4 Datenstrukturen: Objekte und Arrays

Zahlen, boolesche Werte und Strings sind die Atome, aus denen Datenstrukturen aufgebaut sind. Für manche Art von Informationen ist jedoch mehr als ein Atom erforderlich. Mithilfe von *Objekten* können wir Werte – einschließlich anderer Objekte – gruppieren, um komplexere Strukturen aufzubauen.

Die Programme, die wir bisher geschrieben haben, waren dadurch beschränkt, dass sie nur einfache Datentypen verarbeiteten. In diesem Kapitel erhalten Sie eine Einführung in grundlegende Datenstrukturen. Mit den hier vermittelten Kenntnissen können Sie schon beginnen, eigene nützliche Programme zu schreiben.

In diesem Kapitel arbeiten wir ein mehr oder weniger realistisches Programmierbeispiel durch und führen dabei nach und nach verschiedene Konzepte ein, wenn wir sie für das vorliegende Problem benötigen. Der Beispielcode baut dabei oft auf Funktionen und Bindungen auf, die zuvor im Text eingeführt wurden.

Die Online-Programmier-Sandbox zu diesem Buch (*https://eloquentjavascript.net/code*) bietet Ihnen die Möglichkeit, Code im Kontext eines bestimmten Kapitels auszuführen. Wenn Sie die Beispiele in einer anderen Umgebung durcharbeiten möchten, sollten Sie zuvor den gesamten Code für dieses Kapitel von der Sandbox-Seite herunterladen.

4.1 Das Wereichhörnchen

Hin und wieder, gewöhnlich irgendwann zwischen 20 und 22 Uhr, verwandelt sich Jacques in ein kleines, pelziges Nagetier mit buschigem Schwanz. Einerseits ist Jacques ganz froh, dass er nicht an klassischer Lykanthropie leidet, denn eine Verwandlung in ein Eichhörnchen verursacht weniger Probleme als die Verwandlung in einen Wolf. Anstatt sich Sorgen darüber zu machen, dass er versehentlich seinen Nachbarn auffressen könnte (na, das wäre vielleicht peinlich!), muss er jetzt jedoch befürchten, dass ihn die Katze des Nachbarn frisst. Nachdem er schon zweimal nackt und völlig orientierungslos auf einem gefährlich dünnen Ast in der Krone einer Eiche aufgewacht ist, verriegelt er jetzt jede Nacht Türen und Fenster seines

Schlafzimmers und streut ein paar Nüsse auf den Boden, um sich beschäftigt zu halten. Damit ist er zwar das Katzen- und das Baumproblem losgeworden, allerdings würde Jacques nur zu gern komplett auf diese Verwandlungen verzichten. Deren unregelmäßiges Auftreten gibt zu der Vermutung Anlass, dass sie durch irgendeinen Reiz ausgelöst werden. Eine Zeit lang hat Jacques vermutet, dass es nur an Tagen geschieht, in denen er sich in der Nähe von Eichen aufgehalten hat. Allerdings hat auch eine strikte Vermeidung von Eichen sein Problem nicht lösen können.

Jacques verfolgt jetzt einen wissenschaftlicheren Ansatz und führt ein Tagesprotokoll all der Dinge, die er tut, mit dem Vermerk, ob er sich an dem jeweiligen Tag verwandelt hat. Er hofft, mithilfe dieser Daten die Umstände eingrenzen zu können, die die Verwandlungen auslösen.

Als Erstes braucht er dazu eine Datenstruktur, um diese Informationen zu speichern.

4.2 Datenmengen

Um mit einer Menge digitaler Daten arbeiten zu können, brauchen wir zunächst eine Möglichkeit, um sie im Arbeitsspeicher darzustellen. Betrachten wir als Beispiel die Zahlen 2, 3, 5, 7 und 11.

Da Strings eine beliebige Länge annehmen können, lassen sich viele Daten darin speichern. Wir könnten daher Strings auf kreative Weise nutzen und `"2 3 5 7 11"` als Darstellung unserer Zahlenmenge verwenden. Das wäre aber umständlich, denn um auf die Zahlen zugreifen zu können, müssen wir sie erst irgendwie aus dem String entnehmen und wieder in Zahlen umwandeln.

Glücklicherweise bietet JavaScript jedoch einen eigenen Datentyp, um Folgen von Werten zu speichern. Dieser Typ wird als *Array* bezeichnet und als eine in eckige Klammern eingeschlossene Liste von Werten geschrieben, die durch Kommata getrennt sind:

```
let listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[2]);
// → 5
console.log(listOfNumbers[0]);
// → 2
console.log(listOfNumbers[2 - 1]);
// → 3
```

Auch zum Abrufen der Elemente in einem Array wird die Schreibweise mit eckigen Klammern verwendet. Steht ein Paar eckiger Klammern unmittelbar hinter einem Ausdruck und enthält selbst einen weiteren Ausdruck, so wird das Element im linken Ausdruck nachgeschlagen, das dem durch den Ausdruck in Klammern gegebenen *Index* entspricht.

Der erste Index eines Arrays ist 0, nicht 1, weshalb das erste Element mit `listOfNumbers[0]` abgerufen wird. Eine mit 0 beginnende Zählung hat in der Technik eine lange Tradition und ist sicherlich sehr sinnvoll, allerdings muss man

sich erst daran gewöhnen. Stellen Sie sich den Index als die Anzahl der vom Anfang des Arrays an zu überspringenden Elemente vor.

4.3 Eigenschaften

In den vorherigen Kapiteln haben Sie schon einige sonderbare Ausdrücke wie `myString.length` (um die Länge eines Strings zu bestimmen) und `Math.max` (die Maximum-Funktion) gesehen. Solche Ausdrücke greifen auf eine *Eigenschaft* eines Werts zu. Im ersten Fall greifen wir auf die Eigenschaft `length` des Wertes in `myString` zu, im zweiten auf die Eigenschaft `max` des Objekts `Math` (bei dem es sich um eine Sammlung von mathematischen Konstanten und Funktionen handelt).

Fast alle JavaScript-Werte weisen Eigenschaften auf. Die einzigen Ausnahmen sind `null` und `undefined`. Wenn Sie bei diesen Nichtwerten versuchen, auf eine Eigenschaft zuzugreifen, erhalten Sie eine Fehlermeldung:

```
null.length;
// → TypeError: null has no properties
```

Die beiden Hauptmöglichkeiten für den Zugriff auf Eigenschaften in JavaScript sind die Schreibweise mit Punkt und die mit eckigen Klammern. Sowohl `value.x` als auch `value[x]` greifen auf eine Eigenschaft von `value` zu – allerdings nicht unbedingt auf dieselbe. Der Unterschied liegt darin, wie `x` interpretiert wird. Das Wort hinter dem Punkt ist der buchstäbliche Name der Eigenschaft, doch der Ausdruck in den eckigen Klammern wird erst *ausgewertet*, um den Namen der Eigenschaft zu bestimmen. Während `value.x` die Eigenschaft `x` von `value` abruft, wird bei `value[x]` erst der Ausdruck `x` ausgewertet und dann das in einen String konvertierte Ergebnis als Eigenschaftenname verwendet.

