so gar nicht wissen, dass `o2` eine Eigenschaft `o` hat oder dass es `o1` gibt, die beide auch die Methode `m` anbieten — wenn sie ausschließlich `o2.m` verwenden, bleiben sie unabhängig von der Entscheidung, wie `o2` seine Methode `m` realisiert. Außerdem muss `o2` so nicht alle Eigenschaften von `o1` anbieten (im obigen Beispiel `n` nicht), sondern kann sich auf die beschränken, die es anbieten will. Nicht zuletzt kann `o2` auch in den Sprachen, in denen die Vererbungsbeziehung zur Laufzeit unveränderlich ist, seine Beziehung zu `o1` durch die zu einem anderen Objekt ersetzen, an das es seine angebotenen Dienste dann stattdessen delegiert. Wir werden in Abschnitt 19.1, dann im Kontext von Klassen, wieder darauf zurückkommen.

9.9 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 verschiedene Objekte sich 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, abgebildet im Wert eines Feldes, sondern der auch in den mit ihnen verbundenen Funktion unterscheiden). Es ist wohl eher kein Zufall, dass JAVASCRIPT — als Sprache für ausführbaren Inhalt von Webseiten — prototypenbasiert ist.

10 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* der Funktion (engl. „closure“), 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.

10.1 Funktionsliterale

Wie wir in Abschnitt 4.1 gesehen haben, können Funktionen in JAVASCRIPT als Werte von Eigenschaften auftreten, sind also (unter Zugrundelegung einer entsprechenden Definition von Werten; s. Abschnitt 3.3.3) Werte. Als solche haben sie auch eine literale Repräsentation:

```
308 > f = function() {}                                                    JS
    [Function: f]
```

Dabei ist `function() {}` das **Funktionsliteral**[82] einer anonymen und parameterlosen Funktion, die nichts tun soll; die Funktion, für die es steht, ist ein Objekt und wird hier der Variable `f` zugewiesen. Man beachte, dass die Funktion mit der Zuweisung keinen eigenen Namen erhält — `f` ist nur ein Alias für das erzeugte Objekt, die Funktion selbst bleibt, wie ein Lambda-Ausdruck, anonym (vgl. Abschnitt 3.3.1). Dass trotzdem „Function: f“ auf der Konsole zu lesen steht, liegt daran, dass JAVASCRIPT einer anonymen Funktion zu Debuggingzwecken den Namen der Variable gibt, der sie als erstes zugewiesen wurde. Dieser Name bleibt erhalten, selbst nachdem `f` ein anderer Wert zugewiesen wurde; mit `f` aufrufen kann man die Funktion dann aber nicht mehr (obwohl sie intern immer noch „f“ heißt). Der Name der Funktion ist nämlich nur eine Eigenschaft der Funktion als Objekt:

```
309 > f.name                                                               JS
```

[82] Manche würde sagen, dass es sich hierbei nicht um Literal, sondern um einen *Funktionsausdruck* handelt. Jedoch entält dieser Ausdruck keine Elemente, die erst noch ausgewertet werden müssten, bevor ein Objekt daraus wird — die Funktion ist Zeichen für Zeichen die, die da steht.

```
'f'
```

So kann man einer Funktion auch selbst Eigenschaften hinzufügen.

benannte Funktionen als Objekte

Funktionen müssen in JAVASCRIPT nicht anonym bleiben, sondern können auch explizit benannt werden. So ist die rechte Seite der Zuweisung

```
310 > f = function id(x) { return x }                                    JS
    [Function: id]
```

das Literal einer mit „id“ benannten Funktion, die über einen Parameter `x` verfügt und diesen zurückgibt. An der Konsolenausgabe erkennt man, das „id“ den Namen der Variable ersetzt hat, der die Funktion zugewiesen wird; allerdings ist der Name der Funktion selbst (nicht die Eigenschaft `name` des Funktionsobjekts, die wir auf der Konsole sehen) nur innerhalb der Funktion gültig, er kann also nur zu rekursiven Aufrufen eingesetzt werden. So haben wir denn auch

```
311 > id.name                                                            JS
    Uncaught ReferenceError: id is not defined
```

auch Funktionsdeklarationen erzeugen Funktionsobjekte

Dies steht im Gegensatz zu den per Deklarationsanweisung definierten Funktionen, die ebenfalls Objekte sind, deren Funktionsnamen aber (als Variablennamen) im dem Scope, in dem die Deklaration steht, gültig sind (s. Abschnitt 6.5):

```
312 > function id(x) { return x }                                        JS
313 > id.name
    'id'
```

Dass `id` wirklich ein Objekt bezeichnet, sieht man an

```
314 > typeof id                                                          JS
    'function'
```

wobei der Datentyp `'function'` als Spezialfall des Datentyps `'object'` zu verstehen ist. Dass es sich bei `id` tatsächlich um eine Variable handelt, sieht man an

```
315 > id(1)                                                              JS
    1
316 > id = 0
317 > id(1)
    Uncaught TypeError: id is not a function
```

Der semantische Unterschied zwischen einer Funktionsdeklaration wie der von `id` in Zeile 312 und der Definition derselben Funktion als Wert einer Variablen wie in

```
318 var id = function id(x) { return x }                                 JS
```

liegt in JAVASCRIPT lediglich in Nuancen.

