TypeScript - Ein praktischer Einstieg

Typsicheres JavaScript für skalierbare Webanwendungen

KAPITEL 2

Das Typsystem

Die Mächtigkeit von JavaScript
Entspringt der Flexibilität
Lass Vorsicht walten!

In Kapitel 1, »Von JavaScript zu TypeScript«, habe ich kurz von der Existenz eines sogenannten »Typecheckers« in TypeScript gesprochen, der sich Ihren Code ansieht, dessen beabsichtigte Funktion versteht und Ihnen mitteilt, wo Sie möglicherweise Fehler gemacht haben. Aber wie funktioniert ein Typechecker eigentlich?

Was ist ein Typ?

Ein »Typ« beschreibt die *Form* eines JavaScript-Werts. Mit »Form« meine ich die Eigenschaften und Methoden, die ein Wert hat (man sagt auch: die auf einem Wert existieren), und wie der eingebaute Operator `typeof` diesen Wert charakterisieren würde.

Wenn Sie zum Beispiel eine Variable mit dem Anfangswert `"Aretha"` erstellen ...

```
let singer = "Aretha";
```

... kann TypeScript ableiten bzw. herausfinden, dass die Variable `singer` vom *Typ* `string` ist.

Die grundlegenden Typen in TypeScript entsprechen den sieben grundlegenden primitiven Datenypen in JavaScript:

- `null`
- `undefined`
- `boolean` // `true` or `false`
- `string` // `""`, `"Hi!"`, `"abc123"`, ...
- `number` // `0`, `2.1`, `-4`, ...
- `bigint` // `0n`, `2n`, `-4n`, ...
- `symbol` // `Symbol()`, `Symbol("hi")`, ...

Die folgenden Werte erkennt TypeScript als jeweils einen der sieben primitiven Datentypen:

- `null; // null`
- `undefined; // undefined`
- `true; // boolean`
- `"Louise"; // string`
- `1337; // number`
- `1337n; // bigint`
- `Symbol("Franklin"); // symbol`

Falls Sie einmal den Namen eines primitiven Datentyps vergessen sollten, können Sie im TypeScript Playground (*https://typescriptlang.org/play*) oder in einer IDE mithilfe einer `let`-Anweisung einer Variablen einen Primitivwert zuordnen und dann den Mauszeiger über den Namen der Variablen bewegen. Das sich öffnende Popover zeigt den Namen des primitiven Datentyps, wie im folgenden Screenshot (Abbildung 2-1).

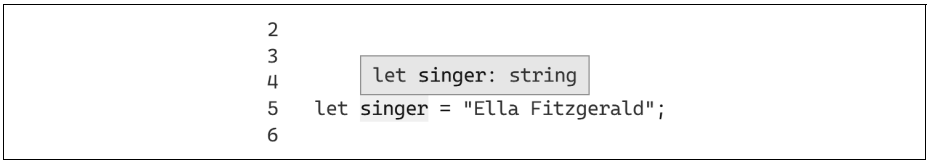

Abbildung 2-1: TypeScript zeigt den Typ einer String-Variablen in einem Popover.

TypeScript ist auch schlau genug, um den Typ einer Variablen mit einem berechneten Anfangswert zu ermitteln. Im folgenden Beispiel erkennt TypeScript, dass der ternäre Ausdruck immer eine Zeichenkette ergibt, die Variable `bestSong` also ein `string` ist:

```
// Abgeleiteter Typ: string
let bestSong = Math.random() > 0.5
  ? "Chain of Fools"
  : "Respect";
```

Wenn Sie im TypeScript Playground (*https://typescriptlang.org/play*) oder in Ihrer IDE mit der Maus über der Variablen `bestSong` hovern, sollten Sie eine Infobox oder eine Meldung sehen, die Ihnen mitteilt, dass TypeScript die Variable `bestSong` auf den Typ `string` zurückgeführt hat (Abbildung 2-2).

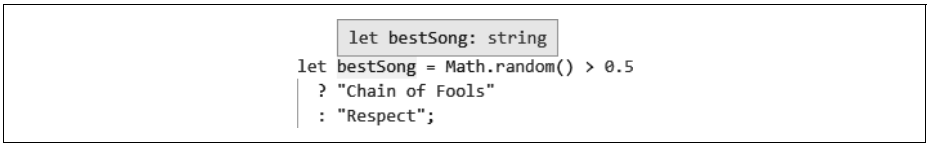

Abbildung 2-2: TypeScript meldet aufgrund einer Ableitung aus dem ternären Ausdruck, dass eine let-Variable den Literaltyp string hat.

