Die Sprache TypeScript

Niklas Mollenhauer

12. November 2018

1 Einleitung

Im Jahr 1995 gestalteten sich Webseiten wenig interaktiv. Mit einem Browser-Marktanteil von 80% wollte die Firma Netscape dies ändern und eine Skriptsprache in ihren Browser integrieren. Die Skriptsprache sollte vom Browser des Nutzers ausgeführt werden und Zugriff auf die Elemente der Webseite haben und diese sogar verändern können. Es sollte bspw. möglich werden, HTML-Formulareingaben auf syntaktische Korrektheit zu prüfen, bevor sie vom Nutzer abgesendet werden. Auch Effekte wie das Ändern von Elementeigenschaften, wenn die Maus über das Element gleitet, sollten möglich sein.

Diese Skriptsprache nannte sich JavaScript (ursprünglich „LiveScript“). Sie wurde von Brendan Eich entwickelt und in den Netscape Navigator integriert. Den Prototypen entwickelte er innerhalb von zehn Tagen. Ihr Name war angelehnt an die damals bereits existierende Sprache Java. Mit Java hat JavaScript bis auf den Namen und einige Elemente der Syntax nichts zu tun.

Aufgrund der Popularität von JavaScript gehörte es mit der Zeit zu der Standardausstattung jedes Webbrowsers. Ein weiterer Grund bestand darin, dass sich keine weitere Alternative durchsetzen konnte. Da bestehende Webseiten bereits JavaScript einsetzten, war es naheliegend, die Sprache nachträglich mit Features zu erweitern, statt eine neue Sprache zu entwickeln.

Im Jahr 2018 ist JavaScript mehr verbreitet denn je und aus dem WWW nicht mehr wegzudenken. Webseiten bestehen nicht nur aus einigen interaktiven Elementen. Stattdessen werden ganze Anwendungen in Form einer Webseite entwickelt („Single Page Applications“). Es existieren zahlreiche Interpreter bzw. Engines der Skriptsprache, deren Entwicklung meist von Browserherstellern vorangetrieben wird. Am bekanntesten ist Googles „V8“, Mozillas „SpiderMonkey“ und Microsofts „Chakra“. Um die Kompatibilität zu wahren, existiert ein JavaScript-Standard namens ECMAScript, welcher von den JavaScript-Engines implementiert wird. Dieser wird mittlerweile jährlich aktualisiert, weshalb sich die Sprache schnell weiterentwickelt.

Verwendet wird JavaScript auch außerhalb des Browsers: Das Projekt „Node.js“ leiht sich die JavaScript-Engine „V8“ und verwendet sie auf der Serverseite und für Konsolenanwendungen. Doch nicht nur im Browser und auf der Serverseite befindet sich JavaScript. Auch alleinstehende Smartphone- und Desktop-Anwendungen können in JavaScript entwickelt werden. JavaScript kann sogar auf Microcontrollern verwendet werden. Die ursprünglich in kürzester Zeit für simple Nutzerinteraktionen entworfene Sprache wird somit heutzutage in weiten Teilen der IT-Industrie verwendet.

Skriptsprachen werden meist für die Entwicklung von kleineren Programmen entworfen und verwendet. Dies war ursprünglich auch bei JavaScript der Fall. Heutzutage bestehen JavaScript-Projekte nicht selten aus mehreren Dateien und werden von ganzen Teams entwickelt. Unter dieser Entwicklung litt die Wartbarkeit dieser Projekte.

Um die Wartbarkeit zu erhöhen wurde im Jahr 2012 eine Lösung von Microsoft entwickelt. Diese Lösung nennt sich TypeScript und hat als Ziel, JavaScript zu erweitern und gleichzeitig die Kompatibilität zu JavaScript zu wahren. Dabei sollen einige Fehlerklassen schon erkannt werden, bevor der Code ausgeführt wird, wie es bei Sprachen wie z. B. Java der Fall ist.