Arrow-Funktionen

In einem Funktionsliteral wie `function(x) {return x}` entspricht `function` einem *Bindungsoperator*, der die Variable `x` an den Rumpf der Funktion bindet. Anonyme Funktionen lassen sich in JAVASCRIPT auch mit dem *Bindungsoperator* `=>` , der uns schon als $\mapsto$ aus dem Lambda-Kalkül bekannt ist (Abschnitt 3.3.1) definieren: So ist

```
319 x => x                                                                    JS
```

eine sog. Arrow-Funktion, die dem Lambda-Ausdruck $\lambda x.\, x$ bzw. $x \mapsto x$, der ja auch Funktion und Wert zugleich ist, entspricht (s. Abschnitt 3.3.3). Soll eine solche Arrow-Funktion keinen oder mehrere Parameter haben, müssen an der Parameterposition Klammern gesetzt werden. So steht `() => {return}` für eine parameterlose Funktion, die nichts (`undefined`) zurückgibt.

Unterschied zu mit `function` erzeugten Funktionen

Auch wenn es sich bei Arrow-Funktionen ebenfalls um Objekte handelt, unterscheiden sie sich von den obigen, mit `function` erzeugten darin, dass sie in ihrem Rumpf die Pseudovariable `arguments` nicht zur Verfügung haben und `this` Teil des *Funktionsabschlusses* ist, also bei Aufruf der Funktion nicht dynamisch gebunden wird (s. Abschnitt 10.3). Letzteres heißt insbesondere, dass es nicht sinnvoll ist, eine Arrow-Funktion für die Definition einer Methode zu verwenden, weil `this` in ihrem Rumpf nicht für das Objekt steht, auf dem die Methode später einmal aufgerufen wird. Dessen unbeschadet sieht man aber manchmal mehrere Arrow-Funktionen als Werte von Name/Wert-Paaren zu einem Objekt zusammengefasst:

```
320 const rechner = {                                                         JS
321   add: (x, y) => x + y,
322   sub: (x, y) => x - y
323 }
```

Das ist freilich unproblematisch, weil hier `this` gar nicht vorkommt.

10.2 Funktionsanwendung

So, wie im Lambda-Kalkül eine Funktion ein Wert ist, der auf andere Werte als Argumente angewendet werden kann, lässt sich auch in JAVASCRIPT eine Funktion als Objekt auf Argumente anwenden. Dabei schreibt man in JAVASCRIPT die Argumente in Klammern hinter die Funktion, wobei hier die Klammern für einen Operator, den *Anwendungsoperator*, stehen. So erhalten wir beispielsweise

```
324 > (x => x) (1)                                                           JS
    1
```

oder auch, unter Verwendung eine *Template-Literals* für eine aussagekräftige Ausgabe,

```
325 > `${(x => x)(y => y)}`                                                  JS
    'y => y'
```

indirekter Funktionsaufruf

Die Zuweisung von Funktionen an Variablen erlaubt einen indirekten Funktionsaufruf:

```
> var id = x => x                                                    JS
> id(1)
1
```

Um `id(1)` auszuwerten, wird erst die Variable `id` ausgewertet und dann ihr Wert als Funktion auf `1` angewendet. Dies entspricht im Wesentlichen dem Ausdruck

$$\text{let } id \;=\; \lambda x.\, x \text{ in } id\ 1$$

aus dem Lambda-Kalkül (s. Abschnitt 3.4.5), wobei der Geltungsbereich von *id* hier auf den In-Teil beschränkt ist, den man in JAVASCRIPT erst noch durch Blockbildung nachbilden müsste. Man beachte, dass es sich bei `id(1)` auch dann nicht um einen klassischen, statisch gebundenen Funktionsaufruf handeln kann, wie man ihn von statischen Sprachen her kennt, wenn `id` als Funktion deklariert wurde, weil, wie wir in Abschnitt 10.1 gesehen haben, in JAVASCRIPT auch eine Funktionsdeklaration nur eine Variable mit dem Namen der Funktion vereinbart, unter deren Verwendung die Funktion dann aufgerufen werden kann. Damit sind lediglich die Aufrufe von Funktionsliteralen wie oben direkt, die jedoch in der Praxis kaum Anwendung finden dürften.

Genau wie im Lambda-Kalkül kann in JAVASCRIPT in einem Funktionsaufruf an der Stelle der Funktion auch jeder andere Ausdruck stehen, der zu einem Funktionsobjekt auswertet. Damit lässt sich auch das *Currying* (Abschnitt 3.3.9) in JAVASCRIPT realisieren. Da es nicht auf Variablensubstitution, sondern auf eine Laufzeitumgebung setzt, brauchen wir dafür Funktionsabschlüsse.

10.3 Funktionsabschlüsse

Der Rumpf einer Funktion hat nicht nur Zugriff auf die Parameter der Funktion und ihre lokalen Variablen, sondern auch auf alle die Variablen, die in dem Geltungsbereich, in dem die Funktion definiert ist, gelesen und geschrieben werden können. Nehmen wir beispielsweise den Block

```
> { const c = 1; var f = () => c }                                   JS
```

dann ist `c` innerhalb des Blocks, und damit für die Definition von `f`, gültig, aber außerhalb des Blocks ungültig:

```
> c                                                                  JS
Uncaught ReferenceError: c is not defined
```

Das liegt daran, dass `c` für die Ausführung des Blocks auf dem Stack angelegt und danach wieder entfernt wird. Für `f` gilt das jedoch nicht, da Var-Deklarationen hochgezogen werden:

```
> f                                                                  JS
[Function: f]
```

Bemerkenswert ist nun, dass `f` immer noch Zugriff auf `c` hat, obwohl `c` zum Zeitpunkt der Ausführung gar nicht mehr da ist:

```
> f()                                                                    JS
1
```

