Fließkommazahlen bilden einen Kompromiss zwischen Genauigkeit und Leistung. Wenn es auf die Genauigkeit ankommt, ist es wichtig, sich über die Grenzen im Klaren zu sein. Eine Lösung besteht darin, nach Möglichkeit mit Integerwerten zu arbeiten, da sie nicht gerundet werden. Bei Berechnungen mit monetären Werten rechnen viele Programmierer die Werte in die kleinste Einheit der betreffenden Währung um, sodass sie mit ganzen Zahlen arbeiten können. Wenn es in der oben stehenden Berechnung beispielsweise um Werte in Euro geht, dann können wir stattdessen in Cent rechnen, sodass wir es nur mit ganzen Zahlen zu tun haben:

```
(10 + 20) + 30; // 60
10 + (20 + 30); // 60
```

Bei Integern müssen Sie jedoch nach wie vor darauf Acht geben, dass die Berechnungen den Bereich zwischen -2^{53} und 2^{53} nicht überschreiten. Allerdings brauchen Sie sich keine Sorgen mehr über Rundungsfehler zu machen.

Was Sie sich merken sollten

- Zahlen in JavaScript sind Fließkommazahlen doppelter Genauigkeit.
- Integer in JavaScript sind lediglich eine Teilmenge der Fließkommazahlen doppelter Genauigkeit und kein eigenständiger Datentyp.
- Bitweise Operatoren behandeln Zahlen als 32-Bit-Integer mit Vorzeichen.
- Bei der Fließkommaarithmetik müssen Sie auf die Grenzen der Genauigkeit achten.

Thema 3 Vorsicht bei der impliziten Typumwandlung

Bei Typfehlern verhält sich JavaScript manchmal überraschend nachsichtig. Viele Sprachen stufen einen Ausdruck wie den folgenden als Fehler ein, denn schließlich sind boolesche Ausdrücke wie `true` mit Rechenaufgaben unvereinbar:

```
3 + true; // 4
```

In einer statisch typisierten Sprache könnte ein Programm mit einem solchen Ausdruck nicht einmal ausgeführt werden. In einigen dyna-

misch typisierten Sprachen würde das Programm zwar laufen, doch dieser Ausdruck würde eine Ausnahme auslösen. JavaScript dagegen führt das Programm nicht nur aus, sondern liefert auch frohgemut das Ergebnis 4!

Es gibt auch in JavaScript eine Handvoll Fälle, in denen ein falscher Typ zu einem sofortigen Fehler führt. Das ist beispielsweise der Fall, wenn Sie etwas aufrufen, was keine Funktion ist, oder wenn Sie versuchen, eine Eigenschaft von `null` auszuwählen:

```
"hello"(1);  // Fehler: Keine Funktion
null.x;      // Fehler: Kann die Eigenschaft 'x' von null
             // nicht lesen
```

Implizite Konvertierung

In vielen anderen Fällen ruft JavaScript keinen Fehler hervor, sondern wandelt den Wert mithilfe verschiedener automatischer Konvertierungsprotokolle *implizit* in den erwarteten Typ um.[2] Beispielsweise wird bei den Operatoren `-`, `*`, `/` und `%` immer versucht, die Argumente in Zahlen umzuwandeln, bevor die Berechnung durchgeführt wird. Beim Operator `+` ist die Sachlage etwas komplizierter, da er überladen ist und je nach Typ der Argumente numerische Additionen oder Stringverkettungen durchführt:

```
2 + 3;                // 5
"hello" + " world";   // "hello world"
```

Was aber geschieht, wenn Sie eine Zahl mit einem String kombinieren? JavaScript entscheidet sich in diesem Fall für die Strings und wandelt auch die Zahl in einen String um:

```
"2" + 3;        // "23"
2 + "3";        // "23"
```

Die Vermischung von Typen auf diese Weise kann verwirrende Formen annehmen, vor allem, da es hierbei auch auf die Reihenfolge der Operationen ankommt. Betrachten Sie als Beispiel folgenden Ausdruck:

```
1 + 2 + "3";    // "33"
```

