

O'REILLY®

Deutsche
Ausgabe

TypeScript

Ein praktischer Einstieg

Typsicheres JavaScript für skalierbare
Webanwendungen



Josh Goldberg
Übersetzung von
Jens Olaf Koch

Inhalt

Cover

Leserstimmen zu TypeScript – Ein praktischer Einstieg

Titel

Impressum

Widmung

Inhalt

Vorwort Josh Goldberg

Vorwort zur deutschen Ausgabe

Teil I Konzepte

1 Von JavaScript zu TypeScript

Die Geschichte von JavaScript

Die Tücken von purem JavaScript

Kostspielige Freiheit

Schwache Dokumentation

Schwächere Entwicklertools

TypeScript!

Erste Schritte auf dem TypeScript-Spielplatz

TypeScript in Aktion

Freiheit durch Restriktion

Präzise Dokumentation

Bessere Entwicklertools

Kompilieren

Erste lokale Schritte

TypeScript lokal ausführen

Editor-Funktionen

Was TypeScript nicht ist

Ein Heilmittel für schlechten Code

Eine Erweiterung für JavaScript (größtenteils)

Langsamer als JavaScript

Ergebnis einer abgeschlossenen Entwicklung

Zusammenfassung

2 Das Typsystem

Was ist ein Typ?

Typsysteme

Fehlerarten

Zuweisbarkeit

Zuweisbarkeitsfehler verstehen

Typanmerkungen

Unnötige Typanmerkungen

Typformen

Module

Zusammenfassung

3 Vereinigungs- und Literaltypen

Vereinigungstypen

Deklaration von Vereinigungstypen

Eigenschaften von Vereinigungstypen

Type Narrowing

Type Narrowing durch Zuweisung

Bedingungsabhängige Prüfungen

typeof-Prüfungen

Literaltypen

Zuweisbarkeit von Literaltypen

Strikte Nullprüfung

Der Milliarden-Dollar-Fehler

Type Narrowing durch Wahrheitsprüfung

Variablen ohne Anfangswerte

Typalias

Typalias sind kein JavaScript

Typalias kombinieren

Zusammenfassung

4 Objekte

Objekttypen

Deklaration von Objekttypen

Aliasing von Objekttypen

Strukturelle Typisierung

Verwendungsprüfung

Prüfung auf überzählige Eigenschaften

Verschachtelte Objekttypen

Optionale Eigenschaften

Vereinigungen von Objekttypen

Abgeleitete Vereinigungen von Objekttypen

Explizite Vereinigungen von Objekttypen

Type Narrowing von Objekttypen

Diskriminierte Vereinigungstypen

Schnittmengentypen

Gefahren von Schnittmengentypen

Zusammenfassung

Teil II Features

5 Funktionen

Funktionsparameter

Erforderliche Parameter

Optionale Parameter
Standard-Parameter
Restparameter
 Rückgabetypen
Explizite Rückgabetypen
 Funktionstypen
Klammersetzung bei Funktionstypen
Ableitung von Parametertypen
Typaliase für Funktionen
 Weitere Rückgabetypen
void zurückgeben
never zurückgeben
 Überladung von Funktionen
Kompatibilität der Aufrufsignaturen
 Zusammenfassung
6 Arrays
 Array-Typen
Array- und Funktionstypen
Arrays mit Vereinigungstyp
Evolving-any-Arrays
Mehrdimensionale Arrays
 Array-Elemente
Fallstrick: Unzuverlässige Ableitungen
 Spreads und Restparameter
Spreads
Spreading von Restparametern
 Tupel
Zuweisbarkeit von Tupeln

Typinferenz bei Tupeln

 Zusammenfassung

7 Interfaces

 Typaliase im Vergleich zu Interfaces

 Eigenschaftstypen

Optionale Eigenschaften

Schreibgeschützte Eigenschaften

Funktionen und Methoden

Aufrufsignaturen

Indexsignaturen

Verschachtelte Interfaces

 Erweiterung von Interfaces

Überschreiben von Eigenschaften

Gleichzeitige Erweiterung mehrerer Interfaces

 Verschmelzung von Interfaces

Namenskonflikte bei Mitgliedern

 Zusammenfassung

8 Klassen

 Klassenmethoden

 Klasseneigenschaften

Funktionseigenschaften

Initialisierungsprüfung

Optionale Eigenschaften

Schreibgeschützte Eigenschaften

 Klassen als Typen

 Klassen und Interfaces

Gleichzeitige Implementierung mehrerer Interfaces

 Eine Klasse erweitern

Zuweisbarkeit erweiterter Klassen
Überschreiben von Konstruktoren
Überschreiben von Methoden
Überschreiben von Eigenschaften
 Abstrakte Klassen
 Sichtbarkeit von Mitgliedern
Statische Feld-Modifizierer
 Zusammenfassung
 9 Typ-Modifizierer
 Top Types
any, schon wieder
unknown
 Typprädikate
 Typoperatoren
keyof
typeof
 Typzusicherungen
Zusicherung des Fehlertyps
Nicht-null-Zusicherung
Fallstricke bei Typzusicherungen
 Const-Zusicherungen
Literale statt primitiver Datentypen
Schreibgeschützte Objekte
 Zusammenfassung
 10 Generics
 Generische Funktionen
Explizite generische Aufruftypen
Multiple Typparameter für Funktionen

Generische Interfaces

Abgeleitete generische Interface-Typen

Generische Klassen

Explizite generische Klassentypen

Erweiterung generischer Klassen

Implementierung von generischen Interfaces

Methoden-Generics

Generics von statischen Klassenmitgliedern

Generische Typalias

Generische diskriminierte Vereinigungstypen

Generische Modifier

Generische Standardwerte

Eingeschränkte generische Typen

keyof und eingeschränkte Typparameter

Promises

Promises erstellen

Asynchrone Funktionen

Generics richtig verwenden

Die goldene Regel der Generics

Namenskonventionen für Generics

Zusammenfassung

Teil III Verwendung

11 Deklarationsdateien

Deklarationsdateien

Laufzeitwerte deklarieren

Globale Werte

Verschmelzung globaler Interfaces

Globale Augmentationen

- Integrierte Deklarationen
- Bibliotheks-Deklarationen
- DOM-Deklarationen
 - Moduldeklarationen
- Moduldeklarationen mit Platzhaltern
 - Typen von Paketen
- declaration
- Typen von Dependency-Paketen
- Paket-Typen bereitstellen
 - DefinitelyTyped
- Verfügbarkeit von Typinformationen
 - Zusammenfassung
- 12 IDE-Funktionen verwenden
 - Im Code navigieren
- Definitionen suchen
- Verwendungen suchen
- Implementierungen suchen
 - Code schreiben
- Namen vervollständigen
- Automatische Import-Updates
- Code-Aktionen
 - Effektiv mit Fehlern umgehen
- Fehler des Sprachdiensts
 - Zusammenfassung
- 13 Konfigurationsoptionen
 - tsc-Optionen
- Pretty-Mode
- Watch-Mode

TSConfig-Dateien

tsc —init

Kommandozeile oder Konfigurationsdatei?

Dateien einschließen

include

exclude

Alternative Dateiendungen

JSX-Syntax

resolveJsonModule

Ausgabe von JavaScript

outDir

target

Ausgabe von Deklarationsdateien

Sourcemaps

noEmit

Typechecking

lib

skipLibCheck

strict-Mode

Module

module

moduleResolution

Interoperabilität mit CommonJS

isolatedModules

JavaScript

allowJs

checkJs

JSDoc-Unterstützung

Konfigurationserweiterungen

extends

Basiskonfigurationsdateien

Projektreferenzen

composite

references

Build-Mode

Zusammenfassung

Teil IV Zugaben

14 Syntaxerweiterungen

Parametereigenschaften in Klassen

Experimentelle Dekoratoren

Enums

Automatische Zuordnung numerischer Werte

Enums mit String-Werten

Const-Enums

Namensräume

Namespace-Exporte

Verschachtelte Namensräume

Namensräume in Typdefinitionen

Geben Sie Modulen den Vorzug vor Namensräumen

Type-Only-Importe und -Exporte

Zusammenfassung

15 Typoperationen

Abgebildete Typen

Abgebildete Typen auf Basis von Objekttypen

Änderung von Modifiern

Generische abgebildete Typen

Bedingte Typen

Generische bedingte Typen

Distributive Typen

Abgeleitete Typen

Abgebildete bedingte Typen

never

never und Schnittmengen- und Vereinigungstypen

never und bedingte Typen

never und abgebildete Typen

Template-Literaltypen

Intrinsische String-Manipulationstypen

Template-Literal Schlüssel

Remapping von Schlüssel

Typoperationen und Komplexität

Zusammenfassung

Glossar

Index

Über den Autor

Kolophon

Objektliterale

Ein Satz von Schlüsseln und Werten

Jeder mit seinem eigenen Typ

In Kapitel 3, »Vereinigungs- und Literaltypen«, ging es um Vereinigungs- und Literaltypen: das Arbeiten mit primitiven Datentypen wie `boolean` und deren literalen Werten wie `true`. Diese primitiven Datentypen kratzen aber nur an der Oberfläche der komplexen Objektformen, die in JavaScript-Code üblicherweise verwendet werden. TypeScript wäre ziemlich unbrauchbar, wäre es nicht in der Lage, diese Objekte darzustellen. In diesem Kapitel geht es darum, wie man komplexe Objektformen beschreibt und wie TypeScript ihre Zuweisbarkeit überprüft.