Das liegt daran, dass in JAVASCRIPT eine Funktion stets einen *Abschluss* über ihre Umgebung bildet (vgl. dazu Abschnitt 3.4.4; jedoch nur, soweit Elemente dieser Umgebung von der Funktion auch verwendet werden); so hat die obige Funktion, die `f` zugewiesen wurde, die Konstante `c` in ihrem Abschluss. Dasselbe funktioniert in JAVASCRIPT (anders als z. B. in JAVA) auch mit Variablen, die dann aus dem Abschluss heraus beschrieben werden können, selbst wenn der in einem anderen Geltungsbereich ausgeführt wird:

```
> function app(f) { f() }                                                JS
> { let x = 0; app(() => x += 1); x }
1
```

Mittels des Funktionsabschlusses kann also nicht nur die Lebenszeit von Variablen verlängert werden, sondern die Variablen können auch in andere Geltungsbereiche transportiert und da gelesen und geschrieben werden. Ob das gut oder schlecht ist, sei dahingestellt; es spielt jedenfalls in den nächsten beiden Abschnitten eine entscheidende Rolle.

Abhängigkeit von der Variablendeklaration

Besonders in Schleifen ist darauf zu achten, wie Variablen in ihren Kontext eingebettet sind. So haben wir einerseits

```
> var a = Array(2)                                                       JS
> for (var i of a.keys()) a[i] = () => i
> a[0]()
1
```

aber andererseits

```
> for (let i of a.keys()) a[i] = () => i                                 JS
0
```

Im ersten Fall teilen sich die beiden in der Schleife erzeugten Abschlüsse die Variable `i`, im zweiten nicht (wie auch bei der Verwendung von `const` anstelle von `let`).

`this` in Abschlüssen über Arrow-Funktionen und Methoden

Umgekehrt ist darauf zu achten, dass die Verwendung von `this` in Arrow-Funktionen als Eigenschaften eines Objekts immer zu einem Funktionsabschluss führt, in dem das `this` aus der lexikalischen Umgebung eingefangen ist:

```
> this.x = 42                                                            JS
> var o = {
.   x: 35,
.   f: () => this.x
. }
> o.f()
42
```

Wäre `f` als Methode definiert worden (mit `function` oder der Methodensyntax), würde erstens kein Abschluss erzeugt und zweitens `this` dynamisch gebunden. Letzteres gilt auch, wenn die Methode `f` von einer anderen als Abschluss zurückgegeben wird.

10.4 Currying

Da in JAVASCRIPT Funktionen Objekte sind, können sie auch von anderen Funktionen zurückgegeben werden. Dies erlaubt die **gecurryte Darstellung** mehrstelliger Funktionen, also beispielsweise die Implementierung der zweistelligen Addition als zwei einstellige Funktionen, von denen die erste die zweite als Ergebnis zurückgibt (womit die erste eine Funktion höherer Ordnung ist). Eine solche Implementierung ist mit

```
344 function addiere(a) {                                                    JS
345   return function (b) {
346     return a + b
347   }
348 }
```

gegeben. Die Funktion `addiere` kann dann beispielsweise per

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

angewendet (oder aufgerufen) werden, wobei die zweite Anwendung (mit Argument `2`) nicht unmittelbar nach der ersten erfolgen muss:

```
350 > addiere(1)                                                            JS
    [Function (anonymous)]
```

Dabei hat eine isolierte Anwendung wie `addiere(1)` durchaus einen tieferen Sinn, liefert sie doch eine Funktion zurück, die, auf eine Zahl angewendet, eine um 1 größere zurückliefert. Diese Funktion hat denn auch ihre eigene Existenzberechtigung und deswegen einen Namen verdient:

```
351 > const succ = addiere(1)                                               JS
352 > succ(0)
    1
```

Anders als im Lambda-Kalkül mit Reduktionssemantik, bei dem der Parameter `a` von `addiere` bei der Anwendung auf 1 im Rumpf der inneren Funktion, `return a + b`, durch 1 ersetzt worden wäre (s. Abschnitt 3.3.9), handelt es sich bei `succ` um einen Funktionsabschluss, in dessen Umgebung der Wert von `a` konserviert ist.

10.5 Funktionen als Parameter: Kontrollabstraktion

Da Funktionen Objekte sind, können sie auch Funktionen und Methoden als Argumente übergeben werden. Eine besonders populäre Anwendung dieser Möglichkeit sind die sog. **internen Iterationen**, wie sie in JAVASCRIPT beispielsweise für Array-Objekte definiert sind:

```
353 > [1, 2].forEach(x => console.log(x))                                    JS
    1
    2
354 > [1, 2, 3].map(x => x + 1)
    [ 2, 3, 4 ]
```

Dabei ist sind die Methoden `forEach` und `map` in `Array.prototype` definiert und werden deswegen von allen Arrays geerbt. Sie nehmen als Argument eine Funktion entgegen (häufig wie hier ein Funktionsliteral), die auf jedes Element des Arrays angewendet wird, wobei die Ergebnisse im Fall von `map` wiederum in einem Array gleicher Länge gesammelt und zurückgegeben werden. Während die Bedeutungen und Funktionsweisen von `forEach`, `map` und anderen Funktionen, die auf Arrays und anderen sog. *Collections* definiert sind, Gegenstand von Kapitel 21 sind, sei hier noch erwähnt, dass beide *Funktionen höherer Ordnung* sind (weil sie Funktionen als Eingaben haben).

