3 Core JavaScript

Nachdem im letzten Kapitel eine Reihe von JavaScript-Laufzeitumgebungen vorgestellt wurden, werden im folgenden Kapitel die Kernfeatures der Sprache JavaScript erläutert. Diese Kernfeatures ähneln verbreiteten Sprachen wie Java. Trotzdem sollte sie auch ein erfahrener Programmierer zumindest überfliegen, um Unterschiede zu bekannten Sprachen zu entdecken. Eine ausführliche Referenz der Kernfeatures von JavaScript befindet sich im Anhang dieses Buchs.

3.1 Skriptsprachen

Eine Skriptsprache wird dazu verwendet, um ein bereits bestehendes System zu verändern, anzupassen oder um wiederkehrende Abläufe zu automatisieren. In einem solchen System existieren bereits Funktionalitäten. In der Regel lassen sie sich entweder über ein User Interface oder über Befehle aufrufen. Wenn das System seine Funktionen über eine programmierbare Schnittstelle anbietet, dann ist es »skriptbar« bzw. skriptfähig. Ein gutes Beispiel für ein Skript ist ein Shell-Skript, das einem lästige Wartungsaufgaben abnimmt.

Bei JavaScript steckt der Charakter der Skriptsprache bereits im Namen. Allerdings wurde JavaScript nicht entwickelt, um Java zu skripten: JavaScript wurde entwickelt, um eine HTML-Seite in einem Browser zu verändern.

Während der Browser Schnittstellen zum Dokumentenbaum und zum Netzwerkprotokoll bietet, beinhaltet die Sprache JavaScript selbst nur die Teile, die unabhängig vom Browser sind. JavaScript selbst ist also nicht abhängig vom Browser und kann auch außerhalb des Browsers verwendet werden.

Egal, ob man in JavaScript im Browser, in einer nativen Applikation oder auf dem Server entwickeln möchte – man sollte die Kernkonzepte der Sprache verstehen. Die Kernkonzepte von JavaScript werden als Core JavaScript bezeichnet. Core JavaScript wird von allen JavaScript-Laufzeitumgebungen implementiert. Auch Adobes Flash baut mit ActionScript auf Core JavaScript auf.

Obwohl JavaScript eine Skriptsprache ist, bietet sie viele Programmierparadigmen. Man kann mit JavaScript imperativ oder funktional, prototypisch oder objektorientiert programmieren. Diese verschiedenen Paradigmen werden in diesem Buch teils in eigenen Kapiteln erläutert.

3.2 Typen und Werte

JavaScript kommt mit wenigen Typen aus. Diese Typen lassen sich direkt im Programmcode verwenden.

3.2.1 Schwache Typisierung

JavaScript ist eine nur schwach typisierte Programmiersprache. Es kennt nur wenige Basistypen: Zahlen (number), boolesche Werte (boolean) und Zeichenfolgen (string). Neben diesen Basistypen gibt es zwei besondere Werte: null und undefined.

Null ist ein Schlüsselwort, das einen null-Wert ausdrückt. Eine Variable hat dann den Wert null, wenn ihr noch kein (anderer) Wert zugewiesen wurde. Null selbst ist (in meinen Augen: fälschlicherweise) ein Objekt.

Undefined ist ein Schlüsselwert, das zum Ausdruck bringt, dass z.B. eine Variable noch nicht definiert ist. Undefined selbst hat keinen Typ, d.h. der Typ von undefined ist ebenfalls undefined.

3.2.2 Literale

Man kann in JavaScript direkt Werte der Basistypen (number, boolean, string) verwenden, ohne dass man sie vorher einer Variablen (dazu später mehr) zugewiesen haben muss. Diese Werte, die direkt verwendet werden können, nennt man Literale. Literale werden also zur Darstellung von Basistypen wie Zahlen oder Strings (Zeichenketten) verwendet.

Literale sind unveränderlich. Man kann den Wert eines Literals nicht verändern, da das Literal selbst der Wert ist.

Anders als in vielen anderen Programmiersprachen besitzen Literale, obwohl sie keine Objekte sind, einen Satz von Standardmethoden. Allerdings lassen sich Standardmethoden nicht direkt auf den Literalen anwenden, sondern erst auf Variablen der meisten Literale.

Literale kann man von Objekten (dazu später mehr) dadurch unterscheiden, dass ihre Typnamen mit Kleinbuchstaben benannt werden. Objekte werden per Konvention durch einen führenden Großbuchstaben gekennzeichnet.

3.2.3 Typeof-Operator

Um zu erkennen, von welchem Typ ein Wert oder eine Variable ist, gibt es in JavaScript den `typeof`-Operator:

```
typeof "Hallo Welt" // string
typeof 1234 //number
typeof true // boolean
typeof {"name": "Peter"} // object
typeof [1,2,3,] // object
```

Der `typeof`-Operator lässt sich allerdings nicht verwenden, um festzustellen, von welchem Typ ein Objekt oder eine Funktion ist, denn bei diesen Typen liefert er folgerichtig stets `function` oder `object`.

