D Einführung in TypeScript

In diesem Anhang geben wir dir eine kurze Einführung in die Sprache TypeScript, die unter anderem ein Typsystem für JavaScript-Code zur Verfügung stellt und sehr gute Unterstützung für den Einsatz in React-Anwendungen bietet.

D.1 Motivation

JavaScript ist eine dynamisch getypte Sprache. Das bedeutet, dass einer Variablen nicht nur jederzeit ein anderer Wert zugewiesen werden kann, sondern dass dieser Wert auch von unterschiedlichen Typen sein kann. Eine Variable `name` kann somit beispielsweise zunächst vom Typ `string` sein und danach zum Typ `number` oder sogar zu einer Funktion werden:

```
let name = "Klaus";      // typeof name === string
name = 77;               // typeof name === number
name = function greet() { return "hello" }
                         // typeof name === function
```

Während die dynamische Typisierung auf der einen Seite sehr praktisch ist, weil wir uns keine Gedanken um die Typen machen müssen, kann sie auf der anderen Seite fehleranfällig und schwer verständlich sein.

Sehen wir uns als Beispiel die Signatur einer Funktion an:

```
function sayHello(person) { ... };
```

Ohne in den Code zu schauen, wissen wir nicht, was `person` sein soll. Ein String? Ein Objekt? Darf der Parameter `null` sein oder gar ganz weggelassen werden? Weiteres Problem: Was mag die Funktion zurückgeben? Auch das ist nicht ersichtlich. Ohne Dokumentation bleibt uns nur der Blick in den Quellcode.

Wenn wir an unsere React-Anwendung denken, haben wir es mit ähnlichen Problemen zu tun. Nehmen wir beispielsweise den State einer Komponente:

```
const [name, setName] = React.useState("");
```

Hier können wir zumindest davon ausgehen, dass der State ein String sein soll. Aber dürfte er auch `null` annehmen? Kann die Anwendung damit umgehen?

Das gleiche Problem gibt es mit den Properties einer Komponente:

```
function Greeter(props) { ... }
```

Von außen betrachtet können wir nicht erkennen, ob und welche Properties diese Komponente erwartet. Aber auch hier ist das Problem, dass ein fehlerhafter Aufruf der Komponente (zu wenig oder falsche Properties) erst zur Laufzeit bemerkt wird und dann auch nur, wenn die Anwendung den fehlerhaften Zustand überhaupt durchläuft, das heißt potenziell nur in sehr wenigen Situationen, die schwer zu finden bzw. zu reproduzieren sind. Darüber hinaus sind Properties read-only. Dennoch würde folgender Code »funktionieren«, also ausgeführt werden, wenngleich es zur Laufzeit zu einem Fehler kommt:

```
function Greeter(props) {
  props.name = ""; // Verboten: Properties sind read-only!
}
```

Linter

Die Probleme, die durch die dynamische Typisierung auftreten, können auf zwei Wegen gelöst werden: durch statische Codeanalyse (Linter) oder durch Type Checker. Ein Linter wie ESLint kann durch statische Codeanalyse zumindest einige der genannten Probleme feststellen, zum Beispiel wenn eine Variable verwendet, aber nicht zuvor definiert wurde:

```
function greet(name) {
  return `Hello, ${firstname}` // firstname not defined
}
```

Hier würde der Linter erkennen, dass `firstname` nicht definiert ist (ein möglicher Programmierfehler, vermutlich ist `name` gemeint).

Allerdings kann der Linter nicht feststellen, ob die Funktion `greet` korrekt aufgerufen wurde, da der Linter zwar weiß, dass es einen Parameter `name` gibt, aber nicht von welchem Typ dieser sein soll und ob er `null` annehmen kann oder gar weggelassen werden darf.

