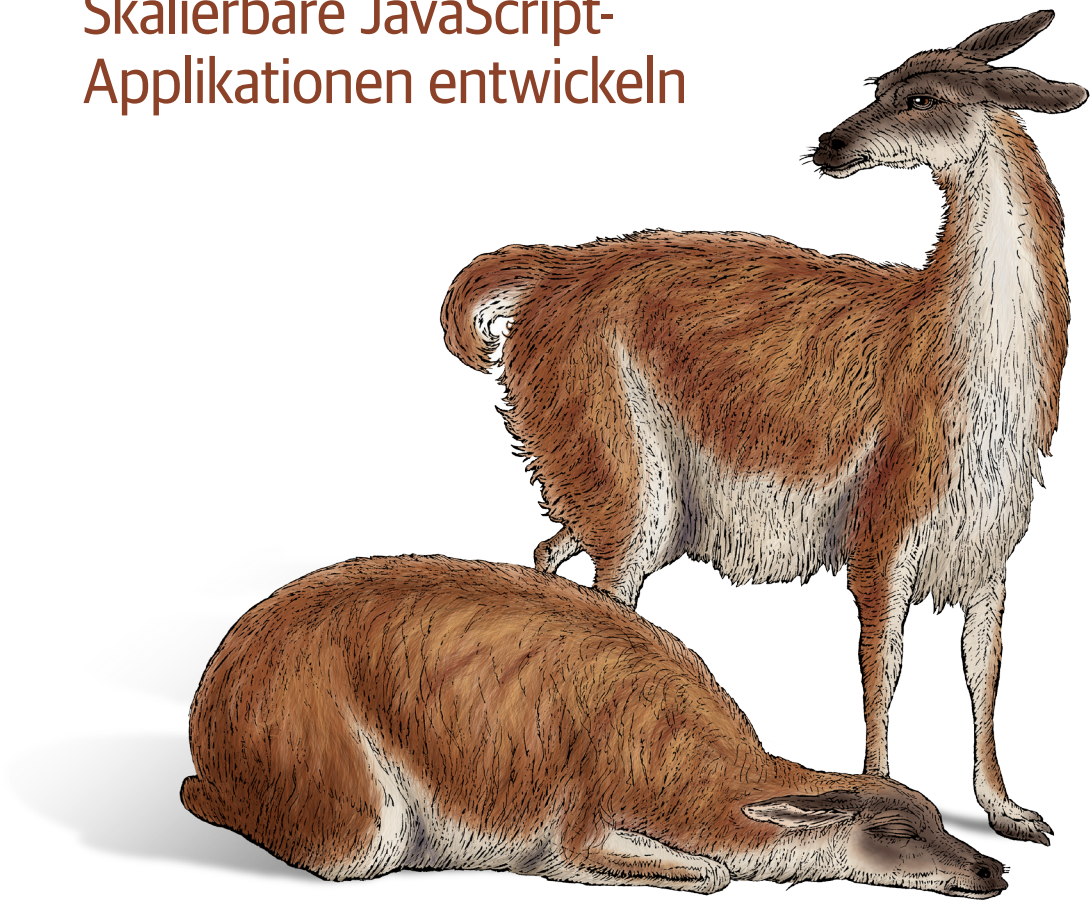
O'REILLY®

Programmieren in TypeScript

Skalierbare JavaScript-Applikationen entwickeln



Boris Cherny

Übersetzung von Jørgen W. Lang

Programmieren in TypeScript

Skalierbare JavaScript-Applikationen entwickeln

Boris Cherny

Deutsche Übersetzung von
Jørgen W. Lang

O'REILLY®

Boris Cherny

Lektorat: Ariane Hesse
Übersetzung: Jørgen W. Lang
Korrektorat: Claudia Lötschert, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-96009-122-6
PDF 978-3-96010-362-2
ePub 978-3-96010-360-8
mobi 978-3-96010-361-5

1. Auflage 2020

Wieblinger Weg 17
69123 Heidelberg

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.
O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Authorized German translation of the English edition of Programming TypeScript ISBN 9781492037651

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: kommentar@oreilly.de.

5 4 3 2 1 0

Für Sasha und Michael, die eines Tages vielleicht
auch ihre Begeisterung für Typen entdecken.

Inhalt

Vorwort

Dieses Buch ist für alle möglichen Programmierer gedacht: professionelle JavaScript-Entwickler, C#-Nutzer, Java-Sympathisanten, Python-Liebhaber, Ruby-Experten, Haskell-Nerds. Egal, welche Sprache Sie bevorzugen – solange Sie etwas Programmiererfahrung haben und sich mit den Grundlagen von Funktionen, Variablen, Klassen und Ausnahmen auskennen, ist dieses Buch für Sie geeignet. Erfahrung mit JavaScript und ein Grundwissen zu Document Objekt Model (DOM) und Netzwerk sind hilfreich. Obwohl wir diese Themen nicht bis ins letzte Detail behandeln, halten sie doch einige ausgezeichnete Beispiele bereit. Wenn Sie mit diesen Konzepten jedoch gar nicht vertraut sind, werden die Beispiele für Sie nicht viel Sinn ergeben.

Unabhängig davon, welche Programmiersprache Sie in der Vergangenheit benutzt haben, teilen wir doch die gleichen Erfahrungen. Das Aufspüren von Ausnahmen und das zeilenweise Durchgehen unseres Codes, um die Probleme zu finden und sie zu beheben, kommen in jeder Sprache vor. TypeScript versucht (recht erfolgreich), Ihnen diese unangenehmen Erfahrungen zu ersparen, indem es Ihren Code automatisch untersucht und Sie auf die übersehenen Fehler hinweist.

