FUNKTIONALE PROGRAMMIERUNG IN JAVASCRIPT

Zentrale Rolle

Prinzipien und Techniken der funktionalen Programmierung in JavaScript.

Neben der objektorientierten Programmierung, die ich Ihnen in Ausgabe 7/2016 der **web & mobile developer** vorgestellt habe, ist die sogenannte funktionale Programmierung ein weiteres wichtiges Programmierparadigma, das in JavaScript eine zentrale Rolle spielt.

Doch was bedeutet funktionale Programmierung überhaupt? Welche Prinzipien liegen diesem Programmierparadigma zugrunde? Was ist der Unterschied zur objektorientierten Programmierung? Und was der Unterschied zur imperativen Programmierung?

Die Prinzipien der funktionalen Programmierung

Die funktionale Programmierung folgt im Wesentlichen vier Prinzipien, die im Folgenden kurz erläutert werden:

- **Prinzip 1:** Funktionen sind Objekte erster Klasse.
- **Prinzip 2:** Funktionen arbeiten mit unveränderlichen Datenstrukturen.
- **Prinzip 3:** Funktionen haben keine Nebeneffekte.
- **Prinzip 4:** Funktionale Programme sind deklarativ.

Funktionen sind Objekte erster Klasse (sogenannte First Class Objects, manchmal auch als First Class Citizens bezeichnet). Funktionen können wie andere Objekte und primitive Werte ebenfalls Variablen zugewiesen werden, sie können als Argumente anderer Funktionen verwendet werden oder als deren Rückgabewert.

In nicht funktionalen Programmiersprachen dagegen (wie beispielsweise Java), beziehungsweise genauer gesagt in Sprachen, die keine funktionalen Konzepte unterstützen, werden Funktionen nicht als Objekte repräsentiert und können folglich auch nicht wie solche behandelt werden.

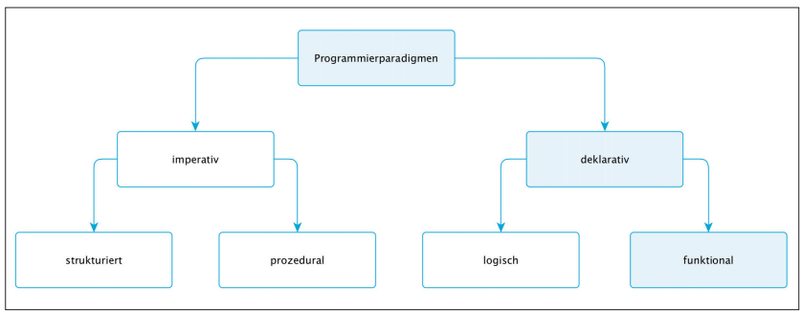
Die Datenstrukturen bei der funktionalen Programmierung sind in der Regel unveränderlich beziehungsweise werden nicht verändert. Vielmehr erzeugen Operationen, die auf Datenstrukturen durchgeführt werden, im Bedarfsfall neue Datenstrukturen und liefern diese als Ergebnis zurück. In rein funktionalen Programmiersprachen können beispielsweise Listen oder andere Datenstrukturen, die einmal angelegt worden sind, nachträglich nicht mehr geändert werden (beziehungsweise nur über Umwege).

Da JavaScript keine rein funktionale Programmiersprache ist, können hier allerdings sehr wohl Arrays (die in der Regel als Datenstruktur bei der funktionalen Programmierung in JavaScript zugrunde liegen) jederzeit verändert werden.

Hinzu kommt, dass bei der funktionalen Programmierung die Funktionen überhaupt keine Nebeneffekte haben und sich eher wie mathematische Funktionen verhalten sollten. Funktionen liefern also bei gleichen Eingaben immer das gleiche Ergebnis, lösen dabei aber keinerlei Nebeneffekte aus. In rein funktionalen Sprachen werden Nebeneffekte bereits durch die Sprache selbst verhindert. JavaScript als nicht rein funktionale Programmiersprache erlaubt es dagegen durchaus, dass Funktionen bei gleichen Eingaben sowohl unterschiedliche Ergebnisse liefern als auch Nebeneffekte haben können.

Funktionale Programme sind deklarativ (Bild 1). Man formuliert sein Programm also eher so, dass man sagt, was gemacht werden soll, und nicht, wie etwas gemacht werden soll. Dadurch sind funktionale Programme gegenüber dem äquivalenten imperativen Code in der Regel besser lesbar, sprechender und kompakter.

Im Unterschied zur objektorientierten Programmierung liegt der Fokus bei der funktionalen Programmierung auf Funktionen, nicht auf Objekten. JavaScript vereint diese beiden Programmierparadigmen. Beispielsweise können Sie damit Ihr Programm objektorientiert strukturieren, das heißt, mit Objekten arbeiten, und innerhalb von Objektmethoden wiederum funktional programmieren – statt beispielsweise imperativ.



