2.4 Falsy- und Truthy-Werte

Dieser »Verrückter Hutmacher«-Abschnitt beschreibt einen verwirrenden Aspekt von JavaScript ausführlicher. Wenn Sie dem Ratschlag aus dem vorherigen Abschnitt folgen und in Bedingungen nur boolesche Werte verwenden, können Sie ihn getrost überspringen.

In JavaScript müssen Bedingungen (also etwa die in einer `if`-Anweisung) nicht unbedingt boolesche Werte sein. Auch die »falschen« Werte `0`, `NaN`, `null`, `undefined` und der leere String lassen eine Bedingung fehlschlagen. Andere Werte werden dagegen als »wahr« gedeutet, sodass die Bedingung als erfüllt gilt. Gewöhnlich spricht man hierbei von Falsy- bzw. Truthy-Werten. Allerdings sind dies keine offiziellen Begriffe der Sprachspezifikation.

Hinweis

Falsy- und Truthy-Werte können auch bei Schleifenbedingungen, den Operanden der booleschen Operatoren `&&`, `||` und `!` sowie dem ersten Operanden von `? :` vorkommen. Alle diese Konstrukte werden noch weiter hinten in diesem Kapitel behandelt.

Auf den ersten Blick erscheinen die Konvertierungsregeln für boolesche Werte vernünftig. Nehmen wir an, Sie wollen lediglich sicherstellen, dass die Variable `performance` nicht `undefined` ist. Also schreiben Sie Folgendes:

```
if (performance) ... // Gefährlich!
```

Der Test schlägt zwar wie erwartet fehl, wenn die Variable `performance` den Wert `undefined` hat (und als Bonus auch, wenn sie `null` ist). Was aber, wenn `performance` ein leerer String oder die Zahl `0` ist? Soll der Test diese Fälle genauso behandeln wie das Fehlen eines Wertes? Manchmal kann das sinnvoll sein, manchmal aber nicht. Besser ist es, wenn der Code deutlich macht, was Sie wirklich erreichen wollen:

```
if (performance !== undefined) ...
```

2.5 Vergleichs- und Gleichheitsoperatoren

JavaScript verfügt über die üblichen Vergleichsoperatoren:

- `<` kleiner als
- `<=` kleiner oder gleich
- `>` größer als
- `>=` größer oder gleich

Beim Vergleich von Zahlen funktionieren diese Operatoren wie erwartet:

```
3 < 4  // true
3 >= 4 // false
```

Jeder Vergleich mit `NaN` ergibt `false`:

```
NaN < 4    // false
NaN >= 4   // false
NaN <= NaN // false
```

Mit denselben Operatoren lassen sich auch Strings vergleichen, wobei die alphabetische Reihenfolge beachtet wird:

```
'Hello' < 'Goodbye' // false: H steht hinter G
'Hello' < 'Hi'      // true: e steht vor i
```

Achten Sie beim Vergleich von Werten mit `<`, `<=`, `>` und `>=` darauf, dass entweder beide Operanden Zahlen oder beide Operanden Strings sind. Wandeln Sie die Operanden ggf. ausdrücklich um. Anderenfalls konvertiert JavaScript die Operanden. Das führt manchmal zu unerwünschten Ergebnissen, wie Sie im nächsten Abschnitt sehen werden.

Zum Test auf Gleichheit verwenden Sie die folgenden Operatoren:

`===` strikt gleich
`!==` nichtstrikt gleich

Diese strikten Gleichheitsoperatoren sind unproblematisch. Operanden unterschiedlicher Typen sind niemals exakt gleich. Die Werte `undefined` und `null` sind nur zu sich selbst strikt gleich. Zahlen, boolesche Werte und Strings sind nur dann strikt gleich, wenn ihre Werte gleich sind.

```
'42' === 42        // false: unterschiedliche Typen
undefined === null // false
'42' === '4' + 2   // tru: derselbe String-Wert, nämlich '42'
```

Daneben gibt es die schwachen Gleichheitsoperatoren `==` und `!=`, die auch Werte unterschiedlicher Typen vergleichen. Das ist im Allgemeinen nicht nützlich. Genaueres dazu erfahren Sie im nächsten Abschnitt.

Vorsicht

Es ist nicht möglich, `x === NaN` zu verwenden, um zu prüfen, ob `x` gleich `NaN` ist. Keine zwei `NaN`-Werte werden als gleich angesehen. Rufen Sie stattdessen `Number.isNaN(x)` auf.

Hinweis

`Object.is(x, y)` ist fast identisch mit `x === y`. Die einzigen Ausnahmen bestehen darin, dass `Object.is(+0, -0)` zu `false` ausgewertet wird und `Object.is(NaN, NaN)` zu `true`.

Wie in Java und Python bedeutet die Gleichheit von Objekten (einschließlich Arrays), dass die beiden Operanden auf dasselbe Objekt verweisen. Verweise auf verschiedene Objekte sind niemals gleich, selbst wenn die beiden Objekte den gleichen Inhalt haben.

```
let harry = { name: 'Harry Smith', age: 42 }
let harry2 = harry
harry === harry2 // true: zwei Verweise auf dasselbe Objekt
let harry3 = { name: 'Harry Smith', age: 42 }
harry === harry3 // false: verschiedene Objekte
```

2.6 Vergleiche unterschiedlicher Typen

Dies ist ein weiterer »Verrückter Hutmacher«-Abschnitt, der einen verwirrenden Aspekt von JavaScript ausführlicher beschreibt. Sie können ihn getrost überspringen, wenn Sie die goldene Regel Nr. 3 befolgen und Vergleiche unterschiedlicher Typen sowie vor allem die schwachen Gleichheitsoperatoren `==` und `!=` vermeiden.