3.2.4 Typenlose Verwendung

Obwohl JavaScript Typen kennt, ist es keine statisch oder stark typisierte Sprache. Typen werden erst zur Laufzeit ermittelt und – wenn möglich – dynamisch angepasst. Es gibt also anders als in statisch tpyisierten Sprachen keinen `cast`-Operator. JavaScript versucht, Typen in ihrem Kontext implizit zu konvertieren. Daher kann man in JavaScript beispielsweise auch Zahlen mit Zeichenketten vergleichen:

```
123 == "123" // true
```

Oft geht diese dynamische Konvertierung gut und das Programm macht genau das, was der Programmierer beabsichtigt hat. Um sich nicht auf den Zufall verlassen zu müssen, ist für Programmierer, die aus dem statisch typisierten Lager kommen, einiges Umdenken gefordert, um sich mit einer dynamisch typisierten Sprache anzufreunden.

3.3 Imperative Programmierung

Wie die meisten bekannten Sprachen lässt sich mit JavaScript imperativ programmieren. Im imperativen Programmierstil schreibt man ein Programm in einer Abfolge von Anweisungen, die den Zustand des Programms verändern. Der imperative Programmierstil wurde bereits in den 50er-Jahren geprägt. Auf ihm basieren die Programmiersprachen Fortran, Pascal und C, die zu den Vorgängern von JavaScript zählen.

3.3.1 Ausdrücke und Operatoren

JavaScript kennt die aus C und Java bekannten Ausdrücke und Operatoren wie Multiplikation (`*`), Divison (`/`) und Modulo (`%`). Addition und Subtraktion lassen sich wie gewohnt, inklusive bekannter Kurzschreibweisen wie ++ oder --, durchführen. Selbst Zeichenketten lassen sich mit dem Plus-Operator verketten.

Bekannte logische Operatoren wie UND (`&&`), ODER (`|||`) und NOT (`!`) können in JavaScript ebenfalls wie gewohnt verwendet werden. Auch Bit-Operatoren wie das Verschieben von Bits oder die Definition von Bitmasken bergen keine Überraschungen.

Da JavaScript dynamisch typisiert ist, lassen sich diese Operatoren auch zum Casten von Werten oder Variablen verwenden. Da beispielsweise ausschließlich Zahlen multipliziert werden können, lassen sich Strings durch eine Multiplikation in eine Zahl wandeln:

```
typeof ("42"*1) // number
```

3.3.2 Vergleiche

Einen etwas anderen Weg als die meisten gebäuchlichen Sprachen geht JavaScript bei den Vergleichsoperatoren. Gleichheit und Ungleichheit können entweder streng oder normal geprüft werden. Eine normale Prüfung (`==`, `!=`) vergleicht nur Werte, Typen werden zum Vergleich dynamisch zur Laufzeit angepasst. Eine strenge Prüfung (`===`, `!==`) vergleicht außerdem den Typ der Operanden. Da die normale Prüfung Überraschungen in sich bergen kann, bevorzugen viele Entwickler die strenge Prüfung.

```
42 == "42" // true
42 === "42"  // false
```

Bei Ausdrücken, die zu wahr oder falsch valuiert werden, gibt es in JavaScript einige Besonderheiten zu beachten. Ein Ausdruck ist dann unwahr, wenn er folgenden Wert hat:

- false
- null
- undefined
- '' (leerer String)
- 0 (die Zahl Null) oder NaN (der Zahlenwert »Not a Number«)

Ein Ausdruck ist wahr für alle Werte, die nicht unwahr sind.

Achtung: Dies gilt auch für den String »false« oder den numerischen Wert -1!

Die Eigenschaft, dass ein Ausdruck wahr für alle Werte ist, die nicht unwahr sind, lässt sich nutzen, um eine Zahl oder einen String in einen booleschen Wert zu wandeln:

```
!!0 // false
!!1 // true
```

Neben Gleichheit und Ungleichheit kann auch auf Größe (<, <=, >, >=) geprüft werden. Wichtig ist, dass JavaScript die Größe meist lexikalisch und ohne strenge Typprüfung prüft. Dies kann zu merkwürdigen Ergebnissen führen, wenn man nicht bedenkt, dass von einer lexikalischen Sortierung ausgegangen wird. So ist die Zeichenkette »42« größer als die Zeichenkette »411«, die Zahl 42 ist allerdings kleiner als die Zeichenkette »411«.