Einordnung der funktionalen Programmierung (Bild 1)

JavaScript bietet für Arrays bereits eine Reihe funktionaler Methoden an, mit denen sich, wie im Folgenden gezeigt, gewisse Problemstellungen einfacher lösen lassen als mit der imperativen Programmierung.

Über Arrays iterieren

Wenn Sie beispielsweise über ein Array iterieren wollen, haben Sie gleich mehrere (imperative) Möglichkeiten:

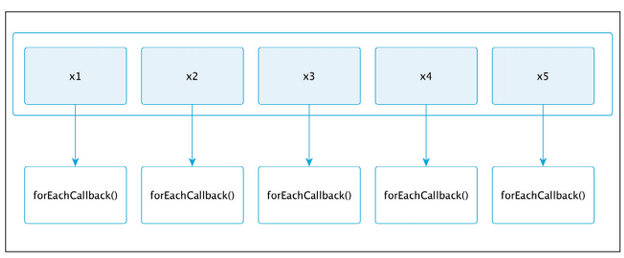
Listing 1: Objektmodell in JSON

```
'use strict';
let persons = [
  {
    "firstName": "Max",
    "lastName": "Mustermann",
    "contacts": [
      {
        "type": "Festnetz",
        "value": "2345/23452345"
      },
      {
        "type": "Mobilnummer",
        "value": "0124/23452345"
      },
      {
        "type": "Mobilnummer",
        "value": "0123/23452345"
      },
      {
        "type": "E-Mail",
        "value": "max.mustermann@example.com"
      }
    ]
  },
  {
    "firstName": "Moritz",
    "lastName": "Mustermann",
    "contacts": [
      {
        "type": "Festnetz",
        "value": "2345/56565656"
      },
      {
        "type": "Mobilnummer",
        "value": "0123/56565656"
      },
      {
        "type": "E-Mail",
        "value": "moritz.mustermann@example.com"
      }
    ]
  }
]
```

- über eine Zählerschleife *(for*-Schleife),
- über eine kopfgesteuerte Schleife *(while*-Schleife),
- über eine fußgesteuerte Schleife *(do-while*-Schleife),
- über eine *for-in*-Schleife,
- über eine *for-of*-Schleife.

Ein Beispiel soll auf Basis des Objektmodells (Listing 1) zeigen, wie die Elemente eines Arrays mit Hilfe einer Zählerschleife ausgegeben werden können. Das Array, über das hier iteriert wird, enthält verschiedene Personen-Objekte, bestehend aus Vorname, Nachname und einer Liste von Kontaktinformationen. Der folgende Code soll lediglich die Vornamen der im Array enthaltenen Personen-Objekte ausgeben:

```
'use strict';
...
for (let i = 0; i < persons.length; i++) {
  console.log(persons[i].firstName);
}
```



forEach(): Veranschaulichung der Methode *forEach()* (Bild 2)

Das ist das, was passieren soll. Dabei beschäftigt sich der Code eigentlich aber viel zu sehr damit, wie das Ganze vonstattengehen soll: Zählervariable initialisieren, Abbruchbedingung überprüfen, indexbasierter Zugriff auf das Array und anschließendes Hochzählen der Zählervariable.

Imperative Programme enthalten also, wie man sieht, viel zusätzlichen Code, der das Programm nicht sehr leserlich macht, sondern eher unnötig aufbläht (Boilerplate-Code). Sicher hat man als Entwickler irgendwann einen Blick für solche imperativen Kontrollstrukturen und weiß in einfachen Fällen auf Anhieb, was eine Zählerschleife im konkreten Fall macht. In der Praxis verhält es sich aber nicht immer so einfach wie im gezeigten Beispiel: Häufig werden Schleifen geschachtelt, innerhalb der Schleifen findet man dann bedingte Anweisungen, Verzweigungen et cetera, sodass der Überblick schnell verloren geht.

Mit Hilfe der funktionalen Programmierung lassen sich Problemstellungen in den meisten Fällen dagegen viel lesbarer formulieren. Für die gegebene Problemstellung eignet sich beispielsweise die Methode *forEach()* viel besser:

```
'use strict';
...
persons.forEach((person, index, array) => {
  console.log(person.firstName);
});
```

Als Argument übergibt man dieser Methode eine Funktion (Callback-Funktion), die dann für jedes Element im Array mit drei Argumenten aufgerufen wird: dem jeweiligen Ele- ▶

ment, dem Index des Elements und schließlich dem Array selbst (Bild 2).