Type Checker

Ein konsequenterer Ansatz zur Vermeidung dieser Probleme ist ein statischer Type Checker. Hier gibt es für JavaScript zwei Werkzeuge, die beide auch mit React, also JSX-Code, umgehen können: Flow und

TypeScript. Flow ist ein reiner Type Checker, der von Facebook entwickelt und auch für den React-Code selbst verwendet wird.

TypeScript

TypeScript ist eine von Microsoft entwickelte Sprache, die auf JavaScript aufbaut. Aufgrund der hohen Verbreitung von TypeScript zeigen wir dir in diesem Anhang, wie TypeScript funktioniert und wie du es in deiner JavaScript- bzw. React-Anwendung einsetzen kannst.

Unabhängig davon, welchen Type Checker, Flow oder TypeScript, du verwendest, werden dir von diesem bereits zur Entwicklungszeit, also zum Beispiel beim Kompilieren deiner Anwendung durch Webpack, oder sogar schon in der Entwicklungsumgebung Fehler angezeigt. Du kannst sie also vermeiden bzw. beheben, bevor du deine Anwendung zum Testen ausführst.

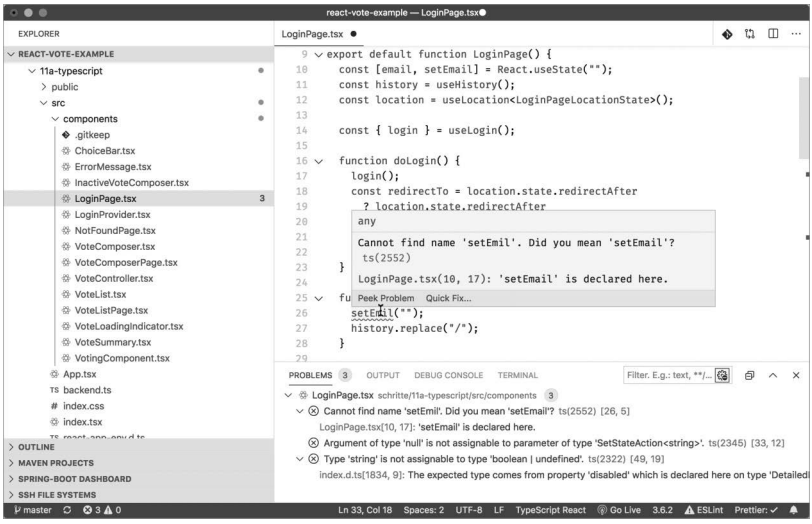

Abb. D–1
TypeScript in Visual Studio Code

Im Folgenden geben wir dir eine kurze, von React unabhängige Einführung in die Sprache TypeScript. Eine ausführlichere Beschreibung findest du unter anderem in dem Online-Handbuch »Deep Dive«-TypeScript[1].

D.2 Die Sprache TypeScript

TypeScript ist eine von Microsoft entwickelte Sprache, die auf JavaScript aufbaut. TypeScript ist dabei eine Obermenge von JavaScript, sodass prinzipiell jeder gültige JavaScript-Code auch gültiger TypeScript-Code ist. Allerdings erweitert TypeScript die Sprache JavaScript um einige Features. Das aus unserer Sicht wichtigste Feature ist der

1 *https://basarat.gitbooks.io/typescript/content/docs/getting-started.html*

statische Type Checker, dazu gleich mehr. Daneben gibt es aber beispielsweise auch Aufzählungstypen, die in JavaScript nicht existieren (Enums), sowie Sichtbarkeiten in Klassen (Methoden und Properites einer Klasse können in TypeScript als `private` oder `protected` deklariert sein und sind dann entweder gar nicht außerhalb der Klasse oder nur in Unterklassen sichtbar).