Da TypeScript auf JavaScript aufbaut, wird im ersten Abschnitt ein kleiner Einblick in die Sprache JavaScript gegeben. Dieser besteht aus einem Überblick über JavaScript, dem Laufzeit-Typsystem, Prototypen und den Programmierparadigmen. Für einen tieferen Einstieg in die JavaScript bietet sich das Buch „Eloquent JavaScript“ an [1]. Es wird anhand eines Beispiels gezeigt, welche Probleme sich bei der JavaScript-Entwicklung in großen Projekten ergeben könnten.

Im darauffolgenden Abschnitt wird TypeScript als Lösung dieser Problemklassen vorgestellt. Es wird gezeigt, wie TypeScript das Typsystem von JavaScript zur Entwicklungszeit abbildet. Außerdem wird darauf eingegangen, wie genau die Kompatibilität zu bestehenden JavaScript-Bibliotheken hergestellt wird. Das

offizielle TypeScript-Handbuch bietet über diese Ausarbeitung hinaus einen ausführlicheren Einstieg in TypeScript [2]. Einen tieferen Einstieg bietet das Buch „TypeScript Deep Dive“, welches zusätzlich einen Einblick in die Interna des TypeScript-Compilers gibt [3].

2 JavaScript

JavaScript ist *dynamisch* und *schwach* typisiert. Dynamisch bedeutet hier, dass Typfehler erst zur Laufzeit einen Fehler produzieren. Die schwache Typisierung führt dazu, dass Werte implizit in Werte anderen Typs umgewandelt werden können. Dies kann bspw. bei Operationen wie der Addition oder bei Vergleichen der Fall sein. Es existieren primitive und nicht-primitive Datentypen. Zu den primitiven zählen unter anderem `string`, `number`, `boolean`. In JavaScript steht der `number`-Datentyp immer für eine Fließkommazahl. Integer existieren in JavaScript nach aktuellem Stand nicht.

Im Mittelpunkt von JavaScript befinden sich Funktionen und prototypenbasierte Programmierung. Grundsätzlich ist JavaScript geeignet für mehrere Programmierparadigmen. Dazu gehört vor allem funktionale und objektorientierte Programmierung.

Für das Nichtvorhandensein von Werten hat JavaScript – anders als in anderen Sprachen – zwei Varianten: `null` und `undefined`. Sie unterscheiden sich darin, dass `undefined` der Wert einer Variable ist, die zwar deklariert, ihr aber nichts zugewiesen wurde. `null` hingegen ist ein „definiertes Nichts“, was einer Variable zugewiesen werden kann, um zu verdeutlichen, dass etwas zugewiesen wurde, aber kein Wert existiert. Es ist trotzdem möglich, einer Variable explizit `undefined` zuzuweisen.

Eine Funktion kann immer mit einer beliebigen Anzahl an Argumenten aufgerufen werden. Wird eine Funktion mit weniger Parametern aufgerufen, als sie in ihrer Methodensignatur auflistet, erhalten die unbesetzten Parameter bei der Ausführung der Funktion den Wert `undefined`.

In JavaScript existieren zwei unterschiedliche Arten von Sichtbarkeiten: *Block-Scoping* und *Function-Scoping*. Beim Block-Scoping sind Variablen immer nur in dem Programmblock sichtbar, in dem sie deklariert sind. Wird eine Variable bspw. innerhalb des Körpers eines `if`-Zweigs deklariert, lässt sie sich nur innerhalb dieses Blocks verwenden. Dies ist auch bei den Sprachen Java, C-Sharp oder C der Fall. Beim Function-Scoping ist eine Variable immer in der gesamten Funktion sichtbar. Dies ist selbst der Fall, wenn sie innerhalb eines `if`-Zweigs deklariert wurde. Variablen mit einer funktionsweiten Sichtbarkeit werden mit `var` deklariert, blockbasierte Sichtbarkeit wird mit `let` erreicht. Es ist zudem möglich, Konstanten mit blockbasierter Sichtbarkeit zu deklarieren.