Auch wenn der Code bezogen auf die Anzahl der Zeilen nicht unbedingt kürzer wird, ist er doch schon um einiges besser lesbar. Im Gegensatz zur imperativen Variante liegt der Fokus nämlich jetzt auf der Logik des Programms, nicht auf der Schleife an sich. Übergibt man anstelle einer anonymen Funktion eine benannte Funktion, lässt sich das Programm fast wie natürliche Sprache lesen:

```
'use strict';
function printFirstName(person) {
  console.log(person.firstName);
}
persons.forEach(printFirstName);
```

Oft ist es so, dass man nicht nur über die Elemente eines Arrays iterieren möchte, sondern zeitgleich auch für jedes Element einen Wert ermitteln und diesen in einem anderen Array speichern will. Beispielsweise, um für ein Array natürlicher Zahlen zu jeder Zahl das Quadrat zu berechnen, oder um für ein Array von Objekten von jedem Objekt eine Eigenschaft auszulesen und den entsprechenden Wert in ein neues Array zu kopieren.

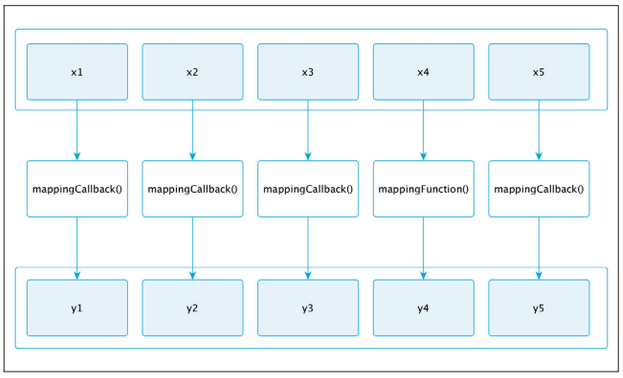
Nehmen wir als Beispiel dazu das bekannte Objektmodell und nehmen an, dass aus dem dort gezeigten Array mit den Personen-Objekten die Vornamen der Personen in ein neues Array kopiert werden sollen. Imperativ würde man diese Problemstellung vermutlich so lösen:

```
'use strict';
let firstNames = [];
for(let i=0; i<persons.length; i++) {
  firstNames.push(persons[i].firstName);
}
console.log(firstNames);
```

Wie schon im Beispiel für die Iteration nimmt auch in diesem Code die *for*-Schleife einen großen Teil des Codes in Anspruch, obwohl sie nur Mittel zum Zweck ist. Mit der Methode *map()* dagegen wird der Code deutlich sprechender:

```
'use strict';
let firstNames = persons.map((person, index, array) => {
  return person.firstName;
});
console.log(firstNames);
function getFirstName(person) {
  return person.firstName;
}
let firstNames = persons.map(getFirstName);
```

Als Argument erwartet diese Methode, ähnlich wie *forEach()*, 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 (Bild 3).



map(): Veranschaulichung der Methode *map()* (Bild 3)

Ein weiterer häufig anzutreffender Anwendungsfall ist das Filtern von Elementen in einem Array. Angenommen, man möchte alle Mobilnummern eines Personen-Objekts herausfiltern. Imperativ würde man diese Aufgabe vermutlich unter Verwendung einer Zählerschleife implementieren:

```
'use strict';
let maxContacts = persons[0].contacts;
let maxMobileNumbers = [];
for(let i=0; i<maxContacts.length; i++) {
  if(maxContacts[i].type === 'Mobilnummer') {
    maxMobileNumbers.push(maxContacts[i]);
  }
}
console.log(maxMobileNumbers);
```

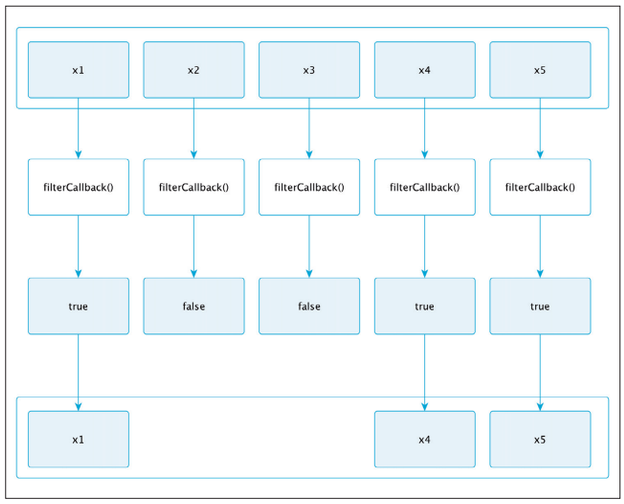
Für jede Kontaktinformation in dem Array, das der Eigenschaft *contact* hinterlegt ist, wird hier innerhalb der Schleife geprüft, ob der Wert für den Typen der Kontaktinformation (Eigenschaft *type)* gleich dem Wert *Mobilnummer* ist. Ist dies der Fall, wird das Element dem Ergebnis-Array *maxMobileNumbers* hinzugefügt.

