O'REILLY®

Programmieren in TypeScript

Skalierbare JavaScript-Applikationen entwickeln

Boris Cherny

Übersetzung von Jørgen W. Lang

Inhalt

Cover

Titel

Impressum

Widmung

Inhalt

Vorwort

1 Einführung

2 TypeScript aus der Vogelperspektive

- Der Compiler
- Das Typsystem
  - TypeScript im Vergleich mit JavaScript
- Einrichtung des Codeeditors
  - tsconfig.json
  - tslint.json
- index.ts
- Übungen

3 Alles über Typen

- Wo wir gerade von Typen sprechen
- Das ABC der Typen
  - any
  - unknown (unbekannt)
  - boolean (boolescher Wert)
  - number (Zahl)
  - bigint
  - string (String, Zeichenkette)

symbol (Symbole)
Objekte
Kurze Unterbrechung: Typaliase, Vereinigungs- und Schnittmengen
Arrays
Tupel
null, undefined, void und never
Enums
Zusammenfassung
Übungen
4 Funktionen
Funktionen deklarieren und aufrufen
Optionale und Standardparameter
Restparameter
call, apply und bind
this typisieren
Generator-Funktionen
Iteratoren
Aufrufsignaturen (Call Signatures)
Kontextabhängige Typisierung
Überladene Funktionstypen
Polymorphismus
Wann werden Generics gebunden?
Wo können Generics deklariert werden?
Generische Typableitung
Generische Typaliase
Begrenzter Polymorphismus
Generische Standardtypen
Typgetriebene Entwicklung

Zusammenfassung

Übungen

5 Klassen und Interfaces

Klassen und Vererbung

super

this als Rückgabetyp verwenden

Interfaces

Deklarationen verschmelzen

Implementierungen

Implementierung von Interfaces im Vergleich mit der Erweiterung abstrakter Klassen

Klassen sind strukturell typisiert

Klassen deklarieren Werte und Typen

Polymorphismus

Mixins

Dekoratoren

Finale Klassen simulieren

Entwurfsmuster (Design Patterns)

Factory-Muster

Builder-Muster

Zusammenfassung

Übungen

6 Fortgeschrittene Typen

Beziehungen zwischen Typen

Subtypen und Supertypen

Varianz

Zuweisbarkeit

Typerweiterung

Typverfeinerung (refinement)
Totalität
Fortgeschrittene Objekttypen
Typoperatoren für Objekttypen
Der Record-Typ
Abgebildete Typen
Das Companion-Objektmuster
Fortgeschrittene Funktionstypen
Typinferenz für Tupel verbessern
Benutzerdefinierte Type Guards (Typschutz)
Konditionale Typen
Distributive Bedingungen
Das Schlüsselwort infer
Eingebaute konditionale Typen
Notausgänge
Typzusicherungen (type assertions)
Nicht-null-Zusicherungen
Zusicherungen für definitive Zuweisungen
Nominale Typen simulieren
Prototypen sicher erweitern
Zusammenfassung
Übungen
7 Fehlerbehandlung
Die Rückgabe von null
Ausnahmen auslösen
Ausnahmen zurückgeben
Der Option-Typ
Zusammenfassung

Übung
8 Asynchrone Programmierung, Nebenläufigkeit und Parallelismus
JavaScripts Eventschleife
Mit Callbacks arbeiten
Promises verwenden, damit die eigene Gesundheit nicht leidet
async und await
Asynchrone Streams
Event-Emitter
Typsicheres Multithreading
Im Browser: Mit Web Workers
In NodeJS: Mit Kindprozessen
Zusammenfassung
Übungen
9 Frontend- und Backend-Frameworks
Frontend-Frameworks
React
Angular 6/7
Typsichere APIs
Backend-Frameworks
Zusammenfassung
10 Namensräume.Module
Eine kurze Geschichte der JavaScript-Module
import, export
Dynamische Importe
CommonJS und AMD Code verwenden
Modulmodus oder Skriptmodus
Namensräume
Kollisionen

Kompilierte Ausgaben
Deklarationsverschmelzung (declaration merging)
Zusammenfassung
Übung
11 Zusammenarbeit mit JavaScript
Typdeklarationen
Ambiente Variablendeklarationen
Ambiente Typdeklarationen
Ambiente Moduldeklarationen
Schrittweise Migration von JavaScript zu TypeScript
Schritt 1: TSC hinzufügen
Schritt 2a: Typechecking für JavaScript aktivieren (optional)
Schritt 2b: Add JSDoc Annotations (Optional)
Schritt 3: Versehen Sie Ihre Dateien mit der Endung .ts
Schritt 4: Verwenden Sie den strict-Modus
Typermittlung für JavaScript
JavaScript von Drittanbietern verwenden
JavaScript mit eigenen Typdeklarationen
JavaScript, für das es Typdeklarationen auf DefinitelyTyped gibt
JavaScript, für das es keine Typdeklarationen auf DefinitelyTyped gibt
Zusammenfassung
12 TypeScript-Projekte erstellen und ausführen
Das TypeScript-Projekt erstellen
Projekt-Layout
Artefakte
Das Kompilierungsziel festlegen
Sourcemaps verwenden
Projektreferenzen

Fehlerberichte
TypeScript auf dem Server ausführen
TypeScript im Browser ausführen
TypeScript-Code über NPM veröffentlichen
Triple-Slash-Direktiven (///)
Die types-Direktive
Die amd-module-Direktive
Zusammenfassung
13 Abschluss
A Typoperatoren
B Hilfsfunktionen für Typen
C Geltungsbereiche für Deklarationen
D Rezepte für das Schreiben von Deklarationsdateien für JavaScript-Module von Drittanbietern
E Triple-Slash-Direktiven
F TSC-Compiler-Flags für mehr Sicherheit
G TSX
Index
Über den Autor

KAPITEL 4

Funktionen

Im vorigen Kapitel haben wir die Grundlagen von TypeScripts Typsystem behandelt: primitive Typen, Arrays, Tupel und Enums sowie die Grundlagen von TypeScripts Typableitung (Inferenz) und die Funktionsweise der Typzuweisbarkeit. (Details hierzu finden Sie in Kapitel 6). Jetzt sind Sie bereit für TypeScripts pièce de résistance (oder raison d'être, wenn Sie die funktionale Programmierung bevorzugen): Funktionen. In diesem Kapitel geht es unter anderem um:

- die verschiedenen Möglichkeiten, Funktionen in TypeScript zu deklarieren und aufzurufen,
- das Überladen von Signaturen,
- polymorphe Funktionen und
- polymorphe Typaliase.

Funktionen deklarieren und aufrufen

In JavaScript sind Funktionen Objekte erster Klasse. Das heißt, Sie können Funktionen wie normale Objekte benutzen: Sie können ihnen Variablen zuweisen, sie an andere Funktionen übergeben, sie aus Funktionen zurückgeben, sie Objekten und Prototypen zuweisen, sie mit Eigenschaften versehen und diese wieder auslesen etc. In JavaScript sind Funktionen sehr vielseitig, und TypeScript bildet das alles mit seinem umfangreichen Typsystem ab.

Hier sehen Sie, wie eine Funktion in TypeScript aussieht (das sollte Ihnen aus dem vorigen Kapitel bekannt vorkommen):

```
function add(a: number, b: number) {
  return a + b
}
```

Üblicherweise werden Funktionsparameter (hier a und b) explizit annotiert. Zwar wird TypeScript die Typen im Funktionskörper immer ableiten, das gilt normalerweise aber nicht für die Parameter (abgesehen von ein paar Ausnahmen, in denen Parametertypen aus dem Kontext abgeleitet werden können. Mehr hierzu in »Kontextabhängige Typisierung« auf Seite 59). Der Rückgabetyp *wird* abgeleitet, aber wenn Sie wollen, können Sie auch ihn explizit annotieren:

```
function add(a: number, b: number): number {
  return a + b
}
```

In diesem Buch werde ich die Rückgabetypen explizit annotieren, sofern es Ihnen, dem Leser, beim Verständnis dessen, was die Funktion tut, hilft. Ansonsten lasse ich die Annotationen weg, weil TypeScript die Typen für uns ableitet. Warum soll man sich doppelte Arbeit machen?

Im letzten Beispiel habe ich die *benannte Funktionssyntax* benutzt, um die Funktion zu deklarieren. Tatsächlich unterstützen JavaScript und TypeScript mindestens fünf Wege, dies zu tun:

```
// Benannte Funktion
function greet(name: string) {
  return ‘hello ‘ + name
```

```
}

// Funktionsausdruck

let greet2 = function(name: string) {

  return ‘hello ‘ + name

}

// Funktionsausdruck mit Pfeilschreibweise

let greet3 = (name: string) => {

  return ‘hello ‘ + name

}

// Kurzschriftversion des Funktionsausdrucks mit
Pfeilschreibweise

let greet4 = (name: string) =>

  ‘hello ‘ + name

// Funktions-Konstruktor

let greet5 = new Function(‘name’, ‘return “hello ” + name’)
```

Neben Funktionskonstruktoren (die vollkommen unsicher sind und daher nur benutzt werden sollten, wenn Sie von einem Schwarm Bienen verfolgt werden)[1], werden alle diese Schreibweisen von TypeScript typsicher unterstützt. Hierbei

folgen sie den gleichen Regeln zu den normalerweise nötigen Annotationen für Parametertypen und optionale Rückgabetypen.

Eine kurze Auffrischung zur Terminologie:

- Ein Parameter besteht aus Daten, die für die Ausführung einer Funktion benötigt werden. Er wird als Teil der Funktionsdeklaration angegeben. Diese Art von Parameter wird auch als *formaler Parameter* bezeichnet.
- Ein Argument ist ein Datenstück, das Sie einer Funktion bei ihrem Aufruf übergeben. Diese Art von Parameter wird auch als *tatsächlicher Parameter* bezeichnet.

Wenn Sie in TypeScript eine Funktion aufrufen, müssen Sie keine zusätzlichen Typinformationen angeben. Übergeben Sie einfach ein paar Argumente, und schon überprüft TypeScript, ob die Typen Ihrer Argumente mit den Typen Ihrer Funktionsparameter kompatibel sind:

```
add(1, 2)         // ergibt 3

greet(‘Crystal’)  // ergibt ‘hello Crystal’
```

Wenn Sie ein Argument vergessen oder ein Argument mit dem falschen Typ übergeben, wird TypeScript Sie umgehend darüber informieren:

```
add(1)            // Error TS2554: Expected 2 arguments,
but got 1.

add(1, ‘a’)       // Error TS2345: Argument of type ‘“a”’
is not assignable

                  // to parameter of type ‘number’.
```

Optionale und Standardparameter

Wie bei Objekt- und Tupel-Typen können Sie Parameter mit einem Fragezeichen (?) als optional kennzeichnen. Bei der Deklaration Ihrer Funktionsparameter werden die erforderlichen Parameter zuerst angegeben, gefolgt von den optionalen Parametern:

```
function log(message: string, userId?: string) {

  let time = new Date().toLocaleTimeString()

  console.log(time, message, userId || ‘Not signed in’)

}

log(‘Page loaded’) // Protokolliert “12:38:31 PM Page
loaded Not signed in”

log(‘User signed in’, ‘da763be’) // Protokolliert “12:38:31
PM

                                 // User signed in da763be”
```

Wie in JavaScript können Sie Standardwerte für optionale Parameter angeben. Semantisch hat das Ähnlichkeit mit der Markierung eines Parameters als optional, weil der Aufrufer diesen Wert nicht länger übergeben muss (ein Unterschied liegt darin, dass Standardparameter nicht unbedingt am Ende der Parameterliste stehen müssen, wie es für optionale Parameter der Fall ist).

Wir können die `log`-Funktion also auch so schreiben:

```
function log(message: string, userId = ‘Not signed in’) {

  let time = new Date().toISOString()

  console.log(time, message, userId)
```

```
}

log('User clicked on a button', 'da763be')

log('User signed out')
```

Bei der Definition eines Standardwerts für `userID` haben wir gleichzeitig die Markierung als optional (?) entfernt. Der Parameter muss auch nicht mehr typisiert werden. TypeScript kann den Parametertyp selbstständig aus seinem Standardwert ableiten. Dadurch wird unser Code kürzer und leichter lesbar.

Natürlich können Sie Ihre Parameter auch weiterhin mit expliziten Typannotationen versehen. Das funktioniert genauso wie für Parameter ohne Standardwerte:

```
type Context = {

  appId?: string

  userId?: string

}

function log(message: string, context: Context = {}) {

  let time = new Date().toISOString()

  console.log(time, message, context.userId)

}
```

Standardparameter werden Sie mit Sicherheit häufiger benutzen als optionale Parameter.

Restparameter

Übernimmt eine Funktion eine Liste mit Argumenten, können Sie diese natürlich einfach als Array übergeben:

```
function sum(numbers: number[]): number {

  return numbers.reduce((total, n) => total + n, 0)

}

sum([1, 2, 3]) // ergibt 6
```

Manchmal wollen Sie lieber eine *variadische* Funktions-API benutzen, die eine beliebige Anzahl von Argumenten übernimmt, als eine API *fester Arität* (oder Stelligkeit), die eine festgelegte Anzahl von Argumenten benötigt. Traditionellerweise wurde hierfür JavaScripts »magisches« `arguments`-Objekt benötigt.

`arguments` ist »magisch«, weil die JavaScript-Runtime es automatisch für Sie in Funktionen bereitstellt und der Argumenteliste zuweist, die an die Funktion übergeben wurde. Da `arguments` nur Array-ähnlich ist, ohne wirklich eines zu sein, müssen Sie es zuerst in ein Array umwandeln, bevor Sie das eingebaute `.reduce` daran aufrufen können:

```
function sumVariadic(): number {

  return Array

    .from(arguments)

    .reduce((total, n) => total + n, 0)

}

sumVariadic(1, 2, 3) // ergibt 6
```

Bei der Verwendung von `arguments` gibt es allerdings ein großes Problem, denn es ist vollkommen unsicher! Wenn Sie den Mauszeiger in Ihrem Codeeditor über `total` oder `n` bewegen, sieht die Anzeige so ähnlich aus wie in Abbildung 4-1.

Abbildung 4-1: arguments ist unsicher

Das heißt, TypeScript leitet für `n` und `total` den Typ `any` ab und lässt sie stillschweigend passieren – es sei denn, Sie verwenden `sumVariadic`:

```
sumVariadic(1, 2, 3) // Error TS2554: Expected 0 arguments,
but got 3.
```

Da wir nicht deklariert haben, dass `sumVariadic` Argumente übernimmt, erwartet TypeScript auch keine. Versuchen wir trotzdem, Argumente zu übergeben, erhalten wir einen `TypeError`.

Wie aber kann man variadische Funktionen sicher typisieren?

Mit Restparametern! Anstelle der magischen, aber unsicheren Variablen `arguments` können wir Restparameter benutzen, damit `sum` eine beliebige Zahl von Argumenten auf sichere Weise übernehmen kann:

```
function sumVariadicSafe(…numbers: number[]): number {

  return numbers.reduce((total, n) => total + n, 0)

}

sumVariadicSafe(1, 2, 3) // ergibt 6
```

Das war schon alles. Der einzige Unterschied zwischen diesem variadischen sum und unserer ursprünglichen sum-Funktion mit nur einem Parameter ist das zusätzliche … in der Parameterliste. Sonst muss nichts verändert werden – und das Ganze ist dabei noch vollkommen typsicher.