Zielversionen

Da durch die Spracherweiterung der TypeScript-Code nicht direkt vom Browser ausgeführt werden kann, bringt TypeScript einen Compiler mit, der den Sourcecode (ähnlich wie Babel) in verschiedene JavaScript-Versionen zurückübersetzen kann. Dabei wird geprüft, ob alle Typen korrekt verwendet werden und die proprietäre Syntax wird entfernt. Da der Compiler auch in der Lage ist, JSX-Code in JavaScript zu übersetzen, kann beim Einsatz von TypeScript in React-Projekten auf Babel grundsätzlich verzichtet werden (Create React App verwendet Babel zusätzlich wegen weiterer Plug-ins).

Das wesentliche Feature von TypeScript ist dessen statisches Typsystem, mit dem eine JavaScript-Anwendung typisiert werden kann.

D.3 Grundlagen des TypeScript-Typsystems

Type Inference

Typen in TypeScript können entweder eingebaute Typen wie `string`, `number`, `boolean` oder ein Array sein oder auch eigene Typen. Die Festlegung von Typen kann explizit oder implizit erfolgen. Dort wo keine Typen explizit angegeben werden, versucht TypeScript die gültigen Typen zu ermitteln (»Type Inference«). Wo das nicht möglich ist, müssen die Typen von dir angegeben werden. An allen anderen Stellen ist es optional. Das führt in der Praxis dazu, dass an nur erstaunlich wenigen Stellen überhaupt zwingend Typangaben erforderlich sind.

Im folgenden Beispiel »weiß« TypeScript, dass `name` vom Typ `string` ist, da der Variablen beim Initialisieren ein String zugewiesen wird. Die darauffolgende Zuweisung einer Zahl an die Variable verbietet TypeScript aus diesem Grund:

Beispiel: Type Inference

```
let name = "Klaus"
name = 77; // Type '77' is not assignable to type 'string'
```

Alternativ kannst du eine Typangabe explizit angeben. Dazu schreibst du durch einen Doppelpunkt getrennt den Typ hinter den Namen der Variablen:

```
let name: string = "Klaus";
name = 77; // ERROR: Type '77' is not assignable to type 'string'
```

Funktionsargumente

Bei Funktionen musst du zwingend für alle Argumente einen Typ angeben, da TypeScript die Typen hier nicht selbst ableiten kann. Auch hier werden die Typangaben mit Doppelpunkt getrennt hinter die einzelnen Funktionsparameter geschrieben. Den Rückgabewert wiederum kann TypeScript selbst herleiten. So erkennt TypeScript im folgenden Beispiel, dass der Rückgabewert vom Typ `string` ist:

```
function sayHello(name: string) {
  return `Hello, ${name.toUpperCase()}`;
}

// Alternativ als Arrow-Funktion:
const sayHello = (name: string) => `Hello, ${name.toUpperCase()}`;

let greet = sayHello("Klaus");

greet = 99;   // ERROR: Type '99' is not assignable
              // to type 'string'

sayHello(77); // ERROR: Argument of type '77' is not assignable
              // to parameter of type 'string'.
```

Einer Variablen oder einem Parameter können in TypeScript auch mehr als ein Typ zugewiesen werden. Das ist einerseits notwendig, weil JavaScript solche Konstrukte grundsätzlich zulässt, und andererseits ist auch `null` und `undefined` in TypeScript ein eigener Typ. Aus diesem Grund akzeptiert die oben gezeigte `sayHello`-Funktion auch kein `null` als Parameter. Auch kann der Parameter nicht ganz weggelassen werden. Beides würde zu einem Compile-Fehler führen. Daraus ergibt sich auch, dass TypeScript sicher sagen kann, dass der Aufruf von `toUpperCase` in jedem Fall funktioniert, denn `name` ist immer ein `string`, niemals `null` oder `undefined` oder von einem anderen Typ.