Werte eines Arrays filtern

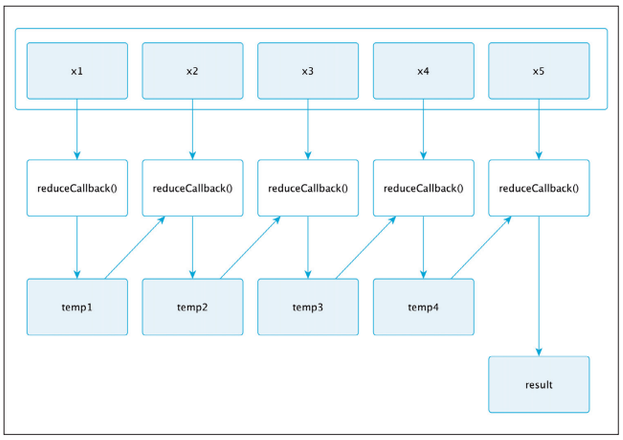
Mit der Methode *filter()* ist dies deutlich einfacher. Als Argument übergibt man auch hier, wie schon bei *forEach()* und *map(),* eine Funktion, die für jedes Element im Array aufgerufen wird:

```
'use strict';
let maxContacts = persons[0].contacts;
let maxMobileNumbers = maxContacts.filter(
  (contact, index, array) => {
    return contact.type === 'Mobilnummer';
  }
);
console.log(maxMobileNumbers);
```

Der Rückgabewert dieser Funktion bestimmt in diesem Fall, ob ein Element in das neue Array übernommen wird: Gibt die Funktion ein *true* zurück, wird es in das neue Array übernommen, andernfalls nicht (Bild 4). Innerhalb der Callback-Funk-



filter(): Veranschaulichung der Methode *filter()* (Bild 4)



reduce(): Veranschaulichung der Methode *reduce()* (Bild 5)

tion hat man dabei wieder Zugriff auf das aktuelle Element, dessen Index sowie auf das gesamte Array.

Eine weitere bekannte Methode im Bunde der funktionalen Methoden für Arrays ist die Methode *reduce()*. Diese Methode dient dazu, ausgehend von den Elementen eines Arrays einen einzigen repräsentativen Wert zu ermitteln, quasi die Elemente eines Arrays zu einem einzelnen Wert zu reduzieren.

Arrays zu einem Wert reduzieren

Zur Veranschaulichung dieser Methode nehmen wir an, dass auf Basis des Objektmodells des Personen-Arrays die Anzahl aller Kontaktinformationen ermittelt werden soll. Auf imperativem Wege würde man diese Problemstellung wahrscheinlich über eine Zählerschleife lösen.

Über die Methode *reduce()* geht das wieder einfacher. Als Argument übergibt man eine Funktion, die wie gewohnt für jedes Element im Array aufgerufen wird.

Wie bei den anderen besprochenen Methoden hat man innerhalb dieser übergebenen Funktion Zugriff auf Element, Index und das gesamte Array. Zusätzlich bekommt die Funktion aber noch den aktuell akkumulierten Wert der vorigen Iteration als Argument übergeben, wobei sich der Startwert optional als zweites Argument der Methode *reduce()* übergeben lässt (Bild 5).

Für jedes Element im Personen-Array wird nun die übergebene Funktion aufgerufen und die Anzahl der Kontaktinformationen der jeweiligen Person auf die Gesamtanzahl aller Kontaktinformationen *(totalContacts)* addiert:

```
'use strict';
let totalContacts = persons.reduce(
  (previousValue, person, index, array) => {
    return previousValue + person.contacts.length;
  },
  0
);
console.log(totalContacts); // 7
```

Man sieht: Bereits die gezeigten standardmäßig zur Verfügung stehenden Methoden für Arrays machen den Quelltext um einiges besser lesbar sowie die Lösung der jeweiligen Problemstellung viel eleganter. Der wahre Vorteil zeigt sich jedoch erst, wenn man die Methoden miteinander kombiniert und die Aufrufe verbindet, etwa wie im folgenden Listing:

```
'use strict';
persons
  .filter(person => person.age > 45)
  .map(person => person.contacts)
  .reduce((allContacts, contacts) =>
   allContacts.concat(contacts), [])
  .filter(contact => contact.type === 'Festnetz')
  .forEach(contact => console.log(contact.value));
```

Hier werden zuerst die Personen herausgefiltert, die älter als 45 sind. Dann werden von diesen Personen die Kontaktinformationen herausgemappt, dann daraus die Festnetznummern herausgefiltert und anschließend darüber iteriert. Der Code, der das erreicht, ist in der funktionalen Variante extrem gut lesbar, vor allem wenn man die einzelnen Callback-Funktionen nicht anonym, sondern mit Namen definiert:

```
'use strict';
persons
  .filter(personOver45)
  .map(getContacts)
  .reduce(mergeContacts)
  .filter(isFixedLine)
  .forEach(print);
```

Es gibt seit ES2015 noch einige weitere Methoden von Arrays, die in diesem Zusammenhang erwähnenswert sind, an dieser Stelle aber nicht im Detail besprochen werden sollen:

- Über die Methode *every()* kann geprüft werden, ob jedes Element in einem Array ein durch die übergebene Callback-Funktion definiertes Kriterium erfüllt.
- Über die Methode *some()* kann geprüft werden, ob mindestens ein Element in einem Array ein durch die überge- ▶

bene Callback-Funktion definiertes Kriterium erfüllt.