Kontrollabstraktion

Die Übergabe von Funktionen als Parameter ermöglicht in der funktionalen Programmierung (und damit auch in der objekt-funktionalen Programmierung) die sog. **Kontrollabstraktion**: Anstatt den Kontrollfluss mit entsprechenden Kontrollstruktur (wie z. B. einer `for...of` Schleife) auszudrücken, wird er in eine Funktion höherer Ordnung ausgelagert (wie hier in `forEach` und `map`). Damit die Kontrollabstraktionen aber das gleiche leisten können wie die Kontrollstrukturen, die sie ersetzen sollen, ist es unverzichtbar, dass die ihnen übergebenen Funktionen Abschlüsse über ihren Kontext bilden. Nur so funktioniert auch die Übersetzung von Konstruktionen wie

```
355 const a = 3                                                               JS
356 const result = []
357 for (const x of [1, 2]) {
358   result.push(x + a)
359 }
```

in eine Kontrollabstraktion wie

```
360 const a = 3                                                               JS
361 [1, 2].map(x => x + a)
```

Kernbeitrag für die objekt-funktionale Programmierung

Kontrollabstraktionen, die es auch schon in SMALLTALK gab, findet man mittlerweile in fast allen objektorientierten Programmiersprachen, die schon allein deswegen eigentlich objekt-funktional genannt werden müssten. Ihre Vorteile für die Programmierung wird niemand ernsthaft bestreiten. Allerdings sind die Möglichkeiten der Kontrollabstraktion in JAVASCRIPT vergleichsweise beschränkt, wovon wir uns am Beispiel von RUBY in der nächsten Lektion (Abschnitte 15.9 und 16.3) selbst überzeugen können.

10.6 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 ansehen kann).

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

```
362 > function wer() { console.log(this.ich) }                    JS
363 > var o1 = { ich: 'o1', hallo: wer }
364 > var o2 = { ich: 'o2', hallo: wer }
365 > o1.hallo()
    o1
366 > o2.hallo()
    o2
```

Hierbei wird ausgenutzt, dass Funktionen Objekte sind, die durch ihren Namen (hier `wer`) bezeichnet werden.

explizite Bindung an Empfänger

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

```
367 > var o1 = { ich: 'o1' }                                      JS
368 > var o2 = { ich: 'o2' }
369 > function hallo() { console.log(this.ich) }
370 > hallo()
    undefined
371 > hallo.bind(o1)()
    o1
372 > hallo.bind(o2)()
    o2
```

Dabei gibt `bind` jeweils eine neue Funktion zurück, die an das jeweilige Argument gebunden ist — die ursprüngliche Funktion `hallo` bleibt davon unberührt. Wie das Beispiel zeigt, wird es damit möglich, Methoden auf Objekten auszuführen, die diese Methoden gar nicht besitzen. Man mag darüber streiten, ob dies mit der *Datenkapselung*, die die Objektorientierung (wie in Abschnitt 7.3.3) beschrieben eigentlich bedeuten soll, bricht — in keinem Fall steht es aber im Einklang mit der Bündelung von Daten und Funktionen in einer Einheit, dem Objekt, für die die Objektorientierung eigentlich steht.

11 Objekte als Funktionen

Wir hatten ja bereits am Beispiel des Lambda-Kalküls gesehen, dass sich Daten als Funktionen codieren lassen, nämlich mit den *Church-Codierungen* aus Abschnitt 3.3.10. Das funktioniert auch in JAVASCRIPT, wie die folgende Beispiele eindrucksvoll zeigen.

Church-Codierungen in JAVASCRIPT

Mit Funktionen als Objekten, Funktionsanwendung, Currying und Funktionsabschlüssen haben wir alles, was wir für die Übertragung der Church-Codierungen vom Lambda-Kalkül in JAVASCRIPT, und damit der Codierung von Objekten als Funktionen, brauchen. So bekommen wir

```
373 > var                                                                    JS
374 .   tru = x => y => x,
375 .   fls = x => y => y,
376 .   ite = i => t => e => i(t)(e),
377 .   and = x => y => x(y)(x),
378 .   not = x => x(fls)(tru)
379 > ite(and(tru)(not(fls)))('richtig')('falsch')
    'richtig'
```

Am Beispiel der Church-Arithmetik sehen wir allerdings, dass JAVASCRIPT per Call-by-Value unter Verwendung einer Laufzeitumgebung mit Funktionsabschlüssen auswertet, so dass Ausdrücke nicht wie in Abschnitt 3.3.10 unter Verwendung von β-Reduktion vollständig reduziert werden:

```
380 > var                                                                    JS
381 .   n0ll = s => z => z,
382 .   eins = s => z => s(z),
383 .   zwei = s => z => s(s(z)),
384 .   add = m => n => s => z => m(s)(n(s)(z)),
385 .   succ = n => s => z => s(n(s)(z))
386 > `${succ(eins)}`
    's => z => s(n(s)(z))'
```

In der Ausgabe ist `n` eine freie Variable, deren Wert `s => z => s(z)` sich aus dem Funktionsabschluss ergibt, den wir als Ergebnis erhalten, an dessen eingefangene Variablen wir aber ohne einen Debugger nicht ohne weiteres herankommen (was durchaus gewollt ist; s. Abschnitt 11.1). Zugleich erkennen wir, dass der Rumpf der erhaltenen Funktion, `s(n(s)(z))`, unter Verwendung der Umgebung noch weiter auswertbar gewesen wäre, die Funktion dafür aber (wegen der Auswertung mit Call-by-Value) erst angewendet werden muss. So ergibt sich dann auch

```
387 > succ(eins)(x => x + 1)(0)                                              JS
    2
388 > add(zwei)(zwei)(x => x + 1)(0)
    4
```

Das funktioniert also schon einmal. Nur sind Wahrheitswerte und Zahlen aber eigentlich Werte und keine Objekte; sie haben insbesondere keinen Zustand, wie man ihn von Objekten erwarten würde.

