KAPITEL 3

Alles über Typen

Im vorigen Kapitel habe ich das Konzept der Typsysteme vorgestellt, aber nicht erklärt, wofür das Wort *Typ* in »Typsystem« eigentlich steht.

> **Typ**
>
> Ein Satz bestimmter Werte und die Dinge, die Sie damit tun können.

Hier ein paar Beispiele, damit Sie nicht zu sehr durcheinanderkommen:

- Der Typ `boolean` steht für alle booleschen Werte (davon gibt es nur zwei: `true` und `false` – wahr und falsch) und die Operationen, die Sie daran ausführen können (wie `||`, `&&` und `!`).
- Der Typ `number` steht für alle Zahlen, die daran ausführbaren Operationen (z.B. `+`, `-`, `*`, `/`, `%`, `||`, `&&` und `?`) und die daran aufrufbaren Methoden `.toFixed`, `.toPrecision`, `.toString` etc.
- Der Typ `string` steht für alle Strings (bzw. Zeichenketten) und die Operationen, die Sie daran ausführen können (wie `+`, `||` und `&&`), sowie die Methoden, die Sie daran aufrufen können, wie `.concat` und `.toUpperCase`.

Wenn etwas den Typ `T` hat, wissen Sie einerseits, dass es sich hier um ein `T` handelt. Gleichzeitig wissen Sie aber auch, *was genau Sie mit diesem* `T` *tun können* (und was nicht). Schließlich soll durch die Verwendung eines Typecheckers verhindert werden, dass Sie ungültige Dinge tun. Um das herauszufinden, sieht er sich an, welche Typen Sie verwenden und was Sie damit anstellen.

In diesem Kapitel gebe ich Ihnen einen Überblick über die in TypeScript verfügbaren Typen und vermittle Ihnen die Grundlagen der Möglichkeiten, die die einzelnen Typen bieten. In Abbildung 3-1 finden Sie eine Übersicht.

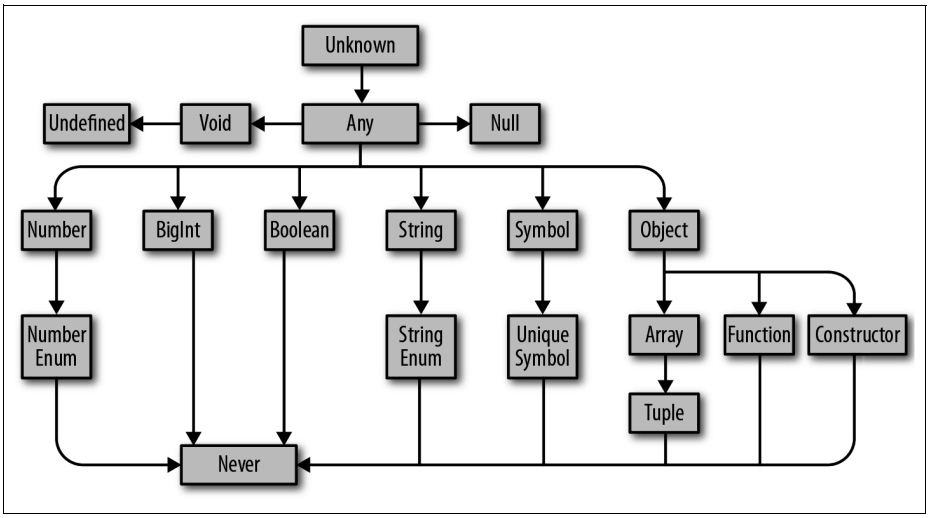

Abbildung 3-1: Die Typhierarchie von TypeScript

Wo wir gerade von Typen sprechen ...

Wenn Programmierer über Typen sprechen, benutzen sie ein präzises gemeinsames Vokabular, um ihre Bedeutung zu beschreiben. Wir werden dieses Vokabular im gesamten Buch verwenden.

Angenommen, Sie haben eine Funktion, die einen Wert übernimmt und diesen Wert mit sich selbst multipliziert zurückgibt:

```
function squareOf(n) {
  return n * n
}
squareOf(2)     // ergibt 4
squareOf('z')   // ergibt NaN (not a number, "keine Zahl")
```

Natürlich funktioniert diese Funktion nur für Zahlen. Übergeben Sie etwas anderes an `squareOf`, ist das Ergebnis ungültig. Wir müssen den Typ des Parameters also explizit *annotieren*:

```
function squareOf(n: number) {
  return n * n
}
squareOf(2)     // ergibt 4
squareOf('z')   // Error TS2345: Argument of type '"z"' is not assignable to
                // parameter of type 'number'.
```

Rufen wir `squareOf` jetzt mit etwas anderem als einer Zahl auf, merkt TypeScript das sofort und beschwert sich. Auch wenn dies ein einfaches Beispiel ist (im nächsten Kapitel gibt es eine Menge mehr zu Funktionen), reicht es aus, um eine Reihe von wichtigen TypeScript-Konzepten vorzustellen. Über das letzte Codebeispiel können wir Folgendes sagen:

1. Der Parameter `n` von `squareOf` *ist* auf den Typ `number` *beschränkt*.
2. Der Typ des Werts `2` kann dem Typ `number` *zugewiesen werden* (oder: *Er ist mit dem Typ* `number` *kompatibel*).

Ohne die Typannotation ist `squareOf` in seinen Parametern unbeschränkt, und es können Argumente beliebigen Typs übergeben werden. Sobald der Typ beschränkt ist, sorgt TypeScript dafür, dass wir die Funktion nur mit einem kompatiblen Argument aufrufen. Hier hat `2` den Typ `number`. Dies kann der Annotation `number` von `squareOf` zugewiesen werden. Also akzeptiert TypeScript unseren Code. Versuchen wir stattdessen, `'z'` zuzuweisen, beschwert sich TypeScript, weil `'z'` ein `string` ist und dem Typ `number` nicht zugewiesen werden kann.

Sie können sich das auch als eine Art der *Beschränkung* vorstellen. Wir haben TypeScript mitgeteilt, dass `number` die *Obergrenze* (*upper bound*) von `n` ist. Das heißt: Jeder an `squareOf` übergebene Wert darf höchstens den Typ `number` haben. Ist der Wert mehr als das (z.B. wenn er `number` oder `string` sein könnte), dann kann er nicht an `n` zugewiesen werden.

Im Kapitel 6 werde ich die Themen Zuweisbarkeit, Begrenzungen (bounds) und Beschränkungen (constraints) genauer erklären. Im Moment müssen Sie nur wissen, dass wir mit diesen Begriffen beschreiben, ob ein Typ an einer bestimmten Stelle verwendet werden kann und darf.

Das ABC der Typen

Sehen wir einmal, welche Typen TypeScript unterstützt, welche Werte sie enthalten und was sie damit anstellen können. Dabei kümmern wir uns auch gleich um ein paar grundsätzliche Sprachmerkmale für die Arbeit mit Typen: Typaliase, Vereinigungs-Typen (union types) und Schnittmengen-Typen (intersection types).

any

`any` ist *Der Pate* der Typen. Er tut fast alles, was Sie wollen, aber die Sache hat ihren Preis. Sie sollten `any` nur um einen Gefallen bitten, wenn Sie wirklich keine andere Wahl haben. In TypeScript muss während der Kompilierungsphase alles einen Typ haben. Dabei ist `any` der Standardtyp für Dinge, deren Typ weder Sie (der Programmierer) noch TypeScript herausfinden können. Es ist der Typ des letzten Auswegs. Daher sollten Sie ihn nach Möglichkeit vermeiden.

Aber warum? Wissen Sie noch, was ein Typ ist? (Eine Reihe von Werten und die Dinge, die Sie damit anstellen können.) `any` steht für *alle* Werte, und Sie können mit `any` *alles* anstellen. Hat ein Wert den Typ `any`, können Sie damit addieren, multiplizieren, `.pizza()` daran aufrufen (und diese mit Ananas belegen) – alles.

Mit `any` verhält sich Ihr Wert wie in normalem JavaScript. Er verhindert, dass der Typechecker seine magischen Fähigkeiten anwenden kann. Wenn Sie `any` in Ihrem

Code verwenden, ist das quasi ein Blindflug. Meiden Sie any daher wie die Pest. Verwenden Sie es nur als wirklich allerletztes Mittel.

