Kapitel 2
Funktionen und funktionale Aspekte

Die funktionalen Aspekte von JavaScript bilden die Grundlage für viele Entwurfsmuster dieser Sprache; ein Grund, sich diesem Thema direkt zu Beginn zu widmen.

Eine der wichtigsten Eigenschaften von JavaScript ist, dass es sowohl funktionale als auch objektorientierte Programmierung ermöglicht. Dieses und das folgende Kapitel stellen die beiden Programmierparadigmen kurz vor und erläutern anschließend jeweils im Detail die Anwendung in JavaScript. Ich starte bewusst mit den funktionalen Aspekten, weil viele der in Kapitel 3, »Objektorientierte Programmierung mit JavaScript«, beschriebenen Entwurfsmuster auf diesen funktionalen Grundlagen aufbauen.

Ziel dieses Kapitels ist es nicht, aus Ihnen einen Profi im funktionalen Programmieren zu machen. Verstehen Sie dies nicht falsch, aber das wäre auf knapp 70 Seiten schon recht sportlich. Vielmehr ist mein Ziel, Ihnen die wichtigsten funktionalen Konzepte, die bei der JavaScript-Entwicklung zum Einsatz kommen, zu veranschaulichen und Ihnen zu jedem Konzept Einsatzgebiete und Anwendungsbeispiele vorzustellen.

2.1 Die Besonderheiten von Funktionen in JavaScript

Fassen wir kurz die Punkte aus dem letzten Kapitel zusammen, die Sie dort über Funktionen gelernt haben:

- Funktionen werden in JavaScript durch Objekte repräsentiert und über das Schlüsselwort `function` definiert (es sei denn, Sie definieren in ES2015 eine Arrow-Funktion oder eine Objektmethode, dazu gleich mehr).
- Funktionen können auf unterschiedliche Arten erzeugt werden: über eine Funktionsanweisung, über einen Funktionsausdruck, über den Aufruf der Konstruktorfunktion `Function` und seit ES2015 als Arrow-Funktion.
- Implizit haben Sie innerhalb der Funktion Zugriff auf alle Funktionsparameter über das `arguments`-Objekt.

Lassen Sie mich im Folgenden auf diese einzelnen Punkte genauer eingehen sowie einige weitere Eigenschaften und Besonderheiten von Funktionen in JavaScript vorstellen.

2.1.1 Funktionen als First-Class-Objekte

In JavaScript werden Funktionen durch Objekte repräsentiert, genauer gesagt als Instanzen des `Function`-Typs. Legt man beispielsweise folgende Funktion an, ...

```
function add(x, y) {
  return x + y;
}
```

Listing 2.1 Definition einer Funktion

... werden eigentlich, wie in Abbildung 2.1 zu sehen, ein Funktionsobjekt mit dem Namen `add` erzeugt sowie eine gleichnamige Variable (`add`), die auf dieses Funktionsobjekt zeigt.

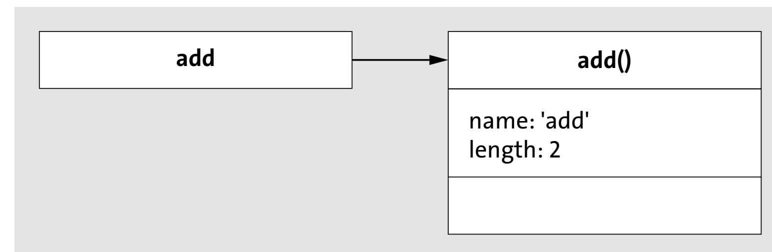

Abbildung 2.1 Funktionen werden durch Objekte repräsentiert.

Jedes Funktionsobjekt verfügt dabei standardmäßig über drei Eigenschaften: `name` enthält den Namen der Funktion, `length` die Anzahl an (in der Deklaration definierten) Funktionsparametern und `prototype` den sogenannten Prototyp der Funktion. Letzterer bezeichnet kurz gesagt das Objekt, auf dem das jeweilige Funktionsobjekt basiert. Details dazu gibt es im nächsten Kapitel, wenn es an die objektorientierten und prototypischen Aspekte von JavaScript geht.

Neben diesen drei Eigenschaften hat jede Funktion ihrerseits eigene Funktionen bzw. Methoden: `bind()`, `apply()` und `call()`. Abschnitt 2.2, »Standardmethoden jeder Funktion«, beschreibt, wozu diese Methoden gut sind und in welchen Fällen Sie sie benötigen.

Definition von Methoden und Funktionen

Im Weiteren wollen wir Funktionen, die als Eigenschaft eines Objekts oder einer anderen Funktion definiert werden, wie im Programmierjargon üblich, *Methoden* nennen. Funktionen, die für sich stehen, nennen wir weiterhin *Funktionen*.

Funktionen sind also Objekte. Das bedeutet logischerweise, dass sie an allen Stellen verwendet werden können, an denen auch »normale« Objekte verwendet werden

können: Sie können Variablen zugewiesen, als Werte innerhalb von Arrays verwendet, innerhalb von Objekten oder gar innerhalb anderer Funktionen definiert und als Parameter oder Rückgabewert von Funktionen verwendet werden. Lassen Sie mich Ihnen in den folgenden Abschnitten die einzelnen dieser Fälle kurz anhand von ein paar Quelltextbeispielen vorstellen.

Definition von Funktionen erster Klasse

Aufgrund ihrer Repräsentation durch Objekte sowie der gerade beschriebenen Verwendungsmöglichkeiten von Funktionen im Code spricht man in diesem Zusammenhang auch von *Funktionen erster Klasse* (*First-Class Functions*). Funktionen haben den gleichen Stellenwert wie Objekte oder primitive Datentypen, sie sind »first class«. Funktionen, die andere Funktionen als Parameter erwarten oder als Rückgabewert liefern, nennt man zusätzlich *Funktionen höherer Ordnung* (*Higher-Order Functions*).

Funktionen Variablen zuweisen

Wenn eine Funktion einer Variablen zugewiesen wird, passiert nichts anderes, als dass die Variable anschließend auf das Funktionsobjekt zeigt. Die Funktion kann dann über die Variable aufgerufen werden, wie Listing 2.2 zeigt, in dem die von eben bekannte Funktion `add` der Variablen `operation` zugewiesen wird:

```
const operation = add;
```

Listing 2.2 Zuweisung einer Funktion zu einer Variablen

Zu beachten ist, dass die Funktion `add` dabei nicht aufgerufen wird, was ein häufig gemachter Flüchtigkeitsfehler wäre, der dazu führen würde, dass in der Variablen `operation` nur der Rückgabewert des Funktionsaufrufs gespeichert würde.

Was durch die Zuweisung geschehen ist, stellt Abbildung 2.2 graphisch dar: Zusätzlich zur vorhin implizit definierten Variable `add` gibt es nun eine weitere (explizit definierte) Variable `operation`, die auf das gleiche Funktionsobjekt zeigt.

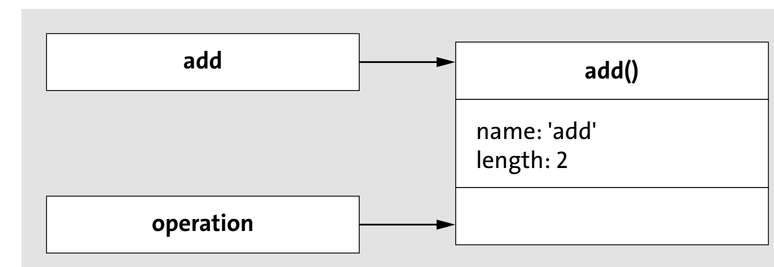

Abbildung 2.2 Funktionen sind first class, sie können beispielsweise Variablen zugewiesen werden.

Die Funktion kann nun über beide Variablen aufgerufen werden:

```
const result = add(2,2);
const result2 = operation(2,2);
```

Listing 2.3 Aufruf einer Funktion über implizite und explizite Variable

Beachten Sie hierbei aber den Hinweis aus Kapitel 1, »Einführung«: Die Eigenschaften der Ursprungsfunktion bleiben erhalten, der Name der Funktion beispielsweise lautet in beiden Fällen `add`. Das macht Sinn: Die neue Variable `operation` stellt wie die ursprüngliche Variable `add` lediglich eine Referenzvariable auf die Funktion mit dem Namen `add` dar.

```
console.log(add.name);       // Ausgabe: add
console.log(operation.name); // Ausgabe: add
```

Listing 2.4 Der Name einer Funktion ist unabhängig vom Variablennamen.

Funktionen in Arrays verwenden

Variablen, die auf Funktionsobjekte zeigen, können Sie an allen Stellen verwenden, an denen Sie auch »normale« Variablen verwenden dürfen. Auch der Einsatz innerhalb von Arrays ist möglich, wie folgendes Beispiel zeigt. Hierbei werden zunächst die vier Funktionen `add`, `subtract`, `multiply` und `divide` definiert und einem Array als Werte übergeben. Innerhalb der Iteration über das Array werden dann die einzelnen Funktionen aufgerufen.

```
function add(x,y) {
  return x+y;
}
function subtract(x,y) {
  return x-y;
}
function multiply(x,y) {
  return x*y;
}
function divide(x,y) {
  return x/y;
}
const operations = [
  add,
  subtract,
  multiply,
  divide
];
```



```
let operation;
for(let i=0; i<operations.length; i++) {
  operation = operations[i];
  const x = (i+1)*2;
  const y = (i+1)*4;
  const result = operation(x,y);
  console.log(result);
}
// Ausgabe:
// 6
// -4
// 72
// 0.5
```

Listing 2.5 Funktionen können als Werte im Array verwendet werden.

> **Hinweis**
>
> In klassenbasierten Programmiersprachen, bei denen Funktionen nicht Objekte erster Klasse sind, würde man ein solches Programm vergleichsweise aufwendig mit dem Command-Entwurfsmuster, einem der berühmten Entwurfsmuster der *Gang of Four* (GoF-Entwurfsmuster), lösen. Ich werde auf diese Thematik noch in Kapitel 8, »Die Entwurfsmuster der Gang of Four«, zurückkommen. Dort werden Sie sehen, dass auch viele andere der GoF-Entwurfsmuster entweder durch funktionale Techniken wegfallen oder viel einfacher zu implementieren sind.

Funktionen als Funktionsparameter verwenden

Funktionen können in JavaScript als Parameter einer anderen Funktion verwendet werden. Listing 2.6 zeigt eine Funktion, der als erster Parameter eine andere Funktion übergeben wird:

```
function metaOperation(operation, x, y) {
  return operation(x,y);
}
```

Listing 2.6 Funktionen können selbst als Parameter einer Funktion verwendet werden.

In der Praxis werden Funktionen als Parameter recht häufig verwendet. Das wohl bekannteste Beispiel dazu ist das sogenannte *Callback-Entwurfsmuster*, das insbesondere bei asynchronen Funktionsaufrufen Anwendung findet, bei denen im Allgemeinen nicht klar ist, wie lange die Berechnung der (asynchronen) Funktion dauert, beispielsweise wie lange der Download einer Datei dauert, wie lange man auf das Ergebnis eines Webservices warten muss oder auf das Persistieren eines Datensatzes.

Die übergebene Funktion (die *Callback-Funktion* oder auch der *Callback-Handler* genannt) wird aufgerufen, wenn das Ergebnis der asynchronen Funktion bereitsteht. Im Detail widmen wir uns diesem Thema etwas später in diesem Kapitel, der prinzipielle Aufbau des Callback-Entwurfsmusters sei aber bereits an dieser Stelle gezeigt:

```
function asyncFunction(callback) {
  let result = 0;
  /* Hier die Berechnung des Ergebnisses */
  callback(result);
}
```

Listing 2.7 Ein bekanntes Entwurfsmuster, bei dem eine Funktion als Parameter einer anderen Funktion übergeben wird, ist das Callback-Entwurfsmuster.

Funktionen als Rückgabewert verwenden

Funktionen können andere Funktionen als Rückgabewert liefern. Bevor wir uns im späteren Verlauf des Kapitels mit einigen fortgeschrittenen Techniken beschäftigen, die sich diese Tatsache zunutze machen, möchte ich Ihnen im Folgenden zunächst ein etwas einfacheres Beispiel zeigen: eine Funktion, die auf Basis eines Parameters eine andere Funktion zurückgibt. Konkret soll die Funktion für jede der vier Grundrechenarten eine entsprechende Funktion zurückgeben, die die jeweilige Grundrechenart implementiert. Die Grundrechenart wird dabei als String übergeben.

```
function operationFactory(name) {
  switch(name) {
    case 'add': return function(x, y) {
      return x + y;
    }
    case 'subtract': return function(x, y) {
      return x - y;
    }
    case 'multiply': return function(x, y) {
      return x * y;
    }
    case 'divide': return function(x, y) {
      return x / y;
    }
    default: return function() {
      return NaN;
    }
  }
}
```



```
const add = operationFactory('add');
console.log(add(2, 2));                        // Ausgabe: 4
const subtract= operationFactory('subtract');
console.log(subtract(2, 2));                   // Ausgabe: 0
const multiply = operationFactory('multiply');
console.log(multiply(2, 2));                   // Ausgabe: 4
const divide = operationFactory ('divide');
console.log(divide(2, 2));                     // Ausgabe: 1
const unknown = operationFactory('unknown');
console.log(unknown(2, 2));                    // Ausgabe: NaN
```

Listing 2.8 Funktionen können andere Funktionen als Rückgabewert liefern.

Sie sehen hier übrigens direkt einen weiteren praktischen Aspekt von JavaScript-Funktionen: Sie können anonym direkt innerhalb eines Ausdrucks definiert werden, ohne überhaupt einer Variablen zugewiesen worden zu sein. So werden im Beispiel die einzelnen Funktionen, die zurückgegeben werden, direkt »on the fly« hinter dem `return` definiert.

> **Hinweis**
> Hinsichtlich der Performance würde man obigen Quelltext in einem Produktivsystem zwar noch etwas anpassen, da bei mehrmaligem Aufruf von `operationFactory()` mit gleichem Parameter jedes Mal ein neues Funktionsobjekt erzeugt wird. So wäre es sicherlich sinnvoll, hier einen Cache einzubauen. Wie sich so etwas mit Hilfe von funktionalen Techniken realisieren lässt, zeige ich Ihnen in Abschnitt 2.5.3, »Closures«.

Die zurückgegebenen Funktionen können auch direkt aufgerufen werden, ohne zuvor einer Variablen zugewiesen worden zu sein, etwa wie folgt:

```
console.log(operationFactory('add')(2,2));       // Ausgabe: 4
console.log(operationFactory('subtract')(2,2));  // Ausgabe: 0
console.log(operationFactory('multiplay')(2,2)); // Ausgabe: 4
console.log(operationFactory('divide')(2,2));    // Ausgabe: 1
```

Listing 2.9 Zurückgegebene Funktionen können direkt aufgerufen werden.

Auf diese Weise – das sehen Sie in obigem Beispiel ansatzweise – können Sie den Quelltext schön knapp und ausdrucksstark halten. Prinzipiell lässt sich das beliebig fortführen: Zurückgegebene Funktionen könnten ihrerseits ebenfalls eine Funktion als Rückgabewert liefern und diese wiederum usw. Sie denken, so etwas würde man in der Praxis nicht machen? Warten Sie ab, bis Sie später in diesem Kapitel die Currying-Technik kennenlernen.

Hinweis: Arrow-Funktionen

Seit ES2015 können Funktionen dank der sogenannten Arrow-Funktionsschreibweise noch platzsparender und einfacher deklariert werden. Der Code für die `operationFactory` könnte dementsprechend wie folgt vereinfacht werden:

```
function operationFactory(name) {
  switch(name) {
    case 'add': return (x, y) => x + y;
    case 'subtract': return (x, y) => x - y;
    case 'multiply': return (x, y) => x * y;
    case 'divide': return (x, y) => x / y;
    default: return() => NaN;
  }
}
```

Listing 2.10 Arrow-Funktionen vereinfachen die Deklaration von Funktionen. [ES2015]

In Kapitel 4, »ECMAScript 2015 und neuere Versionen«, werden wir uns Arrow-Funktionen etwas mehr im Detail anschauen.

Funktionen innerhalb von Funktionen definieren

Funktionen können auch innerhalb anderer Funktionen definiert werden. Gemeint ist hiermit nicht das Definieren einer Funktion als Methode der anderen Funktion (auch das wäre möglich), sondern die Deklaration einer Funktion lokal **innerhalb des Funktionskörpers** der anderen Funktion. In Listing 2.11 werden beispielsweise die vier Funktionen `add()`, `subtract()`, `multiply()` und `divide()` innerhalb der Funktion `operationsContainer()` definiert:

```
function operationsContainer(x, y) {
  const add = function(x, y) {
    return x + y;
  }
  const subtract = function(x, y) {
    return x - y;
  }
  const multiply = function(x, y) {
    return x * y;
  }
  const divide = function(x, y) {
    return x / y;
  }
  console.log(add(x, y));
  console.log(subtract(x, y));
  console.log(multiply(x, y));
  console.log(divide(x, y));
}

operationsContainer(2,2);
```

Listing 2.11 Funktionen können innerhalb anderer Funktionen definiert werden.

Die Funktionen sind allerdings von außerhalb der Funktion `operationsContainer()` nicht sichtbar, können als von dort nicht direkt aufgerufen werden. Als kleiner Vorgriff auf den nächsten Abschnitt sei an dieser Stelle schon einmal verraten, dass Funktionen einen eigenen Sichtbarkeitsbereich definieren: Alles, was innerhalb einer Funktion definiert wird, ist nur innerhalb der Funktion sichtbar, es sei denn, Sie definieren etwas als global.

In Kapitel 3, »Objektorientierte Programmierung mit JavaScript«, werde ich Ihnen noch einige Techniken vorstellen, mit denen Sie Ihre Daten auf Basis von Funktionen kapseln, aber auch, wie Sie Daten, die innerhalb einer Funktion definiert sind, nach außen zugänglich machen.

Funktionen als Objektmethoden definieren

Wenn Sie eine Funktion innerhalb eines Objekts definieren, spricht man wie erwähnt von einer Methode, einer *Objektmethode*. Aufgerufen wird diese Methode dann über die Objektreferenz. Objektmethoden lassen sich auf folgende Weise definieren:

```
const operations = {
  add: function(x, y) {
    return x + y;
  },
  subtract: function(x, y) {
    return x - y;
  },
  multiply: function(x, y) {
    return x * y;
  },
  divide: function(x, y) {
    return x / y;
  }
}

console.log(operations.add(2,2));
console.log(operations.subtract(2,2));
console.log(operations.multiply(2,2));
console.log(operations.divide(2,2));
```

Listing 2.12 Funktionen können innerhalb von Objekten definiert werden, dann spricht man von Methoden bzw. genauer von Objektmethoden.