3.3.3 Variablen

Variablen und Konstanten werden in der Regel Werte (Literale) oder Objekte bzw. Funktionen zugewiesen. Variablen deklariert man mit der `var`-Anweisung.

```
var x;
```

Variablen können bereits in der Deklaration einen Initialwert bekommen, sie lassen sich also gleichzeitig deklarieren und definieren. Solange einer Variablen kein Wert zugewiesen wurde, ist ihr Wert gleich dem besonderen Wert »null«. Ihr Typ wird zu »undefined« ausgewertet.

```
var x;
x == null; // true
typeof x; // undefined
x = 5;
x; // 5
typeof x; // number
```

In einer var-Anweisung können mehr als eine Variable definiert werden. Diese Variablen werden durch Kommas getrennt.

```
var x, y, z;
```

In vielen Coding-Conventions wird verlangt, dass man alle in einem Gültigkeitsbereich verwendeten Variablen zu Beginn des Gültigkeitsbereichs deklariert. Daher ist es nützlich, dass man mehrere Variablen in einer einzigen Zeile deklarieren kann.

3.3.4 Blöcke und Gültigkeit von Variablen

Code lässt sich in JavaScript in Blöcken strukturieren. Blöcke werden wie schon in C und auch in Java in geschweifte Klammern eingeschlossen. Allerdings haben Blöcke weit weniger Bedeutung in JavaScript als in C oder Java.

```
{
   var x = 10;
   print(x); // 10
}
```

JavaScript behandelt nämlich den Gültigkeitsbereich von Variablen anders als C oder Java. Variablen, die in einem Block deklariert wurden, sind nicht nur in diesem Block, sondern in der ganzen Funktion, in der sie deklariert wurden, sichtbar.

```
{
   var x = 10;
   print(x); // 10
}
print(x);  // 10
```

Gültigkeitsbereich für Variablen

Achtung: Anders als in vielen anderen Programmiersprachen, wie Java und C, wird durch einen Block kein neuer Gültigkeitsbereich für Variablen definiert. Das heißt, auch Variablen, die in einem Block definiert wurden, sind außerhalb des Blocks nach ihrer Deklaration sichtbar.

Wenn eine var-Anweisung innerhalb einer Funktion verwendet wird, so wird eine lokale Variable erzeugt. Diese Variable ist nur innerhalb der Funktion sichtbar. Von außen kann nicht auf sie zugegriffen werden. Außerhalb einer Funktion erzeugt die var-Anweisung jedoch eine globale Variable. Globale Variablen sind in der Regel nicht erwünscht, da sie den globalen Namensraum verschmutzen.

Eine Variable kann auch ohne die var-Anweisung erzeugt werden, dann aber stets als globale Variable, was zu einer »Verschmutzung« des globalen Namensraums führt. In ECMAScript 5th Edition ist im Strict Mode daher die Definition einer Variablen ohne die var-Anweisung nicht mehr erlaubt.

Verwende die Var-Anweisung

Eine Variable sollte stets durch die Var-Anweisung definiert werden.

Außer über die var-Anweisung sollen sich lokale Variablen in der kommenden ECMAScript 6th Edition auch mit der let-Anweisung deklarieren und definieren lassen. Die let-Anweisung soll eindeutig zwischen lokalen und globalen Variablen unterscheiden. Zudem hat die let-Anweisung einen Blockgültigkeitsbereich und keinen Funktionsgültigkeitsbereich. Sie kommt daher der Variablendefinition anderer Sprachen wie Java sehr viel näher.

Neben Variablen lassen sich in der kommenden ECMAScript 6th Edition auch Konstanten definieren. Werte werden Konstanten bereits während der Deklaration zugewiesen, sie werden direkt definiert. Diese Werte lassen sich später nicht mehr ändern.

```
const PI = 3.14159265;
```

Wenn man allerdings versucht, den Wert einer Konstanten später zu verändern, dann ignorieren die meisten Laufzeitumgebungen dies einfach, ohne einen Fehler zu verursachen. Dies kann zu schwer zu findenden Bugs im Code führen.

3.3.5 Zahlen

Zahlen, number-Literale, werden als 64-Bit-Fließkommazahlen (64 Bit entsprechen 8 Byte) gespeichert. Dies entspricht dem Typ double in Java. In Zukunft könnte sich dies durchaus ändern. Von der Genauigkeit her bedeutet dies, dass natürliche Zahlen bis zu einer Größe von 15 Zeichen (9E15) als genau angesehen werden können. Bruchzahlen sind nur so genau wie möglich. Dies muss beispielsweise bei Währungsberechnungen berücksichtig werden.

```
0.05 + 0.01 = 0.060000000000000005
```

Es gibt keine Unterscheidung zwischen natürlichen Zahlen und Bruchzahlen.

```
const C1 = 299792458;
const C2 = 2.99792458E8; // C1 == C2
var saldo = -768;
```

Zahlen lassen sich auch hexadezimal ausdrücken.

```
print(0xFF); // liefert 255
```