Union Types

Möchtest du einer Variablen oder einem Parameter mehr als einen Typ zuweisen, kannst du mehrere Typen mit | separiert hinschreiben (das ist dann ein »Union Type«). Wenn du ausdrücken möchtest, dass ein Parameter optional ist, kannst du ihn mit einem Fragezeichen markieren:

```
export function sayHello(name?: string | null) {
  return `Hello, ${name.toUpperCase()}`;
}

sayHello("Klaus") // OK
sayHello(null); // OK
sayHello(); // OK
sayHello().toUpperCase(); // OK: sayHello liefert string zurück
```

Nun wird allerdings die Funktion selbst nicht mehr kompilieren, da `toUpperCase` auf `name` aufgerufen wird, und `name` kann ja nun auch `null` oder `undefined` sein.

Du musst die Funktion also entsprechend anpassen, zum Beispiel:

```
function sayHello(name?: string | null) {
  if (name === null || name === undefined) {
    return null;
  }
  return `Hello, ${name.toUpperCase()}`;
}
```

In dieser Implementierung liefert die Funktion nun `null` zurück, wenn `name` weggelassen oder `null` übergeben wurde. Dadurch verändert sich allerdings auch der Rückgabetyp der Funktion! Die Funktion gibt nun einen `string` oder `null` zurück:

```
function sayHello(name?: string | null) { ... }

sayHello("Klaus").toUpperCase(); // ERROR: Object is
                                 // possibly 'null'
```

Rückgabewerte von Funktionen

Wie du siehst, haben wir durch die Änderung der Funktion ein Problem beim Verwender der Funktion erzeugt. Dieser konnte in der vorhergehenden Variante davon ausgehen, immer einen `string` zu erhalten, nun kann plötzlich auch `null` zurückkommen. Dieses Verhalten kann so gewollt sein, vielleicht ist es aber auch nur versehentlich entstanden und die Funktion sollte eigentlich weiterhin einen String zurückliefern. Um solche Fälle zu verhindern, kannst du bei Methoden explizit einen Rückgabewert angeben. In diesem Fall wird der Fehler (dass nun `null` zurückgegeben wird) in der Funktion angezeigt und nicht mehr beim Aufrufer. Das ist insbesondere bei Funktionen, die Teil einer öffentlichen Schnittstelle sind, sinnvoll, um zu vermeiden, dass eine Funktion »versehentlich« inkompatibel zu einer vorangegangenen Version wird:

Beispiel: Typen für Rückgabewerte

```
function sayHello(name?: string | null): string {
  if (name === null || name === undefined) {
    return null; // ERROR: Type 'null' is not assignable
                 // to type 'string'
  }
  return `Hello, ${name.toUpperCase()}`;
}

// Alternativ als Arrow-Funktion:
const sayHello = (name?: string | null): string => { ... }

// Verwendung
sayHello("Klaus").toUpperCase(); // Kein Fehler
```

Um den Fehler in der Funktion zu beseitigen, könnten wir nun beispielweise statt `null` einen Leerstring zurückgeben.

In unserem Fall möchten wir aber dabeibleiben, dass die Funktion auch `null` zurückliefern kann. Das bedeutet aber, dass der Aufrufer angepasst werden muss und explizit auf `null` prüfen muss:

```
function sayHello(name?: string | null) {
  if (name === null || name === undefined) {
    return null;
  }
  return `Hello, ${name.toUpperCase()}`;
}

const hello = sayHello("Klaus");

if (hello !== null) {
 const greet = hello.toUpperCase(); // OK
 // ...
}
```

Durch diese Prüfung weiß TypeScript, dass im `if`-Block der Typ von `hello` nur noch `string` sein kann. Der andere erlaubte Typ (`null`) wurde durch die `if`-Abfrage ausgeschlossen.