Es ist kein Problem, wenn Sie noch nie mit einer statisch typisierten Sprache gearbeitet haben. Ich werden Ihnen den Umgang mit Typen beibringen und zeige Ihnen, wie sie effektiv genutzt werden können, um Programmabstürze zu vermeiden, Ihren Code besser zu dokumentieren, und wie Sie Ihre Applikationen für mehr Benutzer, Entwickler und Server skalieren können. Ich versuche, ausschweifende Erklärungen zu vermeiden und Ideen intuitiv, leicht zu merken und praktisch zu vermitteln. Dabei verwende ich viele Beispiele, um die Dinge möglichst konkret zu halten.

Im Gegensatz zu anderen Sprachen ist TypeScript unglaublich praktisch. Es besitzt vollkommen neue Konzepte, mit denen Sie knapp und präzise formulieren können, was Sie wollen. Das sorgt nicht nur für moderne und sichere Applikationen, sondern auch dafür, dass Sie dabei Spaß haben.

Wie dieses Buch aufgebaut ist

Dieses Buch verfolgt zwei Ziele: Es soll Ihnen ein tiefgehendes Verständnis von der Funktionsweise von TypeScript vermitteln (Theorie) und Ihnen gleichzeitig möglichst viele praktische Ratschläge zum Schreiben von Code für Produktionsumgebungen (Praxis) geben.

Und weil TypeScript eine so praktische Sprache ist, liegen Theorie und Praxis meist sehr nah beieinander. Der Großteil dieses Buchs ist daher eine Mischung aus beidem. Dabei geht es in den ersten Kapiteln hauptsächlich um Theorie, während die andere Hälfte sich fast nur mit der Praxis beschäftigt.

Ich beginne mit den Grundlagen zu Compilern, Typecheckern und Typen. Danach gebe ich Ihnen einen groben Überblick über die verschiedenen Typen und Typoperatoren in TypeScript und ihre Verwendung. Das Gelernte verwenden wir dann, um uns mit fortgeschrittenen Themen zu beschäftigen. Dazu gehören beispielsweise die fortschrittlichsten Merkmale von TypeScripts Typsystem, die Fehlerbehandlung und asynchrone Programmierung. Zum Abschluss zeige ich Ihnen, wie Sie TypeScript mit Ihren Lieblings-Frameworks (Frontend und Backend) verwenden können, wie Sie bestehende JavaScript-Projekte zu TypeScript migrieren können und was Sie bei der Ausführung Ihrer TypeScript-Applikationen in einer Produktionsumgebung beachten müssen.

Die meisten Kapitel enthalten am Ende eine Reihe von Übungen. Versuchen Sie, diese Übungen selbst zu lösen. So bekommen Sie ein tieferes Verständnis der behandelten Inhalte, als es reines Lesen ermöglichen kann. Die Lösungen für die Übungen finden Sie online unter *https://github.com/bcherny/programming-typescript-answers*.

Coding-Stil

Ich habe versucht, über das gesamte Buch einen Coding-Stil beizubehalten. Einige Aspekte dieses Stils folgen eher meinen persönlichen Vorlieben. Zum Beispiel:

- Ich verwende Semikola (;) nur bei Bedarf.
- Für Einrückungen verwende ich grundsätzlich zwei Leerzeichen.
- Handelt es sich bei einem Codebeispiel nur um einen kurzen Abschnitt oder ist die Programmstruktur wichtiger als die Details, verwende ich kurze Variablennamen wie `a`, `f` oder `_`.

Dann gibt es aber noch Aspekte des Coding-Stils, die Sie meiner Meinung nach ebenfalls verwenden sollten. Dies sind unter anderem:

- Verwenden Sie die aktuellste JavaScript-Syntax und Sprachmerkmale (die neueste JavaScript-Version wird üblicherweise als *esnext* bezeichnet). Dadurch hält sich Ihr Code an die neuesten Standards, was die Interoperabilität und die Auffindbarkeit in Suchmaschinen erhöht und Ihnen möglicherweise sogar bei der Akquise neuer Aufträge hilft. Außerdem können Sie dadurch die Vorteile

mächtiger moderner JavaScript-Features wie Pfeil-Funktionen, Promises und Generatoren nutzen.

- Meistens sollten Sie Ihre Datenstrukturen mithilfe des Spread-Operators (`...`) als immutabel kennzeichnen.[1]
- Stellen Sie sicher, dass alles einen Typ besitzt, der nach Möglichkeit automatisch abgeleitet (inferred) werden kann. Verwenden Sie nicht unnötig explizite Typen. So bleibt Ihr Code klar und knapp. Gleichzeitig erhöht sich die Sicherheit, weil inkorrekte Typen klar erkennbar sind, anders als wenn Sie sie nur behelfsmäßig verarzten.
- Halten Sie Ihren Code wiederverwendbar und allgemein. Polymorphismus ist dabei Ihr bester Freund (siehe »Polymorphismus« auf Seite 66).

Diese Ideen sind natürlich nicht neu. Aber TypeScript funktioniert besonders gut, wenn Sie sich danach richten. Der TypeScript-eigene Downlevel-Compiler (TSC), die Unterstützung für schreibgeschützte Typen, eine mächtige Typableitung, Unterstützung für Polymorphismus und ein vollständig strukturelles Typsystem unterstützen einen guten Coding-Stil. Dabei bleibt die Sprache unglaublich ausdrucksstark und dem zugrunde liegenden JavaScript treu.

Noch ein paar weitere Hinweise, bevor wir anfangen:

JavaScript legt keine Pointer und Referenzen offen. Stattdessen verwendet es Werte- und Referenztypen. Werte sind immutabel. Hierzu gehören beispielsweise Strings, Zahlen und boolesche Werte, während Referenzen auf oftmals mutable Datenstrukturen verweisen, wie Arrays, Objekte und Funktionen. Wenn ich in diesem Buch das Wort »Wert« verwende, ist damit in etwa ein JavaScript-Wert oder eine -Referenz gemeint.