Prototypen

Prototypenbasierte Programmierung bedeutet, dass Objektinstanzen durch das Klonen von Prototypen erstellt werden. Jede Funktion in JavaScript besitzt die Eigenschaft `prototype`. Diese Eigenschaft wird als Vorlage für neue Objekte verwendet, sollte die Funktion, zu der diese Eigenschaft gehört, als Konstruktor verwendet werden. Listing 1 zeigt die Funktion `Vector`, welche in dem Beispiel als Konstruktor fungiert. Ein neues Objekt mit dem `Vector`-Prototyp kann nun mittels `new Vector(13, 37)` erstellt werden. Das resultierende Objekt `first` hat nun die Eigenschaften `x` und `y`, welche durch den Konstruktor gesetzt werden.

Ein Prototyp kann auch nach der Instanziierung eines Objekts verändert werden. Beispielsweise können dem Prototyp so zu jeder Zeit Funktionen hinzugefügt werden. Bereits aus diesem Prototyp erstellte Objekte erhalten diese Änderungen nachträglich, da sie auf ihren Prototypen verweisen. Zeile 8 zeigt wie dem Vector-Prototypen die Funktion `abs` hinzugefügt wird. So erhalten das bereits erstellte Objekt `first` sowie das später erstellte Objekt `second` die Funktion `abs`.

Über dieses Konzept kann in JavaScript objektorientiert entwickelt werden. Im Unterschied zu klassenbasierter Objektorientierung muss für ein Objekt bei der Instanziierung nicht festgelegt sein, welche Eigenschaften das Objekt haben wird. Sie können auch nachträglich dem Prototypen oder dem Objekt selbst hinzugefügt werden. Wird eine Funktion direkt einem Objekt hinzugefügt, wie es mit der Funktion `mult` aus Listing 1 der Fall ist, ist diese Funktion nur bei diesem Objekt verfügbar. Der Prototyp und somit auch die restlichen Instanzen des Prototyps bleiben davon unberührt. Prototypen können in JavaScript auch über ein `class`-Schlüsselwort definiert werden. Dabei bietet das `class`-Schlüsselwort lediglich eine syntaktisch kompaktere Schreibweise für Prototypen, statt das Konzept der Klassen in JavaScript einzuführen.

```
function Vector(x, y) { // Konstruktor
    this.x = x; // Eigenschaften werden durch "this.foo = " definiert und zugewiesen
    this.y = y;
}
let first = new Vector(13, 37); // Neuen Vector erstellen

// Instanzfunktion für alle Vector-Prototyp-Instanzen hinzufügen
Vector.prototype.abs = function() {
    return Math.sqrt(this.x ** 2 + this.y ** 2);
}
let second = new Vector(23, 42);
// mult-Funktion wird **nur** dem Objekt "first" hinzugefügt
first.mult = function (f) { this.x *= f; this.y *= f; }
first.mult(2);
console.log("first.abs(): " + first.abs()); // Ausgabe: first.abs(): 78.434...

let third = { // Instanziierung mit Objekt-Notation
    x: 42,
    "y": 23, // Mit Anführungsstrichen
    abs() { return Math.sqrt(this.x ** 2 + this.y ** 2); }
};
console.log("third.abs(): " + third.abs()); // Ausgabe: third.abs(): 47.885...
```

Listing 1: Prototypen in JavaScript

Objekte können nicht nur mittels `new`-Schlüsselwort instanziiert werden. Eine weitere Möglichkeit ist JavaScript-Object-Notation (JSON) mit und ohne Anführungsstriche für die Eigenschaften bzw. Attribute, zu sehen bei der Variable `third` aus Listing 1. Das `third` -Objekt kopiert seine Eigenschaften dabei nicht vom `Vector`-Prototypen, sondern vom `Object`-Prototypen. Diese dynamische Handhabung von Eigenschaften führt dazu, dass Objekte auch als assoziatives Array bzw. Map-Typ angesehen werden kann.