- Die Methode *reduceRight()* funktioniert vom Prinzip her wie die Methode *reduce()*, arbeitet aber die Elemente nicht von links nach rechts, sondern von rechts nach links ab. Eine Übersicht aller funktionaler Methoden von Arrays zeigt die Tabelle 1.

Funktionale Techniken und Entwurfsmuster

Neben diesen grundlegenden Methoden, die in JavaScript für Arrays bereitstehen, gibt es natürlich noch eine Reihe weiterer funktionaler Techniken und Entwurfsmuster, etwa Closures, Memoization, partielle Auswertung und Currying.

Eine relativ bekannte Technik der funktionalen Programmierung sind sogenannte Closures. Diese kommen bei einer besonderen Art von Funktion höherer Ordnung zustande, nämlich dann, wenn eine Funktion eine andere Funktion zurückliefert, die wiederum auf die Variablen beziehungsweise 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 bereits beendet wurde. Berücksichtigt wird hierbei also 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. Ein Beispiel dazu zeigt das folgende Listing:

```
'use strict';
function counterFactory(name) {
  let i=0;
  return function() {
    i++;
    console.log(name + ,: , + i);
  }
}
```

Tabelle 1: Funktionale Methoden von Arrays

Methode	Beschreibung
every()	Prüft, ob alle Elemente im Array ein bestimmtes Kriterium erfüllen.
filter()	Filtern von Elementen eines Arrays, die ein bestimmtes Kriterium erfüllen.
forEach()	Iteration über alle Elemente eines Arrays.
map()	Elemente eines Arrays auf einen jeweils anderen Wert abbilden.
reduce()	Zusammenfassen der Elemente eines Arrays zu einem einzelnen Wert.
reduce–Right()	Wie *reduce()*, nur dass die Elemente im Array von rechts nach links durchgegangen werden.
some()	Prüft, ob ein oder mehrere Elemente im Array ein bestimmtes Kriterium erfüllen.

Listing 2: Fibonacci-Zahlen

```
'use strict';
let fibonacciWithCache = function() {
  let cache = [0, 1];
  let 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;
};
let fibonacci = fibonacciWithCache();
console.log(,Ergebnis: , + fibonacci(11));
console.log(,Ergebnis: , + fibonacci(11));
console.log(,Ergebnis: , + fibonacci(15));
```

```
let counter1 = counterFactory('Zähler 1');
counter1(); // Zähler 1: 1
counter1(); // Zähler 1: 2
let counter2 = counterFactory('Zähler 2');
counter1(); // Zähler 2: 1
counter2(); // Zähler 2: 2
```

Hier kommen zwei dieser Konstrukte zum Einsatz: *counter1* und *counter2*. Jede dieser über die Funktion *counterFactory()* erzeugten Funktionen hat nur Zugriff auf die Umgebung, in der sie erstellt wurde. Ein Aufruf von *counter1()* beziehungsweise *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, weshalb Closures einen hohen Stellenwert in der fortgeschrittenen JavaScript-Entwicklung haben und in vielen Entwurfsmustern Anwendung finden, wie beispielsweise dem Memoization-Entwurfsmuster.

Memoization

Ein bekanntes Beispiel für die Anwendung einer Closure ist die Implementierung eines Caching-Mechanismus, auch Memoization-Entwurfsmuster genannt. Der Code in Listing 2 zur Berechnung von Fibonacci-Zahlen ist (leicht abgeändert) 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 (Immediately Invoked Function Expressions) anwendet. Der wesentliche Bestandteil des Musters, die Closure, ist aber der Gleiche.

Was passiert hier? Die Funktion *fibonacciWithCache()* verwaltet ein Array von Zahlen (den Cache), erzeugt eine Funktion *(fibonacci())* und gibt diese zurück. Ruft man nun diese

Listing 3: Imperative Programmierung

```
'use strict';
function createPerson(firstName, lastName, age) {
  return {
    firstName: firstName,
    lastName: lastName,
    age: age
  }
}
let firstName = 'Max';
let maxMustermann = createPerson(
  firstName,
  'Mustermann',
  44
);
let maxMueller = createPerson(
  firstName,
  'Müller',
  47
);
let maxMeier = createPerson(
  firstName,
  'Meier',
  55
);
console.log(maxMustermann);
console.log(maxMueller);
console.log(maxMeier);
```