Eine Funktion kann höchstens einen Restparameter besitzen und muss an letzter Position der Parameterliste einer Funktion stehen. Ein Beispiel hierfür ist TypeScripts eigene Deklaration für console.log (Sie müssen noch nicht wissen, was ein Interface ist – in Kapitel 5 gehen wir genauer darauf ein). console.log übernimmt einen optionalen message-Parameter und eine beliebige Anzahl zusätzlicher Argumente:

```
interface Console {
  log(message?: any, …optionalParams: any[]): void
}
```

call, apply und bind

Neben dem Funktionsaufruf mit runden Klammern () unterstützt JavaScript noch mindestens zwei weitere Formen. Sehen wir uns hierzu die add-Funktion vom Anfang des Kapitels noch einmal an:

```
function add(a: number, b: number): number {
  return a + b
}
add(10, 20)                // ergibt 30
add.apply(null, [10, 20])  // ergibt 30
add.call(null, 10, 20)     // ergibt 30
```

```
add.bind(null, 10, 20)()   // ergibt 30
```

`apply` bindet innerhalb der Funktion einen Wert an `this` (in diesem Beispiel verbinden wir `this` und `null`). Das zweite Argument wird auf die Funktionsparameter verteilt. `call` macht das Gleiche, allerdings werden die Argumente in der angegebenen Reihenfolge angewandt, anstatt den Spread-Operator (neu in ES2015) zu verwenden.

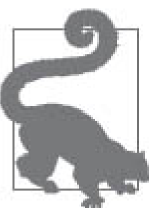
`bind()` funktioniert so ähnlich, indem es ein `this`-Argument und eine Liste mit Argumenten an Ihre Funktion *bindet*. Der Unterschied liegt darin, dass `bind` Ihre Funktion nicht aufruft. Stattdessen gibt es eine neue Funktion zurück, die Sie dann per `()`, `.call` oder `.apply` aufrufen können. Dabei können weitere Argumente übergeben werden, die bei Bedarf an bisher ungebundene Parameter gebunden werden.

TSC Flag: strictBindCallApply

Um `.call`, `.apply` und `.bind` in Ihrem Code sicher nutzen zu können, sollten Sie auf jeden Fall die Option `strictBindCallApply` in Ihrer *tsconfig.json* aktivieren (bei Verwendung des `strict`-Modus ist diese Option automatisch aktiviert).

this typisieren

Ohne JavaScript-Erfahrung überrascht es vielleicht, dass `this` in JavaScript für alle Funktionen definiert ist – nicht nur für solche in Methoden oder Klassen. Je nachdem, wie Sie Ihre Funktion aufrufen, hat `this` einen unterschiedlichen Wert. Das macht `this` ziemlich zerbrechlich und auch noch launisch.



Deshalb verwenden viele Teams `this` nur für Klassenmethoden. Um das auch für Ihre Codebasis zu tun, aktivieren Sie die TSLint-Regel `no-invalid-this`.

Der Grund für die Zerbrechlichkeit von `this` liegt in der Art seiner Zuweisung. Die allgemeine Regel lautet, dass `this` den Wert dessen übernimmt, was sich beim Methodenaufruf links vom Punkt befindet. Zum Beispiel:

```
let x = {
```

```
  a() {

    return this

  }

}

x.a() // this ist das Objekt x im Körper von a()
```

Wenn Sie a vor dem Aufruf neu zuweisen, erhalten Sie dagegen ein anderes Ergebnis!

```
let a = x.a

a() // Jetzt hat this im Körper von a() den Wert undefined
```

Angenommen, Sie verwenden folgende Hilfsfunktion für die Formatierung von Kalenderdaten:

```
function fancyDate() {

  return
  ${this.getDate()}/${this.getMonth()}/${this.getFullYear()
  }

}
```

Sie haben diese API in Ihren Anfangstagen als Programmierer entwickelt (bevor Sie Funktionsparameter kannten). Um `fancyDate` zu verwenden, müssen Sie die Funktion mit einem an `this` gebundenen `Date`-Objekt aufrufen:

```
fancyDate.call(new Date) // ergibt “4/14/2005”
```

Ohne die Bindung von `Date` an `this` wird ein Laufzeitfehler ausgelöst!

```
fancyDate() // Uncaught TypeError: this.getDate is not a
function
```

Eine eingehende Erforschung von `this` würde den Rahmen dieses Buchs leider sprengen.[2] Trotzdem ist ziemlich überraschend, dass `this` von der Art des Funktionsaufrufs abhängt.

Zum Glück hält TypeScript Ihnen den Rücken frei. Verwendet Ihre Funktion `this`, sollten Sie den erwarteten Typ von `this` auf jeden Fall deklarieren (vor allen weiteren Parametern). Dann sorgt TypeScript dafür, dass `this` tatsächlich bei jedem Aufruf Ihren Erwartungen entspricht und es nicht wie die übrigen Parameter behandelt. Wird `this` als Teil einer Funktionssignatur verwendet, gilt es als reserviertes Wort:

```
function fancyDate(this: Date) {

  return
  ${this.getDate()}/${this.getMonth()}/${this.getFullYear()
  }

}
```

Wenn Sie `fancyDate` jetzt aufrufen, passiert Folgendes:

```
fancyDate.call(new Date) // ergibt “6/13/2008”

fancyDate() // Error TS2684: The ‘this’ context of type
‘void’ is

            // not assignable to method’s ‘this’ of type
            ‘Date’.
```

Wir haben einen Laufzeitfehler verhindert und TypeScript mit genug Informationen versorgt, um uns bereits schon während der Kompilierung davor zu warnen.

TSC Flag: noImplicitThis

Um zu erzwingen, dass der Typ von `this` in Funktionen immer explizit annotiert werden muss, können Sie die Einstellung `noImplicitThis` in Ihrer *tsconfig.json* aktivieren. Verwenden Sie den `strict`-Modus, ist `noImplicitThis` bereits standardmäßig aktiviert.

Beachten Sie, dass `noImplicitThis` keine `this`-Annotationen für Klassen oder Funktionen an Objekten erzwingt.

Generator-Funktionen

Generator-Funktionen (oder kurz *Generatoren*) sind eine bequeme Möglichkeit, eine Reihe von Werten zu erzeugen. Sie wollen dem Consumer des Generators eine feine Kontrolle über das Tempo ermöglichen, in dem die Werte produziert werden. Da sie faul sind – d.h., Sie erzeugen den nächsten Wert nur, wenn der Benutzer danach fragt –, können Generatoren Dinge tun, die ansonsten ziemlich schwer sind, zum Beispiel die Erzeugung einer Endlosliste.

Generatoren funktionieren wie hier gezeigt:

```
function* createFibonacciGenerator() { ❶
  let a = 0
  let b = 1
  while (true) { ❷
    yield a; ❸
    [a, b] = [b, a + b] ❹
  }
```

```
}

let fibonacciGenerator = createFibonacciGenerator() //
IterableIterator<number>

fibonacciGenerator.next()   // ergibt {value: 0, done:
false}

fibonacciGenerator.next()   // ergibt {value: 1, done:
false}

fibonacciGenerator.next()   // ergibt {value: 1, done:
false}

fibonacciGenerator.next()   // ergibt {value: 2, done:
false}

fibonacciGenerator.next()   // ergibt {value: 3, done:
false}

fibonacciGenerator.next()   // ergibt {value: 5, done:
false}
```

❶ Der Asterisk (`*`) vor dem Funktionsnamen kennzeichnet die Funktion als Generator. Der Aufruf der Funktion gibt einen iterierbaren Iterator zurück.

❷ Unser Generator kann unendlich viele Werte erzeugen.

❸ Generatoren benutzen das Schlüsselwort `yield`, um Werte zurückzugeben. Fordert ein Consumer den nächsten Wert des Generators an (etwa durch den Aufruf von `next`), gibt `yield` ein Ergebnis an den Consumer zurück und hält die Ausführung an, bis der nächste Wert angefordert wird. Dadurch führt der `while(true)`-Teil nicht sofort dazu, dass das Programm sich in einer Endlosschleife aufhängt und abstürzt.

❹ Um die folgende Fibonacci-Zahl zu berechnen, erhält `a` den Wert von `b` und `b` im gleichen Arbeitsschritt das Ergebnis von `a + b`.

Wir haben `createFibonacciGenerator` aufgerufen und einen `IterableIterator` zurückgegeben. Bei jedem Aufruf von `next` berechnet der Iterator die folgende Fibonacci-Zahl und gibt diese per `yield` an uns zurück. Dabei ist TypeScript in der Lage, den Typ des Iterators aus dem von `yield` zurückgegebenen Wert abzuleiten.

Um den Generator explizit zu annotieren, können Sie den zurückgegebenen Typ mit einem `IterableIterator` umgeben:

```
function* createNumbers(): IterableIterator<number> {

  let n = 0

  while (1) {

    yield n++

  }

}

let numbers = createNumbers()

numbers.next()              // ergibt {value: 0, done:
false}

numbers.next()              // ergibt {value: 1, done:
false}

numbers.next()              // ergibt {value: 2, done:
false}
```

Wir werden in diesem Buch nicht weiter auf Generatoren eingehen. Generatoren sind ein sehr umfangreiches Thema, und da es in diesem Buch um TypeScript geht, möchte ich Sie nicht zu sehr mit JavaScript-Eigenheiten ablenken. Kurz gesagt, sind Generatoren ein supercooles JavaScript-Sprachmerkmal, das auch

von TypeScript unterstützt wird. Mehr zu Generatoren finden Sie auf der entsprechenden Seite im MDN (*https://mzl.la/2UitIk4*).

Iteratoren

Iteratoren sind das Gegenstück zu Generatoren. Während Generatoren einen Wertestrom erzeugen können, bieten Iteratoren die Möglichkeit, diese Werte zu verarbeiten (oder zu »konsumieren«). Die Terminologie kann ziemlich verwirrend sein. Daher beginnen wir mit ein paar grundsätzlichen Definitionen.

Iterable (»iterierbar«)

Jedes Objekt, das eine Eigenschaft namens `Symbol.iterator` besitzt, dessen Wert eine Funktion ist, die einen Iterator zurückgibt.

Iterator

Jedes Objekt, das eine Methode namens `next` definiert, die ein Objekt mit den Eigenschaften `value` und `done` zurückgibt.

Wenn Sie einen Generator erzeugen (zum Beispiel durch den Aufruf von `create FibonacciGenerator`), erhalten Sie einen Wert zurück, der sowohl ein *Iterable* als auch ein *Iterator* ist, weil er nicht nur eine `Symbol.iterator` -Eigenschaft, sondern auch eine `next`-Methode definiert.

Sie können einen Iterator oder ein Iterable auch manuell erzeugen. Erstellen Sie hierfür ein Objekt (oder eine Klasse), das `Symbol.iterator` bzw. `next` implementiert. Im folgenden Beispiel definieren wir einen Iterator, der die Zahlen von 1 bis 10 zurückgibt:

```
let numbers = {
  *[Symbol.iterator]() {
    for (let n = 1; n <= 10; n++) {
      yield n
    }
```

```
  }
}
```

Wenn Sie den oben stehenden Iterator in Ihrem Codeeditor eingeben, sehen Sie, welchen Typ TypeScript dafür ableitet (Abbildung 4-2).

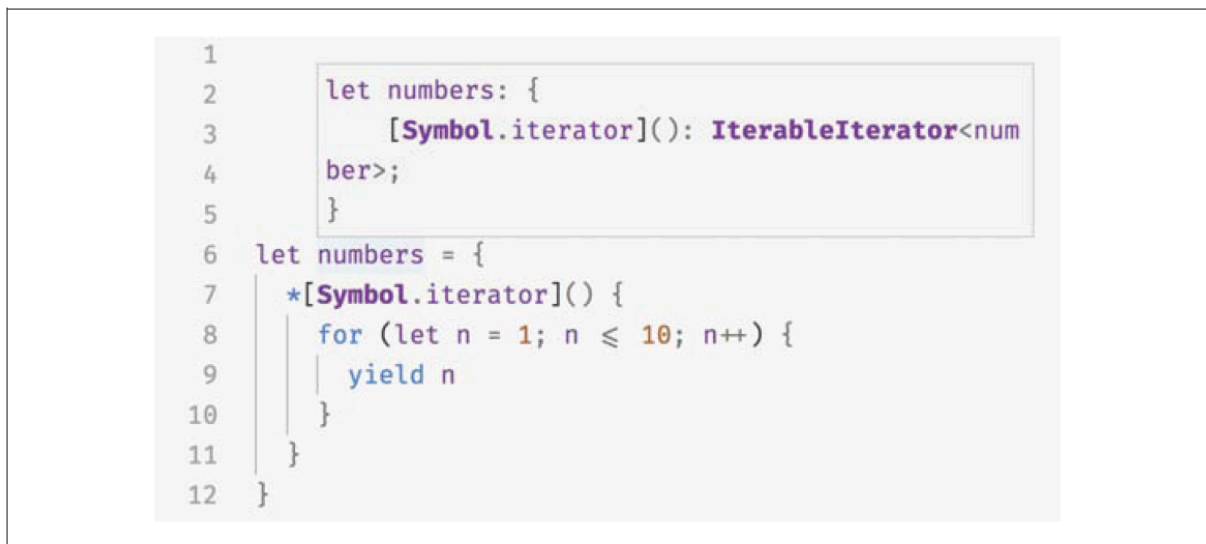


Abbildung 4-2: Einen Iterator manuell definieren

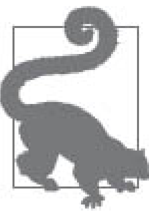
Anders gesagt, ist `numbers` ein Iterator, und der Aufruf der Generatorfunktion `numbers[Symbol.iterator]()` gibt einen iterierbaren Iterator zurück.

Neben der Definition eigener Iteratoren können Sie auch die JavaScript-eigenen Iteratoren für häufige Collection-Typen wie `Array`, `Map`, `Set`, `String`[3] etc. benutzen, um Dinge wie diese zu tun:

```
// Mit for-of über einen Iterator iterieren
for (let a of numbers) {
  // 1, 2, 3, etc.
}

// Einen Iterator per Spreading verwenden
```

```
let allNumbers = […numbers] // number[]

// Einen Iterator destrukturieren

let [one, two, …rest] = numbers // [number, number,
number[]]
```

Auch auf Iteratoren werden wir in diesem Buch nicht weiter eingehen. Weitere Informationen zu allgemeinen und asynchronen Iteratoren finden Sie im MDN (*https://mzl.la/2OAoy1o*).



TSC Flag: downlevelIteration

Wenn Sie Ihren TypeScript-Code in eine JavaScript-Version vor ES2015 kompilieren, können Sie eigene Iteratoren über das `downlevelIteration`-Flag in Ihrer *tsconfig.json* aktivieren.

Wenn Ihre Applikation Probleme mit der Bundle-Größe hat, sollten Sie `downlevelIteration` möglichst deaktiviert lassen: Damit eigene Iteratoren in älteren Umgebungen laufen, wird eine Menge Code benötigt. Das oben gezeigte `numbers`-Beispiel erzeugt fast 1 KB Code (mit gzip komprimiert).

Aufrufsignaturen (Call Signatures)

Bisher haben wir gelernt, Funktionsparameter und Rückgabewerte zu typisieren. Jetzt wollen wir einen höheren Gang einlegen und sehen, wie komplette Funktionen typisiert werden können.

Dafür werfen wir noch einmal einen Blick auf unsere `sum`-Funktion vom Anfang dieses Kapitels:

```
function sum(a: number, b: number): number {

  return a + b

}
```

Welchen Typ hat `sum`? Nun, `sum` ist eine Funktion, also ist ihr Typ:

```
Function
```

Wie Sie sich denken können, werden Sie in den meisten Fällen den Typ `Function` benutzen wollen. Wie `object` alle Objekte beschreibt, steht `Function` für alle Funktionen. Dabei sagt es erst einmal nichts über die tatsächliche Funktion aus, die hier typisiert wird.

Gibt es noch andere Wege für die Typisierung von `sum`? `sum` ist eine Funktion, die zwei Zahlen (vom Typ `number`) übernimmt und einen Wert vom Typ `number` zurückgibt. Daher können wir den Typ in TypeScript folgendermaßen ausdrücken:

```
(a: number, b: number) => number
```

Dies ist die TypeScript-Syntax für den Funktionstyp bzw. ihre *Aufrufsignatur* (oder *Typsignatur*). Die Ähnlichkeit mit der Pfeilschreibweise ist beabsichtigt! Wenn Sie Funktionen als Argumente übergeben oder aus anderen Funktionen zurückgeben, ist dies die übliche Syntax für die Typisierung.