Manchmal geht es aber nicht anders. Dann können Sie any verwenden wie hier gezeigt:

```
let a: any = 666            // any
let b: any = ['danger']     // any
let c = a + b               // any
```

Der dritte Typ sollte eigentlich einen Fehler auslösen (warum versuchen Sie, eine Zahl und ein Array miteinander zu addieren?). Das passiert aber nicht, weil Sie TypeScript gesagt haben, dass Sie zwei any-Werte miteinander addieren. Wenn TypeScript feststellt, dass ein Wert den Typ any hat (etwa weil Sie vergessen haben, einen Funktionsparameter zu annotieren oder ein nicht typisiertes JavaScript-Modul importiert haben), wird ein Compile Time Error (Fehler zur Kompilierungszeit) ausgelöst, der in Ihrem Editor rot unterstrichen dargestellt wird. Wenn Sie a und b explizit mit any annotieren (per : any), vermeiden Sie diese Ausnahme. So teilen Sie TypeScript mit, dass Sie wissen, was Sie tun.

TSC-Flag: noImplicitAny

Standardmäßig ist TypeScript tolerant und beschwert sich nicht über Werte, deren Typ als any erkannt wurde. Damit TypeScript implizite any-Werte anmahnt, müssen Sie in Ihrer *tsconfig.json*-Datei das noImplicitAny-Flag setzen.

noImplicitAny ist Teil der strict-Familie der TSC-Flags. Wenn Sie in *tsconfig.json* bereits den strict-Modus aktiviert haben (wie in unserem Beispiel für »tsconfig.json« auf Seite 12), sind Sie auf der sicheren Seite.

unknown (unbekannt)

Wenn any der Pate ist, dann ist unknown Keanu Reeves als Undercover-FBI-Agent Johnny Utah in *Point Break*: Er macht es sich entspannt zwischen den fiesen Typen (sic!) bequem. Tief im Innern ist er jedoch gesetzestreu und ist eigentlich auf der Seite der Guten. Nur selten, wenn Sie den Typ eines Werts wirklich nicht rechtzeitig herausfinden können, sollten Sie anstelle von any zu unknown greifen. Wie any steht es für einen beliebigen Typ. Allerdings verhindert TypeScript die Verwendung von unknown, es sei denn, Sie führen eine Typklärung (siehe »Typverfeinerung (refinement)« auf Seite 128) durch.

Sie können unknown-Werte vergleichen (per ==, ===, ||, && und ?), sie negieren (mit !) und den genauen Typ mit den JavaScript-Operatoren typeof und instanceof klären. Die Verwendung von unknown sieht so aus:

```
let a: unknown = 30         // unknown
let b = a === 123           // boolean (boolescher Wert)
let c = a + 10              // Error TS2571: Object is of type 'unknown'.
```

```
if (typeof a === 'number') {
  let d = a + 10            // number (Zahl)
}
```

Dieses Beispiel sollte Ihnen eine grobe Idee von der Verwendung von `unknown` geben:

1. TypeScript geht niemals davon aus, dass etwas den Typ `unknown` hat. Das müssen Sie ausdrücklich annotieren (`a`).[1]
2. Sie können Werte mit anderen Werten vergleichen, die den Typ `unknown` haben (`b`).
3. Sie können allerdings keine Aktionen ausführen, bei denen davon ausgegangen wird, dass ein `unknown`-Wert einen bestimmten Typ hat (`c`), sind dagegen nicht möglich. Sie müssen TypeScript erst beweisen, dass der Wert tatsächlich diesen Typ hat (`d`).

boolean (boolescher Wert)

Der Typ `boolean` besitzt zwei mögliche Werte: `true` und `false` (wahr und falsch). Sie können diese vergleichen (per `==`, `===`, `||`, `&&` und `?`), sie negieren (mit `!`), aber nicht viel mehr. Der Typ `boolean` wird folgendermaßen verwendet:

```
let a = true                // boolean
var b = false               // boolean
const c = true              // true
let d: boolean = true       // boolean
let e: true = true          // true
let f: true = false         // Error TS2322: Type 'false' is not assignable
                            // to type 'true'.
```

Sie können TypeScript auf verschiedene Weise mitteilen, dass etwas den Typ `boolean` hat:

1. Sie können dafür sorgen, dass TypeScript den Typ Ihres Werts als `boolean` erkennt (`a` und `b`).
2. Sie können dafür sorgen, dass TypeScript Ihren Wert als einen bestimmten booleschen Wert erkennt (`c`).
3. Sie können TypeScript explizit mitteilen, dass Ihr Wert den Typ `boolean` hat (`d`).
4. Sie können TypeScript explizit mitteilen, dass Ihr Wert ein bestimmter boolescher Wert ist (`e` und `f`).

Üblicherweise werden Sie in Ihren Programmen eine der ersten beiden Wege verwenden. Die vierte Methode sollten Sie möglichst nur verwenden, wenn Sie Ihnen zusätzliche Typsicherheit verschafft (Beispiele hierfür finden Sie an verschiedenen Stellen in diesem Buch). Der dritte Weg kommt so gut wie nie zum Einsatz.

1 Fast. Wenn `unknown` Teil eines Vereinigungstyps ist, ist das Ergebnis der Vereinigung ebenfalls `unknown`. Mehr zu Vereinigungstypen finden Sie unter »Vereinigungs- und Schnittmengentypen« auf Seite 32.

Trotzdem sind der dritte und vierte Fall besonders interessant, weil sie etwas Intuitives tun. Dieses Vorgehen wird nur von wenigen Programmiersprachen unterstützt und ist Ihnen daher vielleicht neu. Im Prinzip habe ich hier gesagt: »Hey TypeScript, die Variable `e` soll nicht einfach nur einen Wert vom Typ `boolean` enthalten, sondern den *genauen* booleschen Wert `true`.« Durch die Verwendung eines Werts als Typ habe ich die möglichen `boolean`-Werte für `e` und `f` auf einen ganz bestimmten booleschen Wert beschränkt. Dieses Merkmal bezeichnet man als *Typliterale*.

> **Typliterale**
>
> Ein Typ, der für einen bestimmten Wert steht und sonst nichts.

Im vierten Fall habe ich meine Variablen explizit mit einem Typliteral annotiert. Im zweiten Fall hat TypeScript das Typliteral für mich abgeleitet, weil ich anstelle von `let` oder `var` das Schlüsselwort `const` verwendet habe. Wird einem primitiven Datentyp ein Wert per `const` zugewiesen, weiß TypeScript, dass sich dieser Wert nicht ändern kann. Für die Typableitung (Inferenz) verwendet es den am engsten definierten Typ. Daher wird im zweiten Fall der Typ von `c` als `true` abgeleitet und nicht als `boolean`. Mehr zu den Gründen, warum TypeScript unterschiedliche Typen für `let` und `const` ableitet, finden Sie unter »Typerweiterung« auf Seite 124.

Typliterale sind ein mächtiges Sprachmerkmal und kommen mehrmals in diesem Buch vor. An vielen Stellen können Typliterale für zusätzliche Sicherheit sorgen. Sie machen TypeScript zu einer einmaligen Sprache und sind etwas, für das Ihre Java-Freunde Sie beneiden werden.

number (Zahl)

`number` bezieht sich auf alle Arten von Zahlen: Ganzzahlen (Integer), Fließkommazahlen (Floats), positive und negative Zahlen, `Infinity`, `NaN` etc. Zahlen beherrschen – na ja, eben so zahlenmäßige Dinge wie Addition (`+`), Subtraktion (`-`), Modulus (`%`) und Vergleiche (`<`). Auch hierzu ein paar Beispiele:

```
let a = 1234                // number
var b = Infinity * 0.10     // number
const c = 5678              // 5678
let d = a < b               // boolean
let e: number = 100         // number
let f: 26.218 = 26.218      // 26.218
let g: 26.218 = 10          // Error TS2322: Type '10' is not assignable
                            // to type '26.218'.
```

Wie im boolean-Beispiel gibt es vier Wege, einen Typ als number zu deklarieren:

1. Sie können TypeScript ableiten lassen, dass Ihr Wert eine Zahl (number) ist (a und b).
2. Sie können const verwenden. Dadurch geht TypeScript davon aus, dass Ihr Wert eine bestimmte Zahl ist (c).
3. Sie können TypeScript explizit mitteilen, dass Ihr Wert den Typ number hat (e).
4. Sie können TypeScript explizit mitteilen, dass Ihr Wert ein ganz bestimmter Wert vom Typ number ist (f und g).

Wie bei den Werten vom Typ boolean lassen Sie TypeScript den Typ üblicherweise ableiten (die erste Möglichkeit). In seltenen Fällen muss in Ihrem Code eine Zahl auf einen bestimmten Wert beschränkt werden (der zweite und vierte Weg). Es gibt eigentlich keinen guten Grund, den Typ von etwas explizit als number zu definieren (der dritte Weg).

