Grunde genommen eine Gehirntransplantation: Die gesamte Vererbungshierarchie des Objekts wird ausgetauscht. Möglicherweise gibt es Ausnahmefälle, in denen eine solche Operation sinnvoll ist, aber der gesunde Menschenverstand sollte einem schon sagen, dass man die Vererbungshierarchie lieber unangetastet lässt.

Object.create

Um neue Objekte mit einer maßgeschneiderten Prototypbeziehung zu erstellen, können Sie in ES5 `Object.create` verwenden. Für Umgebungen, die ES5 nicht implementieren, bietet Thema 33 eine portierbare Implementierung von `Object.create`, die sich nicht auf `__proto__` stützt.

Was Sie sich merken sollten

- Ändern Sie niemals die Eigenschaft `__proto__` eines Objekts.
- Mit `Object.create` können Sie einen maßgeschneiderten Prototyp für neue Objekte erstellen.

Erstellen Sie Konstruktoren, die auch ohne new funktionieren

Thema 33

Wenn Sie einen Konstruktor wie die Funktion `User` aus Thema 30 erstellen, müssen Sie sich darauf verlassen, dass er immer mit dem Operator `new` aufgerufen wird. Beachten Sie, dass die Funktion davon ausgeht, dass der Empfänger ein brandneues Objekt ist:

```
function User(name, passwordHash) {
   this.name = name;
   this.passwordHash = passwordHash;
}
```

Wenn der Aufrufer das Schlüsselwort `new` versehentlich nicht angibt, wird das globale Objekt zum Empfänger der Funktion:

```
var u = User("baravelli", "d8b74df393528d51cd19980ae0aa028e");
u;                    // undefined
this.name;            // "baravelli"
this.passwordHash;    // "d8b74df393528d51cd19980ae0aa028e"
```

Das führt nicht nur dazu, dass die Funktion `undefined` zurückgibt, obwohl es nicht nötig wäre, sondern sie erstellt auch die globalen Variablen `name` und `passwordHash` (oder ändert sie, falls sie bereits vorhanden waren) – und das ist wirklich katastrophal!

Ist die Funktion als ES5-Code im Strict Mode definiert, lautet der Standardempfänger `undefined`:

```
function User(name, passwordHash) {
   "use strict";
   this.name = name;
   this.passwordHash = passwordHash;
}

var u = User("baravelli", "d8b74df393528d51cd19980ae0aa028e");
// Fehler: this ist undefined.
```

In diesem Fall führt der problematische Aufruf sofort zu einem Fehler: Die erste Zeile von `User` versucht eine Zuweisung zu `this.name` durchzuführen, was einen `TypeError`-Fehler auslöst. Wurde in der Konstruktorfunktion der Strict Mode gesetzt, kann der Aufrufer den Bug zumindest schnell erkennen und korrigieren.

In jedem Fall aber ist die Funktion `User` instabil. Wenn sie mit `new` aufgerufen wird, funktioniert sie wie erwartet, aber bei einem Aufruf als normale Funktion schlägt sie fehl. Eine sicherere Vorgehensweise besteht darin, eine Funktion zu schreiben, die unabhängig von der Art ihres Aufrufs als Konstruktor wirkt. Eine einfache Möglichkeit dazu besteht darin, zu prüfen, ob der Empfängerwert eine gültige Instanz von `User` ist:

```
function User(name, passwordHash) {
   if (!(this instanceof User)) {
      return new User(name, passwordHash);
   }
   this.name = name;
   this.passwordHash = passwordHash;
}
```

Auf diese Weise führt der Aufruf von `User` zu einem Objekt, das von `User.prototype` erbt, und zwar unabhängig davon, ob sie als Funktion oder als Konstruktor aufgerufen wird:

```
var x = User("baravelli", "d8b74df393528d51cd19980ae0aa028e");
var y = new User("baravelli", "d8b74df393528d51cd19980ae0aa028e");
x instanceof User; // true
y instanceof User; // true
```

Ein Nachteil dieser Technik besteht jedoch darin, dass dafür ein zusätzlicher Funktionsaufruf erforderlich ist, was sie etwas teurer macht. Außerdem lässt sie sich schwer für variadische Funktionen gebrauchen (siehe Thema 21 und 22), da es kein einfaches Gegenstück zur Methode `apply` für den Aufruf variadischer Funktionen als Konstruktoren gibt.