Type Narrowing

Dieses Feature nennt sich »Type Narrowing«, da die Typen einer Variablen durch entsprechende Abfragen »eingeengt« bzw. eingeschränkt werden. Das funktioniert nicht nur mit Prüfungen auf `null`, sondern auch mit anderen Typen. Schau dir dazu das folgende Beispiel an, in dem die `sayHello`-Funktion nun auch eine `number` als Parameter akzeptiert. Um das Beispiel möglichst verständlich zu machen, ist es etwas ausführlicher implementiert, als es eigentlich sein müsste (`null`- und `typeof`-Check könnten auch zusammengelegt werden):

Beispiel: Type Narrowing

```
function sayHello(name: string | number | null) {

  // Der Typ von name ist hier string, number oder null
  if (typeof name === "number") {

   // Der Typ von name ist hier number
    return null;
  }

// Der Typ von name ist hier nur noch name oder null
  if (name === null) {
    // name ist hier null
    return null;
  }

  // Der Typ von name ist hier nur noch string
  return `Hello, ${name.toUpperCase()}`;
}
```

Du siehst, wie sich der Typ in den Verzweigungen der Funktion »einschränkt«.

In den Beispielen oben, in denen wir zwischen einem konkreten Typ (`string` oder `number`) und `null` und/oder `undefined` unterscheiden wollten, haben wir immer eine explizite Typabfrage gemacht:

```
name === undefined || name === null
```

Eine häufige Fehlerquelle ist es, stattdessen den logischen Nicht-Operator (`!`) zu verwenden. In diesem Fall würde aber durch die implizite Typkonvertierung in JavaScript beispielsweise auch ein Leerstring oder eine `0` zu `true` ausgewertet werden, wie das folgende Beispiel zeigt:

```
function sayHello(name: string | number | null) {

  if (!name) {

   // Der Typ von name ist hier immer noch string, number oder
   // null
    return null;
  }

  // Hier ist name nur noch string oder number

  ...

}
```

Typinformationen nur zur Build-Zeit

Keine Laufzeitinformationen

Die Typinformationen in deiner Anwendung werden vom TypeScript-Compiler beim Build und in der IDE überprüft und dir werden entsprechende Fehler ausgegeben.

Beim Kompilieren werden allerdings die Typangaben vom Compiler vollständig entfernt, sodass diese nicht im erzeugten JavaScript-Code vorhanden sind. Damit stehen die Typinformationen in keiner Form zur Laufzeit zur Verfügung. Eine Art Reflection-API, wie beispielsweise aus Java oder C# bekannt, gibt es in TypeScript nicht. Typinformationen werden in TypeScript ausschließlich zur Build-Zeit verwendet.

D.3.1 Eigene Typen definieren

In TypeScript kannst du eigene Typen definieren, um Strukturen von Objekten zu beschreiben. Dies geschieht über die Schlüsselwörter `type` oder `interface`. Da Semantik, Funktionsweise und Funktionsumfang der beiden Schlüsselwörter mittlerweile fast identisch sind, verwen-

den wir in diesem Buch und in unserer Beispielanwendung ausschließlich `type`[2].

Damit kannst du erzwingen, dass eine Variable oder ein Parameter ein Objekt erhält, das genauso wie beschrieben vorliegt. Dazu gibst du die Namen der Objekteigenschaften sowie deren Typen an:

```
type Person = {
  firstName: string;
  age: number;
}
```

Auch hier gilt, dass die Eigenschaften des Objekts nicht `null` oder `undefined` sein dürfen, solange das nicht explizit angegeben ist:

```
const klaus:Person = { firstName: "Klaus", age: 32 };
const susi:Person = { firstName: "susi",  age: null}; // ERROR: Type
'null' is not assignable to type 'number'
```

Beim Verwenden eines Objekts prüft TypeScript, ob das Objekt auf den erwarteten Typ passt. So musst du zum Beispiel bei der Deklaration der Variablen nicht explizit hinschreiben, dass die Variable vom Typ `Person` ist, du kannst sie aber trotzdem an eine Funktion übergeben, die eine `Person` erwartet, oder einer Variablen zuweisen, die explizit eine `Person` erwartet. Auch wenn der Rückgabewert einer Funktion der erwarteten Struktur eines Typs entspricht, kannst du diesen einer Variablen vom entsprechenden Typ zuweisen:

```
function greet(p: Person) {
  const g = `Hello, ${p.name}`; // ERROR: Property 'name' does not
                                // exist on type 'Person'

  return `Hello, ${p.firstName}`;  // OK
}

const maja = { firstName: "Maja", age: 40 };

// Zuweisung an Variablen
const majaThePerson: Person = maja;

// Verwendung der Funktion
greet(maja);

// Direkte Verwendung ohne Typangabe
greet({
  firstName: "Karl",
  age: 42
```