Die Parameter `a` und `b` dienen hier nur der Dokumentation und haben keinen Einfluss auf die Zuweisbarkeit an eine Funktion dieses Typs.

Aufrufsignaturen von Funktionen enthalten ausschließlich Code auf Typebene (type level code), also nur Typangaben, aber keine Werte. Mit Aufrufsignaturen kann man also Parametertypen, `this`-Typen (siehe »this typisieren« auf Seite 52), Rückgabe-, Rest- und optionale Typen ausdrücken – aber keine Standardwerte (da ein Standardwert ein Wert ist und kein Typ). Und da sie keinen Körper haben, aus dem TypeScript etwas ableiten könnte, werden für Aufrufsignaturen explizite Annotationen der Rückgabetypen gebraucht.

Code auf Typebene und auf Werteebene

In Gesprächen über die Programmierung mit statischen Typen fallen oft Begriffe wie »Typebene« (type-level) oder »Werteebene« (value-level). Ein gemeinsamer Wortschatz kann helfen.

In diesem Buch verwende ich den Begriff *Code auf Typebene*, wenn es um Code geht, der ausschließlich aus Typen und Typoperatoren besteht. Alles andere gilt als *Code auf Wertebene*. Als Faustregel können Sie sich merken: Wenn es sich um gültigen JavaScript-Code handelt, haben Sie Code auf Werteebene vor sich. Handelt es sich dagegen um gültiges TypeScript, aber nicht um gültiges JavaScript, so ist es Code auf Typebene.[4]

Um ganz sicher zu gehen, hier noch ein Beispiel: Der Code auf Typebene ist hier fett hervorgehoben. Alles andere ist Code auf Werteebene:

```
function area(radius: number): number | null {

  if (radius < 0) {

    return null

  }

  return Math.PI * (radius ** 2)

}

let r: number = 3

let a = area(r)

if (a !== null) {

  console.info('result:', a)

}
```

Die fett gedruckten Begriffe auf Typebene sind die Typannotationen und der Vereinigungs-Typoperator (|). Alles andere sind Begriffe auf Werteebene.

Sehen wir uns ein paar Beispielfunktionen aus diesem Kapitel noch einmal an und verschieben wir ihre Typen in eigenständige Aufrufsignaturen, die wir an Typaliase binden:

```
// function greet(name: string)

type Greet = (name: string) => string
```

```
// function log(message: string, userId?: string)

type Log = (message: string, userId?: string) => void

// function sumVariadicSafe(…numbers: number[]): number

type SumVariadicSafe = (…numbers: number[]) => number
```

Merken Sie was? Die Aufrufsignaturen der Funktionen haben erstaunliche Ähnlichkeit mit ihren Implementierungen. Dieses Sprachmerkmal ist Absicht. Es soll den Umgang mit Aufrufsignaturen erleichtern.

Lassen Sie uns die Beziehung zwischen Aufrufsignaturen und ihren Implementierungen weiter vertiefen. Wie können Sie eine Funktion deklarieren, die eine bestimmte Aufrufsignatur implementiert? Sie könnten die Signatur schlicht mit einem passenden Funktionsausdruck kombinieren, wie im folgenden Beispiel, in dem wir unserer Log-Funktion eine nagelneue Aufrufsignatur verpassen:

```
type Log = (message: string, userId?: string) => void

let log: Log = ( ❶

  message, ❷

  userId = ‘Not signed in’ ❸

) => { ❹

  let time = new Date().toISOString()

  console.log(time, message, userId)

}
```

❶ Wir deklarieren einen Funktionsausdruck namens `log` und geben ihm explizit den Typ `Log`.

❷ Wir müssen unsere Parameter nicht zweimal annotieren. Da `message` bereits als Teil der Definition von `Log` als `string` annotiert ist, müssen wir das hier nicht noch einmal tun. Stattdessen lassen wir TypeScript den Typ aus `Log` ableiten.

❸ Wir fügen einen Standardwert für `userId` hinzu. Zwar können wir den Typ von `userId` aus `Log` ableiten, nicht aber den Standardwert, weil `Log` ein Typ ist und keine Werte enthalten kann.

❹ Unseren Rückgabetyp müssen wir nicht erneut annotieren, da wir ihn im `Log`-Typ bereits als `void` gekennzeichnet haben.

Kontextabhängige Typisierung

Das vorige Beispiel ist das erste, in dem wir die Typen unserer Funktionsparameter nicht explizit annotieren mussten. Da wir bereits festgelegt haben, dass `log` den Typ `Log` haben soll, konnte TypeScript aus dem Kontext ableiten, dass `message` den Typ `string` haben muss. Dieses mächtige Feature von TypeScripts Typinferenz wird als *kontextabhängige Typisierung* bezeichnet.

Ein weiterer Punkt, an dem die kontextabhängige Typisierung ein Thema ist, sind Callback-Funktionen, zum Beispiel:[5]

Zur Illustration deklarieren wir die Funktion `times`, die ihre Callback-Funktion `f` n-mal aufruft und ihr dabei den jeweils aktuellen Index von `f` übergibt:

```
function times(
  f: (index: number) => void,
  n: number
) {
  for (let i = 0; i < n; i++) {
```

```
    f(i)
  }
}
```

Deklarieren Sie eine eingebettete Funktion (»inline«), müssen Sie die an `times` übergebene Funktion beim Aufruf von `times` nicht explizit annotieren:

```
times(n => console.log(n), 4)
```

TypeScript leitet aus dem Kontext ab, dass `n` den Typ `number` hat. Wir haben in der Signatur für `times` bereits deklariert, dass das Argument `index` von `f` den Typ `number` haben soll. TypeScript ist schlau genug, zu erkennen, dass `n` dieses Argument ist, also auch den Typ `number` haben muss.

Hätten wir `f` nicht inline deklariert, hätte TypeScript den Typ nicht ableiten können:

```
function f(n) { // Error TS7006: Parameter ‘n’ implicitly
has an ‘any’ type.
  console.log(n)
}
times(f, 4)
```

Überladene Funktionstypen

Im vorigen Abschnitte haben wir folgende Syntax für den Funktionstyp verwendet: `type Fn = (…) => …`. Dies ist eine *Kurzschrift-Aufrufsignatur*. Das hätten wir auch ausführlicher schreiben können, wie hier noch einmal am Beispiel von `Log`:

```
// Kurzschrift-Aufrufsignatur
```

```
type Log = (message: string, userId?: string) => void

// Ausführliche Aufrufsignatur

type Log = {

  (message: string, userId?: string): void

}
```

Beide Deklarationen sind absolut gleichbedeutend und unterscheiden sich nur in der Schreibweise.

Brauchen wir die vollständige Aufrufsignatur dann überhaupt? Für einfache Fälle wie die `Log`-Funktion sollten Sie die Kurzschrift-Version bevorzugen. Bei komplexeren Funktionen gibt es jedoch viele gute Anwendungsfälle für ausführliche Aufrufsignaturen.

Einer dieser Fälle ist das *Überladen* eines Funktionstyps. Bevor wir ins Detail gehen, wollen wird klären, was es heißt, eine Funktion zu überladen.

Überladene Funktion

Eine Funktion mit mehreren Aufrufsignaturen.

Wenn Sie eine Funktion deklarieren, die bestimmte Parameter übernimmt und einen bestimmten Rückgabetyp besitzt, können Sie diese Funktion in den meisten Programmiersprachen mit den gleichen Parametern erneut aufrufen. Dabei wird der Rückgabetyp ebenfalls immer gleich bleiben. In JavaScript ist das anders. Da JavaScript eine sehr dynamische Sprache ist, kann es oft vorkommen, dass eine bestimmte Funktion auf viele verschiedene Arten aufgerufen wird. Hinzu kommt, dass der Rückgabetyp möglicherweise noch vom Typ des übergebenen Eingabe-Arguments abhängen kann!

Diese Dynamik (überladene Funktionsdeklarationen und Ausgabetypen, die vom übergebenen Eingabetyp abhängen) bildet TypeScript mithilfe seines statischen Typsystems nach. Für uns sind diese Sprachmerkmale möglicherweise selbstverständlich. Tatsächlich sind diese Möglichkeiten allerdings ein sehr weit fortgeschrittenes Sprachmerkmal für ein Typsystem!

Mithilfe überladener Funktionssignaturen können sehr ausdrucksstarke APIs entwickelt werden. Als Beispiel erstellen wir eine API zum Buchen von Reisen, die wir `Reserve` nennen wollen. Wir beginnen mit einer Skizze der benötigten Typen (und verwenden hier die ausführliche Schreibweise für die Signatur):

```
type Reserve = {
  (from: Date, to: Date, destination: string): Reservation
}
```

Danach können wir eine Implementierung für `Reserve` zusammenbauen:

```
let reserve: Reserve = (from, to, destination) => {
  // …
}
```

Will ein Benutzer eine Reise nach Bali buchen, ruft er die `reserve`-API mit einem `from`- und einem `to`-Datum (von-bis) und dem String “`Bali`” als Ziel (`destination`) auf.

Wir könnten die API so erweitern, dass auch Reisen ohne festes Enddatum gebucht werden können:

```
type Reserve = {
  (from: Date, to: Date, destination: string): Reservation
  (from: Date, destination: string): Reservation
}
```

Beim Versuch, diesen Code auszuführen, löst TypeScript dort einen Fehler aus, wo Sie versuchen, `Reserve` zu implementieren (siehe Abbildung 4-3).

```
     let reserve: Reserve = (from, to, destination) => {
       // ...
type }
  (f
  (f [ts] Type '(from: any, to: any, destination: any) => v
}    oid' is not assignable to type 'Reserve'.
     let reserve: Reserve
let reserve: Reserve = (from, to, destination) => {
  // ...
}
```

Abbildung 4-3: TypeError wegen einer fehlenden kombinierten Aufrufsignatur

Das hat mit der Funktionsweise von überladenen Signaturen in TypeScript zu tun. Deklarieren Sie mehrere Signaturen, die eine Funktion `f` überladen, ist der Typ dieser Funktion aus Sicht des *Aufrufers* die Vereinigungsmenge dieser Signaturen. Aus Sicht der *Implementierung* von `f` muss es jedoch einen *kombinierten* Typ geben, der tatsächlich implementiert werden kann. Diese kombinierte Aufrufsignatur muss bei der Implementierung von `f` manuell deklariert werden, sie wird nicht automatisch abgeleitet. Für unser `Reserve`-Beispiel können wir die `reserve`-Funktion wie folgt aktualisieren:

```
type Reserve = {
  (from: Date, to: Date, destination: string): Reservation
  (from: Date, destination: string): Reservation
} ❶
let reserve: Reserve = (
  from: Date,
  toOrDestination: Date | string,
```

```
  destination?: string
) => { ❷
  // …
}
```

❶ Wir definieren zwei überladene Funktionssignaturen.

❷ Die Signatur der Implementierung ist das Ergebnis der manuellen Kombination beider Überladungs-Signaturen (d.h., wir haben `Signatur1 | Signatur2` von Hand berechnet). Beachten Sie, dass die kombinierte Signatur dabei für Funktionen, die `reserve` aufrufen, nicht sichtbar ist. Aus Consumersicht lautet die Signatur von `Reserve`:

```
type Reserve = {
  (from: Date, to: Date, destination: string):
  Reservation
  (from: Date, destination: string): Reservation
}
```

Hier wird die von uns erstellte kombinierte Signatur nicht berücksichtigt:

```
// Falsch!
type Reserve = {
  (from: Date, to: Date, destination: string):
  Reservation
  (from: Date, destination: string): Reservation
```

```
    (from: Date, toOrDestination: Date | string,
      destination?: string): Reservation
}
```

Da `reserve` auf zwei Arten aufgerufen werden kann, müssen Sie TypeScript bei der Implementierung von `reserve` beweisen, dass Sie die Art des Funktionsaufrufs überprüft haben:[6]

```
let reserve: Reserve = (
  from: Date,
  toOrDestination: Date | string,
  destination?: string
) => {
  if (toOrDestination instanceof Date && destination !==
  undefined) {
    // Eine Reise ohne Wiederkehr
  } else if (typeof toOrDestination === 'string') {
    // Eine Rundreise
  }
}
```

Möglichst spezifische Überladungs-Signaturen definieren

Einfach gesagt, muss jede Überladungs-Signatur (z.B. `Reserve`) bei der Deklaration eines überladenen Funktionstyps an die Signatur der Implementierung zuweisbar sein (z.B. `reserve`). Sie können bei der Deklaration der Signatur also ziemlich allgemein vorgehen, sofern alle nötigen Überladungen zugewiesen werden können. Das folgende Beispiel funktioniert:

```
let reserve: Reserve = (

  from: any,

  toOrDestination: any,

  destination?: any

) => {

  // …

}
```

Beim Überladen von Funktionen sollte die Implementierungs-Signatur so spezifisch wie möglich sein, weil dies die Implementierung der Funktion erleichtert. `Date` sollte gegenüber `any` bevorzugt werden und (in unserem Beispiel) die Vereinigungsmenge von `Date | string` gegenüber `any`.

Warum erleichtern eng gefasste Typen die Implementierung einer Funktion mit einer bestimmten Signatur? Wenn Sie einen Parameter als `any` typisieren, ihn aber als `Date` verwenden wollen, müssen Sie TypeScript erst beweisen, dass es tatsächlich ein `Date` ist:

```
function getMonth(date: any): number | undefined {

  if (date instanceof Date) {

    return date.getMonth()

  }

}
```

Typisieren Sie den Parameter dagegen gleich als `Date`, können Sie sich die zusätzliche Arbeit bei der Implementierung sparen:

```
function getMonth(date: Date): number {

  return date.getMonth()

}
```

Viele Browser verwenden in ihren DOM-APIs Überladungen. Die `createElement`–DOM-API wird beispielsweise für die Erstellung von HTML-Elementen verwendet. Sie übernimmt einen String, der ein HTML-Tag repräsentiert, und gibt ein neues HTML-Element dieses Typs zurück. TypeScript besitzt eingebaute Typen für alle HTML-Elemente. Hierzu gehören beispielsweise:

- `HTMLAnchorElement` für `<a>`-Elemente
- `HTMLCanvasElement` für `<canvas>`-Elemente
- `HTMLTableElement` für `<table>`-Elemente
- etc.

Es liegt nahe, die Funktionsweise von `createElement` mit überladenen Aufrufsignaturen abzubilden. Überlegen Sie, wie Sie `createElement` typisieren würden. (Versuchen Sie, diese Frage vor dem Weiterlesen selbst zu beantworten!)

Die Antwort:

```
type CreateElement = {

  (tag: ‘a’): HTMLAnchorElement ❶

  (tag: ‘canvas’): HTMLCanvasElement

  (tag: ‘table’): HTMLTableElement

  (tag: string): HTMLElement ❷
```

```
}

let createElement: CreateElement = (tag: string):
HTMLElement => { ❸

  // …

}
```

❶ Wir haben den Parametertyp überladen, sodass er auf die jeweiligen Typen der Stringliterale passt.

❷ Wir haben eine Rückfallebene eingebaut: Übergibt der Benutzer einen Tag-Namen, für den es (noch) keine eingebaute TypeScript-Deklaration gibt, so geben wir ein generisches `HTMLElement` zurück.[7] TypeScript löst die Überladungen in der Reihenfolge ihrer Deklarierung auf. Wird ein String übergeben, für den keine spezifische Überladung definiert wurde, erzeugt TypeScript beim Aufruf von `createElement` ein Element mit dem allgemeinen Typ `HTMLElement`.

❸ Um alle Parameter der Implementierung zu typisieren, kombinieren wir alle Typen, die der Parameter in den Überladungs-Signaturen von `createElement` haben könnte zu `‘a’ | ‘canvas’ | ‘table’ | string`. Da alle drei Stringliteral-Typen Subtypen von `string` sind, reduziert sich der Typ einfach auf `string`.