Objekttypen

Wenn Sie ein Objektliteral mit der Syntax `{...}` erstellen, betrachtet TypeScript es als einen neuen Objekttyp – oder Typform – basierend auf seinen Eigenschaften. Dieser Objekttyp hat die gleichen Eigenschaftsnamen und primitiven Datentypen wie die Werte des Objekts. Der Zugriff auf Eigenschaften des Werts erfolgt entweder mit einem Ausdruck der Form `value.member` oder `value['member']`.

TypeScript erkennt, dass der Typ der folgenden Variable `poet` der eines Objekts mit zwei Eigenschaften ist: `born`, vom Typ `number`, und `name`, vom Typ `string`. Der Zugriff auf diese Member wäre erlaubt, aber der Versuch, auf einen anderen Membernamen zuzugreifen, würde zu einem Typfehler führen, da dieser Name nicht existiert:

```
const poet = {  
  
    born: 1935,  
  
    name: "Mary Oliver",  
  
};  
  
poet['born']; // Typ: number  
  
poet.name; // Typ: string  
  
poet.end;  
  
//   ~~~  
  
// Error: Property 'end' does not exist on  
  
// type '{ born: number; name: string; }'.
```

Objekttypen sind ein Kernkonzept dafür, wie TypeScript JavaScript-Code versteht. Die zugrunde liegende Typform aller Werte außer `null` und `undefined` weist einen Satz von Mitgliedern auf. Für das Typechecking muss TypeScript deshalb den Objekttyp aller Werte verstehen.

Deklaration von Objekttypen

Typen direkt von vorhandenen Objekten abzuleiten, ist schön und gut, aber manchmal möchte man den Typ eines Objekts explizit deklarieren. Man benötigt eine Möglichkeit, eine Objektform getrennt von den Objekten zu beschreiben, die ihr entsprechen.

Objekttypen können mit einer Syntax beschrieben werden, die ähnlich aussieht wie bei Objektliteralen, sich aber auf Typen anstelle von Werten für Felder bezieht. Es ist die gleiche Syntax, die TypeScript in Fehlermeldungen zur Zuweisbarkeit von Typen benutzt.

Die folgende Variable `poetLater` hat den gleichen Typ wie zuvor und besitzt die Eigenschaften `name: string` und `born: number`:

```
let poetLater: {  
  
    born: number;  
  
    name: string;  
  
};  
  
// Ok  
  
poetLater = {  
  
    born: 1935,  
  
    name: "Mary Oliver",  
  
};  
  
poetLater = "Sappho";  
  
// Error: Type 'string' is not assignable to  
  
// type '{ born: number; name: string; }'
```

Aliasing von Objekttypen

Es wäre ziemlich lästig, müsste man Objekttypen wie `{ born: number; name: string; }` ständig ausschreiben. Deshalb wird Typformen meistens mithilfe von Typaliasen ein Name zugewiesen.

Der vorherige Codeausschnitt könnte mithilfe eines `type Poet` umgeschrieben werden, was den zusätzlichen Vorteil hätte, dass die TypeScript-

Fehlermeldung zur Zuweisbarkeit etwas klarer ausfiele:

```
type Poet = {  
  
    born: number;  
  
    name: string;  
  
};  
  
let poetLater: Poet;  
  
// Ok  
  
poetLater = {  
  
    born: 1935,  
  
    name: "Sara Teasdale",  
  
};  
  
poetLater = "Emily Dickinson";  
  
// Error: Type 'string' is not assignable to type 'Poet'.
```



In den meisten TypeScript-Projekten wird das Schlüsselwort `interface` zur Beschreibung von Objekttypen verwendet – ein Feature, das ich erst in Kapitel 7, »Interfaces«, behandeln werde. Objekttyp-Aliase und Interfaces sind fast identisch: Alle Aussagen in diesem Kapitel gelten gleichermaßen für Interfaces.

Ich bespreche Objekttypen an dieser Stelle des Buchs, weil man für das Verständnis des Typsystems von TypeScript wissen muss, wie TypeScript Objektliterale interpretiert. Diese Konzepte werden auch im nächsten Abschnitt

dieses Buchs eine wichtige Rolle spielen, wenn wir uns den Features von TypeScript widmen.

Strukturelle Typisierung

TypeScripts Typsystem ist *strukturell typisiert*: Das bedeutet, dass jeder Wert, der einem Typ entspricht, als Wert dieses Typs verwendet werden darf. Mit anderen Worten: Wenn Sie deklarieren, dass ein Parameter oder eine Variable von einem bestimmten Objekttyp ist, teilen Sie TypeScript dadurch mit, dass alle verwendeten Objekte dieses Typs auch all dessen Eigenschaften aufweisen müssen.

Die folgenden Objekttyp-Aliase `WithFirstName` und `WithLastName` deklarieren beide jeweils nur ein einziges Member vom Typ `string`. Die Variable `hasBoth` hat zufällig beide Typen – auch wenn sie nicht explizit unter Nennung der beiden Objekttypen so deklariert wurde –, sodass sie anderen Variablen zugewiesen werden kann, die als einer der beiden Objekttyp-Aliase deklariert sind:

```
type WithFirstName = {  
  
    firstName: string;  
  
};
```

```
type WithLastName = {  
  
    lastName: string;  
  
};
```

```
const hasBoth = {  
  
    firstName: "Lucille",  
  
    lastName: "Clifton",
```

```

};

// Ok: `hasBoth` hat eine Eigenschaft `firstName` des Typs
`string`

let withFirstName: WithFirstName = hasBoth;

// Ok: `hasBoth` hat eine Eigenschaft `lastName` des Typs
`string`

let withLastName: WithLastName = hasBoth;

```

Strukturelle Typisierung ist nicht dasselbe wie das sogenannte *Duck Typing*, ein Begriff, der auf die Redewendung »Wenn es wie eine Ente aussieht und wie eine Ente quakt, ist es wahrscheinlich eine Ente« zurückgeht.

- Strukturelle Typisierung bedeutet, dass es ein statisches System gibt, das den Typ prüft – im Fall von TypeScript ist das der Typechecker.
- Beim Duck Typing werden Objekttypen erst bei ihrer Verwendung zur Laufzeit überprüft.

Zusammengefasst: *JavaScript* ist *duck-typed*, während *TypeScript* *strukturell typisiert* ist.

Verwendungsprüfung

Wenn ein Wert an einer Stelle bereitgestellt wird, die mit einem Objekttyp annotiert ist, prüft TypeScript, ob der Wert diesem Objekttyp zuweisbar ist. Zunächst muss der Wert die erforderlichen Eigenschaften des Objekttyps aufweisen. Wenn im Objekt ein für den Objekttyp erforderliches Member fehlt, gibt TypeScript einen Typfehler aus.

Der folgende Objekttyp-Alias `FirstAndLastNames` setzt voraus, dass sowohl die Eigenschaften `first` als auch `last` vorhanden sind. Ein Objekt, das beides enthält, darf einer Variablen vom Typ `FirstAndLastNames` zugewiesen werden, ein Objekt ohne diese beiden Elemente jedoch nicht:

```
type FirstAndLastNames = {
```

```

    first: string;

    last: string;

};

// Ok

const hasBoth: FirstAndLastNames = {

    first: "Sarojini",

    last: "Naidu",

};

const hasOnlyOne: FirstAndLastNames = {

    first: "Sappho"

};

// Property 'last' is missing in type '{ first: string; }'

// but required in type 'FirstAndLastNames'.

```

Die Typen der Eigenschaften dürfen ebenfalls nicht voneinander abweichen. Objekttypen spezifizieren sowohl die Namen der erforderlichen Eigenschaften als auch deren Typen. Weicht eine Eigenschaft eines Objekts ab, meldet TypeScript einen Typfehler.

Der folgende `TimeRange`-Typ erwartet, dass das Member `start` vom Typ `Date` ist. Das Objekt `hasStartString` verursacht einen Typfehler, weil sein `start` stattdessen den Typ `string` hat:

```

type TimeRange = {

```

```

    start: Date;

};

const hasStartString: TimeRange = {

    start: "1879-02-13",

    // Error: Type 'string' is not assignable to type 'Date'.

};

```

Prüfung auf überzählige Eigenschaften

TypeScript meldet einen Typfehler, wenn eine Variable mit einem Objekttyp deklariert ist, ihr Anfangswert aber mehr Felder enthält, als dieser Typ beschreibt. Indem man eine Variable als von einem bestimmten Objekttyp deklariert, kann man mithilfe des Typecheckers sicherstellen, dass diese Variable nur die bei diesem Typ zu erwartenden Felder enthält.

Die folgende Variable `poetMatch` hat genau die Felder, die im Objekttyp mit dem Alias `Poet` beschrieben sind, während `extraProperty` einen Typfehler verursacht, weil darin eine zusätzliche Eigenschaft `activity` aufgeführt wird:

```

type Poet = {

    born: number;

    name: string;

}

// Ok: Die Felder entsprechen den für den Typ Poet erwarteten

```

```

const poetMatch: Poet = {

    born: 1928,

    name: "Maya Angelou"

};

const extraProperty: Poet = {

    activity: "walking",

    born: 1935,

    name: "Mary Oliver",

};

// Error: Type '{ activity: string; born: number; name:
string; }'

// is not assignable to type 'Poet'.

//   Object literal may only specify known properties,

//   and 'activity' does not exist in type 'Poet'.

```

Beachten Sie, dass die Prüfung auf überzählige Eigenschaften nur bei Objektliteralen stattfindet, die an Stellen neu erzeugt werden, die als Objekttyp deklariert sind. Wird ein bereits vorhandenes Objektliteral angegeben, findet keine Prüfung auf überzählige Eigenschaften statt.