Aus diesem Grund sind Arrays in JavaScript keine speziellen Typen. Sie sind lediglich Objekte, die Zahlen als Eigenschaften haben. Um die Verwendung als Array zu ermöglichen, existieren zwei Möglichkeiten, um auf die Eigenschaften eines Objekts zuzugreifen. Beispielsweise wäre der Zugriff auf die `name`-Eigenschaft des Objekts `person` über `person.name` als auch über `person["name"]` möglich (vgl. Listing 2). Die letzte Variante bietet die Möglichkeit, die zu lesende Eigenschaft erst zur Laufzeit zu bestimmen, indem statt einem statischen Stringliteral eine Variable verwendet wird. Die Bezeichner von Objekteigenschaften können vom Typ `string` oder `number` sein. Letzteres wird bei Arrays verwendet, ist aber nicht darauf beschränkt.

Aufgrund der schwachen Typisierung existieren in JavaScript zwei Operatoren für Gleichheit: `==` und `===`. Die erste Variante führt implizite Typkonvertierung durch, die zweite prüft auf Wert- und Typgleichheit. Bspw. liefert `"" == false` den Wert `true` zurück, was andeutet, die beiden Operanden seien gleich. Vergleicht man die beiden Werte mit `===`, ist das Ergebnis `false`. Es wird generell empfohlen, die letzte Variante zu verwenden. Dies hat den Hintergrund, dass die Regeln der Werteumwandlung der Operanden bei einem Vergleich mit `==` nicht trivial sind und deshalb für häufige Fehler sorgen.

JavaScript-Code läuft üblicherweise in einem einzigen Thread. Aus diesem Grund läuft die JavaScript-Ausführung im Kontext eines Browsers im gleichen Thread wie das Zeichnen der Webseite. Diese Designentscheidung führt dazu, dass es in JavaScript nicht vorgesehen ist, diesen Thread blockieren zu können[1]. Ein Blockieren dieses Threads würde bedeuten, dass die Webseite nicht mehr auf Nutzereingaben reagiert oder sich nicht neu zeichnet. In JavaScript haben sich aus diesem Grund Event- bzw. Callback-basierte Entwurfsmuster etabliert. Der Speicher von JavaScript wird durch einen Garbage Collector verwaltet. Der Entwickler muss sich also nicht um das Anlegen und Freigeben von Speicher kümmern.

[1] Es existieren Außnahmen wie bspw. die `alert()`-Funktion

```
1  let book = { name: "Der Hobbit", pages: 42 };
2  let person = { name: "Eich", firstName: "Brendan" };
3
4  function getName(item) {
5      return item.name;
6  }
7
8  console.log(book.name.length); // Zugriff über Eigenschaft-Syntax
9  console.log(person["name"].length); // Zugriff über Array-Syntax
10 console.log(getName(book).length);
11 console.log(getName(person).length);
```

Listing 2: Das Umbenennen eines Identifiers ist nicht trivial

Probleme mit JavaScript

Da JavaScript dynamisch typisiert ist, fallen typische Fehler wie Tipp-, Typ- oder Verweisfehler erst zur Laufzeit auf, wenn der Code ausgeführt wird. Für das einfache Umbenennen von Variablen bleibt deshalb oft nichts anderes übrig, als „Suchen-und-Ersetzen“ zu verwenden. Das Beispiel aus Listing 2 zeigt eine Funktion `getName`, welche die `name`-Eigenschaft des übergebenen Parameters zurückgibt. Aufgerufen wird die Funktion mit der Variable `person` bzw. `book` als Parameter. Hier fällt auf: In einer statisch typisierten Sprache hätten die Variablen vermutlich verschiedene Typen, bspw. `Book` und `Person`. Dies ist hier nicht der Fall, zwar sind beide vom Prototyp `Object`, trotzdem haben sie verschiedene Eigenschaften. Die Funktion `getName` kann mit beiden Variablen als Parametern aufgerufen werden, ohne einen Fehler zu produzieren. Solche Muster kommen in der JavaScript-Praxis durchaus vor. Wie Listing 2 zeigt, gestaltet sich schon das einfache Umbenennen einer Variable oder Eigenschaft als kein einfaches Problem.