11.1 Datenkapselung durch Funktionsabschlüsse

Funktionsabschlüsse und das damit verbundene „Retten“ von lokalen (und damit aus weiten Teilen des Programms nicht zugreifbaren) Variablen über ihre eigentliche Lebensdauer hinaus (was wir oben schon am Beispiel von `n` gesehen haben) kann gezielt dazu verwendet werden, Daten zu kapseln. Dies zeigt das folgende Beispiel von der Implementierung eines Zählers als Funktionsabschluss:

```
389 function neuerZähler() {                                                    JS
390   let x = 0
391   return function () { x += 1; return x }
392 }
```

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

```
393 > var z = neuerZähler()                                                     JS
```

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

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

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 mit jedem Aufruf von `neuerZähler` neu (mit neuer Variable `x`) erzeugt wird. Es erfolgt hier also eine **Datenkapselung durch Funktionsabschluss**.

Funktionsabschluss als Objekt mit Zustand

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: Objektorientiert würde man für obigen Zähler so etwas wie

```
396 function Zähler() {                                                         JS
397   this.x = 0
398   this.inkrement = function () { this.x += 1; return this.x }
399 }
400 var z = new Zähler()
```

schreiben und dann mit `z.inkrement()` das gleiche Verhalten wie oben mit `z()`bekommen. Allerdings ist in JAVASCRIPT `x` nicht wirklich gekapselt, da auch noch über `z.x` von außen zugreifbar (s. Abschnitt 7.3.3). Hier ist Datenkapselung durch Funktionsabschlüsse die sicherere Alternative, weswegen sie von manchen bevorzugt wird.

Objekte als Funktionsabschlüsse

Obiges Beispiel hat gezeigt, wie ein Funktionsabschluss ein zustandsbehaftetes Objekt ersetzen kann, sofern dieses Objekt genau eine Methode hat.

Funktionsabschlüsse sind aber auch geeignet, komplexere Objekte zu ersetzen, indem man mehrere Funktionsabschlüsse in einem Container zusammenfasst. Für die Zusammenfassung kann man wieder ein Objekt verwenden, das durch eine einfache Fabrikfunktion erzeugt wird:

```
401 function neuePerson(_name, _alter) {                                    JS
402   return {
403     get name() { return _name },
404     set name(name) { _name = name },
405     get alter () { return _alter },
406     set alter(alter) { _alter = alter }
407   }
408 }
```

Zustand im Abschluss

Dabei hat das durch einen Aufruf von `neuePerson` gelieferte Objekt nur Methoden (hier ausschließlich *Zugriffsmethoden*) und keine Felder — der Zustand liegt allein im Abschluss über die Methoden, die die beiden Parameter `_name` und `_alter` einfangen. Da sich die Getter und Setter die beiden Variablen paarweise teilen, entsprechen diese gleichnamigen Feldern des Objekts.[83] Natürlich kann man noch weitere Methoden (beliebige Queries und Commands) hinzufügen, die jeweils bedarfsweise auch über die Parameter des Konstruktors abschließen; das Ergebnis ist jedoch objektorientierter, wenn diese Methoden stattdessen die Zugriffsmethoden verwenden, weil sich dann auch die auf *Überschreiben* und *dynamisches Binden* basierenden Muster realisieren lassen.

11.2 Funktionale Objekte

Die vorigen Codierungen von Objekten über Funktionsabschlüsse sind nicht funktional im reinen Sinne: Die Objekte verfügen — wie in der imperativen objektorientierten Variante — über einen Zustand, der sich im Verlauf des Programms ändern kann. Will man das nicht, muss jedes *Command* ein neues Objekt zurückliefern, das in der Folge anstelle des alten verwendet wird (wodurch aus dem Command ein *Konstruktor* oder eine *Fabrikmethode* würde). Natürlich geht damit die Identität von Objekten verloren und da tröstet dann auch nicht, dass die funktionalen Objekte unveränderlich sind, denn wenn ein Objekt mit einem anderen Objekt über eine Referenz in Beziehung steht und das andere Objekt, anstatt sich zu ändern, durch ein neues abgelöst wird, dann bekommt die bestehende Referenz davon nichts mit, so dass die Beziehung nur noch eine historische ist. Aber sei's drum, wir schauen uns das an.