Wenn Sie mit langen Zahlen arbeiten, können Sie numerische Trennzeichen verwenden, um die Lesbarkeit zu erhöhen. Numerische Trennzeichen können sowohl in der Typ- wie auch der Wert-Position verwendet werden:

```
let oneMillion = 1_000_000 // entspricht 100000
let twoMillion: 2_000_000 = 2_000_000
```

bigint

bigint ist neu in JavaScript und TypeScript: Dieser Typ ermöglicht die Arbeit mit großen ganzzahligen Werten, ohne dass es zu Rundungsfehlern kommt. number unterstützt Ganzzahlen bis zu einer Höhe von 2^{53}. bigint kann dagegen auch mit größeren Werten umgehen. Dabei steht bigint für alle großen Integerwerte. Er unterstützt Operationen wie Addition (+), Subtraktion (-), Multiplikation (*), Division (/) und Vergleiche (<). Er wird verwendet wie hier gezeigt:

```
let a = 1234n               // bigint
const b = 5678n             // 5678n
var c = a + b               // bigint
let d = a < 1235            // boolean
let e = 88.5n               // Error TS1353: A bigint literal must be an integer.
let f: bigint = 100n        // bigint
let g: 100n = 100n          // 100n
let h: bigint = 100         // Error TS2322: Type '100' is not assignable
                            // to type 'bigint'.
```

Wie bei boolean und number gibt es vier Wege, einen bigint-Wert zu deklarieren. Überlassen Sie es nach Möglichkeit TypeScript, den Typ eines bigint-Werts abzuleiten.

Beim Schreiben dieses Buchs wurde `bigint` noch nicht von allen JavaScript-Engines nativ unterstützt. Wenn Ihre Applikation `bigint` benötigt, sollten Sie auf jeden Fall überprüfen, ob es von Ihrer Zielplattform wirklich unterstützt wird.

string (String, Zeichenkette)

`string` steht für alle Strings (Zeichenketten) und das, was Sie damit anstellen können, wie das Verketten (concatenation, `+`), das Extrahieren von Teilbereichen (slicing, `.slice`) etc. Auch hierzu ein paar Beispiele:

```
let a = 'hello'             // string
var b = 'billy'             // string
const c = '!'               // '!'
let d = a + ' ' + b + c     // string
let e: string = 'zoom'      // string
let f: 'john' = 'john'      // 'john'
let g: 'john' = 'zoe'       // Error TS2322: Type "zoe" is not assignable
                            // to type "john".
```

Wie bei `boolean` und `number` gibt es vier Wege, einen `string`-Typ zu deklarieren. Dabei sollten Sie es nach Möglichkeit TypeScript überlassen, den Typ für Sie abzuleiten.

symbol (Symbole)

`symbol` ist ein relativ neues Sprachmerkmal, das erst mit den neueren Sprachrevisionen (ab ES2015) Teil von JavaScript wurde. Symbole kommen in der Praxis eher selten vor. Sie können alternativ zu stringbasierten Schlüsseln in Objekten und Maps verwendet werden. Das kann sinnvoll sein, wenn Sie sicherstellen möchten, dass Anwender den richtigen und bekannten Schlüssel verwenden und diesen nicht aus Versehen verändern. Das kann beispielsweise der Fall sein, wenn Sie einen Standard-Iterator für ein Objekt (`Symbol.iterator`) festlegen oder zur Laufzeit überschreiben, ob ihr Objekt eine Instanz von etwas (`Symbol.hasInstance`) ist.

Symbole haben den Typ `symbol`. Es kann allerdings nicht besonders viel damit gemacht werden:

```
let a = Symbol('a')         // symbol
let b: symbol = Symbol('b') // symbol
var c = a === b             // boolean
let d = a + 'x'             // Error TS2469: The '+' operator cannot be applied
                            // to type 'symbol'.
```

Verwenden Sie `Symbol('a')` in JavaScript, wird ein neues `symbol` mit dem angegebenen Namen erzeugt. Dieses `symbol` ist einmalig und kann nicht mit anderen Symbolen gleich sein (bei Vergleichen mit `==` oder `===`) – selbst dann nicht, wenn Sie ein zweites Symbol mit exakt dem gleichen Namen erzeugen! Auf ähnliche Weise, wie Zahlen je nach Definition unterschiedlich abgeleitet werden (mit `let` definiert hat

27 den Typ number, mit const definiert hat sie auch den Typ number, aber *mit einem ganz bestimmten Wert*), können auch Symbole per unique symbol explizit typisiert werden: :

```
const e = Symbol('e')                // typeof e
const f: unique symbol = Symbol('f') // typeof f
let g: unique symbol = Symbol('f')   // Error TS1332: A variable whose type is a
                                     // 'unique symbol' type must be 'const'.
let h = e === e             // boolean
let i = e === f             // Error TS2367: This condition will always return
                            // 'false' since the types 'unique symbol' and
                            // 'unique symbol' have no overlap.
```

Dieses Beispiel zeigt verschiedene Methoden, einmalige Symbole zu erzeugen:

1. Wenn Sie ein neues symbol deklarieren und es einer neuen const-Variablen zuweisen (keine let- oder var-Variable), so leitet TypeScript den Typ als unique symbol ab. Im Codeeditor wird es allerdings als typeof *ihrVariablenName* anstelle von unique symbol angezeigt.
2. Sie können den Typ einer const-Variablen explizit als unique symbol annotieren.
3. Ein unique symbol ist immer mit sich selbst identisch.
4. TypeScript weiß bei der Kompilierung, dass ein unique symbol niemals mit einem anderen unique symbol gleich sein kann.

Stellen Sie sich ein unique symbol wie andere literale Typen vor, z.B. 1, true oder "literal". Hiermit können Sie einen Typ erstellen, der für ein ganz bestimmtes symbol steht.

Objekte

Die Objekttypen in TypeScript geben die Form von Objekten an. Dabei wird nicht zwischen einfachen Objekten (die beispielsweise per {} erstellt werden) und komplizierteren (die etwa per new Blah erstellt werden) unterschieden. Das ist Absicht: JavaScript ist im Allgemeinen *strukturell typisiert*. Daher bevorzugt auch TypeScript diesen Programmierstil gegenüber einem *nominell typisierten* Stil.

Strukturelle Typisierung

Ein Programmierstil, bei dem nur darauf geachtet wird, dass ein Objekt bestimmte Eigenschaften hat, und nicht, wie sein Name lautet (nominelle Typisierung). Wird in anderen Sprachen auch als *Duck Typing* bezeichnet (oder eine Sache nicht nur anhand der Äußerlichkeiten beurteilen).

In TypeScript gibt es verschiedene Wege, Objekte mit Typen zu beschreiben. Der erste ist die Deklaration eines Werts als object:

```
let a: object = {
  b: 'x'
}
```

Was passiert beim Zugriff auf `b`?

```
a.b   // Error TS2339: Property 'b' does not exist on type 'object'.
```

Und wofür soll das gut sein? Warum sollten wir etwas als `object` typisieren, wenn wir gar nichts damit anfangen können?

Großartiger Einwand. Tatsächlich ist `object` nur ein wenig enger gefasst als `any`. `object` sagt wenig über den Wert aus, den es beschreibt – außer dass es sich um ein JavaScript-Objekt handelt (das nicht `null` ist).

Was passiert, wenn wir die explizite Annotation weglassen und TypeScript die Arbeit tun lassen?

```
let a = {
  b: 'x'
}            // {b: string}
a.b          // string

let b = {
  c: {
    d: 'f'
  }
}            // {c: {d: string}}
```

Wir haben soeben eine zweite Möglichkeit gefunden, ein Objekt zu typisieren: die objektliterale Syntax (nicht zu verwechseln mit Typliteralen). Sie können die Form des Objekts entweder von TypeScript ableiten lassen oder sie innerhalb der geschweiften Klammern ({}) explizit beschreiben:

```
let a: {b: number} = {
  b: 12
}            // {b: number}
```

Typableitung (Inferenz) bei der Deklaration von Objekten mit const

Was passiert, wenn wir das Objekt stattdessen mit `const` deklarieren?

```
const a: {b: number} = {
  b: 12
}            // Immer noch {b: number}
```

Es überrascht Sie vermutlich, dass TypeScript `b` als `number` erkennt und nicht als das Literal `12`. Schließlich haben wir gelernt, dass die Verwendung von `const` oder `let` TypeScripts Typableitung beeinflusst, wenn wir etwas als `number` oder `string` deklarieren.