Wird die die Eigenschaft `name` der Variable `person` einfach in `lastName` geändert, gibt `getName` in Zeile 11 `undefined` zurück, da die `name`-Eigenschaft nicht mehr existiert. Zur Laufzeit entsteht dort ein Referenzfehler, da die Eigenschaft `length` auf dem Wert `undefined` nicht existiert. Es muss also auch die Funktion `getName` angepasst werden. Wird hier aber einfach `item.lastName` zurückgegeben, entsteht in Zeile 10 ein analoger Fehler, da die `book`-Variable keine `lastName`-Eigenschaft besitzt. Die Lösung besteht darin, in der `getName`-Funktion eine zusätzliche Prüfung einzuführen, die prüft, ob eine `lastName`-Eigenschaft existiert und diese bevorzugt zurückgibt.

Eine weitere Schwierigkeit beim Umbenennen besteht darin, dass auf Eigenschaften auch über die Array-Syntax zugegriffen werden kann. Der Name der Eigenschaft, auf die zugegriffen werden soll, kann somit auch zur Laufzeit ausgemacht werden. In solchen Fällen gestaltet es sich besonders schwierig, eine Eigenschaft umzubenennen.

Wird einer Funktion ein Parameter hinzugefügt, kann es passieren, dass bei einem Refactoring der neue zusätzliche Parameter an einigen Stellen schlicht vergessen wird. Fehler dieser Klassen fallen – wenn überhaupt – erst zur Laufzeit auf. Solche Refactorings kommen bei der Entwicklung jedoch häufig vor. Das kleine Beispiel aus Listing 2 zeigt somit schon einige Probleme auf, die Sprachen wie z. B. Java mit Hilfe statischer Typisierung lösen.

JavaScript-Projekte wurden mit der Zeit immer größer, was letztendlich zu Wartbarkeitsproblemen führte. Da JavaScript im Browser bisher noch alternativlos ist, ist es nicht möglich, ohne Weiteres eine andere Sprache zu verwenden. Microsoft stieß intern selbst auf solche Skalierungsprobleme, weshalb der Mitarbeiter Anders Hejlsberg – der Entwickler von Delphi, Turbo Pascal und C-Sharp – im Jahr 2012 eine Lösung entwickelte.

3 TypeScript

TypeScript hat das Ziel, das Typsystem, das JavaScript zur Laufzeit hat, dem Entwickler schon einen Schritt vorher – in der Entwicklungszeit – bereitzustellen. So sollen typische Fehlerklassen wie Tippfehler, falsche Parameterzahl, Null-Verweise oder unerreichbarer Code schon vor dem Ausführen erkannt werden. Dabei ist

TypeScript keine komplett eigene Skriptsprache, sondern eine *Obermenge* von JavaScript. Jeder syntaktisch gültige JavaScript-Code ist somit gültiger TypeScript-Code. Dies ist vor allem bei der Migration von JavaScript zu TypeScript ein entscheidender Vorteil. TypeScript erweitert JavaScript um optionale, statische Typisierung und wird üblicherweise nicht direkt von einem Interpreter ausgeführt. Stattdessen gibt es einen Compiler, dessen Ausgabe normales JavaScript ist, welches überall ausgeführt werden kann, wo JavaScript ausgeführt werden kann. Der TypeScript-Compiler ist *kein* optimierender Compiler. Er wird üblicherweise nur vom Entwickler verwendet und nicht an bspw. den Browser ausgeliefert, welcher nur das resultierende JavaScript erhält. Entfernt man aus einem TypeScript-Code alles, was mit statischer Typisierung zu tun hat, erhält man normales JavaScript[2]. Die statische Typisierung bietet Features wie Interfaces, Typ-Annotationen an Variablen, Funktionsparametern und -Rückgabewerten, generische Programmierung und Typ-Aliase. TypeScript eignet sich für die selben Einsatzgebiete wie übliches JavaScript.