Am Beispiel des Zählers aus Abschnitt 11.1 sähen die Erzeugung eines *rein funktionalen Zählers* und seine Verwendung ungefähr so aus:

```
409 > function neuerZähler(x = 0) {                                         JS
410 .   return {
411 .     stand: () => x,
412 .     schritt: () => neuerZähler(x + 1)
413 .   }
```

[83] Man erkennt an diesem Beispiel, dass in einem Funktionsabschluss in JavaScript tatsächlich die originalen Variablen überleben und nicht etwa Kopien davon, wie das beispielsweise in Java der Fall ist. So verwenden hier die Getter und Setter jeweils dieselbe Variable.

```
. }
> var z0 = neuerZähler()
> z0.stand()
0
> var z1 = z0.schritt()
> z1.stand()
1
> z0.stand()
0
```

Objektserien Man braucht also für jeden Schritt eine neue Variable (was praktisch nur bei rekursiven Aufrufen mit Verwendung von `z.schritt()` zumutbar ist) oder man akzeptiert imperative Zuweisungen der Form `z = z.schritt()`. Tut man letzteres nicht, dann steht man vor folgender Situation: Wenn ein Objekt einen Zähler als Eigenschaft besitzt und der Zähler macht einen Schritt, dann muss man auch gleich ein neues Objekt erzeugen, wenn die Eigenschaft den neuen Zählerstand widerspiegeln soll. Das ist zwar sehr schön funktional, aber wenig objektorientiert.

funktionale Objekte Wie weit so etwas gehen kann, zeigt ein relativ einfaches und zugleich eindrückliches Beispiel, das an eines meines kürzlich verstorbenen Kollegen WILLIAM COOK angelehnt ist, mit dem die Idee von den (rein) **funktionalen Objekten** (manchmal auch als pure Objekte bezeichnet) oft verbunden wird. Das Beispiel codiert Listen (oder Strings) mittels einer Familie von Konstruktoren[84]:

```JS
function Empty() {
  this.isEmpty = () => true
  this.contains = e => false
  this.concat = s => s
}
function Single(e) {
  this.isEmpty = () => false
  this.contains = e1 => e1 === e
  this.concat = s => new Concat(this, s)
}
function Concat(s1, s2) {
  this.isEmpty = () => s1.isEmpty() && s2.isEmpty()
  this.contains = e => s1.contains(e) || s2.contains(e)
  this.concat = s => new Concat(this, s)
}
```

Man beachte, dass alle mit diesen drei Konstruktoren erzeugten Objekte insofern von derselben Art sind, als sie über den gleichen Satz an Methoden (aber nicht dieselben!) verfügen, der somit eine einheitliche *Schnittstelle* für die Objekte darstellt. Zugleich sieht keiner dieser Konstruktoren Felder vor — genau wie in Abschnitt 11.1 liegen die Daten (hier eher: die Datenstrukturen), die die mit den Konstruktoren erzeugten Objekte repräsentieren, in den jeweiligen Funktionsabschlüssen. So erhalten wir

```JS
> new Empty().concat(new Single(1))
Single {
  isEmpty: [Function (anonymous)],
```

[84] Tatsächlich codiert es das String-Monoid, wie wir in Abschnitt 21.1 sehen werden.

```
  concat: [Function (anonymous)],
  contains: [Function (anonymous)]
}
```

und

```
> new Single(0).concat(new Single(1).concat(new Single(2))).contains(1)
true
```

Dabei werden die beim Aufruf der Konstruktoren `Single` und `Concat` übergebenen Argumente in den Abschlüssen aller Methoden des als Ergebnis jeweils gelieferten neuen Objekts eingefangen. Da so auch `this` eingefangen wird, wird in den Abschlüssen eine rekursive Datenstruktur aufgebaut (wie beispielsweise durch die *Terme* `Single(Empty, 1)` und `Concat(Single(0), Concat(Single(1), Single(2))` repräsentiert[85]), die sich nur über Queries (hier: `isEmpty` und `contains`) inspizieren lässt. Echte *Datenkapselung* also; tatsächlich haben wir es hier mit einer speziellen Implementierung eines *abstrakten Datentyps* zu tun, wie wir in Abschnitt 28.1 sehen werden.

leichte Erweiterbarkeit durch neue Varianten

Ein Charme dieses Ansatzes ist, dass sich die Menge der Konstruktoren damit leicht erweitern lässt. So kann man einen Konstruktor

```
function Any() {
  this.isEmpty = () => false
  this.contains = e => true
  this.concat = l => this
}
```

hinzufügen, dessen gelieferte Objekte eine Art Wildcards repräsentieren (nützlich beispielsweise im Pattern matching von Strings), die ausschließlich über ihre Methoden definiert sind und dafür keine gekapselte Repräsentation brauchen. Tatsächlich weist die Implementierung von Listen mit den Konstruktoren `Empty`, `Single`, `Concat` etc. eine gewisse Ähnlichkeit mit *Summentypen* auf, wie man sie von *algebraischen Datentypen* (Abschnitt 27.3.4) kennt.

Erweiterung der Schnittstelle vergleichsweise aufwendig

Weniger einfach ist jedoch das Hinzufügen neuer Methoden, denn damit muss man alle Konstruktoren anpassen. Dies ist jedoch in der objektorientierten Programmierung nicht ganz so dramatisch, wenn man Vererbung einsetzen kann: So lassen sich die Varianten `Empty`, `Single`, `Concat` und `Any` per Vererbung von einer abstrakten Variante `List` ableiten, die Default-Methodendefinitionen vorsieht, die von den davon abgeleiteten Varianten nach Bedarf überschrieben werden können.

[85] Mit *Termen* sind hier syntaktische Strukturen gemeint, die die Form von Funktionsanwendungen haben, die jedoch nicht ausgewertet werden, sondern selbst Werte (Daten) sind. *Funktoren* wie hier `Empty`, `Single` und `Concat` stehen dann für *uninterpretierte Funktionen*, wie man sie beispielsweise aus PROLOG oder funktionalen Sprachen wie ML oder HASKELL kennt; sie werden auch *Termkonstruktoren* oder nur *Konstruktoren* genannt, sind aber nicht mit den Konstruktoren der objektorientierten Programmierung zu verwechseln, die typischerweise mit `new` aufgerufen und in jedem Fall interpretiert (ausgeführt) werden.