Ein paar Worte zum Abschluss: Manchmal muss Ihr Code mit JavaScript oder nicht korrekt typisierten Drittanbieter-Bibliotheken zusammenarbeiten, oder Sie haben es eilig. Das führt nicht unbedingt zu makellosem TypeScript-Code. Dieses Buch zeigt größtenteils, wie Sie TypeScript schreiben *sollten,* und versucht Ihnen klarzumachen, warum Kompromisse möglichst zu vermeiden sind. In der Praxis entscheiden Sie und Ihr Team jedoch selbst, wie Sie vorgehen.

In diesem Buch verwendete Konventionen

Folgende typografische Konventionen werden in diesem Buch verwendet:

Kursiv
: Bezeichnet neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateiendungen.

1 Ein JavaScript-Beispiel: Angenommen, Sie haben ein Objekt `o`, dem die Eigenschaft `k` mit dem Wert `3` hinzugefügt werden soll. Dann können Sie `o` entweder direkt verändern, indem Sie schreiben `o.k = 3`, oder Sie können die Änderungen durch die Erstellung eines *neuen* Objekts durchführen: `let p = {...o, k: 3}`.

`Nichtproportionalschrift`
: Wird verwendet für Codebeispiele sowie innerhalb von Absätzen für Programmelemente wie Variablen- oder Funktionsnamen, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter.

`Nichtproportionalschrift kursiv`
: Steht für Text, der durch Benutzereingaben ersetzt wird, oder Werte, die durch den Kontext bestimmt werden.

Dieses Element kennzeichnet einen Tipp oder Vorschlag.

Dieses Element bezeichnet einen allgemeinen Hinweis.

Dieses Element kennzeichnet eine Warnung oder einen Gefahrenhinweis.

Codebeispiele in diesem Buch

Zusätzliches Material (Codebeispiele, Übungen etc.) können Sie unter folgender Adresse herunterladen: *https://github.com/bcherny/programming-typescript-answers*.

Dieses Buch soll Ihnen bei Ihrer Arbeit helfen. Grundsätzlich dürfen Sie den Beispielcode aus diesem Buch in Ihren Programmen und Ihrer Dokumentation nutzen. Sie müssen hierfür keine ausdrückliche Genehmigung einholen, es sei denn, es handelt sich um eine größere Menge Code. Wenn Sie beispielsweise ein Programm schreiben, das mehrere Codeabschnitte aus diesem Buch verwendet, ist keine Erlaubnis nötig; für den Verkauf oder Vertrieb einer CD-ROM mit Beispielen aus O'Reilly-Büchern dagegen schon. Das Beantworten einer Frage durch Zitieren von Beispielcode erfordert keine Erlaubnis. Verwenden Sie einen erheblichen Teil der Beispielcodes aus diesem Buch in Ihrer Dokumentation, ist jedoch unsere Erlaubnis nötig.

Eine Quellenangabe ist zwar erwünscht, aber nicht unbedingt notwendig. Hierzu gehört in der Regel die Erwähnung von Titel, Autor, Verlag und ISBN, zum Beispiel: »*Programming TypeScript* von Boris Cherny (O'Reilly). Copyright 2019 Boris Cherny, 978-1-492-03765-1.«

Sollten Sie nicht sicher sein, ob die Nutzung der Codebeispiele außerhalb der hier erteilten Genehmigung liegt, nehmen Sie bitte unter der Adresse *permissions@oreilly.com* Kontakt mit uns auf.

Website zum Buch

Zu diesem Buch gibt es eine Website, auf der Sie Errata, Beispiele und weitere Informationen finden können. Sie finden die Website unter:
https://oreil.ly/programming-typescript.

Danksagungen

Dieses Buch ist das Ergebnis aus Jahren des Sammelns von Schnipseln und Skizzen, gefolgt von Jahren des Schreibens – frühmorgens, an Wochenenden und bei Nacht.

Vielen Dank an O'Reilly für die Möglichkeit, dieses Buch zu schreiben, und an meine Lektorin Angela Rufino für die Unterstützung während des gesamten Prozesses. Dankeschön auch an Nick Nance für seinen Beitrag über »Typsichere APIs« auf Seite 212 und an Shyam Seshadri für seinen Beitrag zu »Angular 6/7« auf Seite 208. Ein Dankeschön an meine technischen Sachverständigen. Das sind: Daniel Rosenwasser vom TypeScript-Team, der *sehr* viel Zeit damit verbracht hat, dieses Manuskript zu lesen und mich durch die Feinheiten von TypeScripts Typsystem zu leiten. Ein weiteres Dankeschön an Jonathan Creamer, Yakov Fain, Paul Buying und Rachel Head für technische Anpassungen und ihr Feedback. Mein herzlicher Dank gilt auch meiner Familie, Liza und Ilya, Vadim, Roza und Alik, Faina und Yosif, die mich ermutigt haben, dieses Projekt durchzuführen.

Meine größte Dankbarkeit gilt meiner Partnerin Sara Gilford, die mich während des Schreibens unermüdlich unterstützt hat, selbst wenn dadurch unsere Wochenendpläne durchkreuzt wurden oder es spätnächtliches Schreiben und Programmieren und viel zu viele unerwartete Gespräche über die Details von Typsystemen bedeutete. Ohne dich hätte ich das nicht geschafft. Ich werde dir immer dankbar für deine Unterstützung sein.

KAPITEL 1

Einführung

Sie haben sich also entschieden, ein Buch über TypeScript zu kaufen. Warum?

Vielleicht haben Sie einfach die Nase voll von seltsamen JavaScript-Fehlern wie `cannot read property blah of undefined`. Oder Sie haben gehört, dass Applikationen mit TypeScript besser skalierbar sind und wollten einfach mal wissen, ob das wirklich stimmt. Oder Sie programmieren funktional und wollten den nächsten logischen Schritt tun. Oder Ihrem Chef hat es so sehr gestunken, dass Ihr Code in der Produktion immer wieder für Fehler sorgte, dass er Ihnen dieses Buch zu Weihnachten geschenkt hat (sagen Sie Bescheid, wenn es zu sehr wehtut ...).