Wenn Sie also bereits wissen, dass die gewünschte Eigenschaft `color` heißt, schreiben Sie einfach `value.color`. Brauchen Sie aber die Eigenschaft, deren Name der von der Bindung `i` festgehaltene Wert ist, dann verwenden Sie `value[i]`. Eigenschaftennamen sind Strings. Es kann sich dabei um beliebige Strings handeln. Allerdings funktioniert die Punktschreibweise nur bei Namen, die wie gültige Bindungsnamen aussehen. Wenn Sie also auf Eigenschaften mit Namen wie `2` oder `John Doe` zugreifen wollen, müssen Sie die Schreibweise mit eckigen Klammern verwenden, also `value[2]` bzw. `value["John Doe"]`.

Die Elemente eines Arrays werden als dessen Eigenschaften gespeichert, wobei Zahlen als Eigenschaftennamen verwendet werden. Da Sie die Punktschreibweise für Zahlen nicht einsetzen können und Sie gewöhnlich ohnehin eine Bindung verwenden möchten, die den Index festhält, müssen Sie die Klammerschreibweise nutzen.

Die Eigenschaft `length` eines Arrays gibt die Anzahl der Elemente an. Da dieser Eigenschaftenname auch ein gültiger Bindungsname ist und wir ihn im Voraus kennen, verwenden wir, um die Länge eines Arrays herauszufinden, gewöhnlich `array.length`. Schließlich lässt sich das einfacher schreiben als `array["length"]`.

4.4 Methoden

Neben der Eigenschaft `length` enthalten sowohl String- als auch Array-Objekte mehrere Eigenschaften, die Funktionswerte festhalten.

```
let doh = "Doh";
console.log(typeof doh.toUpperCase);
// → function
console.log(doh.toUpperCase());
// → DOH
```

Jeder String verfügt über die Eigenschaft `toUpperCase`. Wird sie aufgerufen, gibt sie eine Kopie des Strings zurück, bei der alle Buchstaben in Großbuchstaben umgewandelt wurden. Es gibt auch die Eigenschaft `toLowerCase`, die umgekehrt vorgeht.

Obwohl beim Aufruf von `toUpperCase` keine Argumente übergeben werden, hat die Funktion trotzdem irgendwie Zugriff auf den String `"Doh"`, also den Wert, dessen Eigenschaft aufgerufen wurde. Wie das funktioniert, erfahren Sie im Abschnitt »***Methoden***« auf ***S. 102***. Eigenschaften, die Funktionen enthalten, werden allgemein als *Methoden* des Werts bezeichnet, zu dem sie gehören. Somit ist `toUpperCase` also eine Methode von Strings.

Das folgende Beispiel führt zwei Methoden vor, die Sie zur Bearbeitung von Arrays verwenden können:

```
let sequence = [1, 2, 3];
sequence.push(4);
sequence.push(5);
console.log(sequence);
// → [1, 2, 3, 4, 5]
console.log(sequence.pop());
// → 5
console.log(sequence);
// → [1, 2, 3, 4]
```

Die Methode `push` fügt Werte am Ende eines Arrays hinzu, während die Methode `pop` den letzten Wert eines Arrays entfernt und zurückgibt.

Diese etwas albernen Namen sind die traditionellen Bezeichnungen für Operationen auf einem *Stack*. In der Programmierung versteht man unter einem Stack (wörtlich »Stapel«) eine Datenstruktur, auf der Sie Werte ablegen und anschließend in umgekehrter Reihenfolge wieder herunternehmen können, sodass das letzte hinzugefügte Element als Erstes wieder entfernt wird. Solche Strukturen sind in der Programmierung üblich. Der Funktionsaufrufstack, den wir auf ***S. 48*** behandelt haben, ist ebenfalls eine Anwendung dieses Prinzips.

4.5 Objekte

Zurück zu unserem Wereichhörnchen! Die Menge der täglichen Protokolleinträge kann zwar als Array dargestellt werden, wobei die einzelnen Einträge aber nicht einfach nur Zahlen oder Strings sind. Stattdessen muss jeder Eintrag eine Liste von Tätigkeiten sowie einen booleschen Wert enthalten, der angibt, ob sich Jacques in ein Eichhörnchen verwandelt hat oder nicht. Am besten wäre es, wenn wir alle diese Angaben zu einem Wert gruppieren und die gruppierten Werte dann in Arrays aus Protokolleinträgen aufnehmen könnten.

Werte vom Typ eines *Objekts* sind beliebige Mengen von Eigenschaften. Eine Möglichkeit, um ein Objekt zu erstellen, besteht darin, einen Ausdruck mit geschweiften Klammern zu verwenden:

```
let day1 = {
  squirrel: false,
  events: ["work", "touched tree", "pizza", "running"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

In den geschweiften Klammern steht eine Liste von Eigenschaften, die durch Kommata getrennt sind. Jede Eigenschaft wiederum hat einen Namen, gefolgt von einem Doppelpunkt und einem Wert. Wird ein Objekt über mehrere Zeilen angegeben, so erhöht eine Einrückung wie in diesem Beispiel die Lesbarkeit. Eigenschaften, deren Namen weder gültige Bindungsnamen noch gültige Zahlen sind, müssen in Anführungszeichen stehen.

```
let descriptions = {
  work: "Went to work",
  "touched tree": "Touched a tree"
};
```

Geschweifte Klammern haben also *zwei* Bedeutungen in JavaScript. Stehen sie am Anfang einer Anweisung, so leiten sie einen Anweisungsblock ein. An allen anderen Stellen dagegen beschreiben sie ein Objekt. Glücklicherweise ist es nur selten sinnvoll, eine Anweisung mit einem Objekt in geschweiften Klammern zu beginnen, weshalb diese Mehrdeutigkeit kein großes Problem darstellt.

Der Versuch, eine Eigenschaft zu lesen, die es gar nicht gibt, resultiert in dem Wert `undefined`.

Mit dem Operator `=` ist es möglich, einem Eigenschaftsausdruck einen Wert zuzuweisen. Existiert die Eigenschaft bereits, wird ihr bestehender Wert dadurch ersetzt. Anderenfalls wird eine neue Eigenschaft erstellt.

Um noch einmal kurz zu unserem Tentakelmodell von Bindungen zurückzukehren: Eigenschaften verhalten sich ebenso. Auch sie *halten Werte fest*, wobei auch andere Bindungen und Eigenschaften dieselben Werte ergreifen können. Sie können sich ein Objekt als einen Kraken mit beliebiger Anzahl von Armen vorstellen, auf die jeweils ein Name tätowiert ist.

Der Operator `delete` schneidet einen dieser Tentakel von dem Kraken ab. Es handelt sich dabei um einen unären Operator, der bei der Anwendung auf eine Objekteigenschaft die benannte Eigenschaft von dem Objekt entfernt. Das wird zwar nicht häufig gebraucht, ist aber möglich:

```
let anObject = {left: 1, right: 2};
console.log(anObject.left);
// → 1
delete anObject.left;
console.log(anObject.left);
// → undefined
console.log("left" in anObject);
// → false
console.log("right" in anObject);
// → true
```

Wird der binäre Operator `in` auf einen String und ein Objekt angewendet, teilt er Ihnen mit, ob das Objekt über eine Eigenschaft des gegebenen Namens verfügt. Der Unterschied dazwischen, eine Eigenschaft auf `undefined` zu setzen und sie tatsächlich zu löschen, besteht darin, dass das Objekt im ersten Fall nach wie vor über die Eigenschaft verfügt (sie hat lediglich keinen interessanten Wert mehr), wohingegen die Eigenschaft im zweiten Fall nicht mehr vorhanden ist, weshalb `in` den Wert `false` zurückgibt.

Um herauszufinden, über welche Eigenschaften ein Objekt verfügt, können Sie die Funktion `Object.keys` verwenden. Wenn Sie ihr ein Objekt übergeben, gibt sie ein Array aus Strings zurück, bei denen es sich um die Eigenschaftennamen des Objekts handelt:

```
console.log(Object.keys({x: 0, y: 0, z: 2}));
// → ["x", "y", "z"]
```

Die Funktion `Object.assign` kopiert alle Eigenschaften eines Objekts zu einem anderen:

```
let objectA = {a: 1, b: 2};
Object.assign(objectA, {b: 3, c: 4});
console.log(objectA);
// → {a: 1, b: 3, c: 4}
```

Arrays sind also lediglich spezialisierte Objekte für die Speicherung von Aufzählungen von Dingen. Wenn Sie `typeof []` auswerten, erhalten Sie `"object"`. Sie können sich Arrays als lange, plattgedrückte Kraken vorstellen, deren Tentakel alle sauber in einer Reihe ausgerichtet und mit Zahlen beschriftet sind.

Jacques' Tagebuch können wir nun als Array von Objekten darstellen:

```
let journal = [
  {events: ["work", "touched tree", "pizza",
            "running", "television"],
   squirrel: false},
  {events: ["work", "ice cream", "cauliflower",
            "lasagna", "touched tree", "brushed teeth"],
   squirrel: false},
  {events: ["weekend", "cycling", "break", "peanuts",
            "beer"],
   squirrel: true},
  /* usw. */
];
```

4.6 Veränderbarkeit

Wir werden nun *wirklich* bald zur richtigen Programmierung kommen. Zunächst aber müssen wir uns noch mit einem weiteren Aspekt der Theorie befassen. Wie Sie gesehen haben, können Objekte verändert werden. Die Arten von Werten, die wir in früheren Kapiteln besprochen haben, etwa Zahlen, Strings und boolesche Werte, sind *unveränderbar*. Es ist nicht möglich, den Wert dieser Typen zu ändern. Sie können sie zwar kombinieren und neue Werte von ihnen ableiten, aber wenn sie einen bestimmten Stringwert haben, dann bleibt er immer derselbe. Der enthaltene Text lässt sich nicht ändern. Wenn Sie einen String mit dem Inhalt `"cat"` haben, ist anderer Code nicht in der Lage, die Zeichen in diesem String etwa in `"rat"` zu ändern.

Objekte dagegen verhalten sich anders. Deren Eigenschaften *können* Sie ändern, weshalb ein Objektwert zu unterschiedlichen Zeiten verschiedene Inhalte aufweisen kann.

Wenn wir zwei Zahlen wie 120 und 120 haben, dann betrachten wir sie als dieselbe Zahl, und zwar unabhängig davon, ob sie auf dieselben physischen Bits verweisen. Bei Objekten ist es jedoch einen Unterschied, ob wir zwei Verweise auf dasselbe Objekt oder zwei verschiedene Objekte mit den gleichen Eigenschaften haben. Betrachten Sie dazu den folgenden Code:

```
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};

console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false

object1.value = 15;
console.log(object2.value);
// → 15
```

```
console.log(object3.value);
// → 10
```

Die Bindungen `object1` und `object2` halten *dasselbe* Objekt fest, weshalb sich die Änderung von `object1` auch auf den Wert von `object2` auswirkt. Diese Bindungen haben dieselbe *Identität*. Die Bindung `object3` dagegen zeigt auf ein anderes Objekt, das zwar ursprünglich die gleichen Eigenschaften hat wie `object1`, aber von ihm unabhängig ist.

Auch Bindungen können veränderbar oder konstant sein, aber das hat nichts damit zu tun, wie sich ihre Werte verhalten. Ein Zahlenwert ändert sich zwar nicht, doch bei einer `let`-Bindung können Sie den Wert ändern, auf den sie zeigt, und damit eine sich ändernde Zahl verfolgen. Ähnliches gilt für eine `const`-Bindung an ein Objekt: Auch wenn sie selbst nicht geändert werden kann, sondern immer auf dasselbe Objekt zeigt, ist es möglich, dass sich der *Inhalt* dieses Objekts ändert.

```
const score = {visitors: 0, home: 0};
// Zulässig
score.visitors = 1;
// Unzulässig
score = {visitors: 1, home: 1};
```

Wenn Sie Objekte mit dem JavaScript-Operator `==` vergleichen, untersucht er die Identitäten. Er gibt daher nur dann `true` zurück, wenn beide Objekte tatsächlich derselbe Wert sind. Ein Vergleich verschiedener Objekte ergibt `false`, auch wenn sie die gleichen Eigenschaften haben. In JavaScript gibt es keine »tiefe« Vergleichsoperation, bei der die Objekte anhand ihrer Inhalte verglichen werden, aber Sie können sie selbst schreiben (was eine der Übungsaufgaben am Ende dieses Kapitels ist).

4.7 Das Tagebuch des Wereichhörnchens

Jacques wirft nun also seinen JavaScript-Interpreter an und richtet die Umgebung ein, die er für sein Tagebuch benötigt:

```
let journal = [];