Dies in JAVASCRIPT ohne Klassensyntax hinzuschreiben ist aber länglich, weswegen ich das Beispiel für Kapitel 24 aufgehoben habe.

***keine eindeutige Repräsentation* von Werten**

Ein weiterer Nachteil dieses Ansatzes ist, dass die damit erzeugten internen Datenstrukturen den Vorgang ihrer Erzeugung einfangen, obwohl dieser für ihre Verwendung in den meisten Fällen irrelevant ist. So sollten die durch

```
new Single(0).concat(new Single(1).concat(new Single(2)))
```
JS

und

```
(new Single(0).concat(new Single(1))).concat(new Single(2))
```
JS

erzeugten Listen (deren Repräsentation als Terme mit `Concat(Single(0), Concat(Single(1), Single(2)))` und `Concat(Concat(Single(0), Single(1)), Single(2))` gegeben ist) zwar gleichwertig sein (die Konkatenation von Listen oder Strings ist schließlich eine assoziative Operation), sind aber in ihrer *internen Repräsentationen* ungleich. Dabei haben Listen (im Gegensatz zu den von Cook in seinem Beispiel verwendeten Mengen) eine eindeutige Normalform, nämlich die als flache Folge ihrer Elemente. Gleichheit darf hier also nicht auf Basis dieser internen Repräsentation definiert sein, sondern nur auf Basis einer solche Normalform, wofür Listen dann aber weitere Methoden brauchen. Spätestens dann stellt sich die Frage, ob es nicht grundsätzlich günstiger ist, die Listen gleich in ihrer Normalform zu repräsentieren (eine negative Antwort finden Sie in Abschnitt 21.4).

Zahlen als *funktionale Objekte*

Mit demselben Ansatz lassen sich auch die Codierungen natürlicher Zahlen aus Abschnitt 3.3.10 nachempfinden. Zunächst die *Peano-Zahlen* mit Addition:

```
> function Zero() {
.   this.add = m => m
. }
> function Succ(n) {
.   this.add = m => new Succ(n.add(m))
. }
> var n0ll = new Zero()
> var eins = new Succ(n0ll)
> var zwei = new Succ(eins)
> eins.add(eins)
Succ { app: [Function (anonymous)], add: [Function (anonymous)] }
```
JS

Allerdings sieht man hier wieder nicht viel vom Ergebnis. Wir können aber die beiden Konstruktoren um eine Funktion `app` mit zwei Parametern `s` und `z` ergänzen und so die *Church-Codierungen* aus Abschnitt 3.3.10 simulieren:

```
function Zero() {
  this.app = (s, z) => z
  this.add = m => m
}
function Succ(n) {
  this.app = (s, z) => s(n.app(s, z))
```
JS

```
460   this.add = m => new Succ(n.add(m))
461 }
```

Wir erhalten dann mit obigen Konstanten für Zahlen

```
462 > zwei.app(x => x + 1, 0)                                                    JS
    2
463 > eins.add(eins).app(x => x + 1, 0)
    2
```

Die originalen Church-Codierungen (mit der originalen Addition, die ich auch schon eingangs dieses Kapitels verwendet habe) lassen sich auch auf diese Weise nachbauen, aber darauf will ich hier verzichten.

12 Reflektion

Wie wir bereits an mehreren Stellen gesehen haben, kann ein JavaScript-Programm nicht nur die Struktur seiner Objekte erkennen, sondern diese auch verändern, indem es Felder und Methoden zu Objekten hinzufügt oder löscht. Man nennt den Umstand, dass ein Programm Einsicht in seinen eigenen Aufbau hat und diesen auch verändern kann, **Reflektion**. Reflektion ist eine Form der *Metaprogrammierung*; zu ihr gehört auch die sog. *Interzession*, mit der einen Programm auch seinen Ablauf erkennen und auf nicht unmittelbar in ihm vorgesehene Weise ändern kann. Das ist z. B. der Fall, wenn ein Programm im Nachhinein Zugriffsmethoden für Felder einfügt und so den Zugriff auf die Felder abzufangen erlaubt.

Kann ein Programm seinen Aufbau nur erkennen, ohne eine Möglichkeit der Änderung, spricht man auch von *Introspektion*. Ein klassisches Anwendungsbeispiel für die Introspektion ist die Serialisierung von Objekten, wie sie z. B. beim Debuggen (Anzeige von komplexeren Objektstrukturen auf der Konsole) oder beim Persistieren von Daten (Speichern in einer Datei) gebraucht wird.

12.1 Serialisierung 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. Tatsächlich ist die Syntax der Objektliterale von JavaScript Pate ein weit verbreitetes Datenaustauschformate, JSON (für JavaScript Object Notation) genannt.

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

```
464 > var o = new Person('Hein', 35)                                           JS
465 > for (const name in o) console.log(name + ': ' + o[name])
```

```
name: Hein
alter: 35
```

alle Eigenschaften (Name/Wert-Paare) des Objekts `o`, die als `enumerable` attributiert sind, auf der Konsole ausgeben.[86] Um Objekte vollständig zu serialisieren, fehlt hier allerdings die rekursive Serialisierung der Objekte, die Werte von Eigenschaften des zu serialisierenden Objekts sind. Das erledigt die folgende Funktion in einem ersten Ansatz:

```
466 function serialisiere(o) {                                                    JS
467   if (typeof o !== 'object' || o === null)
468     return typeof o === 'string' ? `"${o}"` : String(o)
469   let s = '{ '
470   for (const name in o) s += `${name}: ${serialisiere(o[name])}, `
471   if (s.length > 2) s = s.slice(0, -2) // entfernt das letzte ", "
472   s += ' }'
473   return s
474 }
```

Jedoch 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.