Da die Addition gruppenweise von links nach rechts verläuft (sie ist *linksassoziativ*), entspricht dies dem folgenden Ausdruck:

```
(1 + 2) + "3";  // "33"
```

Der folgende Ausdruck dagegen ergibt den String `"123"`:

```
1 + "2" + 3;    // "123"
```

---

2. *Im Englischen wird diese implizite Typumwandlung Coercion genannt.*

Auch hier schreibt die Linksassoziativität wieder vor, dass der Ausdruck identisch mit dem folgenden ist, bei dem die Addition auf der linken Seite in Klammern steht:

```
(1 + "2") + 3; // "123"
```

Wie in Thema 2 beschrieben, wandeln die bitweisen Operatoren ihre Argumente nicht nur in Zahlen, sondern auch in die Teilmenge der Zahlen um, die sich als 32-Bit-Integer darstellen lassen. Das betrifft die bitweisen arithmetischen Operatoren (`~`, `&`. `^` und `|`) sowie die Verschiebeoperatoren (`<<`, `>>` und `>>>`).

Diese impliziten Umwandlungen können verführerisch bequem sein – beispielsweise um Strings aus Benutzereingaben, Textdateien oder Netzwerkstreams automatisch konvertieren zu lassen:

```
"17" * 3;    // 51
"8" | "1";   // 9
```

Achtung: Fehler können verschleiert werden!

Durch implizite Typumwandlungen können aber auch Fehler verschleiert werden. Beispielsweise ruft eine Variable, die sich als `null` herausstellt, in einer arithmetischen Berechnung keinen Fehler hervor, sondern wird stillschweigend zu `0` konvertiert. Eine Variable, die `undefined` ist, wird in den besonderen Fließkommawert `NaN` umgewandelt (was paradoxerweise »not a number«, also »keine Zahl« heißt; das liegt am Fließkommastandard der IEEE). Anstatt sofort eine Ausnahme auszulösen, sorgen die impliziten Konvertierungen dafür, dass die Berechnungen fortgesetzt werden, was dann häufig zu verwirrenden und unvorhersehbaren Ergebnissen führt. Leider ist es ganz besonders schwer, einen Test auf Vorliegen des `NaN`-Werts durchzuführen, und das aus zwei Gründen. Erstens folgt JavaScript dem IEEE-Fließkommastandard mit seiner verwirrenden Anforderung, dass `NaN` als ungleich zu sich selbst behandelt werden muss. Daher kann man nicht prüfen, ob ein Wert gleich `NaN` ist:

```
var x = NaN;
x === NaN;          // false
```

Zweitens ist auch die standardmäßige Bibliotheksfunktion `isNaN` nicht sehr zuverlässig, da sie eine eigene implizite Typumwandlung durchführt und ihr Argument vor dem Test erst in eine Zahl umwandelt. (Ein passenderer Name für `isNaN` wäre wahrscheinlich `convertsToNaN` gewesen.) Wenn Sie bereits wissen, dass ein Wert eine Zahl ist, können Sie mit `isNaN` wie folgt prüfen, ob er `NaN` ist:

```
isNaN(NaN);         // true
```

Andere Werte, die zwar definitiv nicht `NaN` sind, aber implizit in `NaN` umgewandelt werden können, sind für `isNaN` jedoch nicht unterscheidbar:

```
isNaN("foo");       // true
isNaN(undefined);   // true
isNaN({});                      // true
isNaN({ valueOf: "foo" });      // true
```

Zum Glück gibt es jedoch ein Idiom für den Test auf `NaN`, das sowohl zuverlässig als auch knapp ist, jedoch nicht unbedingt unmittelbar einsichtig. Da `NaN` der einzige Wert in JavaScript ist, der als ungleich mit sich selbst behandelt wird, können Sie immer prüfen, ob ein Wert `NaN` ist, indem Sie die Gleichheit mit sich selbst testen:

```
var a = NaN;
a !== a;                    // true
var b = "foo";
b !== b;                    // false
var c = undefined;
c !== c;                    // false
var d = {};
d !== d;                    // false
var e = { valueOf: "foo" };
e !== e;                    // false
```

