Einleitung

Als in den 70er-Jahren des vergangenen Jahrhunderts die ersten Personal Computer auf den Markt kamen, waren die meisten von ihnen mit einer einfachen Programmiersprache ausgestattet – gewöhnlich mit einer BASIC-Variante –, die zur Interaktion mit dem Computer erforderlich war. Für die technisch Begabten war der Übergang von der einfachen Computernutzung zur Programmierung daher ein natürlicher Schritt.

Die heutigen Computer, die um ein Mehrfaches leistungsfähiger und billiger sind als die Modelle aus den 70er-Jahren, weisen Softwareschnittstellen auf, die eine schicke, grafische Oberfläche haben und mit der Maus und nicht mehr mit einer Sprache bedient werden. Dadurch sind Computer viel zugänglicher geworden, was im Großen und Ganzen eine starke Verbesserung darstellt. Allerdings hat sich dadurch auch eine Kluft zwischen Computerbenutzern und der Welt der Programmierung aufgetan. Hobbyprogrammierer müssen erst einmal eine Programmierumgebung *suchen*, statt dass sie eine vorfinden, sobald sie den Computer einschalten.

Hinter den Kulissen werden unsere Computersysteme jedoch immer noch von verschiedenen Programmiersprachen beherrscht. Die meisten dieser Sprachen sind viel anspruchsvoller als die BASIC-Dialekte der ersten Personal Computer. Die Sprache JavaScript, um die es in diesem Buch geht, ist beispielsweise in jedem handelsüblichen Webbrowser vorhanden.

Programmierung

> *Ich werde nicht diejenigen erleuchten, die nicht begierig sind zu lernen, noch werde ich diejenigen wachrütteln, die sich fürchten, selbst eine Erklärung zu geben. Wenn ich eine Ecke eines Quadrats vorstelle und sie nicht in der Lage sind, mir die drei anderen zu zeigen, werde ich die Erklärungen nicht noch einmal durchgehen.*
>
> Konfuzius

In diesem Buch möchte ich Ihnen nicht nur JavaScript erklären, sondern Sie auch in die Grundprinzipien der Programmierung einführen. Die Programmierung, so hat sich gezeigt, ist eine schwierige Aufgabe. Die Grundregeln sind gewöhnlich einfach und klar. Programme stützen sich zwar auf diese Grundregeln, neigen aber dazu, durch die Einführung ihrer eigenen Regeln und Vielschichtigkeiten sehr kompliziert zu werden. Daher ist Programmierung nur selten einfach und vorhersagbar. Wie Donald Knuth, einer der Gründerväter auf diesem Gebiet, sagte, ist sie eher eine *Kunst* als eine *Wissenschaft*.

Um aus diesem Buch etwas mitnehmen zu können, dürfen Sie es nicht nur passiv lesen. Bleiben Sie aufmerksam, versuchen Sie, den Beispielcode nachzuvollziehen, und lesen Sie nur dann weiter, wenn Sie den gerade behandelten Stoff in ausreichendem Maße verstanden haben.

> *Ein Computerprogrammierer ist der Schöpfer eines Universums, für das er allein verantwortlich zeichnet. In Form von Computerprogrammen können Universen mit unbeschränkter Vielschichtigkeit erschaffen werden.*
>
> Joseph Weizenbaum, *Die Macht der Computer und die Ohnmacht der Vernunft*

Ein Programm ist vieles zugleich: Es ist ein Text, den ein Programmierer eingegeben hat, es ist die lenkende Kraft, die einen Computer dazu bringt, seine Aufgaben zu erfüllen, es ist eine Menge von Daten im Arbeitsspeicher des Computers, die gleichzeitig die Maßnahmen steuern, die an diesem Arbeitsspeicher vorgenommen werden. Die Vergleiche, in denen Programme mit vertrauten Dingen gleichgesetzt werden, hinken gewöhnlich, aber ein zumindest oberflächlich passendes Bild ist das einer Maschine. Die Zahnräder einer mechanischen Uhr passen raffiniert zusammen, und wenn der Uhrmacher etwas taugt, zeigt sie Ihnen viele jahrelang die genaue Zeit an. Die Elemente eines Programms passen auf ähnliche Weise zusammen, und wenn der Programmierer weiß, was er tut, läuft sein Programm, ohne abzustürzen.