12.2 Selbstmodifikation

Während Introspektion in JAVASCRIPT durch die Wahl der Repräsentation von Objekten durch eine dynamische Datenstruktur (einen Container von der Art eines Dictionaries) automatisch gegeben ist (und JAVASCRIPT somit den Unterschied zwischen einer „normalen“, nicht reflexiven Programmierung und Reflektion zumindest in Hinsicht auf Objekte weitgehend auflöst), ist die Möglichkeit, ein Programm sich selbst neue Programmelemente hinzufügen zu lassen, von einer anderen Qualität. Aber auch das ist in JAVASCRIPT kein Problem, wie das folgende Beispiel zeigt:

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

Hier wird aus einzelnen Strings, die für die Parameter und den Rumpf einer Funktion stehen und die noch nicht einmal als Literale gegeben sein müssen, eine neue Funktion erzeugt, die sich von einer durch ein gleichwertiges Funktionsliteral entstandenen nicht unterscheidet. Dabei ist ein Funktionsliteral ein Teil des durch das Programm selbst unveränderlichen Programmtexts und hat keine variablen Anteile wie etwa ein Objektliteral. Das funktioniert natürlich nur, weil JAVASCRIPT-Programme zur Laufzeit Zugriff auf den Compiler bzw. Interpreter haben.

[86] Wie zentral diese Fähigkeit für JAVASCRIPT ist, erkennt man daran, dass es hierfür eine eigene For-Schleife (for-in) vorsieht (bislang hatten wir nur for-of verwendet).

Es geht in JavaScript aber sogar noch freier:

Auswertung beliebiger Strings

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

Eine derartig unbeschränkte Reflektion 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.

13 Das Wesen von Objekten

In der objektorientierten Programmierung ist der Begriff des Objekts unauflösbar an den Begriff der Identität gebunden: Zwei Objekte können allein aufgrund ihrer Identität unterschieden werden, auch wenn sie sich ansonsten völlig gleichen. Dies ist kein Implementierungsdetail, sondern Wesenskern der objektorientierten Programmierung: Es grenzt sie insbesondere von der funktionalen Programmierung ab, in der zwei auf die gleiche Weise konstruierte — und damit gleich zusammengesetzte — Werte derselbe Wert und somit ununterscheidbar sind (sie sind tatsächlich nur ein Wert).

Damit geht aber einher, dass nicht alle Funktionen in der objektorientierten Programmierung frei von Nebeneffekten sein können: Die Funktion zur Objekterzeugung (`new`) muss bei jedem Aufruf ein neues Objekt mit eigener Identität zurückgeben, was allein schon dem Gedanken der *referenziellen Transparenz*, und damit einem Wesenskern der funktionalen Programmierung, widerspricht. Dazu muss diese Funktion aber einen internen Zustand haben, der sich mit jedem ihrer Aufrufe ändert. Das Privileg des Zustands auch Objekten einzuräumen ist also kein Tabubruch mehr, sondern mit den Voraussetzungen der Objektorientierung konsistent, und tatsächlich gehört der Umstand, dass Objekte i. A. veränderlich sind, ebenfalls zu ihrem Wesen.

Die dritte Eigenschaft, die Objekte auszeichnet, ist, dass ihnen ein Verhalten zugeordnet werden kann, das von ihrem jeweiligen Zustand abhängt. Die aus der nicht objektorientierten Programmierung bekannte Aufteilung eines Programms in Daten- und Funktions- oder Prozedurdefinitionen, nach der die Daten (spätestens über die Einführung von lokalen Variablen) den Funktionen bzw. Prozeduren untergeordnet sind, wird hier umgekehrt, indem die Funktionen, Methoden genannt, Objekten als Datencontainern untergeordnet werden. Daraus ergibt sich ein prinzipiell anderer Aufbau von Programmen, der die Betrachtung der Objektorientierung als eigenständiges Paradigma rechtfertig.

Damit aber über die Gliederung eines Programms in Objekte die gleiche Strukturierungstiefe erreicht werden kann wie mit Prozeduren und Funktionen, deren Definitionen die Definitionen anderer Prozeduren und Funktionen enthalten können, müssen auch Objektdefinitionen geschachtelt werden können. Diese Form des Containments (nach der ein Objekt echter Teil eines anderen, also von ihm existenzabhängig ist), war uns bislang aber noch nicht begegnet — statt-

dessen haben wir Objekte nur in lose Beziehung zueinander gesetzt, wodurch die Struktur flach bleibt. Containment von Objekten ist ein wesentliches Merkmal der Programmiersprache SCALA, mit der wir uns in Lektion 5 befassen werden.

Die objekt-funktionalen Programmierung komplettiert die objektorientierte Programmierung, indem in ihr auch Funktionen als Objekte aufgefasst werden. Das daraus resultierende unipolare Weltbild mag zunächst bezwingend erscheinen, ist jedoch auf fundamentale Weise fehlerhaft: Objekte werden nicht durch andere Objekte beschrieben, sondern durch Werte, für die es keine Identität, sondern nur Gleichheit gibt. In einer rein objektorientierten Welt müssen Werte aber über explizite Definitionen von Gleichheit simuliert werden, was zwangsläufig zu Brüchen führt. Damit werden wir uns in Kapitel 20 ausführlicher befassen.