Denken Sie an die Unterschiede zwischen Objekten und primitiven Datentypen in JavaScript: Klassen wie `Boolean` oder `Number` umhüllen ihre primitiven Entsprechungen. In TypeScript ist es allgemein üblich, sich auf die kleingeschriebenen Namen zu beziehen, also z. B. `boolean` oder `number`.

Typsysteme

Ein *Typsystem* ist ein Satz von Regeln, der für eine Programmiersprache festlegt, welche Typen die Konstrukte in einem Programm haben dürfen.

Im Kern funktioniert das Typsystem von TypeScript folgendermaßen:

- Es liest Ihren Code und versteht alle vorhandenen Typen und Werte.
- Es erkennt für jeden Wert, welchen Typ dieser laut seiner ursprünglichen Deklaration haben darf.
- Es überprüft die spätere Verwendung aller Werte im Code.
- Es gibt eine Meldung aus, wenn die Verwendung eines Werts nicht mit seinem Typ übereinstimmt.

Lassen Sie uns diesen Prozess der Typinferenz im Detail durchgehen.

Betrachten Sie den folgenden Codeausschnitt, in dem TypeScript einen Typfehler bezüglich einer Member-Eigenschaft ausgibt, die fälschlicherweise als Funktion aufgerufen wird:

```
let firstName = "Whitney";
firstName.length();
//        ~~~~~~
//  This expression is not callable.
//    Type 'Number' has no call signatures
```

TypeScript ist aufgrund folgender Schritte zu dieser Beschwerde gekommen:

1. Es hat den Code gelesen und verstanden, dass es eine Variable mit dem Namen `firstName` gibt.
2. Es hat aus der Tatsache, dass der Anfangswert von `firstName` ein `String` – `"Whitney"` – ist, die Schlussfolgerung gezogen, dass die Variable vom Typ `string` ist.
3. Es hat gesehen, dass im Code versucht wird, auf das Element `.length` von `firstName` zuzugreifen und es wie eine Funktion aufzurufen.
4. Es hat darauf hingewiesen, dass das Element `.length` einer Zeichenkette eine Zahl und keine Funktion ist *(es kann nicht wie eine Funktion aufgerufen werden)*.

Will man TypeScript-Code verstehen, muss man das Typsystem von TypeScript verstehen. Die Codebeispiele in diesem Kapitel und im Rest des Buchs stellen zunehmend komplexere Typen vor, die TypeScript aus dem Code ableiten kann.

Fehlerarten

Beim Schreiben von TypeScript werden Sie meist auf zwei Arten von »Fehlern« stoßen:

Syntaxfehler
: Diese Fehler blockieren die Umwandlung von TypeScript in JavaScript.

Typfehler
: Der Typechecker hat eine Unstimmigkeit festgestellt.

Die Unterschiede zwischen beiden Fehlerarten sind wichtig.

Syntaxfehler

Syntaxfehler treten auf, wenn TypeScript auf ungültige Anweisungen stößt, die es nicht als Code interpretieren kann. Diese Fehler verhindern, dass TypeScript aus Ihrer Datei korrektes JavaScript generieren kann. Abhängig von den Werkzeugen und Einstellungen, die Sie für die Konvertierung Ihres TypeScript-Codes in JavaScript verwenden, erhalten Sie möglicherweise immer noch irgendeine Art von JavaScript-Ausgabe (bei den Standardeinstellungen von `tsc` ist dies der Fall). Entsprechender Output wird aber wahrscheinlich nicht so aussehen, wie Sie es erwarten.

Die folgende TypeScript-Zeile enthält einen Syntaxfehler aufgrund einer unerwarteten `let`-Anweisung:

```
let let wat;
//      ~~~
// Error: ',' expected.
```

Die kompilierte JavaScript-Ausgabe kann, je nach Version des TypeScript-Compilers, etwa so aussehen:

```
let let, wat;
```

Obwohl TypeScript sein Bestes tut, um JavaScript-Code ungeachtet von Syntaxfehlern auszugeben, wird der resultierende Code wahrscheinlich nicht Ihren Vorstellungen entsprechen. Am besten beheben Sie Syntaxfehler, bevor Sie versuchen, den resultierenden JavaScript-Code auszuführen.

