heitliche Schreibweisen erlauben es den Benutzern, zu erraten, welche Eigenschaften und Methoden zur Verfügung stehen, ohne dass sie sie nachschlagen müssen, oder umgekehrt das Verhalten von gegebenen Eigenschaften und Methoden aus den Namen zu schließen.

Was Sie sich merken sollten

- Verwenden Sie einheitliche Schreibweisen für Variablennamen und Funktionssignaturen.
- Weichen Sie nicht von den Konventionen ab, die in anderen Teilen der Entwicklungsplattform Ihrer Benutzer vorherrschen.

Behandeln Sie »undefined« als »nicht vorhanden« Thema 54

`undefined` ist ein besonderer Wert. Wenn JavaScript keinen bestimmten Wert hat, den es bereitstellen kann, gibt es `undefined` aus. Nicht zugewiesene Variablen haben zu Anfang den Wert `undefined`:

```
var x;
x; // undefined
```

Auch der Zugriff auf nicht vorhandene Eigenschaften eines Objekts ergibt `undefined`:

```
var obj = {};
obj.x; // undefined
```

Wenn eine Funktion die Steuerung zurückgibt, ohne einen Wert bereitzustellen, oder wenn am Ende des Funktionsrumpfes eine Prüfung durchfällt, ist der Rückgabewert ebenfalls `undefined`:

```
function f() {
   return;
}
function g() { }
f(); // undefined
g(); // undefined
```

Funktionsparameter, für die keine echten Argumente bereitgestellt werden, haben ebenfalls den Wert `undefined`:

```
function f(x) {
   return x;
}

f(); // undefined
```

Undefined als Lückenbüßer

In all diesen Situationen bedeutet der Wert `undefined`, dass die Operation keinen konkreten Wert ergeben hat. Es klingt natürlich ein bisschen paradox, wenn ein Wert bedeutet, dass es keinen Wert gibt. Da aber jede Operation *irgendetwas* ergeben muss, verwendet JavaScript `undefined` sozusagen als Lückenbüßer.

Andere Verwendungen

Die Sprache selbst legt fest, dass `undefined` als das Fehlen eines Wertes behandelt werden sollte. Die Verwendung von `undefined` auf andere Weise ist riskant. Betrachten wir als Beispiel eine Bibliothek für die Elemente von Benutzeroberflächen, die die Methode `highlight` zur Änderung der Hintergrundfarbe eines Elements enthält:

```
element.highlight();          // Verwendet die Standardfarbe
element.highlight("yellow"); // Verwendet eine benutzerdefinierte
                              // Farbe
```

Was geschieht, wenn wir eine Zufallsfarbe haben möchten? Wir könnten `undefined` als besonderen Wert für diesen Zweck verwenden:

```
element.highlight(undefined); // Verwendet eine Zufallsfarbe
```

Das steht aber in Konflikt mit der üblichen Bedeutung von `undefined`. Dadurch kann sehr leicht ein falsches Verhalten hervorgerufen werden, wenn Sie den Wert aus einer anderen Quelle beziehen, vor allem dann, wenn diese Quelle gar keinen Wert bereitstellt. Nehmen wir an, ein Programm verwendet ein Konfigurationsobjekt mit einer optionalen Voreinstellung für die Farbe:

```
var config = JSON.parse(preferences);
// ...
element.highlight(config.highlightColor); // Kann zufällig sein
```

Wenn in der Voreinstellung keine Farbe angegeben ist, sollte normalerweise die Standardfarbe eingestellt werden – ganz so, als ob gar kein Wert angegeben wurde. Wenn wir aber `undefined` zweckentfremden, sorgen wir dafür, dass dieser Code eine zufällige Farbe auswählt. Besser ist es, in der API einen eigenen Namen für eine Zufallsfarbe zu verwenden:

```
element.highlight("random");
```

Manchmal ist es jedoch nicht möglich, dass eine API einen besonderen Stringwert außerhalb der üblichen von der Funktion akzeptierten Stringwerte auswählt. In solchen Fällen können Sie aber auf andere Sonderwerte als `undefined` zurückgreifen, z.B. auf `null` oder `true`. Dadurch wird der Code aber schlecht lesbar:

```
element.highlight(null);
```

Für jemanden, der Ihre Bibliothek nicht in- und auswendig kennt, ist dieser Code ziemlich schleierhaft. Wahrscheinlich wird er spontan davon ausgehen, dass hierdurch die Hervorhebung ganz ausgeschaltet wird. Eine ausführlichere und deutlichere Möglichkeit besteht darin, die Zufallsfarbe als Objekt mit der Eigenschaft `random` darzustellen (siehe Thema 55 über Optionsobjekte):

```
element.highlight({ random: true });
```

Eine weitere Stelle, an der Sie auf `undefined` aufpassen müssen, ist die Implementierung von optionalen Argumenten. Theoretisch ist es mit dem `arguments`-Objekt (siehe Thema 51) möglich, zu erkennen, ob ein Argument übergeben wurde, doch in der Praxis ergeben sich stabilere APIs, wenn Sie einen Test auf `undefined` durchführen. Beispielsweise kann ein Webserver optional einen Hostnamen akzeptieren:

```
var s1 = new Server(80, "example.com");
var s2 = new Server(80); // Standardwert ist "localhost"
```

