Was Sie sich merken sollten

- Bei Methodenaufrufen wird das Objekt, in dem die Methodeneigenschaft nachgeschlagen wird, als Empfänger übergeben.
- Funktionsaufrufe stellen das globale Objekt als Empfänger zur Verfügung (oder `undefined` bei strengen Funktionen). Der Aufruf von Methoden mit der Syntax von Funktionsaufrufen ist nur selten sinnvoll.
- Konstruktoren werden mit `new` aufgerufen und erhalten ein neues Objekt als Empfänger.

Thema 19 Keine Angst vor Funktionen höherer Ordnung

Der Begriff *Funktionen höherer Ordnung* war einmal das Erkennungswort der Eingeweihten aus dem Zirkel der funktionalen Programmierung, eine esoterische Bezeichnung für etwas, das eine fortgeschrittene Programmiertechnik zu sein schien. Nichts könnte aber weiter von der Wahrheit entfernt sein. Die prägnante Eleganz von Funktionen zu nutzen, kann zu einfacherem und kompakterem Code führen. Im Laufe der Jahre haben Skriptsprachen diese Techniken übernommen und damit den Schleier des Geheimnisvollen gelüftet, der die besten Idiome der funktionalen Programmierung umgab.

Callback-Funktionen

Funktionen höherer Ordnung sind nichts anderes als Funktionen, die andere Funktionen als Argumente übernehmen oder als Ergebnis eine Funktion zurückgeben. Eine Funktion als Argument zu übernehmen (die häufig als *Callback-Funktion* bezeichnet wird, da sie von der Funktion höherer Ordnung »zurückgerufen« wird), ist ein besonders leistungsfähiges und ausdrucksstarkes Idiom, das in JavaScript-Programmen häufig eingesetzt wird.

Betrachten Sie die Standardmethode `sort` für Arrays. Damit sie bei allen möglichen Arrays funktioniert, muss der Aufrufer bestimmen, wie zwei Elemente in dem Array verglichen werden:

```
function compareNumbers(x, y) {
   if (x < y) {
      return -1;
   }
   if (x > y) {
      return 1;
   }
   return 0;
}
[3, 1, 4, 1, 5, 9].sort(compareNumbers); // [1, 1, 3, 4, 5, 9]
```

In der Standardbibliothek hätte man auch verlangen können, dass der Aufrufer ein Objekt mit einer compare-Methode übergibt, aber wenn nur eine einzige Methode benötigt wird, ist es einfacher und kürzer, direkt eine Funktion zu übernehmen. Das obenstehende Beispiel lässt sich mithilfe einer anonymen Funktion sogar noch weiter vereinfachen:

```
[3, 1, 4, 1, 5, 9].sort(function(x, y) {
   if (x < y) {
      return -1;
   }
   if (x > y) {
      return 1;
   }
   return 0;
}); // [1, 1, 3, 4, 5, 9]
```

Wenn Sie wissen, wie Sie Funktionen höherer Ordnung einsetzen, können Sie damit Ihren Code oftmals vereinfachen und auf mühselige Wiederholungen von 08/15-Code verzichten. Für viele gebräuchliche Arrayoperationen gibt es wunderbare Abstraktionen höherer Ordnung, bei denen es sich wirklich lohnt, sich mit ihnen vertraut zu machen. Betrachten Sie als Beispiel die einfache Aufgabe, ein Array aus Strings komplett in Großbuchstaben umzuwandeln. Um das in einer Schleife zu erledigen, müssen wir Folgendes schreiben:

```
var names = ["Fred", "Wilma", "Pebbles"];
var upper = [];
for (var i = 0, n = names.length; i < n; i++) {
   upper[i] = names[i].toUpperCase();
}
upper; // ["FRED", "WILMA", "PEBBLES"]
```

Bei der praktischen Arraymethode map (die in ES5 eingeführt wurde) brauchen wir uns jedoch um die Einzelheiten der Schleife keine Sorgen zu machen, sondern müssen lediglich die elementweise Umwandlung mit einer lokalen Funktion implementieren:

```
var names = ["Fred", "Wilma", "Pebbles"];
var upper = names.map(function(name) {
   return name.toUpperCase();
});
upper; // ["FRED", "WILMA", "PEBBLES"]
```