Typfehler

Typfehler treten auf, wenn zwar Ihre Syntax gültig ist, aber bei der Typprüfung ein Fehler festgestellt wurde. Diese Fehler verhindern zwar nicht, dass TypeScript-Syntax in JavaScript umgewandelt wird. Sie deuten jedoch oft darauf hin, dass etwas abstürzen oder sich unerwartet verhalten könnte, wenn Ihr Code ausgeführt wird.

Ich habe Ihnen in Kapitel 1, »Von JavaScript zu TypeScript«, das `console.blub`-Beispiel gezeigt, bei dem der Code zwar syntaktisch gültig war, TypeScript aber erkennen konnte, dass er bei der Ausführung wahrscheinlich abstürzen würde:

```
console.blub("Nothing is worth more than laughter.");
//      ~~~~
// Error: Property 'blub' does not exist on type 'Console'.
```

Auch wenn TypeScript trotz des Vorhandenseins von Typfehlern JavaScript-Code ausgeben kann, sind diese Fehler im Allgemeinen ein Zeichen dafür, dass das ausgegebene JavaScript wahrscheinlich nicht in der beabsichtigten Weise ausgeführt werden kann. Am besten beheben Sie die gemeldeten Probleme, bevor Sie den Code laufen lassen.

Einige Projekte sind so konfiguriert, dass die Ausführung von Code während der Entwicklung so lange blockiert wird, bis alle TypeScript-Typfehler (nicht nur Syntaxfehler) behoben sind. Viele Entwickler, mich eingeschlossen, empfinden das im Allgemeinen als lästig und unnötig. Die meisten Projekte kann man z. B. mithilfe der Datei *tsconfig.json* und den Konfigurationsoptionen, die in Kapitel 13, »Konfigurationsoptionen«, behandelt werden, problemlos so einrichten, dass dies nicht geschieht.

Zuweisbarkeit

TypeScript liest die Anfangswerte von Variablen, um zu bestimmen, welchen Typ diese Variablen haben dürfen. Wenn Variablen später ein neuer Wert zugewiesen wird, wird geprüft, ob der Typ des neuen Werts mit dem der Variablen übereinstimmt.

In TypeScript ist es kein Problem, einer Variablen später einen anderen Wert desselben Typs zuzuweisen. Wenn eine Variable z. B. zunächst einen `string`-Wert hat, ist es vollkommen in Ordnung, ihr später einen anderen `string` zuzuweisen:

```
let firstName = "Carole";
firstName = "Joan";
```

Stößt TypeScript aber auf die Zuweisung eines anderen Typs, gibt es einen Typfehler aus. Wir könnten zum Beispiel eine Variable nicht zunächst mit einem `string`-Wert deklarieren und ihr dann später einen `boolean`-Wert zuweisen:

```
let lastName = "King";
lastName = true;
// Error: Type 'boolean' is not assignable to type 'string'.
```

Wird ein Wert an einen Funktionsaufruf oder eine Variable übergeben, prüft TypeScript dessen *Zuweisbarkeit* (assignability). Nur wenn der Typ dieses Werts mit dem erwarteten Typ übereinstimmt, ist er *zuweisbar*. Die Zuweisbarkeit wird in späteren Kapiteln eine wichtige Rolle spielen, wenn wir komplexere Objekte vergleichen.

Zuweisbarkeitsfehler verstehen

Fehlern des Musters »Type ... is not assignable to type ...« (»Typ ... ist nicht zuweisbar an Typ ...«) werden Sie beim Schreiben von TypeScript-Code mit am häufigsten begegnen.

Der erste Typ, der in dieser Fehlermeldung erwähnt wird, ist der Wert, der im Code einem Empfänger zugewiesen werden soll. Der zweite erwähnte Typ ist der Empfänger, dem der erste Typ zugewiesen werden soll. Im vorherigen Codebeispiel haben wir mit `lastName = true` versucht, den Wert von `true` (Typ `boolean`) der Empfängervariablen `lastName` (Typ `string`) zuzuweisen.

Im Laufe dieses Buchs werden Sie immer komplexere Zuweisungsprobleme kennenlernen. Achten Sie sorgfältig darauf, die gemeldeten Unterschiede zwischen tatsächlichen und erwarteten Typen zu verstehen. Dadurch wird es viel einfacher, mit TypeScript zu arbeiten, wenn Typfehler auftreten.

