JavaScript objektorientiert

Verständlicher, flexibler, effizienter programmieren

Bearbeitet von
Nicholas Zakas

1. Auflage 2014. Taschenbuch. XIV, 122 S. Paperback
ISBN 978 3 86490 202 4
Format (B x L): 15,6 x 22,4 cm

4 Objekte

Es gibt in JavaScript zwar eine Reihe eingebauter Referenztypen, doch werden Sie trotzdem relativ häufig eigene Objekte erstellen müssen. Denken Sie dabei jedoch immer daran, dass Objekte in JavaScript dynamisch sind, sich also während der Codeausführung ändern können. In klassenbasierten Sprachen werden Objekte auf die Klassendefinition festgelegt, doch für JavaScript-Objekte gibt es solche Einschränkungen nicht.

Ein Großteil der Arbeit bei der JavaScript-Programmierung dreht sich um die Verwaltung solcher Objekte. Darum ist ein Verständnis von Objekten entscheidend, um JavaScript als Ganzes meistern zu können. Dies wird im Verlauf dieses Kapitels noch ausführlicher besprochen.

4.1 Eigenschaften definieren

In Kapitel 2 haben Sie gelernt, dass es zwei grundlegende Möglichkeiten gibt, um eigene Objekte zu erstellen, nämlich mit dem Konstruktor `Object` und mit einem Objektliteral:

```
var person1 = {
   name: "Nicholas"
};

var person2 = new Object();
person2.name = "Nicholas";

person1.age = "Redacted";       ❶
person2.age = "Redacted";

person1.name = "Greg";          ❷
person2.name = "Michael";
```

Sowohl `person1` als auch `person2` sind Objekte mit einer Eigenschaft namens `name`. Im Verlauf dieses Beispiels wird den beiden Objekten noch

die Eigenschaft age zugewiesen ❶. Das kann unmittelbar nach der Definition des Objekts erfolgen, aber auch viel später. Die Objekte, die Sie erstellen, sind immer offen für Änderungen, sofern Sie nichts Gegenteiliges festlegen (siehe Abschnitt 4.7). Im letzten Teil des Beispiels wird der Wert von name für beide Objekte geändert ❷. Auch Eigenschaftswerte lassen sich jederzeit ändern.

Wird einem Objekt eine neue Eigenschaft hinzugefügt, so wendet JavaScript die interne Methode [[Put]] auf das Objekt an. [[Put]] legt in dem Objekt eine Stelle an, an der die Eigenschaft gespeichert wird. Das entspricht in etwa dem Hinzufügen eines neuen Schlüssels in einer Hashtabelle. Bei dieser Operation wird nicht nur der Anfangswert festgelegt, sondern auch einige Attribute der Eigenschaft. Wenn in dem vorstehenden Beispiel die Eigenschaften name und age zum ersten Mal für die einzelnen Objekte definiert werden, wird für sie daher jeweils die Methode [[Put]] aufgerufen.

Infolge dieses Aufrufs von [[Put]] wird in dem Objekt eine *eigene Eigenschaft* erstellt. Das bedeutet einfach nur, dass die Eigenschaft der vorliegenden Instanz des Objekts gehört. Die Eigenschaft wird direkt in der Instanz gespeichert, und alle Operationen an dieser Eigenschaft müssen über das Objekt erfolgen.

Hinweis

Es ist wichtig, eigene Eigenschaften von den *Prototypeigenschaften* zu unterscheiden, die in Kapitel 5 besprochen werden.

Wird einer vorhandenen Eigenschaft ein neuer Wert zugewiesen, so erfolgt dies mit einer anderen Operation, nämlich [[Set]]. Dadurch wird der vorhandene Wert der Eigenschaft durch den neuen ersetzt. Wenn in dem vorherigen Beispiel name auf den zweiten Wert gesetzt wird, führt dies zu einem Aufruf von [[Set]]. Abbildung 4–1 zeigt Schritt für Schritt, was hinter den Kulissen in person1 geschieht, wenn die Eigenschaften name und age geändert werden.