Listing 4: Wiederverwendung von Funktionen

```
'use strict';
function createPerson(firstName, lastName, age) {
  return {
    firstName: firstName,
    lastName: lastName,
    age: age
  }
}
function createPersonWithFirstNameMax(lastName, age)
{
  return createPerson('Max', lastName, age);
}
let maxMustermann = createPersonWithFirstNameMax(
  'Mustermann',
  44
);
let maxMueller = createPersonWithFirstNameMax(
  'Müller',
  47
);
let maxMeier = createPersonWithFirstNameMax(
  'Meier',
  55
);
console.log(maxMustermann);
console.log(maxMueller);
console.log(maxMeier);
```

Funktion erstmals mit einem bestimmten Zahlenwert auf (zum Beispiel *fibonacci(11)*), so werden die entsprechenden Fibonacci-Zahlen rekursiv bis zu diesem Wert ermittelt und das Ergebnis wird zurückgegeben. So weit noch nichts Besonderes. Zusätzlich wird aber jede ermittelte Fibonacci-Zahl nach ihrer Berechnung auch in den Cache geschrieben, sodass nachfolgende Aufrufe der Funktion *fibonacci()* die entsprechenden Zahlen nur dann neu berechnen, wenn sie sich noch nicht im Cache befinden.

Partielle Auswertung

Hin und wieder kann es der Fall sein, dass man eine Funktion mehrmals mit den gleichen beziehungsweise zu Teilen gleichen Parameterwerten aufrufen möchte. Bei der imperativen Programmierung ist es in solchen Fällen üblich, die wieder verwendeten Werte in entsprechenden Variablen zu speichern und diesen Variablen dann der jeweiligen Funktion als Parameter zu übergeben (Listing 3).

Bei der funktionalen Programmierung geht das dank der sogenannten partiellen Auswertung (beziehungsweise Partial Application) einfacher. Die Idee dabei ist, eine Funktion zunächst mit den gleichbleibenden Parametern auszuwerten (diese Parameter werden dabei gebunden) und eine neue Funktion zu erstellen, die nur noch die verbleibenden Parameter (das heißt, die ungebundenen Parameter) erwartet.

Um dieses Prinzip besser zu verstehen, ist es am besten, zunächst zu verstehen, welche gedankliche Zwischenschritte der partiellen Auswertung vorausgehen. Der erste Schritt ist dabei leicht nachvollziehbar, denn so würde man auch bei der imperativen Programmierung vorgehen: Man definiert eine neue, speziellere Funktion, welche die alte Funktion mit den vorbelegten Parametern aufruft (Listing 4).

Der Nachteil hiervon ist natürlich, dass neue Funktionen immer einzeln und händisch definiert werden müssen. Für jeden vorbelegten Wert von *firstName* müsste eine neue Funktion deklariert werden:

```
'use strict';
...
function createPersonWithFirstNameMoritz(lastName, age)
{
  return createPerson('Moritz', lastName, age);
}
```

Das geht besser, wie im Folgenden zu sehen. Und zwar kommen die funktionalen Aspekte von JavaScript zu Hilfe, denn dank derer ist es möglich, eine Funktion zu erstellen, die dynamisch eine andere Funktion zurückgibt. Warum also nicht eine Funktion erstellen, die für eine beliebige Belegung von *firstName* eine entsprechende Funktion zurückgibt, in der ▶

firstName belegt ist und in der *createPerson()* mit dem entsprechenden Wert aufgerufen wird? Wie das geht, zeigt das Listing 5, wobei die aus dem Vorangehenden bekannten Closures zum Einsatz kommen.

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

Die Implementierung einer solchen Funktion ist eigentlich relativ einfach (Listing 6). Was man möchte, ist ja, dass ein Teil der Parameter in einer 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, sowie ein Array von Parametern, die erst beim Aufruf der inneren Funktion gebunden werden (und solange ungebunden bleiben).

Beide Arrays ergeben sich aus den *arguments*-Objekten der äußeren und der inneren Funktion. Um diese Objekte jeweils in ein Array umzuwandeln, kommt die Technik des Methodenborgens zum Einsatz (mit ES2015 beziehungsweise Rest-Parametern lässt sich das Ganze wie später gezeigt noch weiter vereinfachen).

Auf diese Weise erhält man zwei Arrays: *parameterBound*, das beim Aufruf der äußeren Funktion *(createPersonFactory('Max'))* erzeugt wird, und *parameterUnbound,* das erst beim Aufruf der inneren Funktion erzeugt wird. Letzterer Aufruf führt auch dazu, dass beide Arrays zum Array *allParameters* zusammengefasst werden und mit diesen Parametern dann die Funktion *createPerson()* aufgerufen wird.

Wenn man sich schließlich die Funktion *createPersonFactory()* genau ansieht, fällt auf, dass nur an einer Stelle noch ein Bezug zu der Funktion *createPerson()* 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 und als Parameter übergeben?