Im Gegensatz zu den bisherigen primitiven Typen wie `boolean`, `number`, `bigint`, `string` und `symbol` weist die Deklaration eines Objekts mit `const` TypeScript nicht an, dessen Typ enger abzuleiten. Das liegt daran, dass JavaScript-Objekte mutabel sind. TypeScript weiß nur, dass deren Felder nach seiner Erzeugung noch veränderbar sind.

Diesen Gedanken erforschen wir genauer (inklusive der Möglichkeit einer engeren Typableitung) in »Typerweiterung« auf Seite 124.

Die objektliterale Syntax sagt im Prinzip: »Hier ist ein Ding mit dieser Form.«. Dabei kann das »Ding« ein Objektliteral oder eine Klasse sein:

```
let c: {
  firstName: string
  lastName:  string
} = {
  firstName: 'john',
  lastName:  'barrowman'
}

class Person {
  constructor(
    public firstName: string,   // public ist eine Abkürzung für
                                // this.firstName = firstName
    public lastName: string
  ) {}
}
c = new Person('matt', 'smith') // OK
```

`{firstName: string, lastName: string}` beschreibt die *Form* eines Objekts. Sowohl das Objektliteral wie auch die Klasseninstanz aus dem letzten Beispiel sind mit dieser Form kompatibel. Daher erlaubt TypeScript die Zuweisung von `Person` an `c`.

Sehen wir, was passiert, wenn wir weitere Eigenschaften hinzufügen oder erforderliche Eigenschaften weglassen:

```
let a: {b: number}

a = {}  // Error TS2741: Property 'b' is missing in type '{}'
        // but required in type '{b: number}'.

a = {
  b: 1,
  c: 2  // Error TS2322: Type '{b: number; c: number}' is not assignable
}       // to type '{b: number}'. Object literal may only specify known
        // properties, and 'c' does not exist in type '{b: number}'.
```

Definitive Zuweisung

Im ersten Beispiel haben wir eine Variable (`a`) deklariert und sie dann mit bestimmten Werten initialisiert (`{}` und `{b: 1, c: 2}`). Diese in JavaScript übliche Vorgehensweise wird auch von TypeScript unterstützt.

Wenn Sie eine Variable an einer Stelle deklarieren und erst später initialisieren, stellt TypeScript sicher, dass ihr bei ihrer Verwendung *auf jeden Fall* (definitiv) ein Wert zugewiesen ist:

```
let i: number
let j = i * 3  // Error TS2454: Variable 'i' is used
               // before being assigned.
```

Keine Sorge. TypeScript erzwingt das auch dann für Sie, wenn Sie keine explizite Typannotation verwenden:

```
let i
let j = i * 3  // Error TS2532: Object is possibly
               // 'undefined'.
```

Standardmäßig ist TypeScript bei den Objekteigenschaften ziemlich pingelig. Wenn Sie sagen, das Objekt solle eine Eigenschaft namens `b` vom Typ `number` haben, dann erwartet TypeScript tatsächlich `b` und nur `b`. Gibt es weitere Eigenschaften, wird TypeScript sich beschweren.

Können wir TypeScript denn mitteilen, dass etwas optional ist oder dass es vielleicht mehr Eigenschaften als geplant geben kann? Aber sicher:

```
let a: {
  b: number ❶
  c?: string ❷
  [key: number]: boolean ❸
}
```

❶ `a` besitzt eine Eigenschaft `b` des Typs `number`.

❷ `a` hat möglicherweise eine Eigenschaft `c` des Typs `string`. Wenn `c` einen Wert hat, ist dieser möglicherweise `undefined`.

❸ `a` kann eine beliebige Anzahl numerischer Eigenschaften vom Typ `boolean` haben.

Sehen wir einmal, welche Objekttypen wir `a` zuweisen können:

```
a = {b: 1}
a = {b: 1, c: undefined}
a = {b: 1, c: 'd'}
a = {b: 1, 10: true}
a = {b: 1, 10: true, 20: false}
a = {10: true}          // Error TS2741: Property 'b' is missing in type
                        // '{10: true}'.
a = {b: 1, 33: 'red'}   // Error TS2741: Type 'string' is not assignable
                        // to type 'boolean'.
```

Indexsignaturen

Die Schreibweise `[Schlüssel: T]: U` wird auch als *Indexsignatur* bezeichnet. Auf diese Weise teilen Sie TypeScript mit, dass ein Objekt weitere Schlüssel enthalten kann. Man könnte auch sagen: »Für dieses Objekt müssen alle Schlüssel des Typs `T` Werte vom Typ `U` haben.« Durch Indexsignaturen können Sie ein Objekt – zusätzlich zu den explizit deklarierten – auf sichere Weise um weitere Schlüssel erweitern.

Hierbei müssen Sie allerdings eine wichtige Regel beachten: Der Typ des Schlüssels der Indexsignatur (`T`) muss entweder `number` oder `string` zuweisbar sein.[2]

Für den Namen des Indexsignatur-Schlüssels können Sie ein beliebiges Wort verwenden – es muss also nicht unbedingt `"Schlüssel"` sein:

```
let airplaneSeatingAssignments: {
  [seatNumber: string]: string
} = {
  '34D': 'Boris Cherny',
  '34E': 'Bill Gates'
}
```

Bei der Deklaration von Objekttypen ist das Fragezeichen (für »optional«, `?`) nicht der einzige mögliche Modifier. Mithilfe von `readonly` können Sie Felder als schreibgeschützt kennzeichnen (d.h., der zugewiesene Startwert kann nicht verändert werden, so ähnlich wie `const` für Objekteigenschaften):

```
let user: {
  readonly firstName: string
} = {
  firstName: 'abby'
}

user.firstName // string
user.firstName =
  'abbey with an e' // Error TS2540: Cannot assign to 'firstName' because it
                    // is a read-only property.
```

Bei der objektliteralen Schreibweise gibt es einen Sonderfall: leere Objekttypen (`{}`). Jeder Typ – außer `null` und `undefined` – kann einem leeren Objekttyp zugewiesen werden, was die Verwendung etwas erschweren kann. Versuchen Sie daher, leere Objekttypen nach Möglichkeit zu vermeiden.

```
let danger: {}
danger = {}
danger = {x: 1}
danger = []
danger = 2
```

2 JavaScript-Objekte verwenden Strings als Schlüssel; Arrays sind besondere Objekte, die numerische Schlüssel verwenden.

Es gibt noch eine weitere Möglichkeit, etwas als Objekt zu typisieren: `Object`. Das ist fast das Gleiche wie {} und sollte ebenfalls möglichst vermieden werden.[3]

Hier noch einmal eine Zusammenfassung der vier Möglichkeiten, Objekte in TypeScript zu deklarieren:

1. Die objektliterale Schreibweise (wie `{a: string}`) wird auch als *Form* (shape) bezeichnet. Verwenden Sie diese Schreibweise, wenn Sie wissen, welche Felder Ihre Objekte haben könnten, oder wenn alle Werte Ihres Objekts den gleichen Typ haben.
2. Leere objektliterale Schreibweise ({}). Sollte möglichst vermieden werden.
3. Der Typ `object`. Das können Sie benutzen, wenn Sie einfach ein Objekt brauchen und es Ihnen egal ist, welche Felder es hat.
4. Der Typ `Object`. Sollte ebenfalls möglichst vermieden werden.

Verwenden Sie In Ihren TypeScript-Programmen möglichst die erste oder die dritte Option. Vermeiden Sie die zweite und vierte Form, wo es geht. Benutzen Sie einen Linter (z.B. TSLint (*https://palantir.github.io/tslint/*)), der Sie davor warnt; beschweren Sie sich in Codereviews darüber; drucken Sie ein Poster; verwenden Sie das bevorzugte Werkzeug Ihres Teams, um die Kollegen so weit wie möglich von Ihrer Codebasis fernzuhalten.

In Tabelle 3-1 finden Sie eine praktische Referenz für die Optionen 2–4 der vorigen Liste.

Tabelle 3-1: Ist der Wert ein gültiges Objekt?

Wert	{}	object	Object
{}	Ja	Ja	Ja
['a']	Ja	Ja	Ja
function () {}	Ja	Ja	Ja
new String('a')	Ja	Ja	Ja
'a'	Ja	**Nein**	Ja
1	Ja	**Nein**	Ja
Symbol('a')	Ja	**Nein**	Ja
null	**Nein**	**Nein**	**Nein**
undefined	**Nein**	**Nein**	**Nein**

3 Dabei gibt es nur einen kleinen technischen Unterschied: Mit {} können Sie beliebige Typen für die eingebauten Methoden des `Object`-Prototyps definieren, wie `.toString` und `.hasOwnProperty` (mehr Informationen zu Prototypen finden Sie im MDN (*https://mzl.la/2VSuDJz*)), während `Object` erzwingt, dass die von Ihnen deklarierten Typen denen des `Object`-Prototypen zuweisbar sind.