Die folgende Variable `extraPropertyButOk` löst keinen Typfehler bezüglich des Typs `Poet` aus dem vorherigen Beispiel aus, weil ihr Anfangswert strukturell mit allen für `Poet` erforderlichen Eigenschaften übereinstimmt:

```
const existingObject = {  
  
    activity: "walking",  
  
    born: 1935,  
  
    name: "Mary Oliver",  
  
};  
  
const extraPropertyButOk: Poet = existingObject; // Ok
```

Prüfungen auf überzählige Eigenschaft werden überall dort ausgelöst, wo ein neues Objekt an einer Stelle erzeugt wird, an der erwartet wird, dass das Objekt mit einem Objekttyp übereinstimmt – was, wie Sie in späteren Kapiteln sehen werden, Array-Member, Klassenfelder und Funktionsparameter einschließt. Das Verbot überzähliger Eigenschaften ist eine weitere Unterstützung seitens TypeScript, um sicherzustellen, dass Ihr Code korrekt ist und sich wie erwartet verhält. Überzählige Eigenschaften, die nicht in ihren Objekttypen deklariert sind, gehen meist auf falsch geschriebene Eigenschaftsnamen oder unbenutzten Code zurück.

Verschachtelte Objekttypen

Da JavaScript-Objekte Member anderer Objekte sein können, müssen TypeScript's Objekttypen in der Lage sein, verschachtelte Objekttypen im Typsystem darzustellen. Die Syntax dafür ist die gleiche wie zuvor, aber mit einem Objekttyp { ... } anstelle des Namens eines primitiven Datentyps.

Der folgende Typ Poem wird als Objekt deklariert, dessen Eigenschaft author selbst ein Objekt mit den Eigenschaften firstName: string und lastName: string ist. Die Variable poemMatch kann Poem zugewiesen werden, weil sie mit dessen Struktur übereinstimmt, während poemMismatch nicht zugewiesen werden kann, weil ihre author-Eigenschaft anstelle von firstName und lastName die Eigenschaft name enthält:

```
type Poem = {
```

```
    author: {  
        firstName: string;  
        lastName: string;  
    };  
    name: string;  
};  
  
// Ok  
  
const poemMatch: Poem = {  
    author: {  
        firstName: "Sylvia",  
        lastName: "Plath",  
    },  
    name: "Lady Lazarus",  
};  
  
const poemMismatch: Poem = {  
    author: {  
        name: "Sylvia Plath",
```

```

    },

    // Error: Type '{ name: string; }' is not assignable

    // to type '{ firstName: string; lastName: string; }'.

    // Object literal may only specify known properties, and
    'name'

    // does not exist in type '{ firstName: string;
    lastName: string; }'.

    name: "Tulips",

};

```

Man könnte den Typ Poem auch deklarieren, indem man die Form der Eigenschaft author in einem eigenen Objekttyp-Alias Author definiert. Extrahiert man verschachtelte Typen in ihre eigenen Typaliase, ermöglicht das TypeScript zudem, informativere Typ-Fehlermeldungen auszugeben. Im folgenden Fall kann sich die Fehlermeldung auf Author anstelle von '{ firstName: string; lastName: string; }' beziehen:

```

type Author = {

    firstName: string;

    lastName: string;

};

type Poem = {

    author: Author;

```

```

    name: string;

};

const poemMismatch: Poem = {

    author: {

        name: "Sylvia Plath",

    },

    // Error: Type '{ name: string; }' is not assignable to
    // type 'Author'.

    // Object literal may only specify known properties,

    // and 'name' does not exist in type 'Author'.

    name: "Tulips",

};

```



Es ist normalerweise eine gute Idee, verschachtelten Objekttypen eigene Typnamen zu geben. Das sorgt zugleich für besser lesbaren Code und verständlichere TypeScript-Fehlermeldungen.

In späteren Kapiteln werden Sie sehen, dass Objekttyp-Member auch andere Typen wie Arrays und Funktionen haben können.

Optionale Eigenschaften

Eigenschaften von Objekttypen sind im Objekt normalerweise erforderlich, aber es gibt einen Ausweg: Mit einem ? vor dem : in der Typanmerkung einer Eigenschaft können Sie festlegen, dass es sich um eine optionale Eigenschaft handelt.

Der folgende Typ Book erfordert nur eine Eigenschaft pages und erlaubt optional einen author. Objekte, die sich an diese Definition halten, können, sofern pages existiert, author nach Belieben liefern oder weglassen:

```
type Book = {  
  
    author?: string;  
  
    pages: number;  
  
};  
  
// Ok  
  
const ok: Book = {  
  
    author: "Rita Dove",  
  
    pages: 80,  
  
};  
  
const missing: Book = {  
  
    author: "Rita Dove",  
  
};  
  
// Error: Property 'pages' is missing in type
```

```
// '{ author: string; }' but required in type 'Book'.
```

Denken Sie daran, dass es einen Unterschied gibt zwischen optionalen Eigenschaften und solchen, deren Typ ein Vereinigungstyp ist, der als eine seiner Konstituenten `undefined` enthält. Eine Eigenschaft, die mit `?` als optional deklariert ist, darf fehlen. Eine Eigenschaft, die als erforderlich, aber mit dem Zusatz `| undefined` deklariert wurde, darf dagegen nicht fehlen – allerdings darf ihr Wert `undefined` lauten.

Die Eigenschaft `editor` des folgenden Typs `Writers` kann bei der Deklaration von Variablen ausgelassen werden, da ihre Deklaration ein `?` enthält. Die Deklaration der Eigenschaft `author` enthält kein `?` – sie muss also vorhanden sein, darf jedoch den Wert `undefined` annehmen:

```
type Writers = {  
  
    author: string | undefined;  
  
    editor?: string;  
  
};  
  
// Ok: author wird der Wert undefined zugewiesen  
  
const hasRequired: Writers = {  
  
    author: undefined,  
  
};  
  
const missingRequired: Writers = {};  
  
//      ~~~~~  
  
// Error: Property 'author' is missing in type
```

```
// '{}' but required in type 'Writers'.
```

In Kapitel 7, »Interfaces«, werden weitere Arten von Eigenschaften behandelt, während es in Kapitel 13, »Konfigurationsoptionen«, um TypeScript's striktheitsbezogene Einstellungen hinsichtlich optionaler Eigenschaften geht.

Vereinigungen von Objekttypen

Es kann sinnvoll sein, in TypeScript einen Typ beschreiben zu wollen, bei dem es sich um einen oder mehrere Objekttypen mit leicht unterschiedlichen Eigenschaften handeln kann. Darüber hinaus möchten Sie diese Objekttypen vielleicht auf der Grundlage des Werts einer bestimmten Eigenschaft weiter eingrenzen.

Abgeleitete Vereinigungen von Objekttypen

Wenn eine Variable einen Anfangswert erhält, der von einem von mehreren Objekttypen sein kann, leitet TypeScript ihren Typ als eine Vereinigung von Objekttypen ab. Dieser Vereinigungstyp enthält für jede der möglichen Objektformen eine eigene Konstituente. Jede dieser Konstituenten wird alle möglichen Eigenschaften des Typs aufweisen, allerdings sind die Eigenschaften, denen kein Anfangswert zugewiesen wurde, in den entsprechenden Konstituenten mit ? als optional gekennzeichnet.

Der folgende Wert von poem besitzt in allen Fällen eine Eigenschaft name des Typs string, während die Eigenschaften pages und rhymes optionaler Natur sind:

```
const poem = Math.random() > 0.5

  ? { name: "The Double Image", pages: 7 }

  : { name: "Her Kind", rhymes: true };

// Typ:

// {
```

```
// name: string;

// pages: number;

// rhymes?: undefined;

// }

// |

// {

// name: string;

// pages?: undefined;

// rhymes: boolean;

// }

poem.name; // string

poem.pages; // number | undefined

poem.rhymes; // boolean | undefined
```

Explizite Vereinigungen von Objekttypen

Alternativ können Sie Ihre Objekttypen explizit als Vereinigung von Objekttypen definieren. Das erfordert ein wenig mehr Code, hat aber den Vorteil, dass Sie mehr Kontrolle über Ihre Objekttypen erhalten. Ist der Typ eines Werts eine Vereinigung von Objekttypen, erlaubt TypeScript's Typsystem nur den Zugriff auf

Eigenschaften, die auf ausnahmslos allen Objekttypen des Vereinigungstyps existieren.

Die folgende Version der zuvor gezeigten Variablen `Poem` ist explizit als Vereinigungstyp angelegt, der immer eine Eigenschaft `name` und eine der beiden Eigenschaften `pages` und `rhymes` besitzt. Der Zugriff auf `name` ist im folgenden Beispiel erlaubt, weil diese Eigenschaft immer existiert, während dies bei `pages` und `rhymes` nicht garantiert ist:

```
type PoemWithPages = {  
  
    name: string;  
  
    pages: number;  
  
};  
  
type PoemWithRhymes = {  
  
    name: string;  
  
    rhymes: boolean;  
  
};  
  
type Poem = PoemWithPages | PoemWithRhymes;  
  
const poem: Poem = Math.random() > 0.5  
  
    ? { name: "The Double Image", pages: 7 }  
  
    : { name: "Her Kind", rhymes: true };  
  
poem.name; // Ok  
  
poem.pages;
```

```

// ~~~~~

// Property 'pages' does not exist on type 'Poem'.

// Property 'pages' does not exist on type 'PoemWithRhymes'.

poem.rhymes;

// ~~~~~

// Property 'rhymes' does not exist on type 'Poem'.

// Property 'rhymes' does not exist on type 'PoemWithPages'.