Typanmerkungen

Manchmal besitzt eine Variable keinen Anfangswert, den TypeScript lesen kann. TypeScript versucht in diesem Fall aber nicht, den Anfangstyp der Variablen aus späteren Verwendungen abzuleiten. Stattdessen wird die Variable standardmäßig als implizit vom Typ `any` betrachtet, was bedeutet, dass sie jeden beliebigen Wert annehmen kann.

Variablen, deren anfänglicher Typ nicht abgeleitet werden kann, durchlaufen einen Prozess, der als *evolving* `any` (sich entwickelndes `any`) bezeichnet wird: Anstatt einen ganz bestimmten Typ zu verlangen, entwickelt TypeScript sein Verständnis des Variablentyps jedes Mal weiter, wenn ein neuer Wert zugewiesen wird.

Im folgenden Beispiel wird der sich entwickelnden `any`-Variablen `rocker` zunächst eine Zeichenkette zugewiesen, was bedeutet, dass sie über String-Methoden wie `toUpperCase` verfügt, bevor sich die Variable dann zum Typ `number` weiterentwickelt:

```
let rocker; // Typ: any

rocker = "Joan Jett"; // Typ: string
rocker.toUpperCase(); // Ok

rocker = 19.58; // Typ: number
rocker.toPrecision(1); // Ok

rocker.toUpperCase();
//     ~~~~~~~~~~~
// Error: 'toUpperCase' does not exist on type 'number'.
```

TypeScript konnte erkennen, dass wir die Methode `toUpperCase()` für eine Variable vom Typ `number` aufgerufen haben. Es konnte jedoch nicht herausfinden, ob es von uns beabsichtigt war, die Variable von `string` in `number` umzuwandeln.

Lässt man zu, dass Variablen als evolving `any` typisiert werden – bzw. generell die Verwendung des Typs `any` –, untergräbt das teilweise den Zweck der Typprüfung! TypeScript funktioniert am besten, wenn es weiß, welchen Typ Ihre Werte haben sollen. Ein Großteil der Typprüfung von TypeScript kann nicht auf als `any` typisierte Werte angewendet werden, da diese keine bekannten Typen haben, die über-

prüft werden können. Die Konfiguration der Beanstandungen impliziter any-Typen wird in Kapitel 13, »Konfigurationsoptionen«, behandelt.

TypeScript bietet eine Syntax, mit deren Hilfe der Typ einer Variablen deklariert werden kann, ohne ihr einen Anfangswert zuweisen zu müssen: *Typanmerkungen* (bzw. *Typannotationen*). Eine Typanmerkung steht hinter dem Namen einer Variablen und besteht aus einem Doppelpunkt, gefolgt von einer Typbezeichnung.

Die folgende Typanmerkung legt fest, dass die Variable `rocker` vom Typ `string` sein soll:

```
let rocker: string;
rocker = "Joan Jett";
```

Diese Typanmerkungen existieren nur innerhalb von TypeScript – sie haben keine Auswirkungen auf den Laufzeitcode und sind keine gültige JavaScript-Syntax. Wenn Sie `tsc` ausführen, um TypeScript-Quellcode zu JavaScript zu kompilieren, werden sie entfernt. Das vorherige Beispiel würde etwa zu folgendem JavaScript kompiliert werden:

```
// Ausgegebene .js-Datei
let rocker;
rocker = "Joan Jett";
```

Die Zuweisung eines Werts, dessen Typ nicht dem per Typanmerkung genannten Typ der Variablen zugewiesen werden kann, führt zu einem Typfehler.

Der folgende Codeausschnitt weist einer Variablen vom Typ `rocker`, die zuvor als `string` deklariert wurde, eine Zahl zu und verursacht einen Typfehler:

```
let rocker: string;
rocker = 19.58;
// Error: Type 'number' is not assignable to type 'string'.
```

In den nächsten Kapiteln werden Sie sehen, wie Sie durch Typanmerkungen das Codeverständnis von TypeScript erweitern können, sodass es Sie während der Entwicklung besser unterstützen kann. TypeScript enthält eine ganze Reihe neuer Syntaxelemente, wie z. B. diese Typanmerkungen, die nur innerhalb des Typsystems existieren.

Nichts, was nur im Typsystem existiert, wird in das emittierte JavaScript übernommen. TypeScript-Typen haben keinerlei Einfluss auf das ausgegebene JavaScript.