Dieser Code führt zum Beispiel einen Typecheck aus: `let a: {} = {toString() { return 3 }}`. Wenn Sie die Typannotation in `Object` ändern, beschwert sich TypeScript: `let b: Object = {toString() { return 3 }}` ergibt `Error TS2322: Type 'number' is not assignable to type 'string'`.

Kurze Unterbrechung: Typaliase, Vereinigungs- und Schnittmengen

Auf Ihrem Weg zu einem versierten TypeScript-Programmierer machen Sie große Fortschritte. Sie haben verschiedene Typen und ihre Funktionsweise kennengelernt, Sie kennen sich mit Konzepten der Typsysteme, Typen und Sicherheit aus. Jetzt ist es Zeit, etwas mehr in die Tiefe zu gehen.

Wie Sie wissen, können Sie an einem Wert bestimmte Operationen durchführen, je nachdem, welche ihr jeweiliger Typ erlaubt. Um zwei Zahlen zu addieren, können Sie `+` verwenden oder `.toUpperCase`, um einen String in Großbuchstaben umzuwandeln.

Aber auch an einem *Typ* können Sie verschiedene Operationen ausführen. Ich werde Ihnen jetzt einige Operationen auf Typebene vorstellen. Später im Buch kommen noch einige hinzu, aber die hier genannten kommen so häufig vor, dass ich sie so früh wie möglich einführen möchte.

Typaliase

Analog zur Verwendung von Variablendeklarationen (`let`, `const` und `var`), die auf einen Wert verweisen, können Sie ein Typalias verwenden, um auf einen bestimmten Typ zu verweisen. Das sieht zum Beispiel so aus:

```
type Age = number

type Person = {
  name: string
  age: Age
}
```

`Age` ist einfach eine Zahl (`number`). Das kann helfen, die Form einer `Person` besser zu verstehen. Aliase werden niemals von TypeScript abgeleitet. Sie müssen die Typisierung also explizit vornehmen:

```
let age: Age = 55

let driver: Person = {
  name: 'James May'
  age: age
}
```

Da `Age` nur ein Alias für `number` ist, kann es dem Typ `number` auch zugewiesen werden. Wir können den obigen Code also auch so schreiben:

```
let age = 55

let driver: Person = {
  name: 'James May'
  age: age
}
```

Wann immer Sie ein Typalias sehen, können Sie es durch den Typ, auf den es verweist, austauschen, ohne die Bedeutung Ihres Programms zu verändern.

Wie bei Variablendeklarationen in JavaScript (`let`, `const` und `var`) können Sie auch einen Typ nur einmal deklarieren:

```
type Color = 'red'
type Color = 'blue'  // Error TS2300: Duplicate identifier 'Color'.
```

Und wie `let` und `const` haben Typaliase einen blockbezogenen Geltungsbereich. Jeder Block und jede Funktion hat ihren eigenen Geltungsbereich. Dabei verdecken innere Alias-Deklarationen die äußeren:

```
type Color = 'red'

let x = Math.random() < .5

if (x) {
  type Color   = 'blue'  // dies verdeckt das oben deklarierte Color.
  let b: Color = 'blue'
} else {
  let c: Color = 'red'
}
```

Typaliase helfen, das DRY-Prinzip (»Don't Repeat Yourself«) umzusetzen und die Wiederholung komplexer Typen zu vermeiden.[4] Außerdem schaffen sie Klarheit über die Verwendung der Variablen (einige Menschen bevorzugen beschreibende Typnamen gegenüber beschreibenden Variablennamen!). Um zu entscheiden, ob Sie einen Typ mit einem Alias versehen sollten, können Sie die gleichen Überlegungen anstellen wie bei der Entscheidung, ob Sie einen Wert in einer eigenen Variablen speichern sollten.

Vereinigungs- und Schnittmengentypen

Angenommen, Sie haben zwei Dinge namens `A` und `B`, dann ist die *Vereinigungsmenge* dieser Dinge ihre Summe (alles in `A` *oder* `B` oder beidem). Die Schnittmenge ist, was beide gemeinsam haben (alles in `A` *und* `B`). Am einfachsten kann man sich das als Sets vorstellen. In Abbildung 3-2 stelle ich zwei Sets als Kreise dar. Links sehen Sie die Vereinigungsmenge oder *Summe* beider Sets, rechts sehen Sie ihre Schnittmenge oder ihr *Produkt*.

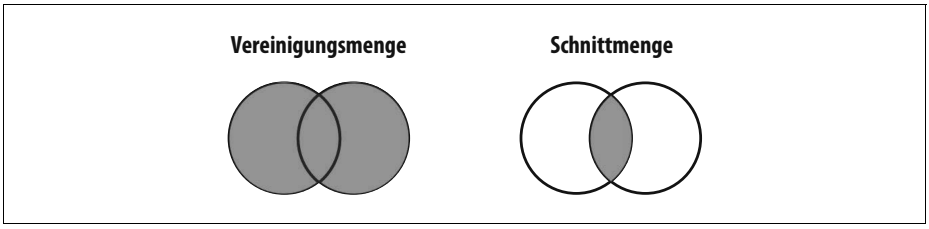

Abbildung 3-2: Vereinigungsmenge (&) und Schnittmenge (|)

4 Das Akronym DRY steht für »Don't Repeat Yourself« – »Wiederholen Sie sich nicht«. Man soll vermeiden, den gleichen Code mehrmals zu schreiben. Es wurde von Andrew Hunt und David Thomas in Ihrem Buch *The Pragmatic Programmer: From Journeyman to Master* (Addison-Wesley) eingeführt.

In TypeScript gibt es spezielle Typoperatoren, um die Vereinigungs- und Schnittmengen von Typen zu beschreiben: | für Vereinigungsmengen und & für Schnittmengen. Da Typen große Ähnlichkeit mit Sets haben, können wir sie uns auf die gleiche Weise vorstellen:

```
type Cat = {name: string, purrs: boolean}
type Dog = {name: string, barks: boolean, wags: boolean}
type CatOrDogOrBoth = Cat | Dog
type CatAndDog = Cat & Dog
```

Was wissen wir über ein Ding, das den Typ `CatOrDogOrBoth` (KatzeOderHundOderBeides) hat? Sie wissen, dass es eine `name`-Eigenschaft besitzt, die ein String ist – mehr aber auch nicht. Und was können Sie einem `CatOrDogOrBoth` zuweisen? Genau, eine Katze (`Cat`), einen Hund (`Dog`) oder beides:

```
// Katze (Cat)
let a: CatOrDogOrBoth = {
  name: 'Bonkers',
  purrs: true
}

// Hund (Dog)
a = {
  name: 'Domino',
  barks: true,
  wags: true
}

// Beides (Both)
a = {
  name: 'Donkers',
  barks: true,
  purrs: true,
  wags: true
}
```

Das kann man ruhig noch einmal sagen: Ein Wert mit einem Vereinigungs-Typ (|) ist nicht unbedingt ein bestimmtes Element Ihrer Vereinigungsmenge. Tatsächlich kann er beide Elemente auf einmal sein![5]

Was wissen wir dagegen über `CatAndDog` (KatzeUndHund)? Dieses canino-feline Superhaustier hat nicht nur einen Namen (`name`), es kann auch schnurren (`purr`), bellen (`bark`) und mit dem Schwanz wedeln (`wag`).

```
let b: CatAndDog = {
  name: 'Domino',
  barks: true,
  purrs: true,
  wags: true
}
```

5 Springen Sie zum Abschnitt »Unterscheidung von Vereinigungstypen« auf Seite 130, um zu lernen, wie Sie TypeScript mitteilen können, dass Ihre Vereinigungsmenge entkoppelt (disjoint) ist und der Typ der Vereinigungsmenge entweder der eine oder der andere, aber nicht beides sein kann.

Vereinigungs-Typen können deutlich mehr als Schnittmengen-Typen. Nehmen Sie beispielsweise diese Funktion:

```
function trueOrNull(isTrue: boolean) {
  if (isTrue) {
    return 'true'
  }
  return null
}
```

Welchen Typ hat der Rückgabewert? Es könnte ein `string` sein oder auch `null`. Wir können den Rückgabetyp folgendermaßen ausdrücken:

```
type Returns = string | null
```

Oder wie wäre es hiermit?

```
function(a: string, b: number) {
  return a || b
}
```

Ist `a` »truthy« (subjektiv wahr), so ist der Rückgabetyp `string`, ansonsten `number`. Anders gesagt: `string | number`.