```

Auf diese Weise der Zugriff auf potenziell nicht existierende Member von Objekten zu verhindern, kann die Codesicherheit erhöhen. Kann ein Wert einen von mehreren Typen haben, ist für Eigenschaften, die nicht auf all diesen Typen existieren, ansonsten nicht sichergestellt, dass sie auf dem Objekt existieren.

Genauso wie Vereinigungstypen aus Literalstypen und/oder primitiven Datentypen durch Type Narrowing eingeengt werden müssen, um auf Eigenschaften zuzugreifen, die nicht auf allen Konstituenten vorhanden sind, müssen auch Vereinigungen von Objekttypen eingeengt werden.

Type Narrowing von Objekttypen

Wenn der Typechecker feststellt, dass ein Codebereich nur ausgeführt wird, wenn ein Wert mit einem Vereinigungstyp eine bestimmte Eigenschaft besitzt, schränkt sie den Typ des Werts auf diejenigen Konstituenten ein, die diese Eigenschaft aufweisen. Mit anderen Worten: Type Narrowing findet auch bei Objekten statt, sofern Sie im Code deren Form prüfen.

In Fortsetzung des explizit typisierten poem-Beispiels prüfen wir jetzt, ob "pages" in poem als Type Guard funktioniert und sicherstellen kann, dass es sich um ein PoemWithPages oder – im else-Fall – um ein PoemWithRhymes handelt:

```

if ("pages" in poem) {

    poem.pages; // Ok: poem wurde eingeengt auf PoemWithPages

} else {

    poem.rhymes; // Ok: poem wurde eingeengt auf
    PoemWithRhymes

}

```

Beachten Sie, dass TypeScript keine Wahrheitsprüfungen wie `if (poem.pages)` zulässt. Der Versuch, auf eine Eigenschaft eines Objekts zuzugreifen, die möglicherweise nicht existiert, wird als Typfehler betrachtet, selbst wenn er auf eine Weise erfolgt, die sich wie ein Type Guard zu verhalten scheint:

```

if (poem.pages) { /* ... */ }

//      ~~~~~

// Property 'pages' does not exist on type 'Poem'.

//   Property 'pages' does not exist on type 'PoemWithRhymes'.

```

Diskriminierte Vereinigungstypen

Bei einer weiteren in JavaScript und TypeScript beliebten Art von Objekten mit Vereinigungstyp wird durch eine dafür speziell vorgesehene Eigenschaft des Objekts angegeben, welche Form das Objekt aufweist. Diese Art von Typform wird als *diskriminierter Vereinigungstyp* (»diskriminiert« im Sinne von »unterschieden«) bezeichnet, und die Eigenschaft, deren Wert den Typ des Objekts angibt, als *Diskriminante*. Wenn Type Guards zur Prüfung eine Diskriminante nutzen, kann TypeScript Type Narrowing anwenden.

Zum Beispiel beschreibt der folgende Typ `Poem` ein Objekt, das einen der folgenden geänderten Typen `PoemWithPages` oder `PoemWithRhymes` haben kann. Der Typ wird durch die als Diskriminante dienende Eigenschaft `type` angegeben. Hat `poem.type` den Wert `"pages"`, kann TypeScript daraus ableiten, dass `poem` vom Typ `PoemWithPages` sein muss. Ohne diese Typeinengung könnte für keine der beiden Eigenschaften garantiert werden, dass sie auf dem Wert existiert:

```
type PoemWithPages = {  
  
    name: string;  
  
    pages: number;  
  
    type: 'pages';  
  
};  
  
type PoemWithRhymes = {  
  
    name: string;  
  
    rhymes: boolean;  
  
    type: 'rhymes';  
  
};  
  
type Poem = PoemWithPages | PoemWithRhymes;  
  
const poem: Poem = Math.random() > 0.5  
  
    ? { name: "The Double Image", pages: 7, type: "pages" }  
  
    : { name: "Her Kind", rhymes: true, type: "rhymes" };
```

```

if (poem.type === "pages") {

    console.log(`It's got pages: ${poem.pages}`); // Ok

} else {

    console.log(`It rhymes: ${poem.rhymes}`);

}

poem.type; // Type: 'pages' | 'rhymes'

poem.pages;

// ~~~~~

// Error: Property 'pages' does not exist on type 'Poem'.

// Property 'pages' does not exist on type 'PoemWithRhymes'.

```

Diskriminierte Vereinigungstypen sind mein Lieblingsfeature in TypeScript, weil sie ein gängiges, elegantes JavaScript-Muster hervorragend mit dem Type Narrowing von TypeScript kombinieren. In Kapitel 10, »Generics«, und den dort behandelten Projekten erfahren Sie mehr über die Verwendung von diskriminierten Vereinigungstypen für generische Datenoperationen.

Schnittmengentypen

Mit dem Operator `|` konstruierte Vereinigungstypen repräsentieren den Typ eines Werts, der einen von mehreren Typen haben kann. In JavaScript dient der Laufzeitoperator `|` als Gegenstück zum `&`-Operator. In vergleichbarer Weise erlaubt TypeScript die Darstellung eines Typs, der gleichzeitig mehrere Typen darstellt, und zwar einen mit `&` gebildeten sogenannten *Schnittmengentyp*. Schnittmengentypen (intersection types) werden in der Regel mithilfe von Objekttyp-Aliasen als Kombination bestehender Objekttypen deklariert.

Im folgenden Beispiel wird aus den Typaliasen `Artwork` und `Writing` ein kombinierter Typ `WrittenArt` gebildet, der die Eigenschaften `genre`, `name` und `pages` besitzt:

```
type Artwork = {  
  
    genre: string;  
  
    name: string;  
  
};  
  
type Writing = {  
  
    pages: number;  
  
    name: string;  
  
};  
  
type WrittenArt = Artwork & Writing;  
  
// Äquivalent zu:  
  
// {  
  
//     genre: string;  
  
//     name: string;  
  
//     pages: number;  
  
// }
```

Schnittmengentypen und Vereinigungstypen¹ können auch kombiniert werden, was manchmal nützlich sein kann, um diskriminierte Vereinigungen in einem einzigen Typ zu beschreiben.

Der folgende Typ `ShortPoem` besitzt immer eine Eigenschaft `author` und ist zugleich ein diskriminierter Vereinigungstyp mit einer Eigenschaft `type` als Diskriminante:

```
type ShortPoem = { author: string } & (  
  
    | { kigo: string; type: "haiku"; }  
  
    | { meter: number; type: "villanelle"; }  
  
);  
  
// Ok  
  
const morningGlory: ShortPoem = {  
  
    author: "Fukuda Chiyo-ni",  
  
    kigo: "Morning Glory",  
  
    type: "haiku",  
  
};  
  
const oneArt: ShortPoem = {  
  
    author: "Elizabeth Bishop",  
  
    type: "villanelle",  
  
};
```

```

};

// Error: Type '{ author: string; type: "villanelle"; }'

// is not assignable to type 'ShortPoem'.

// Type '{ author: string; type: "villanelle"; }' is not
assignable to

// type '{ author: string; } & { meter: number; type:
"villanelle"; }'.

// Property 'meter' is missing in type '{ author: string;
type: "villanelle"; }'

// but required in type '{ meter: number; type:
"villanelle"; }'.

```

Gefahren von Schnittmengentypen

Schnittmengentypen sind ein nützliches Konstrukt, aber bei ihrer Verwendung kann man leicht sich selbst oder den TypeScript-Compiler verwirren. Falls Sie sie einsetzen, empfehle ich Ihnen, den Code so einfach wie möglich zu halten.

Überlange Fehlermeldungen zur Zuweisbarkeit

Wenn Sie einen komplexen Schnittmengentyp verwenden, z.B. in einer Kombination mit einem Vereinigungstyp, werden Fehlermeldungen, die die Zuweisbarkeit betreffen, zunehmend unverständlich. Das ist ein grundsätzliches Problem des Typsystems von TypeScript (und typisierter Programmiersprachen im Allgemeinen): Mit zunehmender Komplexität wird es um so schwieriger, die Meldungen des Typecheckers zu verstehen.

Im Fall des Typs `ShortPoem` des vorherigen Codebeispiels wäre der Lesbarkeit gedient, würde man den Typ in eine Reihe von Objekttyp-Aliase aufteilen, damit TypeScript bei Fehlermeldungen darauf zurückgreifen kann:

```
type ShortPoemBase = { author: string };
```

```
type Haiku = ShortPoemBase & { kigo: string; type: "haiku" };
```

```
type Villanelle = ShortPoemBase & { meter: number; type: "villanelle" };
```

```
type ShortPoem = Haiku | Villanelle;
```

```
const oneArt: ShortPoem = {
```

```
    author: "Elizabeth Bishop",
```

```
    type: "villanelle",
```

```
};
```

```
// Type '{ author: string; type: "villanelle"; }'
```

```
// is not assignable to type 'ShortPoem'.
```

```
//   Type '{ author: string; type: "villanelle"; }'
```

```
//   is not assignable to type 'Villanelle'.
```

```
//     Property 'meter' is missing in type
```

```
//     '{ author: string; type: "villanelle"; }'
```

```
//     but required in type '{ meter: number; type: "villanelle"; }'.
```

never

Schnittmengentypen sind außerdem leicht zu missbrauchen, um unmögliche Typen zu erzeugen. Primitive Datentypen können nicht als Konstituenten in

einem Schnittmengentyp zusammengeführt werden, da es logisch unmöglich ist, dass ein Wert gleichzeitig von mehreren unterschiedlichen primitiven Datentypen sein kann. Der Versuch, zwei primitive Datentypen durch & miteinander zu verbinden, erzeugt einen Typ `never`, dargestellt durch das gleichlautende Schlüsselwort `never`:

```
type NotPossible = number & string;
```

```
// Type: never
```

Das Schlüsselwort und der Typ `never` werden in Programmiersprachen als *Bottom Type* oder leerer Typ bezeichnet. Ein Bottom Type ist ein Typ, für den es keine möglichen Werte gibt und auf den nicht zugegriffen werden kann. Es gibt keine Typen, die Elementen mit Bottom Type zugewiesen werden können:

```
let notNumber: NotPossible = 0;
```

```
// ~~~~~
```

```
// Error: Type 'number' is not assignable to type 'never'.
```

```
let notString: never = "";
```

```
// ~~~~~
```

```
// Error: Type 'string' is not assignable to type 'never'.
```

In TypeScript-Projekten wird normalerweise nur sehr selten – wenn überhaupt je – der Typ `never` verwendet. Er wird hin und wieder eingesetzt, um unmögliche Zustände darzustellen. Taucht er ansonsten auf, handelt es sich in den meisten Fällen jedoch um einen Fehler, der durch die falsche Verwendung von Schnittmengentypen entsteht. Mehr dazu erfahren Sie in Kapitel 15, »Typoperationen«.