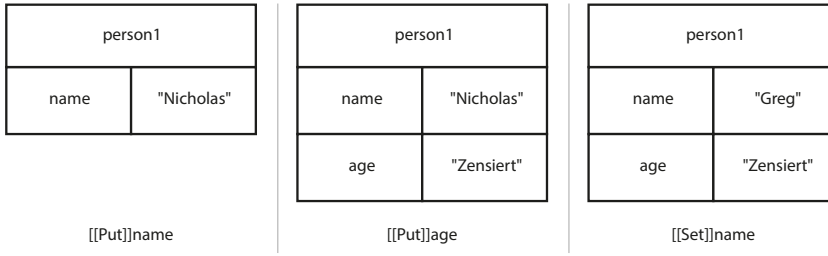

Abb. 4–1 Eigenschaften eines Objekts hinzufügen und ändern

Im ersten Teil der Abbildung wird ein Objektliteral verwendet, um das Objekt `person1` zu erstellen. Dadurch wird implizit `[[Put]]` für die Eigenschaft `name` ausgeführt. Die Zuweisung eines Werts zu `person.age` führt `[[Put]]` für die Eigenschaft `age` aus. Wird `person1.name` dagegen auf einen neuen Wert gesetzt (hier `"Greg"`), erfolgt eine `[[Set]]`-Operation für die Eigenschaft `name`, wodurch der vorhandene Eigenschaftswert überschrieben wird.

4.2 Eigenschaften ermitteln

Da Eigenschaften jederzeit hinzugefügt werden können, ist es manchmal erforderlich zu prüfen, ob eine bestimmte Eigenschaft in dem vorliegenden Objekt vorhanden ist oder nicht. Von Entwicklern, die neu mit JavaScript arbeiten, werden dazu fälschlicherweise häufig Konstruktionen wie die folgende verwendet:

```
// Unzuverlässig
if (person1.age) {
   // Macht irgendetwas mit age
}
```

Das Problem bei dieser Konstruktion ist die Auswirkung der impliziten JavaScript-Typumwandlung auf das Ergebnis. Die `if`-Bedingung wird zu `true` ausgewertet, wenn der Wert »truthy« ist (ein Objekt, ein nicht leerer String, eine von 0 verschiedene Zahl oder `true`), und zu `false`, wenn der Wert »falsy« ist (`null`, `undefined`, `0`, `false`, `NaN` oder ein leerer String). Da eine Objekteigenschaft einen dieser »falsy«-Werte enthalten kann, ist es möglich, dass der Beispielcode fälschlicherweise ein negatives Ergebnis liefert. Ist `person1.age` beispielsweise `0`, so wird die `if`-Bedingung nicht erfüllt, obwohl die Eigenschaft existiert. Eine zuverlässigere Mög-

lichkeit, um das Vorhandensein einer Eigenschaft zu prüfen, bietet der Operator `in`.

Dieser Operator sucht in einem bestimmten Objekt nach einer Eigenschaft mit einem gegebenen Namen und gibt `true` zurück, wenn er sie findet. Im Grunde genommen prüft der Operator, ob der angegebene Schlüssel in der Hashtabelle vorhanden ist. Das folgende Beispiel zeigt die Verwendung von `in` für einige Eigenschaften des Objekts `person1`:

```
console.log("name" in person1);    // true
console.log("age" in person1);     // true
console.log("title" in person1);   // false
```

Da Methoden einfach nur Eigenschaften sind, die auf Funktionen verweisen, können Sie auf dieselbe Weise auch das Vorhandensein einer Methode überprüfen. Der folgende Code fügt `person1` die neue Funktion `sayName()` hinzu und wendet dann `in` an, um das Vorhandensein dieser Funktion zu bestätigen:

```
var person1 = {
   name: "Nicholas",
   sayName: function() {
      console.log(this.name);
   }
};

console.log("sayName" in person1);   // true
```

In den meisten Fällen ist der Operator `in` die beste Möglichkeit, festzustellen, ob eine Eigenschaft in einem Objekt vorhanden ist. Außerdem bietet er den zusätzlichen Vorteil, dass er den Wert der Eigenschaft nicht auswertet, was wichtig sein kann, wenn eine solche Auswertung ein Leistungsproblem oder einen Fehler hervorrufen könnte.