Unnötige Typanmerkungen

Mit Typanmerkungen können wir TypeScript Informationen geben, die es allein nicht hätte herausfinden können. Man könnte sie auch für Variablen verwenden, die unmittelbar ableitbare Typen haben, aber man würde damit TypeScript nichts mitteilen, was es nicht ohnehin schon weiß.

Die folgende Typanmerkung `: string` ist überflüssig, da TypeScript bereits ableiten kann, dass `firstName` vom Typ `string` sein muss:

```
let firstName: string = "Tina";
//           ~~~~~~~~ Bietet dem Typsystem keine zusätzliche Information ...
```

Wenn Sie einer Variablen mit einem Anfangswert eine Typanmerkung hinzufügen, prüft TypeScript, ob sie mit dem Typ des Variablenwerts übereinstimmt.

Die folgende Variable `firstName` ist als vom Typ `string` deklariert, wird aber mit der Zahl `42`, also einem Wert vom Typ `number` initialisiert, was TypeScript als inkompatibel betrachtet:

```
let firstName: string = 42;
//  ~~~~~~~~~
// Error: Type 'number' is not assignable to type 'string'.
```

Viele Entwicklerinnen und Entwickler – mich eingeschlossen – ziehen es normalerweise vor, bei Variablen, bei denen Typanmerkungen keinen Vorteil bieten, diese wegzulassen. Die manuelle Eingabe von Typanmerkungen kann mühsam sein – vor allem, wenn sich Änderungen ergeben und speziell für die komplexen Typen, die ich Ihnen später in diesem Buch zeigen werde.

Allerdings kann es manchmal nützlich sein, Variablen explizite Typanmerkungen hinzuzufügen, um den Code klar zu dokumentieren und/oder TypeScript gegen versehentliche Änderungen des Variablentyps zu schützen. Wir werden in späteren Kapiteln sehen, wie man mit Typanmerkungen TypeScript explizit Informationen mitteilen kann, die es ansonsten nicht alleine hätte herleiten können.

Typformen

TypeScript überprüft nicht nur, ob die den Variablen zugewiesenen Werte ihren ursprünglichen Typen entsprechen. TypeScript weiß auch, welche Member-Eigenschaften auf Objekten vorhanden sein sollten. Wenn Sie versuchen, auf eine Eigenschaft einer Variablen zuzugreifen, stellt TypeScript sicher, dass diese Eigenschaft auf dem Typ der Variablen existiert.

Angenommen, wir deklarieren eine `rapper`-Variable vom Typ `string`. Später, wenn wir diese `rapper`-Variable verwenden, sind nur solche Operationen erlaubt, von denen TypeScript weiß, dass sie auf Strings funktionieren:

```
let rapper = "Queen Latifah";
rapper.length; // Ok
```

Operationen, für die das nicht gilt, werden verhindert:

```
rapper.push('!');
//     ~~~~
// Property 'push' does not exist on type 'string'.
```

Typen können auch komplexere Formen annehmen, vor allem Objekte. Im folgenden Codeausschnitt erkennt TypeScript, dass das Objekt `cher` keinen Schlüssel `middleName` besitzt, und beschwert sich:

```
let cher = {
  firstName: "Cherilyn",
  lastName: "Sarkisian",
};

cher.middleName;
//   ~~~~~~~~~~
//   Property 'middleName' does not exist on type
//   '{ firstName: string; lastName: string; }'.
```

TypeScripts Verständnis von Objektformen ermöglicht es, dass nicht nur Probleme bei der Zuweisbarkeit, sondern auch bei der Verwendung von Objekten gemeldet werden. In Kapitel 4, »Objekte«, werden weitere leistungsstarke Sprachmerkmale von TypeScript rund um Objekte und Objekttypen beschrieben.

Module

Die Programmiersprache JavaScript enthielt bis vor relativ kurzer Zeit keine Spezifikation dazu, wie Dateien untereinander Code teilen können. Mit ECMAScript 2015 wurden »ECMAScript-Module« (ESM) eingeführt, um die Import- und Exportsyntax zwischen Dateien zu standardisieren.