Der Konstruktor `Server` kann durch eine Prüfung von `arguments.length` implementiert werden:

```
function Server(port, hostname) {
   if (arguments.length < 2) {
      hostname = "localhost";
   }
   hostname = String(hostname);
   // ...
}
```

Hier stellt sich jedoch ein ähnliches Problem wie bei der zuvor erwähnten Methode `element.highlight`. Wenn ein Programm ein explizites Argument bereitstellt, indem es einen Wert aus einer anderen Quelle anfordert, beispielsweise aus einem Konfigurationsobjekt, dann kann dabei `undefined` herauskommen:

```
var s3 = new Server(80, config.hostname);
```

Falls `config` keine Voreinstellung für `hostname` festlegt, wird normalerweise auf den Standardwert `"localhost"` zurückgegriffen. Die obenstehende Implementierung führt aber dazu, dass letzten Endes der Hostname `"undefined"` verwendet wird. Da es durchaus möglich ist, dass das Argument weggelassen oder ein Argumentausdruck bereitgestellt wurde, der sich als `undefined` herausstellt, ist es besser, zu prüfen, ob `undefined` vorliegt:

```
function Server(port, hostname) {
   if (hostname === undefined) {
```

```
        hostname = "localhost";
    }
    hostname = String(hostname);
    // ...
}
```

Eine sinnvolle Alternative besteht darin, zu prüfen, ob `hostname` in `true` konvertiert werden kann (siehe Thema 3). Das lässt sich mithilfe logischer Operatoren unkompliziert durchführen:

Hier wird der logische `OR`-Operator (`||`) verwendet, der das erste Argument zurückgibt, wenn es sich dabei um einen in `true` konvertierbaren Wert handelt, und anderenfalls das zweite. Wenn `hostname` also `undefined` oder ein leerer String ist, führt die Auswertung des Ausdrucks (`hostname || "localhost"`) zu `"localhost"`. Technisch gesehen, findet hier eine Prüfung auf noch mehr Werte als nur auf `undefined` statt, denn tatsächlich werden alle in `false` konvertierbaren Werte genauso gehandhabt wie `undefined`. Da ein leerer String kein gültiger Hostname sein kann, ist diese Vorgehensweise für `Server` durchaus akzeptabel. Wenn Sie also mit einer nicht so strengen API leben können, die alle in `false` konvertierbaren Werte in den Standardwert umwandelt, haben Sie mit der Prüfung auf »Truthiness« daher eine kompakte Möglichkeit, um Standardwerte für Parameter vorzusehen.

Aber Achtung: Die Prüfung auf Truthiness ist nicht in allen Fällen gefahrlos, da sie einen leeren String durch den Standardwert ersetzt, wobei es aber durchaus sein kann, dass eine Funktion leere Strings als gültige Werte übernehmen kann. Aus den gleichen Gründen dürfen Sie für Funktionen, die Zahlen entgegennehmen und dabei auch `0` akzeptieren (oder `NaN`, was jedoch seltener vorkommt), keine Truthiness-Prüfung vornehmen.

Betrachten wir als Beispiel eine Funktion, die Elemente einer Benutzeroberfläche anlegt. Dabei soll es zulässig sein, Elemente mit einer Breite oder Höhe von 0 zu erstellen. Für fehlende Angaben wird dagegen ein anderer Standardwert bereitgestellt:

```
var c1 = new Element(0, 0); // width: 0, height: 0
var c2 = new Element();     // width: 320, height: 240
```

Eine Implementierung mit einem Truthiness-Test würde zum falschen Ergebnis führen:

```
function Element(width, height) {
    this.width = width || 320;   // Falscher Test
    this.height = height || 240; // Falscher Test
    // ...
}
```

```
var c1 = new Element(0, 0);

c1.width;  // 320
c1.height; // 240
```

Hier müssen wir einen ausführlicheren Test auf `undefined` durchführen:

```
function Element(width, height) {
   this.width  = width  === undefined ? 320 : width;
   this.height = height === undefined ? 240 : height;
   // ...
}

var c1 = new Element(0, 0);

c1.width;  // 0
c1.height; // 0

var c2 = new Element();

c2.width;  // 320
c2.height; // 240
```

Was Sie sich merken sollten

- Verzichten Sie darauf, durch `undefined` irgendetwas anderes darzustellen als das Fehlen eines Werts.
- Verwenden Sie zur Darstellung anwendungsspezifischer Flags beschreibende Stringwerte oder Objekte mit benannten booleschen Eigenschaften statt `undefined` oder `null`.
- Um Standardwerte für Parameter bereitzustellen, führen Sie einen Test auf `undefined` durch, anstatt `arguments.length` zu untersuchen.
- Verwenden Sie für die Standardwerte von Parametern, die auch `0`, `NaN` oder leere Strings als gültige Argumente annehmen können, keine Truthiness-Tests.

Zu viele Parameter? Nutzen Sie Optionsobjekte! Thema 55

Wie in Thema 53 gesagt, ist es sehr wichtig, eine einheitliche Argumentreihenfolge einzuhalten, damit sich die Programmierer besser merken können, was die einzelnen Argumente in einem Funktionsaufruf bedeu-