Dem kritischen Betrachter wird auffallen, dass die Angabe des `function`-Schlüsselwortes eigentlich überflüssig ist. In Java beispielsweise definiert man Objektmethoden einfacher: Methodenname, geklammerte Parameter sowie geschweifte Klammern für den Methodenkörper – und der Java-Compiler erkennt, dass es sich um eine Methode handelt. Mit ES2015 wurde eine ähnliche Syntax in JavaScript eingeführt, das heißt, Sie können Objektmethoden alternativ wie folgt definieren:

```
const operations = {
  add(x, y) {
    return x + y;
  },
  subtract(x, y) {
    return x - y;
  },
  multiply(x, y) {
    return x * y;
  },
  divide(x, y) {
    return x / y;
  }
}
```

Listing 2.13 Seit ES2015 besteht die Möglichkeit, Objektmethoden ohne das Schlüsselwort »function« zu definieren. [ES2015]

2.1.2 Funktionen haben einen Kontext

Wenn Sie bereits in C# oder in Java programmiert haben, kennen Sie die dortige Bedeutung von `this`. Über dieses Schlüsselwort spricht man innerhalb einer Objektmethode (oder eines Konstruktors) die jeweilige Objektinstanz an, das aktuelle Objekt, eben »dieses« Objekt, das genau jenes ist, für das die Methode definiert wurde (bzw. das eine Instanz der Klasse ist, für die sie definiert wurde).

In JavaScript ist das anders, was nicht selten für Verwirrung sorgt, insbesondere bei Entwicklern, die bereits Erfahrung in einer der oben genannten Sprachen haben. Die unterschiedliche Bedeutung liegt vor allem darin begründet, dass in JavaScript Funktionen selbst Objekte sind und nicht wie in Java und C# zu einem Objekt oder zu einer Klasse »gehören«. Ich habe Ihnen bisher noch nicht gezeigt wie, aber die Dynamik von JavaScript erlaubt es, Funktionen, die an einer Stelle im Code definiert sind, an ganz anderer Stelle wiederzuverwenden, beispielsweise eine global definierte Funktion als Objektmethode oder umgekehrt eine Objektmethode als globale Funktion.

Dies führt dazu, dass sich `this` innerhalb einer Funktion nicht auf das Objekt bezieht, in dem die Funktion **definiert** wurde, sondern auf das Objekt, auf dem die Funktion

ausgeführt wird (*Ausführungskontext*). Sie können sich `this` ein bisschen wie eine Eigenschaft der Funktion vorstellen, die bei deren Aufruf mit dem Wert des Objekts belegt wird, auf dem sie aufgerufen wird (genauer gesagt ist `this` wie schon zuvor `arguments` sogar ein impliziter Parameter, der bei jedem Funktionsaufruf innerhalb der Funktion zur Verfügung steht).

Je nachdem also, ob eine Funktion als globale Funktion oder als Methode eines Objekts aufgerufen wird, hat `this` einen anderen Wert. Betrachten wir dazu zunächst den einfachen Fall einer Objektmethode, in der per `this` eine Eigenschaft des Objekts ausgelesen wird.

```
const person = {
    firstName: 'Max', // Objekteigenschaft
    getFirstName: function() {
        return this.firstName;
    }
}

console.log(person.getFirstName()); // Ausgabe: Max
```

Listing 2.14 »this« im Kontext eines Objekts bezieht sich auf das Objekt.

Die Ausgabe des Programms ist hier wie erwartet `Max`, denn `this` bezieht sich hier auf das Objekt `person`. Das ist intuitiv und leuchtet wahrscheinlich auch jedem bei bloßem Betrachten des Quelltextes ein. So weit also nichts Neues für C#- und Java-Entwickler.

Gehen wir einen Schritt weiter und definieren zusätzlich eine globale Funktion `getFirstNameGlobal()`:

```
function getFirstNameGlobal() {
  return this.name;
}
console.log(getFirstNameGlobal()); // undefined
```

Listing 2.15 Eine einfache globale Funktion, in der »this« verwendet wird

Rufen wir diese Funktion wie in der letzten Zeile in Listing 2.15 aufgerufen, bezieht sie sich auf den globalen Kontext. In diesem Kontext ist die Variable `firstName` nicht definiert, weswegen wir den Wert `undefined` als Rückgabewert erhalten.

Dass sich `this` in einer globalen Funktion auf das globale Objekt bezieht, können Sie einfach testen, indem Sie eine globale Variable `firstName` anlegen und die Funktion `getFirstNameGlobal()` erneut aufrufen:

```
firstName = 'globaler Name';
function getFirstNameGlobal() {
    return this.firstName;
}
console.log(getFirstNameGlobal()); // Ausgabe: globaler Name
```

Listing 2.16 »this« im globalen Kontext bezieht sich auf das globale Objekt.

Das globale Objekt

Das globale Objekt ist von Laufzeitumgebung zu Laufzeitumgebung verschieden. In Browsern ist das globale Objekt das `window`-Objekt, in Node.js ist es ein anderes. Sobald eine Funktion im globalen Scope aufgerufen wird, bezieht sich `this` auf das globale Objekt (außer im strikten Modus: Hier hat `this` innerhalb einer globalen Funktion den Wert `undefined`).

Im strikten Modus führt das obige Programm übrigens zu einem Fehler, da der Zugriff `this.firstName` aufgrund des nicht definierten `this` fehlschlägt:

```
const firstName = 'globaler Name';
function getFirstNameGlobal() {
  return this.firstName;
}
console.log(getFirstNameGlobal()); // Fehler: this ist nicht definiert
```

Listing 2.17 Im strikten Modus ist »this« im globalen Kontext undefiniert.

Nehmen wir jetzt noch zwei Objekte, die jeweils die Eigenschaft `firstName` definieren und die globale Funktion `getFirstNameGlobal()` als Objektmethode wiederverwenden. Dies erreichen Sie, indem Sie von der Objektmethode (`getFirstName()`) wie folgt auf die globale Funktion referenzieren:

```
const anotherPerson = {
  firstName: 'Moritz',
  getFirstName: getFirstNameGlobal
}

const yetAnotherPerson = {
  firstName: 'Peter',
  getFirstName: getFirstNameGlobal
}
console.log(anotherPerson.getFirstName());    // Ausgabe: Moritz
console.log(yetAnotherPerson.getFirstName()); // Ausgabe: Peter
```

Listing 2.18 »this« bezieht sich auf den Kontext der Funktion.

Damit ist klar, dass `this` dynamisch bei Funktionsaufruf gesetzt wird: Die Objektmethode `getFirstNameGlobal()` im Kontext von `anotherPerson` liefert den Wert »`Moritz`, im Kontext von `yetAnotherPerson` den Wert `Peter`. Abbildung 2.3 stellt diesen Zusammenhang grafisch dar.

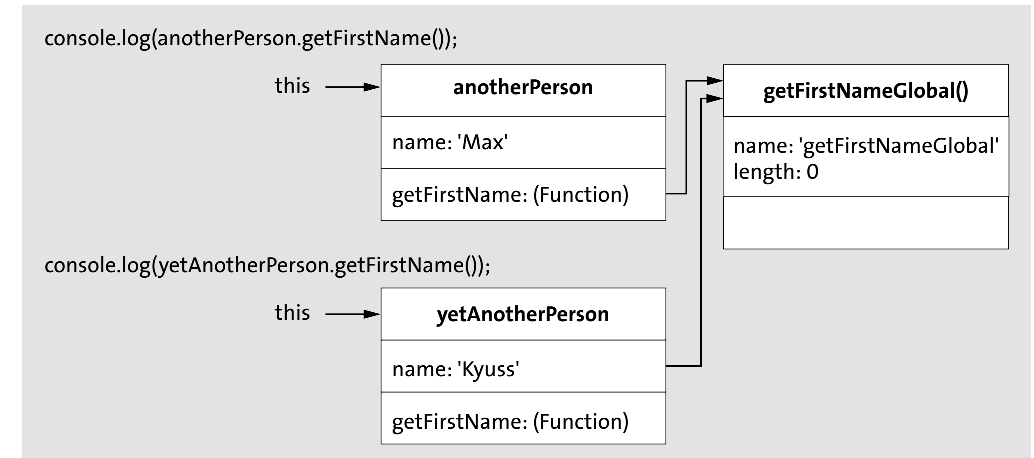

Abbildung 2.3 »this« wird dynamisch bei Funktionsaufruf ermittelt und an die aufgerufene Funktion übergeben.

Die Variable `this` hat also abhängig vom Kontext, in dem die Funktion aufgerufen wird, einen anderen Wert. Zusammenfassend gelten folgende Regeln:

- Bei Aufruf einer globalen Funktion bezieht sich `this` auf das globale Objekt bzw. ist im strikten Modus nicht definiert.
- Wird eine Funktion als Objektmethode aufgerufen, bezieht sich `this` auf das Objekt.
- Wird eine Funktion als Konstruktorfunktion aufgerufen (Details dazu in Abschnitt 2.1.5, »Funktionen als Konstruktorfunktionen«), bezieht sich `this` auf das Objekt, das durch den Funktionsaufruf erzeugt wird.
- Unachtsam programmiert, sorgen insbesondere folgende vier Fälle in der Praxis recht häufig für Laufzeitfehler (der Einfachheit halber nenne ich Funktionen, die auf `this` zugreifen, `this`-Funktion):
  - wenn eine `this`-Funktion einer Variablen zugewiesen wird
  - wenn eine `this`-Funktion als Callback einer anderen Funktion verwendet wird
  - wenn sich ein Objekt eine `this`-Funktion eines anderen Objekts »leiht« (*Function Borrowing* bzw. *Method Borrowing*)
  - wenn `this` innerhalb einer inneren Funktion vorkommt

Problematisch sind diese Fälle, weil sie oft dazu führen, dass der Ausführungskontext einer Funktion nicht dem entspricht, was man als Entwickler erwartet. In Abschnitt 2.2, »Standardmethoden jeder Funktion«, werde ich Ihnen diesbezüglich die Standardmethoden `bind()`, `call()` und `apply()` vorstellen, mit denen Sie den Ausführungskontext einer Funktion dynamisch definieren.

2.1.3 Funktionen definieren einen Sichtbarkeitsbereich

Im Gegensatz zu vielen anderen Programmiersprachen kennt JavaScript keinen Block-Scope für Variablen, mit anderen Worten: `{` und `}` spannen keinen Gültigkeitsbereich bzw. Sichtbarkeitsbereich für Variablen auf. Stattdessen wird der Gültigkeitsbereich von solchen Variablen durch die umgebende Funktion begrenzt.

Man spricht daher auch von *Function-Level-Scope*: Variablen, die innerhalb einer Funktion definiert werden, sind innerhalb der gesamten Funktion sichtbar sowie innerhalb anderer (innerer) Funktionen, die in der (äußeren) Funktion definiert sind.

Hinweis

Dieses Verhalten gilt zumindest für Variablen, die über das Schlüsselwort `var` angelegt werden, weswegen wir in den folgenden Beispielen die Variablen auch über dieses Schlüsselwort erzeugen werden. Bei `let` sieht das etwas anders aus, wie Sie im Verlauf des Buches noch sehen werden.

Lassen Sie mich die Besonderheiten hierbei anhand einiger Codebeispiele erläutern. Dazu vorab ein paar Regeln, die beim Zugriff auf Variablen gelten:

- Zugriff auf Variablen, die deklariert, aber nicht initialisiert sind, ergibt den Wert `undefined`:

```
function example() {
  var y;
  console.log(y);
}
example(); // Ausgabe: undefined
```

- Zugriff auf Variablen, die nicht deklariert sind, führt zu einem `ReferenceError`:

```
function example() {
  console.log(y);
}
example(); // ReferenceError
```

- Zugriff auf Variablen, die deklariert und initialisiert sind, ergibt (nicht anders zu erwarten und nur der Vollständigkeit halber aufgeführt) den Wert der Variablen:


```
function example() {
  var y = 4711;
  console.log(y);
}
example(); // Ausgabe: 4711
```

Werfen Sie jetzt einen Blick auf Listing 2.19: Trotz der Tatsache, dass die Variablen `y` und `i` innerhalb der `if`- bzw. innerhalb der `for`-Anweisung deklariert und initialisiert werden, kann von außerhalb der jeweiligen Codeblöcke auf beide Variablen zugegriffen werden. Ausgegeben wird zweimal der Wert `4711`.

```
function example(x) {
  if(x) {
    var y = 4711;
  }
  for(var i=0; i<4711; i++) {
    /* Irgendwas machen */
  }
  console.log(y);
  console.log(i);
}
example(true);
```

Listing 2.19 Variablen sind überall in einer Funktion sichtbar.

Verschieben wir jetzt die Konsolenausgaben an den Beginn der Funktion, könnte man meinen, dass die Ausgabe nicht mehr `4711` ist, sondern ein `ReferenceError` erzeugt wird. Laut der Beschreibung von oben tritt dieser ja immer dann auf, sobald man versucht, auf eine Variable zuzugreifen, die noch nicht deklariert ist. Doch diese Vermutung bestätigt sich nur zur Hälfte: Der Wert `4711` wird tatsächlich nicht mehr ausgegeben (das wäre auch mehr als merkwürdig). Allerdings kommt es auch nicht zum erwarteten `ReferenceError`. Die Ausgabe ist stattdessen zwei Mal `undefined`:

```
function example(x) {
  console.log(y);
  console.log(i);
  if(x) {
    var y = 4711;
  }
  for(var i=0; i<4711; i++) {
    /* Irgendwas machen */
```

```
  }
}
example(true); // Ausgabe: undefined
```

Listing 2.20 Variablendeklarationen sind bereits zu Beginn einer Funktion bekannt

Der Grund hierfür ist, dass alle Variablendeklarationen bereits zu Beginn der Funktion bekannt sind. Alle Deklarationen innerhalb eines Sichtbarkeitsbereichs werden vom JavaScript-Interpreter nämlich bereits am Anfang des jeweiligen Bereichs ausgeführt, unabhängig davon, an welcher Stelle die Deklaration eigentlich steht.

Der obige Quelltext wird vom Interpreter wie folgt interpretiert:

```
function example(x) {
  var y;
  var i;
  console.log(y);
  console.log(i);
  if(x) {
    y = 4711;
  }
  for(i=0; i<4711; i++) {
  }
}
example(true);
```

Listing 2.21 Hoisting von Variablen

Dieses Verhalten wird *Hoisting* oder *Variablen-Hoisting* genannt: Die Variablendeklarationen werden an den Beginn der jeweiligen Funktion »gehoben« (von engl. *to hoist* – heben). Um sich dieses Hoistings bewusst zu sein (bzw. es gar nicht erst dazu kommen zu lassen), hat es sich etabliert, alle Variablen einer Funktion bereits zu Beginn der Funktion in einer einzigen Anweisung zu deklarieren. Dies gilt als guter Stil, sorgt dafür, dass es keine Verwirrung bezüglich des Variablen-Hoistings gibt, und verhindert zudem, dass man versehentlich zwei Variablen mit gleichem Namen anlegt.

```
function example(x) {
  var y, i;
  console.log(y);
  console.log(i);
  if(x) {
    y = 4711;
  }
  for(i=0; i<4711; i++) {
```



```
  }
}
example(true);
```

Listing 2.22 Best Practice für die Deklaration von Variablen

Implizite globale Variablen

Geben Sie bei einer Variablendeklaration weder `var` noch `let` noch `const` an, wird die Variable automatisch als globale Variable angelegt. Dies gilt es zu verhindern, denn dadurch wird der Code relativ unübersichtlich, und es kann zu Namenskonflikten und ungewollten Seiteneffekte kommen, beispielsweise wenn Sie eine bereits existierende (globale) Variable, die eigentlich für einen anderen Zweck vorgesehen ist, dadurch überschreiben.

Später in diesem Kapitel werden Sie mit dem *IIFE-Entwurfsmuster* (*Immediately-Invoked Function Expression*) eine gängige Technik kennenlernen, die häufig innerhalb einer Funktion dazu verwendet wird, verschiedene voneinander abgeschirmte Sichtbarkeitsbereiche zu definieren.

2.1.4 Alternativen zum Überladen von Methoden

In Sprachen wie C# oder Java ist es möglich, innerhalb einer Klasse mehrere Methoden mit gleichem Namen zu definieren, die sich nur in den Typen bzw. der Anzahl der Parameter unterscheiden. Dies ist möglich, da in diesen Sprachen die Eindeutigkeit einer Methode innerhalb einer Klasse aus der Kombination von Methodennamen und Parametern ermittelt wird. Oder anders gesagt: Die **Signatur** einer Methode muss innerhalb einer Klasse eindeutig sein, nicht der Name. Anhand der Anzahl der Parameter und der Typen der Parameter wird dann ermittelt, welche Methode aufgerufen wird.

Das Überladen von Methoden kommt in der Regel dann zum Einsatz, wenn man die implementierte Funktionalität für verschiedene Typen von Parametern oder eine variierende Anzahl von Parametern zur Verfügung stellen möchte. Der Vorteil ist dabei, dass Sie sich nicht für jede Konstellation von Parametern einen neuen Methodennamen überlegen müssen. Um in Java beispielsweise ein Objekt vom Typ `Person` anhand verschiedener Parameter zu erstellen, könnten Sie dort zwei (überladene) Methoden verwenden:

```
public Person createPerson(String firstName, String lastName) {
  ... // weiterer Code
}
```

```
public Person createPerson(String firstName, String lastName, int age) {
  ... // weiterer Code
}
```

Listing 2.23 Überladen von Methoden in Java

In JavaScript ist das Überladen von Methoden bzw. Funktionen aus drei Gründen nicht so ohne weiteres möglich: JavaScript kennt keine Typangaben innerhalb der Funktionsdeklaration, anhand der, wie oben beschrieben, entschieden werden könnte, welche Methode aufgerufen wird. Zweitens können Funktionen in JavaScript mit einer beliebigen Anzahl an Parametern aufgerufen werden, auch wenn diese nicht explizit in der Funktionsdeklaration angegeben werden. Drittens sind Objektmethoden ja letztendlich nichts anderes als Objekteigenschaften mit dahinterliegender Funktion, und es kann keine Objekte mit zwei gleichbenannten Eigenschaften geben. Definieren Sie mehrere Funktionen mit gleichem Namen in einem Kontext (beispielsweise in einem Objekt), überschreibt die zuletzt definierte Funktion alle vorhergehenden gleichnamigen Funktionen. Folgender Code würde beispielsweise lediglich dazu führen, dass die erste Version von `createPerson()` durch die zweite Implementierung überschrieben würde:

```
function createPerson(firstName, lastName) {
  return {
    firstName: firstName,
    lastName: lastName
  };
}

function createPerson(firstName, lastName, age) {
  return {
    firstName: firstName,
    lastName: lastName,
    age: age
  };
}
```

Listing 2.24 Trotz verschiedener Anzahl an Parametern überschreiben sich gleichnamige Funktionen.