In allen Beispielen dieses Abschnitts haben wir Funktionsausdrücke überladen. Wie aber können wir eine Funktions*deklaration* überladen? Wie üblich hilft TypeScript Ihnen auch hier mit einer passenden Syntax. Um das zu illustrieren, schreiben wir die Überladungen für `createElement` neu:

```
function createElement(tag: ‘a’):
HTMLAnchorElement

function createElement(tag: ‘canvas’):
HTMLCanvasElement
```

```
function createElement(tag: 'table'):
HTMLTableElement

function createElement(tag: string):
HTMLElement {

  // …

}
```

Am Ende ist es Ihre Entscheidung, welche Syntax Sie benutzen. Es kommt darauf an, welche Art von Funktion Sie überladen wollen (Funktionsausdrücke oder -deklarationen).

Vollständige Typsignaturen sind nicht darauf beschränkt, Funktionsaufrufe zu überladen. Sie können auch benutzt werden, um Eigenschaften an Funktionen zu modellieren. Da JavaScript-Funktionen einfach nur aufrufbare Objekte sind, können Sie ihnen Eigenschaften zuweisen, um etwa Folgendes zu tun:

```
function warnUser(warning) {

  if (warnUser.wasCalled) {

    return

  }

  warnUser.wasCalled = true

  alert(warning)

}

warnUser.wasCalled = false
```

Um die vollständige Signatur von `warnUser` zu typisieren, können wir TypeScript folgendermaßen verwenden:

```
type WarnUser = {
  (warning: string): void
  wasCalled: boolean
}
```

Danach können wir die Implementierung von `warnUser` als Funktionsausdruck neu schreiben:

```
let warnUser: WarnUser = (warning: string) => {
  if (warnUser.wasCalled) {
    return
  }
  warnUser.wasCalled = true
  alert(warning)
}
warnUser.wasCalled = false
```

Dabei merkt TypeScript von selbst, dass wir `wasCalled` erst nach der Deklaration der `warnUser`-Funktion zugewiesen haben.

Polymorphismus

Bisher haben wir uns in diesem Buch über das Wie und Warum konkreter Typen und Funktionen, die diese verwenden, gekümmert. Aber *was ist ein konkreter Typ*? Tatsächlich sind alle Typen, die wir bisher gesehen haben, konkret.

- `boolean`
- `string`
- `Date[]`
- `{a: number} | {b: string}`
- `(numbers: number[]) => number`

Konkrete Typen sind nützlich, wenn Sie genau wissen, welchen Typ Sie erwarten, und überprüfen wollen, dass genau dieser Typ auch übergeben wurde. Manchmal wissen Sie das aber nicht vorher. Trotzdem wollen Sie das Verhalten Ihrer Funktion nicht auf einen bestimmten Typ beschränken.

Als Beispiel implementieren wir die Funktion `filter`. Mit `filter` iterieren Sie über ein Array und sortieren dabei unerwünschte Elemente aus. In JavaScript könnte das etwa so aussehen:

```
function filter(array, f) {

  let result = []

  for (let i = 0; i < array.length; i++) {

    let item = array[i]

    if (f(item)) {

      result.push(item)

    }

  }
```

```
  return result
}

filter([1, 2, 3, 4], _ => _ < 3) // ergibt [1, 2]
```

Wir beginnen mit der Formulierung der vollständigen Typsignatur für `filter`. Für die Typen verwenden wir zunächst den Platzhalter `unknown`:

```
type Filter = {
  (array: unknown, f: unknown) => unknown[]
}
```

Dann versuchen wir, die Typen anzugeben, zum Beispiel als `number`:

```
type Filter = {
  (array: number[], f: (item: number) => boolean): number[]
}
```

Hier funktioniert die Typisierung der Arrayelemente als `number` recht gut. Aber `filter` ist eher als allgemeine Funktion gedacht. Sie soll Arrays mit Zahlen, Strings, Objekten, anderen Arrays – eigentlich allem – filtern können. Das geht mit unserer Signatur nicht. Sie »versteht« nur Zahlen. Probieren wir, die Funktion zu überladen, damit das Array auch Strings enthalten kann:

```
type Filter = {
  (array: number[], f: (item: number) => boolean): number[]
  (array: string[], f: (item: string) => boolean): string[]
```

```
}
```

Schon besser (allerdings wird eine Überladung für jeden möglichen Typ schnell unübersichtlich). Was passiert, wenn das Array Objekte enthält?

```
type Filter = {
  (array: number[], f: (item: number) => boolean): number[]
  (array: string[], f: (item: string) => boolean): string[]
  (array: object[], f: (item: object) => boolean): object[]
}
```

Das sieht auf den ersten Blick ganz in Ordnung aus. Um die Probleme zu finden, benutzen wir die Funktion einfach. Wenn Sie `filter` mit einer Signatur implementieren (also: `filter: Filter`) und versuchen, sie zu verwenden, passiert das hier:

```
let names =[
  {firstName: ‘beth’},
  {firstName: ‘caitlyn’},
  {firstName: ‘xin’}
]
let result = filter(
  names,
```

```
  _ => _.firstName.startsWith('b')

) // Error TS2339: Property 'firstName' does not exist on
type 'object'.

result[0].firstName // Error TS2339: Property 'firstName'
does not exist

                    // on type 'object'.
```

Inzwischen sollte klar sein, warum hier ein Fehler auftritt. Wir haben TypeScript mitgeteilt, dass wir ein Array mit Zahlen, Strings oder Objekten an `filter` übergeben. Wir haben ein Array mit Objekten übergeben. Wie Sie wissen, sagt `object` aber nichts über die Form des Objekts aus. Da wir TypeScript nicht mitgeteilt haben, welche Form das Objekt genau hat, wird ein Fehler ausgelöst, sobald wir versuchen, auf eine Objekteigenschaft zuzugreifen.

Was können wir tun?

Wenn Sie von einer Sprache kommen, die generische Typen unterstützt, verdrehen Sie vermutlich schon die Augen und rufen: »DAFÜR SIND GENERISCHE TYPEN DOCH DA!!11!«

Die gute Nachricht lautet: Sie haben vollkommen recht (die schlechte Nachricht ist, dass Sie mit Ihrem Geschrei gerade das Kind der Nachbarn geweckt haben).

Für den Fall, dass Sie keine Erfahrung mit generischen Typen haben, gebe ich Ihnen zuerst die Definition und dann ein Beispiel mit unserer `filter`-Funktion.

Generischer Typparameter

Ein Platzhalter-Typ, mit dem an mehreren Stellen eine Beschränkung auf Typebene erzwungen werden kann. Auch als *polymorpher Typparameter* bezeichnet.

Sehen wir uns nun an, wie unser `filter`-Beispiel aussieht, wenn wir es mit einem generischen Typparameter (T) schreiben:

```
type Filter = {

  <T>(array: T[], f: (item: T) => boolean): T[]
```

```
}
```

Im Prinzip sagen wir damit: »Die `filter`-Funktion verwendet einen generischen Typparameter T. Wir wissen noch nicht, welcher Typ das sein wird. Daher wäre es großartig, wenn du – TypeScript – den Typ bei jedem Aufruf von `filter` neu ableiten könntest.« TypeScript leitet T von dem Typ ab, den wir für `array` übergeben. Sobald es ermittelt hat, was T für einen bestimmten Aufruf von `filter` bedeutet, ersetzt es jedes T durch den tatsächlich abgeleiteten Typ. T ist ein Platzhalter-Typ, der je nach Kontext vom Typechecker ersetzt wird. Er *parametrisiert* den Typ von `Filter`. Daher kommt auch die Bezeichnung »generischer Typ-*Parameter*«.

Weil »generischer Typparameter« nicht besonders angenehm auszusprechen ist, kürzen viele Leute das als »generischer Typ« oder einfach »Generic« ab. Ich benutze diese Begriffe in diesem Buch gleichbedeutend und austauschbar.

Um generische Typparameter zu deklarieren, verwenden wir diese seltsamen spitzen Klammern (<>) (so eine Art `type`-Schlüsselwort für generische Typen). Die spitzen Klammern definieren den Geltungsbereich der generischen Typen (es gibt nur ein paar Orte, an denen sie benutzt werden können). TypeScript sorgt dafür, dass innerhalb dieses Geltungsbereichs alle Vorkommen des generischen Typs rechtzeitig an die entsprechenden konkreten Typen gebunden werden. Durch die Position der spitzen Klammern in unserem Beispiel bindet TypeScript bei einem Aufruf von `filter` konkrete Typen an unser generisches T. Innerhalb der spitzen Klammern können Sie, durch Kommas getrennt, beliebig viele generische Typparameter angeben.

T ist einfach ein Typname. Wir hätten auch einen beliebigen anderen Namen wählen können, wie A, `Z3br4` oder `Wurbelspunst`. Per Konvention werden jedoch einzelne Großbuchstaben verwendet, in alphabetischer Reihenfolge, beginnend mit T, je nachdem, wie viele generische Parameter gebraucht werden.

Wenn Sie viele Generics auf einmal definieren müssen oder auf komplizierte Weise verwenden, sollten Sie eventuell von der Konvention abweichen und beschreibendere Namen wie `Value` oder `WidgetType` verwenden.

Einige Programmierer bevorzugen A gegenüber T. Je nach Programmierhintergrund gibt es hier verschiedene Vorlieben: Anwender funktionaler Sprachen bevorzugen oft A, B, C etc. wegen der Ähnlichkeit zu den griechischen Buchstaben α, β und γ, die in mathematischen Beweisen zu finden sind. Anwender objektorientierter Sprachen tendieren eher zu T für »Typ«. Auch wenn TypeScript beide Programmierstile unterstützt, ist die zweite Form (T) hier deutlich weiter verbreitet.

So wie Funktionsparameter bei jedem Aufruf der Funktion neu gebunden werden, erhält auch jeder Aufruf von `filter` seine eigene Bindung für T:

```
type Filter = {
  <T>(array: T[], f: (item: T) => boolean): T[]
}
let filter: Filter = (array, f) => // …
// (a) T wird an number gebunden
filter([1, 2, 3], _ => _ > 2)
// (b) T wird an string gebunden
filter([‘a’, ‘b’], _ => _ !== ‘b’)
// (c) T wird an {firstName: string} gebunden
let names = [
  {firstName: ‘beth’},
  {firstName: ‘caitlyn’},
  {firstName: ‘xin’}
```

```
]

filter(names, _ => _.firstName.startsWith('b'))
```

TypeScript leitet diese generischen Bindungen von den Typen der übergebenen Argumente ab. Sehen wir uns Schritt für Schritt an, wie TypeScript T die Bindungen im Fall (a) vornimmt:

1. Aus der Typsignatur von `filter` weiß TypeScript, dass `array` ein Array mit Elementen des generischen Typs T ist.
2. TypeScript stellt fest, dass wir das Array `[1, 2, 3]` übergeben haben. Also muss T den Typ `number` haben.
3. Überall, wo TypeScript ein T findet, wird dies nun durch den Typ `number` ersetzt. Dadurch wird der Parameter `f: (item: T) => boolean` zu `f: (item: number) => boolean` und der Rückgabetyp `T[]` zu `number[]`.
4. TypeScript überprüft, ob alle Typen und auch die als `f` übergebene Funktion der frisch abgeleiteten Signatur zugewiesen werden können.

Generics bieten eine mächtige Möglichkeit, die Arbeitsweise einer Funktion allgemeiner zu beschreiben, als das mit konkreten Typen möglich wäre. Am besten stellt man sich Generics als *Beschränkungen* (constraints) vor. Wie die Annotierung eines Funktionsparameters als `n: number` den Wert des Parameters n auf den Typ `number` beschränkt, so sorgt die Verwendung eines generischen T dafür, dass überall, wo T steht, immer der gleiche Typ benutzt wird.

Generische Typen können auch in Typaliasen, Klassen und Interfaces benutzt werden. Wir machen in diesem Buch reichlich Gebrauch davon. Wenn wir weitere Themen behandeln, werde ich im Kontext darauf hinweisen.

Verwenden Sie Generics, wo immer das möglich ist. Sie helfen Ihnen, Ihren Code allgemein, wiederverwendbar und knapp zu halten.

Wann werden Generics gebunden?

Der Ort einer Typdeklaration definiert nicht nur den Geltungsbereich des Typs, sondern auch, wann TypeScript den konkreten Typ an das generische T bindet. Aus dem vorigen Beispiel:

```
type Filter = {

  <T>(array: T[], f: (item: T) => boolean): T[]

}

let filter: Filter = (array, f) =>

  // …
```

Da wir <T> als Teil der Aufrufsignatur deklariert haben (direkt vor der öffnenden runden Klammer), bindet TypeScript einen konkreten Typ an T, sobald eine Funktion des Typs `Filter` aufgerufen wird.

Hätten wir T dagegen im Geltungsbereich des Typalias `Filter` definiert, hätten wir den Typ bei der Verwendung von `Filter` explizit binden müssen:

```
type Filter<T> = {

  (array: T[], f: (item: T) => boolean): T[]

}

let filter: Filter = (array, f) => // Error TS2314: Generic
type ‘Filter’

  // …                            // requires 1 type
  argument(s).

type OtherFilter = Filter           // Error TS2314: Generic
type ‘Filter’

                                    // requires 1 type
argument(s).
```

```
let filter: Filter<number> = (array, f) =>
  // …
type StringFilter = Filter<string>
let stringFilter: StringFilter = (array, f) =>
  // …
```

Allgemein gesprochen, bindet TypeScript konkrete Typen an Ihre Generics, wenn sie verwendet werden: für Funktionen beim Aufruf, für Klassen bei ihrer Instanziierung (mehr hierzu unter »Polymorphismus« auf Seite 102) und für Typaliase und Interfaces (siehe »Interfaces« auf Seite 92), wenn Sie diese verwenden oder implementieren.

Wo können Generics deklariert werden?

Für jeden Weg, in TypeScript eine Aufrufsignatur zu deklarieren, gibt es auch eine passende Möglichkeit, einen generischen Typ hinzuzufügen:

```
type Filter = { ❶
  <T>(array: T[], f: (item: T) => boolean): T[]
}
let filter: Filter = // …
type  Filter<T> = { ❷
  (array:  T[], f: (item: T) => boolean): T[]
}
```

```
let filter: Filter<number> = // …

type Filter = <T>(array: T[], f: (item: T) => boolean) =>
T[] ❸

let filter: Filter = // …

type Filter<T> = (array: T[], f: (item: T) => boolean) =>
T[] ❹

let filter: Filter<string> = // …

function filter<T>(array: T[], f: (item: T) => boolean):
T[] { ❺

  // …

}
```

- ❶ Eine vollständige Aufrufsignatur, bei der der Geltungsbereich von T mit einer eigenen Signatur festgelegt wird. Da der Geltungsbereich von T auf eine einzelne Signatur beschränkt ist, bindet TypeScript T in dieser Signatur an einen konkreten Typ, wenn eine Funktion des Typs `filter` aufgerufen wird. Jeder Aufruf von `filter` erhält eine eigene Bindung für T.
- ❷ Eine vollständige Aufrufsignatur, wobei der Geltungsbereich für T sich auf *alle* Signaturen erstreckt. Da T als Teil des Typs `Filter` deklariert wurde (und nicht als Teil eines bestimmten Signaturtyps) führt TypeScript die Bindung von T durch, wenn eine Funktion des Typs `Filter` deklariert wird.
- ❸ Wie ❶, aber mit einer Kurzschrift-Aufrufsignatur.
- ❹ Wie ❷, aber mit einer Kurzschrift-Aufrufsignatur.
- ❺ Die Aufrufsignatur einer benannten Funktion, wobei der Geltungsbereich von T an die Signatur gekoppelt ist. Beim Aufruf von

`filter` bindet TypeScript den konkreten Typ an T. Jeder Aufruf von `filter` erhält seine eigene Bindung für T.