Der letzte Ort, an dem Vereinigungstypen natürlicherweise auftreten, sind Arrays (besonders die heterogene Variante), um die es im folgenden Abschnitt geht.

Arrays

Wie in JavaScript sind die Arrays auch in TypeScript als bestimmte Objekte definiert, die beispielsweise Verkettung (concatenation, `+`), Hinzufügen (`push`), Durchsuchen (`find`) und das Extrahieren von Teilbereichen (`slice`) unterstützen. Zeit für ein paar Beispiele:

```
let a = [1, 2, 3]           // number[]
var b = ['a', 'b']          // string[]
let c: string[] = ['a']     // string[]
let d = [1, 'a']            // (string | number)[]
const e = [2, 'b']          // (string | number)[]

let f = ['red']
f.push('blue')
f.push(true)                // Error TS2345: Argument of type 'true' is not
                            // assignable to parameter of type 'string'.

let g = []                  // any[]
g.push(1)                   // number[]
g.push('red')               // (string | number)[]

let h: number[] = []        // number[]
h.push(1)                   // number[]
h.push('red')               // Error TS2345: Argument of type '"red"' is not
                            // assignable to parameter of type 'number'.
```

TypeScript unterstützt zwei Schreibweisen für Arrays: `T[]` und `Array<T>`. Sie sind in Bedeutung und Performance gleich. Um dieses Buch kurz zu halten, benutzen wir hier nur die `T[]`-Schreibweise. Sie sollten die Syntax verwenden, die für Ihren Programmierstil am besten passt.

Vermutlich ist Ihnen beim Lesen der Beispiele aufgefallen, dass alles außer `c` und `h` implizit typisiert ist. Und vielleicht haben Sie auch gemerkt, dass es in TypeScript Regeln gibt, die festlegen, was in einem Array gespeichert werden kann und was nicht.

Die allgemeine Regel lautet, dass Sie Arrays *homogen* halten sollten. Mischen Sie keine Äpfel mit Birnen und Zahlen im gleichen Array. Versuchen Sie, Ihre Programme so zu schreiben, dass alle Elemente im Array den gleichen Typ haben. Ansonsten ist der Aufwand deutlich größer, um TypeScript zu beweisen, dass das, was Sie tun, wirklich sicher ist.

Um zu sehen, warum homogene Arrays sich leichter handhaben lassen, sehen Sie sich das Beispiel `f` an. Hier habe ich ein Array mit dem String `'red'` initialisiert. (Bei der Deklaration enthielt das Array nur Strings. Daher geht TypeScript davon aus, dass es sich um ein Array mit `string`s handeln muss.) Danach habe ich per `push` den String `'blue'` hinzugefügt. `'blue'` ist ein String, also lässt TypeScript ihn passieren. Dann habe ich versucht, den booleschen Wert `true` hinzuzufügen. Das hat nicht funktioniert, weil `f` ein Array mit `string`s ist und `true` nun einmal kein String ist.

Bei der Initialisierung von `d` habe ich dagegen Werte vom Typ `number` und `string` verwendet. TypeScript leitet also ab, dass es sich um ein Array vom Typ `number | string` handeln muss. Da jedes Element entweder eine Zahl oder ein String sein muss, müssen Sie vor der Verwendung überprüfen, welchen Typ der aktuelle Wert gerade hat. Im folgenden Beispiel iterieren wir per `map` über das Array. Dabei sollen alle Strings in Großbuchstaben umgewandelt und alle Zahlen verdreifacht werden:

```
let d = [1, 'a']

d.map(_ => {
  if (typeof _ === 'number') {
    return _ * 3
  }
  return _.toUpperCase()
})
```

Bevor Sie eine Operation ausführen können, müssen Sie für jedes Element per `typeof` herauszufinden, ob der Typ `number` oder `string` ist.

Wie bei Objekten bewirkt die Erstellung von Arrays mit `const` für TypeScript keine engere Typableitung. Daher leitet TypeScript die Typen beider Arrays (`d` und `e`) als `number|string` ab.

`g` ist ein Sonderfall: Wenn Sie ein leeres Array initialisieren, weiß TypeScript nicht, welchen Typ die enthaltenen Elemente haben sollen, und leitet sie als `any` ab. Wenn

Sie das Array verändern, z.B. indem Sie Elemente hinzufügen, beginnt TypeScript, den Typ zu erkennen. Sobald das Array den Geltungsbereich seiner Definition verlässt (etwa wenn Sie das Array in einer Funktion definieren und es dann zurückgeben), weist TypeScript ihm seinen finalen Typ zu, der danach nicht mehr erweitert werden kann.

```
function buildArray() {
  let a = []                // any[]
  a.push(1)                 // number[]
  a.push('x')               // (string | number)[]
  return a
}

let myArray = buildArray()  // (string | number)[]
myArray.push(true)          // Error 2345: Argument of type 'true' is not
                            // assignable to parameter of type 'string | number'.
```

Da die Verwendungsmöglichkeiten von `any` recht beschränkt sind, sollten Sie sich darüber aber nicht zu viele Sorgen machen.

Tupel

Tupel sind Subtypen von `array`. Sie sind eine Sonderform der Typisierung von Arrays, da sie eine feste Länge besitzen, bei der die Werte an jedem Index spezifische und bekannte Typen haben. Im Gegensatz zu den meisten anderen Typen müssen Tupel bei ihrer Deklaration explizit typisiert werden. Der Grund ist, dass die JavaScript-Syntax für Tupel und Arrays gleich ist (beide verwenden eckige Klammern) und TypeScript bereits Regeln besitzt, mit denen Array-Typen aus eckigen Klammern abgeleitet werden können:

```
let a: [number] = [1]

// Ein Tupel aus: [vorname, nachname, geburtsjahr]
let b: [string, string, number] = ['malcolm', 'gladwell', 1963]

b = ['queen', 'elizabeth', 'ii', 1926]  // Error TS2322: Type 'string' is not
                                        // assignable to type 'number'.
```

Tupel unterstützen auch optionale Elemente. Wie bei den Objekttypen steht `?` für »optional«:

```
// Ein Array mit Fahrkartenpreisen, die nach Strecke unterschiedlich sind.
let trainFares: [number, number?][] = [
  [3.75],
  [8.25, 7.70],
  [10.50]
]

// Oder auch:
let moreTrainFares: ([number] | [number, number])[] = [
  // ...
]
```

Tupel unterstützen auch Restelemente, die Sie verwenden können, um Tupel mit minimalen Längen zu typisieren:

```
// Eine Liste mit Strings,
// die mindestens ein Element enthalten muss:
let friends: [string, ...string[]] = ['Sara', 'Tali', 'Chloe', 'Claire']

// Eine heterogene Liste:
let list: [number, boolean, ...string[]] = [1, false, 'a', 'b', 'c']
```

Tupel können nicht nur heterogene Listen auf sichere Weise codieren, sondern auch die Länge der durch sie typisierten Liste festhalten. Diese Möglichkeiten bieten deutlich mehr Sicherheit als einfache Arrays. Verwenden Sie sie oft.

Immutable (schreibgeschützte) Arrays und Tupel

Üblicherweise sind Arrays meist mutabel (d.h., Sie können beispielsweise `.push` oder `.splice` darauf anwenden und sie direkt aktualisieren). Manchmal benötigen Sie aber ein schreibgeschütztes (immutables) Array. Hierfür können Sie das Original-Array verändern, indem Sie daraus ein neues Array erzeugen, während das Original unverändert bleibt.

Für die Erstellung immutabler Arrays gibt es in TypeScript standardmäßig einen mit `readonly` gekennzeichneten Arraytyp. Schreibgeschützte Arrays verhalten sich wie normale Arrays – nur können sie nicht direkt aktualisiert werden. Um ein immutables Array zu erstellen, wird eine explizite Typannotation gebraucht. Um es zu verändern, müssen Sie anstelle von `.push` und `.splice` Methoden verwenden, die keine Änderungen am Original ausführen, wie `.concat` und `.slice`.

```
let as: readonly number[] = [1, 2, 3]     // number[] (Nur Lesen)
let bs: readonly number[] = as.concat(4)  // number[] (Nur Lesen)
let three = bs[2]                         // number
as[4] = 5            // Error TS2542: Index signature in type
                     // 'readonly number[]' only permits reading.
as.push(6)           // Error TS2339: Property 'push' does not
                     // exist on type 'readonly number[]'.
```

Wie `Array` verfügt TypeScript über verschiedene ausführlichere Schreibweisen, um immutable Arrays und Tupel zu deklarieren:

```
type A = readonly string[]          // string[] (schreibgeschützt)
type B = ReadonlyArray<string>      // string[] (schreibgeschützt)
type C = Readonly<string[]>         // string[] (schreibgeschützt)

type D = readonly [number, string]   // [number, string] (schreibgeschützt)
type E = Readonly<[number, string]>  // [number, string] (schreibgeschützt)
```

Welche Syntax Sie benutzen – das kürzere `readonly` oder eine der längeren Formen per `Readonly` oder `ReadonlyArray` –, ist letztlich eine Frage des persönlichen Geschmacks.