Achtung

Übrigens werden Funktionen auch überschrieben, wenn im gleichen Kontext eine gleich benannte Variable deklariert wird. Mit `createPerson = true;` beispielsweise würden oben genannte Funktionen ebenfalls überschrieben.

Um in JavaScript überladene Methoden bzw. Funktionen nachzubilden, kommen Sie also gar nicht darum herum, die Funktionalität, die im Fall von C# und Java auf mehrere Methoden verteilt ist, innerhalb einer Funktion zu implementieren, die dynamisch die Anzahl und die Typen der beim Funktionsaufruf übergebenen Parameter ermittelt und abhängig davon die jeweilige Funktionalität durchführt. Dazu haben Sie verschiedene Möglichkeiten, die ich in den folgenden Unterabschnitten erläutern werde.

Explizite Angabe aller Parameter

Da Sie Funktionen auch mit weniger als den in einer Funktionsdeklaration angegebenen Parametern aufrufen können, ist die explizite Angabe aller Parameter in der Funktionsdeklaration die offensichtlichste Lösung: Pflichtparameter geben Sie dabei zuerst, optionale Parameter zuletzt an. Innerhalb der Funktion wird dann anhand der optionalen Parameter entschieden, welche (optionalen) Programmzweige innerhalb der Funktion ausgeführt werden.

Eine Additionsfunktion, die optional das Ergebnis auf der Konsole ausgibt, könnten Sie auf diese Weise wie folgt implementieren:

```
function add(x, y, log) {
  const result = x + y;
  if(log) {
    console.log(result);
  }
  return result;
}
add(2,2);       // Aufruf ohne Logging
add(2,2,true);  // Aufruf mit Logging
```

Listing 2.25 Optionale Parameter, Möglichkeit 1:
konkrete Angabe der optionalen Parameter

Allerdings hat diese Vorgehensweise auch Nachteile, und zwar, wenn es mehrere optionale Parameter gibt, Sie aber nur einen der »hinteren« optionalen Parameter beim Funktionsaufruf angeben möchten, denn in diesem Fall müssen Sie auch für alle vorangehenden Parameter entsprechende Werte übergeben.

Weglassen der optionalen Parameter

Wie Sie bereits wissen, besteht innerhalb einer Funktion über das `arguments`-Objekt Zugriff auf alle Parameter, die beim Funktionsaufruf übergeben wurden. Alternativ zur obigen Variante ist es somit möglich, die optionalen Parameter nicht explizit in der Funktionsdeklaration anzugeben, sondern implizit über das `arguments`-Objekt zu handhaben. Die Additionsfunktion von eben sähe damit wie folgt aus:

```
function add(x, y) {
  const result = x + y;
  if(arguments[2]) { // Zugriff auf den optionalen Parameter
    console.log(result);
  }
  return result;
}
add(2,2);       // Aufruf ohne Logging
add(2,2,true);  // Aufruf mit Logging
```

Listing 2.26 Optionale Parameter, Möglichkeit 2: Weglassen der optionalen Parameter und Zugriff auf das »arguments«-Objekt

> **Hinweis**
> Seit ES2015 steht mit sogenannten *Rest-Parametern* eine einfachere Möglichkeit zur Verfügung, auf optionale Parameter zuzugreifen.

Der Nachteil ist aber auch hier wie in der ersten Lösung der gleiche: Optionale Parameter, die nicht benötigt werden, müssen trotzdem im Funktionsaufruf übergeben werden.

Optionale Parameter als Konfigurationsobjekt

Um dieser Problematik entgegenzuwirken, hat sich als Best Practice für optionale Parameter das sogenannte *Konfigurationsobjekt* durchgesetzt. Die Idee ist dabei folgende: Statt einzelner optionaler Parameter werden Parameter in einem Objekt zusammengefasst, das als Parameter der entsprechenden Funktion übergeben wird. Anstatt innerhalb der Funktion auf die Parameter zuzugreifen, wird dann auf die äquivalenten Eigenschaften des Konfigurationsobjekts zugegriffen.

```
function add(x, y, config) {
  const result = x + y;
  if(config && config.log) {
    console.log(result);
  }
  return result;
}
add(2,2);                // Aufruf ohne Logging
add(2,2,{ log: true });  // Aufruf mit Logging
```

Listing 2.27 Optionale Parameter, Möglichkeit 3: Zusammenfassen aller optionalen Parameter in einem Konfigurationsobjekt

2.1.5 Funktionen als Konstruktorfunktionen

In JavaScript gibt es keine Konstruktoren im dem Sinne, wie es sie in C# oder Java gibt. Um neue Objektinstanzen erzeugen zu können, ist es aber möglich, eine Funktion als Konstruktorfunktion aufzurufen. Dabei stellen Sie dem Funktionsaufruf das Schlüsselwort `new` voran. Nach Konvention werden Funktionen, die als Konstruktorfunktion aufgerufen werden können, großgeschrieben.

```
function Album(title) {
  this.title = title;
}
const album = new Album('Sky Valley');
console.log(album.title);  // Ausgabe: "Sky Valley"
```

Listing 2.28 Deklaration und Aufruf einer Konstruktorfunktion

Über die Objekteigenschaft `constructor` des auf diese Weise erstellten Objekts lässt sich die Konstruktorfunktion ermitteln, mit der das Objekt erzeugt wurde:

```
console.log(album.constructor); // [Function: Album]
```

Listing 2.29 Die »constructor«-Eigenschaft zeigt auf die Konstruktorfunktion.

In Abschnitt 3.3.2 werden wir uns im Rahmen der *pseudoklassischen Vererbung* erneut den Konstruktorfunktionen zuwenden.

2.2 Standardmethoden jeder Funktion

Funktionen sind Objekte, was bedeutet, dass sie ihrerseits Methoden enthalten können. Standardmäßig stellt jede Funktion, die Sie implementieren, bereits drei Methoden zur Verfügung: `apply()`, `call()` und `bind()`. Was diese Methoden machen, worin sie sich unterscheiden und in welchen Fällen man sie verwendet, zeige ich Ihnen in diesem Abschnitt.

2.2.1 Objekte binden mit der Methode »bind()«

Sie haben bereits gesehen, dass sich `this` innerhalb einer Funktion auf den Kontext bezieht, in dem die Funktion aufgerufen wird, nicht auf den, in dem sie definiert wurde. Dieses Verhalten ist nicht immer wünschenswert. Stellen Sie sich vor, Sie möchten eine Objektmethode, die auf `this` zugreift, als Parameter einer Funktion übergeben, beispielsweise als *Callback-Handler* (siehe auch Abschnitt 2.5.7, »Das Callback-Entwurfsmuster«). Dann kann dies zu einem Laufzeitfehler führen, wenn diese übergebene Funktion innerhalb der anderen Funktion aufgerufen wird, wie Listing 2.30 zeigt:

```
const button = {
  handler : null,
  // Funktion, die einen Callback-Handler erwartet
  onClick : function(handler) {
    this.handler = handler;
  },
  click : function() {
    this.handler();
  }
};

const handler = {
  log : function() {
    console.log('Button geklickt.');
  },
  // Objektmethode, die weiter unten als Callback-Handler registriert wird
  handle: function() {
    this.log();
  }
}
// Registrieren des Callback-Handlers
button.onClick(handler.handle);
// Implizites Aktivieren des Callback-Handlers
button.click();
```

Listing 2.30 Wird eine Objektmethode außerhalb eines Objekts verwendet, bezieht sich »this« nicht mehr auf das Objekt.

Das Programm endet mit einem Fehler `TypeError: Object #<Object> has no method 'log'`. Das Problem ist die Übergabe von `handler.handle` als Callback-Funktion an die `onClick()`-Methode von `button`. Sobald diese Callback-Funktion innerhalb von `click()` aufgerufen wird (`this.handler()`), bezieht sich `this` innerhalb der Callback-Funktion nicht auf das Objekt `handler`, sondern auf das Objekt `button`, das wiederum keine Methode `log()` hat.

Wie lässt sich das Problem lösen? Relativ einfach, wenn man weiß, wie. Über die Funktion `bind()` lässt sich `this` für eine Funktion an ein bestimmtes Objekt (den Ausführungskontext) binden. `bind()` rufen Sie dabei auf der entsprechenden Funktion auf, als Parameter übergeben Sie optional das Objekt, das den Ausführungskontext darstellt. Hat die Funktion ihrerseits Parameter, werden diese einfach hinten angehängt. Als Ergebnis liefert `bind()` ein neues Funktionsobjekt, das identisch mit der Ursprungsfunktion ist, den Ausführungskontext aber an das übergebene Objekt gebunden hat.


```
button.onClick(handler.handle.bind(handler));
```

Listing 2.31 Über »bind()« lässt sich der Ausführungskontext einer Funktion definieren.

Der Übersichtlichkeit halber ließe sich auch Folgendes schreiben:

```
const functionBoundToHandler = handler.handle.bind(handler);
button.onClick(functionBoundToHandler);
```

Listing 2.32 Explizite Zuweisung der gebundenen Funktion zu einer Variablen

Hinweis

Mit dem jetzigen Wissen müssen wir die Bedeutung von `this` etwas präzisieren: Ich hatte vorhin gesagt, dass sich »`this` innerhalb einer Funktion auf den Kontext bezieht, in dem die Funktion aufgerufen wird, nicht auf den, in dem sie definiert wurde«". Exakter ist aber, zu sagen, dass sich `this` innerhalb einer Funktion auf den Kontext bezieht, **in dem die Funktion gebunden ist**.

Hintergrund

Die Methode `bind()` steht erst seit ES5 zur Verfügung. Sollten Sie eine Anwendung für einen älteren Browser programmieren, der diese Version von ECMAScript noch nicht unterstützt, gibt es die Möglichkeit – funktionalen Techniken sei Dank –, `bind()` selbst zu implementieren. Diesen Sonderfall wollen wir an dieser Stelle aber nicht betrachten. Bei Interesse und Bedarf finden Sie entsprechende Implementierungen relativ einfach im Internet.

Noch ein Hinweis an dieser Stelle: Die obige Problemstellung lässt sich alternativ über eine anonyme Funktion lösen, die als Callback übergeben wird und den Aufruf an das `handler`-Objekt steuert:

```
button.onClick(function() {
  handler.handle();
});
```

Listing 2.33 Anonyme Funktionen stellen häufig eine Alternative zu direkt übergebenen Funktionen dar.

Das funktioniert deshalb, weil jetzt `handle()` auf dem Objekt `handler` aufgerufen wird (nicht wie zuvor im Kontext von `button`) und sich `this` somit auf `handler` bezieht.

2.2.2 Funktionen aufrufen über die Methode »call()«

Mit der Methode `call()` ist es ebenfalls möglich, den Ausführungskontext einer Funktion zu definieren. Allerdings wird bei `call()` nicht wie im Fall von `bind()` ein

neues Funktionsobjekt erstellt, sondern die entsprechende Funktion direkt aufgerufen. Der Ausführungskontext wird dabei als erster Parameter übergeben, optional können Sie über weitere Parameter die Funktionsparameter der aufzurufenden Funktion definieren.

Ein besonders häufig anzutreffender Anwendungsfall für die Verwendung von `call()` ist die Iteration über das `arguments`-Objekt unter Verwendung der `forEach()`-Methode. Letztere steht eigentlich nur echten Arrays zur Verfügung, `arguments` ist jedoch – wie Sie wissen – kein echtes Array. Folglich würde folgender Quelltext zu einem Fehler führen:

```
function printNames() {
  console.log(arguments); // Ausgabe: { '0': 'Max', '1': 'Moritz' }
  /* Fehler: arguments ist kein Array
  arguments.forEach(function(argument) {
    console.log(argument);
  });
  */
}
printNames('Max', 'Moritz');
```

Listing 2.34 Da das »arguments«-Objekt kein echtes Array ist, verfügt es nicht über die »forEach()«-Methode.

Die Funktionalität der `forEach()`-Methode lässt sich allerdings auch für arrayähnliche Objekte wie `arguments` verwenden. Der Fachbegriff für diese Technik lautet *Function Borrowing* bzw. *Method Borrowing*, also so viel wie das »Ausleihen« von Funktionen oder Methoden.

Die einfachste Möglichkeit, über ein arrayähnliches Objekt zu iterieren, sehen Sie in Listing 2.35:

```
function printNames() {
  Array.prototype.forEach.call(arguments, function(argument) {
    console.log(argument);
  });
}
printNames('Max', 'Moritz');
```

Listing 2.35 Iteration über das Array-ähnliche Objekt »arguments« durch Ausleihen der Methode »forEach()«

Was passiert hier genau? Zunächst einmal greifen wir mit `Array.prototype` auf den globalen Array-Typ bzw. seinen Prototyp zu. Dieser Prototyp enthält all jene Methoden, die den Array-Instanzen im Rahmen der prototypischen Vererbung zur Verfügung stehen. Die Methode `forEach()` ist eine davon. Mit `Array.prototype.forEach`

greifen Sie auf diese Methode zu, ohne sie aufzurufen. Der Aufruf geschieht erst durch das folgende `call()`.

Der erste Parameter von `call()` definiert den Ausführungskontext, im Beispiel das Objekt `arguments`. Der zweite Parameter ist derjenige, der an die Methode `forEach()` weitergereicht wird: eine Callback-Funktion, die für jedes Element im Array bzw. hier im `arguments`-Objekt aufgerufen wird.

2.2.3 Funktionen aufrufen über die Methode »apply()«

Die Methode `apply()` funktioniert vom Prinzip her ähnlich wie die Methode `call()`, mit dem Unterschied, dass die Parameter der aufgerufenen Funktion nicht als einzelne Parameter übergeben werden, sondern gesammelt als Array. Das Beispiel von eben ließe sich mit `apply()` wie folgt realisieren:

```
function printNames() {
  Array.prototype.forEach.apply(arguments, [function(argument) {
    console.log(argument);
  }]);
}
printNames('Max', 'Moritz');
```

Listing 2.36 »apply()« funktioniert ähnlich wie »call()«, erwartet als zweiten Parameter aber ein Array statt einzelner Werte.

Anwendungsbeispiel: Aufruf variadischer Funktionen

Ein weiterer bekannter Anwendungsfall für die Methode `apply()` ist der Aufruf sogenannter *variadischer Funktionen*, also solcher Funktionen, die mit einer variablen Anzahl an Parametern aufgerufen werden können. Stehen die Parameter als Array zur Verfügung, ist ein Aufruf solcher Funktionen über `apply()` bequemer, als manuell die einzelnen Array-Einträge auf die Funktionsparameter zu verteilen.

Ein Beispiel für eine variadische Funktion ist die Methode `max()` des `Math`-Objekts aus der JavaScript-Standardbibliothek:

```
console.log(Math.max(24, 44));
console.log(Math.max(24, 14, 44, 88));
console.log(Math.max(24, 14, 4711, 44, 88));
console.log(Math.max(24, 14, 5678, 4711, 44, 88));
```

Listing 2.37 Variadische Funktionen können mit einer beliebigen Anzahl an Parametern aufgerufen werden.

Allerdings ist es bei dieser Methode nicht möglich, alternativ ein Array als (einzigen) Parameter zu übergeben. Folgender Quelltext ergibt die Ausgabe `NaN`:

```
const numbers = [24, 14, 44, 88];
console.log(Math.max(numbers));
```

Listing 2.38 Dieser Aufruf schlägt fehl, weil »Math.max()« nicht mit Arrays umgehen kann.

Die Methode müsste umständlich mit allen einzelnen Werten des Arrays aufgerufen werden:

```
const numbers = [24, 14, 44, 88];
console.log(Math.max(
  numbers[0],
  numbers[1],
  numbers[2],
  numbers[3]
));
```

Listing 2.39 Unbequemer Aufruf einer variadischen Funktion

Mit `apply()` geht das dagegen einfacher:

```
console.log(Math.max.apply(null, numbers));
```

Listing 2.40 Bequemer Aufruf einer variadischen Funktion

> **Hinweis**
> Wenn der Ausführungskontext keine Rolle spielt, ist es Best Practice, stattdessen wie im Beispiel den Wert `null` zu übergeben.

2.3 Einführung in die funktionale Programmierung

Nachdem Sie nun die Grundlagen von Funktionen und deren Standardmethoden kennen, schauen wir uns als Nächstes die sogenannte *funktionale Programmierung* an. Bei dieser Art der Programmierung liegt der Fokus im Gegensatz zur prozeduralen oder objektorientierten Programmierung auf Funktionen, nicht auf Objekten.

2.3.1 Eigenschaften funktionaler Programmierung

Die besonderen Merkmale bei der funktionalen Programmierung sind folgende:

- Funktionen sind erstklassige Objekte. Was das bedeutet, haben Sie in diesem Kapitel bereits gesehen: Funktionen können Variablen zugewiesen werden, können als Parameter anderer Funktionen verwendet werden oder als deren Rückgabewert.

- Die Datenstrukturen bei der funktionalen Programmierung sind in der Regel unveränderlich bzw. werden nicht verändert. Operationen, die auf Datenstrukturen durchgeführt werden, erzeugen falls nötig neue Datenstrukturen. In *rein funktionalen* Programmiersprachen können beispielsweise Listen oder andere Datenstrukturen, die einmal angelegt worden sind, nachträglich nicht mehr geändert werden. Das heißt, es können weder Elemente aus der Liste gelöscht noch ihr hinzugefügt werden (JavaScript übrigens ist keine rein funktionale Programmiersprache).
- Fassen wir den vorigen Punkt etwas weiter, kommt noch hinzu, dass bei der funktionalen Programmierung die Funktionen in der Regel überhaupt keine Nebeneffekte haben und sich eher wie mathematische Funktionen verhalten. Das heißt außerdem, dass sie für gleiche Eingaben immer das gleiche Ergebnis liefern.
- Funktionale Programme sind deklarativ: Man sagt, **was** das Programm macht, **nicht wie** es etwas macht.
- Funktionale Programme sind in der Regel schlanker als die äquivalente Variante in der objektorientierten oder imperativen Programmierung.

2.3.2 Unterschied zur objektorientierten Programmierung

Bei der objektorientierten Programmierung sind die Daten und die Operationen, die man auf den Daten anwendet, eng aneinander gebunden. Die Objekte, die die Daten enthalten, definieren in der Regel auch die Methoden (bzw. Operationen), die diese Daten verwenden. Bei der funktionalen Programmierung sind die Daten dagegen nur lose an die Funktionen gekoppelt. Sprich: Sie können durchaus Funktionen implementieren, die auf verschiedenen Datenstrukturen arbeiten, oder Datenstrukturen, auf denen verschiedene Funktionen ausgeführt werden können.