2 Eine Beschreibung der Unterschiede zwischen type und interface findest du im neuen TypeScript-Handbuch, das gerade geschrieben wird: *https://microsoft.github.io/TypeScript-New-Handbook/chapters/everyday-types/#interface-vs-alias*.

```
});

// createPerson liefert ein Objekt zurück
function createPerson(name: string, alter: number) {
  return { name: firstName, age: alter };
}

// Struktur des Rückgabewerts von createPerson
// passt auf Person
const cathy: Person = createPerson("Cathy", 35);
```

Arrays

Auch Arrays lassen sich mit TypeScript typsicher beschreiben. Dazu kannst du den Typnamen, der ein Array enthalten soll, gefolgt von einem leeren Paar eckiger Klammern hinschreiben oder die generische Schreibweise verwenden:

Beispiel: Arrays

```
function greetAll(persons: Person[]){
  return persons.map(person => {
    // Typ von "person" ist hier "Person"
    const lastName = person.lastName;
     // ERROR: Property 'lastName' does not exist on type 'Person'

    return person.firstName; // OK
  });
}

// Alternativ:
function greetAll(persons: Array<Person>) { ... }

const klaus: Person = { firstName: "Klaus", age: 32 };
const susi = { firstName: "Susi", age: 34 };

// Alle drei Parameter sind strukturidentisch zu Person
const greetings = greetAll([
  klaus,
  susi,
  {
    firstName: "Karl",
    age: 42
  }
]);

console.log(greetings[1]); // "Susi"
console.log(greetings.toUpperCase()); // FEHLER: Property
'toUpperCase' does not exist on type 'string[]'
```

Im Beispiel oben siehst du, wie TypeScript den Typ auch in der `map`-Funktion automatisch erkennt und die korrekte Verwendung sicherstellen kann. Auch der Rückgabetyp der `greetAll`-Funktion (ein Array von Strings) wird korrekt erkannt.

Funktionen in Objekten

Da Objekte in JavaScript auch Funktionen aufnehmen können, kannst du auch Funktionen mitsamt ihren erwarteten Parametern angeben. Die Methodenparameter gibst du dabei in Klammern an, den Rückgabewert mit einem Pfeil dahinter:

Beispiel: Typ-Angaben für Funktionen

```
type Person = {
  name: string;
  updateName: (name: string) => string;
}

const klaus: Person = {
  name: "Klaus",
  updateName: (name: string) => { console.log(name) } // OK
};

const wrong: Person = {
  name: "Klaus",
  updateName: (name: boolean) => { ... }
      // ERROR: Types of parameters 'name' and 'name'
      // are incompatible.
};
```

Typen erweitern

Typen können erweitert werden. Dazu kannst du mehrere Typen mit dem &-Operator verknüpfen (»Intersection Types«):

```
type Employee = Person & { salary: number; }

const klaus:Person = {
  firstName: "Klaus", age: 37, salary: 60000
};
```

Du kannst übrigens überall dort, wo du benannte Typen hinschreibst, auch direkt die Typdefinition angeben, wenn das für dich einfacher ist:

```
function greetAll(persons: { firstName: string, age: number }[]):
string[] { ... }
```

Und natürlich kannst du auch Destructuring bei Parametern durchführen. In dem Fall musst du den Typ hinter die geschweifte Klammer schreiben:

```
function greet({firstName}: Person) { ... }
```