function addEntry(events, squirrel) {
  journal.push({events, squirrel});
}
```

Das Objekt, das er dem Tagebuch hinzufügt, sieht ein bisschen merkwürdig aus. Anstatt Eigenschaften wie `events: events` zu deklarieren, sind hier nur die Eigenschaftennamen angegeben. Das ist eine Kurzschreibweise, die aber das Gleiche bedeutet. Wenn auf einen Eigenschaftenname in geschweiften Klammern kein Wert folgt, wird dieser Wert aus der Bindung mit demselben Namen übernommen.

Jeden Abend um 22 Uhr – oder am nächsten Morgen, nachdem er vom obersten Brett seines Bücherregals heruntergeklettert ist – zeichnet Jacques also seinen Tag auf:

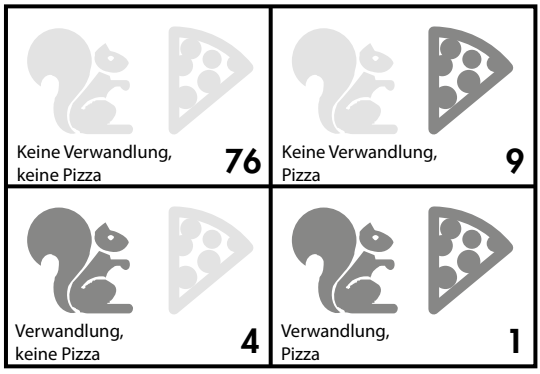
```
addEntry(["work", "touched tree", "pizza", "running",
          "television"], false);
addEntry(["work", "ice cream", "cauliflower", "lasagna",
          "touched tree", "brushed teeth"], false);
addEntry(["weekend", "cycling", "break", "peanuts",
          "beer"], true);
```

Sobald er genügend Datenpunkte gesammelt hat, möchte er statistische Methoden anwenden, um herauszufinden, mit welcher dieser Tätigkeiten seine Eichhörnchenisierung zusammenhängen könnte.

Die *Korrelation* ist ein Maß für die Abhängigkeit zwischen statischen Variablen. In der Statistik ist eine Variable etwas anderes als in der Programmierung. Gewöhnlich haben Sie in der Statistik eine Menge von *Messungen*, bei denen jeweils alle Variablen erfasst werden. Die Korrelation zwischen den Variablen wird normalerweise als Wert aus dem Bereich von –1 bis 1 ausgedrückt. Eine Korrelation von 0 bedeutet, dass kein Zusammenhang zwischen den Variablen besteht. Bei einer Korrelation von 1 ist der Zusammenhang vollkommen – wenn Sie die eine Variable kennen, dann kennen Sie auch die andere. Das gilt ebenfalls bei einer Korrelation von –1, wobei die Variablen dann jeweils das Gegenteil voneinander sind. Ist die eine wahr, so ist die andere falsch.

Um die Korrelation zwischen zwei booleschen Variablen zu berechnen, können wir den *Phi-Koeffizienten* (Φ) verwenden. In diese Formel geht eine Häufigkeitstabelle ein, die angibt, wie oft die verschiedenen Kombinationen der Variablen beobachtet wurden. Das Ergebnis der Formel ist eine Zahl zwischen –1 und 1, die die Korrelation wiedergibt.

Betrachten wir als Beispiel das Ereignis, Pizza zu essen. Das können wir wie folgt in eine Häufigkeitstabelle aufnehmen, wobei die einzelnen Zahlen angeben, wie oft die jeweiligen Kombinationen bei unseren Messungen aufgetreten sind:



Wenn wir die Tabelle n kennen, können wir Φ wie folgt berechnen:

$$\Phi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet}n_{\bullet 1}n_{\bullet 0}}}$$

Wenn Sie jetzt das Buch beiseitelegen wollen, um sich an das zu erinnern, was Sie mal in der Mittelstufe in Mathematik gelernt haben – lassen Sie es sein! Ich will Sie hier nicht seitenlang mit kryptischen Formeln quälen. Diese eine reicht, und selbst die brauchen wir nur, um sie in JavaScript umzuwandeln.

Die Schreibweise n_{01} steht für die Anzahl der Messungen, bei denen die erste Variable (Verwandlung in ein Eichhörnchen) falsch ist (0), die zweite Variable (Pizza) dagegen wahr (1). In unserer Häufigkeitstabelle hat n_{01} den Wert 9.

Der Wert $n_{1\bullet}$ bezieht sich auf die Summe aller Messungen, bei denen die erste Variable wahr ist. In der Beispieltabelle ist das 5. Ebenso bedeutet $n_{\bullet 0}$ die Summe aller Messungen, in denen die zweite Variable falsch ist.

Für unsere Pizza-Tabelle ist der Term über dem Bruchstrich (der Dividend) also $1 \times 76 - 4 \times 9 = 40$ und der Term unter dem Bruchstrich (Divisor) die Quadratwurzel von $5 \times 85 \times 10 \times 80$, also $\sqrt{340.000}$. Das führt zu $\Phi \approx 0{,}069$, was ziemlich gering ist. Der Verzehr von Pizza hat also wahrscheinlich keinen Einfluss auf die Verwandlungen.

4.8 Korrelationen berechnen

In JavaScript können wir eine Tabelle aus zwei Spalten und zwei Zeilen als Array mit vier Elementen darstellen (`[76, 9, 4, 1]`). Wir können auch andere Darstellungen wählen, etwa ein Array, das wiederum zwei Arrays mit je zwei Elementen enthält (`[[76, 9], [4, 1]]`), oder ein Objekt mit Eigenschaftennamen wie `"11"` und `"01"`. Aber das lineare Array ist einfach und macht die Ausdrücke für den Zugriff auf die Tabelle angenehm kurz. Die Indizes im Array interpretieren wir als zweistellige Binärzahlen, wobei sich die linke (signifikanteste) Stelle auf die Verwandlungsvariable bezieht und die rechte (am wenigsten signifikante) auf die Ereignisvariable. Beispielsweise verweist die Binärzahl `10` auf den Fall, in dem Jacques sich in ein Eichhörnchen verwandelt hat, das Ereignis (etwa `"pizza"`) aber nicht eingetreten ist. Das ist viermal geschehen. Da die Binärzahl `10` einer 2 in Dezimalschreibweise entspricht, speichern wir diese Zahl daher im Index 2 des Arrays.

Aus einem solchen Array können wir nun den Koeffizienten Φ mit der folgenden Funktion berechnen:

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}
```

```
console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

Dies ist eine direkte Übersetzung der Formel für Φ in JavaScript. Bei `Math.sqrt` handelt es sich um die Quadratwurzelfunktion, die das Objekt `Math` in der Standardumgebung von JavaScript bereitstellt. Da die Summen der Zeilen und Spalten nicht direkt in unserer Datenstruktur gespeichert werden, müssen wir jeweils zwei Felder der Tabelle addieren, um Terme wie $n_{1,}$ darzustellen.

Jacques führt sein Tagebuch drei Monate lang. Die resultierende Datenmenge finden Sie in der Programmier-Sandbox für dieses Kapitel (*https://eloquentjavascript.net/code#4*). Sie ist dort in der Bindung `JOURNAL` sowie als herunterladbare Datei gespeichert.

Um dem Tagebuch eine 2 × 2-Tabelle für ein bestimmtes Ereignis zu entnehmen, müssen wir alle Einträge in einer Schleife durchlaufen und zählen, wie oft das Ereignis in Zusammenhang mit einer Verwandlung in ein Eichhörnchen auftritt:

```
function tableFor(event, journal) {
  let table = [0, 0, 0, 0];
  for (let i = 0; i < journal.length; i++) {
    let entry = journal[i], index = 0;
    if (entry.events.includes(event)) index += 1;
    if (entry.squirrel) index += 2;
    table[index] += 1;
  }
  return table;
}

console.log(tableFor("pizza", JOURNAL));
// → [76, 9, 4, 1]
```

Arrays verfügen über die Methode `includes`, die prüft, ob ein gegebener Wert in dem Array vorhanden ist. Unsere Funktion nutzt diese Methode, um zu bestimmen, ob der Ereignisname, an dem wir interessiert sind, in der Ereignisliste eines gegebenen Tages vorkommt.

Der Rumpf der Schleife in `tableFor` ermittelt, in welche Tabellenzelle die einzelnen Tagebucheinträge jeweils fallen. Dazu wird jeweils geprüft, ob ein Eintrag das gewünschte Ereignis enthält und ob es zusammen mit einer Eichhörnchenverwandlung aufgetreten ist. Anschließend fügt die Schleife den Eintrag in die richtige Tabellenzelle ein.

Damit haben wir jetzt ein Werkzeug, um die verschiedenen Korrelationen zu berechnen. Jetzt müssen wir nur noch die Korrelation für jeden aufgezeichneten Ereignistyp ermitteln und prüfen, ob dabei irgendetwas hervorsticht.

4.9 Array-Schleifen

Die Funktion `tableFor` enthält eine Schleife der folgenden Form:

```
for (let i = 0; i < JOURNAL.length; i++) {
```

```
  let entry = JOURNAL[i];
  // Macht irgendetwas mit dem Eintrag
}
```

Diese Art von Schleife ist in klassischem JavaScript üblich. Es kommt häufig vor, dass Arrays elementweise durchlaufen werden. Dazu wird ein Zähler über die ganze Länge des Arrays geführt und nacheinander jedes Element herausgegriffen.

In modernem JavaScript gibt es jedoch eine einfachere Möglichkeit, um solche Schleifen zu schreiben:

```
for (let entry of JOURNAL) {
  console.log(`${entry.events.length} events.`);
}
```

Eine `for`-Schleife dieser Art, bei der hinter einer Variablendefinition das Wort `of` steht, durchläuft die Elemente des Wertes, der hinter `of` angegeben ist. Das funktioniert nicht nur bei Arrays, sondern auch bei Strings und einigen anderen Datenstrukturen. Wie es funktioniert, sehen wir uns in Kapitel 6 an.

4.10 Die endgültige Analyse

Wir müssen jetzt noch die Korrelation für jeden Ereignistyp berechnen, der in der Datenmenge vorkommt. Dazu müssen wir zunächst jeden Ereignistyp *finden*:

```
function journalEvents(journal) {
  let events = [];
  for (let entry of journal) {
    for (let event of entry.events) {
      if (!events.includes(event)) {
        events.push(event);
      }
    }
  }
  return events;
}

console.log(journalEvents(JOURNAL));
// → ["carrot", "exercise", "weekend", "bread", ...]
```

Die Funktion geht alle Ereignisse durch und fügt diejenigen zum Array `events` hinzu, die sich noch nicht darin befinden. Dadurch sammelt sie sämtliche Ereignistypen.

Damit können wir nun alle Korrelationen berechnen:

```
for (let event of journalEvents(JOURNAL)) {
  console.log(event + ":", phi(tableFor(event, JOURNAL)));
}
// → carrot: 0.0140970969
// → exercise: 0.0685994341
```

```
// → weekend: 0.1371988681
// → bread: -0.0757554019
// → pudding: -0.0648203724
// usw.
```

Die meisten Korrelationen liegen sehr nah an null. Der Verzehr von Karotten, Brot und Pudding löst offensichtlich keine Wereichhörnchen aus. Allerdings *scheinen* die Verwandlungen an Wochenenden häufiger vorzukommen. Wir wollen die Ergebnisse nun so filtern, dass nur Korrelationen größer als 0,1 oder kleiner als –0,1 angezeigt werden:

```
for (let event of journalEvents(JOURNAL)) {
  let correlation = phi(tableFor(event, JOURNAL));
  if (correlation > 0.1 || correlation < -0.1) {
    console.log(event + ":", correlation);
  }
}
// → weekend: 0.1371988681
// → brushed teeth: -0.3805211953
// → candy: 0.1296407447
// → work: -0.1371988681
// → spaghetti: 0.2425356250
// → reading: 0.1106828054
// → peanuts: 0.5902679812
```

Aha! Es gibt zwei Faktoren, deren jeweilige Korrelation deutlich stärker ist als die der anderen. Der Verzehr von Erdnüssen hat eine starke positive Auswirkung auf die Wahrscheinlichkeit einer Verwandlung in ein Eichhörnchen, das Zähneputzen dagegen eine signifikant negative.

Das ist bemerkenswert. Probieren wir also Folgendes aus:

```
for (let entry of JOURNAL) {
  if (entry.events.includes("peanuts") &&
     !entry.events.includes("brushed teeth")) {
    entry.events.push("peanut teeth");
  }
}
console.log(phi(tableFor("peanut teeth", JOURNAL)));
// → 1
```

Das Ergebnis ist eindeutig. Das Phänomen tritt genau dann auf, wenn Jacques Erdnüsse isst und vergisst, sich die Zähne zu putzen. Wenn er nicht so schlampig wäre, was die Zahnpflege angeht, hätte er seine Veranlagung nicht einmal bemerkt! Angesichts dieser Erkenntnisse verzichtet Jacques fortan komplett auf Erdnüsse, woraufhin die Verwandlungen nicht mehr auftreten.

Einige Jahre lang geht alles gut. Dann aber verliert er seinen Job. Da er in einem ungemütlichen Land lebt, in dem Arbeitslosigkeit auch bedeutet, keine medizinische Versorgung zu haben, sieht er sich gezwungen, eine Stellung als »der unglaubliche Eichhörnchenmann« bei einem Zirkus anzunehmen. Vor jedem Auftritt stopft

er sich den Mund mit Erdnussbutter voll. Eines Tages jedoch, als er von seiner erbärmlichen Existenz schon genug hat, verwandelt er sich nicht mehr in seine menschliche Form zurück, sondern hoppelt durch einen Riss im Zirkuszelt, verschwindet im Wald und ward nie mehr gesehen.

4.11 Arrayologie für Fortgeschrittene

Bevor ich mit diesem Kapitel zum Abschluss komme, möchte ich noch einige weitere Dinge im Zusammenhang mit Objekten erwähnen. Als Erstes möchte ich dabei einige allgemein nützliche Array-Methoden vorstellen.

Im Abschnitt »*Methoden*« auf *S. 64* haben Sie bereits die Methoden `push` und `pop` kennengelernt, mit denen Sie Elemente am Ende eines Arrays hinzufügen und entfernen können. Die entsprechenden Methoden zum Hinzufügen und Entfernen von Elementen am Anfang eines Arrays heißen `unshift` und `shift`:

```
let todoList = [];
function remember(task) {
  todoList.push(task);
}
function getTask() {
  return todoList.shift();
}
function rememberUrgently(task) {
  todoList.unshift(task);
}
```

Dieses Programm verwaltet eine Warteschlange mit Aufgaben. Um Aufgaben am Ende der Warteschlange hinzufügen, rufen Sie z.B. `remember("groceries")` auf. Wenn Sie bereit sind, etwas zu erledigen, rufen Sie `getTask()` auf, um das erste Element der Warteschlange abzurufen (und zu entfernen). Auch die Funktion `rememberUrgently` fügt eine Aufgabe hinzu, allerdings nicht am Ende der Warteschlange, sondern vorn.

Um nach einem bestimmten Wert zu suchen, gibt es die Array-Methode `indexOf`. Sie durchsucht das Array von Anfang bis Ende und gibt den Index zurück, an dem sich der angegebene Wert befindet, oder `-1`, falls der Wert nicht zu finden ist. Um das stattdessen von hinten nach vorn zu durchsuchen, verwenden Sie die Methode `lastIndexOf`.