Zusammenfassung

In diesem Kapitel konnten Sie Ihr Verständnis des TypeScript-Typsystems vertiefen und haben gelernt, mit Objekten zu arbeiten. Wir haben uns mit folgenden Themen beschäftigt:

- Ableitung von Typen aus Objekttyp-Literalen
- Beschreibung von Objektliteral-Typen, einschließlich verschachtelter und optionaler Eigenschaften
- Deklaration, Ableitung und Type Narrowing von Vereinigungen von Objektliteralen
- Diskriminierte Vereinigungstypen und Diskriminanten
- Kombination von Objekttypen in Schnittmententypen



Nachdem Sie dieses Kapitel gelesen haben, sollten Sie das Gelernte auf <https://learningtypescript.com/objects> anwenden und einüben.

Wie deklariert ein Anwalt seinen TypeScript-Typ?

»Einspruch!« (»I object!«)

Symbole

! (Ausrufezeichen)

Initialisierungsprüfung von Klasseneigenschaften deaktivieren 129

Nicht-null-Zusicherung 156

? (Fragezeichen)

optionale Eigenschaften 73–74

optionale Parameter 85

... (Ellipse), Spread-Operator 86

für Arrays 101

Tupel als Restparameter 104

() (runde Klammern)

Arrays und 98

in Funktionstypen 90

@ts-check (Kommentar) 243

& (Ampersand), Schnittmengentypen 78

(Raute), private Klassenmember 141–143

| (Pipe)-Operator 56

A

abgebildete Typen

auf Basis vorhandener Typen 272–273

bedingte Typen und 280

generische 275

- never (Typ) und 282
- Schlüssel remappen 286–287
- Signaturen und 273–274
- Zugriffs-Modifizier ändern 274–275
- Zweck 271–272
- abgeleitete Interfaces, überschriebene Eigenschaften 120
- abgeleitete Klassen *siehe* Subklassen
- abgeleitete Typen 279
- abgeleitete Vereinigungstypen von Objekten 74
- ableiten, Variablentyp 43
- abstrakte Klassen
 - Beschreibung 140–141
 - Implementierungen suchen von 207
- allowJs (Compileroption) 242
- allowSyntheticDefaultImports (Compileroption) 241
- ambienter Kontext 188, 189
- Ampersand (&), Schnittmengentypen 78
- Anfangswert, fehlend 64
- any (Typ)
 - ambienter Kontext 189
 - Beschreibung 147–148
 - evolving any 48, 99
 - noImplicitAny (Compileroption) 234
 - useUnknownInCatchVariables (Compileroption) 237–238
- Array-Elemente, abfragen 100–101
- Arrays
 - Beschreibung 97
 - Elemente, abfragen 100–101
 - Restparameter 86–87
 - Tupel
 - ableiten durch Typinferenz 105
 - Beschreibung 102
 - const-Zusicherungen 106
 - explizite Typen 105

- als Restparameter 104
- Zuweisbarkeit 103
- als Typen
 - Evolving-any-Arrays 99
 - Funktionsstypen und 98
 - multidimensionale Arrays 99
 - Typanmerkungen für 98
 - Vereinigungstypen und 98–99
- Zusammenführung mit Spread-Operator 101
- Arrays mit fester Größe *siehe* Tupel
- as (Schlüsselwort) 154
- as const-Operator 106–107
- async-Funktionen
 - Promises und 180
- Aufrufsignaturen in Interfaces 114
- ausführen
 - Terminal-Compiler 215
 - TypeScript lokal 37–38
- Ausgabe von JavaScript 227
 - declaration (Compileroption) 229
 - emitDeclarationOnly (Compileroption) 230
 - outDir (Compileroption) 227–228
 - Sourcemaps 231
 - target (Compileroption) 228–233
- Ausrufezeichen (!)
 - Initialisierungsprüfung von Klasseneigenschaften deaktivieren 129
 - Nicht-null-Zusicherung 156
- automatische Zuordnung numerischer Werte
 - in Enums 261–262
- Autovervollständigung beim Schreiben von Code 35–36, 208–209

B

- Babel 36, 227, 269
- Basiskonfigurationsdateien (TSConfig) 246

bedingte Typen

abgebildete Typen und 280

abgeleitete 279

Distributivität 279

generische 277–278

never (Typ) und 281

Zweck 277

bedingungsabhängige Prüfungen, Type Narrowing durch 58

Bibliotheks-Deklarationsdateien 192–194

bigint (primitiver Datentyp) 43

bivariante Funktionsparameter 237

boolean (primitiver Datentyp) 43, 60

Bottom Type 80

Build-Mode (tsc-Befehl) 249–250

C

checkJs (Compileroption) 243

Code-Aktionen beim Schreiben von Code

Quick-Fixes 212

Refaktorisieren mit 212

Umbenennen mit 211

Zweck 210

Code-Navigation

Implementierungen, suchen 207

Referenzen, suchen 206

Typdefinitionen, suchen 205

Code schreiben

Autovervollständigung 208–209

Code-Aktionen

Quick-Fixes 212

Refaktorisieren mit 212

Umbenennen mit 211

Zweck 210

Import-Updates 209–210

- Code-Sharing mit Modulen 51–52
- Code-Stil in TypeScript 39
- CommonJS-Interoperabilität 240–241
- Compiler
 - Definition 32
 - Fehlerbehandlung 37, 213–217
 - Kompilieren von TypeScript 36
- composite (Compileroption) 248
- const-Enums 263–264
- const-Variable 60
- const-Zusicherungen 106–107
 - Literale als Literaltyp 160
 - schreibgeschützte Objektliterale 161
 - Zweck 159

D

- Dateien einschließen 223–224
- Dateierweiterungen
 - JSON-Dateien 226
 - JSX-Syntax 224–226
- declaration (Compileroption) 196, 229
- declarationMap (Compileroption) 231
- declare (Schlüsselwort) 188–189
- DefinitelyTyped-Repository 199–200
- Deklaration
 - Eigenschaften von Klassen 126–127
 - Funktionen als Generics 164
 - Interfaces als Generics 167
 - Klassen als Generics 169–170
 - Laufzeitwerte
 - mit declare (Schlüsselwort) 188–189
 - für globale Augmentationen 191
 - als globale Werte 190
 - zur Verschmelzung von Interfaces 190

- von Funktionen in Interfaces 113–114
- von Objekten 68
- von Vereinigungstypen 56
- Deklarationsdateien
 - ausgeben 229
 - declarationMap (Compileroption) 231
 - integrierte Deklarationen
 - Bibliotheksdateien 192–194
 - DOM-Typen 194–195
 - Zweck 192
 - für Laufzeitwerte
 - declare (Schlüsselwort) 188–189
 - globale Augmentationen 191
 - globale Werte 190
 - Verschmelzung von Interfaces 190
 - für Module 195–196
 - mit Platzhaltern 195
 - Paket-Typen
 - bereitstellen 198
 - declaration (Compileroption) 196
 - DefinitelyTyped-Repository 199–200
 - Dependency (Abhängigkeit) 197
 - Zweck 187–188
- Dekoratoren 258–259
- Dependency-Paket-Typen 197
- Diskriminante 77
- diskriminierte Vereinigungstypen
 - Beschreibung 77
 - Generics für 174
- Distributivität von bedingten Typen 279
- Dokumentation
 - in JavaScript 31, 244
 - in TypeScript 34
- DOM-Deklarationen 194–195