Egal, was Ihre Gründe sind: Was Sie gehört haben, ist wahr. TypeScript wird die nächste Generation von Web-Applikationen, mobiler Apps, NodeJS-Projekten und »Internet of Things (IoT)«-Geräten antreiben. TypeScript macht Ihre Programme sicherer, indem es auf häufige Fehler überprüft, als Dokumentation für Sie und Ihre Wartungsprogrammierer dient, Refaktorierungen schmerzlos macht und mindestens die Hälfte Ihrer Unit Tests (»Was für Unit Tests?«) unnötig macht. TypeScript wird Ihre Produktivität als Programmierer verdoppeln und garantiert Ihnen eine Verabredung mit dem (oder der) Barista von gegenüber.

Bevor Sie gleich blindlings über die Straße rennen, wollen wir die Dinge zunächst etwas genauer betrachten. Wir beginnen mit der Frage: »Was meine ich eigentlich mit ›sicherer‹?« Ich meine natürlich *Typsicherheit*.

> **Typsicherheit**
>
> Die Verwendung von Typen, um zu verhindern, dass Programme ungültige Aktionen ausführen.[1]

1 Je nachdem, welche statisch typisierte Sprache Sie benutzen, kann »ungültig« eine Reihe von Dingen bedeuten. Das können Programme sein, die bei der Ausführung abstürzen, bis hin zu Situationen, in denen nichts abstürzt, aber trotzdem unsinnige Dinge passieren.

Hier ein paar Beispiele für ungültige Operationen:

- Die Multiplikation einer Zahl mit einer Liste
- Der Aufruf einer Funktion mit einer Liste aus Strings, wenn eigentlich eine Liste mit Objekten gebraucht wird
- Der Aufruf einer nicht vorhandenen Methode an einem Objekt
- Der Import eines Moduls, das kürzlich an einem anderen Ort abgelegt wurde

Manche Sprachen versuchen, solche Fehler, so gut es geht, abzufangen und zu umgehen. Sie versuchen herauszufinden, was Sie beim Auftreten des Fehlers tatsächlich tun wollten. Schließlich tun auch Sie Ihr Bestes, oder? Nehmen Sie beispielsweise dieses JavaScript-Beispiel:

```
3 + []            // Ergibt den String "3"

let obj = {}
obj.foo           // Ergibt undefined

function a(b) {
  return b/2
}
a("z")            // Ergibt NaN
```

Wenn Sie etwas offensichtlich Ungültiges tun, löst JavaScript keine Ausnahme aus. Stattdessen versucht es, das Beste aus der Situation zu machen, und vermeidet Ausnahmen, soweit wie möglich. Ist JavaScript hilfreich? Sicherlich! Hilft es Ihnen, die Fehler schneller zu finden? Wahrscheinlich nicht.

Jetzt stellen Sie sich vor, JavaScript würde mehr Ausnahmen auslösen, anstatt stillschweigend zu versuchen, das Beste aus den Dingen zu machen, die es zur Verfügung hat. Dann würden wir möglicherweise Rückmeldungen wie diese bekommen:

```
3 + []            // Fehler: Wollten Sie wirklich eine Zahl und ein Array addieren?

let obj = {}
obj.foo           // Fehler: Sie haben vergessen, die Eigenschaft "foo" an obj zu
                  // definieren.

function a(b) {
  return b/2
}
a("z")            // Fehler: Die Funktion "a" erwartet eine Zahl,
                  // Sie haben aber einen String übergeben.
```

Verstehen Sie mich nicht falsch: Der Versuch, Ihre Fehler auszubügeln, ist für eine Programmiersprache eine tolle Eigenschaft (es wäre schön, wenn es das nicht nur für Programme gäbe). Für JavaScript trennt dieses Vorgehen jedoch die Verbindung zwischen dem Moment, wenn der Fehler auftritt, und dem, wenn Sie *herausfinden*, dass Ihr Code einen Fehler enthält. Das heißt, oftmals hören Sie erst von jemand anderem, dass Sie einen Fehler gemacht haben.

Die Frage ist also: *Wann genau* teilt Ihnen JavaScript mit, dass Sie einen Fehler gemacht haben?

Richtig. Erst wenn Sie das Programm tatsächlich *ausführen*. Das kann beispielsweise sein, wenn Sie das Programm in einem Browser testen, ein Benutzer Ihre Website besucht oder Sie einen Unit-Test durchführen. Wenn Sie diszipliniert sind und viele Unit- und End-to-end-Tests schreiben, Ihren Code vor der Veröffentlichung einer strengen Überprüfung und internen Tests unterziehen, finden Sie den Fehler hoffentlich, bevor es die Benutzer tun. Was aber, wenn nicht?

Da kommt TypeScript ins Spiel. Noch cooler als die Tatsache, dass TypeScript Ihnen hilfreiche Fehlermeldungen anzeigt, ist, *wann* es das tut: TypeScript zeigt Ihnen die Fehlermeldungen *in Ihrem Codeeditor. Während Sie schreiben*. Sie müssen sich also nicht mehr auf Unit Tests, Smoke Tests oder Ihre Mitarbeiter verlassen, um diese Fehler zu finden: TypeScript findet sie für Sie und warnt Sie bereits beim Schreiben des Programms. Sehen wir mal, was TypeScript zu unseren vorigen Beispielen zu sagen hat:

```
3 + []            // Error TS2365: Operator '+' cannot be applied to types '3'
                  // and 'never[]'.

let obj = {}
obj.foo           // Error TS2339: Property 'foo' does not exist on type '{}'.

function a(b: number) {
  return b / 2
}
a("z")            // Error TS2345: Argument of type '"z"' is not assignable to
                  // parameter of type 'number'.
```

TypeScript hilft aber nicht nur, ganze Klassen von Typ-bezogenen Fehlern zu vermeiden. Es verändert die Art, wie Sie Code schreiben. Sie werden Programme zunächst auf Typebene skizzieren, um sie dann auf Werteebene mit Inhalt zu füllen.[2] Sie werden bereits beim Schreiben des Programms an mögliche Sonderfälle denken, und nicht erst wenn das Programm schon fertig ist. Sie werden lernen, Programme zu entwickeln, die einfacher, schneller, verständlicher und wartbarer sind.