Auch wenn immutable Arrays Ihren Code in manchen Fällen besser nachvollziehbar machen, indem die Mutabilität vermieden wird, basieren sie auch weiterhin auf

einfachen JavaScript-Arrays. Selbst für kleine Aktualisierungen eines Arrays muss also zunächst eine Kopie angelegt werden. Wenn Sie nicht aufpassen, kann das zu Leistungseinbußen während der Laufzeit führen. Bei kleinen Arrays ist dies kaum wahrnehmbar. Sobald die Arrays größer werden, kann dieser Overhead starke Auswirkungen haben.

Wenn Sie immutable Arrays häufiger einsetzen wollen, sollten Sie möglicherweise auf eine effizientere Implementierung zurückgreifen, z.B. Lee Byrons ausgezeichnetes `immutable` (*https://www.npmjs.com/package/immutable*).

null, undefined, void und never

In JavaScript gibt es zwei Werte, mit denen das Fehlen von etwas ausdrückt werden kann: `null` und `undefined`. TypeScript unterstützt beide Werte und stellt hierfür eigene Typen zur Verfügung. Wie heißen die wohl? Richtig: ebenfalls `null` und `undefined`.

Beide sind in TypeScript spezielle Typen, denn nur der Wert `undefined` kann auch den Typ `undefined` haben, und nur der Wert `null` kann den Typ `null` haben.

JavaScript-Programmierer benutzen beide Werte meist analog zueinander. Dabei gibt es subtile semantische Unterschiede: `undefined` bedeutet, dass etwas *noch nicht definiert* wurde, während `null` bedeutet, dass *ein Wert fehlt* (z.B. wenn Sie versuchen, einen Wert zu berechnen, dabei aber ein Fehler auftrat). Dies sind nur Konventionen, und TypeScript zwingt Sie nicht, sich daran zu halten. Dennoch kann es nützlich sein, den Unterschied zu kennen.

Zusätzlich zu `null` und `undefined` gibt es in TypeScript außerdem die Typen `void` und `never`. Sie sind sehr spezifisch und sollen noch feiner zwischen nicht existierenden Dingen unterscheiden: `void` ist der Rückgabetyp einer Funktion, die nicht explizit etwas zurückgibt (wie `console.log`), während der Typ `never` der Rückgabetyp einer Funktion ist, die gar nichts zurückgibt (etwa eine Funktion, die eine Ausnahme auslöst oder eine Endlosschleife enthält):

```
// (a) Eine Funktion, die eine Zahl oder null zurückgibt:
function a(x: number) {
  if (x < 10) {
    return x
  }
  return null
}

// (b) Eine Funktion, die undefined zurückgibt:
function b() {
  return undefined
}

// (c) Eine Funktion, die void zurückgibt:
function c() {
```

```
  let a = 2 + 2
  let b = a * a
}

// (d) Eine Funktion, die nie "zurückkehrt":
function d() {
  throw TypeError('I always error')
}

// (e) Noch eine Funktion, die nie "zurückkehrt":
function e() {
  while (true) {
    doSomething()
  }
}
```

(a) und (b) geben explizit `null` bzw. `undefined` zurück. (c) gibt `undefined` zurück, allerdings ohne eine explizite `return`-Anweisung. (d) löst eine Ausnahme aus, und (e) enthält eine Endlosschleife. Daher haben (d) und (e) den Rückgabetyp `never`.

Wenn `unknown` der Supertyp aller anderen Typen ist, dann ist `never` der Subtyp aller anderen Typen. Wir sprechen hier auch vom sogenannten *bottom type*. Das heißt, er kann jedem anderen Typ zugewiesen werden. Ein Wert des Typs `never` kann überall sicher benutzt werden. Das hat zwar fast nur theoretische Bedeutung[6], kann aber nützlich sein, wenn Sie mit anderen Computersprachen-Nerds über TypeScript sprechen.

Tabelle 3-2 enthält eine Zusammenfassung der Typen, die das Fehlen von etwas bedeuten, und ihrer Verwendung.

Tabelle 3-2: Typen, die das Fehlen von etwas bezeichnen

Typ	Bedeutung
`null`	Fehlen eines Werts
`undefined`	Variable, der noch kein Wert zugewiesen wurde
`void`	Funktion ohne `return`-Anweisung
`never`	Funktion, die nie zurückkehrt

Strikte Überprüfung auf Nullwerte

In älteren TypeScript-Versionen (oder wenn die TSC-Option `strictNullChecks` auf `false` eingestellt ist) verhält sich `null` ein wenig anders: Hier ist `null` ein Subtyp aller Typen außer `never`. Das heißt, jeder Typ ist nullwertfähig und Sie können keinem Typ einfach so vertrauen, ohne zuerst zu testen, ob er `null` ist oder nicht. Angenom-

6 Den *bottom type* kann man sich als Typ ohne Werte vorstellen. Ein bottom type entspricht einer mathematischen Aussage, die immer falsch ist.

men, jemand übergibt die Variable `pizza` an ihre Funktion und Sie wollen daran die Methode `.addAnchovies` aufrufen. Dann müssen Sie zuerst überprüfen, ob `pizza` eventuell `null` ist, bevor Sie sie mit leckeren kleinen Fischen belegen. In der Praxis kann diese Überprüfung jeder einzelnen Variablen ziemlich mühsam sein und wird daher oft »vergessen«. Wenn dann tatsächlich etwas `null` ist, wird zur Laufzeit die äußerst unangenehme *Null Pointer Exception* ausgelöst:

```
function addDeliciousFish(pizza: Pizza) {
  return pizza.addAnchovies()  // Uncaught TypeError: Cannot read
}                              // property 'addAnchovies' of null

// TypeScript lets this fly with strictNullChecks = false
addDeliciousFish(null)
```

Der Mensch, der `null` in den 1960er-Jahren eingeführt hat, nannte ihn den »Milliarden-Dollar-Fehler« (*http://bit.ly/2WEdZNO*). Das Problem mit `null` besteht darin, dass die Typsysteme der meisten Sprachen ihn nicht ausdrücken oder darauf testen können. Versucht ein Programmierer also, mit einer Variablen zu arbeiten, die er für definiert hält, die aber zur Laufzeit `null` ist, löst der Code einen Laufzeitfehler aus!

Fragen Sie mich nicht, warum das so ist. Ich schreibe bloß dieses Buch. Aber die Sprachen kommen langsam darauf, `null` in ihren Typsystemen zu codieren, und TypeScript ist ein Beispiel dafür, wie es richtig gemacht wird. Wenn Sie möglichst viele Bugs schon bei der Kompilierung finden wollen (also bevor der Benutzer sie sieht), dann ist die Überprüfung auf `null` in einem Typsystem unverzichtbar.

Enums

Enums bieten die Möglichkeit, die möglichen Werte eines Typs *aufzuzählen*. Dabei handelt es sich um eine ungeordnete Datenstruktur, die Schlüssel auf Werte abbildet. Sie können sich das vorstellen wie Objekte, deren Schlüssel bei der Kompilierung feststehen. Dadurch kann TypeScript überprüfen, ob ein bestimmter Schlüssel tatsächlich existiert, wenn Sie darauf zugreifen.

Es gibt zwei Arten von Enums: Enums, die Strings auf Strings abbilden, und Enums, die Strings auf Zahlen abbilden. Sie sehen aus wie folgt:

```
enum Language {
  English,
  Spanish,
  Russian
}
```

Per Konvention wird der erste Buchstabe eines Enum-Namens großgeschrieben und steht im Singular. Enum-Schlüssel beginnen ebenfalls mit einem Großbuchstaben.