Der Begriff *Komposition* in der objektorientierten Programmierung bezieht sich auf die Komposition von Objekten: Neue Objekte erzeugt man, indem man von bestehenden Objekten (bzw. deren zugehörigen Klassen) ableitet und diesen neues (oder zusätzliches) Verhalten in Form von Methoden hinzufügt. *Komposition* in der funktionalen Programmierung bedeutet die Komposition von (einfachen) Funktionen hin zu neuen (komplexeren) Funktionen.

> **Hinweis**
> Trotz der unterschiedlichen Denkweisen sind funktionale und objektorientierte Programmierung keine gegensätzlichen Konzepte, sondern zwei, die sich sehr gut ergänzen.

2.3.3 Unterschied zur imperativen Programmierung

Während man bei der funktionalen Programmierung beschreibt, **was** gemacht werden soll, beschreibt man bei der imperativen Programmierung, **wie** etwas gemacht werden soll. Imperative Programme verwenden dazu explizite Schleifenanweisungen (`while`-Schleifen, `for`-Schleifen etc.), bedingte Anweisungen (`if else`) und Sequenzen davon. Bei der objektorientierten Programmierung beispielsweise wird häufig innerhalb von Methoden zu großen Teilen imperativ programmiert, weil objektorientierte Sprachen häufig keine funktionalen Konzepte kennen. In JavaScript können Sie beides miteinander kombinieren.

2.3.4 Funktionale Programmiersprachen und JavaScript

Beispiele für (nahezu) reine funktionale Programmiersprachen sind Haskell, Lisp und Miranda. JavaScript ist, wie bereits gesagt, keine rein funktionale Programmiersprache. So können beispielsweise Datenstrukturen wie Arrays auch nach ihrer Definition noch verändert werden, in Haskell hingegen ist dies nicht erlaubt.

Außerdem können Funktionen in JavaScript durchaus für gleiche Eingaben unterschiedliche Ergebnisse liefern. In rein funktionalen Sprachen ist dies nicht ohne weiteres möglich. Zudem bietet JavaScript neben funktionalen Aspekten auch prototypische und objektorientierte Aspekte, beides ebenfalls in rein funktionalen Sprachen nicht vorhanden.

Hinzu kommt, dass nicht alle funktionalen Konzepte in JavaScript zur Verfügung stehen, beispielsweise *Homoikonizität*, *Lazy Evaluation* und *Pattern Matching*.

2.4 Von der imperativen Programmierung zur funktionalen Programmierung

Im Folgenden möchte ich Ihnen zeigen, welche Vorteile die funktionale Programmierung gegenüber der imperativen Programmierung bietet. Dazu stelle ich Ihnen vier Methoden vor, die in JavaScript für Arrays zur Verfügung stehen: `forEach()` für die Iteration über Arrays (diese Methode kennen Sie bereits aus diesem Kapitel), `map()` für das Abbilden von Elementen eines Arrays auf neue Elemente in einem anderen Array, `filter()` für das Filtern von Elementen in einem Array und schließlich `reduce()` für das Reduzieren der Elemente eines Arrays auf einen einzelnen Wert.

2.4.1 Iterieren mit der Methode »forEach()«

Wenn Sie über ein Array oder eine Liste iterieren wollen, haben Sie gleich mehrere Möglichkeiten: Sie haben die Auswahl zwischen einer `for`-Schleife, einer `while`-Schleife und einer `do-while`-Schleife.

Die normale `for`-Schleife kennen Sie ja bereits aus Kapitel 1, »Einführung«:

```
const artists = [
  'Kyuss',
  'Dozer',
  'Spiritual Beggars',
  'Monster Magnet'
];
for(let i= 0, l=artists.length; i<l; i++) {
  console.log(artists[i]);
}
```

Listing 2.41 Iteration mit einer normalen »for«-Schleife

Diese `for`-Schleife, bestehend aus Initialisierung, Bedingung und Inkrementierungsausdruck, ist ein klassisches Beispiel für imperative Programmierung. Der Quelltext in Listing 2.41 soll lediglich jeden Wert des Arrays ausgeben. Das ist das, **was** passieren soll. Der Code beschäftigt sich dabei allerdings viel zu sehr damit, **wie** das Ganze vonstattengehen soll: Zählervariable initialisieren, Abbruchbedingung überprüfen, indexbasierter Zugriff auf das Array und anschließend Hochzählen der Zählervariablen. Wie oft hat man als Entwickler diese Schritte schon niedergetippt und sich im Stillen gefragt: Geht das nicht auch einfacher?

Die Antwort lautet: Ja, es geht einfacher. Zumindest in JavaScript. Und zwar mit der Methode `forEach()`, die ich Ihnen in Abschnitt 2.2.2, »Funktionen aufrufen über die Methode »call()««, schon kurz vorgestellt habe. Als Parameter übergeben Sie dieser Methode eine Funktion, die ihrerseits für jedes Element im Array mit drei Parametern aufgerufen wird: dem jeweiligen Element, dem Index des Elements und dem Array selbst.

Der imperative Code von eben ließe sich mit `forEach()` folgendermaßen implementieren:

```
const artists = [
  'Kyuss',
  'Dozer',
  'Spiritual Beggars',
  'Monster Magnet'
];
```

```
artists.forEach((artist, index, artists) => {
  console.log(artist);
});
```

Listing 2.42 Funktionale Iteration über die Methode »forEach()«

Auch wenn der Code bezogen auf die Anzahl der Zeilen nicht unbedingt kürzer wird, ist er doch schon um einiges lesbarer: Im Gegensatz zur imperativen Variante liegt jetzt der Fokus auf der Logik (dem Was), nicht auf der Schleife (dem Wie).

Hinweis

In der Praxis ist es häufig so, dass man die letzten beiden Parameter der Callback-Funktion (in Listing 2.42 `index` und `artists`) gar nicht mit in der Funktionsdeklaration aufführt, da man oft ohnehin innerhalb der Funktion nur auf das Element selbst zugreifen möchte.

2.4.2 Werte abbilden mit der Methode »map()«

Die Methode `map()` bietet sich an, wenn man nicht nur über die Elemente eines Arrays iterieren, sondern zeitgleich für jedes Element einen Wert ermitteln und diesen Wert in einem neuen Array speichern möchte, beispielsweise um für ein Array natürlicher Zahlen zu jeder Zahl das Quadrat zu berechnen oder (weniger langweilig) für ein Array von Objekten von jedem Objekt eine Eigenschaft auszulesen und den entsprechenden Wert abzuspeichern.

Imperativ würde man diese Problemstellung wie folgt lösen:

```
const artists = [
  {
    name: 'Nick Cave'
  },
  {
    name: 'Ben Harper'
  }
];
const names = [];
for(let i=0; i<artists.length; i++) {
  names.push(artists[i].name);
}
console.log(names); // ['Nick Cave', 'Ben Harper']
```

Listing 2.43 Erstellen eines Arrays auf Basis eines vorhandenen Arrays mit imperativer Programmierung

Wie schon im `forEach()`-Beispiel nimmt auch in diesem Code die `for`-Schleife einen großen Teil des Codes in Anspruch. Mit der Methode `map()` dagegen wird der Code deutlich sprechender. Als Parameter erwartet diese Methode eine Funktion, die dann für jedes Element im Array aufgerufen wird. Der Rückgabewert dieser Funktion bestimmt dabei den Wert, der für das jeweilige Element in das Ziel-Array geschrieben werden soll.

```
const artists = [
  {
    name: 'Nick Cave'
  },
  {
    name: 'Ben Harper'
  }
];
const names = artists.map(
  (artist, index, artists) => artist.name
);
console.log(names); // ['Nick Cave', 'Ben Harper']
```

Listing 2.44 Erstellen eines Arrays auf Basis eines vorhandenen Arrays mit funktionaler Programmierung

2.4.3 Werte filtern mit der Methode »filter()«

Eine weitere, häufig anzutreffende Problemstellung ist, aus einem bestehenden Array eine gewisse Anzahl an Elementen anhand bestimmter Kriterien herauszufiltern. Nehmen wir als Beispiel ein Array, das verschiedene Objekte enthält, die jeweils ein Musikalbum repräsentieren. Ausgehend von diesem Array sollen alle Alben herausgefiltert werden, die vor der Jahrtausendwende erschienen sind.

Das Objektmodell würde wie folgt aussehen:

```
const albums = [
  {
    title: 'Push the Sky Away',
    artist: 'Nick Cave',
    released: 2013
  },
  {
    title: 'No more shall we part',
    artist: 'Nick Cave',
    released: 2001
  },
```

```
  {
    title: 'Live from Mars',
    artist: 'Ben Harper',
    released: 2003
  },
  {
    title: 'The Will to Live',
    artist: 'Ben Harper',
    released: 1997
  }
];
```

Listing 2.45 Ein simples Objektmodell

Imperativ wäre die Problemstellung beispielsweise wie folgt zu lösen:

```
const releasedBefore2000 = [];
for(let i=0, l=albums.length; i<l; i++) {
  if(albums[i].released < 2000) {
    releasedBefore2000.push(albums[i]);
  }
}
console.log(releasedBefore2000);
```

Listing 2.46 Filtern bestimmter Elemente eines Arrays mit imperativer Programmierung

Und ohne jetzt die Leier von eben bezüglich der funktionalen Schreibweise zu wiederholen, hier die entsprechende Variante unter Verwendung der Methode `filter()`:

```
const releasedBefore2000 = albums.filter(
  (album, index, albums) => album.released < 2000
);
```

Listing 2.47 Filtern bestimmter Elemente eines Arrays mit funktionaler Programmierung

Als Parameter übergeben wir auch hier eine Callback-Funktion. Deren Rückgabewert bestimmt in diesem Fall, ob ein Element in das neue Array übernommen wird oder nicht. Gibt die Funktion ein `true` zurück, wird das entsprechende Element in das neue Array übernommen, andernfalls nicht. Wie auch schon bei `forEach()` und `map()` haben wir innerhalb der Callback-Funktion Zugriff auf das aktuelle Element, dessen Index sowie auf das gesamte Array.

2.4.4 Einen Ergebniswert ermitteln mit der Methode »reduce()«

Die vierte im Bunde der »funktionalen Methoden« für Arrays, die ich Ihnen zeigen möchte (es gibt noch ein paar mehr), ist die Methode `reduce()`. Diese Methode dient dazu, ausgehend von den Elementen eines Arrays einen einzigen repräsentativen Wert zu ermitteln.

Angenommen, Sie möchten herausfinden, wie viele Alben sich in Ihrer Musiksammlung befinden, und zufälligerweise stehen diese Daten bereits als JavaScript-Objekt zur Verfügung. Listing 2.48 zeigt ein Beispiel:

```
const artists = [
  {
    name: 'Nick Cave',
    albums: [
      {
        title: 'Push the Sky Away'
      },
      {
        title: 'No more shall we part'
      }
    ]
  },
  {
    name: 'Ben Harper',
    albums: [
      {
        title: 'Live from Mars'
      },
      {
        title: 'The Will to Live'
      }
    ]
  }
];
```

Listing 2.48 Das Objektmodell, das die Musiksammlung repräsentiert

Dann würde man die Anzahl auf imperativem Weg wieder über eine `for`-Schleife lösen (die Ihnen hoffentlich mittlerweile gehörig auf die Nerven geht):

```
let totalNumberOfAlbums  = 0;
for(let i=0, l=artists.length; i<l; i++) {
  totalNumberOfAlbums += artists[i].albums.length;
}
console.log(totalNumberOfAlbums);
```

Listing 2.49 Imperative Variante zur Ermittlung der gesamten Anzahl an Musikalben

Und hier die – wie erwartet – schlankere funktionale Variante:

```
const totalNumberOfAlbums = artists.reduce(
  (result, artist, index, artists) => {
    return result + artist.albums.length;
  },
  0
);
```

Listing 2.50 Funktionale Variante zur Ermittlung der gesamten Anzahl an Musikalben

Die Methode `reduce()` erwartet als Parameter eine Callback-Funktion, die wie gewohnt für jedes Element im Array aufgerufen wird. Wie bei den anderen besprochenen Methoden haben Sie innerhalb des Callbacks Zugriff auf Element, Index und das gesamte Array. Zusätzlich bekommt die Funktion aber den aktuell akkumulierten Wert der vorigen Iteration. Den Startwert geben Sie optional über den zweiten Parameter von `reduce()` an.

Im Beispiel ist der initiale Wert der Akkumulation 0. Für jeden Interpreten wird die übergebene Callback-Funktion aufgerufen und die Anzahl der Alben des jeweiligen Interpreten zur Gesamtanzahl aller Alben addiert.

2.4.5 Kombination der verschiedenen Methoden

Sie sehen schon: Bereits diese Auswahl an standardmäßig zur Verfügung stehenden Methoden für Arrays machen den Quelltext lesbarer und die Lösung der jeweiligen Problemstellung viel eleganter.

Der wahre Vorteil zeigt sich jedoch erst, wenn man die Methoden miteinander kombiniert und die Aufrufe verbindet, beispielsweise um erst die Interpreten aus dem Interpreten-Array herauszufiltern, die mindestens ein Album haben, das nach 2000 erschienen ist, von diesen Interpreten dann die Namen herauszumappen und anschließend darüber zu iterieren. Der Code, der das erreicht, ist in der funktionalen Variante extrem gut lesbar, vor allem, wenn wir die einzelnen Callback-Funktionen statt anonym separat definieren (siehe Listing 2.51):


```
function releasedAfter2000(album) {
  return album.released > 2000;
}
function hasAlbumReleasesdAfter2000(artist) {
  return artist.albums.filter(releasedAfter2000).length > 0;
};
function toArtistName(artist) {
  return artist.name;
};
artists
  .filter(hasAlbumReleasesdAfter2000)
  .map(toArtistName)
  .forEach(console.log);
```

Listing 2.51 Insbesondere die Kombination der vorgestellten Methoden sorgt für sehr lesbaren Code.

Besonders praktisch: `console.log` kann ebenfalls als Parameter für `forEach()` verwendet werden. Ausgegeben werden dann alle Parameter, die einem `forEach()`-Callback übergeben werden: Wert, Index und komplettes Array. Die Ausgabe des obigen Programms lautet daher:

```
Nick Cave 0 [ 'Nick Cave', 'Ben Harper' ]
Ben Harper 1 [ 'Nick Cave', 'Ben Harper' ]
```

Listing 2.52 Ausgabe des obigen Programms

Das imperative Äquivalent würde um einiges umfangreicher sein und diverse `if`-Abfragen und geschachtelte `for`-Schleifen enthalten. Den Code dazu erspare ich Ihnen lieber an dieser Stelle. Ich hoffe, Sie sind auf den Geschmack gekommen, sich (soweit nicht längst geschehen) mehr mit funktionalen Techniken auseinanderzusetzen. Mit den bis hier besprochenen Methoden für Arrays haben Sie nun einen ersten Eindruck dessen, was mit funktionaler Programmierung möglich ist. Im Folgenden möchte ich Ihnen einige weitere teils komplexe funktionale Techniken vorstellen, über die sich weitere Vereinfachungen erreichen lassen.

2.5 Funktionale Techniken und Entwurfsmuster

Dieser Abschnitt behandelt verschiedene funktionale Techniken und Entwurfsmuster für die JavaScript-Entwicklung.

2.5.1 Komposition

In der Mathematik bezeichnet *Komposition* das Hintereinanderschalten bzw. Verketten von Funktionen. Die Komposition zweier Funktionen *f* und *g* schreibt man dabei in der Regel mit dem Verkettungszeichen: $f \circ g$ beschreibt die Funktion, die für ein gegebenes x Folgendes liefert: $f(g(x))$.

Demnach gilt:

```
(f ○ g)(x) = f(g(x))
```

Listing 2.53 Mathematische Definition einer Funktionskomposition

Die Komposition der Funktionen `g` und `f` liefert also für ein `x` das gleiche Ergebnis, als wenn man zuerst die Funktion `g` mit `x` als Parameter aufruft und anschließend mit deren Ergebnis die Funktion `f`.

In der funktionalen Programmierung ist Komposition ein mächtiges Werkzeug, über das es möglich ist, aus bestehenden Funktionen neue Funktionen zu generieren. Die naive Implementierung der obigen Definition in JavaScript sähe wie folgt aus:

```
const compositionSimple = function(f, g) {
  return function(x) {
    return f(g(x));
  };
};
```

Listing 2.54 Naive Implementierung einer Funktion für die Komposition von Funktionen

Allerdings berücksichtigt diese Implementierung nicht den Ausführungskontext der Funktion. Daher ist es besser, die Implementierung wie folgt zu ändern:

```
const compositionWithContext = function(f, g) {
  return function() {
    return f.call(this, g.apply(this, arguments));
  };
};
```

Listing 2.55 Kompositionsfunktion, die den Ausführungskontext berücksichtigt

Mit Hilfe dieser generischen Funktion lassen sich nun Funktionen miteinander kombinieren:

```
function addFour(x) {
  return x + 4;
}
```



```
function multiplyWithSeven(x) {
  return x * 7;
}
const addFourThenMultiplyWithSeven = compositionWithContext(
  multiplyWithSeven,
  addFour
);
const multiplyWithSevenThenAddFour = compositionWithContext(
  addFour,
  multiplyWithSeven
);
console.log(addFourThenMultiplyWithSeven(2));    // 42
console.log(multiplyWithSeven(addFour(2)));      // 42
console.log(multiplyWithSevenThenAddFour(2));    // 18
console.log(addFour(multiplyWithSeven(2)));      // 18
```

Listing 2.56 Beispiel für die Komposition zweier Funktionen

Zu beachten sind zwei Dinge:

- Die übergebenen Funktionen werden von rechts nach links ausgeführt, das heißt, zuerst die zweite übergebene Funktion, danach mit deren Rückgabewert die erste Funktion.
- Das Ganze funktioniert nur, wenn der Rückgabewert der zweiten Funktion als Parameter von der ersten Funktion verarbeitet werden kann. Im Beispiel erwarten sowohl `addFour()` als auch `multiplyWithSeven()` eine Zahl und liefern eine Zahl als Rückgabewert. Insofern funktioniert die Komposition beider Funktionen in beide Richtungen.