Sind Sie für Ihre Reise bereit? Dann los!

2 Keine Sorge, wenn der Begriff »Typebene« für Sie noch nicht klar ist. Wir werden ihn in späteren Kapiteln genauer erklären.

KAPITEL 2

TypeScript aus der Vogelperspektive

In den folgenden Kapiteln stelle ich Ihnen TypeScript vor, gebe Ihnen einen Überblick über die Funktionsweise des TypeScript-Compilers (TSC) und zeige Ihnen, welche Fähigkeiten TypeScript besitzt und welche Muster Sie damit entwickeln können. Wir beginnen mit dem Compiler.

Der Compiler

Je nachdem, welche Programmiersprache(n) Sie bereits benutzt haben (d. h., bevor Sie dieses Buch gekauft und sich für ein Leben in Sicherheit entschieden haben), haben Sie vermutlich ein anderes Verständnis davon, wie Programme funktionieren. Im Vergleich zu anderen beliebten Sprachen wie JavaScript oder Java funktioniert TypeScript eher ungewöhnlich. Daher ist es sinnvoll, erst einmal auf dem gleichen Stand zu sein, bevor wir weitermachen.

Beginnen wir ganz allgemein: Programme sind Dateien, die von Ihnen – den Programmierern – geschriebenen Text enthalten. Der Text wird von einem speziellen Programm namens *Compiler* untersucht, interpretiert (»geparst«) und in einen *abstrakten Syntaxbaum* (»abstract syntax tree«, AST) umgewandelt. Das ist eine Datenstruktur, die z. B. Leerzeichen, Kommentare und Ihre Meinung zur »Leerzeichen oder Tabs«-Debatte ignoriert. Danach konvertiert der Compiler den AST in eine niedriger angesiedelte (lower level) Form namens *Bytecode*. Diesen Bytecode können Sie dann einem Programm namens *Runtime* (oder Laufzeitumgebung) übergeben, das den Bytecode auswertet und das Ergebnis zurückgibt. Wenn Sie ein Programm ausführen, weisen Sie also tatsächlich die Laufzeitumgebung an, den Bytecode auszuführen, den der Compiler aus dem AST erzeugt hat, nachdem er diesen aus Ihrem Quellcode geparst hat. Die Details können sich unterscheiden, sind aber für die meisten Hochsprachen gleich oder zumindest ähnlich.

Noch einmal, die Schritte sind:

1. Programm (Quellcode) wird in einen AST geparst.
2. AST wird in Bytecode kompiliert.
3. Bytecode wird von der Laufzeitumgebung ausgeführt.

Eine Besonderheit von TypeScript ist, dass es nicht direkt in Bytecode kompiliert wird, sondern nach ... JavaScript-Code! Diesen können Sie dann wie üblich in Ihrem Browser, mit NodeJS oder manuell auf dem Papier ausführen (Letzteres nur für den Fall, dass Sie dies erst nach dem Aufstand der Maschinen lesen).

Sehr wahrscheinlich fragen Sie sich jetzt: »Moment mal! Im vorigen Kapitel haben Sie gesagt, dass TypeScript meinen Code sicherer macht. An welcher Stelle passiert das denn jetzt?«

Gute Frage: Den wichtigen Schritt habe ich übersprungen: Nachdem der TypeScript-Compiler den AST für Ihr Programm erzeugt, aber bevor es den Code ausgibt, führt er einen *Typecheck* (bitte merken Sie sich dieses Wort!) für Ihren Code aus.

> **Typechecker**
>
> Ein spezielles Programm, das sicherstellt, dass Ihr Code typsicher ist.

Das Typechecking ist die wahre Magie hinter TypeScript. So stellt TypeScript sicher, dass Ihr Programm wie erwartet funktioniert und es keine offensichtlichen Fehler gibt und dass der/die hübsche Barista von gegenüber Sie auch wirklich zurückruft. (Haben Sie etwas Geduld. Er/sie ist vermutlich bloß gerade sehr beschäftigt.)

Wenn wir das Typechecking und die Ausgabe von JavaScript mit einbeziehen, sieht die Kompilierung von TypeScript ungefähr so aus wie in Abbildung 2-1:

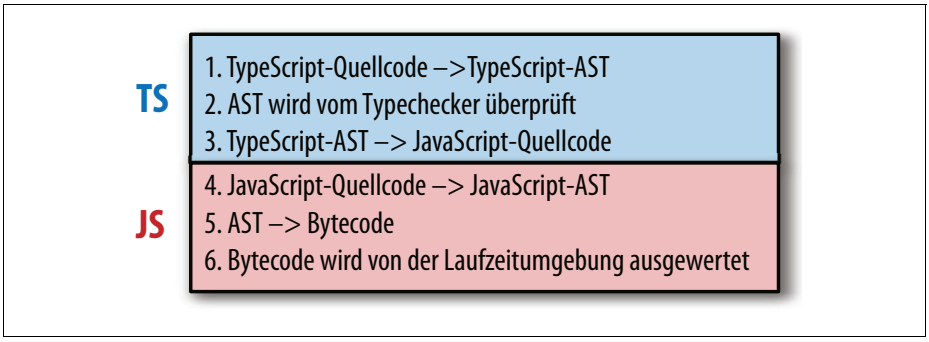


Abbildung 2-1: TypeScript kompilieren und ausführen

Die Schritte 1–3 werden von TSC übernommen, die Schritte 4–6 werden von der JavaScript-Runtime ausgeführt, die in Ihrem Browser, NodeJS oder einer anderen von Ihnen verwendeten JavaScript-Engine lebt.