NaN

`NaN` (Not a Number) ist keine Zahl. Über die Funktion `isNaN` lässt sich überprüfen, ob ein Wert eine Zahl ist. `NaN` lässt sich nicht vergleichen, auch nicht mit sich selbst.

```
var notANumber1 = NaN;
print(notANumber1 == NaN); // liefert false
print(isNaN("Hello")); // liefert true
print(isNaN("3.27E6")); // liefert false
```

Infinity

Eine besondere Zahl ist `Infinity` (Unendlich). Unendlich sind alle Zahlen, die größer sind als der Wertebereich, den eine Zahl annehmen kann. Vergleicht man zwei infinite Zahlen miteinander, so ist das Ergebnis wahr.

```
var infinite = 2E308;
print(infinite); // liefert Infinity
print(Infinity == infinite); // liefert true
print(infinite == infinite * 2); // liefert true
```

Formatierung und Konvertierung

Mit Zahlen möchte man nicht nur rechnen, sondern man möchte die Zahlen auch formatiert als String zur Anzeige bringen. Zur Formatierung von Zahlen gibt es verschiedene Methoden wie `number.toFixed`, `number.toPrecission`, `number.toString` oder `number.toExponential`. Einzelheiten dazu lassen sich dem Anhang dieses Buchs entnehmen. Um Strings (zurück) in Zahlen zu wandeln, gibt es in JavaScript zwei globale Funktionen: `parseInt` und `parseFloat`. Auch diese werden im Anhang näher beschrieben.

3.3.6 Zeichenketten

Zeichenketten werden im String-Literal gespeichert.

```
typeof "Hello World";
/String
```

Als JavaScript entwickelt wurde, war Unicode noch ein 16-Bit-Zeichensatz. Darum ist ein Zeichen in JavaScript 16 Bit breit. Es gibt kein Literal für ein einzelnes Zeichen (Character). Ein einzelnes Zeichen lässt sich durch eine ein Zeichen lange Zeichenkette ausdrücken.

Zeichenketten werden durch einfache oder doppelte Anführungszeichen umschlossen. Als Escape-Zeichen wird der Backslash »\« verwendet.

```
var myString = "Hello World\nJetzt kommt eine neue Zeile";
```

Unicode-Zeichen lassen sich direkt über ihren Unicode-Wert ausdrücken.

```
print("¢" === "\u00A2"); // true
```

Die Länge eines Strings lässt sich über die `length`-Eigenschaft ermitteln. Dies ist keine Methode, sondern eine echtes Attribut.

```
print("Hello World".length); //11
```

Wie alle Literale sind Strings unveränderlich. Methoden, die auf einem String aufgerufen werden, verändern diesen also nicht, sondern geben einen neuen String zurück.

Die Methoden des String-Literals entsprechen den Methoden des String-Objekts, da intern das Literal zu einem Objekt wird, wenn eine Methode aufgerufen wird.

3.3.7 Boolesche Werte

Das Boolean-Literal hat zwei Werte: `true` und `false`.

```
typeof true;
/boolean
```

3.3.8 Arrays

Ein Array ist eine lineare Liste von Werten, bei denen über Positionsangaben auf einzelne Werte zugegriffen werden kann. Man kann in einem Array Werte unterschiedlicher Typen mischen.

```
var planets = ["Merkur", "Venus", "Erde", "Mars"];
print(planets[2]); // Erde
```

Array-Literal und Array-Objekt

Array-Literale erzeugen Array-Objekte. Das heißt, eine durch ein Array-Literal definierte Variable ist nicht vom Typ »array«, sondern vom Typ »object«.

```
typeof planets // object
```

Dies ist ein typisches Problem bei der Verwendung von Arrays. Oft wird in einer Methode als Eingabeparameter ein Array erwartet, man bekommt aber ein Objekt (z.B. einen String) oder es wird ein Objekt (z.B. einen String) erwartet und man bekommt ein Array. Es obliegt dem Programmierer zu prüfen, ob er ein Array oder ein anderes Objekt bekommen hat.

Der typeof-Operator ist hier nicht hilfreich, da dieser sowohl bei einem String-Objekt als auch bei einem Array schlicht object zurückliefert. Zu prüfen, ob das übergebene Objekt die length-Eigenschaft hat, funktioniert leider auch nicht, denn Array-ähnliche Objekte wie Strings haben auch diese Eigenschaft. Man muss also zusätzlich prüfen, ob der Konstruktor des übergebenen Objekts der Array-Konstruktor ist:

```
function isArray(value) {
   return (value && // value ist defined
   typeof value === "object" && // value ist ein Objekt
   value.constructor === Array) // Konstruktor ist Array
}
print(isArray(["eins", "zwei", "drei"])); // true
print (isArray("Hello")); // false
```

toArray-Hilfsmethode

Manchmal ist es auch egal, ob das übergebene Objekt ein Array oder ein String ist. Array-ähnliche Objekte wie Strings lassen sich mit einer Hilfsmethode, die sich u.a. in der jQuery-Library von John Resig befindet, in Arrays wandeln. Inhalte von Arrays selbst werden durch diese Hilfsmethode nicht verändert:

```
function toArray(value) {
   return Array().slice.call(value, 0);
}
print(toArray("Hello")); // H,e,l,l,o
print(toArray(47)); // empty
print(toArray(["eins", "zwei", "drei"])); // eins,zwei,drei
```