Duck Typing 70

dynamisch typisierte Sprachen 30

E

ECMAScript

- Modul-Import/Export 238–240

- neue Versionen 29

ECMAScript-Module (ESM) 51, 268

Editor-Funktionen (TypeScript) 38

Eich, Brendan 29

Eigenschaften

- abgebildete Typen 273–274

- von Interfaces

 - als Funktionen 113–114

 - Indexsignaturen und 116–117

 - Namenskonflikte 122

 - optional 111

 - schreibgeschützt (read-only) 111–112

 - überschrieben 120

 - verschachtelt 118

- von Klassen

 - deklarieren 126–127

 - als Funktionen 127–128

 - Initialisierungsprüfung 128–129

 - Initialisierungsprüfung deaktivieren 129

 - optionale 130

 - schreibgeschützte (readonly) 130–131

 - überschreiben 139

- von Objekten, optionale 73–74

- von Parametern 256–257

Prüfung auf überzählige Eigenschaften 71–72

- von Vereinigungstypen 56–57

eingeschränkte (constrained) Generics 177–178

Ellipse (...), Spread-Operator 86

- für Arrays 101
 - Tupel als Restparameter 104
- emitDeclarationOnly (Compileroption) 230
- Entwicklertools
 - IDEs *siehe* IDEs
 - für JavaScript 31
 - für TypeScript 35–36
- Enums
 - automatische Zuordnung numerischer Werte 261–262
 - const-Enums 263–264
 - String-Werte 262–263
 - Zweck 259–261
- erforderliche Parameter 84
- Erweitern
 - generische Klassen 171–172
- Interfaces
 - multiple 120
 - überschriebene Eigenschaften 120
 - Zweck 119
- Klassen 136
 - Eigenschaften überschreiben 139
 - Konstruktor überschreiben 137
 - Methoden überschreiben 139
 - Zuweisbarkeit 136–137
- ESM (ECMAScript-Module) 51, 268
- esModuleInterop (Compileroption) 240
- Evolving-any-Typ 48, 99
- exclude (Eigenschaft) 224
- experimentalDecorators (Compileroption) 258–259
- explizite Rückgabetypen 87
- explizite Tupel-Typen 105
- explizite Typannotationen 189
- explizite Typargumente
 - für generische Funktionen 165–166

- für generische Klassen 170–171
- explizite Vereinigungstypen von Objekten 75–76
- Exportieren
 - mit Modulen 51–52
 - mit Namespaces 265–266
 - Type-Only-Importe und -Exporte 269–270
 - über Module
 - Konfigurationsoptionen 238–242
- extends (Compileroption) 245–246

F

- falsiness 63
- Fehler
 - Syntaxfehler 46
 - Typfehler 46
 - Zuweisungsfehler 47
 - bei Funktionstypen 89
- Fehlerbehandlung
 - mit IDEs 213–214
 - Registerkarte Problems 214
 - Terminal-Compiler ausführen 215
 - Typinformationen 216–217
 - mit Typzusicherungen 155
 - useUnknownInCatchVariables (Compileroption) 237–238
- Fehlermeldungen, Zuweisungsfehler bei Schnittmengentypen 79
- Fragezeichen (?)
 - optionale Eigenschaften 73–74
 - optionale Parameter 85
- Freiheit
 - in JavaScript 30
 - in TypeScript 33
- Funktionen
 - async, Promises und 180
 - Generics für

- Deklaration 164
 - explizite Typargumente 165–166
 - multiple Typargumente 166
- in Interfaces
 - deklarieren 113–114
 - überladen 122
- Eigenschaften von Klassen als 127–128
- Parameter
 - erforderliche 84
 - optionale 85
 - Restparameter 86–87
 - Standardwert 86
 - Typanmerkungen für 83–84
- Rückgabetypen
 - Beschreibung 87
 - explizite 87
 - never (Typ) 93
 - void 91–93
- strictBindCallApply (Compileroption) 235–236
- strictFunctionTypes (Compileroption) 236
- als Typen
 - Ableitung von Parametertypen 90
 - Array-Typen und 98
 - Beschreibung 88–90
 - Klammersetzung 90
 - Typaliase 91
- Typprädikate 149–151
- Überladung 93–95

G

- Generic-Parameter für Arrays 98
- Generics
 - für diskriminierte Vereinigungstypen 174
 - eingeschränkte (constrained) 177–178

für Funktionen

 Deklaration 164

 explizite Typargumente 165–166

 multiple Typparameter 166

für Interfaces 167–169, 172

für Klassen

 Deklaration 169–170

 erweitern 171–172

 explizite Typen 170–171

 Interfaces implementieren 172

 Methoden-Generics 173

 statische Member 173

Namenskonventionen 182

richtig verwenden 181–182

Standardwerte 175–177

für Typaliase 174

Zweck 163

generische abgebildete Typen 275

generische bedingte Typen 277–278

generische Pfeilfunktionen in .tsx-Dateien 225

Geschichte

 von JavaScript 29

 von TypeScript 32

globale Augmentationen 191

globale Verschmelzung von Interfaces 190

globale Werte, Laufzeitwerte deklarieren als 190

H

Hejlsberg, Anders 29, 32

Hoare, Tony 62

Hover-Infoboxen (IDEs) mit Typinformationen 216–217

I

IDEs (Integrated Development Environments)

 Code-Navigation

- Implementierungen, suchen 207
- Referenzen, suchen 206
- Typdefinitionen, suchen 205
- Code schreiben
 - Autovervollständigung 208–209
 - Code-Aktionen 210–212
 - Import-Updates 209–210
- Fehlerbehandlung 213–214
 - Registerkarte Problems 214
 - Terminal-Compiler ausführen 215
 - Typinformationen 216–217
- Kontextmenüs und Tastaturkürzel 203–204
- if-Anweisungen, Type Narrowing durch 58
- Implementierungen, suchen im Code 207
- Implementierungssignatur 93–95
- Import-Anweisungen aktualisieren 209–210
- Importieren
 - mit Modulen 51–52
 - Type-Only-Importe und -Exporte 269–270
 - über Module, Konfigurationsoptionen 238–242
- Import-Updates beim Schreiben von Code 209–210
- include (Eigenschaft) 223
- Indexsignaturen in Interfaces
 - Eigenschaften und 116–117
 - numerisch 117
 - Zweck 115–116
- init (tsc-Befehl) 222
- Initialisierungsprüfung von Klasseneigenschaften 128–129
 - deaktivieren 129
- Installation von TypeScript 36
- integrierte Deklarationen
 - Bibliotheksdateien 192–194
 - DOM-Typen 194–195
 - Zweck 192

Integrierte Entwicklungsumgebungen *siehe* IDEs

interface (Schlüsselwort) 69

Interfaces

Aufrufsignaturen 114

Eigenschaften

optional 111

schreibgeschützt (read-only) 111–112

Erweiterung

mehrerer Interfaces 120

überschriebene Eigenschaften 120

Zweck 119

Funktionen in, deklarieren 113–114

Generics für 167–169, 172

Implementierungen suchen von 207

Indexsignaturen

Eigenschaften und 116–117

numerisch 117

Zweck 115–116

Klassen und 133–135

Typaliase im Vergleich zu 109–110

verschachtelt 118

Verschmelzung 121–122, 190

intrinsische String-Typen 284

isolatedModules (Compileroption) 241

J

JavaScript

ausgeben 227

declaration (Compileroption) 229

emitDeclarationOnly (Compileroption) 230

noEmit (Compileroption) 232

outDir (Compileroption) 227–228

Sourcemaps 231

target (Compileroption) 228–233

- Beziehung zu TypeScript 39
- Einschränkungen 30–32
- Geschichte 29
- Geschwindigkeit im Vergleich zu TypeScript 40
- Kompilieren von TypeScript in 36
- Konfigurationsoptionen 242
 - allowJs (Compileroption) 242
 - checkJs (Compileroption) 243
 - JSDoc-Unterstützung 244
- primitive Datentypen 43
- Syntaxerweiterungen
 - Dekoratoren 258–259
 - Enums 259–264
 - Nachteile 255
 - Namespaces 264–268
 - Parametereigenschaften in Klassen 256–257
 - Type-Only-Importe und -Exporte 269–270
- Typalias und 65
- Typanmerkungen 39
- vanilla 30
- JSDoc 31, 244
- JSON-Dateien 226
- jsx (Compileroption) 225
- JSX-Syntax 224–226

K

- Kernighan, Brian 288
- keyof (Typoperator)
 - abgebildete Typen 272–273
 - Beschreibung 151–152
 - eingeschränkte (constrained) Typparameter 178
- keyof typeof (Typoperator) 153
- Klammern (), in Funktionstypen 90
- Klassen

abstrakte

Beschreibung 140–141

Implementierungen suchen von 207

Eigenschaften

deklarieren 126–127

erweitern 131

als Funktionen 127–128

Initialisierungsprüfung 128–129

Initialisierungsprüfung deaktivieren 129

optionale 130

schreibgeschützte (readonly) 130–131

Erweitern

Eigenschaften überschreiben 139

Konstruktor überschreiben 137

Methoden überschreiben 139

Zuweisbarkeit 136–137

erweitern 136

Generics für

Deklaration 169–170

erweitern 171–172

explizite Typen 170–171

Interfaces implementieren 172

Methoden-Generics 173

statische Member 173

Interfaces und 133–135

Konstruktoren

Parameter 126

überschreiben 137

Member-Sichtbarkeit 141–144

Methoden 125–126

Parametereigenschaften 256–257

strictPropertyInitialization (Compileroption) 237

als Typen 132–133

Konfigurationsoptionen

- Ausgabe von JavaScript 227
 - declaration (Compileroption) 229
 - emitDeclarationOnly (Compileroption) 230
 - outDir (Compileroption) 227–228
 - Sourcemaps 231
 - target (Compileroption) 228–233
- Dateierweiterungen
 - JSON-Dateien 226
 - JSX-Syntax 224–226
- JavaScript ausgeben, noEmit (Compileroption) 232
- für JavaScript-Dateien 242
 - allowJs (Compileroption) 242
 - checkJs (Compileroption) 243
 - JSDoc-Unterstützung 244
- für Modul-Import/Export 238
 - CommonJS-Interoperabilität 240–241
 - isolatedModules (Compileroption) 241
 - module (Compileroption) 239
 - moduleResolution (Compileroption) 239
- Projektreferenzen
 - Build-Mode 249–250
 - composite (Compileroption) 248
 - references (Compileroption) 248
 - Zweck 247
- tsc (Befehl) 219–221
- TSConfig-Dateien
 - Basiskonfigurationsdateien 246
 - Dateien einschließen 223–224
 - erstellen 222
 - extends (Compileroption) 245–246
 - im Vergleich zur Kommandozeile 222
 - Zweck 221–222
- Typechecking
 - lib (Compileroption) 232

skipLibCheck (Compileroption) 233

strict-Mode 233–238

Konstituenten 56

Konstruktoren (von Klassen)