JavaScript-Compiler und -Runtimes werden oft zu einem gemeinsamen Programm namens *Engine* kombiniert. Das ist das Ding, mit dem Sie als Programmierer normalerweise interagieren. So funktionieren beispielsweise V8 (die Engine hinter NodeJS, Chrome und Opera), SpiderMonkey (Firefox), JSCore (Safari) und Chakra (Edge). Sie geben JavaScript das Aussehen einer *interpretierten* Sprache.

In den Schritten 1 und 2 werden dabei die Typen Ihres Programms benutzt, in Schritt 3 jedoch nicht. Das darf man ruhig noch mal wiederholen: *Wenn TSC Ihren Code von TypeScript nach JavaScript kompiliert, erfolgt dies ohne Prüfung der Typen.*

Die Typen in Ihrem Programm haben also keinen Einfluss auf die von Ihrem Programm erzeugten Ausgaben und werden nur für das Typechecking verwendet. Dadurch ist es quasi narrensicher, mit den Typen Ihres Programms herumzuspielen, sie zu aktualisieren und zu verbessern, ohne dass Ihre Applikation dabei versehentlich Schaden nehmen könnte.

Das Typsystem

Alle modernen Sprachen besitzen das eine oder andere *Typsystem*.

> **Typsystem**
>
> Ein Satz von Regeln, die der Typechecker verwendet, um Typen in Ihrem Programm zuzuweisen.

Allgemein gibt es zwei Arten von Typsystemen: solche, in denen Sie dem Compiler in expliziter Syntax mitteilen müssen, welche Typen die Dinge haben, und Typsysteme, die Typen automatisch ableiten. Beide Ansätze haben ihre Vor- und Nachteile.[1]

TypeScript ist von beiden Arten der Typsysteme inspiriert: Sie können Ihre Typen explizit annotieren oder Sie können die meisten Typen automatisch ableiten lassen.

Um TypeScript Ihre Typen explizit mitzuteilen, verwenden Sie *Annotationen*. Diese haben die Form *Wert: Typ* und sagen dem Typechecker: »Der Typ dieses *Werts* lautet *Typ*. Am besten sehen wir uns hierzu ein paar Beispiele an (in den Kommentaren zu jeder Zeile sehen Sie die tatsächlich von TypeScript abgeleiteten Typen):

1 In diesem Bereich gibt es viele verschiedene Sprachen: JavaScript, Python und Ruby leiten die Typen zur Laufzeit ab; Haskell und OCaml überprüfen auf fehlende Typen bzw. leiten die Typen während der Kompilierung ab; Scala und TypeScript benötigen einige explizite Typen. Der Rest wird während der Kompilierung abgeleitet und überprüft. Java und C erfordern explizite Annotationen für fast alles, wobei die Überprüfung während der Kompilierung stattfindet.

```
let a: number = 1                // a ist eine Zahl
let b: string = 'hello'          // b ist ein String
let c: boolean[] = [true, false] // c ist ein Array mit booleschen Werten
```

Wollen Sie, dass TypeScript die Typen für Sie ableitet, können Sie die Annotationen weglassen und TypeScript die Arbeit erledigen lassen:

```
let a = 1                        // a ist eine Zahl
let b = 'hello'                  // b ist ein String
let c = [true, false]            // c ist ein  Array mit booleschen Werten
```

Man erkennt schnell, wie gut TypeScript Typen für Sie ableiten kann. Auch wenn Sie die Annotationen weglassen, bleiben die Typen gleich! In diesem Buch werden wir Annotationen nur bei Bedarf verwenden. Ansonsten überlassen wir es nach Möglichkeit TypeScript, die Ableitungen für uns vorzunehmen.

Allgemein gilt es als guter Programmierstil, TypeScript so viele Typen wie möglich automatisch ableiten zu lassen und so wenig explizit typisierten Code wie möglich zu verwenden.

TypeScript im Vergleich mit JavaScript

Als Nächstes werfen wir einen genaueren Blick auf das Typsystem von TypeScript und sehen, welche Unterschiede und Gemeinsamkeiten im Vergleich zum Typsystem von JavaScript bestehen. Einen Überblick sehen Sie in Tabelle 2-1. Ein gutes Verständnis dieser Unterschiede ist der Schlüssel zum Verständnis der Funktionsweise von TypeScript.

Tabelle 2-1: Ein Vergleich der Typsysteme von JavaScript und TypeScript

Typsystem-Merkmal	JavaScript	TypeScript
Wie werden Typen begrenzt (bounding)?	Dynamisch	Statisch
Werden Typen automatisch konvertiert?	Ja	Nein (meistens)
Wann werden Typen überprüft?	Zur Laufzeit	Bei der Kompilierung
Zu welchem Zeitpunkt werden Fehler ausgelöst?	Zur Laufzeit (meistens)	Bei der Kompilierung (meistens)

Wie werden Typen begrenzt (bounding)

JavaScripts dynamische Typbindung bedeutet, dass es Ihr Programm ausführen muss, um die darin verwendeten Typen zu ermitteln. JavaScript kennt Ihre Typen vor der Ausführung des Programms nicht.

TypeScript ist eine *graduell typisierte* Sprache. Dadurch funktioniert TypeScript am besten, wenn es die Typen aller Dinge in Ihrem Programm bereits bei der Kompilierung kennt. Damit das Programm kompiliert werden kann, muss aber nicht zwingend alles bekannt sein. Selbst in einem nicht typisierten Programm kann TypeScript einige Typen für Sie ableiten und Fehler abfangen. Ohne dass

alle Typen bekannt sind, ist es aber fast unvermeidlich, dass einige Fehler es bis zu Ihren Benutzern schaffen.