Type Aliase

Außerdem kannst du mit type Aliase für bestimmte Typen vergeben, um zum Beispiel wiederkehrende Konstrukte zu vereinfachen oder Dinge fachlich zu beschreiben:

```
type Greetable = string | null;
function sayHello(name: Greetable) { ... }

type PrimaryKey = string;
function loadVote(id: PrimaryKey) { ... }
```

Typen exportieren

Genau wie Klassen, Funktion etc. in JavaScript kannst du in TypeScript auch Typen aus einem Modul exportieren, um den Typ in anderen Modulen verwenden zu können. Somit könntest du zum Beispiel ein Modul mit fachlichen Typen anlegen, das du dann innerhalb deiner Anwendung verwenden kannst:

```
// domain.ts
export type Person = { ... }
export type Employee = { ... }

// greeter.js
import { Person } from "./domain";

export function sayHello(person: Person) { ... }
```

D.3.2 Klassen in TypeScript

Neben den gezeigten Typen lassen sich mit TypeScript auch Klassen verwenden. Hierbei ist die Syntax zunächst identisch mit der Syntax der ES6-Klassen. Neu ist allerdings, dass du sämtliche Felder der Klasse sowie die Parameter der Methoden zwingend beschreiben musst. Dabei kannst du auch angeben, ob eine Variable oder eine Methode *protected* oder *private* sein soll. Protected Member sind nur in der Klasse selbst und in allen Unterklassen sichtbar, private Member nur in der Klasse selbst. Wenn du keine Sichtbarkeit hinschreibst, ist der Member überall sichtbar (public), so wie in JavaScript auch:

```
class Greeter {
  // private ist nur innerhalb Greeter sichtbar
  private phrase: string;

  constructor(phrase: string) {
    this.phrase = phrase;

    this.name = "Klaus"; // Property 'name' does not
                         // exist on type 'Greeter'
  }

  // greet ist public und von überall aus aufrufbar
  greet(name: string) {
    const greeting = `${phrase}, ${name}`; // ERROR: Cannot find
         // name 'phrase'. Did you mean the instance member
         //'this.phrase'

    return `${this.phrase}, ${name}`; // OK
  }
}

const wrong = new Greeter(); // ERROR: Expected 1 arguments,
                             // but got 0.
```

```
const g = new Greeter("Hello");
console.log(g.greet("Susi")); // Hello, Susi

const greetingPhrase = g.phrase; // ERROR Property 'phrase' is
                                 // private

g.phrase = "Moin"; // ERROR Property 'phrase' is private
```

D.3.3 Der any-Type

Bei der Arbeit mit TypeScript kann es dir passieren, dass du an Stellen gelangst, an denen du keine Typdefinition hinschreiben kannst oder willst, insbesondere wenn du mit Legacy-Code arbeitest, der (noch) in JavaScript geschrieben ist. In solchen Fällen kannst du den Typ `any` verwenden. Dieser schaltet die Typüberprüfung praktisch aus, sodass du an eine Variable bzw. einen Parameter, der vom Typ `any` ist, alle Typen zuweisen kannst und auch mit diesem Typen dann alles machen kannst. Das sollte wirklich nur der letzte Ausweg sein, da du damit die Vorteile von TypeScript verwirfst.

Beispiel: Verwendung von any

```
function sayHello(name: any) {
  name++; // OK
  name = true; // OK
  return name.toUpperCase(); // OK
}

sayHello(null); // OK
sayHello(„Klaus"); // OK
sayHello(123); // OK
```

Für den gezeigten Code wird TypeScript keine Compiler-Fehler ausgeben, trotzdem wird der Code natürlich zur Laufzeit mutmaßlich nicht funktionieren ...