Listing 7 zeigt die generische Funktion *partial()*, die für beliebige Funktionen eine beliebige Anzahl an Parametern entgegennimmt und eine Funktion zurückgibt, in der diese Parameter gebunden, die restlichen Parameter jedoch ungebunden sind.

Der erste Parameter dieser Funktion ist diesmal die Funktion, die partiell ausgewertet werden soll. Alle weiteren Pa-

Listing 6: Neuer Wert von firstName

```
'use strict';
...
function createPersonFactory() {
  let parameterBound = Array.prototype
  .slice.call(arguments, 0);
  return function() {
    let parameterUnbound = Array.prototype
    .slice.call(arguments, 0);
    let allParameters = parameterBound
    .concat(parameterUnbound);
    return createPerson
    .apply(this, allParameters);
  };
}
let createPersonWithFirstNameMax =
createPersonFactory('Max');
let createPersonWithFirstNameMoritz =
createPersonFactory('Moritz');
let maxMustermann =
createPersonWithFirstNameMax('Mustermann', 44);
let maxMueller =
createPersonWithFirstNameMax('Müller', 47);
let maxMeier =
createPersonWithFirstNameMax('Meier', 55);
```

Listing 5: Generische partielle Auswertung

```
'use strict';
function createPersonWithFirstName(firstName) {
  return function(lastName, age) {
    return createPerson(firstName, lastName, age);
  }
}
let createPersonWithFirstNameMax =
createPersonWithFirstName('Max');
let createPersonWithFirstNameMoritz =
createPersonWithFirstName('Moritz');
let maxMustermann =
createPersonWithFirstNameMax('Mustermann', 44);
let maxMueller =
createPersonWithFirstNameMax('Müller', 47);
let maxMeier =
createPersonWithFirstNameMax('Meier', 55);
console.log(maxMustermann);
console.log(maxMueller);
console.log(maxMeier);
let moritzMustermann =
createPersonWithFirstNameMoritz('Mustermann', 25);
let moritzMueller =
createPersonWithFirstNameMoritz('Müller', 26);
let moritzMeier =
createPersonWithFirstNameMoritz('Meier', 27);
console.log(moritzMustermann);
console.log(moritzMueller);
console.log(moritzMeier);
```

Listing 7: Generische Funktions-Fabrik

```
'use strict';
...
function partial(fn /*, parameter...*/) {
  let parameterBound = Array.prototype
  .slice.call(arguments, 1);
  return function() {
    let parameterUnbound = Array.prototype
    .slice.call(arguments, 0);
    return fn.apply(this, parameterBound
    .concat(parameterUnbound));
  };
}
let createPersonWithFirstNameMax =
partial(createPerson, 'Max');
let createPersonWithFirstNameMoritz =
partial(createPerson, 'Moritz');
let maxMustermann =
createPersonWithFirstNameMax('Mustermann', 44);
let maxMueller =
createPersonWithFirstNameMax('Müller', 47);
let maxMeier =
createPersonWithFirstNameMax('Meier', 55);
let createMaxMustermann =
partial(createPerson, 'Max', 'Mustermann');
let maxMustermann2 = createMaxMustermann(55);
```

rameter werden weiterhin nicht explizit angegeben, beim Umwandeln des *arguments*-Objekts in das *parameterBound*-Array müssen Sie aber diesmal *slice()* ab Index *1* anwenden, also hinter dem Parameter, der das Funktionsobjekt enthält.

In ES2015 ist die Implementierung dank REST-Parameter und Spread-Operator sogar noch eleganter:

```
'use strict';
function partial(fn, ...parameterBound) {
  return function (...parameterUnbound) {
    return fn(...parameterBound, ...parameterUnbound);
  };
}
```

Alle gezeigten Implementierungen von *partial()* haben jedoch eine Einschränkung: Es besteht lediglich die Möglichkeit, Parameter von links beginnend zu binden. Das heißt beispielsweise, dass mit *partial()* keine Variante auf Basis von *createPerson()* erzeugt werden, in der nur der Parameter *age* 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 *createPerson()* den Parameter *lastName*.

Im Folgenden seien daher zwei Varianten von *partial()* vorgestellt: die partielle Auswertung von rechts ausgehend sowie die partielle Auswertung mit Platzhaltern.