Als zweites Beispiel schreiben wir eine `map`-Funktion. Sie funktioniert so ähnlich wie `filter`. Anstatt aber Elemente aus einem Array zu entfernen, verwendet `map` eine Mapping-Funktion, um die Elemente umzuwandeln. Wir beginnen wieder mit einer Skizze der Implementierung:

```
function map(array: unknown[], f: (item: unknown) =>
unknown): unknown[] {

  let result = []

  for (let i = 0; i < array.length; i++) {

    result[i] = f(array[i])

  }

  return result

}
```

Vor der Umsetzung sollten Sie überlegen, wie Sie `map` generisch machen können, damit jedes Vorkommen von `unknown` durch einen bestimmten Typ ersetzt wird. Wie viele generische Typen brauchen Sie? Wie werden sie deklariert und wie wird die `map`-Funktion als Geltungsbereich festgelegt? Welche Typen sollen `array`, `f` und der Rückgabewert bekommen?

Bereit? Wenn Sie es noch nicht selbst ausprobiert haben, dann ist jetzt der richtige Moment. Sie können das. Bestimmt!

Also gut. Hier ist die Antwort (aber nicht meckern!):

```
function map<T, U>(array: T[], f: (item: T) => U): U[] {

  let result = []
```

```
    for (let i = 0; i < array.length; i++) {
      result[i] = f(array[i])
    }
    return result
  }
```

Wir brauchen genau zwei Typen: T für den Typ der empfangenen Arrayelemente und U für den Typ der ausgehenden Arrayelemente. Wir übergeben ein Array mit Ts sowie eine Mapping-Funktion. Diese übernimmt ein T und wandelt es in ein U um. Am Schluss geben wir ein Array mit Us zurück.

filter und map in der Standardbibliothek

Unsere Definitionen von `filter` und `map` haben eine erstaunlich große Ähnlichkeit mit denen, die TypeScript beiliegen:

```
interface Array<T> {
  filter(
    callbackfn: (value: T, index: number, array: T[]) => any,
    thisArg?: any
  ): T[]
  map<U>(
    callbackfn: (value: T, index: number, array: T[]) => U,
    thisArg?: any
  ): U[]
}
```

Wir haben Interfaces bis jetzt noch nicht behandelt, aber diese Definition besagt, dass `filter` und `map` Funktionen sind, die ein Array des Typs T übernehmen. Außerdem übernehmen beide eine Callback-Funktion (`callbackfn`) und einen Typ für `this` innerhalb der Funktion.

`filter` übernimmt ein T, dessen Geltungsbereich das gesamte `Array`-Interface ist. `map` verwendet ebenfalls T und fügt ein zweites Generic namens U hinzu, dessen Geltungsbereich auf die `map`-Funktion beschränkt ist. Dadurch bindet TypeScript bei der Erstellung eines Arrays einen konkreten Typ an T, der bei jedem Aufruf von `filter` oder `map` an dem Array für beide Funktionen gleich ist. Außerdem erhält `map` bei jedem Aufruf – neben dem bereits vorhandenen Zugriff auf das bereits gebundene T – zusätzlich eine eigene Bindung für U.

Viele Funktionen der JavaScript-Standardbibliothek sind generisch, besonders die für den `Array`-Prototyp. Arrays können Werte beliebigen Typs enthalten. Wir können diesen Typ T nennen und dann sagen: »`.push` übernimmt ein Argument des Typs T« oder »`.map` wandelt ein Array aus Ts in ein Array aus Us um«.

Generische Typableitung

Normalerweise kann TypeScript generische Typen recht gut für Sie ableiten. Wenn Sie die `map`-Funktion von vorhin aufrufen, leitet TypeScript ab, dass T den Typ `string` hat und U den Typ `boolean`:

```
function map<T, U>(array: T[], f: (item: T) => U): U[] {

  // …

}

map(

  [‘a’, ‘b’, ‘c’],  // Ein Array mit Ts

  _ => _ === ‘a’    // Eine Funktion, die ein U zurückgibt

)
```

Daneben ist es auch möglich, Generics explizit zu annotieren. Hierbei gilt: »Alles oder nichts!«. Entweder Sie annotieren jeden benötigten generischen Typ oder

keinen:

```
map     <string, boolean>(

  [‘a’, ‘b’, ‘c’],

  _ => _ === ‘a’

)

map     <string>( // Error TS2558: Expected 2 type
arguments, but got 1.

  [‘a’, ‘b’, ‘c’],

  _ => _ === ‘a’

)
```

TypeScript überprüft, dass jeder abgeleitete generische Typ dem entsprechenden explizit gebundenen generischen Typ zuweisbar ist. Falls nicht, wird ein Fehler ausgelöst:

```
// Okay, weil boolean an boolean | string zugewiesen werden
kann

map<string, boolean | string>(

  [‘a’, ‘b’, ‘c’],

  _ => _ === ‘a’

)

map<string, number>(
```

```
  [‘a’, ‘b’, ‘c’],

  _ => _ === ‘a’   // Error TS2322: Type ‘boolean’ is not
  assignable

)                  // to type ‘number’.
```

Da TypeScript die an die generische Funktion übergebenen Argumente verwendet, um die konkreten Typen für Ihre Generics abzuleiten, können gelegentlich Fälle wie dieser auftreten:

```
let promise = new Promise(resolve =>

  resolve(45)

)

promise.then(result => // Abgeleitet als {}

  result * 4 // Error TS2362: The left-hand side of an
  arithmetic operation must

)            // be of type ‘any’, ‘number’, ‘bigint’, or an
enum type.
```

Was ist da denn los? Warum hat TypeScript `result` als {} abgeleitet? Weil wir TypeScript nicht genug Informationen gegeben haben. Da TypeScript nur die Typen der Argumente einer generischen Funktion verwendet, um den Typ eines Generics zu ermitteln, hat es T hier mit dem Standardwert {} versehen!

Um das zu reparieren, müssen wir den generischen Typparameter von `Promise` explizit annotieren:

```
let promise = new Promise<number>(resolve =>
```

```
    resolve(45)
  )
  promise.then(result => // number
    result * 4
  )
```

Generische Typaliase

Wir kennen generische Typaliase bereits aus unserem `filter`-Beispiel weiter vorn in diesem Kapitel. Wenn Sie sich die Typen `Array` und `ReadonlyArray` aus dem vorigen Kapitel noch einmal ansehen (siehe »Immutable (schreibgeschützte) Arrays und Tupel« auf Seite 37), werden Sie feststellen, dass es sich dabei ebenfalls um generische Typaliase handelt! Jetzt wollen wir uns genauer mit der Verwendung von Generics in Typaliasen beschäftigen. Am besten geht das mit einem kurzen Beispiel.

Wir definieren den Typ `MyEvent`, der ein DOM-Event, z.B. einen Mausklick (`click`) oder eine gedrückte Maustaste (`mousedown`), beschreibt:

```
  type MyEvent<T> = {
    target: T
    type: string
  }
```

Dies ist übrigens der einzige gültige Ort, an dem ein generischer Typ in einem Typalias definiert werden kann: direkt nach dem Namen des Typalias und vor der Zuweisung (=).

Die `target`-Eigenschaft von `MyEvent` verweist auf das Element, an dem das Event ausgelöst wurde: ein `<button>`, ein `<div>` etc. Ein Button-Event

könnten wir beispielsweise so beschreiben:

```
type ButtonEvent = MyEvent<HTMLButtonElement>
```

Wenn Sie einen generischen Typ wie `MyEvent` verwenden, müssen Sie dessen Typparameter explizit binden. Er wird nicht automatisch für Sie abgeleitet:

```
let myEvent: Event<HTMLButtonElement | null> = {
  target: document.querySelector('#myButton'),
  type: 'click'
}
```

Sie können `MyEvent` auch verwenden, um einen anderen Typ zu erstellen, z.B. `Timed Event`. Wird das generische T in `TimedEvent` gebunden, bindet es TypeScript auch für `MyEvent`:

```
type TimedEvent<T> = {
  event: MyEvent<T>
  from: Date
  to: Date
}
```

Generische Typaliase können auch in einer Funktionssignatur verwendet werden. Wenn TypeScript einen konkreten Typ an T bindet, wird diese Bindung auch für `MyEvent` durchgeführt:

```
function triggerEvent<T>(event: MyEvent<T>): void {
```

```
  // …
}
triggerEvent({ // T hat den Typ Element | null
  target: document.querySelector('#myButton'),
  type: 'mouseover'
})
```

Das sehen wir uns am besten Schritt für Schritt an:

1. Wir rufen `triggerEvent` mit einem Objekt auf.
2. TypeScript sieht anhand der Funktionssignatur, dass das übergebene Argument den Typ `MyEvent<T>` haben muss. Es merkt außerdem, dass wir `MyEvent <T>` als `{target: T, type: string}` definiert haben.
3. TypeScript stellt fest, dass das `target`-Feld des übergebenen Objekts den Wert `document.querySelector('#myButton')` hat. Also muss T den gleichen Typ haben wie `document.querySelector('#myButton')`, nämlich: `Element | null`. Entsprechend wird T an den Typ `Element | null`. gebunden.
4. TypeScript ersetzt nun jedes Vorkommen von T durch `Element | null`.
5. TypeScript stellt sicher, dass alle unsere Typen zuweisbar sind. Das funktioniert, und der Typecheck für unseren Code war erfolgreich.

Begrenzter Polymorphismus

In diesem Abschnitt verwende ich einen Binärbaum als Beispiel. Machen Sie sich keine Sorgen, wenn Sie noch nicht mit binären Baumstrukturen gearbeitet haben. Für unsere Zwecke reicht es, wenn Sie Folgendes wissen:

- Ein binärer Baum ist eine Datenstruktur.
- Ein Binärbaum besteht aus Knoten (nodes).
- Ein Knoten enthält einen Wert und kann auf bis zu zwei Kindknoten verweisen.
- Ein Knoten kann einen von zwei Typen haben: einen *Blattknoten* (d.h., er hat keine Kindelemente) oder einen *inneren Knoten* (d .h., er hat mindestens ein Kindelement).

Manchmal ist es nicht genug, wenn man sagt: »Dieses Ding hat den generischen Typ `T`, und jenes Ding hat den gleichen Typ `T`.« Manchmal wollen Sie sagen können: »Der Typ U sollte *mindestens den Typ T* haben.« Das heißt, wir legen für U eine Obergrenze (upper bound) fest.

Wie lässt sich das umsetzen? Angenommen, wir implementieren einen Binärbaum. Dafür verwenden wir drei verschiedene Knoten-Typen:

1. Reguläre Baumknoten (`TreeNode`)
2. Blattknoten (`LeafNode`), d.h. Baumknoten (`TreeNode`) ohne Kindelemente.
3. Innere Knoten (`InnerNode`), d.h. Baumknoten (`TreeNode`) mit Kindelementen.

Wir beginnen mit der Deklaration unserer Typen:

```
type TreeNode = {
  value: string
}
type LeafNode = TreeNode & {
```

```
    isLeaf: true
  }
  type InnerNode = TreeNode & {
    children: [TreeNode] | [TreeNode, TreeNode]
  }
```

Das bedeutet, `TreeNode` ist ein Objekt mit einer einzelnen Eigenschaft: `value`. Der Typ `LeafNode` (Blattknoten) hat die gleichen Eigenschaften wie `TreeNode`, besitzt aber zusätzlich die Eigenschaft `isLeaf` (istBlattknoten), die grundsätzlich wahr (`true`) ist. `InnerNode` besitzt ebenfalls die Eigenschaften von `TreeNode`, hat aber zusätzlich die Eigenschaft `children`, die auf ein oder zwei Kindelemente verweist.

Jetzt wollen wir eine Funktion namens `mapNode` schreiben. Sie übernimmt einen `TreeNode`, wandelt ihn um und gibt einen neuen `TreeNode` zurück. Am Ende soll `map  Node` benutzt werden wie folgt:

```
  let a: TreeNode = {value: 'a'}
  let b: LeafNode = {value: 'b', isLeaf: true}
  let c: InnerNode = {value: 'c', children: [b]}
  let a1 = mapNode(a, _ => _.toUpperCase()) // TreeNode
  let b1 = mapNode(b, _ => _.toUpperCase()) // LeafNode
  let c1 = mapNode(c, _ => _.toUpperCase()) // InnerNode
```

Bevor Sie weitermachen, überlegen Sie, wie Sie eine `mapNode`-Funktion schreiben könnten. Sie soll einen Subtyp von `TreeNode` übernehmen und den gleichen Subtyp zurückgeben. Wird ein `LeafNode` übergeben, sollte auch ein

`LeafNode` wieder zurückgegeben werden, wird ein `InnerNode` übergeben, sollte der Rückgabewert ebenfalls ein `InnerNode` sein. Geht das überhaupt?

Hier die Antwort:

```
function mapNode<T extends TreeNode>( ❶
  node: T, ❷
  f: (value: string) => string
): T { ❸
  return {
    …node,
    value: f(node.value)
  }
}
```

❶ `mapNode` ist eine Funktion, die einen einzelnen generischen Typparameter T definiert. Die Obergrenze von T ist `TreeNode`. Das heißt, T muss entweder selbst ein `TreeNode` oder ein Subtyp davon sein.

❷ `mapNode` übernimmt zwei Parameter. Der erste ist ein Knoten des Typs T. In ❶ haben wir gesagt: `node extends TreeNode`. Übergeben wir etwas, das kein `TreeNode` ist – vielleicht ein leeres Objekt `{}`, `null` oder ein Array mit `TreeNode` –, würde unser Codeeditor das sofort mit einer roten Unterschlängelung quittieren. `node` muss entweder selbst ein `TreeNode` oder ein Subtyp von `TreeNode` sein.

❸ `mapNode` gibt einen Wert des Typs T zurück. Das heißt, T kann ein `TreeNode` oder einer seiner Subtypen sein.

Warum mussten wir T auf diese Weise deklarieren?

- Hätten wir T nur als T typisiert (und `extends TreeNode` weggelassen), hätte `mapNode` einen Kompilierungsfehler ausgelöst. TypeScript hätte uns mitgeteilt, dass `node.value` an einem ungebundenen `node` des Typs T nicht auf sichere Weise lesen kann (was, wenn der Benutzer eine Zahl übergibt?).
- Hätten wir T komplett weggelassen und `mapNode` als `(node: TreeNode, f: (value: string) => string) => TreeNode` deklariert, hätten wir nach der Umwandlung des Knotens Informationen verloren: `a1`, `b1` und `c1` wären nur noch einfache `TreeNode`s.

Indem wir sagen, `T extends TreeNode`, erhalten wir den spezifischen Typ des Eingabeknotens (`TreeNode`, `LeafNode` oder `InnerNode`) auch nach der Umwandlung.

Begrenzter Polymorphismus mit mehrfachen Beschränkungen (constraints)

Im vorigen Beispiel haben wir T mit nur einer Beschränkung versehen. T musste mindestens ein `TreeNode` sein. Was machen Sie aber, wenn Sie mehrere Typbeschränkungen benötigen?

Erweitern Sie einfach die Vereinigungsmenge (&) dieser Beschränkungen:

```
type HasSides = {numberOfSides: number}

type SidesHaveLength = {sideLength: number}

function logPerimeter< ❶

  Shape extends HasSides & SidesHaveLength ❷

>(s: Shape): Shape { ❸

  console.log(s.numberOfSides * s.sideLength)
```

```
  return s
}
type Square = HasSides & SidesHaveLength
let square: Square = {numberOfSides: 4, sideLength: 3}
logPerimeter(square) // Square, logs “12”
```

❶ `logPerimeter` ist eine Funktion, die ein einzelnes Argument `s` des Typs `Shape` übernimmt.

❷ `Shape` (Form) ist ein generischer Typ, der sowohl den Typ `HasSides` als auch den Typ `SidesHaveLengths` erweitert. Auf Deutsch: Eine Form (Shape) muss mindestens Seiten (Sides) haben, die ihrerseits eine Länge (Length) besitzen.

❸ Der Rückgabewert von `logPerimeter` entspricht exakt dem übergebenen Typ.