3.3.9 Reguläre Ausdrücke

Reguläre Ausdrücke in JavaScript entsprechen im Wesentlichen den regulären Ausdrücken in Perl. Reguläre Ausdrücke lassen sich in JavaScript direkt als Literal angeben. Sie werden durch einen Slash eingeleitet und auch beendet.

```
var regexp = /\((\d*)\)/;
```

Allerdings gibt es keinen Basistyp regexp. Ein solches Literal erzeugt eine Funktion.

```
typeof /\((\d*)\)/; // function
```

Da reguläre Ausdrücke eine eigene Sprache bilden, können sie sehr komplex werden. Darum erfolgt an dieser Stelle nur eine kurze Einführung.

Reguläre Ausdrücke werden verwendet, um Zeichenketten zu validieren bzw. diese Zeichenketten auf Muster abzubilden. Um eine Zeichenkette zu validieren, muss man ein Muster (Pattern) definieren, das einem Suchkriterium entspricht. Dieses Suchkriterium kann man dann auf eine Zeichenkette anwenden.

Muster werden aus String-Literalen und Metazeichen gebildet. Der oben bereits definierte reguläre Ausdruck sucht in einer Telefonnummer die Vorwahl.

Um zu testen, ob eine Zeichenkette eine Vorwahl enthält, lässt sich die `string.search`-Methode verwenden:

```
var regexp = /\([\d\w]*\)/;
print("Telefon: (040) 55555".search(regexp)); //9
```

Der reguläre Ausdruck sucht eine öffnende Klammer »\(«, gefolgt von beliebig vielen Zahlen »\d« oder Leerzeichen »\w« ausgezeichnet durch »*«, gefolgt einer schließenden Klammer »/)«.

»*«, »\d« und »\w« sind Metazeichen.

Auf die Zeichenkette »Telefon: (040) 55555« erfolgt eine Abbildung des regulären Ausdrucks. Die `string.search`-Methode liefert eine positive Position zurück, an der das gesuchte Muster beginnt. Also enthält die Zeichenkette eine Vorwahl.

Um zu überprüfen, ob eine Zeichenkette ausschließlich eine Vorwahl enthält, muss der reguläre Ausdruck durch weitere Metazeichen ergänzt werden.

```
var vorwahl = /^\([\d\w]*\)$/;
print("Telefon: (040) 55555".search(vorwahl)); //-1
print("(404)".search(vorwahl)); // 0
```

Das Metazeichen »^« drückt aus, dass vom Anfang, das Metazeichen »$«, dass bis zum Ende der Zeichenkette der reguläre Ausdruck abgebildet werden muss.

Neben Metazeichen gibt es Flags, die einen regulären Ausdruck unabhängig von Groß-/Kleinschreibung machen oder ihm Mitteilen, dass über Zeilengrenzen hinweg abgebildet werden soll. Diese Flags werden an das Ende des regulären Ausdrucks angehängt.

3.3.10 Kommentare

Code lässt sich durch Kommentare leichter verständlich machen. Es gibt zwei Arten von Kommentaren: Blockkommentare und Zeilenkommentare.

Blockkommentare

Blockkommentare können über Zeilen hinweg laufen. Sie werden durch /* */ eingeschlossen. Diese Zeichenfolge wurde gewählt, da sie in gewöhnlichem Programmcode nur sehr selten vorkommt. Trotzdem ist diese Zeichenfolge nicht gänzlich ausgeschlossen.

```
/*
var matches = /\d*/.match("1234");
*/
```

Dieses Listing führt zu einem Syntaxfehler, da in regulären Ausdrücken (siehe oben!) durchaus diese Zeichenfolge vorkommen kann.

```
$ v8 blockcomments.js
blockcomments.js:2: SyntaxError: Unexpected token .
var matches = /\d*/.match(1234);
^
SyntaxError: Unexpected token .
```

Zeilenkommentare

Zeilenkommentare gelten nur bis zum nächsten Zeilenumbruch. Sie beginnen mit einem Double-Slash //. Einige Autoren empfehlen, lediglich Zeilenkommentare zu verwenden.

```
// Folgender Codeblock demonstriert den Zeilenkommentar.
// Es wird der Wert 5 ausgegeben, da das Inkrement auskommentiert wurde.
var a = 5;
// a++;
print(a);
```

Für reine Dokumentationszwecke sind jedoch Blockkommentare lesbarer.

```
/*
   Folgender Codeblock demonstriert den Zeilenkommentar.
   Es wird der Wert 5 ausgegeben, da das Inkrement auskommentiert wurde.
*/
var a = 5;
// a++;
print(a);
```

3.3.11 Tokens und Whitespaces

Der Quelltext eines JavaScript-Programms wird in eine Reihe von Tokens, Kommentaren und Whitespaces gewandelt. Der Interpreter wertet diese von links nach rechts aus.