Eigene Funktionen schreiben

Wenn Sie erst einmal den Dreh heraushaben, wie Sie Funktionen höherer Ordnung verwenden, können Sie auch die Gelegenheiten erkennen, an denen es sich lohnt, eigene zu schreiben. Sichere Kennzeichen dafür, dass Ihr Code nur auf eine Abstraktion höherer Ordnung wartet, sind Vorkommen von doppeltem oder ähnlichem Code. Nehmen wir bei-

spielsweise an, in einem Teil eines Programms wird wie folgt ein String aus allen Buchstaben des Alphabets zusammengestellt:

```
var aIndex = "a".charCodeAt(0); // 97

var alphabet = "";
for (var i = 0; i < 26; i++) {
   alphabet += String.fromCharCode(aIndex + i);
}
alphabet; // "abcdefghijklmnopqrstuvwxyz"
```

An einer anderen Stelle des Programms wird ein String aus Ziffern erstellt:

```
var digits = "";
for (var i = 0; i < 10; i++) {
   digits += i;
}
digits; // "0123456789"
```

Schließlich baut das Programm an einer dritten Stelle einen String aus zufälligen Zeichen zusammen:

```
var random = "";

for (var i = 0; i < 8; i++) {
   random += String.fromCharCode(Math.floor(Math.random() * 26)
                                 + aIndex);
}
random; // "bdwvfrtp" (jedes Mal ein anderes Ergebnis)
```

An jeder dieser drei Stellen wird ein anderer String erstellt, aber die Logik ist die gleiche. In jeder Schleife wird ein String durch die Verkettung der Ergebnisse von Berechnungen für die einzelnen Abschnitte zusammengestellt. Die gemeinsame Logik können wir in eine Hilfsfunktion auslagern:

```
function buildString(n, callback) {
   var result = "";
   for (var i = 0; i < n; i++) {
      result += callback(i);
   }
   return result;
}
```

Die Implementierung von `buildString` enthält alle Gemeinsamkeiten der drei Schleifen, verwendet anstelle der abweichenden Teile aber Parameter: Aus der Anzahl der Schleifeniterationen wird die Variable `n`,

aus der Konstruktion der einzelnen Stringabschnitte ein Aufruf der Funktion `callback`. Mit `buildString` können wir nun alle drei Beispiele vereinfachen:

```
var alphabet = buildString(26, function(i) {
   return String.fromCharCode(aIndex + i);
});
alphabet; // "abcdefghijklmnopqrstuvwxyz"

var digits = buildString(10, function(i) { return i; });
digits; // "0123456789"

var random = buildString(8, function() {
   return String.fromCharCode(Math.floor(Math.random() * 26)
                              + aIndex);
});
random; // "ltvisfjr" (jedes Mal ein anderes Ergebnis)
```

Abstraktionen höherer Ordnung bieten viele Vorteile. Knifflige Teile der Implementierung, z.B. die richtigen Randbedingungen der Schleife, werden in die Implementierung der Funktion höherer Ordnung verlagert. Dadurch müssen Sie Fehler in der Logik nur an einer Stelle korrigieren, anstatt nach jedem Vorkommen des betreffenden Codemusters im gesamten Programm Ausschau zu halten. Wenn Sie die Operation effizienter gestalten wollen, müssen Sie die Änderungen ebenfalls nur an einer einzigen Stelle vornehmen. Wenn Sie der Abstraktion schließlich noch einen erläuternden Namen wie `buildString` geben, wird für jeden Leser deutlich, was der Code macht, ohne dass er sich die Einzelheiten der Implementierung genauer ansehen muss.

Wenn Sie Funktionen höherer Ordnung anstreben, sobald Sie feststellen, dass Sie wiederholt Code im gleichen Muster schreiben, führt das zu knapperem Code, höherer Produktivität und besserer Lesbarkeit. Gewöhnen Sie sich an, nach häufig vorkommenden Mustern Ausschau zu halten und sie in Hilfsfunktionen höherer Ordnung zu verlagern.

Was Sie sich merken sollten

- Funktionen höherer Ordnung nehmen andere Funktionen als Argumente an oder geben sie als Ergebnis zurück.
- Machen Sie sich mit den Funktionen höherer Ordnung in den bestehenden Bibliotheken vertraut.
- Lernen Sie häufig vorkommende Codemuster zu erkennen, die sich durch Funktionen höherer Ordnung ersetzen lassen.