Computer sind Geräte, die als Wirte für diese immateriellen Maschinen fungieren. Für sich allein können sie nur stumpfsinnig einfache Dinge tun. Ihre Nützlichkeit besteht darin, dass sie diese Dinge mit unglaublicher hoher Geschwindigkeit erledigen. Ein Programm kann auf raffinierte Weise eine enorme Anzahl solcher einfachen Tätigkeiten kombinieren, um sehr komplizierte Aufgaben auszuführen.

Für einige ist das Schreiben von Computerprogrammen ein faszinierendes Spiel: Ein Programm ist ein Gedankengebäude. Es lässt sich kostenlos bauen, hat kein Gewicht und wächst rasch unter tippenden Fingern. Wenn wir nicht aufpassen, können die Größe und Komplexität jedoch außer Kontrolle geraten, sodass das Programm sogar seinen Autor verwirrt. Das ist das Hauptproblem bei der Programmierung: Programme unter Kontrolle zu halten. Wenn ein Programm funktioniert, ist das schön. Die Kunst der Programmierung besteht darin, seine Vielschichtigkeit zu beherrschen. Ein gutes Programm ist gebändigt und trotz seiner Komplexität einfach.

Viele Programmierer glauben heute, dass sich die Komplexität am besten dadurch beherrschen lässt, dass man nur wenige gut verstandene Techniken in seinen Programmen einsetzt. Sie haben strenge Regeln (»best practices«) darüber aufgestellt, welche Form Programme haben sollen, und die besonders Eifrigen unter ihnen erklären all diejenigen, die diese Regeln brechen, zu *schlechten* Programmierern.

Was für eine Feindseligkeit gegenüber den mannigfaltigen Möglichkeiten der Programmierung! Dadurch versucht man, Programmierung zu einer linearen und vorhersagbaren Aufgabe zu reduzieren und alle bizarren und schönen Programme für tabu zu erklären. Die Palette der Programmiertechniken ist sehr breit, faszinierend in ihrer Vielseitigkeit und zu einem großen Teil noch nicht erforscht. Natürlich lauern überall Schlingen und Fallgruben, die unerfahrene Programmierer zu allen Arten von schrecklichen Fehlern verleiten. Das heißt aber nur, dass wir vorsichtig vorgehen und unseren Verstand gebrauchen müssen. Wie Sie noch sehen werden, gibt es immer wieder neue Herausforderungen und neue Gebiete zu erforschen. Programmierer, die kein Interesse mehr haben, weiterhin zu forschen, werden sicherlich stagnieren, keine Freude mehr an ihrer Arbeit haben und den Wunsch zu programmieren verlieren (und dann werden sie Manager).

Von der Wichtigkeit der Sprache

Zu Beginn der elektronischen Datenverarbeitung gab es noch keine Programmiersprachen. Programme sahen wie folgt aus:

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

Dieses Programm addiert die Zahlen von 1 bis 10 und gibt das Ergebnis aus (`1 + 2 + ... + 10 = 55`). Es könnte auf einem sehr einfachen, hypothetischen Computer laufen. Um die ersten Rechner zu programmieren, war es notwendig, lange Reihen von Schaltern in die richtige Stellung zu kippen oder Löcher in Pappkarten zu stanzen und diese in den Computer einzuspeisen. Sie können sich vorstellen, was für eine mühselige, fehleranfällige Arbeit das war. Selbst das Schreiben eines einfachen Programms erforderte viel Klugheit und Disziplin, und anspruchsvollere Programme waren praktisch unvorstellbar.

Die manuelle Eingabe eines geheimnisvollen Musters aus *Bits* (wie die zuvor gezeigten Einsen und Nullen allgemein genannt werden) gab den Programmierern natürlich das Gefühl, mächtige Zauberer zu sein, was wohl einen wichtigen Beitrag zur Zufriedenheit mit dem Job geleistet hat.