Parameter 126

überschreiben 137

Kontextmenüs in IDEs 203–204

kontravariante Funktionsparameter 237

L

Laufzeit-Syntaxerweiterungen *siehe* Syntaxerweiterungen

Laufzeitwerte, deklarieren

mit declare (Schlüsselwort) 188–189

für globale Augmentationen 191

als globale Werte 190

zur Verschmelzung von Interfaces 190

lib (Compileroption) 232, 233

Literale als Literaltyp 160

Literaltypen

Beschreibung 59–60

Zuweisbarkeit 61

M

Member-Sichtbarkeit in Klassen 141–144

Methoden

abgebildete Typen 273–274

Generics für 173

von Interfaces

Funktionsdeklarationen 113–114

überladen 122

von Klassen

Beschreibung 125–126

überschreiben 139

Milliarden-Dollar-Fehler 61–62

Modifier, ändern 274–275

Modulauflösung 239

Module

- Beschreibung 51–52

- Deklarationsdateien 195–196

- erweitern (extending) 246

- im Vergleich mit Namespaces 265, 268

- Konfigurationsoptionen 238

 - CommonJS-Interoperabilität 240–241

 - isolatedModules (Compileroption) 241

 - module (Compileroption) 239

 - moduleResolution (Compileroption) 239

- module (Compileroption) 239

- moduleResolution (Compileroption) 239

- multidimensionale Arrays 99

- multiple Interfaces

 - erweitern 120

 - Implementierung in Klassen 134–135

- multiple Typargumente für generische Funktionen 166

N

- Namenskonflikte beim Verschmelzen von Interfaces 122

- Namenskonventionen

 - für Generics 182

 - für Typparameter 164

- Namensräume *siehe* Namespaces

- Namespaces

 - Exporte 265–266

 - im Vergleich mit Modulen 265, 268

 - in Typdefinitionen 267

 - verschachtelte 267

 - Zweck 264–265

- Navigieren im Code

 - Implementierungen, suchen 207

 - Referenzen, suchen 206

- Typdefinitionen, suchen 205
- never (Typ) 79
 - abgebildete Typen und 282
 - bedingte Typen und 281
 - als Funktionsrückgabety 93
 - Schnittmengen- und Vereinigungstypen 281
 - Zweck 281
- Nicht-null-Zusicherung 156–157
- noEmit (Compileroption) 232
- noImplicitAny (Compileroption) 234
- null (Typ)
 - als primitiver Datentyp 43, 60
 - strictNullChecks (Compileroption) 237
- number (primitiver Datentyp) 43, 60
- numerische Indexsignaturen 117

O

- Objekte
 - Beschreibung 67–68
 - Deklaration 68
 - im Vergleich zu primitiven Datentypen 45
 - Interfaces *siehe* Interfaces
 - schreibgeschützte 161
 - strukturelle Typisierung
 - Beschreibung 69–70
 - optionale Eigenschaften 73–74
 - Prüfung auf überzählige Eigenschaften 71–72
 - verschachtelte Objekte 72–73
 - Verwendungsprüfung 70
 - Typalias 68
 - Typformen 50–51
 - Vereinigungstypen
 - abgeleitete 74
 - diskriminierte 77

- explizite 75–76
 - Type Narrowing 76
- optionale Eigenschaften
 - von Interfaces 111
 - von Klassen 130
 - von Objekten 73–74
- optionale Parameter 85
- outDir (Compileroption) 227–228
- Output *siehe* Ausgabe von JavaScript

P

Paket-Typen

- bereitstellen 198
- declaration (Compileroption) 196
- DefinitelyTyped-Repository 199–200
- Dependency (Abhängigkeit) 197

Parameter

- Eigenschaften 256–257
- erforderliche 84
- Klassenkonstruktoren 126
- optionale 85
- Restparameter
 - als Array 86–87
 - Tupel als 104
- Standardwert 86
- Typ ableiten 90
- Typanmerkungen für 83–84

Pipe (|)-Operator 56

Platzhalter-Moduldeklarationen 195

Playground 32–36

Pretty-Mode (tsc-Befehl) 220

primitive Datentypen 43–44

- im Vergleich zu Objekten 45
- Literale als Literaltyp 160

Literaltypen

Beschreibung 59–60

Zuweisbarkeit 61

never (Typ) 79

private Klassenmember 141–143

Problems-Registerkarte (VS Code) 214

Programmiersprache, Definition 32

Projektreferenzen

Build-Mode 249–250

composite (Compileroption) 248

references (Compileroption) 248

Zweck 247

Promises

async-Funktionen und 180

erstellen 179–180

Zweck 179

protected (geschützte) Klassenmember 141–143

Prüfung auf überzählige Eigenschaften bei struktureller Typisierung 71–72

public (öffentliche) Klassenmember 141–143

R

Raute (#), private Klassenmember 141–143

readonly (schreibgeschützte), Klasseneigenschaften 130–131

Refaktorisieren mit Code-Aktionen 212

references (Compileroption) 248

Referenzen, suchen im Code 206

Remapping von Schlüsseln bei abgebildeten Typen 286

resolveJsonModule (Compileroption) 226

Restparameter

als Array 86–87

Tupel als 104

Restriktionen in TypeScript 33

Rückgabetypen

Beschreibung 87

- explizite 87
- never (Typ) 93
- void 91–93

runde Klammern (), Arrays 98

S

Schlüssel abgebildeter Typen remappen 286–287

Schnittmengentypen

- Beschreibung 78
- Gefahren von 79–80
- never (Typ) und 281

Schreiben von Code, Code-Aktionen, Zweck 210

schreibgeschützte Eigenschaften von Interfaces 111–112

schreibgeschützte Objekte 161

Sichtbarkeit von Klassenmitgliedern 141–144

skipLibCheck (Compileroption) 233

Skripte 51–52

sourceMap (Compileroption) 231

Sourcemaps 231

Sprachdienst, Definition 32

Spread-Operator (...) 86

- für Arrays 101
- Tupel als Restparameter 104

Standardparameter 86

Standardwerte, Generics 175–177

static (Schlüsselwort) 144

statische Klassenmember, Generics für 173

strict (Compileroption) 234

strictBindCallApply (Compileroption) 235–236

strictFunctionTypes (Compileroption) 236

strict-Mode (Typechecking) 233–238

strictNullChecks (Compileroption) 237

strictPropertyInitialization (Compileroption) 237

strikte Nullprüfung

- Milliarden-Dollar-Fehler 61–62
 - ohne Anfangswert 64
 - durch Type Narrowing mittels Wahrheitsprüfung 63
- string (primitiver Datentyp) 43, 60
- Strings, Template-Literaltypen
 - intrinsische Typen 284
 - Schlüssel abgebildeter Typen remappen 286–287
 - Template-Literalschlüssel 285
 - Zweck 283–284
- String-Werte in Enums 262–263
- strukturelle Typisierung
 - Beschreibung 69–70
 - optionale Eigenschaften 73–74
 - Prüfung auf überzählige Eigenschaften 71–72
 - verschachtelte Objekte 72–73
 - Verwendungsprüfung 70
- Subklassen 136
 - Eigenschaften überschreiben 139
 - Konstruktor überschreiben 137
 - Methoden überschreiben 139
 - Zuweisbarkeit 136–137
- Suchen
 - Implementierungen im Code 207
 - Referenzen im Code 206
 - Typdefinitionen im Code 205
- super (Schlüsselwort) 137
- symbol (primitiver Datentyp) 43
- Syntaxerweiterungen
 - Dekoratoren 258–259
 - Enums
 - automatische Zuordnung numerischer Werte 261–262
 - const-Enums 263–264
 - String-Werte 262–263
 - Zweck 259–261

Nachteile 255

Namespaces

- Exporte 265–266

- im Vergleich mit Modulen 265, 268

- in Typdefinitionen 267

- verschachtelte 267

- Zweck 264–265

Parametereigenschaften in Klassen 256–257

Type-Only-Importe und -Exporte 269–270

Syntaxfehler 46

Syntaxgültigkeit, Typfehler und 37

T

target (Compileroption) 228–233

Tastaturkürzel

- Code-Aktionen in VS Code öffnen 210

- in IDEs 203–204

TC39 29, 39

Template-Literalschlüssel 285

Template-Literaltypen

- intrinsische Typen 284

- Schlüssel abgebildeter Typen remappen 286–287

- Zweck 283–284

Terminal-Compiler, ausführen 215

ternäre Anweisung, Type Narrowing durch 59

Thenable 180

Top Type 147–149

tsc (Befehl) 36–37

- Build-Mode 249–250

- init 222

- Konfigurationsoptionen 219–221

- Pretty-Mode 220

- TSCConfig-Dateien im Vergleich zu 222

- Überwachungsmodus 220–221

@ts-check (Kommentar) 243

tsconfig.json-Datei *siehe* TSConfig-Dateien

TSConfig-Dateien 37

Basiskonfigurationsdateien 246

Dateien einschließen 223–224

erstellen 222

extends (Compileroption) 245–246

im Vergleich zur Kommandozeile 222

Projektpreferenzen

Build-Mode 249–250

composite (Compileroption) 248

references (Compileroption) 248

Zweck 247

Zweck 221–222

.tsx-Dateien, generische Pfeilfunktionen in 225

Tupel

ableiten durch Typinferenz 105

Beschreibung 102

const-Zusicherungen 106–107

explizite Typen 105

als Restparameter 104

Zuweisbarkeit 103

Typaliase

Beschreibung 64

für Funktionen 91

Generics für 174

Interfaces im Vergleich zu 109

JavaScript und 65

kombinieren 65

für Objekte 68

Typanmerkungen

für Arrays 98

Beschreibung 48–50

explizite 189

- für Funktionsrückgabetypen 87
- für Funktionsparameter 83
- in JavaScript 39
- Typzusicherungen im Vergleich zu 158
- für Vereinigungstypen 60