Implizites Zuweisen von any

Wenn du einen Typ nicht explizit angibst und TypeScript es nicht schafft, den Typ herzuleiten (z.B. bei einem Parameter einer Funktion), weist TypeScript diesem Typ automatisch (implizit) den Typ `any` zu. Das implizite Zuweisen von `any` kann aber per Konfiguration verboten werden (damit das nicht »aus Versehen« passiert und versehentlich die Typprüfung abgeschaltet wird). Im Workspace unserer Beispielanwendung ist die implizite Zuweisung verboten und auch in Projekten, die neu mit Create React App erzeugt werden, ist diese Konfiguration aktiv:

```
function sayHello(name) { // ERROR: Parameter 'name' implicitly
                          // has an 'any' type.
 ...
}

function sayHello(name: any) { // OK
  ...
}
```

D.4 Externe Typbeschreibungen

Im vorherigen Kapitel haben wir dir gezeigt, wie du Typdefinitionen für deinen bestehenden JavaScript-Code schreiben kannst, damit der TypeScript-Compiler ihn auf korrekte Verwendung hin überprüfen kann.

Für bestehenden JavaScript-Code, den du nicht selbst geschrieben hast, also für von dir verwendete externe Module, wie beispielsweise React, wäre es sehr lästig, wenn du dafür ebenfalls die Typdefinitionen schreiben müsstest. Glücklicherweise enthalten mittlerweile einige in JavaScript geschriebene Module bereits auch TypeScript-Typdefinitionen (z.B. Redux). In diesem Fall musst du nichts weiter tun. TypeScript findet die Typdefinitionen automatisch im entsprechenden Verzeichnis unterhalb von `node_modules`. Alternativ stehen für die allermeisten JavaScript-Module externe Typdefinitionen zur Verfügung. Diese werden meist von der Community erstellt und gepflegt, häufig auch gemeinsam mit den Entwicklern der entsprechenden Bibliothek. Diese Typdefinitionen werden auf der Plattform DefinitelyTyped[3] zentral gesammelt. Du kannst sie wie ein gewöhnliches `npm`-Modul mittels `npm install` zu deinem Projekt hinzufügen. Üblicherweise heißen die Typmodule genauso wie auch das Modul, das sie beschreiben, nur mit dem `npm`-Scope `@types`. So kannst du beispielsweise die Typdefinitionen für React und React-DOM hinzufügen:

```
npm install --save @types/react @types/react-dom
```

Wenn du Create React App verwendest, sind diese Typdefinitionen für dich bereits hinzugefügt. Analog kannst du aber auch beispielsweise die Typen für den React Router oder React Redux hinzufügen.

Während dieses Vorgehen in der Regel sehr praktisch und bequem ist, kann es zu Problemen hinsichtlich der Versionierung kommen. So müssen die Version von TypeScript, die Version des Moduls und die Version des Typmoduls zusammenpassen. Natürlich muss eine Änderung an einem der drei Artefakte nicht notwendigerweise dazu führen, dass die anderen Artefakte nicht mehr dazu passen, aber das kann passieren. Aus diesem Grund ist es manchmal ratsam, insbesondere beim Erscheinen einer neuen TypeScript- oder React-Version, zunächst ein paar Tage abzuwarten, bis die Typdefinitionen angepasst wurden, und dann erst umzustellen. Das wird in der Regel relativ zügig passieren und dann solltest du auch nach Möglichkeit die Versionen in deinem Projekt aktualisieren, um zu vermeiden, dass du später große Versionssprünge mit vielen Anpassungen (und entsprechend komplexer Fehlersuche) vornehmen musst.

---

3 *http://definitelytyped.org/*

Eigene Typdeklarationen

Wenn es keine fertigen Typdeklarationen für eine Bibliothek gibt, kannst du die Typdeklaration auch selbst schreiben. Dazu musst du eine sogenannte Typdeklarationsdatei anlegen (diese endet mit `.d.ts`), in der du ein Modul deklarierst. Wie das funktioniert, ist im Detail in der TypeScript-Dokumentation beschrieben.