Jede Zeile des Programms enthält eine einzelne Anweisung. Auf Deutsch ließe sich das etwa wie folgt ausdrücken:

1. Speichere die Zahl 0 an Speicherposition 0.
2. Speichere die Zahl 1 in Speicherposition 1.
3. Speichere den Wert von Speicherposition 1 an Speicherposition 2.
4. Subtrahiere die Zahl 11 von dem Wert in Speicherposition 2.
5. Wenn der Wert in Speicherposition 2 die Zahl ist, fahre mit Anweisung 9 fort.
6. Addiere den Wert von Speicherposition 1 zu Speicherposition 0.
7. Addiere die Zahl 1 zum Wert von Speicherposition 1.
8. Fahre mit Anweisung 3 fort.
9. Gib den Wert an Speicherposition 0 aus.

Das ist zwar schon besser lesbar als der Binärbrei, aber immer noch ziemlich unschön. Die Verwendung von Namen statt Zahlen für die Anweisungen und Speicherpositionen schafft bereits ein bisschen Abhilfe:

```
 Set 'total' to 0
 Set 'count' to 1
[loop]
 Set 'compare' to 'count'
 Subtract 11 from 'compare'
 If 'compare' is zero, continue at [end]
 Add 'count' to 'total'
 Add 1 to 'count'
 Continue at [loop]
[end]
 Output 'total'
```

In dieser Form ist nicht mehr so schwer zu erkennen, wie das Programm funktioniert. Können Sie es nachvollziehen? Die ersten beiden Zeilen weisen zwei Speicherpositionen ihre Startwerte zu: `total` wird verwendet, um das Ergebnis der Berechnung aufzubauen, während `count` die Zahl festhält, die wir uns gerade

ansehen. Die Zeilen, in denen `compare` verwendet wird, sind wahrscheinlich die sonderbarsten. Das Programm muss herausfinden, ob `count` gleich 11 ist, um zu entscheiden, ob die Berechnung abgeschlossen ist. Da der Computer sehr primitiv ist, kann er nur prüfen, ob eine Zahl gleich null ist, und aufgrund dieses Vergleichs eine Entscheidung (Verzweigung) fällen. Daher verwendet er die Speicherposition `compare`, um den Wert von `count - 11` zu berechnen, und trifft seine Entscheidung aufgrund dieses Werts. Jedes Mal, wenn das Programm festgestellt hat, dass es noch nicht bei 11 angelangt ist, addiert es in den nächsten beiden Zeilen den Wert von `count` zum Ergebnis und erhöht `count` um 1.

In JavaScript sieht dieses Programm folgendermaßen aus:

```
var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
print(total);
```

Hier sehen wir einige weitere Verbesserungen. Vor allem müssen wir nicht mehr ausdrücklich angeben, in welcher Weise das Programm vor- und zurückspringen soll, denn darum kümmert sich jetzt das Zauberwort `while`. Es führt die Zeilen darunter aus, solange die angegebene Bedingung `count <= 10` wahr ist, die bedeutet: »`count` ist kleiner oder gleich 10.« Wir müssen also keinen temporären Wert mehr erstellen und mit null vergleichen. Das war eine uninteressante Einzelheit, und die Stärke von Programmiersprachen liegt darin, dass sie sich an unserer Stelle um solche uninteressanten Details kümmern.

Wenn uns praktische Funktionen wie `range` und `sum` zur Verfügung stehen, die alle Zahlen eines Bereichs erfassen bzw. die Summe aus einer Menge von Zahlen bilden, sieht das Programm wie folgt aus:

```
print(sum(range(1, 10)));
```

Die Moral von der Geschicht' lautet, dass ein Programm sowohl lang als auch kurz, sowohl unverständlich als auch lesbar ausgedrückt werden kann. Die erste Version des Programms war völlig rätselhaft, während die letzte fast wie Alltagsenglisch gelesen werden kann: »`print` the `sum` of the `range` of numbers from `1` to `10`«, also »*Drucke* die *Summe* des *Bereichs* der Zahlen von *1* bis *10*.« (Wie Sie so etwas wie `sum` und `range` selbst erstellen können, erfahren Sie in einem der späteren Kapitel.)