Sind die Werte der einzelnen Member eines Enums Zahlen, so leitet TypeScript diese automatisch als `number` ab. Die implizite Ableitung im vorigen Beispiel können Sie aber auch, wie im folgenden Beispiel gezeigt, explizit angeben:

```
enum Language {
  English = 0,
  Spanish = 1,
  Russian = 2
}
```

Um einen Wert aus einem Enum auszulesen, können Sie entweder per Punkt-Schreibweise oder über die Verwendung eckiger Klammern darauf zugreifen, so, als würden Sie einen Wert aus einem normalen Objekt auslesen:

```
let myFirstLanguage = Language.Russian      // Language
let mySecondLanguage = Language['English']  // Language
```

Sie können ein Enum auf mehrere Deklarationen verteilen, die TypeScript automatisch für Sie zusammenfügt (mehr hierzu finden Sie unter »Deklarationsverschmelzung (declaration merging)« auf Seite 229). Beachten Sie: Wenn Sie mehrere Enum-Deklarationen verwenden, kann TypeScript nur die Werte für eine dieser Deklarationen ableiten. Daher ist es sinnvoll, jedem Enum-Member einen Wert zuzuweisen:

```
enum Language {
  English = 0,
  Spanish = 1
}

enum Language {
  Russian = 2
}
```

Sie können auch berechnete Werte verwenden, ohne sie alle definieren zu müssen (TypeScript tut sein Bestes, um die fehlenden Dinge abzuleiten):

```
enum Language {
  English = 100,
  Spanish = 200 + 300,
  Russian                // TypeScript leitet 501 ab (die nächste Zahl nach 500)
}
```

Sie können für Enums auch String-Werte verwenden oder sogar String- und Zahlenwerte miteinander mischen:

```
enum Color {
  Red = '#c10000',
  Blue = '#007ac1',
  Pink = 0xc10050,        // Ein hexadezimales Literal
  White = 255             // Ein dezimales Literal
}

let red = Color.Red       // Color
let pink = Color.Pink     // Color
```

In TypeScript können Sie sowohl über den Schlüssel wie auch anhand des Werts auf Enums zugreifen. Allerdings wird das schnell unsicher:

```
let a = Color.Red         // Color
let b = Color.Green       // Error TS2339: Property 'Green' does not exist
                          // on type 'typeof Color'.
let c = Color[0]          // string
let d = Color[6]          // string (!!!)
```

Eigentlich sollte es nicht möglich sein, auf `Color[6]` zuzugreifen, aber TypeScript hält Sie nicht davon ab! Wir können TypeScript anweisen, diese Art unsicherer Zugriffe zu verhindern, indem wir eine sichere Untermenge von `enum` verwenden: `const enum`. Das `Language`-Enum von vorhin sähe dann so aus:

```
const enum Language {
  English,
  Spanish,
  Russian
}

// Zugriff auf einen gültigen Enum-Schlüssel:
let a = Language.English  // Language

// Zugriff auf einen ungültigen Enum-Schlüssel:
let b = Language.Tagalog  // Error TS2339: Property 'Tagalog' does not exist
                          // on type 'typeof Language'.

// Zugriff auf einen gültigen Enum-Schlüssel:
let c = Language[0]       // Error TS2476: A const enum member can only be
                          // accessed using a string literal.

// Zugriff auf einen ungültigen Enum-Schlüssel:
let d = Language[6]       // Error TS2476: A const enum member can only be
                          // accessed using a string literal.
```

Bei der Verwendung von `const enum` ist der Zugriff über den Wert nicht möglich, und das Enum verhält sich fast wie ein reguläres JavaScript-Objekt. Standardmäßig wird auch kein JavaScript-Code erzeugt. Stattdessen wird der Wert des Enums an den Stellen eingefügt, wo er gebraucht wird (d.h., TypeScript ersetzt jedes Vorkommen von `Language.Spanish` mit dessen Wert `1`).

TSC-Flag: preserveConstEnums

Das Einbetten von `const enum` kann zu Sicherheitsproblemen führen, wenn Sie ein `const enum` aus Fremdcode importieren: Aktualisiert der Autor des Enums seinen TypeScript-Code, kann es passieren, dass Ihre Version und die des anderen Autors zur Laufzeit auf verschiedene Werte zeigen, und das macht TypeScript auch nicht schlauer.

Daher sollten Sie bei der Verwendung von `const enum` vorsichtig sein und ihre Einbettung auf TypeScript-Programme unter Ihrer Kontrolle beschränken. Vermeiden Sie die Verwendung in Programmen, die über NPM veröffentlicht oder anderen als Bibliothek zugänglich gemacht werden.

Um die Codeerzeugung zur Laufzeit für `const enum` zu aktivieren, müssen Sie den Wert der TSC-Einstellung `preserveConstEnums` in Ihrer *tsconfig.json*-Datei auf `true` setzen:

```
{
  "compilerOptions": {
    "preserveConstEnums": true
  }
}
```

Hier noch ein Beispiel für die Verwendung von `const enum`:

```
const enum Flippable {
  Burger,
  Chair,
  Cup,
  Skateboard,
  Table
}

function flip(f: Flippable) {
  return 'flipped it'
}

flip(Flippable.Chair)     // 'flipped it'
flip(Flippable.Cup)       // 'flipped it'
flip(12)                  // 'flipped it' (!!!)
```

Alles klappt wunderbar. Das Umdrehen (engl. *to flip*) von Stühlen (`Chair`s) und Tassen (`Cup`s) funktioniert wie erwartet – bis Sie feststellen, dass alle Zahlen auch Enums zugewiesen werden können! Dieses Verhalten ist eine unglückliche Folge von TypeScripts Zuweisungsregeln (mehr dazu in Kapitel 6). Um das Problem zu lösen, müssen Sie besonders darauf achten, die Werte auf Strings zu beschränken:

```
const enum Flippable {
  Burger = 'Burger',
  Chair = 'Chair',
  Cup = 'Cup',
  Skateboard = 'Skateboard',
  Table = 'Table'
}

function flip(f: Flippable) {
  return 'flipped it'
}

flip(Flippable.Chair)     // 'flipped it'
flip(Flippable.Cup)       // 'flipped it'
flip(12)                  // Error TS2345: Argument of type '12' is not
                          // assignable to parameter of type 'Flippable'.
flip('Hat')               // Error TS2345: Argument of type '"Hat"' is not
                          // assignable to parameter of type 'Flippable'.
```

Schon ein mieser kleiner numerischer Wert kann Ihr gesamtes Enum unsicher machen.

Da es so viele Fallstricke bei der sicheren Verwendung von Enums gibt, rate ich von ihrer Verwendung ab. Es gibt zahlreiche bessere Möglichkeiten, sich in TypeScript auszudrücken.

Falls trotzdem ein Mitarbeiter auf die Verwendung von Enums besteht und Sie dessen Meinung absolut nicht ändern können, hilft es, heimlich ein paar TSLint-Regeln zu erstellen, die Sie vor nichtnumerischen Werten und Enums ohne `const` warnen.

Zusammenfassung

Einfach gesagt, besitzt TypeScript eine Reihe eingebauter Typen. TypeScript kann Typen für Sie anhand ihrer Werte ableiten (Inferenz), oder Sie können Ihre Werte explizit typisieren. Durch die Verwendung von `const` können spezifischere Typen abgeleitet werden. Bei `let` und `var` ist die Ableitung allgemeiner. Die meisten Typen haben allgemeinere und spezifischere Gegenstücke, wobei letztere Subtypen der ersteren sind (siehe Tabelle 3-3).

Tabelle 3-3: Typen und ihre spezifischeren Subtypen

Typ	Subtyp
`boolean`	boolesches Literal
`bigint`	BigInt-Literal
`number`	literale Zahl
`string`	literaler String
`symbol`	`unique symbol`
`object`	Objektliteral
Array	Tuple
`enum`	`const enum`

Übungen

1. Welchen Typ leitet TypeScript für diese Werte ab?
    a. `let a = 1042`
    b. `let b = 'apples and oranges'`
    c. `const c = 'pineapples'`
    d. `let d = [true, true, false]`
    e. `let e = {type: 'ficus'}`
    f. `let f = [1, false]`
    g. `const g = [3]`

h. `let h = null` (probieren Sie diese Aufgabe in Ihrem Codeeditor aus. Wenn Sie das Ergebnis überrascht, werfen Sie einen Blick auf »Typerweiterung« auf Seite 124!)

2. Warum werden hier die gezeigten Fehler ausgelöst?

a.
```
let i: 3 = 3
i = 4 // Error TS2322: Type '4' is not assignable to type '3'.
```

b.
```
let j = [1, 2, 3]
j.push(4)
j.push('5') // Error TS2345: Argument of type '"5"' is not
            // assignable to parameter of type 'number'.
```

c.
```
let k: never = 4 // Error TSTS2322: Type '4' is not assignable
                 // to type 'never'.
```

d.
```
let l: unknown = 4
let m = l * 2 // Error TS2571: Object is of type 'unknown'.
```