Als Referenz importiert die folgende Moduldatei einen Wert `value` aus einer Geschwisterdatei `/values` und exportiert eine Variable `doubled` mit dem Doppelten des importierten Werts:

```
import { value } from "./values";

export const doubled = value * 2;
```

Um der ECMAScript-Spezifikation zu entsprechen, werde ich in diesem Buch die folgende Nomenklatur verwenden:

Modul
: Eine Datei, die ganz oben eine `export`- oder `import`-Anweisung enthält

Skript
: Eine Datei, die kein Modul ist

TypeScript ist in der Lage, sowohl mit diesen modernen Moduldateien als auch mit älteren Dateien zu arbeiten. Alles, was in einem Modul deklariert ist, steht nur innerhalb dieser Datei zur Verfügung, es sei denn, eine in dieser Datei enthaltene explizite `export`-Anweisung exportiert es. In einem Modul darf durchaus eine Variable mit einem Namen deklariert werden, der schon in einem anderen Modul verwendet wird. Ein Namenskonflikt entsteht erst, wenn eine Datei die Variable der anderen Datei importiert.

Die folgenden Dateien `a.ts` und `b.ts` sind zwei Module, die problemlos identisch benannte `shared`-Variablen exportieren. Die Datei `c.ts` verursacht allerdings einen

Typfehler, weil zwischen einem importierten `shared` und dem eigenen, lokal definierten Wert ein Namenskonflikt entsteht:

```
// a.ts
export const shared = "Cher";

// b.ts
export const shared = "Cher";

// c.ts
import { shared } from "./a";
//       ~~~~~~
// Error: Import declaration conflicts with local declaration of 'shared'.

export const shared = "Cher";
//           ~~~~~~
// Error: Individual declarations in merged declaration
// 'shared' must be all exported or all local.
```

Ist eine Datei jedoch ein Skript, betrachtet TypeScript sie als dem globalen Geltungsbereich zugehörig, was bedeutet, dass alle Skripte Zugriff auf ihren Inhalt haben. Daraus folgt, dass Variablen über alle Skriptdateien hinweg eindeutige Namen haben müssen.

Die folgenden Dateien `a.ts` und `b.ts` werden als Skripte betrachtet, da sie keine `export`- oder `import`-Anweisungen im Modulstil enthalten. Das bedeutet, dass ihre gleichnamigen Variablen genauso miteinander in Konflikt geraten, als wären sie in derselben Datei deklariert:

```
// a.ts
const shared = "Cher";
//    ~~~~~~
// Cannot redeclare block-scoped variable 'shared'.

// b.ts
const shared = "Cher";
//    ~~~~~~
// Cannot redeclare block-scoped variable 'shared'.
```

Wenn Ihnen solche »Cannot redeclare...«-Fehler in einer TypeScript-Datei begegnen, kann es daran liegen, dass Sie der Datei noch eine `export`- oder `import`-Anweisung hinzufügen müssen. Gemäß der ECMAScript-Spezifikation können Sie die Moduleigenschaft einer Datei auch ohne explizite `export`- oder `import`-Anweisung erzwingen, indem Sie irgendwo innerhalb der Datei ein `export {};` hinzufügen:

```
// a.ts und b.ts
const shared = "Cher"; // Ok

export {};
```

TypeScript erkennt keine Typen von Importen und Exporten in Dateien, die mithilfe älterer Modulsysteme wie CommonJS geschrieben wurden. TypeScript betrachtet Werte, die von CommonJS-artigen `require`-Funktionen zurückgegeben werden, als vom Typ `any`.

Zusammenfassung

In diesem Kapitel ging es darum, wie das Typsystem von TypeScript im Kern funktioniert, und im Einzelnen um folgende Themen:

- Was ein »Typ« ist und welche primitiven Datentypen TypeScript erkennt
- Was ein »Typsystem« ist und wie das Typsystem von TypeScript Code versteht
- Wie sich Typfehler von Syntaxfehlern unterscheiden
- Abgeleitete Variablentypen und die Zuweisbarkeit von Variablen
- Typanmerkungen zur expliziten Deklaration von Variablentypen und die Vermeidung von evolving any-Typen
- Überprüfung von Objektmembern bei Typformen
- Geltungsbereich von Deklarationen in ECMAScript-Moduldateien im Vergleich zu Skriptdateien

Nachdem Sie dieses Kapitel gelesen haben, sollten Sie das Gelernte auf *https://learningtypescript.com/the-type-system* anwenden und einüben.

Warum haben sich `number` und `string` getrennt?

Sie waren nicht der Typ füreinander.