Eine gute Programmiersprache hilft dem Programmierer dadurch, dass sie ihm abstraktere Ausdrucksmöglichkeiten an die Hand gibt. Sie verbirgt die uninteressanten Einzelheiten, stellt praktische Bausteine bereit (wie das Konstrukt `while`) und erlaubt dem Programmierer meistens auch, neue Bausteine hinzuzufügen (wie die Operationen `sum` und `range`).

Was ist JavaScript?

JavaScript ist die zurzeit am häufigsten verwendete Sprache, um alle möglichen intelligenten (und manchmal nervtötenden) Dinge mit den Seiten im World Wide Web anzustellen. In den letzten Jahren wurde die Sprache auch in anderen Zusammenhängen verwendet. So hat beispielsweise das Framework *node.js*, mit dem sich schnelle serverseitige Programme in JavaScript schreiben lassen, sehr viel Aufmerksamkeit erregt. Wenn Sie sich für Programmierung interessieren, ist JavaScript sicherlich eine der Sprachen, die zu lernen sich lohnt. Selbst wenn Sie nicht viel Webprogrammierung machen, so werden doch einige der Programme, die ich Ihnen in diesem Buch vorführe, in Ihrem Gedächtnis haften bleiben, Sie verfolgen und Einfluss darauf nehmen, wie Sie Programme in anderen Sprachen schreiben.

Bestimmt werden Sie viele *schreckliche* Dinge über JavaScript hören. Viele davon sind wahr. Als ich zum ersten Mal etwas in JavaScript schreiben musste, begann ich die Sprache schnell zu verachten: Sie akzeptierte fast alles, was ich eingab, interpretierte es aber völlig anders, als ich es beabsichtigt hatte. Das lag, wie ich gestehen muss, vor allem daran, dass ich keine Ahnung hatte, was ich eigentlich tat, aber es gibt auch tatsächlich ein Problem: JavaScript entscheidet mit grotesker Freizügigkeit, was zulässig ist und was nicht. Dahinter steckt der Gedanke, die Programmierung in JavaScript für Anfänger so einfach wie möglich zu machen. In Wirklichkeit sorgt es aber vor allem dafür, dass es schwieriger wird, Fehler in Programmen zu finden, da das System nicht mit dem Finger darauf zeigt.

Die Flexibilität dieser Sprache ist jedoch auch ein Vorteil. Sie lässt Raum für viele Techniken, die in strengeren Sprachen unmöglich wären, und wie wir in späteren Kapiteln noch sehen werden, ermöglicht sie es auch, einige ihrer Unzulänglichkeiten auszugleichen. Nachdem ich JavaScript richtig gelernt und eine Weile damit gearbeitet hatte, begann ich die Sprache wirklich zu schätzen.

Trotz des Namens hat JavaScript nur sehr wenig mit der Programmiersprache Java zu tun. Die Bezeichnung wurde weniger aufgrund tiefsinniger Überlegungen, sondern eher aus Vermarktungsgründen gewählt. Als Netscape im Jahre 1995 JavaScript vorstellte, wurde die Sprache Java stark angepriesen und nahm an Beliebtheit zu. Offensichtlich hatte damals jemand einen Einfall, um auf dieser Erfolgswelle mitzuschwimmen. Und jetzt müssen wir mit diesem Namen leben.

Eng im Zusammenhang mit JavaScript steht das sogenannte ECMAScript. Als auch andere Browser als der von Netscape begannen, JavaScript oder etwas Ähnliches zu unterstützen, wurde ein Dokument abgefasst, das genau festlegte, wie ein JavaScript-System zu funktionieren hat. Die in diesem Dokument beschriebene Sprache wurde nach der Organisation, die die Standardisierung vorgenommen hat, ECMAScript genannt. ECMAScript beschreibt eine Allzweck-Programmiersprache und sagt nichts über die Verknüpfung dieser Sprache mit einem Webbrowser aus.