Begrenzten Polymorphismus für die Modellierung von Arität verwenden

Eine weitere Situation, in der begrenzter Polymorphismus verwendet wird, ist die Modellierung variadischer Funktionen (Funktionen, die eine beliebige Menge von Argumenten übernehmen). Als Beispiel wollen wir unsere eigene Version der JavaScript-eigenen Funktion `call` implementieren (`call` übernimmt eine Funktion und eine beliebige Anzahl von Argumenten und wendet diese dann auf die übergebene Funktion an).[8] Die Definition und Verwendung sehen Sie unten. Da wir die Typen später ergänzen, benutzen wir erst einmal den Typ `unknown`:

```
function call(
  f: (…args: unknown[]) => unknown,
  …args: unknown[]
): unknown {
```

```
  return f(…args)
}
function fill(length: number, value: string): string[] {
  return Array.from({length}, () => value)
}
call(fill, 10, ‘a’) // ergibt ein Array mit 10 ‘a’s
```

Jetzt wollen wir `unknown` durch die tatsächlichen Typen ersetzen. Folgende Beschränkungen sollen ausgedrückt werden:

- Die Funktion `f` soll eine Reihe von Argumenten T übernehmen und einen Wert vom Typ R zurückgeben. Wir wissen vorher nicht, wie viele Argumente übergeben werden.
- `call` übernimmt `f` mit den gleichen Argumenten vom Typ T wie `f` selbst. Auch hier ist die Anzahl der Argumente vorher nicht bekannt.
- `call` hat den gleichen Rückgabetyp wie `f`, nämlich R.

Wir brauchen zwei Typparameter: T, ein Array mit Argumenten, und R, das für einen beliebigen Rückgabewert steht. Ergänzen wir die Typen:

```
function call<T extends unknown[], R>( ❶
  f: (…args: T) => R, ❷
  …args: T ❸
): R  { ❹
  return f(…args)
```

```
}
```

Wie funktioniert das im Einzelnen? Am besten sehen wir uns das Schritt für Schritt an:

- ❶ `call` ist eine variadische Funktion (d.h., sie übernimmt eine beliebige Anzahl von Argumenten) mit zwei Typparametern: T und R. T ist ein Subtyp von `unknown[]`, d.h., T ist ein Array oder ein Tupel eines beliebigen Typs.
- ❷ Der erste Parameter von `call` ist eine Funktion `f`. `f` ist ebenfalls variadisch. Ihre Argumente haben den gleichen Typ wie `args`: Egal, welchen Typ `args` hat, die Argumente von `f` haben exakt den gleichen Typ.
- ❸ Zusätzlich zur Funktion `f` kann `call` per …`args` beliebig viele zusätzliche Parameter übernehmen. `args` ist als Restparameter definiert, der eine variable Anzahl von Argumenten beschreibt. Der Typ von `args` ist T, dabei muss T ein Subtyp von `Array` sein. (Hätten wir das vergessen anzugeben, hätte uns TypeScript dafür eine rote Linie gezeigt.) Dadurch leitet TypeScript, basierend auf den tatsächlichen Argumenten in `args`, einen *Tupel-Typ* für T ab.
- ❹ `call` hat den Rückgabetyp R. Dabei ist R an den jeweiligen Rückgabetyp von `f` gebunden.

Wenn wir `call` aufrufen, weiß TypeScript genau, was der Rückgabetyp ist, und wird sich beschweren, wenn wir die falsche Anzahl an Argumenten übergeben:

```
let a = call(fill, 10, ‘a’)      // string[]


let b = call(fill, 10)           // Error TS2554: Expected
3 arguments; got 2.


let c = call(fill, 10, ‘a’, ‘z’) // Error TS2554: Expected
3 arguments; got 4.
```

In »Typinferenz für Tupel verbessern« auf Seite 142 verwenden wir übrigens eine ähnliche Technik, um die TypeScript-Ableitung für Tupel zu verbessern.

Generische Standardtypen

Genauso, wie Sie Standardwerte für Funktionsparameter definieren, können Sie auch generische Typparameter mit Standardwerten versehen. Sehen wir uns hierzu noch einmal den Typ `MyEvent` aus »Generische Typaliase« auf Seite 74 an, den wir benutzt haben, um DOM-Events zu modellieren:

```
type MyEvent<T> = {
  target: T
  type: string
}
```

Um ein neues Event zu erstellen, mussten wir explizit einen generischen Typ an `MyEvent` binden. Dieser stand für ein bestimmtes HTML-Element, an dem das Event ausgelöst wurde:

```
let buttonEvent: MyEvent<HTMLButtonElement> = {
  target: myButton,
  type: string
}
```

Aus Bequemlichkeit und für Situationen, in denen wir das spezifische Element nicht kennen (an das `MyEvent` gebunden wird), können wir `MyEvent` mit einem Standardtypen versehen:

```
type MyEvent<T = HTMLElement> = {
  target: T
```

```
  type: string
}
```

Bei dieser Gelegenheit können wir auch gleich weiter vorn Gelerntes anwenden und versehen T mit einer Obergrenze, um sicherzustellen, dass T ein HTML-Element ist:

```
type MyEvent<T extends HTMLElement = HTMLElement> = {
  target: T
  type: string
}
```

Jetzt können wir auf einfache Weise ein Event erstellen, das zu Beginn auf kein bestimmtes `HTMLElement` beschränkt ist. Außerdem ist es bei der Erstellung des Events nicht mehr nötig, das T von `MyEvent` manuell an `HTMLElement` zu binden.

```
let myEvent: MyEvent = {
  target: myElement,
  type: string
}
```

Beachten Sie, dass generische Typen mit Standardwerten, wie optionale Funktionsparameter, nach den generischen Typen ohne Standardwerte angegeben werden müssen:

```
// Gut
```

```
type MyEvent2<
  Type extends string,
  Target extends HTMLElement = HTMLElement,
> = {
  target: Target
  type: Type
}

// Schlecht
type MyEvent3<
  Target extends HTMLElement = HTMLElement,
  Type extends string  // Error TS2706: Required type
  parameters may
> = {                  // not follow optional type
parameters.
  target: Target
  type: Type
}
```

Typgetriebene Entwicklung

Ein leistungsstarkes Typsystem bringt große Macht mit sich. Wenn Sie mit TypeScript programmieren, werden Sie oft eine »Typen zuerst«-Strategie verfolgen. Damit ist natürlich die *typgetriebene Entwicklung* gemeint.

> **Typgetriebene Entwicklung**
>
> Ein Programmierstil, bei dem Sie zuerst die Typsignaturen skizzieren und die Werte später ergänzen.

Statische Typsysteme existieren, um die Wertetypen, die Ausdrücke enthalten könnten, zu beschränken. Je differenzierter ein Typsystem ist, desto mehr sagt es über den im Ausdruck enthaltenen Wert aus. Wenn Sie ein ausdrucksstarkes Typsystem auf eine Funktion anwenden, teilt Ihnen die Typsignatur der Funktion bereits das meiste mit, was Sie über die Funktion wissen müssen.

Sehen wir uns hierzu noch einmal die Typsignatur der map-Funktion aus diesem Kapitel an:

```
function map<T, U>(array: T[], f: (item: T) => U): U[] {
  // …
}
```

Selbst wenn Sie die map-Funktion nie zuvor gesehen haben, können Sie schnell einen Eindruck davon bekommen, was map tut: Es übernimmt ein Array des Typs T und eine Funktion f, die ein T in ein U umwandelt. Rückgabewert ist ein Array des Typs U. Um das herauszufinden, müssen Sie nicht einmal die Implementierung der Funktion kennen![9]

Wenn Sie ein TypeScript-Programm schreiben, sollten Sie mit der Definition Ihrer Funktionssignaturen beginnen – *Typen zuerst* – und die Implementierungen später vornehmen. Indem Sie Ihr Programm zuerst auf Typebene skizzieren, sorgen Sie dafür, dass auf einer höheren Ebene alles einen Sinn ergibt, bevor Sie sich an die Implementierungen machen.

Bis jetzt haben wir das eher umgekehrt gemacht: Wir haben mit der Implementierung begonnen und daraus die benötigten Typen abgeleitet. Jetzt wissen Sie aber, wie Funktionen in TypeScript geschrieben und typisiert werden.

Daher schalten wir um und skizzieren die Typen zuerst und tragen die Details später nach.

Zusammenfassung

In diesem Kapitel ging es um die Deklaration und den Aufruf von Funktionen. Wir haben besprochen, wie Parameter typisiert werden und wie häufige Merkmale von JavaScript-Funktionen wie Standardparameter, Restparameter, Generatorfunktionen und Iteratoren in TypeScript ausgedrückt werden können. Wir haben uns damit beschäftigt, welcher Unterschied zwischen den Aufrufsignaturen und den Implementierungen von Funktionen besteht. Wir haben gesehen, was kontextuelle Typisierung ist, und die verschiedenen Möglichkeiten, Funktionen zu überladen, kennengelernt. Am Schluss haben wir uns eingehend mit Polymorphismus für Funktionen und Typaliase beschäftigt: warum Polymorphismus nützlich ist, wie und wo generische Typen deklariert werden, wie TypeScript generische Typen ableitet und wie sie mit Beschränkungen und Standardtypen versehen werden können. Wir haben das Kapitel mit einem kurzen Hinweis zur typgetriebenen Entwicklung beendet und erklärt, was das ist und wie Sie Ihr neues Wissen zu Funktionstypen dafür benutzen können.

Übungen

1. Welche Teile der Funktionssignatur einer Funktion leitet TypeScript ab: die Parameter, den Rückgabetyp oder beides?
2. Ist das JavaScript-Objekt `argument` typsicher? Falls nicht, was können Sie stattdessen verwenden?
3. Sie wollen einen Urlaub buchen, der sofort beginnt. Aktualisieren Sie die überladene `reserve`-Funktion aus diesem Kapitel (»Überladene Funktionstypen« auf Seite 60) mit einer dritten Aufrufsignatur, die nur ein Reiseziel ohne ein spezifisches Startdatum übernimmt. Aktualisieren Sie die Implementierung von `reserve`, um die neue überladene Signatur zu unterstützen.
4. [Schwer] Aktualisieren Sie unsere Implementierung von `call` aus diesem Kapitel (»Begrenzten Polymorphismus für die Modellierung von Arität verwenden« auf Seite 78), sodass sie nur für Funktionen nutzbar ist, deren zweites Argument den Typ `string` hat. Für alle anderen Funktionen sollte Ihre Implementierung in der Kompilierungsphase einen Fehler auslösen.

5. Implementieren Sie eine kleine typsichere Bibliothek zur Überprüfung von Zusicherungen (assertions) mit dem Namen `is`. Skizzieren Sie zuerst die nötigen Typen. Wenn Sie fertig sind, sollten Sie die Bibliothek benutzen können wie folgt:

```
// Zwei Strings vergleichen

is('string', 'otherstring') // falsch

// Zwei boolesche Werte vergleichen

is(true, false) // falsch

// Zwei Zahlen vergleichen

is(42, 42) // wahr

// Der Vergleich zweier unterschiedlicher Typen sollten
einen

// Fehler bei der Kompilierung auslösen

is(10, 'foo') // Error TS2345: Argument of type '"foo"' is
not assignable

             // to parameter of type 'number'.

// [Schwer] Es soll eine beliebige Anzahl

// Argumente übergeben werden können

is([1], [1, 2], [1, 2, 3]) // falsch
```

Index

Symbole

25–30

! (Ausrufungszeichen), Nicht-null-Zusicherung, Operator 152

? (Fragezeichen)

  für optionale Eigenschaften in Objekten 28

  für optionale Elemente in Tupeln 36

  für optionale Funktionsparameter 49

… (Ellipse) in variadischer Funktionsparameterliste 51

() (runde Klammern), Funktionsaufruf mit 52

[] (eckige Klammern)

  Enum-Werte auslesen 41

  Indexsignatur, Syntax 29

  Operator für schlüsselbasierten Zugriff (indexed access operator) 148

  in Tupel-Deklarationen 36

{} (geschweifte Klammern), objektliterale Typen definieren mit 26

@ts-check-Kommentar 242

@ts-ignore-Kommentar 250

@ts-nocheck-Kommentar 242

* (Asterisk, Sternchen), vor Funktionsnamen 54

/// (triple-slash directives) 287

& (Ampersand-Zeichen), für Schnittmengen-Typen 33, 134

– (Minus), Typoperator 141

+ (Plus), Typoperator 141

< > (spitze Klammern)

  generische Typparameter umgeben mit 68

  in Typzusicherungen 150

| (Pipe-Symbol), für Vereinigungstypen (union types) 33, 134, 192

A

abgebildete (mapped) Typen 139–141

  eingebaut 141

für Erstellung typsicherer Event-Emitter 187
abstrakte Klassen 88, 90
Erweiterung im Vergleich mit Implementierung von Interfaces 98
abstrakter Syntaxbaum (AST) 5
Ahead-of-Time-(AoT)Compiler (Angular) 209, 211
Aliase, für Typen *siehe* Typaliase
allowJs, TSC-Flag 241
ambiente Moduldeklarationen 239
Erstellen für Drittanbieter-JavaScript 250
ambiente Typdeklarationen 238
TODO als Typalias für any definieren 245
ambiente Variablendeklarationen 237
amd-module-Direktive 268
AMD-Modulstandard 219
JavaScript-Modul einbinden, das den AMD-Modulstandard verwendet 223
Angular 6/7 208–212
CLI installieren 209
Dienste 210
Komponenten 209
Projekt initialisieren 209
angularCompilerOptions 210
any (Typ) 19
Ableitung für leere Arrayelemente 35
als überladener Funktionsparameter-Typ 64
Erweiterung von Variablen, die als ‚null‘ oder ‚undefined‘ initialisiert wurden 125
JavaScript, Typen abgeleitet als 242
Objekt im Vergleich mit 26
TODO, ambiente Typdeklaration als Typalias für 245
Typzusicherungen als 150
APIs
für Interaktion von Datenbank-Applikationen 213
typsicher, für Verwendung mit Frameworks und TypeScript 212–213
apply-Funktion 52
Argumente
definiert 49

Funktionsaufrufe übergeben 49
variable oder feste Anzahl 50
Arität
Funktionen, variadisch, im Vergleich mit fester Arität 50
Modellierung anhand von begrenztem Polymorphismus 78
Arrays 34–38
Array (Typ), sicher erweitern 156
Array.prototype.includes (ES2016) 259
Filterung von 67
generische Funktionen am Array-Typ 73
homogen 35
Kovarianz in Array-Membern 120
Listen mit Funktionsparametern übergeben als 50
schreibgeschützt (read-only) 37
Syntax, T[] und Array<T> 35
Tupel 36
as (Typzusicherung) 150
as const 125
AST (abstrakter Syntaxbaum) 5
async und await, Syntax 176, 185
asynchrone Programmierung 175–189
async und await 185
asynchrone Streams Event-Emitter 186–189
Callbacks 178–180
Eventschleife in JavaScript 176–178
Promises 181–185
Aufrufsignaturen 57–59, 60
Deklaration von Generics als Teil von 70
Implementierung, Beispiel 59
in typgetriebener Entwicklung 81
Kurzschriftversion im Vergleich mit ausführlicher Schreibweise 60
überladen 61–66
Ausnahmen 161
auslösen 163–165
in Promises 184

JavaScript und 2
zurückgeben 165–167

B

Babel, mit TypeScript-Typechecking getesteter Code 258
Backend-Frameworks 213
bedingungsabhängige Typen 145–149
distributiv 145
eingebaut 148
infer-Schlüsselwort, Generics inline deklarieren 147
begrenzter Polymorphismus
mit mehrfachen Begrenzungen 78
zur Modellierung von Arität 78
Begrenzungen 19
benutzerdefinierter Typschutz (type guards) 143
Beschränkungen (constraints) 19
Generics als 70
bigint (Typ) 23
Binärbäume 76
bind-Funktion 52
Bindung
für generische Typparameter 69
von Generics, Zeitpunkt 70–73
Blattknoten (leaf nodes) 76
boolean (Typ) 17, 21
bottom type 39
Browser
Bevorzugung von Modulen gegenüber Namensräumen 229
JavaScript ausführen in, Polyfills und 258
JavaScript laden und ausführen 218
Modul-ladefähiges Format für 220
TypeScript-Code ausführen in 263
typsicheres Multithreading mit Web Workers 189–198
typsichere Protokolle 196
Überladungen in DOM-APIs 64