Whitespaces, also Leerzeichen, Tabs und Zeilenumbrüche, werden verwendet, um einzelne Tokens voneinander zu trennen. Darüber hinaus haben sie keine Bedeutung. Man setzt sie ein, um Code zu strukturieren.

Eine Anweisung kann man mit einem Semikolon abschließen. Dieses Semikolon ist allerdings optional. In Fällen, in denen Semikola fehlen, wird stets die längst mögliche Sequenz von Zeichen interpretiert. Daher führt folgendes Skript, wenn man es nicht in einer REPL verwendet, zu einer unerwarteten Ausgabe: `1.625`.

```
var a = 5 / 8
-8 + 9
print(a) // 1.625
```

Die Zeilen (1) und (2) werden von der Laufzeitumgebung nämlich zu einer einzigen Zeile zusammengefasst und dann wie folgt interpretiert.

```
var a = 5 / 8 - 8 + 9;
print(a);
```

Es gibt zwar einige Programmierer, die einen möglichst minimalen Stil pflegen und auf alle überflüssigen Zeichen verzichten. Trotzdem empfiehlt es sich, auch wenn Semikola optional sind, diese zur besseren Lesbarkeit des Codes und zur Vermeidung von Fehlern zu verwenden.

3.3.12 Kontrollstrukturen

Kontrollstrukturen und Schleifen sind – neben Funktionen und Prozeduren – die wesentlichen Merkmale der strukturierten Programmierung. Wie auch Java und C bietet JavaScript die bekannten Kontrollstrukturen wie die `if-else`-Anweisung, die mit Fortran bereits in den 50er-Jahren erfunden wurde.

If-Anweisung

Wenn der Ausdruck in der `if`-Anweisung wahr ist, dann wird der `if`-Block ausgeführt, ansonsten der `else`-Block.

```
if (x === y) {
   // Then-Block
   print("x ist y");
} else {
   // Else-Block
   print("x ist ungleich y");
}
```

Switch-Anweisung

Mehrfachverzweigungen werden durch die `switch`-Anweisung eingeleitet. Die `switch`-Anweisung kann eine beliebige Auswahl von bedingten Auswahlblöcken enthalten.

```
switch (x) {
   case 3: {
      print("x ist 3");
      break; // ansonsten wird default auch abgearbeitet
   }
   default: {
      print("Default");
   }
}
```

Welcher Block aufgerufen wird, hängt vom Wert des Auswahlausdrucks ab (hier: x). Der Auswahlausdruck kann eine Zahl oder ein String sein bzw. eine Zahl oder einen String erzeugen. Es werden alle Auswahlausdrücke von oben nach unten abgearbeitet. Um diese Abarbeitung zu unterbrechen, muss man die

break-Anweisung verwenden. Wie auch in anderen Sprachen, die eine switch-case-Anweisung kennen, ist es hilfreich, das Fehlen der break-Anweisung zu kommentieren, da nachfolgende Entwickler sonst vermuten könnten, dass die break-Anweisung vergessen wurde. Trifft kein Auswahlausdruck zu, so wird die optionale default-Anweisung ausgeführt.

3.3.13 Schleifen

Schleifen sind eine Art von Steuerstrukturen, die verwendet werden, um gänzlich auf Sprunganweisungen zu verzichten. Sie wurden mit der Programmiersprache Pascal in den 70er-Jahren eingeführt. JavaScript kennt die üblichen Schleifen: die while-Schleife, die do-while-Schleife und die for-Schleife. Die aus Pascal bekannte repeat-until-Schleife gibt es in JavaScript nicht, sie lässt sich allerdings durch die do-while-Schleife äquivalent nachbilden.

While-Schleife

In der while-Schleife wird der Block der Schleife so lange ausgeführt, wie die Schleifenbedingung wahr ist. Die Schleifenbedingung wird ausgewertet, bevor der Block der Schleife selbst ausgeführt wird.

```
var x = 0;
while (x < 5) {
   print(x); // 0, 1, 2, 3, 4
   x++;
}
```

Do-While-Schleife

Die do-while-Anweisung gleicht der while-Anweisung, nur dass die Schleifenbedingung ausgewertet wird, nachdem der Block der Schleife selbst ausgeführt wurde.

```
var x = 0;
do {
   print(x); // 0, 1, 2, 3, 4, …
   x++;
} while (x < 10);
```

For-Schleife

Die am häufigsten verwendete und älteste Schleifenform ist aber sicherlich die for-Schleife. Sie wurde bereits mit Algol in den 60er-Jahren eingeführt. JavaScript kennt, wie die meisten modernen Programmiersprachen, gleich mehrere Formen der for-Schleife.