Es gibt verschiedene »Versionen« von JavaScript. In diesem Buch beschreibe ich ECMAScript 3, die erste Version, die weitgehend von verschiedenen Browsern unterstützt wurde. In den letzten Jahren hat es verschiedene Initiativen gegeben, um die Sprache weiterzuentwickeln, aber zumindest für die Webprogrammierung sind solche Erweiterungen nur dann sinnvoll, wenn sie von Browsern allgemein unterstützt werden, wobei die Browserhersteller den Entwicklungen jedoch hinterherhinken. Glücklicherweise sind die neueren Versionen von JavaScript meistens nur Erweiterungen von EMCAScript 3, weshalb all das, was Sie in diesem Buch lesen, auch in Zukunft noch Gültigkeit haben wird.

Die Programme ausprobieren

Wenn Sie den Code in diesem Buch ausführen und mit ihm herumspielen möchten, können Sie *http://eloquentjavascript.net/* aufsuchen und die dort bereitgestellte Onlineumgebung nutzen.

Stattdessen können Sie jedoch auch einfach eine HTML-Datei erstellen, die das Programm enthält, und in Ihren Browser laden. Beispielsweise können Sie eine Datei namens `test.html` mit dem folgenden Inhalt anlegen:

```
<html><body><script type="text/javascript">

var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
document.write(total);

</script></body></html>
```

In den späteren Kapiteln erfahren Sie mehr über HTML und die Art und Weise, wie Browser HTML-Code interpretieren. Beachten Sie, dass die Operation `print` aus dem Beispiel durch `document.write` ersetzt wurde. Wie Sie die Funktion `print` schreiben können, erfahren Sie in Kapitel 10.

Das Buch im Überblick

Die ersten drei Kapitel geben Ihnen eine Einführung in die Sprache JavaScript und zeigen Ihnen, wie Sie grammatikalisch korrekte JavaScript-Programme schreiben. Hier lernen Sie Steuerstrukturen (wie das Wort `while`, das Sie in dieser Einleitung schon gesehen haben), Funktionen (selbst geschriebene Operationen) und Datenstrukturen kennen. Das reicht aus, um einfache Programme zu schreiben.

Die nächsten vier Kapitel bauen auf diesen Grundlagen auf und erläutern fortgeschrittene Techniken, mit denen Sie anspruchsvollere Programme schreiben können, ohne dass dabei ein unverständliches Durcheinander herauskommt. Als Erstes geht es in Kapitel 4 um den Umgang mit Fehlern und unerwarteten Situationen. In den Kapiteln 5 und 6 werden zwei wichtige Vorgehensweisen zur Abstraktion vorgestellt, nämlich die funktionale und die objektorientierte Programmierung. Kapitel 7 zeigt, wie Sie Ihre Programme gliedern können.

Der Schwerpunkt der restlichen Kapitel liegt weniger auf der Theorie, als vielmehr mehr auf den Möglichkeiten, die die JavaScript-Umgebung bietet. In Kapitel 8 wird eine Art »Unter-«Sprache für die Textverarbeitung eingeführt, und die Kapitel 9 bis 12 beschreiben, welche Einrichtungen einem Programm zur Verfügung stehen, wenn es in einem Browser ausgeführt wird. Hier lernen Sie, wie Sie Webseiten bearbeiten, auf Benutzeraktionen reagieren und mit einem Webserver kommunizieren.

Schreibweisen

In diesem Buch steht Text in `nicht proportionaler Schrift` für Programmelemente. Manchmal handelt es sich dabei um eigenständige Fragmente, manchmal aber auch um einzelne Bestandteile eines im Kontext beschriebenen Programms. Programme (von denen Sie jetzt schon einige gesehen haben) werden wie folgt dargestellt:

```
function fac(n) {
  return n == 0 ? 1 : n * fac(n - 1);
}
```

Wenn ich vorführen möchte, was bei der Auswertung einzelner Ausdrücke geschieht, steht der Ausdruck in Fettdruck und das Ergebnis, eingeleitet durch einen Pfeil, darunter:

```
1 + 1
→ 2
```