Browserify 219
Builder-Entwurfsmuster 110
build-Flag (TSC) 261
Build-Werkzeuge 264
Bytecode 5

C

Callbacks 178–180
    appendAndRead-Funktion, Beispiel 181
    Callback-Pyramiden 180
call-Funktion 52
catch-Klausel 183, 185
checkJs, TSC-Flag 242
classes, Bindung von Generics 71
classic-Modus (moduleResolution-Flag) 230
class-Schlüsselwort 90
CLI (Kommandozeilen-Interface) in Angular 209
    Installation 209
Client/Server-Kommunikation 212–213
Code 222
Codebeispiele in diesem Buch XVI
Code-Editoren 10
    für TypeScript einrichten 11–14
Coding-Stil in diesem Buch XIV
CommonJS, Modulstandard 219
    JavaScript-Modul einbinden, das CommonJS verwendet 223
    Probleme mit 220
CommonJS-Modul, Einstellung (TSC) 256
Companion-Objektmuster 141, 229
Compiler
    angularCompilerOptions 210
    AoT Angular-Compiler 211
    TypeScript-Compiler 5–7
composite (Compiler-Option) 260
const

Arraydeklarationen mit 35
Enum-Deklarationen mit 42
Probleme 42
Objekte deklarieren mit 26
Symbole deklarieren mit 25
Typdeklarationen mit 44
Variablendeklarationen mit 32
const-Werte 22
core-js, Bibliothek 258
createElement, DOM-API 64
CSS-Dateien, laden 240

D

.d.ts, Dateiendung (Typdeklarationen) 234
Dateisystem, Modulnamen als Dateipfade in 221
Datenbank, Interaktion von Applikationen mit 213
declaration (Compiler-Option) 260
declaration merging 277
declarationMap (Compiler-Option) 261
declare module-Syntax 239
declare-Schlüsselwort 234, 235
DefinitelyTyped
JavaScript mit Typdeklarationen auf 249
JavaScript ohne Typdeklarationen auf 249
definitive Zuweisungs-Zusicherungen 153
Deklarations-Maps 255
Deklarationsverschmelzung (declaration merging) 94, 229–230
Dekoratoren 106–108
Beispiel, @serializable-Dekorator 107
erwartete Typsignaturen für Dekorator-Funktionen 106
Verwendung durch Angular 210
Dependency Injector (Angular) 210
Dialog, schließen mit Nicht-null-Zusicherungen 151
Dienste (Angular) 210
Direktiven

Triple-Slash 267–269, 287
amd-Module 268
Typen 267
Distributivgesetz 145
DOM (Document Object Model) XIII
DOM-APIs in TSC-Compiler aktivieren 201
DOM-Typdeklarationen aktivieren 259
Events, in TypeScript-Standardbibliothek typisiert 188
Überladungen in Browser-DOM-APIs 64
DOM-Bibliothek für TSC 190
Don’t Repeat Yourself (DRY) 32
downlevelIteration, TSC-Flag 57
Drittanbieter-JavaScript, Verwendung 248–251
DRY (Don’t Repeat Yourself) 32
Duck Typing (nominelle Typisierung) *siehe auch* strukturelle Typisierung 25
dynamische Importe 222

E

Editoren 10
automatische Vervollständigung in 202
Eigenschaften
Funktionen, Modellierung mit kompletten Aufrufsignaturen 65
Objekt und Klasse, Kovarianz 119
Prüfung auf zusätzliche 125
Typableitung für JavaScript-Code 243
Zugriffsmodifier für 88
einmalige Symbole 25, 155
Either (Typ) 167
Engines (JavaScript) 7
Entwurfsmuster 109–112
Builder-Muster 110
Factory-Muster 109
Enums 40–44, 155
auf mehrere Deklarationen verteilen 41
const 42

mit abgeleiteten numerischen Werten, Beispiel 40
mit expliziten numerischen Werten, Beispiel 41
mit teilweise abgeleiteten Werten, Beispiel 41
Namenskonventionen 40
preserveConstEnums, TSC-Flag 42
Verwendung vermeiden 44
Zuweisbarkeit 123
ES2015 220
esModuleInterop, TSC-Flag 204, 224
esnext, Modulmodus 223
ESNext-Features 257
Event-Emitter 186, 189
Verwendung bei Multithreading mit Web Workers 191–196
Eventschleife 176–178
Event-Warteschlange 177
Executor 182
experimentalDecorators, TSC-Flag 106
Exporte
CommonJS 219
ES2015, Standard für 220–221
mit CommonJS- und AMD-Code 223
module.exports 220
Modulmodus im Vergleich mit Skriptmodus 224
Namensraum 225
extends-Option (TSC) 262
extends-Schlüsselwort 90

F

Factory-Entwurfsmuster 109
Fehler
Behandlung 161–174
Ausnahmen auslösen 163–165
Ausnahmen zurückgeben 165–167
in Promises 184
null zurückgeben 162

Option (Typ), Verwendung 167–173
Fehlermeldungen in TypeScript 3
in Callbacks 180
in JavaScript 2
in Promise-Implementierung 182
TypeScript-Fehler
TS1332 25
TS1353 23
TS2300 32, 95
TS2304 204, 237
TS2314 70, 148
TS2322 21, 23, 24, 27, 30, 36, 74, 127
TS2339 3, 26, 37, 42, 68, 144, 202
TS2345 3, 18, 34, 36, 43, 49, 99, 119, 121, 122, 126, 127, 138, 151, 156
TS2362 74
TS2365 3, 9
TS2366 133
TS2367 25
TS2393 227
TS2428 96
TS2430 94
TS2454 153
TS2469 24
TS2476 42
TS2511 88
TS2531 151
TS2532 28
TS2540 29
TS2542 37
TS2554 49, 51, 80
TS2558 73
TS2571 20
TS2684 54
TS2706 81
TS2717 95

TS2739 139
TS2741 27, 28
TS7006 60
TS7030 133
Überwachung in TypeScript-Projekten 262
Zeitpunkt des Auftretens, in JavaScript im Vergleich mit TypeScript 11
finale Klassen, Simulation 108
finally-Klausel 185
flussbasierte Typableitung 128
formale Parameter *siehe* Parameter
Formen
Regeln für Sub- und Supertypen 119
Formen (shapes)
Subtypen erstellen 118
Frameworks 201–215
Backend 213
Frontend 201–212
Angular 6/7 208–212
DOM-APIs, Aktivierung in tsconfig.json 201
React 203–208
typsichere APIs für 212–213
»frisches« Objektliteral (Typ) 126
fullTemplateTypeCheck, TSC-Flag 210
Funktionen 47–83, 142–145
benutzerdefinierter Typschutz (type guards) 143
Bindung an Generics 71
Deklaration und Aufruf 47–66
aufrufen 49
Aufrufsignaturen 57–59
Deklaration mit Syntax für benannte Funktionen 48
Generatorfunktionen 54
Iteratoren 55–57
kontextabhängige Typisierung 60
optionale und Standardparameter 49
Restparameter 50

Typisierung von this-Variable 52
überladene Funktionstypen 60–66
Dekorator 106
null, undefined, void und never als Rückgabetypen 38
Parametertypen 18
Polymorphismus 66–81
begrenzt 76–80
generische Standardtypen 80
Typableitung für Tupel verbessern 142
typgetriebene Entwicklung 81
überladene ambiente Funktionsdeklarationen 227
Varianz 120
kovariante Rückgabetypen 122
Funktionsausdrücke, sofort aufgerufene (IIFEs) 218
Funktionstyp 57

G

Generatorfunktionen 54
Generics 68
begrenzter Polymorphismus 76–80
Bindung, Zeitpunkt 70–73
Deklaration als Teil einer Bedingung 145, 147
Deklaration per Interface 95
Erstellung von Subtypen 117
Standardtypen 80
Typableitung (Inferenz) 73–74
Typaliase 75
Unterstützung durch Klassen und Interfaces 102–103
generische Typen 68
Promise 182
generische Typparameter
als Beschränkungen 70
kommagetrennte Liste in eckigen Klammern 69
Namenskonventionen 69
generische Typparameter *siehe auch* generische Typen 68

global.JSX, Namensraum 205
    globale Variablen (Browser) 220, 224
    globaler Namensraum 157, 228
    graduell typisierte Sprachen 8

H

Hilfsfunktionen für Typen 275
homogene Arrays 35
HTML-Templates (Angular) 210

I

immutable Arrays 38
implements-Schlüsselwort 96
import-Anweisungen 157, 223
Importe
    CommonJS 219
    CommonJS- und AMD-Code verwenden 223
    dynamisch 222
    ES2015-Standard für 220–221
    Import-Auslassung (elision) 267
    Modulmodus im Vergleich mit Skriptmodus 224
    Modulname für exakten Modulpfad 239
    untypisiert, Whitelisting 250
import-Funktion 223
index.ts 14
Indexsignaturen 29
    Record-Objekt 139
infer-Schlüsselwort 147
innere Knoten 76
instanceof, Operator 100
Instanzmethoden 90
Interfaces 92–98
    Angular, Lebenszyklus-Hooks 210
    Bindung von Generics 71
    Deklarationsverschmelzung (declaration merging) 94
    Implementierung 96

Implementierung im Vergleich mit Erweiterung abstrakter Klassen 98
kombinieren 229
Objekte, Klassen und andere Interfaces erweitern 94
Polymorphismus 103
Top-Level, in Typdeklarationsdateien 237
Vergleich mit Typaliasen 92
interne Triple-Slash-Direktive 288
interpretierte Sprachen 7
intrinsic elements (TSX) 291
Invarianz 119
isomorphe Programme 255
IterableIterator (Typ) 55
Iterables 55
Iteratoren 55–57
Definition 55
eingebaut, für häufige Collection-Typen 56
iterierbare Iteratoren 56

J

JavaScript XIII
Bundles für schnelleres Laden optimieren 264
eingebaute Iteratoren für Collection-Typen 56
Engine 7
Eventschleife 176–178
Kompilierungsziel für TypeScript-Projekte 255–259
Sprachversionen und Releases 257
Konfigurierung der Build-Pipeline für Kompilierung von TypeScript-Code in 264
Module, kurze Geschichte 218–220
neueste Syntax- und Sprachmerkmale XIV
TypeScript-Programm, Kompilierung zu 254
Typsystem, Vergleich mit TypeScript 8–11
Zusammenarbeit mit 233–251
Möglichkeiten für Verwendung von JavaScript aus TypeScript heraus 251
schrittweise Code-Migration zu TypeScript 240–246
Typdeklarationen verwenden 234–240

Typermittlung für JavaScript 246–248
Verwendung von Drittanbieter-Code 248–251
JSDoc-Annotationen 243
JSON (JavaScript Object Notation) Laden von .json-Dateien 240
JSX (JavaScript XML)
Verbindung von React zu TSX für korrekte Typisierung von JSX 292
JSX (JavaScript-XML) 203
TSX mit React verwenden 205–208
TSX und 204
jsx-Direktive 205

K

Kanäle, Events senden auf 186
keyof-Operator 136
keyofStringsOnly, TSC-Flag 138
Klassen 85–92
Deklaration, Typdeklarationen und 235
Dekoratoren 106–108
Entwurfsmuster 109–112
Builder-Muster 110
Factory-Muster 109
finale Klassen simulieren 108
Interfaces implementieren 96
im Vergleich mit Erweiterung abstrakter Klassen 98
Kovarianz in Klassen-Membern 120
Mixins 103–106
Polymorphismus 102
strukturelle Typisierung 98
super-Aufrufe 90
this als Rückgabetyp, Verwendung 91
und Vererbung 85
abstrakte Klassen 88
Zugriffsmodifier für Eigenschaften und Methoden 88
Werte und Typen deklarieren 99
Knoten (Binärbaum) 76

Kompilierungsziel für TypeScript-Projekte 255–259
target, Einstellung 256
Komponenten
Angular 209
React 203
Funktions- und Klassenkomponenten 205
konkrete Typen 66
Bindung an generische Typparameter 69
Konstruktoren
erweiterte Klassenkonstruktoren 107
Klassen, Mixins und 104
private Klassenkonstruktoren 108
Sicherheitsprobleme mit Funktionskonstruktoren 48
super-Aufrufe 91
Konstruktor-Signaturen 101
kontextuelle Typisierung 60
Kontravarianz 120
in Funktionsparametern und this-Typen 123
Kovarianz 119
in Rückgabetypen von Funktionen 122

L

LABjs, Module laden mit 219
Code nur bei Bedarf laden (lazy loading) 222
Laufzeitumgebung (Runtime) 5
Ausnahmen 161
Lebenszyklus-Hooks (Angular) 210
let
const verwenden anstelle von 22
Typdeklarationen mit 44
Variablendeklarationen mit 32
lib, tsconfig.json-Option 202
lib-Direktive 287
lib-Einstellung (TSC) 238, 256, 258
Listener, Erstellung 188

onmessage-API in Web Workers 191

M

MatrixProtocol 196
Message-Passing (Web Workers) 190
Message-Passing-API 190
onmessage-API 191
Methoden 90
abstrakt 88
Zugriffsmodifier für 88
Mixins 103–106
Module 217–225
ambiente Moduldeklarationen 239
amd-module-Direktive 268
aus JavaScript importiert, Typermittlung für 246–248
Bevorzugung gegenüber Namensräumen 228
Import, Export 220–221
CommonJS- und AMD-Code verwenden 223
dynamische Importe 222
Modulmodus im Vergleich mit Skriptmodus 224
kurze Geschichte der JavaScript-Module 218–220
module-Feld, tsconfig.json 241
per NPM installiertes Drittanbieter-Modul, TypeScript informieren über 236
untypisiert, Import aus Drittanbieter-JavaScript 249
Ziel-Modulsystem für Kompilierung durch TypeScript wählen 263
Module asynchron bzw. bei Bedarf laden (lazy loading) 218
Module asynchron, bzw. bei Bedarf laden (lazy loading)
Codeteile nur bei Bedarf laden 222
module, Einstellung (tsconfig.json) 228
module.exports 220
module-Einstellung (TSC) 256
moduleResolution, TSC-Flag 230
modules
[Schreiben von] Deklarationsdateien für JavaScript-Module von Drittanbietern 279–285
Modulmodus im Vergleich mit Skriptmodus 224

Moment.js 222
Multithreading
im Browser mit Web Workers 189–198
typsicheres Multithreading in NodeJS mit Kindprozessen 198–199

N

Namensräume 225–229
Bevorzugung von Modulen gegenüber 228
Exporte 225
Kollisionen 227
Kombinieren 229
kompilierte Ausgabe 228
lang, Verwendung von Aliasen zur Verkürzung 226
untergeordnete 226
namespace-Schlüsselwort 225
never (Typ) 38, 147
Verwendung, Zusammenfassung 39
new-Operator 101
new(…any[])-Syntax für erweiterte Klassenkonstruktoren 107
next-Methode 56
Nicht-null-Zusicherungen 150
Node-API für Redis 187
NodeJS
callback-basierte fs.readFile-API 178–180
Callback-basierte fs.readFileAPI mit Promise-basierter API umgeben 182
EventEmitter 186, 193
Installation und Ausführung 11
Module 219
Module gegenüber Namensräumen bevorzugen 229
TypeScript-Code ausführen in 263
typsicheres Multithreading mit Kindprozessen 198–199
node-Modus (moduleResolution-Flag) 230
noImplicitAny-Flag in tsconfig.json 20
noImplicitAny-Flag in tsconfig.sh 242
noImplicitReturns, TSC-Flag 133

noImplicitThis TSC-Option 54
no-invalid-this, TSLint-Regel 52
nominelle Typisierung 25
Notausgänge 149–154
definitive Zuweisungs-Zusicherungen 153
Nicht-null-Zusicherungen 150
nominelle Typen simulieren 154–156
Typ-Zusicherungen 149–150
NPM Package-Manager 11
Installation von Drittanbieter-JavaScript 248
Typdeklaration senden an 250
TypeScript veröffentlichen über 265–267
null (Typ) 38
als Rückgabewert von Funktionen 162
Nicht-null-Zusicherungen 150
Probleme mit, in verschiedenen Programmiersprachen 40
strikte Überprüfung auf Nullwerte 39
Variablen initialisiert mit, Typerweiterung 125
Verwendung, Zusammenfassung 39
Verwendungsbeispiel 38
number (Typ) 17, 22