```
console.log([1, 2, 3, 2, 1].indexOf(2));
// → 1
console.log([1, 2, 3, 2, 1].lastIndexOf(2));
// → 3
```

Sowohl `indexOf` als auch `lastIndexOf` nehmen ein optionales zweites Argument entgegen, das angibt, wo mit der Suche begonnen werden soll.

Eine weitere grundlegende Array-Methode ist `slice`, die einen Start- und einen Endindex entgegennimmt und ein Array zurückgibt, das nur die dazwischen lie-

genden Elemente enthält, wobei der Startindex in dem Bereich eingeschlossen ist, der Endindex dagegen nicht:

```
console.log([0, 1, 2, 3, 4].slice(2, 4));
// → [2, 3]
console.log([0, 1, 2, 3, 4].slice(2));
// → [2, 3, 4]
```

Wird kein Endindex angegeben, nimmt `slice` alle Elemente beginnend mit dem Startindex. Sie können auch den Startindex weglassen, um das gesamte Array zu kopieren.

Die Methode `concat` dient dazu, Arrays zu einem neuen Array zusammenzufügen. Sie funktioniert ähnlich wie der Operator + für Strings.

Im folgenden Beispiel sehen Sie sowohl `slice` als auch `concat` in Aktion. Die Funktion nimmt ein Array und einen Index entgegen und gibt ein neues Array zurück, das eine Kopie des ursprünglichen ist, wobei allerdings das Element am angegebenen Index entfernt wurde:

```
function remove(array, index) {
  return array.slice(0, index)
    .concat(array.slice(index + 1));
}
console.log(remove(["a", "b", "c", "d", "e"], 2));
// → ["a", "b", "d", "e"]
```

Wenn Sie `concat` ein Argument übergeben, das kein Array ist, wird der Wert zu dem neuen Array hinzugefügt, als wäre er ein Array aus einem einzigen Element.

4.12 Eigenschaften von Strings

Bei Stringwerten können Sie Eigenschaften wie `length` und `toUpperCase` nutzen. Dagegen ist es nicht möglich, ihnen eine neue Eigenschaft hinzuzufügen:

```
let kim = "Kim";
kim.age = 88;
console.log(kim.age);
// → undefined
```

String-, numerische und boolesche Werte sind keine Objekte. Es gibt zwar keine Fehlermeldung, wenn Sie versuchen, ihnen neue Eigenschaften hinzuzufügen, doch werden diese Eigenschaften nicht gespeichert. Wie bereits erwähnt, sind Werte dieser Art unveränderbar.

Allerdings haben diese Typen eingebaute Eigenschaften. Jeder String verfügt über eine Reihe von Methoden. Zu den besonders nützlichen zählen `slice` und `indexOf`, die den gleichnamigen Array-Methoden ähneln:

```
console.log("coconuts".slice(4, 7));
// → nut
```

```
console.log("coconut".indexOf("u"));
// → 5
```

Ein Unterschied besteht darin, dass die Stringmethode `indexOf` nach einem String aus mehr als einem Zeichen suchen kann, wohingegen die vergleichbare Array-Methode nur nach einem einzelnen Element Ausschau hält:

```
console.log("one two three".indexOf("ee"));
// → 11
```

Die Methode `trim` entfernt Weißraum (Leerzeichen, Zeilenumbrüche, Tabulatoren u.Ä.) vom Anfang und Ende eines Strings:

```
console.log(" okay \n ".trim());
// → okay
```

Die Funktion `zeroPad` aus dem vorherigen Kapitel gibt es auch als Methode. Sie heißt `padStart` und nimmt die gewünschte Länge und das Auffüllzeichen als Argumente entgegen:

```
console.log(String(6).padStart(3, "0"));
// → 006
```

Mit `split` können Sie einen String bei jedem Vorkommen eines gegebenen enthaltenen Strings trennen. Umgekehrt lassen sich Strings mit `join` zusammensetzen:

```
let sentence = "Secretarybirds specialize in stomping";
let words = sentence.split(" ");
console.log(words);
// → ["Secretarybirds", "specialize", "in", "stomping"]
console.log(words.join(". "));
// → Secretarybirds. specialize. in. stomping
```

Um einen String zu wiederholen, verwenden Sie die Methode `repeat`. Sie erstellt einen neuen String, der aus mehreren zusammengefügten Kopien des ursprünglichen Strings besteht:

```
console.log("LA".repeat(3));
// → LALALA
```

Die Stringeigenschaft `length` haben Sie bereits kennengelernt. Der Zugriff auf einzelne Zeichen eines Strings erfolgt ähnlich wie der Zugriff auf Array-Elemente (wobei es einen kleinen Haken gibt, den wir in »***Strings und Zeichencodes***« auf ***S. 94*** besprechen werden).

```
let string = "abc";
console.log(string.length);
// → 3
console.log(string[1]);
// → b
```

4.13 Restparameter

Manchmal muss eine Funktion auch in der Lage sein, eine beliebige Anzahl von Argumenten entgegenzunehmen. So berechnet beispielsweise `Math.max` das Maximum aller ihr übergebenen Argumente.

Um eine solche Funktion zu schreiben, setzen Sie vor Ihren letzten Parameter drei Punkte:

```
function max(...numbers) {
  let result = -Infinity;
  for (let number of numbers) {
    if (number > result) result = number;
  }
  return result;
}
console.log(max(4, 1, 9, -2));
// → 9
```

Beim Aufruf einer solchen Funktion wird der *Restparameter* an ein Array gebunden, das alle weiteren Argumente enthält. Stehen vor diesem Parameter noch andere, gehören deren Werte nicht zu dem Array. Ist der Restparameter der einzige Parameter, wie es bei `max` der Fall ist, so enthält das Array alle Argumente.

Eine ähnliche Schreibweise mit drei Punkten können Sie auch verwenden, um eine Funktion mit einem Argument-Array *aufzurufen*:

```
let numbers = [5, 1, 7];
console.log(max(...numbers));
// → 7
```

Dadurch wird das Array in dem Funktionsausruf ausgepackt, d. h., seine Elemente werden als einzelne Argumente übergeben. Es ist auch möglich, ein Array wie dieses zusammen mit anderen Argumenten anzugeben, etwa wie in `max(9, ...numbers, 2)`.

Bei der Verwendung eckiger Klammern kann der Drei-Punkte-Operator ein Array in ausgepackter Form in ein neues Array einbauen:

```
let words = ["never", "fully"];
console.log(["will", ...words, "understand"]);
// → ["will", "never", "fully", "understand"]
```

4.14 Das Objekt Math

Wie Sie bereits gesehen haben, ist `Math` eine Wundertüte voller arithmetischer Hilfsfunktionen wie `Math.max` (Maximum), `Math.min` (Minimum) und `Math.sqrt` (Quadratwurzel). Das Objekt `Math` dient als Container, um verwandte Funktionen zu bündeln. Es gibt nur ein einziges `Math`-Objekt, und als Wert hat es praktisch keine sinnvolle Bedeutung. Stattdessen bietet es einen *Namespace* oder *Namensraum*, damit all seine Funktionen und Werte keine globalen Bindungen aufweisen

müssen. Durch zu viele globale Bindungen wird ein Namensraum »verunreinigt«. Je mehr Namen genutzt werden, umso wahrscheinlicher ist es, dass man versehentlich den Wert einer bereits vorhandenen Bindung überschreibt. Beispielsweise kann es durchaus vorkommen, dass in Ihrem Programm etwas vorkommt, das Sie `max` nennen möchten. Da die eingebaute JavaScript-Funktion `max` sicher im Objekt `Math` liegt, müssen Sie sich keine Sorgen darüber machen, dass Sie sie überschreiben könnten.

Viele Sprachen hindern Sie daran, eine Bindung mit einem Namen zu definieren, den es bereits gibt, oder werfen in einem solchen Fall zumindest eine Warnung aus. JavaScript macht das bei Bindungen, die Sie mit `let` oder `const` deklarieren, aber absurderweise nicht bei Standardbindungen oder solchen, die mit `var` oder `function` deklariert werden.

Nun aber zurück zu `Math`! Auch wenn Sie trigonometrische Berechnungen durchführen müssen, kann Ihnen dieses Objekt helfen. Es enthält die Funktionen `cos` (Kosinus), `sin` (Sinus) und `tan` (Tangens) sowie die Umkehrfunktionen `acos`, `asin` und `atan`. Die Zahl Pi – oder zumindest die genaueste Näherung, die in eine JavaScript-Zahl passt – steht als `Math.PI` zur Verfügung. Nach einer alten Programmiertradition werden die Namen von Konstanten in Großbuchstaben geschrieben.