Da sich JavaScript rasant entwickelt und an neuen Features gewinnt, ergibt sich eine weitere Problematik: Nicht jeder Benutzer verwendet den neusten Browser, die immer die neusten JavaScript-Features unterstützen. Ein weiteres Ziel von TypeScript ist deshalb, neuere Sprachfeatures verwendbar zu machen, ohne, dass der Endnutzer die neuste JavaScript-Engine hat. Umgesetzt wird dies dadurch, dass der TypeScript-Compiler eingestellt werden kann, zu welchem JavaScript-Standard kompatibler Code ausgeben werden soll.

Da die statische Typisierung optional ist, bezeichnet man TypeScript auch als „graduell getypt“. Wird kein Typ angegeben, wird er – wenn möglich – über Typinferenz hergeleitet. Ist dies nicht möglich, wird der TypeScript-spezifische Typ `any` verwendet, welcher für alle primitiven und nicht-primitiven Werte steht.

Da sich TypeScript darauf fokussiert, lediglich statisch typisiertes JavaScript zu sein, entwickelt man mit TypeScript meist in den selben Programmierparadigmen, die bei JavaScript üblich sind. TypeScript definiert zusätzlich zu den JavaScript-Typen eigene Typen, die es in JavaScript in der Form nicht gibt, da sie zur Laufzeit auch nicht mehr relevant sind und nur vom TypeScript-Typechecker verwendet werden. Beispielsweise gibt es den bereits genannten Typ `any`. Andere, TypeScript-exklusive Typen, sind bspw. `never` und `this`. Diese haben den Zweck, die Semantik deutlicher ausdrücken zu können. Hat der Compiler geschlussfolgert, dass eine Funktion den Rückgabetyp `never` hat, so wird die Funktion zur Laufzeit nicht terminieren. Dies ist bspw. der Fall, wenn in der Funktion eine triviale Endlosschleife gefunden wurde. Gibt eine Funktion die `this`-Referenz zurück, besitzt die Funktion den Rückgabetyp `this`. So erhält der Compiler die Semantik, dass eine zurückgegebene Referenz nicht nur vom selben Typ, sondern sogar das selbe Objekt ist.

3.1 Union Types und Kontrollfluss

In JavaScript ist eine Variable nicht fest an ihren Typen gebunden. Es ist problemlos möglich, nach einer Zuweisung wie `pages = 42` eine weitere Zuweisung mit einem anderen Typ durchzuführen, bspw. `pages = "a lot"`. Auch eine Funktion ist nicht daran gebunden, Werte eines bestimmten Typs zurückzugeben, wie die in Listing 3 deklarierte Funktion zeigt. Um dieses Verhalten in TypeScript zu modellieren, wurden „Union Types“ eingeführt. Union Types bieten die Möglichkeit, Typen zu „verodern“. So kann der Typ `number | string` verwendet werden, wenn die entsprechende Variable oder der Rückgabewert sowohl eine `number` als auch ein `string` sein kann. Bevor die Variable wie eine `number` oder wie ein `string` verwendet wird, muss explizit geprüft werden, welchen konkreten Typ die Variable hat. Dies ist mit dem `typeof`-Operator aus JavaScript möglich. Das Beispiel aus Listing 3 zeigt ebenfalls, dass der Typ einer Variable immer vom aktuellen Kontext bzw. dem Kontrollfluss des Programms abhängig ist. Abfragen in der Form von `if` oder einem `switch`-Block haben somit die Möglichkeit, den Typen einer Variable innerhalb des Abfragenkörpers zu konkretisieren.