Die obige Implementierung der Komposition hat eine Einschränkung: Sie funktioniert nur für zwei Funktionen. Mit ein bisschen JavaScript-Zauberei ist es aber möglich, die Implementierung unter Verwendung einer *Closure* (siehe Abschnitt 2.5.3) so weit anzupassen, dass sie für eine beliebige Anzahl an übergebenen Funktionen ein entsprechendes Ergebnis liefert (siehe Listing 2.57). Hierbei wird von rechts nach links über alle übergebenen Funktionen iteriert und diese jeweils mit dem Ergebnis der vorigen Funktion aufgerufen.

```
const compositionGeneric = function() {
  const functions = arguments;
  return function() {
    let args = arguments;
    for (let i = functions.length; i-- > 0;) {
      args = [functions[i].apply(this, args)];
```

```
    }
    return args[0];
  };
};
```

Listing 2.57 Generische Kompositionsfunktion für beliebig viele Funktionen

Mit dieser generischen Kompositionsfunktion lassen sich beliebig viele Funktionen verknüpfen:

```
const addEightThenMultiplyWithSeven = compositionGeneric(
  multiplyWithSeven,
  addFour,
  addFour
);
console.log(addEightThenMultiplyWithSeven(2));      // 70
```

Listing 2.58 Beispiel für die Komposition mehrerer Funktionen

Auch hier gelten die gleichen Hinweise wie eben, das heißt, die Ausführung der Funktionen geschieht von rechts nach links, und der Rückgabewert jeder Funktion muss im Wertebereich der jeweils nächsten Funktion liegen, um von dieser verarbeitet werden zu können.

2.5.2 Rekursion

Auch wenn Ihnen *Rekursion* sicherlich bereits ein Begriff sein wird, soll diese Technik der Vollständigkeit halber nicht unerwähnt bleiben, handelt es sich doch um eine funktionale Technik: Eine rekursive Funktion ist eine Funktion, die sich, um das Ergebnis zu berechnen, selbst aufruft. Ein klassisches Beispiel dafür ist die Berechnung der Fibonacci-Zahlen:

```
const fibonacciRecursive = function(n) {
  return n < 2 ? n : fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
};
console.log(fibonacciRecursive(11)); // 89
```

Listing 2.59 Rekursive Funktion für die Berechnung von Fibonacci-Zahlen

Zum Vergleich – die Implementierung auf imperativem Weg ist um einiges aufwendiger:

```
function fibonacciImperative(n){
  const fibonacciNumbers = new Array();
  fibonacciNumbers.push(0);
  fibonacciNumbers.push(1);
  for(let i=0; i<n; i++){
    fibonacciNumbers.push(fibonacciNumbers[0] + fibonacciNumbers[1]);
    fibonacciNumbers.shift();
  }
  return fibonacciNumbers[0];
}
console.log(fibonacciImperative(11)); // 89
```


Listing 2.60 Imperative Funktion für die Berechnung von Fibonacci-Zahlen

Obwohl die rekursive Variante recht schlank ist, sind iterative Programme in der Regel effizienter als ihre rekursiven Äquivalente: Für jeden (rekursiven) Aufruf einer Funktion wird immerhin der Kontext der Funktion auf dem Funktionsaufruf-Stack gespeichert, was bei einer großen Anzahl an Funktionsaufrufen Auswirkungen auf die Performance haben kann.

2.5.3 Closures

Closures kommen zustande bei einer besonderen Art von Funktionen höherer Ordnung: wenn eine äußere Funktion aufgerufen wird und eine innere Funktion zurückliefert, die wiederum auf die Variablen bzw. Parameter der äußeren Funktion zugreift.

Was sie dabei so besonders macht, ist die Tatsache, dass die Variablen (der ursprünglich äußeren Funktion) auch noch zur Verfügung stehen, wenn die äußere Funktion beendet wurde. Berücksichtigt wird hierbei jeweils die aktuelle Belegung der Variablen. Die zurückgegebene Funktion schließt die Variablen sozusagen ein – daher der Name Closure. Das bedeutet, dass sich eine äußere Funktion durchaus mehrmals aufrufen lässt und verschiedene Closures ausgehend von der aktuellen Umgebung mit unterschiedlichen Variablenbelegungen zurückliefert.

Sehen Sie sich dazu Listing 2.61 an:

```
function counter(name) {
  let i=0;
  return function() {
    i++;
    console.log(name + ': ' + i);
  }
}
const counter1 = counter('Zähler 1');
counter1(); // Zähler 1: 1
counter1(); // Zähler 1: 2
```

```
const counter2 = counter('Zähler 2');
counter2(); // Zähler 2: 1
counter2(); // Zähler 2: 2
```

Listing 2.61 Bei einer Closure werden die Variablen und Parameter der äußeren Funktion in der inneren Funktion »eingeschlossen«.

Hier kommen zwei dieser Konstrukte zum Einsatz: `counter1` und `counter2`. Jede der Funktionen hat nur Zugriff auf die Umgebung, in der sie erstellt wurde. Ein Aufruf von `counter1()` bzw. `counter2()` verändert nicht die Zählervariable `i` des jeweils anderen Zählers.

Ein weiterer Vorteil: Von außen lässt sich die Variable `i` nicht ändern. Eine wichtige Grundlage für die Datenkapselung, weswegen Closures einen hohen Wert in der fortgeschrittenen JavaScript-Entwicklung haben und in vielen Entwurfsmustern Anwendung finden. Details dazu gibt es in Kapitel 3, »Objektorientierte Programmierung mit JavaScript«.

Anwendungsbeispiel: Memoization

Ein bekanntes Beispiel für die Anwendung einer Closure ist die Implementierung eines Caching-Mechanismus, auch *Memoization*-Entwurfsmuster genannt. Der Quelltext aus Listing 2.62 ist ähnlich auch in Douglas Crockfords lesenswertem Klassiker »JavaScript – The Good Parts« zu finden. Das Beispiel dort ist zwar noch etwas eleganter, weil es zusätzlich die IIFE-Technik anwendet (die ich erst gegen Ende des Kapitels besprechen werde), der wesentliche Bestandteil des Musters, die Closure, ist aber der gleiche:

```
const fibonacciWithCache = function() {
  const cache = [0, 1];
  const fibonacci = function(n) {
    let result = cache[n];
    if (typeof result !== 'number') {
      console.log('Neuberechnung für: ' + n)
      result = fibonacci(n - 1) + fibonacci(n - 2);
      cache[n] = result;
    }
    return result;
  };
  return fibonacci;
};
const fibonacci = fibonacciWithCache();
console.log(fibonacci(11));
console.log(fibonacci(11));
```

Listing 2.62 Funktion zur Berechnung der Fibonacci-Zahlen mit eingebautem Cache

Die Ausgabe lautet wie in Listing 2.63. Sie sehen, dass die Berechnung der Fibonacci-Zahl für 11 beim zweiten Aufruf aus dem Cache geladen wird, es findet keine Neuberechnung statt.

```
Neuberechnung für: 11
Neuberechnung für: 10
Neuberechnung für: 9
Neuberechnung für: 8
Neuberechnung für: 7
Neuberechnung für: 6
Neuberechnung für: 5
Neuberechnung für: 4
Neuberechnung für: 3
Neuberechnung für: 2
89
89
```

Listing 2.63 Nur die Fibonacci-Zahlen, die sich noch nicht im Cache befinden, werden neu berechnet.

2.5.4 Partielle Auswertung

Hin und wieder möchte man eine Funktion mehrmals mit den gleichen oder zu Teilen gleichen Parameterwerten aufrufen. Bei der imperativen Programmierung ist es in solchen Fällen üblich, diesen immer wieder verwendeten Wert in einer Variablen zu speichern und diese Variable dann der entsprechenden Funktion als Parameter zu übergeben:

```
function volume(x, y, z) {
  return x * y * z;
}

const volumeX = 5;
console.log(volume(volumeX, 2, 2));
console.log(volume(volumeX, 3, 3));
console.log(volume(volumeX, 4, 4));
console.log(volume(volumeX, 5, 5));
```

Listing 2.64 Aufruf einer Funktion mit gleichem Parameter bei der imperativen Programmierung

Bei der funktionalen Programmierung geht das dank der sogenannten *partiellen Auswertung* (oder *Partial Application*) einfacher. Die Idee ist dabei, eine Funktion zunächst mit den gleichbleibenden Parametern auszuwerten (die Parameter werden

dabei *gebunden*) und eine neue Funktion zu erstellen, die nur noch die variablen Parameter (das heißt die *ungebundenen* Parameter) erwartet.

Um dieses Prinzip besser zu verstehen, ist es am besten, wenn ich Ihnen zunächst zeige, welche gedanklichen Zwischenschritte der partiellen Auswertung vorausgehen. Der erste Schritt ist dabei leicht nachvollziehbar, denn so würde man auch bei der imperativen Programmierung vorgehen: Sie definieren eine neue speziellere Funktion, die die alte Funktion aufruft. Die neue Funktion hat dabei weniger Parameter und setzt für die wegfallenden Parameter einfach die festen Werte ein, in unserem Beispiel also den Wert `5` für `x`:

```
function volume(x, y, z) {
  return x * y * z;
}

function volumeX5(y, z) {
  return volume(5, y, z);
}

console.log(volumeX5(2, 2));
console.log(volumeX5(3, 3));
console.log(volumeX5(4, 4));
console.log(volumeX5(5, 5));
```

Listing 2.65 Wiederverwendung von Funktionen

Der Nachteil hierbei ist natürlich, dass neue Funktionen immer einzeln und händisch definiert werden müssen. Für jeden vorbelegten Wert von `x` müssten Sie eine neue Funktion deklarieren:

```
function volumeX6(y, z) {
  return volume(6, y, z);
}
```

Listing 2.66 Für jeden neuen Wert von x muss eine neue Funktion definiert werden.

Das geht besser, wie Sie im Folgenden sehen werden.

Generische Funktion mit Closures

Hier kommen die funktionalen Aspekte von JavaScript zu Hilfe, dank denen es möglich ist, eine Funktion zu erstellen, die dynamisch eine andere Funktion zurückgibt. Warum also nicht eine Funktion erstellen, die für ein beliebiges `x` eine entsprechende Funktion zurückgibt, in der `x` belegt ist und `volume()` mit diesem `x` aufgerufen wird? Eine Technik, dies zur erreichen, haben Sie eben bereits kennengelernt: Die Rede ist von Closures.

Listing 2.67 zeigt, wie eine generische Funktion aussehen könnte, die konkrete Funktionen zurückgibt, in denen `x` belegt ist:

```
function volume(x, y, z) {
  return x * y * z;
}

function volumeX(x) {
  return function(y, z) {
    return volume(x, y, z);
  }
}

const volumeX5 = volumeX(5);
console.log(volumeX5(2, 2));
console.log(volumeX5(3, 3));
console.log(volumeX5(4, 4));
console.log(volumeX5(5, 5));
```

Listing 2.67 Aufruf einer Funktion mit gleichem Parameter bei der funktionalen Programmierung in der generischen partiellen Auswertung

Dies ist schon besser und deckt alle Fälle ab, in denen der Wert von `x` feststeht. Was aber, wenn nicht nur `x`, sondern auch `y` vorbelegt werden sollen? In diesem Fall funktioniert das oben gezeigte Vorgehen nicht mehr. Der nächste Schritt wäre also, die Funktion so generisch zu machen, dass sie mit beliebigen vorgegebenen Parametern zurechtkommt.

Generische Funktionsfabrik zur Erzeugung partieller Funktionen

Die Implementierung einer generischen Funktion, die eine beliebige Anzahl an Parametern binden kann, ist eigentlich relativ einfach: Was man ja möchte, ist, dass eine Reihe von Parametern in der Closure gebunden wird und der Rest der Parameter ungebunden bleibt.

Man könnte auch von zwei Arrays sprechen: ein Array von Parametern, die beim Aufruf der äußeren Funktion gebunden werden, und ein Array von Parametern, die erst beim Aufruf der inneren Funktion gebunden werden.

Beide Arrays ergeben sich aus den `arguments`-Objekten der äußeren und der inneren Funktion (siehe Listing 2.68). Um diese Objekte jeweils in ein Array umzuwandeln, kommt die bereits aus diesem Kapitel bekannte Technik des Methodenborgens zum Einsatz. Dieses Mal borgen wir uns jedoch die Array-Methode `slice()`, die ein Array oder arrayähnliches Objekt (wie im Beispiel das Objekt `arguments`) an einem definier-

ten Index teilt und die restlichen Elemente als neues Array zurückgibt. Wird als Index der Wert 0 angegeben, werden alle Element in das neue Array übernommen.

Auf diese Weise erhält man zwei Arrays: `parametersBound`, das beim Aufruf der äußeren Funktion (`volumeFactory(2,4)`) erzeugt wird, und `parametersUnbound`, das erst beim Aufruf der inneren Funktion erzeugt wird (`volumeX2Y4(5)`). Letzterer Aufruf führt auch dazu, dass beide Arrays zum Array `allParameters` zusammengefasst werden und mit diesen Parametern dann die Funktion `volume()` aufgerufen wird.

```
function volume(x, y, z) {
  return x * y * z;
}

function volumeFactory() {
  const parametersBound = Array.prototype.slice.call(arguments, 0);
  console.log(parametersBound); // im Beispiel: [2, 4]
  return function() {
    const parametersUnbound = Array.prototype.slice.call(arguments, 0);
    console.log(parametersUnbound); // im Beispiel: [5]
    const allParameters = parametersBound.concat(parametersUnbound);
    console.log(allParameters); // im Beispiel: [2, 4, 5]
    return volumen.apply(this, allParameters);
  };
}
const volumeX2Y4 = volumeFactory(2, 4);
console.log(volumeX2Y4(5));
```

Listing 2.68 Funktionsfabrik zur Erzeugung partieller »volume()«-Funktionen

Wenn Sie sich jetzt die Funktion `volumeFactory` genau anschauen, fällt Ihnen auf, dass nur an einer Stelle noch ein Bezug zu der Funktion `volume` besteht, nämlich genau dann, wenn diese Funktion über `apply()` aufgerufen wird. Warum aber nicht auch das noch auslagern und die Funktion, die aufgerufen werden soll, allgemein halten?

Listing 2.69 zeigt die generische Variante (`partial`), die für beliebige Funktionen eine beliebige Anzahl an Parametern entgegennimmt und eine Funktion zurückgibt, die die restlichen Parameter als Funktionsparameter hat:

```
function partial(aFunction /*, parameter...*/) {
  const parametersBound = Array.prototype.slice.call(arguments, 1);
  return function() {
    const parametersUnbound = Array.prototype.slice.call(arguments, 0);
    return aFunction.apply(
      this,
      parametersBound.concat(parametersUnbound)
```



```
    );
  };
}
```

Listing 2.69 Generische Funktionsfabrik zur Erzeugung partieller Funktionen

Der erste Parameter dieser Funktion ist diesmal die Funktion, die partiell ausgewertet werden soll. Alle weiteren Parameter werden weiterhin nicht explizit angegeben, beim Umwandeln des `arguments`-Objekts in das `parametersBound`-Array müssen Sie aber diesmal `slice()` ab Index 1 anwenden, also hinter dem Parameter, der das Funktionsobjekt enthält.

Mit `partial()` lassen sich nun sowohl speziellere Varianten von `volume()` erstellen …

```
const volumeX5 = partial(volume, 5);
const volumeX5Y5 = partial(volume, 5, 5);
```

Listing 2.70 Anwendung der Funktion »partial()«

… als auch speziellere Varianten beliebiger anderer Funktionen:

```
function createPerson(firstName, lastName) {
  return {
    firstName: firstName,
    lastName: lastName
  };
}
const createMustermann = partial(createPerson, 'Mustermann');
const max = createMustermann('Max');
// Ausgabe: { firstName: 'Max', lastName: 'Mustermann',  }
console.log(max);
// Ausgabe: { firstName: 'Moritz', lastName: 'Mustermann',  }
const moritz = createMustermann('Moritz');
console.log(moritz);
```

Listing 2.71 Alternatives Beispiel für die Verwendung der Funktion »partial()«

Hinweis zu ECMAScript 5

Sie haben bereits zu Anfang dieses Kapitels die Funktion `bind()` kennengelernt, die seit ES5 zum Umfang von JavaScript gehört. Diese Funktion kann nicht nur dazu verwendet werden, `this` zu binden, sondern auch bei der partiellen Auswertung helfen, wie der Quelltext in Listing 2.72 zeigt:

```
function createPerson(firstName, lastName) {
  return {
```

```
    firstName: firstName,
    lastName: lastName
  }
}

const createMustermann = createPerson.bind(null, 'Mustermann');
const max = createMustermann('Max');
// Ausgabe: { firstName: 'Max', lastName: 'Mustermann' }
console.log(max);
const moritz = createMustermann('Moritz');
// Ausgabe: { firstName: 'Moritz', lastName: 'Mustermann' }
console.log(moritz);
```

Listing 2.72 Die Methode »bind()«, die es seit ECMAScript 5 gibt, kann ebenfalls dazu verwendet werden, nur bestimmte Parameter zu binden.

Hinweis zu ES2015

In Kapitel 4, »ECMAScript 2015 und neuere Versionen«, werden Sie einige neue Features kennenlernen, mit denen die obige Implementierung von `partial()` noch etwas schlanker wird. Und zwar zum einen Rest-Parameter, mit denen Sie statt des `arguments`-Objekts direkt ein Array zur Verfügung gestellt bekommen, und zum anderen den Spread-Operator, über den Sie die Werte eines Arrays auf einen Funktionsaufruf abbilden können. Die Funktion `partial()` würden Sie dort dann wie folgt implementieren:

```
function partial(aFunction, ...parametersBound) {
  return function (...parametersUnbound) {
    return aFunction(...parametersBound, ...parametersUnbound);
  };
}
```

Listing 2.73 Generische Funktionsfabrik zur Erzeugung partieller Funktionen [ES2015]

Alle gezeigten Implementierungen von `partial()` haben jedoch eine Einschränkung: Es besteht lediglich die Möglichkeit, Parameter von links beginnend zu binden. Das heißt beispielsweise, mit `partial()` kann keine Funktion von `volume()` erzeugt werden, in der nur der Parameter `z` gebunden ist. Hierzu müssten die Parameter von rechts beginnend gebunden werden. Des Weiteren ist es mit den bisher gezeigten Implementierungen nicht möglich, irgendeinen beliebigen Parameter »mittendrin« zu binden, beispielsweise für `volume()` nur den Parameter `y`.

Im Folgenden möchte ich Ihnen daher zwei Varianten von `partial()` vorstellen: die partielle Auswertung von rechts ausgehend sowie die partielle Auswertung mit Platzhaltern.