Diese Technik können Sie zu einer sprechend benannten Hilfsfunktion abstrahieren:

```
function isReallyNaN(x) {
   return x !== x;
}
```

Der Test eines Wertes auf Ungleichheit mit sich selbst ist jedoch so knapp, dass er gewöhnlich ohne eine Hilfsfunktion durchgeführt wird. Daher ist es wichtig, diesen Test und seinen Zweck zu begreifen.

Die stillschweigende Konvertierung kann das Debugging von fehlerhaften Programmen erheblich erschweren, da sie die Ursachen verschleiert und die Diagnose erschwert. Wenn eine Berechnung schiefgeht, besteht die beste Vorgehensweise zur Behebung darin, die Zwischenergebnisse zu untersuchen und sich bis zu dem Punkt zurückzuarbeiten, an dem die Probleme aufzutreten beginnen. Von dieser Stelle aus können Sie dann die Argumente der einzelnen Operationen untersuchen und nach solchen Ausschau halten, die den falschen Typ aufweisen. Es kann verschiedene Arten von Fehlern geben: logische, bei denen der falsche arithmetische Operator verwendet wird, und Typfehler, beispielsweise die Übergabe von `undefined` statt einer Zahl.

Objekte werden zu Strings.

Objekte können auch implizit in primitive Datentypen umgewandelt werden. Das wird am häufigsten für die Konvertierung in Strings genutzt:

```
"the Math object: " + Math;  // "the Math object: [object Math]"
"the JSON object: " + JSON;  // "the JSON object: [object JSON]"
```

toString

Bei der impliziten Umwandlung von Objekten in Strings wird ihre `toString`-Methode aufgerufen. Das können Sie sich ansehen, indem Sie diese Methode selbst aufrufen:

```
Math.toString();   // "[object Math]"
JSON.toString();   // "[object JSON]"
```

valueOf

Auf ähnliche Weise lassen sich Objekte mithilfe ihrer `valueOf`-Methode auch in Zahlen umwandeln. Die Typumwandlung von Objekten können Sie durch die Definition dieser Methoden beeinflussen:

```
"J" + { toString: function() { return "S"; } };  // "JS"
2 * { valueOf: function() { return 3; } };       // 6
```

Der +-Operator ist überladen.

Auch hier wird die Sache etwas kompliziert, da der Operator + überladen ist, um sowohl Stringverkettungen als auch Additionen durchzuführen. Vor allem bei Objekten, die sowohl über eine `toString`- als auch über eine `valueOf`-Methode verfügen, ist es nicht offensichtlich, welche dieser Methoden der Operator + aufrufen soll. Er soll zwar anhand der Typen zwischen Verkettung und Addition wählen, aber bei der impliziten Konvertierung sind die Typen nicht gegeben! In JavaScript wird diese Mehrdeutigkeit dadurch gelöst, dass `valueOf` willkürlich gegenüber `toString` bevorzugt wird. Das bedeutet aber, dass Sie ein unerwartetes Ergebnis bekommen können, wenn Sie versuchen, eine Stringverkettung mit einem Objekt durchzuführen:

```
var obj = {
   toString: function() {
      return "[object MyObject]";
   },
   valueOf: function() {
      return 17;
   }
};
"object: " + obj; // "object: 17"
```

Die Moral von der Geschicht' lautet, dass `valueOf` wirklich nur für Objekte geeignet ist, die numerische Werte darstellen, z. B. `Number`-Objekte. Für solche Objekte geben die Methoden `toString` und `valueOf` konsistente Ergebnisse zurück – eine String- oder Zahlendarstellung derselben Zahl. Daher verhält sich der überladene Operator + stets einheitlich, unabhängig davon, ob das Objekt zur Verkettung oder zur Addition verwendet wird. Im Allgemeinen ist die implizite Konvertie-

rung in Strings weit gebräuchlicher und nützlicher als die in Zahlen. Am besten ist es, `valueOf` zu vermeiden, sofern das Objekt nicht tatsächlich eine numerische Abstraktion ist und `obj.toString()` eine Stringdarstellung von `obj.valueOf()` ausgibt.