```
function randomPointOnCircle(radius) {
  let angle = Math.random() * 2 * Math.PI;
  return {x: radius * Math.cos(angle),
          y: radius * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
// → {x: 0.3667, y: 1.966}
```

Machen Sie sich keine Sorgen, wenn Sie mit Sinus und Kosinus nicht vertraut sind. Wenn wir diese Funktionen in Kapitel 14 verwenden, werde ich sie erklären.

Im vorstehenden Beispiel wurde auch `Math.random` verwendet. Diese Funktion gibt bei jedem Aufruf eine neue Pseudozufallszahl zwischen einschließlich 0 und ausschließlich 1 zurück:

```
console.log(Math.random());
// → 0.36993729369714856
console.log(Math.random());
// → 0.727367032552138
console.log(Math.random());
// → 0.40180766698904335
```

Computer sind zwar deterministische Maschinen, d.h., sie reagieren auf dieselben Eingaben stets auf dieselbe Weise, aber es ist trotzdem möglich, sie Zahlen produzieren zu lassen, zu zufällig erscheinen. Wenn Sie den Computer nach einem Zufallswert fragen, führt er komplizierte Berechnungen an einem gespeicherten verborgenen Wert aus, um einen neuen zu erzeugen. Er speichert diesen neuen Wert

und gibt eine davon abgeleitete Zahl zurück. Dadurch kann er immer neue, schwer vorherzusehende Zahlen produzieren, die zufällig zu sein *scheinen*.

Wenn Sie zufällige ganze Zahlen statt Bruchzahlen brauchen, können Sie `Math.floor` auf das Ergebnis von `Math.random` anwenden. Dadurch wird es auf die nächste ganze Zahl abgerundet:

```
console.log(Math.floor(Math.random() * 10));
// → 2
```

Durch die Multiplikation der Zufallszahl mit 10 erhalten wir eine Zahl, die größer oder gleich 0 und kleiner als 10 ist. Da `Math.floor` nach unten rundet, ergibt dieser Ausdruck eine beliebige Zahl von 0 bis 9.

Darüber hinaus gibt es noch die Funktionen `Math.ceil` (für »ceiling«, also »Decke«), die auf die nächste ganze Zahl aufrundet, `Math.round`, die eine reguläre Rundung auf die nächste ganze Zahl durchführt, und `Math.abs`, die den Absolutwert einer Zahl zurückgibt, also negative Zahlen negiert, aber positive unverändert lässt.

4.15 Zerlegung

Sehen wir uns noch einmal unsere Funktion `phi` an:

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}
```

Diese Funktion ist etwas umständlich zu lesen, was unter anderem daran liegt, dass hier eine Bindung auf das Array zeigt. Es wäre besser, wenn wir Bindungen für die *Elemente* des Arrays hätten, also `let n00 = table[0]` usw. Glücklicherweise gibt es in JavaScript eine Möglichkeit, um dies kurz und bündig zu tun:

```
function phi([n00, n01, n10, n11]) {
  return (n11 * n00 - n10 * n01) /
    Math.sqrt((n10 + n11) * (n00 + n01) *
              (n01 + n11) * (n00 + n10));
}
```

Das funktioniert auch bei Bindungen, die mit `let`, `var` oder `const` erstellt wurden. Wenn Sie wissen, dass der Wert, den Sie binden möchten, ein Array ist, können Sie eckige Klammern verwenden, um »in den Wert hineinzusehen« und seine Inhalte zu binden.

Einen ähnlichen Trick gibt es auch für Objekte, wobei jedoch geschweifte statt eckiger Klammern verwendet werden:

```
let {name} = {name: "Faraji", age: 23};
console.log(name);
// → Faraji
```

Wenn Sie versuchen, `null` oder `undefined` zu zerlegen, erhalten Sie ebenso eine Fehlermeldung, als würden Sie versuchen, direkt auf Eigenschaften dieser Werte zuzugreifen.

4.16 JSON

Da Eigenschaften ihre Werte nur festhalten und nicht enthalten, werden Objekte und Arrays im Arbeitsspeicher als Folgen von Bits gespeichert, die die *Adressen* – also den Speicherort im Arbeitsspeicher – ihrer Inhalte angeben. Ein Array, das ein anderes Array enthält, besteht also aus (mindestens) einer Speicherregion für das innere Array sowie einer für das äußere, die (unter anderem) eine Binärzahl zur Angabe der Position des inneren Arrays enthält.

Wenn Sie die Daten in einer Datei speichern oder über das Netzwerk an einen anderen Computer senden möchten, müssen Sie dieses Gewirr von Speicheradressen in eine Beschreibung umwandeln, die sich auch tatsächlich speichern bzw. senden lässt. Sie *könnten* zwar Ihren gesamten Arbeitsspeicher zusammen mit der Adresse des Wertes senden, an dem Sie interessiert sind, aber das ist offensichtlich nicht gerade eine ideale Vorgehensweise.

Stattdessen *serialisieren* wir die Daten, d.h., wir wandeln sie in eine lineare Beschreibung um. Ein weit verbreitetes Serialisierungsformat ist *JSON* (ausgesprochen wie der englische Name »Jason«), was für JavaScript Object Notation steht. Es wird im Web weiträumig als Format zur Datenspeicherung und Kommunikation genutzt, auch in anderen Sprachen als JavaScript.

JSON ähnelt der JavaScript-Schreibweise von Arrays und Objekten, wobei es jedoch einige Einschränkungen gibt. Alle Eigenschaftennamen müssen in doppelte Anführungszeichen eingeschlossen sein, und nur einfache Datenausdrücke sind erlaubt, keine Funktionsaufrufe, Bindungen oder irgendetwas anderes, das eine Berechnung erfordert. Auch Kommentare sind in JSON nicht zulässig.

In JSON-Form kann einer unserer Tagebucheinträge wie folgt aussehen:

```
{
  "squirrel": false,
  "events": ["work", "touched tree", "pizza", "running"]
}
```

JavaScript enthält die Funktionen `JSON.stringify` und `JSON.parse`, um Daten in dieses Format bzw. aus ihm zu konvertieren. Die erste nimmt einen JavaScript-Wert entgegen und gibt einen JSON-String zurück, die zweite wandelt einen solchen String in den entsprechenden Wert um:

```
let string = JSON.stringify({squirrel: false,
                             events: ["weekend"]});
console.log(string);
```

```
// → {"squirrel":false,"events":["weekend"]}
console.log(JSON.parse(string).events);
// → ["weekend"]
```

4.17 Zusammenfassung

Objekte und Arrays (bei denen es sich um eine besondere Art von Objekten handelt) bieten Möglichkeiten, um Werte zu einem einzigen Wert zu gruppieren. Das ist so, als würden wir mehrere zusammengehörige Gegenstände in einen Sack stecken und diesen Sack mit uns führen, anstatt alle einzelnen Gegenstände auf unseren Armen zu balancieren und zu versuchen, sie festzuhalten.