Partielle Auswertung von rechts ausgehend

Die obige generische Variante der Funktion wird auch `partialLeft()` genannt. Analog dazu gibt es die Funktion `partialRight()`, bei der die Parameter von rechts beginnend ausgewertet werden. Das Einzige, was Sie hierfür an der bisherigen Implementierung ändern müssen, ist die Reihenfolge, in der die beiden Parameter-Arrays miteinander konkateniert werden.

```
function partialRight(aFunction/*, parameters...*/) {
  const parametersBound = Array.prototype.slice.call(arguments, 1);
  return function() {
    const parametersUnbound = Array.prototype.slice.call(arguments);
    return aFunction.apply(
      this,
      parametersUnbound.concat(parametersBound)
    );
  };
}
```

Listing 2.74 Generische Funktionsfabrik zur Erzeugung partieller Funktionen bei Anwendung der Parameter von rechts

Angewendet wird `partialRight()` auf die gleiche Weise:

```
const volumeZ5 = partialRight(volume, 5);
console.log(volumeZ5(2, 2)); // 20
console.log(volumeZ5(3, 3)); // 45
console.log(volumeZ5(4, 4)); // 80
console.log(volumeZ5(5, 5)); // 125
```

Listing 2.75 Anwendung von »partialRight()«

Partielle Auswertung mit Platzhaltern

Wenn eine Funktion hinsichtlich beliebiger Parameter partiell ausgewertet werden soll, funktionieren die bisherigen Lösungen nicht mehr: `partial()` bzw. `partialLeft()` wertet die Parameter von links aus, `partialRight()` von rechts. Um beliebige Parameter zu erlauben, muss man einige Erweiterungen durchführen. Das Prinzip dabei ist, mit einem bestimmten Platzhalterwert zu arbeiten, dann innerhalb der `partial()`-Funktion zu prüfen, ob ein übergebener Parameter diesem Platzhalterwert entspricht, und abhängig davon das Parameter-Array zu bilden. Hier der entsprechende Code:

```
const _ = {}; // Platzhalter
function partialWithPlaceholders(aFunction/*, parameters...*/) {
  const parametersBound = Array.prototype.slice.call(arguments, 1);
```

```
  return function() {
    let i,
        parameters = [],
        parametersUnbound = Array.prototype.slice.call(arguments, 0);
    for(i=0; i<parametersBound.length; i++) {
      if(parametersBound[i] !== _) {
        parameters[i] = parametersBound[i];
      } else {
        parameters[i] = parametersUnbound.shift();
      }
    }
    return aFunction.apply(this, parameters.concat(parametersUnbound));
  };
};
```

Listing 2.76 Partielle Auswertung mit Platzhaltern

Die wesentlichen Änderungen spielen sich in der inneren Funktion ab. Hier wird zunächst ein neues Array erstellt, in dem alle konkreten Parameter gesammelt werden. Dazu wird über das Array gebundener Parameter iteriert. Wenn es sich bei einem Parameter nicht um den Platzhalter handelt, wird der Parameter direkt in das Ziel-Array übernommen. Für den Fall dagegen, dass es sich bei dem Parameter um den Platzhalterwert handelt, wird der Parameter aus dem Array `parametersUnbound` verwendet. Hierbei wird die Methode `shift()` aufgerufen, die das erste Element aus einem Array löscht sowie gleichzeitig zurückgibt. Übrig bleiben auf diese Weise alle hinten stehenden Parameter, die bei der partiellen Auswertung überhaupt nicht übergeben wurden (auch nicht als Platzhalter).

Mit dieser Funktion ist es nun möglich, die partielle Auswertung auf beliebige Parameter anzuwenden, im Code in Listing 2.77 beispielsweise auf den zweiten Parameter `y` der Funktion `volume`:

```
const volumeY5 = partialWithPlaceholders(volume, _, 5);
console.log(volumeY5(2, 2)); // 20
console.log(volumeY5(3, 3)); // 45
console.log(volumeY5(4, 4)); // 80
console.log(volumeY5(5, 5)); // 125
```

Listing 2.77 Anwendung der partiellen Auswertung mit Platzhaltern

Hinweis

Normalerweise würden Sie den Platzhalter _ und die Funktion `partialWithPlaceholders` zusammen kapseln, üblicherweise mit Hilfe des IIFE-Entwurfsmusters, das ich Ihnen am Ende dieses Kapitels vorstellen werde.

2.5.5 Currying

Unter dem Begriff *Currying* versteht man in der funktionalen Programmierung eine Technik, bei der eine Funktion mit mehreren Parametern in mehrere Funktionen mit jeweils einem Parameter umgewandelt wird. Der verkettete Aufruf dieser einparametrigen Funktionen führt dann zu dem gleichen Ergebnis wie der Aufruf der einzelnen mehrparametrigen Funktion.

Nehmen wir als Beispiel die bereits bekannte Funktion `volume()`, eine Funktion mit drei Parametern:

```
function volume(x, y, z) {
  return x * y * z;
}
```

Listing 2.78 Das Ausgangsbeispiel: eine einfache Funktion mit drei Parametern

Die Curry-Variante dieser Funktion sehen Sie in Listing 2.79: eine Funktion, die eine Funktion zurückgibt (und `x` in einer Closure einschließt), die wiederum eine Funktion zurückgibt (und `y` in einer Closure einschließt):

```
function volumeCurry(x) {
  return function(y) {
    return function(z) {
      return x * y * z;
    }
  }
}
console.log(volumeCurry(5)(5)(5));
```

Listing 2.79 Prinzip des Curryings

Der Aufruf von `volumeCurry(5)(5)(5)` führt also zu dem gleichen Ergebnis wie der Aufruf von `volume(5,5,5)`.

Doch JavaScript wäre nicht JavaScript, wenn man nicht eine generische Funktion implementieren könnte, die zu jeder Funktion eine äquivalente Curry-Variante erzeugt. Listing 2.80 zeigt, wie es geht:

```
function curry(firstParameter) {
  let n = 0;
  let aFunction = null;
  const parametersBound = Array.prototype.slice.call(arguments, 1);
  if(typeof firstParameter === 'function') {
    aFunction= firstParameter;
    n = firstParameter.length;
```

```
  } else {
    aFunction = parametersBound.shift();
    n = firstParameter;
  }
  return function() {
    const parametersUnbound = Array.prototype.slice.call(arguments);
    const parameters = parametersBound.concat(parametersUnbound);
    return parameters.length < n
      ? curry.apply(this, [n, aFunction].concat(parameters))
      : aFunction.apply(this, parameters);
  }
}
```

Listing 2.80 Generische »curry()«-Funktion

Mit Hilfe dieser generischen Funktion lassen sich nun beliebige mehrparametrige Funktionen in eine Kombination mehrerer einparametriger Funktionen umwandeln:

```
const volumeCurry = curry(volume);
console.log(volumeCurry(5)(5)(5)); // 125

const volumeX5 = volumeCurry(5);
console.log(volumeX5(2)(2)); // 20
console.log(volumeX5(3)(3)); // 45
console.log(volumeX5(4)(4)); // 80
console.log(volumeX5(5)(5)); // 125
```

Listing 2.81 Verwendung der generischen »curry()«-Funktion

Fazit

Sowohl Currying als auch Partial Application ermöglichen es, Ihren JavaScript-Code schlanker zu machen. Beide Techniken sind dabei von der Zielsetzung her recht ähnlich, unterscheiden sich aber in der Umsetzung. Bei der partiellen Auswertung erhalten Sie das finale Ergebnis der Ursprungsfunktion in zwei Schritten: Im ersten Schritt wird eine Funktion mit gebundenen Parametern erzeugt, im zweiten Schritt die Ursprungsfunktion mit allen Parametern aufgerufen. Beim Currying dagegen hängt die Anzahl der notwendigen Funktionsaufrufe, um das finale Ergebnis zu ermitteln, von der Anzahl der Parameter der Ursprungsfunktion ab. Welche der beiden Techniken Sie verwenden und welche für den jeweiligen Anwendungsfall besser geeignet ist, müssen Sie im Einzelfall entscheiden.

Neben den bisher vorgestellten grundlegenden funktionalen Konzepten gibt es weitere funktionale Techniken für die JavaScript-Programmierung, die man vielleicht eher in die Kategorie Entwurfsmuster einordnen würde:

- das *IIFE-Entwurfsmuster* für den sofortigen Aufruf einer deklarierten Funktion
- das bereits mehrfach angesprochene *Callback-Entwurfsmuster*, bei dem Funktionen als Parameter übergeben und zu einem späteren Zeitpunkt aufgerufen werden
- *Self-defining Functions*, also Funktionen, die sich nach Aufruf selbst neu definieren

Diese Entwurfsmuster möchte ich Ihnen im Folgenden vorstellen.

2.5.6 Das IIFE-Entwurfsmuster

Unter einer *IIFE* (*Immediately Invoked Function Expression*) versteht man eine (in der Regel anonyme) Funktion, die direkt aufgerufen wird, sobald sie deklariert ist.

```
(function() {
  console.log('Diese Funktion wird deklariert und sofort aufgerufen.')
})();
```

Listing 2.82 Definition und direkter Aufruf einer Funktion bezeichnet man als IIFE.

Wie Sie sehen, wird die Funktion mit Klammern umgeben, bevor sie aufgerufen wird. Dadurch wird die Funktion nicht als Deklaration gewertet (was einen Syntaxfehler hervorrufen würde), sondern als Ausdruck.

Der Code macht so gesehen also nichts anderes als dieser Code:

```
const aFunction = (function() {
    console.log('Diese Funktion wird deklariert und sofort aufgerufen.')
});
aFunction();
```

Listing 2.83 Eine IIFE entspricht in gewisser Weise diesem Code.

Allerdings gibt es einen feinen Unterschied zwischen diesem Code und einer IIFE: Das direkte Ausführen bei einer IIFE sorgt dafür, dass sie auch wirklich nur ein einziges Mal aufgerufen wird. Wenn wir dies jetzt kombinieren mit der Tatsache, dass Funktionen einen Sichtbarkeitsbereich aufspannen, können Sie sich vielleicht schon denken, worauf das hinausläuft: IIFEs ermöglichen das Emulieren von Block-Scope (bzw. eigentlich sogar das Definieren beliebiger Scopes, das heißt auch solcher, die nicht durch einen Codeblock begrenzt sind).

Sehen Sie sich Listing 2.84 an. Unabhängig davon, dass man hier als Entwickler noch mal die Variablennamen überdenken sollte, ist hier schön zu sehen, wie Block-Scope mittels IIFEs emuliert werden kann:

```
(function() {
  const x = 11;
  if(x<20) {
    (function() {
      const x = 4;
      console.log(x); // Ausgabe: 4
    })();
  }
  if(x>2) {
    (function() {
      const x = 7;
      console.log(x); // Ausgabe: 7
    })();
  }
  console.log(x);     // Ausgabe: 11
})();
```

Listing 2.84 Übertriebenes Emulieren von Block-Scope mit IIFEs

Anwendungsbeispiele

IIFEs können außerdem zum Einsatz kommen, wenn der eigene (initial auszuführende) Code in einem Kontext ausgeführt wird, in dem man eventuell durch die eigenen Variablendeklarationen bereits existierende Variablen überschreiben würde, beispielsweise im globalen Scope. Darüber hinaus spielen IIFEs eine entscheidende Rolle bei einem der wichtigsten JavaScript-Entwurfsmuster, dem Emulieren von Modulen (siehe Kapitel 3, »Objektorientierte Programmierung mit JavaScript«).

2.5.7 Das Callback-Entwurfsmuster

Das klassische Anwendungsbeispiel, bei dem einer Funktion eine andere Funktion als Parameter übergeben wird, sind sogenannte *Callback-Funktionen* (auch *Callback-Handler*, einfach nur *Callbacks* oder im Deutschen streng genommen *Rückruffunktion* genannt). Dabei wird die übergebene Funktion im Laufe der Ausführung der aufgerufenen Funktion von dieser aufgerufen (bzw. zurückgerufen, daher der Name Callback).

Liefert die Funktion ein Ergebnis, wird dieses als Parameter der Callback-Funktion übergeben. Diese Vorgehensweise, also Übergabe einer Funktion und anschließender Aufruf, wird als *Callback-Entwurfsmuster* bezeichnet. Insbesondere bei asynchronen Funktionsaufrufen sind Callbacks ein oft genutztes Mittel (um nicht zu sagen ein zu oft genutztes Mittel – aber dazu später mehr), um den aufrufenden Code über Ergebnisse einer aufgerufenen Funktion zu informieren.

Der generelle Aufbau beim Callback-Pattern ist folgender:

```
function aFunction(callback) {
  // hier weiterer Code
  console.log('Vor Callback');
  callback();
  console.log('Nach Callback');
  // hier weiterer Code
  }
```

Listing 2.85 Genereller Aufbau des Callback-Patterns

Der Aufruf sieht dann wie folgt aus:

```
function anotherFunction() {
  console.log('Callback')
}
aFunction(anotherFunction);
// Ausgabe:
// "Vor Callback"
// "Callback"
// "Nach Callback"
```

Listing 2.86 Anwendung des Callback-Patterns

Hier wird die Funktion `anotherFunction` als Callback übergeben. Vermeiden Sie dabei folgenden oft gemachten Flüchtigkeitsfehler, der dazu führt, dass die als Callback-Funktion gedachte Funktion aufgerufen wird, womit fälschlicherweise ihr Ergebnis als vermeintliche Callback-Funktion übergeben wird. Ist das Ergebnis dann keine Funktion, führt das zu dem Fehler »object is not a function«, wie in Listing 2.87 zu sehen, und das Programm bricht ab.

```
aFunction(anotherFunction());
```

Listing 2.87 Kein Callback-Pattern: Hier wird nicht die Funktion übergeben, sondern ihr Rückgabewert.

Prinzipiell ist es daher immer guter Stil, zu prüfen, ob es sich bei dem übergebenen Parameter wirklich um eine Funktion handelt. Dies ist nicht nur sinnvoll, um den oben genannten Fehler zu vermeiden, sondern generell, um einer falschen Verwendung der entsprechenden Funktion vorzubeugen.

Best Practice: Den Typ überprüfen

Ob es sich bei einem Objekt um eine Funktion handelt, ermitteln Sie mit dem typeof-Operator. Der Typ einer Funktion ist, wie bereits in Kapitel 1, »Einführung«, erwähnt, nicht object, sondern function, was an dieser Stelle praktisch ist:

```
function aFunction(callback) {
  if(typeof callback === 'function') {
    callback();
  } else {
    // Fehlerbehandlung
  }
}
```

Listing 2.88 Vor dem Aufruf eines Callback-Handlers sollten Sie überprüfen, ob es sich tatsächlich um eine Funktion handelt.

Anonyme Funktion als Callback-Handler

In dem Eingangsbeispiel oben haben wir eine benannte Funktion als Callback-Handler übergeben. Dies bietet sich an, wenn die Funktion an mehreren Stellen als Callback-Handler verwendet wird. Oft ist dies aber gar nicht nötig. Dann ist es bequemer, den Callback-Handler als anonyme Funktion zu definieren.

```
aFunction(function() {
  console.log('anonyme Funktion');
});
```

Listing 2.89 Anonyme Funktion als Callback-Handler

Callbacks mit Parametern

Callback-Funktionen können selbstverständlich auch selbst Parameter enthalten. In der Praxis wird dies beispielsweise genutzt, um aus einer asynchronen Funktion heraus den aufrufenden Code über Fehler zu informieren oder ihm den »Rückgabewert«, das Ergebnis der asynchronen Funktion, mitzuteilen, wobei sich seit ES2015 in den meisten Fällen bei der Definition von Callbacks Arrow Functions anbieten:

```
function sum(x, y, callback) {
  const result = x + y;
  if(typeof callback === 'function') {
    callback(result);
  }
}
sum(2, 2, function(result) {
  console.log(`Das Ergebnis lautet: ${result}`);
});
```



```
// Oder als Arrow Function:
sum(2, 2, (result) => {
  console.log(`Das Ergebnis lautet: ${result}`);
});
```

Listing 2.90 Callback-Handler können Parameter haben.

Anwendungsbeispiel: Callbacks als Event-Listener

Ein typischer Anwendungsfall, bei dem das Callback-Entwurfsmuster zum Einsatz kommt, ist das Registrieren von Event-Listenern an UI-Komponenten innerhalb einer Webanwendung. Listing 2.91 gibt dazu ein Beispiel. Der abgebildete Webseitencode enthält eine Schaltfläche sowie einen Textbereich, in dem bei Betätigen der Schaltfläche eine Meldung erscheint.

Dazu wird über die Funktion getElementById() zunächst das UI-Element ermittelt, an dem der Event-Listener anschließend über die Methode addEventListener() registriert wird. Letztere erwartet dabei als Parameter den Typ des Events sowie den Event-Listener in Form einer Funktion.

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
    <script>

      function init() {
        document.getElementById('button').addEventListener(
          'click',
          function(event) {
            document.getElementById('message').value = 'Geklickt';
          },
          false);
      }

  </script>
  </head>
  <body onload="init()">
    <button id="button">Zeige Meldung</button>
    <textarea id="message"></textarea>
  </body>
</html>
```

Listing 2.91 Beispiel für den Einsatz von Callback-Funktionen als Event-Listener

Anwendungsbeispiel: Callbacks zur asynchronen Programmierung

Ein weiteres Anwendungsbeispiel für Callbacks ist, wie bereits erwähnt, der Aufruf von asynchronen Funktionen. Diese können weder über `return` einen Rückgabewert liefern noch Fehler werfen.

Warum nicht? Bei der synchronen Programmierung läuft der Code synchron ab, das bedeutet, eine Funktion wird aufgerufen, ausgeführt und liefert optional einen Rückgabewert. Im Fehlerfall besteht zusätzlich die Möglichkeit, dass eine Exception geworfen wird. Mit asynchronen Funktionen funktioniert das aber nicht. Der Rückgabewert ist nach Aufruf der asynchronen Funktion noch nicht sofort bekannt (er wird schließlich asynchron ermittelt), und Fehler, die eine asynchrone Funktion werfen würde, könnten im synchronen Code nicht abgefangen werden. Da es nicht sicher ist, ob und wann asynchrone Operationen einen Ergebniswert liefern, können solche Operationen also nicht wie synchrone Operationen aufgerufen werden.

Lassen Sie uns das an einem Beispiel verdeutlichen. Hierzu nehmen wir eine einfache asynchrone Funktion, die eine zufällige Anzahl an Millisekunden wartet, bevor sie den Ergebniswert bereitstellt:

```
function asyncFunction() {
  let x;
  setTimeout(() => {
    x = 4711; // Das hier passiert erst nach zwei Sekunden
    console.log(x);
  }, Math.random() * 2000);
  return x;
}
```

Listing 2.92 Eine asynchrone Funktion, in der »return« verwendet wird

Wird diese asynchrone Funktion aufgerufen, als wäre sie eine synchrone Funktion, führt dies schnell zu einem ungewollten Programmverhalten. Das Programm läuft bereits weiter, ohne auf das Ergebnis der asynchronen Funktion zu warten, das Ergebnis ist demnach `undefined`.