Die Notwendigkeit für eine Prüfung kann auch fallspezifisch mit einer sogenannten „Type Assertion“ abgeschaltet werden. Type Assertions sind das Pendant zu Type-Casts, wie es sie bspw. in Java gibt. Der Unterschied ist allerdings, dass eine Type Assertion nur einen Einfluss auf den Typechecker hat. Der erzeugte JavaScript-Code wird davon nicht verändert. Der Entwickler kann so einen Compilerfehler unterdrücken, ist aber in der eigenen Verantwortung, dass seine Annahme erfüllt wird. Eine Type Assertion ist in Listing 3 bei der Variable `baz` in Zeile 16 zu sehen. Hier wird angenommen, dass die Variable `bar` ein `string` ist.

[2] Ggf. wird Zusatzcode eingefügt, um fehlende JavaScript-Sprachfeatures bereitzustellen, sollte der Ziel-JavaScript-Standard diesen nicht unterstützen.

Der Compiler meldet deshalb keinen Fehler, allerdings kann das Programm zur Laufzeit an der Stelle einen Typfehler melden, falls `bar` zur Laufzeit eine `number` ist.

```ts
function foo(i: number) { // Rückgabetyp geschlussfolgert durch returns: number | string
    if (i % 2 === 0) { return 1; }
    else { return "odd"; }
}

let bar = foo(2); // bar hat Typ: number | string
if (typeof bar === "number") {
    // bar hat Typ: number
    console.log(bar * 2); // Multiplikation zulässig, da number
    console.log("Länge: " + bar.length); // Compilerfehler, bar.length ist unmöglich
} else {
    // bar hat Typ: string (number wurde explizit ausgeschlossen)
    console.log("Länge: " + bar.length); // .length zulässig, da string
}

let baz = bar as string; // Type Assertion zu string
console.log(baz.length); // baz ist string, kann aber zur Laufzeit Fehler verursachen
```

Listing 3: Kontrollfluss-abhängige Typen

Es können alle Typen (also primitive und nicht-primitive) „verodert“ werden. Union Types lassen sich nicht nur für Variablen verwenden, sondern an allen Stellen, bei denen ein Typ spezifiziert werden kann. Sie können zudem auch das Ergebnis einer Typinferenz des Compilers sein, wie bspw. der Rückgabetyp aus der Funktion in Listing 3.

3.2 Nullable und Non-Nullable Types

Wie in der Einführung bereits angeführt, existieren in JavaScript zwei verschiedene Werte für Nichtvorhandensein: `null` und `undefined`. Dies führt dazu, dass Null-Checks in JavaScript oft nur die Hälfte der Fälle abdecken, da den Entwicklern nicht immer klar ist, ob auf `null` oder `undefined` – oder sogar beides – geprüft werden muss.

Hier bietet TypeScript Abhilfe: Der Compiler kann so eingestellt werden[3], dass Variablen jeden Typs nicht die Werte `null` oder `undefined` annehmen können. Abbildung 1 zeigt die Werte, die eine Variable vom entsprechenden Typ annehmen kann, wie es ohne diese Unterstützung ist. Abbildung 2 hingegen zeigt, dass `null` und `undefined` nun nicht mehr in der jeweiligen Menge an Werten enthalten sind.