Listing 8: Generische Funktions-Fabrik

```
'use strict';
function partialRight(fn /*, parameter...*/) {
  let parameterBound = Array.prototype
  .slice.call(arguments, 1);
  return function() {
    let parameterUnbound = Array.prototype
    .slice.call(arguments);
    return fn.apply(this, parameterUnbound
    .concat(parameterBound));  };
}
let createPersonWithAge44 =
partialRight(createPerson, 44);
let createPersonWithAge55 =
partialRight(createPerson, 55);

let maxMustermann = createPersonWithAge44(
  'Max',
  'Mustermann'
);
let moritzMustermann = createPersonWithAge55(
  'Moritz',
  'Mustermann'
);
```

Die eben gezeigte generische Funktion wird auch *partialLeft()* genannt. Analog dazu gibt es die Funktion *partialRight()*, bei der die Parameter von rechts beginnend ausgewertet werden. Das Einzige, was hierfür an der bisherigen Implementierung geändert werden muss, ist die Reihenfolge, in der die beiden Parameter-Arrays miteinander konkateniert werden (Listing 8). Die ES2015-Variante sieht so aus:

```
'use strict';
function partialRight(fn, ...parameterBound) {
  return function (...parameterUnbound) {
    return fn(...parameterUnbound, ...parameterBound);
  };
}
```

Wenn eine Funktion hinsichtlich beliebiger Parameter partiell ausgewertet werden soll, funktionieren die bisherigen Lösungen nicht mehr: *partial()* beziehungsweise *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 Platzhalter-Wert zu arbeiten, dann innerhalb der *partial()*-Funktion zu prüfen, ob ein übergebener Parameter diesem Platzhalter-Wert entspricht, und abhängig davon das Parameter-Array zu bilden (Listing 9).

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 ▶

Listing 9: Partielle Auswertung mit Platzhaltern

```
'use strict';
let _ = {}; // Platzhalter
function partialWithPlaceholders(f /*, parameter...*/)
{
  let parameterBound = Array.prototype
  .slice.call(arguments, 1);
  return function() {
    let i,
    parameter = [],
    parameterUnbound = Array.prototype
    .slice.call(arguments, 0);
    for(i=0; i<parameterBound.length; i++) {
      if(parameterBound[i] !== _) {
        parameter[i] = parameterBound[i];
      } else {
        parameter[i] = parameterUnbound.shift();
      }
    }
    return f.apply(this, parameter
    .concat(parameterUnbound));
  };
};

let createPersonWithLastNameMustermann =
partialWithPlaceholders(createPerson, _,
'Mustermann', _);
let maxMustermann =
createPersonWithLastNameMustermann('Max', 44);
let moritzMustermann =
createPersonWithLastNameMustermann('Moritz', 55);
console.log(maxMustermann);
console.log(moritzMustermann);
```

wird über das Array gebundener Parameter iteriert. Wenn es sich bei einem Parameter nicht um den Platzhalter handelt, wird der Parameter direkt in das Zielarray übernommen. Für den Fall dagegen, dass es sich bei dem Parameter um den Platzhalter-Wert handelt, wird der Parameter aus dem Array *parameterUnbound* 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).

Currying

Unter dem Begriff Currying versteht man 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 *createPerson()*, eine Funktion mit drei Parametern: *firstName, lastName* und *age*. Die Curry-Variante dieser Funktion gibt eine Funktion zurück (und schließt *firstName* in einer Closure ein), die wiederum eine Funktion zurückgibt (die *lastName* in einer Closure einschließt), die erneut eine Funktion zurückgibt.

Doch JavaScript wäre nicht JavaScript, wenn man nicht auch hier eine generische Funktion implementieren könnte, die zu beliebigen Funktionen die äquivalente Curry-Variante erzeugt.

Fazit

Die funktionale Programmierung ist ein mächtiges Programmierparadigma, mit dessen Hilfe sich gegenüber der imperativen Programmierung viele Problemstellungen einfacher lösen lassen. JavaScript ist zwar keine rein funktionale Programmiersprache, unterstützt aber verschiedene funktionale Prinzipien und Konzepte. Möchte man die funktionalen Konzepte nicht selbst implementieren, kann man auf eine der zahlreichen zur Verfügung stehenden funktionalen Bibliotheken zurückgreifen, die in Tabelle 2 aufgelistet sind.

Tabelle 2: Bibliotheken und Anlaufstellen

Bezeichnung	URL
Awesome FP JS	https://github.com/stoeffel/awesome-fp-js
101	https://github.com/tjmehta/101
Folktale	http://folktalejs.org
functional.js	http://functionaljs.com
immutable-js	https://github.com/facebook/immutable-js
Lodash	https://lodash.com
Lodash FP	https://github.com/lodash/lodash/wiki/FP-Guide
Ramda	https://github.com/ramda/ramda
Underscore.js	http://underscorejs.org

Im nächsten Beitrag dieser Artikelserie geht es um die reaktive Programmierung, die der funktionalen Programmierung nicht ganz unähnlich ist. ■

Philip Ackermann
arbeitet beim Fraunhofer-Institut für Angewandte Informationstechnologie FIT in den Bereichen Web Compliance, IoT und eHealth. Er ist Autor mehrerer Fachbücher und Fachartikel über Java und JavaScript.
http://philipackermann.de