Truthiness

Die letzte Art von impliziter Konvertierung wird manchmal auch als *Truthiness* bezeichnet. Operatoren wie `if`, `||` und `&&` arbeiten logisch mit booleschen Werten, akzeptieren tatsächlich aber beliebige Werte. In JavaScript werden die Eingangswerte aufgrund einer einfachen impliziten Konvertierung als boolesche Werte aufgefasst. Dabei werden die meisten Werte implizit in `true` umgewandelt (solche Werte werden manchmal als »*truthy*« bezeichnet). Das gilt beispielsweise für alle Objekte. Im Gegensatz zur automatischen Konvertierung in Strings und Zahlen werden bei der Truthiness jedoch keine impliziten Umwandlungsmethoden aufgerufen. Es gibt genau sieben Werte, die in `false` konvertiert werden (»*falsy*«): `false`, `0`, `-0`, `""`, `NaN`, `null` und `undefined`. Da Zahlen und Strings zu `false` umgewandelt werden können, ist es nicht immer verlässlich, mithilfe der Truthiness zu prüfen, ob ein Funktionsargument oder eine Objekteigenschaft definiert ist. Betrachten Sie als Beispiel eine Funktion, die optionale Argumente mit Standardwerten annimmt:

```
function point(x, y) {
   if (!x) {
      x = 320;
   }
   if (!y) {
      y = 240;
   }
   return { x: x, y: y };
}
```

Diese Funktion ignoriert alle Argumente, die implizit in `false` konvertiert werden, also auch `0`:

```
point(0, 0); // { x: 320, y: 240 }
```

typeof

Eine zuverlässigere Möglichkeit für einen Test auf `undefined` bietet `typeof`:

```
function point(x, y) {
   if (typeof x === "undefined") {
      x = 320;
   }
   if (typeof y === "undefined") {
      y = 240;
   }
   return { x: x, y: y };
}
```

Diese Version von `point` unterscheidet korrekt zwischen `0` und `undefined`:

```
point();     // { x: 320, y: 240 }
point(0, 0); // { x: 0, y: 0 }
```

Vergleich mit undefined

Eine andere Möglichkeit besteht darin, einen Vergleich mit `undefined` durchzuführen:

```
if (x === undefined) { ... }
```

In Thema 54 geht es um die Bedeutung von Truthiness-Tests für die Gestaltung von Bibliotheken und APIs.

Was Sie sich merken sollten

- Typfehler können durch implizite Typumwandlungen verschleiert werden.
- Der Operator `+` ist überladen, sodass er je nach Typ seiner Argumente eine Addition oder eine Stringverkettung durchführt.
- Objekte werden mit `valueOf` in Zahlen und mit `toString` in Strings umgewandelt.
- Objekte mit einer `valueOf`-Methode sollten eine `toString`-Methode implementieren, die eine Stringdarstellung der von `valueOf` ausgegebenen Zahl liefert.
- Als Test auf undefinierte Werte sollten Sie `typeof` oder einen Vergleich mit `undefined` verwenden und keine Truthiness-Prüfung.

Thema 4 Verwenden Sie primitive Datentypen statt Objektwrappern

Neben Objekten kennt JavaScript noch fünf Arten von primitiven Datentypen:

- boolesche Werte
- Zahlen
- Strings
- `null`
- `undefined`

(Verwirrenderweise meldet der Operator `typeof` den Typ `null` als Objekt, obwohl er im ECMAScript-Standard als individueller Typ bezeichnet wird.) Die Standardbibliothek bietet jedoch auch Konstruk-