```
const result = asyncFunction();
console.log(result); // undefined
```

Listing 2.93 Das Ergebnis der asynchronen Funktion ist »undefined«.

Genauso unmöglich ist es, einen Fehler abzufangen, der in einer asynchronen Funktion auftritt. Zu dem Zeitpunkt, an dem der Fehler geworfen wird, ist der aufrufende Code bereits weitergelaufen, und es ist »niemand mehr da«, um den Fehler zu fangen.


```
function asyncFunction() {
  let x;
  setTimeout(() => {
    throw new Error('Testfehler');
    x = 4711;
  }, Math.random() * 2000);
  return x;
}

try {
  const result = asyncFunction();
} catch(error) {
  console.error('Fehler: ' + error); // Das wird nicht aufgerufen
}
```

Listing 2.94 Fehler, die von asynchronen Funktionen geworfen werden, können nicht gefangen werden. [ES2015]

Asynchrone Funktionen können also auf »normalem« Weg dem aufrufenden Code weder einen Rückgabewert zurückgeben noch über Fehler informieren. Genau an dieser Stelle kommen jetzt Callbacks ins Spiel, denn die stellen sozusagen das Bindeglied zwischen asynchroner Funktion und aufrufendem synchronen Code dar: Der aufrufende Code übergibt die Callback-Funktion, die asynchrone Funktion ruft diese Callback-Funktion mit dem Ergebniswert (dem »Rückgabewert«) oder im Fehlerfall mit dem Fehlerobjekt als Parameter auf. Der aufrufende Code kann dann innerhalb der Callback-Funktion auf beides reagieren.

Bezüglich der Anzahl der Callback-Funktionen, über die Ergebnis und Fehler an den aufrufenden Code übergeben werden, gibt es dabei verschiedene Vorgehensweisen.

Best Practice: Zwei Callback-Funktionen mit jeweils einem Parameter

Hierbei werden der asynchronen Funktion zwei Callback-Funktionen übergeben, eine für den Normalfall und eine für den Fehlerfall. Üblicherweise werden diese Funktionen, wie in Listing 2.95 zu sehen, `success` und `error` genannt.

```
function asyncFunction(success, error) {
  setTimeout(function() {
    const result = 4711; // Hier normalerweise mehr Code
    if(result < 0) {
      error(new Error('Ergebnis kleiner 0'));
    } else {
      success(result);
    }
```

```
  }, 2000);
}
asyncFunction(
  // anonyme Implementierung von success()
  (result) => console.log(result),
  // anonyme Implementierung von error()
  (error) => console.error(error)
);
```

Listing 2.95 Normalfall und Fehlerfall über zwei getrennte Callback-Funktionen [ES2015]

Best Practice: Eine Callback-Funktion mit zwei Parametern

Hierbei wird der asynchronen Funktion statt zweier Callback-Funktionen mit jeweils einem Parameter eine einzige Callback-Funktion mit zwei Parametern übergeben. Üblicherweise wird dabei als erster Parameter der Fehler und als zweiter Parameter der Ergebniswert verwendet. Gibt es keinen Fehler, wird der erste Parameter mit `null` belegt. Insbesondere bei Node.js-Modulen hat sich diese Vorgehensweise als Quasistandard etabliert.

```
function asyncFunction(callback) {
  setTimeout(function() {
    const result = 4711; // Hier normalerweise mehr Code
    if(result < 0) {
      callback(new Error('Ergebnis kleiner 0'), result);
    } else {
      callback(null, result);
    }
  }, 2000);
}
asyncFunction(
  (error, result) => {
    if(error) {
      console.error(error);
    } else {
      console.log(result);
    }
  }
);
```

Listing 2.96 Normalfall und Fehlerfall über eine Callback-Funktion mit zwei Parametern [ES2015]

Callbacks und die Pyramid of Doom

Der übermäßige Gebrauch von Callbacks kann zu einem Codegebilde führen, das unter JavaScript-Entwicklern unter dem Begriff *Pyramid of Doom* bekannt ist. Dies tritt auf, wenn asynchrone Funktionsaufrufe übertrieben oft geschachtelt werden, also im Callback einer asynchronen Funktion eine weitere asynchrone Funktion aufgerufen wird, in deren Callback wieder usw.

In Kapitel 4, in dem es um neuere Features von ECMAScript geht, werde ich Ihnen diese Problematik anhand eines Codebeispiels genauer erläutern. Dort werden Sie unter anderem mit sogenannten *Promises* ein neues Sprachfeature kennenlernen, mit dessen Hilfe Sie die Problematik der Pyramid of Doom vermeiden können.

2.5.8 Self-defining Functions

Ein besonders nettes Entwurfsmuster bei der funktionalen Programmierung ist das der sogenannten *Self-defining Functions* bzw. der *Self-overwriting Functions*. Die Idee dabei ist, dass sich eine Funktion bei Aufruf selbst neu definiert.

In Listing 2.97 beispielsweise definiert sich die Funktion `firstPrintOneThenPrintTwo()` beim ersten Aufruf neu und überschreibt sich somit selbst. Die Folge: Beim ersten Aufruf der Funktion wird eine `1` ausgegeben, bei allen folgenden Aufrufen eine `2`.

```
function firstPrintOneThenPrintTwo() {
  console.log(1);
  firstPrintOneThenPrintTwo = function() {
    console.log(2);
  }
}

firstPrintOneThenPrintTwo(); // Ausgabe: 1
firstPrintOneThenPrintTwo(); // Ausgabe: 2
```

Listing 2.97 Eine sich selbst definierende Funktion überschreibt sich selbst.

> **Hinweis**
>
> Bedenken Sie aber: Dadurch, dass eine Funktion sich selbst überschreibt, werden andere Variablen, die auf das gleiche Funktionsobjekt zeigen, nicht überschrieben. Diese behalten die Referenz auf die ursprüngliche Funktion. In Listing 2.98 wird dies deutlich: Die Funktion `firstPrintOneThenPrintTwo()` überschreibt sich beim ersten Aufruf selbst, gibt also erst eine `1` aus, danach immer eine `2`. Der Aufruf von `functionReference()` dagegen liefert immer eine `1`, diese Funktion wird nicht überschrieben!

```
function firstPrintOneThenPrintTwo() {
  console.log(1);
  firstPrintOneThenPrintTwo = function() {
    console.log(2);
  }
}
const functionReference = firstPrintOneThenPrintTwo;
firstPrintOneThenPrintTwo(); // 1
firstPrintOneThenPrintTwo(); // 2
functionReference(); // 1
functionReference(); // 1
```

Listing 2.98 Funktionsreferenzen werden bei selbstüberschreibenden Funktionen nicht überschrieben.

Anwendungsbeispiel: Selbstüberschreibende Funktionen zur Emulation von Lazy Instantiation

In der Praxis kommen Self-defining Functions vor allem zum Einsatz, um *Lazy Instantiation* zu emulieren. Grundsätzlich bedeutet Lazy Instantiation, dass eine Variable nicht sofort initialisiert wird, sondern erst auf Anfrage. In Java beispielsweise ist folgender Code ein oft gesehenes Entwurfsmuster:

```
public int getResult() {
  if(this.result == null) {
    this.result = this.calculate();
  }
  return result;
}
```

Listing 2.99 Lazy Instantiation in Java

Lazy Instantiation bietet sich vor allem dann an, wenn die Berechnung des entsprechenden Ergebniswertes oder die Initialisierung der Variablen relativ aufwendig ist (beispielsweise das Parsen eines Dokuments oder das Lesen einer Datei). Erst wenn der Wert benötigt wird, wird er berechnet. Insbesondere bei der Webentwicklung wird das Entwurfsmuster oft dazu verwendet, Werte einmalig vom Server zu laden und ab dann auf Clientseite zu cachen.

Mit einer Self-defining Function würde das Entwurfsmuster in JavaScript wie in Listing 2.100 umgesetzt. Beim ersten Aufruf von `getResult()` wird dort zunächst `init()` aufgerufen und die `result`-Variable initialisiert. Anschließend wird `getResult` neu definiert: Bei dieser Neudefinition der Funktion reicht es aus, einfach die (bereits initialisierte) Variable `result` zurückzugeben.


```
function calculate() {
  console.log('calculate()');
  return 'Ergebnis';
}
function getResult() {
  const result = calculate(); // Hier die einmalige Berechnung
  getResult = function() {
    return result;            // ab zweitem Aufruf
  }
  return result;              // erster Aufruf
}
console.log(getResult());     // Ausgabe: "calculate()", dann "Ergebnis"
console.log(getResult());     // Ausgabe: "Ergebnis"
console.log(getResult());     // Ausgabe: "Ergebnis"
console.log(getResult());     // Ausgabe: "Ergebnis"
```

Listing 2.100 Eine sich selbst definierende Funktion zur Emulation von Lazy Instantiation

Dem aufmerksamen Leser wird aufgefallen sein, dass im Endeffekt auch hier nichts anderes als eine Closure zum Einsatz kommt, auch wenn nicht explizit eine Funktion zurückgegeben wird. Dennoch wird die Variable `result` in der neu definierten Variante von `getResult()` »eingeschlossen«.

2.6 Funktionale reaktive Programmierung

In diesem Abschnitt zeige ich Ihnen, was es mit der *funktionalen reaktiven Programmierung* auf sich hat.

2.6.1 Einführung

Die Prinzipien der funktionalen Programmierung sind Ihnen noch aus dem vorigen Abschnitt bekannt: Zum einen liegt dort der Fokus auf Funktionen und ihrer Wiederverwendung (beispielsweise durch Komposition, partielle Auswertung oder Currying), zum anderen arbeiten Funktionen im besten Fall ohne Nebenwirkungen auf Datenstrukturen und liefern als Ergebnis neue Datenstrukturen zurück. Zum Einsatz kommen dabei beispielsweise Methoden wie `map()`, `filter()` oder `reduce()`.

Bei der *reaktiven Programmierung* handelt es sich um ein Programmierparadigma, bei dem Zustandsänderungen innerhalb eines Softwaresystems über Datenflüsse bzw. Datenströme an andere Komponenten des Systems propagiert werden. Dabei kommen in der Regel Entwurfsmuster wie das *Observer-Pattern* oder das *Publish-Subscribe-Pattern* zum Einsatz. Einzelne Komponenten registrieren sich als Observer an

den Datenströmen (die übrigens auch als *Observable* bezeichnet werden), um über neue Daten informiert zu werden.

Die *funktionale reaktive Programmierung* kombiniert die funktionale und die reaktive Programmierung (siehe Abbildung 2.4), indem sich auf den genannten Datenströmen funktionale Operationen wie die genannten Methoden `map()`, `filter()` und `reduce()` anwenden lassen.

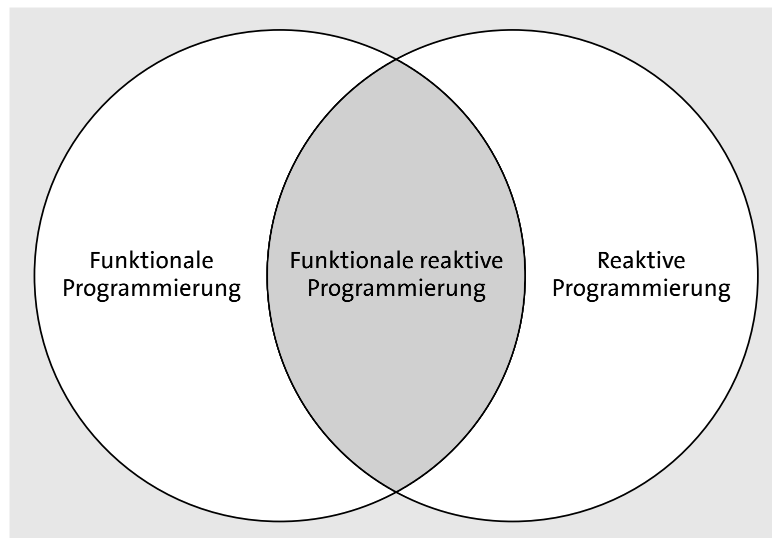

Abbildung 2.4 Zusammenhang zwischen funktionaler, reaktiver und funktionaler reaktiver Programmierung

Reaktive Systeme

Im Zusammenhang mit der reaktiven Programmierung wird häufig auch der Begriff der *reaktiven Systeme* genannt. Diese haben laut dem 2013 verfassten *Reaktiven Manifest* (*http://www.reactivemanifesto.org/*) folgende vier wesentliche Eigenschaften:

- Reaktive Systeme sind **antwortbereit** (englisch *responsive*), sprich, Antworten werden innerhalb einer akzeptablen Zeit gegeben, und eventuelle Fehler werden schnell erkannt.
- Reaktive Systeme sind **widerstandsfähig** (englisch *resilient*), sprich, sie bleiben im Falle eines Fehlers weiterhin antwortbereit.
- Reaktive Systeme sind **elastisch** (englisch *elastic*), sprich, sie skalieren gut und bleiben auch bei unterschiedlichen Lastbedingungen antwortbereit.
- Reaktive Systeme sind **nachrichtenorientiert** (englisch *message-driven*), sprich, sie entkoppeln einzelne Komponenten durch asynchrone Nachrichtenübermittlung.

Auch wenn im »Reaktiven Manifest« nicht explizit die Rede vom reaktiven Programmierparadigma ist, erleichtert dieses Paradigma die Entwicklung reaktiver Systeme.

2.6.2 ReactiveX und RxJS

Eine API für die Programmierung mit Datenströmen existiert in Form von *ReactiveX* (*http://reactivex.io/*). Implementierungen dieser API stehen für verschiedene Programmiersprachen wie Java, C#, Scala, Clojure, C++, Ruby, Python, Groovy, Swift und PHP zur Verfügung (siehe *http://reactivex.io/languages.html*).

Für JavaScript ist die Bibliothek *RxJS* (*https://github.com/ReactiveX/RxJS*) eine der bekannteren Implementierungen von ReactiveX, die sowohl unter Node.js als auch im Browser verwendet werden kann.

Hinweis

Bei Verwendung unter Node.js muss RxJS zunächst über den Node.js Package Manager NPM mit dem Befehl `npm install rxjs` installiert werden. Details zu Node.js und NPM bzw. Hinweise zu ihrer Installation finden Sie in Kapitel 5, »Der Entwicklungsprozess«.

Bevor wir uns ein erstes Beispiel für die funktionale reaktive Programmierung mit RxJS anschauen, sei vorher noch ein Blick auf ein Beispiel der funktionalen Programmierung geworfen: In Listing 2.101 sehen Sie dazu ein klassisches Beispiel für die Anwendung der Methoden `map()`, `filter()` und `reduce()`. Gegeben ist hier ein Array von Zeichenketten, von dem ausgehend über die Methode `map()` zunächst für jede Zeichenkette versucht wird, diese jeweils in eine Zahl umzuwandeln. Anschließend werden über `filter()` die Werte herausgefiltert, die keine Zahl sind (bzw. für die die Funktion `parseInt()` zuvor zu dem Wert `NaN` führte) und schließlich über `reduce()` zu einem einzelnen Wert addiert.

```
const array = [
  '1', 'Max', '2', '3', '4', '5', 'IoT', '6', '7', '8','9'
];
const result = array
  .map(x => parseInt(x))
  .filter(x => !isNaN(x))
  .reduce((x, y) => x + y);
console.log(result);
```

Listing 2.101 Funktionale Programmierung am Beispiel von Arrays [ES2015]

In Listing 2.102 sehen Sie, wie Sie die gleiche Problemstellung mit Hilfe von RxJS und getreu dem funktionalen reaktiven Programmierparadigma implementieren. Nachdem über `require()` zunächst das Node.js-Modul »rx« importiert und das Array mit Zeichenketten erstellt wurde, wird über den Aufruf `Rx.Observable.from()` ausgehend von diesem Array ein Datenstrom erzeugt (Datenströme werden in RxJS bzw. in der API ReactiveX durch den Typen `Observable` repräsentiert).

Wie auch »normale« Arrays stellen Observables die Methoden `map()`, `filter()` und `reduce()` zur Verfügung, so dass der Code relativ ähnlich zu dem aus Listing 2.101 ist. Über den Aufruf von `subscribe()` können dabei Observer (bzw. Callback-Funktionen) für den Datenstrom registriert werden (Abbildung 2.5). Der erste Parameter ist ein Observer, der für den jeweils nächsten Wert im Datenstrom aufgerufen wird, der zweite Parameter ein Observer, der im Fehlerfall aufgerufen wird, und der dritte Parameter ein Observer, der aufgerufen wird, sobald die jeweilige Operation beendet wurde (Im Beispiel wird der erste Observer nur einmal aufgerufen, da sich aufgrund der Anwendung von `reduce()` im resultierenden Datenstrom nur ein Wert befindet).