In JavaScript haben alle Werte mit Ausnahme von `null` und `undefined` Eigenschaften. Der Zugriff auf diese Eigenschaften erfolgt mit `value.prop` oder `value["prop"]`. Objekte haben gewöhnlich einen mehr oder weniger festen Satz von Eigenschaften, die üblicherweise Namen tragen. Arrays dagegen enthalten gewöhnlich verschiedene Mengen gleichartiger Werte und nutzen Zahlen (beginnend mit 0) zur Bezeichnung ihrer Eigenschaften.

Es gibt jedoch auch einige benannte Eigenschaften in Arrays, z. B. `length` und eine Reihe von Methoden. Bei Letzteren handelt es sich um Funktionen, die in Eigenschaften untergebracht sind und sich (gewöhnlich) auf den Wert auswirken, dessen Eigenschaft sie sind.

Arrays lassen sich mit einer besonderen Art von `for`-Schleife durchlaufen: `for (let element of array)`.

4.18 Übungen

Summe eines Bereichs

In der Einleitung dieses Buches wurde der folgende Code als praktische Möglichkeit angesprochen, um die Summe eines Zahlenbereichs zu berechnen:

```
console.log(sum(range(1, 10)));
```

Schreiben Sie eine Funktion namens `range`, die die beiden Argumente `start` und `end` entgegennimmt und ein Array mit allen Zahlen von `start` bis einschließlich `end` zurückgibt.

Schreiben Sie dann die Funktion `sum`, die ein Array aus Zahlen entgegennimmt und die Summe dieser Zahlen zurückgibt. Führen Sie ein Beispielprogramm aus und vergewissern Sie sich, dass das Ergebnis tatsächlich 55 lautet.

Als Zusatzaufgabe ändern Sie die Funktion `range` so ab, dass Sie als optionales drittes Argument eine Schrittweite zum Aufbau des Arrays entgegennimmt, wobei die Elemente wie zuvor jeweils um 1 inkrementiert werden, wenn keine Schrittweite angegeben ist. Beispielsweise sollte der Funktionsaufruf `range(1, 10, 2)` das Array `[1, 3, 5, 7, 9]` zurückgeben. Sorgen Sie dafür, dass dies auch bei negativen Schrittweiten funktioniert, sodass `range(5, 2, -1)` das Array `[5, 4, 3, 2]` ergibt.

Arrays umkehren

Arrays verfügen über die Methode `reverse`, die die Reihenfolge der Elemente umkehrt. Schreiben Sie die beiden Funktionen `reverseArray` und `reverseArrayInPlace`. Dabei soll `reverseArray` ein Array als Argument entgegennehmen und ein *neues* Array mit denselben Elementen, aber in umgekehrter Reihenfolge, zurückgeben. Die zweite Funktion, `reverseArrayInPlace`, soll sich dagegen genauso verhalten wie `reverse`, also das als Argument übergebene Array *ändern*, indem sie die Reihenfolge der Elemente umkehrt. Für keine dieser beiden Funktionen darf die Standardmethode `reverse` verwendet werden.

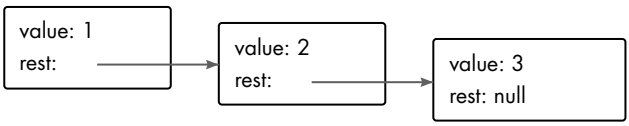
Denken Sie an das zurück, was Sie im Abschnitt »***Seiteneffekte***« auf ***S. 57*** über Seiteneffekte und reine Funktionen gelesen haben. Welche Version sollte demnach in mehr Situationen nützlich sein? Welche läuft schneller?

Listen

Da Objekte ganz allgemein Mengen von Werten sind, lassen sich damit alle möglichen Arten von Datenstrukturen erstellen. Eine häufig verwendete Datenstruktur ist die *Liste* (nicht zu verwechseln mit einem Array). Eine Liste ist eine Menge verschachtelter Objekte, wobei das erste Objekt einen Verweis auf das zweite festhält, das zweite einen Verweis auf das dritte usw.

```
let list = {
  value: 1,
  rest: {
    value: 2,
    rest: {
      value: 3,
      rest: null
    }
  }
};
```

Die resultierenden Objekte bilden eine Kette:



Eine schöne Sache von Listen ist, dass sie einen Teil ihrer Struktur gemeinsam verwenden können. Wenn ich beispielsweise die beiden neuen Werte `{value: 0, rest: list}` und `{value: -1, rest: list}` erstelle (wobei `list` auf die zuvor definierte Bindung verweist), habe ich zwei unabhängige Listen, die aber gemeinsam die Struktur verwenden, die ihre letzten drei Elemente ausmacht. Auch die ursprüngliche Liste ist nach wie vor eine gültige Liste aus drei Elementen.

Schreiben Sie die Funktion `arrayToList`, die eine Listenstruktur wie die hier gezeigte erstellt, wenn ihr `[1, 2, 3]` als Argument übergeben wird. Schreiben Sie

des Weiteren die Funktion `listToArray`, die ein Array aus einer Liste gewinnt. Fügen Sie die Hilfsfunktion `prepend` hinzu, die ein Element und eine Liste entgegennimmt und eine neue Liste erstellt, bei der das Element vorn an die übergebene Liste angefügt ist. Schreiben Sie auch die Hilfsfunktion `nth`, die eine Liste und eine Zahl entgegennimmt und das Element an der gegebenen Position in der Liste zurückgibt (wobei das erste Element die Position 0 hat) bzw. `undefined`, wenn es kein solches Element gibt.

Schreiben Sie außerdem eine rekursive Version von `nth`, falls Sie das noch nicht getan haben.

Tiefer Vergleich

Der Operator `==` vergleicht Objekte anhand ihrer Identität. Manchmal ist es jedoch sinnvoll, die tatsächlichen Werte der Eigenschaften zu vergleichen.

Schreiben Sie die Funktion `deepEqual`, die zwei Werte entgegennimmt und nur dann `true` zurückgibt, wenn die Werte gleich sind oder Objekte mit den gleichen Eigenschaften sind und sich die Werte dieser Eigenschaften wiederum bei einem rekursiven Aufruf von `deepEqual` als gleich erweisen.

Um herauszufinden, ob die Werte direkt verglichen werden sollen (mit dem Operator `===`) oder ihre Eigenschaften, können Sie den Operator `typeof` verwenden. Wenn er für beide Werte `"object"` zurückgibt, müssen Sie einen tiefen Vergleich durchführen. Beachten Sie dabei aber eine absurde Ausnahme: Aufgrund einer historischen Fehlentscheidung ergibt `typeof null` ebenfalls `"object"`.

Um die Eigenschaften eines Objekts für den Vergleich zu durchlaufen, ist die Funktion `Object.keys` hilfreich.

»Es gibt zwei Arten des Softwaredesigns: Eine Möglichkeit besteht darin, sie so einfach zu machen, dass es offensichtlich keine Mängel gibt. Die andere besteht darin, sie so kompliziert zu machen, dass es keine offensichtlichen Mängel gibt.«

– C. A. R. Hoare, Rede anlässlich der Verleihung des ACM Turing Award 1980