Die gewöhnliche Form der for-Anweisung hat drei Klauseln: die Schleifeninitialisierung, die Schleifenbedingung und der finale Ausdruck. Zuerst wird die

Schleife initialisiert. Dann wird die Schleifenbedingung ausgewertet. Wenn diese wahr ist, dann wird der Schleifenblock ausgeführt. Sowohl die Initialisierung als auch die Schleifenbedingung sind optional. Bei nicht existierender Schleifenbedingung wird von einer wahren Schleifenbedingung ausgegangen. Nach der Ausführung des Blocks wird der finale Ausdruck ausgewertet. Dieser ist in der Regel ein Inkrement. Auch der finale Ausdruck ist optional.

```
for (var i = 0, j = 100; i < 100; i++, j--) {
   print(i);
   print(j);
};
```

Die zweite Form der `for`-Anweisung (`for-in`-Anweisung) iteriert über Eigenschaftsnamen eines Objekts. Die `for-in`-Anweisung iteriert nicht über eingebaute Objekteigenschaften, sondern nur über selbst definierte Objekteigenschaften, auch die, die eingebaute Eigenschaften überschreiben. Die Reihenfolge, in der über die Eigenschaften iteriert wird, ist nicht festgelegt. Daher kann nicht von einer festen Reihenfolge ausgegangen werden.

Veränderung von Eigenschaften in einer Schleife

Wenn die Eigenschaften in der Schleife hinzugefügt wurden, dann kann man sich nicht darauf verlassen, dass diese Eigenschaften in der Iteration berücksichtigt werden. Daher sollte das Objekt in der Schleife nicht verändert werden.

```
var foobar = {"foo": "hello", "bar": "world"};
for (var i in foobar) {
   print (i + " ist " + foobar[i]);
}
```

Die `for-in`-Anweisung sollte nicht verwendet werden, um über ein Array bzw. ein Objekt, das einem Array ähnelt, zu iterieren, da so auch über alle Eigenschaften des Arrays und nicht nur über die Inidizes iteriert wird. Man sollte für Arrays bzw. Array-ähnliche Objekte stattdessen die gewöhnliche Form der `for`-Anweisung mit einem Schleifenzähler verwenden.

Sprunganweisungen

Die Programmiersprache C führte 1973 die Möglichkeit ein, Schleifen durch eine Sprunganweisung abzubrechen. Auch in JavaScript lassen sich Schleifen durch die `break`-Sprunganweisung vorzeitig beenden. Durch die `continue`-Sprunganweisung lässt sich die aktuelle Iteration abbrechen, die Schleife wird dann mit der nächsten Iteration fortgeführt. Wenn mehrere Schleifen ineinander geschachtelt sind, kann zudem über die `label`-Anweisung festlegt werden, welche Schleife durch die `break`-Anweisung beendet bzw. welche Schleife durch die `continue`-Anweisung fortgeführt werden soll.

```
outerloop: for (var x = 0; x < 100; x ++) {
   innerloop: for (var y = 0; y < 100; y++) {
      print (x + " : " + y);
      if (y > 10) break outerloop;
   }
}
```

3.3.14 Ausnahmebehandlung

Code kann schnell komplex werden, wenn nicht nur der Erfolgsfall, sondern sämtliche Fehlerfälle behandelt werden müssen. Daher gibt es die Technik der Ausnahmebehandlung, der Anwendungscode von Code zur Ausnahmebehandlung trennt. Der Anwendungscode wirft dann, im Fall einer Ausnahme (engl. Exception), die er nicht selbst behandeln kann oder möchte, eine Exception. Diese Exception wandert dann so lange den Stack nach oben, bis sie ein aufrufender Code fängt. Die Ausnahmebehandlung baut man möglichst an der Stelle in den Code ein, an der man am besten auf die Ausnahme reagieren kann. Wenn man nicht auf die Ausnahme reagieren kann, wenn also die Exception nicht gefangen wird, dann bricht das Programm ab.