```
const Rx = require('rxjs');
const array = [
  '1', 'Max', '2', '3', '4', '5', 'IoT', '6', '7', '8','9'
];
const stream = Rx.Observable.from(array);
stream
  .map(x => parseInt(x))
  .filter(x => !isNaN(x))
  .reduce((x, y) => x + y)
  .subscribe(
    x => console.log(x),
    error => console.error(error),
    () => console.log('Fertig')
  );
```

Listing 2.102 Funktionale reaktive Programmierung am Beispiel von Arrays [ES2015]

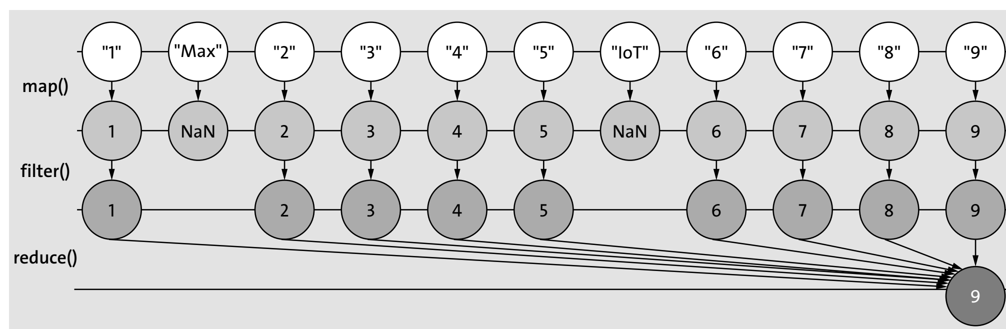

Abbildung 2.5 Die verschiedenen Datenströme für das Zahlenbeispiel

Auch wenn der Code von Listing 2.101 und Listing 2.102 relativ ähnlich aussieht und auch in der funktionalen reaktiven Programmierung die Methoden `map()`, `filter()` und `reduce()` zur Verfügung stehen, gibt es einen wichtigen Unterschied bezüglich der Funktionsweise dieser Methoden: Bei der funktionalen Programmierung ist es

so, dass die Methoden `map()`, `filter()` und `reduce()` synchron hintereinander ablaufen. Das heißt, zunächst wird die Methode `map()` für das jeweilige Array ausgeführt und damit die entsprechende Callback-Funktion für jedes Element in diesem Array aufgerufen. Anschließend wird die Methode `filter()` für das Array ausgeführt (bzw. die entsprechende Callback-Funktion für jedes Element im Array aufgerufen) und danach die Methode `filter()` (bzw. wieder die entsprechende Callback-Funktion für jedes Element).

Bei dem Beispiel in Listing 2.102 (bzw. der funktionalen reaktiven Programmierung) geschehen die drei Schritte des Mappens, des Filterns und des Reduzierens jeweils pro Element. Mit anderen Worten: Zuerst wird das erste Element gemappt, gefiltert und reduziert (bzw. auf einen Gesamtwert akkumuliert), dann das zweite, dann das dritte und so weiter. Dieser Ansatz skaliert wesentlich besser und spiegelt auch mehr die Tatsache wider, dass die Anzahl an Elementen in einem Datenstrom in der Regel nicht von vornherein feststeht.

Observables lassen sich dank entsprechender Helferfunktionen bequem auf Basis verschiedener Quellen erzeugen, sei es wie gezeigt auf Basis von Arrays, auf Basis von Callback-Funktionen, auf Basis von Promises, auf Basis von Ajax-Anfragen oder auf Basis von Nutzerinteraktionen bzw. Events.

Alle diese Datenquellen werden in der funktionalen reaktiven Programmierung als Datenströme gesehen, die zu unbestimmten Zeitpunkten Daten zur Verfügung stellen: Die Callback-Funktion oder das Promise-Objekt liefern ein Ergebnis, sobald die entsprechende asynchrone Funktion abgeschlossen wurde, Eingabefelder lösen durch Nutzereingaben Events aus, und bei Ajax-Anfragen werden die entsprechend registrierten Event-Handler aufgerufen, sobald die Antwort vom Server bereitsteht.

Auf den Datenströmen, die durch die Observable-Objekte repräsentiert werden, können Sie dann verschiedene Operationen bzw. Methoden aufrufen, wie beispielsweise die schon im Eingangsbeispiel gezeigten Methoden `map()`, `filter()` und `reduce()`. Weitere Operationen sind dagegen in Tabelle 2.1 aufgelistet. Beispielsweise lassen sich zwei oder mehrere Datenströme zu einem Datenstrom zusammenfassen (`merge()`), Elemente innerhalb eines Datenstroms suchen (`find()`) und Elemente überspringen (`skip()`).

Methode	Beschreibung
`concat()`	Konkateniert eine Reihe von Datenströmen.
`count()`	Zählt die Elemente in einem Datenstrom, die ein bestimmtes Kriterium erfüllen.
`delay()`	Verzögert einen Datenstrom um eine bestimmte Zeit.

Tabelle 2.1 Übersicht über verschiedene Operatoren

Methode	Beschreibung
every()	Prüft, ob alle Elemente in einem Datenstrom ein bestimmtes Kriterium erfüllen.
filter()	Filtert die Elemente in einem Datenstrom anhand eines bestimmten Kriteriums.
find()	Sucht nach dem ersten Vorkommen eines Elements, das ein bestimmtes Kriterium erfüllt.
first()	Sucht nach dem ersten Vorkommen eines Elements, das ein bestimmtes Kriterium erfüllt.
mergeMap()	Mappt jedes Element eines Datenstroms in einen neuen Datenstrom und fügt die resultierenden Datenströme zusammen.
last()	Sucht nach dem letzten Vorkommen eines Elements, das ein bestimmtes Kriterium erfüllt.
map()	Mappt jedes Element in einem Datenstrom auf ein neues Element.
reduce()	Akkumuliert die Elemente in einem Datenstrom zu einem einzelnen Wert.
skip()	Überspringt eine bestimmte Anzahl an Elementen in einem Datenstrom.
takeUntil()	Liefert die Elemente eines Datenstroms so lange, bis der übergebene Datenstrom ein Element erzeugt.
takeWhile()	Liefert die Elemente eines Datenstroms so lange, wie ein bestimmtes Kriterium erfüllt ist.

Tabelle 2.1 Übersicht über verschiedene Operatoren (Forts.)

2.6.3 Praxisbeispiel: Drag & Drop

Zum besseren Verständnis zeige ich im Folgenden, wie Sie die funktionale reaktive Programmierung dazu einsetzen, eine Drag-and-Drop-Funktionalität auf einer Webseite zu implementieren. Den entsprechenden Code sehen Sie in Listing 2.103. Was im Code passiert, ist der Übersicht halber in Abbildung 2.6 zusätzlich graphisch dargestellt.

Zunächst wird über getElementById() das Element selektiert, das durch die Drag-and-Drop-Operation verschoben werden soll (im Folgenden *Drag-Element* genannt). Anschließend werden über Aufrufe der Methode Rx.Observable.fromEvent() drei Datenströme erzeugt: einer für das »mousedown«-Event auf dem Drag-Element, einer für das »mousemove«-Event auf dem document-Objekt und einer für das

»mouseup«-Event auf dem document-Objekt. Mit anderen Worten: Tritt ein »mousedown«-Event auf dem Drag-Element auf oder ein »mouseup«-Event oder ein »mousemove«-Event auf dem document-Objekt, wird dieses auf den jeweiligen Datenstrom weitergeleitet.

Sobald nun auf dem Datenstrom für die »mousedown«-Events (auf dem Drag-Element) ein Ereignis auftritt, werden alle Ereignisse, die auf dem »mousemove«-Datenstrom ausgelöst werden, jeweils in ein Objekt »gemappt«, das die Pixelabstände der aktuellen Mauszeigerposition zur Ausgangsposition enthält. Dies wird so lange gemacht, bis auf dem »mouseup«-Datenstrom ein Ereignis auftritt, das wiederum über die Methode takeUntil() abgefangen werden kann.

Die einzelnen Datenströme, die hierdurch erzeugt werden (in Listing 2.103 exemplarisch zwei Datenströme), werden anschließend durch den Aufruf mergeMap() zu einem einzelnen Datenstrom zusammengefasst.

```
function init() {
  const dragTarget = document.getElementById('drag-target');
  // Datenstrom für das mouseup-Event
  const mouseUpStream = Rx.Observable.fromEvent(
    document, 'mouseup'
  );
  // Datentrom für das mousemove-Event
  const mouseMoveStream = Rx.Observable.fromEvent(
    document, 'mousemove'
  );
  // Datenstrom für das mousedown-Event
  const mouseDownStream = Rx.Observable.fromEvent(
    dragTarget, 'mousedown'
  );
  mouseDownStream
    .mergeMap(mouseDownEvent =>
      mouseMoveStream
        .map(mouseMoveEvent => {
          mouseMoveEvent.preventDefault();
            return {
              left: mouseMoveEvent.clientX -
                    mouseDownEvent.offsetX,
              top: mouseMoveEvent.clientY -
                   mouseDownEvent.offsetY,
            }
        })
        .takeUntil(mouseUpStream)
      )
```

```
    .subscribe(position => {
      console.log(
        'Relative Position:',
        'links:', position.left,
        'oben:', position.top
      );
    });
}
document.addEventListener('DOMContentLoaded', init);
```

Listing 2.103 Drag & Drop mit funktionaler reaktiver Programmierung [ES2015]

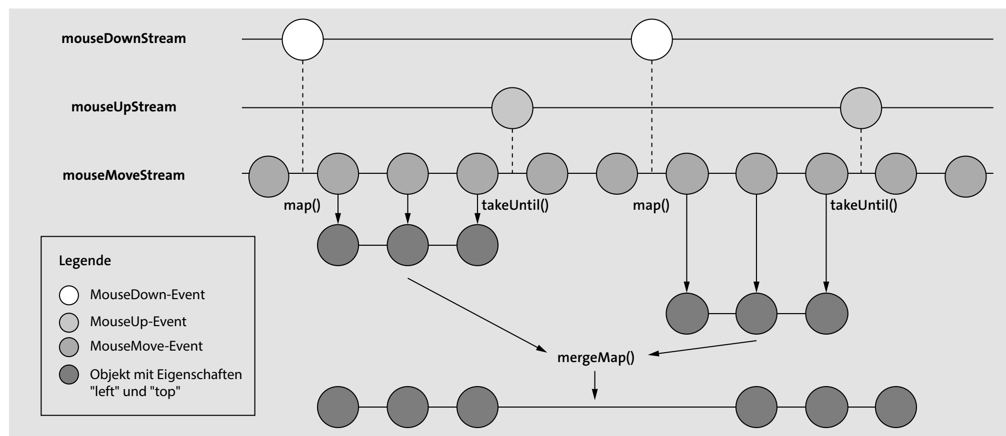

Abbildung 2.6 Die verschiedenen Datenströme für das Drag-and-Drop-Beispiel

2.6.4 Praxisbeispiel: Echtzeitdaten über Web-Sockets

Die Bibliothek RxJS kann durch verschiedene Module erweitert werden. Eines dieser Module ist *RxJS-DOM* (*https://github.com/Reactive-Extensions/RxJS-DOM*), das zusätzliche Methoden zur Verfügung stellt, mit denen Sie Datenströme auf Basis von Server-sent Events (`Rx.DOM.fromEventSource()`), auf Basis von Web Workern (`Rx.DOM.fromWorker()`), von Mutation Observern (`Rx.DOM.fromMutationObserver()`) oder auch von Web-Sockets (`Rx.DOM.fromWebSocket()`)erzeugen.

Ein Beispiel für die Verwendung letzterer Methode ist Listing 2.104 gezeigt. Das Ziel: Daten per Web-Socket-Verbindung von einem Socket-Server abzurufen und diese (mehr oder weniger in Echtzeit) mit Hilfe von *D3.js* (*https://d3js.org/*) und *NVD3.js* (*http://nvd3.org/*) in einem Liniendiagramm darzustellen.

Einen Ausschnitt des Codes für die Serverseite sehen Sie in Listing 2.104: Hier wird ein Socket-Server gestartet, der alle 200 Millisekunden eine zufällig generierte Zahl zwischen 1 und 100 an den jeweiligen Socket-Client sendet.


```
...
const wsServer = new WebSocketServer({
  httpServer: server,
  autoAcceptConnections: false
});

wsServer.on('request', (request) => {
  const connection = request.accept(
    'echo-protocol',
    request.origin
  );
  setInterval(() => {
    connection.sendUTF(Math.floor(Math.random() * 100) + 1);
  }, 200);
  connection.on('close', (reasonCode, description) => {
    console.log((new Date()) + ' Peer ' +
      connection.remoteAddress + ' disconnected.'
    );
  });
});
```

Listing 2.104 Der Code für den Web-Socket-Server [ES2015]

Den Code für die Clientseite sehen Sie in Listing 2.105. Über die Methode `Rx.DOM.fromWebSocket()` wird zunächst ein Datenstrom auf Basis eines Web-Sockets erzeugt. Jedes Mal, wenn der Socket-Server nun eine Zufallszahl generiert und dadurch eine entsprechende Nachricht auf dem Datenstrom erzeugt wird, wird der registrierte Observer aufgerufen und damit ein entsprechender Datenpunkt (bestehend aus Zufallszahl und aktuellem Zeitpunkt) für die Darstellung im Graphen generiert sowie dem Array `values` hinzugefügt.

Für Letzteres wird dabei ebenfalls ein Datenstrom erzeugt (`chartDataStream`), so dass jedes Hinzufügen eines Datenpunkts in das Array an den am Datenstrom registrierten Observer weitergeleitet wird. Der Aufruf `chart.update()` innerhalb des Observers sorgt dann für eine Aktualisierung des Liniendiagramms.

```
function init() {
  const MAX_DATAPOINTS = 150;
  let chart;
  let values = [];
  let chartData = [{
    values: values,
    key: 'Datenstrom',
    color: '#ff7f0e'
```

```
}]
const openObserver = Rx.Observer.create((event) => {
  console.info('Socket geöffnet');
});
const closingObserver = Rx.Observer.create(() => {
  console.log('Socket geschlossen');
});
const stream = Rx.DOM.fromWebSocket(
  'ws://localhost:3000',
  'echo-protocol',
  openObserver,
    closingObserver
);
stream.subscribe(
  (event) => {
    values.push({
      x: new Date(),
      y: parseFloat(event.data)
    });
    if (values.length > MAX_DATAPOINTS)
    {
      values.shift();
    }
    chart.update();
  },
  (error) => {
    console.error('error: %s', error);
  },
  () => {
    console.info('Socket geschlossen');
  }
);

const chartDataStream = Rx.Observable.from(values);
chartDataStream.subscribe(
  (event) => {
    if(typeof chart !== 'undefined') {
      chart.update();
    }
  }
);

nv.addGraph(() => {
```



```
    chart = nv.models.lineChart()
      .interpolate('basis')
      .margin({left: 100})
      .showLegend(true)
      .showYAxis(true)
      .showXAxis(true);
    chart.xAxis
     .axisLabel('Time (ms)'
     .tickFormat(d3.format('.02f'));
    chart.yAxis
      .axisLabel('Voltage (v)')
      .tickFormat(d3.format('.02f'));
    d3.select('#chart svg')
      .datum(chartData)
      .call(chart);
    nv.utils.windowResize(() => {
      chart.update()
    });
    return chart;
  });
}
document.addEventListener('DOMContentLoaded', init);
```

Listing 2.105 Der Code für den Web-Socket-Client [ES2015]

2.7 Zusammenfassung und Ausblick

Im Folgenden eine kurze Zusammenfassung der wichtigsten Punkte aus diesem Kapitel:

- In JavaScript sind Funktionen »First-Class«-Objekte, das heißt, sie können Variablen zugewiesen, als Parameter oder Rückgabewert von Funktionen verwendet und beispielsweise als Werte in einem Array gespeichert werden.
- Der *Ausführungskontext* einer Funktion wird bei Aufruf der Funktion gesetzt und steht innerhalb dieser über die Referenz `this` zur Verfügung. Diese Variable hat je nach Ausführungskontext (global, Objektmethode etc.) also einen anderen Wert.
- In JavaScript gibt es keinen Block-Scope, sondern Function-Scope. Allerdings wird diese Einschränkung aufgehoben mit der in ES2015 eingeführten Möglichkeit, Variablen über das neue Schlüsselwort `let` zu deklarieren.
- Jede Funktion in JavaScript hat drei Methoden: `bind()`, die den Ausführungskontext einer Funktion definiert, sowie `call()` und `apply()`, die Funktionen aufzuru-

fen und dabei die Parameter entweder kommasepariert oder in Form eines Arrays zu übergeben.

- Bei der imperativen Programmierung liegt der Fokus auf dem Wie, bei der funktionalen Programmierung auf dem Was.
- Wichtige Array-Methoden, die das funktionale Konzept (nämlich das Arbeiten auf Daten) widerspiegeln, sind folgende:
  - `forEach()` für die komfortable Iteration über die Elemente eines Arrays
  - `map()`, die die Elemente eines Arrays auf neue Werte abbildet
  - `filter()` für das Herausfiltern von Elementen
  - `reduce()`, die die Elemente auf einen einzelnen Wert reduziert
- Als grundlegende funktionale Techniken in JavaScript haben Sie folgende kennengelernt:
  - *Komposition*: vorhandene Funktionen zu neuen Funktionen kombinieren
  - *Rekursion*: Definition einer Funktion durch sich selbst
  - *Closures*: Einschließen von Variablen und Parametern der äußeren Funktion in der inneren Funktion
  - *Partielle Auswertung*: Teilweise Anwendung von Funktionsparametern, als Rückgabewert erhält man eine Funktion für die restlichen Parameter.
  - *Currying*: Umwandeln einer mehrparametrigen Funktion in mehrere einparametrige Funktionen, deren aneinander gereihter Aufruf zu dem gleichen Ergebnis kommt
- Zudem haben Sie folgende Entwurfsmuster kennengelernt:
  - *Immediately Invoked Function Expression* (IIFE): sofortiger Aufruf einer Funktion nach ihrer Deklaration
  - *Callbacks*: Übergabe einer Funktion als Parameter einer anderen Funktion, die die übergebene Funktion zu einem (un)bestimmten Zeitpunkt aufruft. In der Regel wird dieses Entwurfsmuster dazu verwendet, bei einem asynchronen Funktionsaufruf über Ergebnis bzw. Fehler zu informieren.
  - *Self-defining Functions*: Neudefinition einer Funktion durch sich selbst, in der Regel nach erstem Aufruf
- Bei der *reaktiven Programmierung* handelt es sich um ein Programmierparadigma, bei dem Zustandsänderungen innerhalb eines Softwaresystems über Datenflüsse bzw. Datenströme propagiert werden.
- Die *funktionale reaktive Programmierung* kombiniert die funktionale und die reaktive Programmierung, indem sich auf Datenströmen funktionale Operationen anwenden lassen.

Ganz schön viel Stoff. Nichtsdestotrotz haben wir nur an der Oberfläche dessen gekratzt, was alles in der funktionalen Programmierung möglich ist. Für das weitere Studium und unter der Voraussetzung, dass Sie sich eingehender mit der funktionalen Programmierung in JavaScript beschäftigen möchten, empfehle ich Ihnen folgende Bücher:

- Michael Fogus: *Functional JavaScript: Introducing Functional Programming with Underscore.js*, O'Reilly 2013
- Jens Ohlig, Stefanie Schirmer, Hannes Mehnert: *Das Curry-Buch. Funktional programmieren lernen mit JavaScript*, O'Reilly 2013
- Luis Atencio: *Functional Programming in JavaScript: How to Improve Your JavaScript Programs Using Functional Techniques*, Manning 2016
- Reg Braithwaite: *JavaScript Allongé, the »Six« Edition*, Leanpub 2017

Die in diesem Kapitel vorgestellten Techniken und Entwurfsmuster werden Ihnen bei der JavaScript-Entwicklung an vielen Stellen begegnen. Einige Anwendungsbeispiele habe ich Ihnen dazu bereits gezeigt, ein paar mehr stelle ich Ihnen im folgenden Kapitel vor. Denn wie bereits gesagt, viele Konzepte, die man aus klassisch objektorientierten Sprachen kennt, werden in JavaScript über funktionale Techniken nachgebildet.