Manchmal wollen Sie jedoch nicht nur wissen, ob überhaupt eine bestimmte Eigenschaft vorhanden ist, sondern ob es sich um eine eigene Eigenschaft handelt. Der Operator `in` sucht jedoch sowohl nach eigenen Eigenschaften als auch nach Prototypeigenschaften, weshalb Sie ihn hierfür nicht verwenden können. Hier kommt die Methode `hasOwnProperty()` ins Spiel, die in allen Objekten vorhanden ist und nur dann `true` zurückgibt, wenn die Eigenschaft existiert und eine eigene Eigenschaft ist. Der

folgende Code vergleicht die Ergebnisse der Verwendung von `in` und von `hasOwnProperty()` für unterschiedliche Eigenschaften von `person1`:

```
var person1 = {
   name: "Nicholas",
   sayName: function() {
      console.log(this.name);
   }
};

console.log("name" in person1);                    // true
console.log(person1.hasOwnProperty("name"));       // true

console.log("toString" in person1);                // true
console.log(person1.hasOwnProperty("toString"));   // false      ❶
```

In diesem Beispiel ist `name` eine eigene Eigenschaft von `person1`, sodass sowohl der Operator `in` als auch `hasOwnProperty()` den Wert `true` zurückgeben. Bei der Methode `toString()` dagegen handelt es sich um eine Prototypeigenschaft, die in allen Objekten vorhanden ist. Der Operator `in` gibt auch dafür `true` zurück, `hasOwnProperty()` dagegen `false` ❶. Dieser wichtige Unterschied wird in Kapitel 5 ausführlicher besprochen.

4.3 Eigenschaften entfernen

Eigenschaften können nicht nur jederzeit zu Objekten hinzugefügt, sondern auch von ihnen entfernt werden. Das geschieht allerdings nicht, wenn Sie die Eigenschaft einfach nur auf `null` setzen. Dadurch wird lediglich `[[Set]]` mit dem Wert `null` aufgerufen, und wie Sie in diesem Kapitel schon gesehen haben, führt das nur dazu, dass der Wert der Eigenschaft ersetzt wird. Um eine Eigenschaft komplett von einem Objekt zu entfernen, müssen Sie den Operator `delete` verwenden.

Dieser Operator bezieht sich immer nur auf eine einzige Objekteigenschaft und ruft die interne Operation `[[Delete]]` auf. Das können Sie sich so vorstellen, als würde ein Schlüssel/Wert-Paar aus einer Hashtabelle entfernt. Nach erfolgreicher Ausführung gibt der Operator `delete` den Wert `true` zurück. (Es gibt auch Eigenschaften, die sich nicht entfernen lassen, was wir uns weiter hinten in diesem Kapitel noch genauer

ansehen werden.) Das folgende Beispiel zeigt den Operator `delete` in Aktion:

```
var person1 = {
   name: "Nicholas"
};

console.log("name" in person1);    // true

delete person1.name;               // true - keine Ausgabe
console.log("name" in person1);    // false
console.log(person1.name);         // undefined          ❶
```

In diesem Beispiel wird die Eigenschaft `name` von `person1` entfernt. Nach Abschluss dieser Operation gibt der Operator `in` den Wert `false` zurück. Beachten Sie, dass der Versuch, auf eine nicht vorhandene Eigenschaft zuzugreifen, `undefined` liefert ❶. Abbildung 4–2 zeigt, wie sich `delete` auf ein Objekt auswirkt.

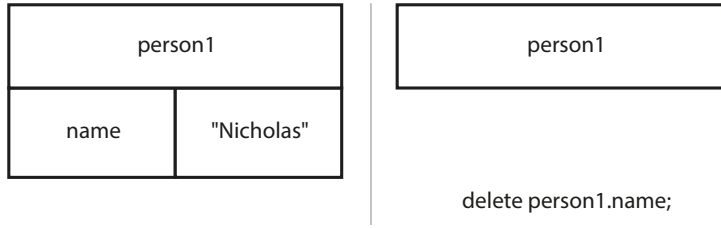

Abb. 4–2 Wenn Sie die Eigenschaft name mit `delete` löschen, wird sie komplett von `person1` entfernt.