Die `try`-Anweisung schließt einen Codeblock ein, in dem eine Exception auftreten kann. Wenn in diesem Codeblock z.B. durch einen Laufzeitfehler oder die `throw`-Anweisung eine Exception geworfen wird, dann wird die Ausführung des Blocks abgebrochen. Im `catch`-Block befindet sich der Code, der die Exception behandelt. Die gefangene Exception steht zur Fehlerbehandlung zur Verfügung. Egal, ob eine Exception behandelt wurde oder nicht, der optionale `finally`-Block wird stets ausgeführt, falls er vorhanden ist.

```
try {
   print(x); // x ist nicht definiert
} catch (e) {
   print(e); // ReferenceError: x is not defined
} finally {
   print("Continue"); // Continue
}
```

Durch die `throw`-Anweisung lässt sich eine Exception werfen.

```
var ammount = -10;
try {
   if (ammount < 0) {
      throw("Ammount is negative");
   }
} catch (e) {
   print("An error occured: " + e); // An error occured:
   // Ammount is negative
}
```

Die Exception selbst kann ein Literal (in diesem Beispiel ein String-Literal) oder ein Objekt sein.

3.4 Objektliteral

In JavaScript gibt es Anweisungen, Operatoren und Literale. Alles andere, auch Funktionen und reguläre Ausdrücke, sind Objekte. Selbst für die Basistypen Zahlen, Zeichenketten und boolesche Werte gibt es Objekte, die sich in Form von Literalen auf die Basistypen abbilden lassen. Ein Objekt selbst ist nichts weiter als ein Container für Schlüssel-Wert-Paare. Unter einem Namen (Schlüssel), der entweder ein beliebiger String oder ein gültiger JavaScript-Name[1] sein muss, kann ein beliebiger Wert gespeichert werden. Der Wert kann ein Literal, eine Funktion oder selbst wieder ein ganzes Objekt sein. Diese Art von Objekten hat JavaScript von Self »geerbt«.

Ein Objektliteral wird durch geschweifte Klammern umschlossen. Leider kennzeichnen geschweifte Klammern auch den Anfang eines Blocks, sodass Objektliterale nicht am Anfang eines Ausdrucks verwendet werden können. Eigenschaftsnamen, die meist als Strings dargestellt werden, müssen nicht mit Hochkommas umschlossen werden, sofern der Eigenschaftsname ein gültiger JavaScript-Name ist. Trotzdem ist es guter Stil, auch gültige JavaScript-Namen in Anführungszeichen zu setzen.

```
var animal = {
   "name" : "Tiger"
};
print(animal.name); // Tiger
```

Objekte können selbst auch wieder Objekte enthalten.

```
var neat = {
   "name" : "Rind",
   "sex" : {
      "masculine" : "Stier",
      "female": "Kuh"
   }
}
print(neat.sex.masculine); // Stier
```

Statt über die Punktschreibweise kann auf Eigenschaften eines Objekts auch über die Klammerschreibweise zugegriffen werden.

```
print(neat["sex"]["masculine"]); // Stier
```

Statt einer Zeichenkette lassen sich auch Zahlen als Eigenschaftsnamen verwenden.

```
var many = {
   1: "one",
   2: "two"
}
print(many[1]); // one
```

1. Reservierte Wörter wie undefined sind beispielsweise keine gültigen JavaScript-Namen. Gültige Namen werden im Anhang behandelt.

Auf eine Zahl als Eigenschaftsnamen lässt sich aber nicht über die Punktschreibweise zugreifen, sondern nur über die Klammerschreibweise. Da so der Eindruck eines Arrays entsteht, sollte man auf numerische Eigenschaftbezeichner verzichten.

3.5 Zusammenfassung

Die wichtigsten Kernfeatures der Sprache wurden in diesem Kapitel vorgestellt: von Anweisungen und Operatoren über Kontrollstrukturen bis hin zu Werten und Literalen. JavaScript folgt nicht einem strengen Programmierparadigma wie der Objektorientierung oder der funktionalen Programmierung. Man kann ein komplettes JavaScript-Programm daher auch prozedural wie ein BASIC-Skript schreiben.

Mit dem Grundlagenwissen, das in diesem Kapitel vermittelt wurde, sollten sich die meisten einfachen Probleme imperativ lösen lassen. Ein Entwickler kann so – ohne in die Tiefen von JavaScript einzusteigen – mit einer Teilmenge der Sprache schnell zum Ziel kommen. Dies ist evtl. auch der Grund, warum JavaScript von vielen Entwicklern nicht ernst genommen wird, da es auf eben diese Teilmenge reduziert wird.

Man kann in JavaScript aber nicht nur kleine Programme schreiben, sondern auch große und komplexe Systeme bauen. JavaScript ist so flexibel, dass sich verschiedene Paradigmen abbilden lassen. Je nach Anwendungsfall kann der Entwickler das jeweils passende Programmiermodell wählen. Anspruchsvollere Programmieraufgaben lassen sich beispielsweise funktional oder objektorientiert lösen. In JavaScript lassen sich Muster und Idiome aus anderen Sprachen nachbilden. Dies wird in den folgenden Kapiteln behandelt.