Diese graduelle Typisierung ist besonders nützlich, wenn Legacy-Code von untypisiertem JavaScript zu typisiertem TypeScript migriert werden muss (mehr hierzu in »Schrittweise Migration von JavaScript zu TypeScript« auf Seite 240). Grundsätzlich sollten Sie versuchen, eine hundertprozentige Typabdeckung zu erzielen – es sei denn, Sie befinden sich gerade mitten in der Migration einer Codebasis. Dieser Ansatz wird auch in diesem Buch umgesetzt. In Ausnahmefällen weise ich ausdrücklich darauf hin.

Werden Typen automatisch konvertiert?

JavaScript ist schwach typisiert. Das heißt, wenn Sie etwas Ungültiges tun wie etwa die Addition einer Zahl mit einem Array (siehe Kapitel 1), wird eine Reihe von Regeln angewandt, um herauszufinden, was Sie tatsächlich gemeint haben. So wird versucht, das Beste aus dem übergebenen Code zu machen. Sehen wir uns am Beispiel von `3 + [1]` einmal genau an, wie JavaScript vorgeht:

1. JavaScript bemerkt, dass `3` eine Zahl ist und `[1]` ein Array.
2. Da wir `+` benutzen, geht JavaScript davon aus, dass wir beides miteinander verketten wollen.
3. `3` wird implizit in einen String umgewandelt und ergibt `"3"`.
4. `[1]` wird ebenfalls implizit in einen String umgewandelt und ergibt `"1"`.
5. Beide Strings werden verkettet, das Ergebnis lautet `"31"`.

Das könnten wir auch expliziter schreiben (sodass JavaScript die Schritt 1, 3 und 4 vermeidet):

```
3 + [1];                        // ergibt "31"

(3).toString() + [1].toString()  // ergibt ebenfalls "31"
```

JavaScript versucht, Ihnen durch seine intelligente Typumwandlungen zu helfen; TypeScript beschwert sich dagegen, sobald Sie etwas Ungültiges tun. Wenn Sie den gleichen JavaScript-Code an TSC übergeben, erhalten Sie eine Fehlermeldung:

```
3 + [1];                        // Error TS2365: Operator '+' cannot be applied
                                // to types '3' and 'number[]'.

(3).toString() + [1].toString()  // ergibt "31"
```

Sobald Sie etwas tun, das nicht korrekt erscheint, beschwert sich TypeScript. Wenn Sie Ihre Absichten dagegen explizit erklären, steht es Ihnen aber auch nicht im Weg. Das erscheint sinnvoll: Niemand im Vollbesitz seiner geistigen Kräfte würde versuchen, eine Zahl und ein Array miteinander zu addieren, und als Ergebnis einen String erwarten (abgesehen vielleicht von der JavaScript-Hexe Bavmorda,

die ihre Zeit damit verbringt, im Licht schwarzer Kerzen im Keller ihres Start-ups Dämonenbeschwörungen zu programmieren).

Fehler, die durch diese Art impliziter Typumwandlungen in JavaScript verursacht werden, lassen sich oft nur schwer finden und sind für viele JavaScript-Programmierer kein Spaß. Sie erschweren einzelnen Entwicklern die Arbeit und machen es noch schwerer, den Code für ein größeres Team zu skalieren. Schließlich muss jeder Programmierer verstehen, von welchen impliziten Annahmen Ihr Code ausgeht.

Kurz gesagt: Wenn Sie eine Typumwandlung vornehmen müssen, tun Sie das explizit.

Wann werden die Typen überprüft?

Meistens ist es JavaScript egal, welche Typen Sie ihm übergeben. Stattdessen versucht es, das Übergebene so gut wie möglich in das umzuwandeln, was Sie eigentlich erwarten.

Im Gegensatz dazu überprüft TypeScript Ihre Typen während der Kompilierungsphase (erinnern Sie sich an Schritt 2 am Anfang dieses Kapitels?). Sie müssen den Code also nicht erst laufen lassen, um den `Error` aus dem vorigen Beispiel zu sehen. TypeScript führt eine *statische Analyse* Ihres Codes durch, um Fehler wie diese zu finden, und sagt Bescheid, bevor der Code ausgeführt wird. Wird der Code nicht kompiliert, heißt das ziemlich sicher, dass Sie einen Fehler gemacht haben, den Sie vor der Ausführung beheben sollten.

Abbildung 2-2 zeigt, was passiert, wenn ich das letzte Codebeispiel in VSCode (den Codeeditor meiner Wahl) eingebe.

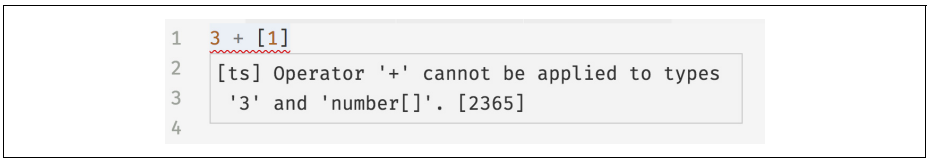


Abbildung 2-2: Von VSCode ausgegebener TypeError

Wenn Ihr bevorzugter Codeeditor eine gute TypeScript-Erweiterung besitzt, wird der Fehler bereits *bei der Eingabe* (mit einer roten Unterschlängelung) deutlich hervorgehoben. Die Rückmeldung kommt bereits beim Schreiben des Codes, und Sie können den Fehler sofort beheben.

Wann werden die Fehler ausgelöst?