Object.create

Es gibt jedoch die folgende, etwas exotisch anmutende Vorgehensweise, die die ES5-Funktion `Object.create` nutzt:

```
function User(name, passwordHash) {
   var self = this instanceof User
      ? this
      : Object.create(User.prototype);
   self.name = name;
   self.passwordHash = passwordHash;
   return self;
}
```

`Object.create` nimmt ein Prototypobjekt entgegen und gibt ein neues Objekt zurück, das von diesem Prototyp erbt. Wenn diese Version von `User` als Funktion aufgerufen wird, ergibt sich daher ein neues Objekt, das von `User.prototype` erbt und in dem die Eigenschaften `name` und `passwordHash` initialisiert sind.

Zwar ist `Object.create` nur in ES5 verfügbar, doch können Sie in älteren Umgebungen etwas Ähnliches tun, indem Sie einen lokalen Konstruktor erstellen und mit `new` instanziieren:

```
if (typeof Object.create === "undefined") {
   Object.create = function(prototype) {
      function C() { }
      C.prototype = prototype;
      return new C();
   };
}
```

(Beachten Sie, dass hiermit nur eine Version von `Object.create` mit einem einzigen Argument implementiert wird. Die echte Version nimmt auch ein optionales zweites Argument an, das einen Satz von Eigenschaftsbeschreibungen zur Definition des neuen Objekts angibt.)

Konstruktorüberschreibung

Was aber geschieht, wenn jemand diese neue Version von `User` mit `new` aufruft? Dank der Technik der *Konstruktorüberschreibung* (*Constructor Override*) verhält sie sich genauso wie bei einem Funktionsaufruf. Das liegt daran, dass es in JavaScript zulässig ist, das Ergebnis eines `new`-Ausdrucks mit einem ausdrücklichen `return` einer Konstruktorfunktion zu überschreiben. Wenn `User` den Wert `self` zurückgibt, erhält der `new`-Ausdruck das Ergebnis `self`, was ein anderes Objekt sein kann als das an `this` gebundene.

Einen Konstruktor vor der falschen Verwendung zu schützen, ist nicht immer die Mühe wert, vor allem, wenn Sie den Konstruktor nur lokal einsetzen. Trotzdem ist es wichtig zu wissen, welche schlimmen Auswirkungen es haben kann, wenn ein Konstruktor auf die falsche

Weise aufgerufen wird. Zumindest sollten Sie die Stellen dokumentieren, an denen eine Konstruktorfunktion erwartet, mit `new` aufgerufen zu werden, vor allem wenn sie in umfangreichem Code oder in einer gemeinsamen Bibliothek steht.

Was Sie sich merken sollten

- Machen Sie Konstruktoren unabhängig von der Syntax des Aufrufers, indem Sie sie selbst mit `new` oder `Object.create` neu aufrufen.
- Dokumentieren Sie deutlich die Stellen, an denen eine Funktion mit `new` aufgerufen werden muss.

Thema 34 Speichern Sie Methoden mithilfe von Prototypen

Es ist durchaus möglich, JavaScript-Programme ohne Prototypen zu schreiben. Die Klasse `User` aus Thema 30 könnten wir wie folgt definieren, ohne irgendetwas Besonderes in ihrem Prototyp zu definieren:

```
function User(name, passwordHash) {
   this.name = name;
   this.passwordHash = passwordHash;
   this.toString = function() {
      return "[User " + this.name + "]";
   };
   this.checkPassword = function(password) {
      return hash(password) === this.passwordHash;
   };
}
```

In den meisten Verwendungszwecken verhält sich diese Klasse genauso wie die ursprüngliche Implementierung. Wenn wir aber mehrere Instanzen von `User` erstellen, zeigt sich ein entscheidender Unterschied:

```
var u1 = new User(/* ... */);
var u2 = new User(/* ... */);
var u3 = new User(/* ... */)
```

Abbildung 4–3 zeigt, wie diese drei Objekte und ihr Prototypobjekt aussehen. Anstatt die Methoden `toString` und `checkPassword` des Prototyps gemeinsam zu nutzen, enthält jede Instanz ein Exemplar dieser beiden Methoden, was insgesamt sechs Funktionsobjekte ergibt.

In Abbildung 4–4 können Sie sehen, wie diese drei Objekte und ihr Prototypobjekt in der ursprünglichen Definition aussehen. Hier werden die Methoden `toString` und `checkPassword` nur einmal erstellt und dann über den Prototyp von allen Instanzen gemeinsam verwendet.