O

Objekte 25–30, 134–142
abgebildete (mapped) Typen 139–141
Companion, Objektmuster 141
Deklaration in TypeScript 30
Eigenschaften 28
Filterung von Arrays mit 67
Form 27
Indexsignatur 29
invariante Eigenschaftstypen in einigen Sprachen 120
kovariante Member 118
leer 29
Record (Typ) 138

Typoperatoren für 134–138
keyof-Operator 136
schlüsselbasierter Zugriff (indexed access operator) 134
Objektliterale 26
»frischer« literaler Objekttyp 126
objektrelationale Mapper (ORMs) 214
Observables 186
Operationen verketten
in Callbacks 180
in Promises 183
Operator für schlüsselbasierten Zugriff 134
Option (Typ) 167–173
flatMap- und getOrElse-Methoden 169
Implementierung in Some<T>- und None-Klassen 169
Implementierung von flatMap und get OrElse für Some<T> und None 170
Option-Funktion implementieren 172
Verwendung überladener Signaturen, um flatMap mit spezifischeren Typen zu versehen 171
Ordnerstruktur für TypeScript-Projekte 14, 253
ORMs (objektrelationale Mapper) 214
outFile, TSC-Flag 264

P

Parallelismus *siehe auch* Multithreading 189
Parameter
definiert 49
generischer Typ 68
Kompatibilität von Argumenten mit 49
Kontravarianz in Funktionsparametern 122
optional und Standard 49
optionale Funktionsparameter bei Typableitung in JavaScript 243
Rest 50
Typ 18
path-Direktive 287
Pfade (Module) 221, 223, 230
Polyfills 256

für process.env.NODE_ENV 237
für Zielumgebung ohne Unterstützung f. neuere Bibliotheks-Features 258
Polymorphismus 66–81
begrenzt 76–80
mit mehrfachen Beschränkungen 78
Verwendung zur Modellierung von Arität 78
Bindung von Generics 70–73
generische Standardtypen 80
generische Typableitung 73–74
generische Typaliase 75
generische Typparameter 68
Klassen und Interfaces 102–103
preserveConstEnums, TSC-Flag 42
private (Zugriffsmodifier) 88
Klassen mit privaten Feldern 99
Klassenkonstruktoren 108
Programmierstil, Konfigurierung für TSLint 13
Projekte, für TypeScript einrichten 14
Projektreferenzen 260–262
Promises 181–185
async und await verwenden mit 185
Callback-basierte NodeJS-API fs.read-File, mit Promise-basierter API umgeben 182
Promise, Implementierung 181
Promises ablehnen, mit Fehlermeldung 184
Promises mit then und catch verketten 183
Promise-Implementierung generischer Typ 182
von import-Funktion zurückgegeben 223
protected (Zugriffsmodifier) 88
Klassen mit geschützten Feldern 99
Protokolle
Client/Server-Kommunikation über 212
typsicher 196, 212
Prototypen, sicher erweitern 156–158
Prüfung auf zusätzliche Eigenschaften 125
public (Zugriffsmodifier) 88

R

React 203–208
- JSX, Überblick 203
- TSX- oder JSX-Unterstützung in TypeScript 204
- TSX verwenden mit 205
- Verbindung zu TSX für sichere Typisierung von JSX 292

ReadonlyArray (Typ) 37
readonly-Modifier 29
- Verwendung mit Arrays und Tupeln 37

reaktive Programmierungs-Bibliotheken 186
Record (Typ) 138
Redis-Bibliothek, Verwendung 187
references (Compiler-Option) 261
Referenzen XV
require-Aufrufe 220
Restparameter 50
return-Anweisungen, Überprüfung auf Vollständigkeit 132
rollenbasierte Programmierung 103
Rückgabetypen
- Annotationen 48
- Kovarianz in 122
- this-Variable 91

RxJS-Bibliothek 186

S

safety flags (TSC) 289
schlüsselbasierter Zugriff, Operator für 148
Schnittmengen-Typen 32, 94
- array 35

schwach typisierte Sprachen 9
selbstkompilierende (self-hosting) Compiler 11
Skriptmodus im Vergleich mit Modulmodus 224
- Skriptmodus für Typdeklarationen 237

Sourcemaps 254
- aktivieren 259

in NodeJS-Prozess integrieren 263
Speichersicherheit 190
SQL-Aufrufe 213
src/index.ts 14
Standardwerte
für Funktionsparameter 49
generischer Typ 80
strictBindCallApply-Option 52
strict-Flags in TSC 20, 289
strictFunctionTypes, TSC-Flag 123
strict-Modus (TSC) 242
strictNullChecks und strictPropertyInitialization TSC-Flags 87
strictNullChecks-Option 39
string (Typ) 17, 24
Strings
Array aus, filtern 67
automatische Typumwandlung in JavaScript 9
strukturelle Typisierung 25
Klassen in TypeScript 98
Subtypen 116
Erstellung komplexerer Typen, Subtypen-Regeln 117
Formen und Arrays 118
Zuweisbarkeit und 123
Super-Aufrufe 90
Supertypen 116
Objekttyp anstelle von erwartetem Supertyp verwenden 118
Zuweisbarkeit und 123
symbol (Typ) 24
Verwendungsbeispiel für einmaliges Symbol 25
Symbol.iterator 56
Syntax für benannte Funktionen 48

T

Tags für Vereinigungstypen 131
target, Einstellung (TSC) 256

Tasks 177
tatsächliche Parameter *siehe* Argumente
Tests, zum Auffinden von Fehlern 3
then-Klausel 183
  async und await stattdessen verwenden 185
this-Variable 52
  Kontravarianz in this-Typen von Funktionen 122
  Typisierung 52
  Verwendung als Rückgabetyp für Methoden 91
Totalität (Vollständigkeitsüberprüfung) 132–134
Traits *siehe auch* Mixins 103
Transkompilierung 256
  Code für ältere JavaScript-Versionen, TSC-Unterstützung für Transkompilierung 256
  JavaScript-Features, nicht von TSC transkompiliert 256
Triple-Slash-Direktiven 267–269, 287–288
  amd-module 268
  Typen 267
Try (Typ) 167
try/catch-Anweisungen 163, 183
TSC-Compiler 6, 11
  checkJs-Flag 242
  declarations-Flag 235
  DOM-APIs aktivieren 201
  DOM-Bibliothek aktivieren für 190
  downlevelIteration-Flag 57
  Einstellungen für Zielumgebung 256
  erzeugte Artefakte 254
  esModuleInterop-Flag 204, 224
  experimentalDecorators-Flag 106
  fullTemplateTypeCheck-Flag 210
  Installation mit npm 11
  JavaScript-Bundles erzeugen 264
  JavaScript-Code kompilieren 241
  jsx-Flag 204
  keyofStringsOnly-Flag 138

Konfigurierung mit tsconfig.json 12
lib, Einstellung 238
moduleResolution-Flag 230
Namensräume, Ausgaben 228
noImplicitAny-Flag aktivieren 242
noImplicitReturns-Flag 133
noImplicitThis-Option 54
preserveConstEnums-Option 42
Projektreferenzen 260–262
safety flags 289
strictBindCallApply-Option 52
strictFunctionTypes-Flag 123
strictNullChecks- und strictPropertyInitialization-Flags 87
strictNullChecks-Option 39
stringentere Flags für migrierten JavaScript-Code, Verwendung 245
types- und typeRoots-Einstellungen 247
webworker-Bibliothek aktivieren für 190
tsconfig.json 12
lib-Option 202
module:esnext 223
noImplicitAny-Flag aktivieren 20
Optionen 12
TSLint
Installation 11
Konfigurierung mit tslint.json 13
no-angle-bracket-type-assertion (Regel) 150
no-invalid-this-Regel 52
tslint.json 13
TSX
Referenz 291–293
Verwendung mit React 205–208
Tupel 36
optionale Elemente 36
read-only 37
Restelemente 37

Typableitung verbessern für 142
Typableitung (Inferenz)
verbessern für Tupel 142
Typaliase 31
Bindung von Generics 71
für Namensräume 226
generisch 75
Vergleich mit Interfaces 92
Typdeklarationen 234–240, 255
ambiente Moduldeklarationen 239
ambiente Typdeklarationen 238
ambiente Variablendeklarationen 237
Beispiel mit gleichwertigem TypeScript-Code 234
DOM-Typdeklarationen, aktivieren 259
Erstellung für Drittanbieter-JavaScript und Bereitstellung über NPM 250
in JavaScript auf DefinitelyTyped 249
JavaScript-Code ohne, auf DefinitelyTyped 249
Unterschiede zu regulärer TypeScript-Syntax 234
Verwendung von Typdeklarationsdateien 236
Typebene, Code auf 58
Type-Branding 155
Typechecker 6, 17
Typechecking 6
für Angular-Templates 210
JavaScript im Vergleich mit TypeScript 10
JavaScript-Code 242
Typen
ableiten (Inferenz) 8
bedingungsabhängig 145–149
distributive Bedingungen 145
eingebaut 148
inline-Deklaration von Generics mit infer-Schlüsselwort 147
Beziehungen zwischen 115–132
Subtypen und Supertypen 116
Typerweiterung 124–125

Varianz 117–123
Verfeinerung 128–132
Zuweisbarkeit 123
Bindung, JavaScript im Vergleich mit TypeScript 8
Definition 17
Deklarationen, Erzeugung 277
DOM-Deklaration aktivieren 202
erweitern *siehe* Typerweiterung
fortgeschrittene Objekttypen 134–142
abgebildete (mapped) Typen 139–141
Companion-Objektmuster 141
Record 138
Typoperatoren 134–138
Funktion 142–145
benutzerdefinierter Typschutz (type guards) 143
Typableitung für Tupel verbessern 142
in TypeScript unterstützt,
Typen und ihre Subtypen 44
Klassen, gemeinsame Deklaration von Typen und Werten durch 99
Notausgänge 149–154
benannte Typen simulieren 154–156
definitive Zuweisungs-Zusicherungen 153
Nicht-null-Zusicherungen 150
Typ-Zusicherungen 149–150
Prototypen sicher erweitern 156–158
Terminologie und Vokabular 18
Top-Level, in Typdeklarationsdateien 237
Totalität, Vollständigkeitsüberprüfung 132–134
Typen und Werte kombinieren 229
Typermittlung für JavaScript 246–248
Umwandlungen, JavaScript im Vergleich mit TypeScript 9
Unterstützung durch TypeScript 19–44
any 19
Arrays 34–38
bigint 23

boolesche 21
null, undefined, void und never 38–40
number 22
Objekte 25–30
string 24
symbol 24
Tupel 36
Typaliase 31
unknown 20
Vereinigungs- und Schnittmengentypen 32
Variable mit this-Typ deklarieren 53
zuerst, Faustregel 82
zur Skizzierung von Programmen 271
typeof-Operator 100, 144
Verwendung mit Arrayelementen 35
Typerweiterung 124–125
Prüfung auf zusätzliche Eigenschaften 125
types- und typeRoots-Einstellungen (TSC) 247
TypeScript
Ausführung auf dem Server 263
Bereitstellung von Code über NPM 265–267
Code-Editoren einrichten 11–14
Compiler 5–7
ein Projekt erstellen 253–262
Fehlerbehandlung 3
im Browser ausführen 263
Module, einbinden und exportieren 220
Projekt erstellen
Fehlerberichte 262
Kompilierungsziel angeben 255–259
Projektlayout 253
Projektreferenzen 260–262
Sourcemaps, Aktivierung 259
von TSC erzeugte Artefakte 254
Projekte einrichten, index.ts-Datei 14

schrittweise Migration von JavaScript-Code zu 240–246
Dateiendungen ändern in .ts 244
JSDoc-Annotationen hinzufügen 243
TSC zu JavaScript-Projekten hinzufügen 241
TSC-strict-Flags, Verwendung 245
Typechecking für JavaScript aktivieren 242
Triple-Slash-Direktiven 267–269
Typsystem 7–11
Vergleich mit JavaScript 8–11
types-Direktive 267
TypeSearch 249
typgetriebene Entwicklung 81
Typnamen, frei wählbar 69
Typoperatoren 273
– (Minus) und + (Plus) 141
für Objekttypen 134–138
keyof 136
Operator für schlüsselbasierten Zugriff (indexed access operator) 134
Typschutz (type guards) 144
Typsicherheit 1
Typsignaturen *siehe* Aufrufsignaturen Typsysteme 7–11
TypeScript-Typsystem 271
Vergleich von JavaScript mit TypeScript 8–11
Typumwandlungen 9
Typverfeinerung *siehe* Verfeinerung (Typen)

U

überladene ambiente Funktionsdeklarationen 227
überladene Funktionstypen 60–66
filter-Funktion 67
überladene Signaturen spezifisch halten 63
undefined (Typ)
Nicht-null-Zusicherungen 150
Variablen initialisieren mit, Typerweiterung 125
Verwendung, Zusammenfassung 39

Verwendungsbeispiel 38
ungültige Aktionen 2
unknown (Typ) 20

V

var
const verwenden anstelle von 22
Typdeklarationen mit 44
Variablendeklarationen mit 32
Variablen
ambiente Variablendeklarationen 237
Deklarationen, ambiente, im Vergleich zu regulären 236
immutabel, keine Typerweiterung 124
mit explizit annotierten Typen 7
mit implizit abgleiteten Typen 8
mutabel, Typerweiterung 124
variadische Funktionen 50
Varianz 117–123
Arten von 120
Form und Array 118
Funktion 120
Vereinigungstypen 32
bedingunsabhängigen Typ hinzufügen 146
Definition für Kommunikation zwischen Threads 193
unterschieden 130
Vererbung 85
mehrfache Vererbung simulieren 103
Verfeinerung (Typen) 128–132
typeof-Operator und Wechsel des Geltungsbereichs (scope) 144
unterschiedene Vereinigungstypen 130
View-Schicht bei Frontend-Applikationen (React) 203
void (Typ) 38
Verwendung, Zusammenfassung 39
Vollständigkeitsüberprüfung *siehe auch* Totalität 132
VSCode 11

W

wahre und falsche Werte (boolescher Typ) 21
Web Workers, typsicheres Multithreading mit 189–198
  Message-Passing-API 190
  onmessage-API 191
  snowflake-API hinter EventEmitterbasierter API abstrahieren 193
  typsichere Protokolle 196
Website zum Buch XVII
Wert: Typ-Annotationen 7
wertbasierte Elemente (TSX) 291
Werte XV
  erzeugt durch Deklaration 277
  Klassen, gemeinsame Deklaration von Werten und Typen durch 99
  null und undefined 38
  Top-Level, in Typdeklarationsdateien 237
  Typdeklarationen und 234
  Werte und Typen kombinieren 229
Wertebene, Code auf 58
WindowEventMap-Interface 188
Without (Typ) 146

X

XML
  JSX (JavaScript-XML) 203
  Tags in Triple-Slash-Direktiven 287
  TSX (JSX und TypeScript) 204
  Verwendung von TSX mit React 205–208

Y

yield-Schlüsselwort 54

Z

Zahlen, numerische Trennzeichen in 23
Zusicherungen
  definitive Zuweisung 153

Nicht-null 151
Typ 149
Zuweisbarkeit 19, 123
abgeleitete generische Typen für explizit gebundenes Generic 74
für Enums 43
Zuweisung, this-Variable 52