In JavaScript werden Ausnahmen und implizite Typumwandlungen zur Laufzeit ausgelöst bzw. vorgenommen.[2] Das heißt, Sie müssen Ihr Programm erst ausführen,

2 Nur um das klarzustellen: Einige Syntaxfehler werden von JavaScript bereits nach dem Parsen, aber vor der Ausführung des Programms ausgelöst. Wenn Sie JavaScript als Teil eines Build-Prozesses (zum Beispiel mit Babel) parsen, können Sie diese Fehler bereits während der Build-Phase zum Vorschein bringen.

damit Sie eine nützliche Rückmeldung darüber erhalten, ob Sie etwas Ungültiges getan haben. Im besten Fall passiert das als Teil eines Unit Tests, schlimmstenfalls bekommen Sie eine unfreundliche E-Mail von einem Benutzer.

TypeScript löst sowohl Syntax- als auch Typ-bezogene Fehler während der Kompilierungsphase aus. Dadurch werden Fehler wie diese schon bei der Eingabe in Ihrem Codeeditor angezeigt. Wenn Sie noch nie mit einer inkrementell kompilierten, statisch typisierten Sprache gearbeitet haben, kann das eine großartige Erfahrung sein.[3]

Trotzdem gibt es eine Reihe von Fehlern, die auch TypeScript nicht während der Kompilierung abfangen kann. Hierzu gehören Stack-Überläufe, unterbrochene Netzwerkverbindungen und falsch formatierte Benutzereingaben. Ausnahmen wie diese werden zur Laufzeit ausgelöst. Allerdings kann TypeScript die meisten Fehler, die in einer reinen JavaScript-Welt erst zur Laufzeit auftreten, so umwandeln, dass sie schon bei der Kompilierung erkannt werden.

Einrichtung des Codeeditors

Nachdem Sie einen ersten Eindruck des TypeScript-Compilers und des Typsystems bekommen haben, wollen wir Ihren Codeeditor einrichten, damit wir möglichst schnell mit echtem Code arbeiten können.

Beginnen Sie mit dem Download eines Codeeditors, in dem Sie Ihren Code schreiben. Ich persönlich mag VSCode, weil dieser Editor besonders gut für die Arbeit mit TypeScript geeignet ist. Andere Editoren wie Sublime Text, Atom, Vim, WebStorm oder ein anderer Editor Ihrer Wahl funktionieren aber auch. Programmierer sind bei der Wahl ihrer IDE sehr eigen, daher überlasse ich Ihnen diese Entscheidung. Wenn Sie VSCode verwenden wollen, folgen Sie den Installationsanweisungen auf der Website (*https://code.visualstudio.com/*).

TSC selbst ist ein Kommandozeilenprogramm, das in TypeScript geschrieben ist. Das heißt, Sie benötigen NodeJS, um den Compiler auszuführen. Folgen Sie den Anweisungen auf der offiziellen NodeJS-Website (*https://nodejs.org*), um es auf Ihrem Rechner zum Laufen zu bringen.[4]

Zu NodeJS gehört NPM, ein Paketmanager, mit dem Sie die Abhängigkeiten Ihrer Projekte verwalten und den Build-Prozess steuern können. Wir beginnen mit der Installation von TSC und TSLint (einem Linter für TypeScript). Öffnen Sie hierfür Ihr Terminal, erstellen Sie einen neuen Ordner und initialisieren Sie darin NPM:

```
# Einen neuen Ordner anlegen
mkdir chapter-2
```

3 Inkrementell kompilierte Sprachen können bei kleinen Änderungen schnell rekompiliert werden, ohne dass hierfür das gesamte Programm (inklusive der Teile, die Sie nicht verändert haben) neu kompiliert werden muss.

4 Damit gehört TSC zur mystischen Klasse der sogenannten *Self-Hosting Compiler*, d.h. Compiler, die sich selbst kompilieren.

```
cd chapter-2

# Ein neues NPM-Projekt initialisieren (folgen Sie den Anweisungen)
npm init

# TSC, TSLint und die Typendeklarationen für NodeJS TSC installieren
npm install --save-dev typescript tslint @types/node
```

tsconfig.json

Für jedes TypeScript-Projekt gibt es in dessen Wurzelverzeichnis eine Datei namens *tsconfig.json*. In *tsconfig.json* wird beispielsweise definiert, welche Dateien kompiliert werden sollen, in welches Verzeichnis sie kompiliert werden sollen und welche JavaScript-Version ausgegeben werden soll.

Erstellen Sie im Wurzelverzeichnis Ihres Projekts eine neue Datei namens *tsconfig.json* (`touch tsconfig.json`). Danach können Sie die Datei in Ihrem Codeeditor öffnen und Folgendes eingeben:[5]

```
{
  "compilerOptions": {
    "lib": ["es2015"],
    "module": "commonjs",
    "outDir": "dist",
    "sourceMap": true,
    "strict": true,
    "target": "es2015"
  },
  "include": [
    "src"
  ]
}
```

Werfen wir zunächst einen Blick auf die verschiedenen Optionen und ihre Bedeutung (Tabelle 2-2):

Tabelle 2-2: tsconfig.json

Option	Beschreibung
`include`	In welchen Ordnern soll TSC nach TypeScript-Dateien suchen?
`lib`	Welche APIs soll TSC in der Umgebung erwarten, in der Sie Ihren Code ausführen? Hierzu gehören Dinge wie ES5s `Function.prototype.bind`, ES2015s `Object.assign` und der DOM-eigene `document.querySelector`.
`module`	Für welches Modulsystem soll TSC Ihren Code kompilieren (CommonJS, SystemJS, ES2015 etc.)?
`outDir`	In welchem Ordner soll TSC den erzeugten JavaScript-Code speichern?

5 Für diese Übungen erzeugen wir *tsconfig.json* manuell. Wenn Sie in Zukunft ein TypeScript-Projekt erstellen, können Sie den TSC-eigenen Initialisierungsbefehl `./node_modules/.bin/tsc --init` verwenden, um die Datei automatisch erstellen zu lassen.