4.4 Aufzählung

Standardmäßig sind alle Eigenschaften, die Sie einem Objekt hinzufügen, *aufzählbar*, das heißt, Sie können sie in einer `for-in`-Schleife durchlaufen. Bei aufzählbaren Eigenschaften ist das interne Attribut `[[Enumerable]]` auf `true` gesetzt. Eine `for-in`-Schleife kann alle aufzählbaren Eigenschaften eines Objekts aufführen und die Eigenschaftsnamen einer Variablen zuweisen. So gibt beispielsweise die folgende Schleife die Eigenschaftsnamen und -werte eines Objekts aus:

```
var property;

for (property in object) {
   console.log("Name: " + property);
   console.log("Value: " + object[property]);
}
```

Bei jedem Durchlauf der `for-in`-Schleife wird die Variable `property` mit der nächsten aufzählbaren Eigenschaft des Objekts gefüllt, bis alle diese Eigenschaften genannt wurden. An diesem Punkt wird die Schleife beendet und die Codeausführung fortgesetzt. In diesem Beispiel wird die Klammerschreibweise verwendet, um den Wert der Objekteigenschaft abzurufen und in der Konsole auszugeben. Dies ist einer der wichtigsten Verwendungszwecke für diese Schreibweise in JavaScript.

Wenn Sie zur späteren Verwendung in dem Programm eine Liste der Eigenschaften eines Objekts benötigen, können Sie die in ECMAScript 5 eingeführte Methode `Object.keys()` verwenden, die ein Array der Namen aufzählbarer Eigenschaften abruft:

```
var properties = Object.keys(object);              ❶

// Wenn Sie das for-in-Verhalten nachahmen wollen
var i, len;

for (i=0, len=properties.length; i < len; i++){
   console.log("Name: " + properties[i]);
   console.log("Value: " + object[properties[i]]);
}
```

In diesem Beispiel werden die aufzählbaren Eigenschaften eines Objekts mit `Object.keys()` abgerufen ❶. Anschließend werden die Eigenschaften in einer `for`-Schleife durchlaufen, um den Namen und den Wert auszugeben. Gewöhnlich verwenden Sie `Object.keys()` in Situationen, in denen Sie mit einem Array von Eigenschaftsnamen arbeiten müssen, und `for-in`, wenn Sie kein Array brauchen.

Beachten Sie aber, dass nicht alle Eigenschaften aufzählbar sind. Tatsächlich ist das Attribut `[[Enumerable]]` für die meisten der nativen Methoden von Objekten auf `false` gesetzt. Ob eine Eigenschaft aufzählbar ist, können Sie mit der Methode `propertyIsEnumerable()` herausfinden, die in jedem Objekt vorhanden ist:

Hinweis

Es gibt einen Unterschied zwischen den aufzählbaren Eigenschaften, die in einer `for-in`-Schleife aufgezählt werden, und denen, die `Object.keys()` zurückgibt. Die Schleife zählt auch die Prototypeigenschaften auf, `Object.keys()` dagegen gibt nur die eigenen (Instanz-)Eigenschaften zurück. Der Unterschied zwischen Prototyp- und eigenen Eigenschaften wird in Kapitel 5 besprochen.

```
var person1 = {
   name: "Nicholas"
};

console.log("name" in person1);                          // true
console.log(person1.propertyIsEnumerable("name"));       // true  ❶

var properties = Object.keys(person1);

console.log("length" in properties);                     // true
console.log(properties.propertyIsEnumerable("length")); // false ❷
```

Hier ist die Eigenschaft `name` aufzählbar, da es sich um eine vom Entwickler definierte Eigenschaft von `person1` handelt ❶. Dagegen ist die Eigenschaft `length` des Arrays `properties` nicht aufzählbar ❷, weil dies eine inhärente Eigenschaft von `Array.prototype` ist. Sie werden feststellen, dass viele native Eigenschaften standardmäßig nicht aufzählbar sind.

4.5 Arten von Eigenschaften

Es gibt zwei verschiedene Arten von Eigenschaften, nämlich Dateneigenschaften und Zugriffseigenschaften. *Dateneigenschaften* enthalten Werte. Dazu gehört etwa die Eigenschaft `name` aus den vorherigen Beispielen in diesem Kapitel. Das Standardverhalten der Methode `[[Put]]` besteht darin, eine Dateneigenschaft zu erstellen, und in allen bisherigen Beispielen in diesem Kapitel haben wir Dateneigenschaften gezeigt. *Zugriffseigenschaften* dagegen weisen keinen Wert auf, sondern definieren Funktionen, die beim Lesen (*Get-Funktion*) und beim Schreiben der Eigenschaft (*Set-Funktion*) aufgerufen werden. Zugriffseigenschaften können sowohl eine Get- als auch eine Set-Funktion haben, es ist aber nur eine davon erforderlich.