Als Erstes sehen wir uns hier Vergleiche unterschiedlicher Typen mit den Operatoren `<`, `<=`, `>` und `>=` an.

Wenn ein Operand eine Zahl ist, wird der andere ebenfalls in eine Zahl umgewandelt. Nehmen wir an, der zweite Operator ist ein String. In diesem Fall wird er in den zugehörigen numerischen Wert umgewandelt, wenn er die String-Darstellung einer Zahl ist, in `0`, wenn der String leer ist, und in allen anderen Fällen in `NaN`. Jeder Vergleich mit `NaN` ergibt `false`, selbst `NaN <= NaN`.

```
'42' < 5     // false: '42' wird in die Zahl 42 umgewandelt
'' < 5       // true: '' wird in die Zahl 0 umgewandelt
'Hello' <= 5 // false: 'Hello' wird in NaN umgewandelt
5 <= 'Hello' // false: 'Hello' wird in NaN umgewandelt
```

3

Funktionen und funktionale Programmierung

In diesem Kapitel erfahren Sie, wie Sie in JavaScript Funktionen schreiben. JavaScript ist eine funktionale Programmiersprache. Funktionen sind ebenso wie Zahlen oder Strings Werte erster Klasse. Sie können Funktionen entgegennehmen und andere Funktionen ausgeben. Für die Arbeit mit modernem JavaScript ist es unverzichtbar, die Kunst der funktionalen Programmierung beherrschen zu lernen.

Des Weiteren behandelt dieses Kapitel die Regeln für die Parameterübergabe und den Gültigkeitsbereich in JavaScript sowie das Auslösen und Abfangen von Exception.

3.1 Funktionen deklarieren

Um in JavaScript eine Funktion zu deklarieren, müssen Sie Folgendes angeben:

1. den Namen der Funktion,
2. die Namen der Parameter,
3. den Rumpf der Funktion, der das Funktionsergebnis berechnet und zurückgibt.

Die Typen der Funktionsparameter und des Ergebnisses geben Sie dagegen nicht an. Betrachten Sie dazu das folgende Beispiel:

```
function average(x, y) {
  return (x + y) / 2
}
```

Die Anweisung `return` sorgt dafür, dass der in der Funktion berechnete Wert zurückgegeben wird.

Um diese Funktion aufzurufen, müssen Sie lediglich die erforderlichen Argumente übergeben:

```
let result = average(6, 7) // result wird auf 6.5 gesetzt
```

Was aber geschieht, wenn Sie etwas anderes als Zahlen übergeben? Was auch immer passiert, passiert. Zum Beispiel:

```
result = average('6', '7') // result wird auf 33.5 gesetzt
```

Wenn Sie Strings übergeben, werden sie durch den Operator + im Funktionsrumpf verkettet. Der resultierende String `'67'` wird dann vor der Division durch 2 in eine Zahl umgewandelt.

Auf Java-, C#- oder C++-Programmierer, die an eine Typüberprüfung zur Kompilierzeit gewöhnt sind, mag das geradezu fahrlässig wirken. Tatsächlich können Sie Fehler bei den Argumenttypen erst erkennen, wenn zur Laufzeit merkwürdige Dinge passieren. Aber dadurch ist es möglich, Funktionen zu schreiben, die Argumente verschiedener Typen akzeptieren, was sehr komfortabel sein kann.

Die Anweisung `return` gibt die Steuerung unmittelbar zurück und verwirft den Rest der Funktion. Betrachten Sie dazu die folgende Funktion `indexOf`, die den Index eines Wertes in einem Array berechnet:

```
function indexOf(arr, value) {
  for (let i in arr) {
    if (arr[i] === value) return i
  }
 return -1
}
```

Sobald eine Übereinstimmung gefunden wurde, wird der Index zurückgegeben und die Funktion beendet.

In einer Funktion muss nicht unbedingt ein Rückgabewert angegeben sein. Wenn der Funktionsrumpf ohne `return` beendet wird oder wenn kein Ausdruck auf das Schlüsselwort `return` folgt, gibt die Funktion `undefined` zurück. Das geschieht gewöhnlich, wenn die Funktion einzig und allein aufgrund ihrer Seiteneffekte aufgerufen wird.

Tipp

Wenn eine Funktion nur manchmal ein Ergebnis zurückgibt, Sie aber nicht wollen, dass Sie jemals irgendetwas zurückgibt, dann legen Sie das explizit fest:

```
return undefined
```

Hinweis

Wie in Kapitel 2 bereits erwähnt, muss bei einer `return`-Anweisung immer mindestens ein Token vor dem Zeilenende stehen, um zu verhindern, dass automatisch ein Semikolon eingefügt wird. Wenn beispielsweise eine Funktion ein Objekt zurückgibt, müssen Sie daher wenigstens die öffnende geschweifte Klammer in dieselbe Zeile schreiben:

```
return {
  average: (x + y) / 2,
  max: Math.max(x, y),
  ...
}
```

3.2 Funktionen höherer Ordnung

JavaScript ist eine funktionale Programmiersprache. Funktionen sind Werte und können daher in Variablen gespeichert, als Argumente übergeben und als Ergebnisse von anderen Funktionen zurückgegeben werden.

Beispielsweise können wir die Funktion `average` in einer Variablen speichern:

```
let f = average
```

Anschließend können Sie die Funktion wie folgt aufrufen:

```
let result = f(6, 7)
```