Typannotationen *siehe* Typanmerkungen

Typargumente

- für generische Klassen, Klassen erweitern 171–172
- für generische diskriminierte Vereinigungstypen 174
- für generische Funktionen
 - explizite Argumente 165–166
 - multiple Argumente 166
- für generische Typaliase 174

Typdefinitionen

- im Code suchen 205
- Namespaces in 267

Type Casts *siehe* Typzusicherungen

Typechecking

- Definition 32
- in TypeScript 33, 43–45
- Konfigurationsoptionen
 - lib (Compileroption) 232
 - skipLibCheck (Compileroption) 233
 - strict-Mode 233–238

Type Guard 57

Typen

- abgebildete Typen
 - auf Basis vorhandener Typen 272–273
 - generische 275
 - never (Typ) und 282
 - Signaturen und 273–274
 - Zugriffs-Modifier ändern 274–275
 - Zweck 271–272
- Arrays

- Evolving-any-Arrays 99
- Funktionsstypen und 98
- multidimensionale Arrays 99
- Typanmerkungen für 98
- Vereinigungstypen und 98–99
- bedingte Typen
 - abgebildete Typen und 280
 - abgeleitete 279
 - Distributivität 279
 - generische 277–278
 - never (Typ) und 281
 - Zweck 277
- Beschreibung 43–45
- Bottom Type 80
- DefinitelyTyped-Repository 199–200
- Deklarationsdateien *siehe* Deklarationsdateien
- Duck Typing 70
- evolving any 48
- Funktionen
 - Ableitung von Parametertypen 90
 - Array-Typen und 98
 - Beschreibung 88–90
 - Klammersetzung 90
 - Typaliase 91
- Funktionsrückgabetypen
 - Beschreibung 87
 - explizite 87
 - never (Typ) 93
 - void 91–93
- Generics
 - für diskriminierte Vereinigungstypen 174
 - eingeschränkte (constrained) 177–178
 - für Funktionen 164–166
 - für Interfaces 167–169, 172

- für Klassen 169–173
- Namenskonventionen 182
- richtig verwenden 181–182
- Standardwerte 175–177
 - für Typaliase 174
- Informationen in IDEs 216–217
- Klassen als 132–133
- Literaltypen
 - Beschreibung 59–60
 - Zuweisbarkeit 61
- never (Typ)
 - abgebildete Typen und 282
 - bedingte Typen und 281
 - Schnittmengen- und Vereinigungstypen 281
 - Zweck 281
- Objekte
 - abgeleitete Vereinigungstypen 74
 - Beschreibung 67–68
 - Deklaration 68
 - diskriminierte Vereinigungstypen 77
 - explizite Vereinigungstypen 75–76
 - Interfaces *siehe* Interfaces
 - Typaliase 68
 - Type Narrowing 76
- primitive Datentypen 43–44
- Schnittmengentypen
 - Beschreibung 78
 - Gefahren von 79–80
- strikte Nullprüfung
 - Milliarden-Dollar-Fehler 61–62
 - ohne Anfangswert 64
 - durch Type Narrowing mittels Wahrheitsprüfung 63
- strukturelle Typisierung
 - Beschreibung 69–70

- optionale Eigenschaften 73–74
- Prüfung auf überzählige Eigenschaften 71–72
- verschachtelte Objekte 72–73
- Verwendungsprüfung 70
- Template-Literaltypen
 - intrinsische Typen 284
 - Schlüssel abgebildeter Typen remappen 286–287
 - Template-Literalschlüssel 285
 - Zweck 283–284
- Top Type 147–149
- Tupel als
 - const-Zusicherungen 106–107
 - explizite Tupel-Typen 105
- Type Narrowing
 - durch bedingungsabhängige Prüfungen 58
 - Beschreibung 55, 57
 - durch typeof (Typoperator) 59
 - durch Zuweisung 57–58
- Vereinigungstypen
 - Array-Typen und 98–99
 - Beschreibung 55–56
 - deklarieren 56
 - Eigenschaften 56–57
 - kombiniert mit Schnittmengentypen 78
 - Typalias 64–66
 - Typanmerkungen 60
- Zuweisbarkeit 47
- Type Narrowing
 - durch bedingungsabhängige Prüfungen 58
 - Beschreibung 55, 57
 - Klasseneigenschaften 131
 - durch Wahrheitsprüfung 63
 - durch typeof (Typoperator) 59
 - von Objekten 76

durch Zuweisung 57–58

typeof (Typoperator)

Beschreibung 153

Type Narrowing durch 59

Type-Only-Importe und -Exporte 269–270

TypeScript

Dokumentation 34

Editor-Funktionen 38

Einschränkungen 38–40

Entwicklertools 35–36

Freiheit und Restriktion 33

Geschichte 32

Geschwindigkeit im Vergleich zu JavaScript 40

installieren 36

kompilieren 36

lokal ausführen 37–38

Module 51–52

Playground 32–36

ständige Änderungen 40

Typechecking 33, 43–45

Typsystem 45

Zweck 32

Typfehler 37, 46

Typformen 50–51

Typoperationen

abgebildete Typen 271–276

bedingte Typen 277–280

Komplexität von 288

never (Typ) 281–282

Template-Literaltypen 283–287

Typoperatoren

keyof 151–152

keyof typeof 153

typeof 153

Zweck 151

Typparameter

für eingeschränkte (constrained) Generics 177–178

für generische Funktionen 164

für generische Interfaces 167–169

für generische Klassen

 Deklaration 169–170

 explizite Typen 170–171

 Interfaces implementieren 172

 Methoden-Generics 173

 statische Member 173

für generische Standardwerte 175–177

Namenskonventionen 164

Zweck 163–164

Typprädikate 149–151

Typprüfung *siehe* Typechecking

Typsystem 45

Typzusicherungen

 doppelte 159

 Fallstricke 157

 Fehlerbehandlung mit 155

 Nicht-null-Zusicherung 156–157

 Typanmerkungen im Vergleich zu 158

 Zuweisbarkeit 158

 Zweck 154–155

U

Überladungssignatur 93–95

Überladung von Interface-Funktionen 122

Überschreiben

 Interface-Eigenschaften 120

 Klasseneigenschaften 139

 Klassenkonstruktoren 137

 Klassenmethoden 139

- Überwachungsmodus (tsc-Befehl) 220–221
- Umbenennen mit Code-Aktionen 211
- undefined (primitiver Datentyp) 43, 60
 - im Vergleich zu void als Rückgabebetyp 92
 - optionale Eigenschaften im Vergleich zu 74
 - für optionale Parameter 85
 - für Standardparameter 86
 - wegen fehlenden Anfangswerts 64
- unknown (Typ)
 - Beschreibung 148
 - useUnknownInCatchVariables (Compileroption) 237–238
- useUnknownInCatchVariables (Compileroption) 237–238

V

- Vanilla-JavaScript 30

- Variablen

- siehe auch* Typen

- const 60

- evolving any 48

- globale Augmentationen 191

- ohne Anfangswert 64

- Typanmerkungen 48–50

- Typformen 50–51

- Zuweisbarkeit von Typen 47

- Vereinigungstypen

- Array-Typen und 98–99

- Beschreibung 55–56

- deklarieren 56

- diskriminierte 77, 174

- Distributivität von bedingten Typen 279

- Eigenschaften 56–57

- kombiniert mit Schnittmengentypen 78

- never (Typ) und 281

- von Objekten

- abgeleitete 74
- explizite 75–76
- Type Narrowing 76
- Typaliase
 - Beschreibung 64
 - JavaScript und 65
 - kombinieren 65
- Typanmerkungen 60
- verschachtelte Interfaces 118
- verschachtelte Namespaces 267
- verschachtelte Objekte 72–73
- Verschmelzung von Interfaces 121–122, 190
- Verwendungsprüfung bei struktureller Typisierung 70
- void als Rückgabebetyp 91–93
- VS Code
 - siehe auch* IDEs
 - Code-Aktionen öffnen 210
 - TypeScript-Unterstützung 38

W

Watch-Mode *siehe* Überwachungsmodus

Z

- Zielversionen für Bibliotheks-Deklarationsdateien 193–194
- Zugriffs-Modifizier, ändern 274–275
- Zusammenführung von Arrays mit Spread-Operator 101
- Zusicherung *siehe* Typzusicherungen sowie const-Zusicherungen
- Zuweisbarkeit
 - Beschreibung 47
 - Fehler 47
 - bei Funktionstypen 89
 - Fehlermeldungen bei Schnittmengentypen 79
 - von Subklassen 136–137
 - von Tupeln 103
 - von Typzusicherungen 158

von Literaltypen 61
zuweisungsbasiertes Type Narrowing 57–58