Zur Definition einer Zugriffseigenschaft mit einem Objektliteral ist eine besondere Syntax erforderlich:

```
var person1 = {
   _name: "Nicholas",                                    ❶

   get name() {                                          ❷
      console.log("Reading name");
      return this._name;
   },

   set name(value) {                                     ❸
      console.log("Setting name to %s", value);
      this._name = value;
   }
};

console.log(person1.name); // "Reading name", dann "Nicholas"

person1.name = "Greg";
console.log(person1.name); // "Setting name to Greg", dann "Greg"
```

In diesem Beispiel wird die Zugriffseigenschaft `name` definiert. Außerdem gibt es die Dateneigenschaft `_name`, die den eigentlichen Wert dieser Eigenschaft enthält ❶. (Der vorangestellte Unterstrich ist die übliche Schreibweise dafür, dass die Eigenschaft als privat angesehen wird; allerdings ist sie in Wirklichkeit immer noch öffentlich.) Die Syntax zur Definition der Get-Funktion ❷ und der Set-Funktion ❸ ähnelt sehr stark einer normalen Funktion, aber ohne das Schlüsselwort `function`. Hier werden vor dem Namen der Zugriffseigenschaft die Schlüsselwörter `get` und `set` verwendet, gefolgt von Klammern und dem Funktionsrumpf. Get-Funktionen geben einen Wert zurück, während Set-Funktionen den Wert, der der Eigenschaft zugewiesen werden soll, als Argument entgegennehmen.

In diesem Beispiel wird zur Speicherung der Eigenschaftsdaten zwar `_name` verwendet, doch könnten Sie die Daten auch genauso gut in einer Variablen oder in einem anderen Objekt ablegen. Außerdem wird das Verhalten der Eigenschaft in diesem Beispiel um eine Protokollierung ergänzt. Es gibt gewöhnlich keinen Grund dafür, Zugriffseigenschaften zu benutzen, wenn Sie lediglich Daten in einer anderen Eigenschaft speichern wollen – dazu können Sie einfach die Eigenschaft selbst verwen-

den. Zugriffseigenschaften sind dann nützlich, wenn Sie durch die Zuweisung eines Werts noch ein anderes Verhalten auslösen wollen oder wenn das Lesen eines Werts die Berechnung des gewünschten Rückgabewerts erforderlich macht.

Hinweis

Es ist nicht notwendig, sowohl eine Get- als auch eine Set-Funktion zu definieren. Wenn Sie nur eine Get-Funktion anlegen, ist die Eigenschaft schreibgeschützt. Jegliche Versuche, die Eigenschaft zu überschreiben, schlagen im normalen Modus (nonstrict) stillschweigend fehl und lösen im strengen Modus (strict) einen Fehler aus. Definieren Sie dagegen nur eine Set-Methode, ist die Eigenschaft nur schreibbar. Versuche, den Wert zu lesen, schlagen sowohl im Strict- als auch im normalen Modus stillschweigend fehl.

4.6 Eigenschaftsattribute

Vor ECMAScript 5 gab es keine Möglichkeit, um anzugeben, ob eine Eigenschaft aufzählbar sein soll. Tatsächlich gab es überhaupt keine Möglichkeit, um auf die internen Attribute von Eigenschaften zuzugreifen. In ECMAScript 5 wurde dies durch die Einführung mehrerer Möglichkeiten zum direkten Umgang mit den Eigenschaftsattributen und durch die Einführung neuer Attribute für zusätzliche Merkmale geändert. Es ist jetzt möglich, Eigenschaften zu erstellen, die sich genauso verhalten wie die eingebauten JavaScript-Eigenschaften. In diesem Abschnitt geht es um die Attribute von Daten- und von Zugriffseigenschaften. Dabei beginnen wir mit den Attributen, die diesen beiden Arten von Eigenschaften gemeinsam sind.

4.6.1 Gemeinsame Attribute

Daten- und Zugriffseigenschaften verfügen über zwei gemeinsame Attribute. Das eine ist `[[Enumerable]]`, das angibt, ob eine Iteration über die Eigenschaft möglich ist. Das andere heißt `[[Configurable]]` und bestimmt, ob die Eigenschaft geändert werden kann. Eine konfigurierbare Eigenschaft können Sie mit `delete` entfernen, außerdem können Sie jederzeit ihre Attribute ändern. (Das bedeutet auch, dass Sie aus konfigurierbaren Dateneigenschaften Zugriffseigenschaften machen können und