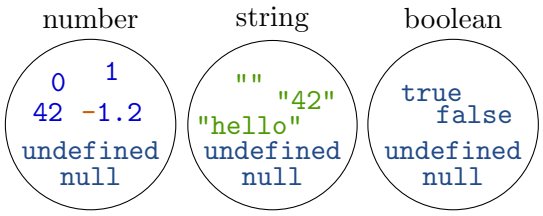

Abbildung 1: `strictNullChecks` ausgeschaltet

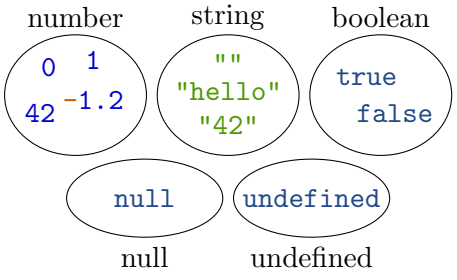

Abbildung 2: `strictNullChecks` eingeschaltet

Wird mit eingeschalteter strikter Null-Prüfung versucht, `let name: string = undefined;` zu kompilieren, meldet der TypeScript-Compiler, dass die Zuweisung nicht zulässig ist, da der Typ `string` nicht den Wert `undefined` annehmen kann. Die Werte `null` und `undefined` haben nun jeweils ihren eigenen,

[3] Compiler-Flag: `strictNullChecks`

gleichnamigen Typ. Variablen vom Typ `undefined` können so nur noch ausschließlich den Wert `undefined` annehmen. Für Funktionen, die bspw. angeben, einen `string` zurückzugeben, garantiert nun der Compiler, dass diese niemals `null` oder `undefined` zurückgeben.

Es kommt vor, dass ein Entwickler in einer Funktion sowohl `null` als auch einen `string` zurückgeben will. Hier bietet das Union-Types-Feature abhilfe: Der Rückgabetyp kann als `string | null` angegeben werden, denn `null` ist mit dem Strict-Null-Checks-Feature ein normaler Typ. So forciert der Compiler – ähnlich wie in Listing 3 – dass vor der Verwendung der entsprechenden Variable eine entsprechende Überprüfung vorangehen muss. Analog funktioniert dies auch mit `undefined`.

Die Zuweisung müsste also in `let name: string | undefined = undefined;` geändert werden. Greift der Entwickler nun auf die `.length`-Eigenschaft der Variable `name` zu, darf kein Code-Pfad existieren, der zu dieser Stelle führt, bei der keine Prüfung auf `undefined` stattgefunden hat. Auf dieselbe Art und Weise garantiert der Compiler, dass einer Variable etwas zugewiesen wurde, bevor sie verwendet wird.

3.3 Interfaces und strukturelle Typen

In TypeScript können Objektstrukturen über Interfaces beschrieben werden. Interfaces können hierbei nicht nur Funktionen beinhalten, sondern auch Eigenschaften bzw. Attribute. Der TypeScript-Compiler achtet dabei nur auf die Struktur der Objekte, weshalb Interfaces *nicht explizit* von Prototypen oder Funktionen implementiert werden müssen. Dies ist anders als z. B. in der Sprache Java, bei der ein Objekt nur kompatibel mit einer Variable vom Typ eines Interfaces ist, wenn die dazugehörige Klasse das Interface explizit implementiert. Diese Eigenschaft von TypeScript wird auch „strukturelles Typsystem“ genannt.

Durch diese Gegebenheit ist es möglich, Interface-Deklarationen zu erweitern, wie Listing 4 zeigt. Bei diesem Konzept der „offenen Interfaces“ kann dies geschehen, indem ein neues Interface mit denselben Namen angelegt wird. Der TypeScript-Compiler fügt die Interface-Deklarationen zu einer zusammen. Dieses Vorgehen soll wiederspiegeln, dass es in JavaScript möglich ist, Prototypen nachträglich zu verändern. In Listing 4 wird das Interface `Book`, welches ursprünglich nur die Eigenschaft `title` definiert, um die Eigenschaft `pages` erweitert. Alle Stellen, die nun das `Book`-Interface referenzieren, verweisen eigentlich auf eine zusammengeführte Version beider Interfaces.

```
1  interface Book { title: string; }
2  // ...
3  interface Book { pages: number; } // Erweiterung des "Book"-Interfaces
4
5  let bar: Book = { pages: 13 }; // Fehler, "title" fehlt
6
7  let foo: Book = {
8      title: "Per Anhalter durch die Galaxis",
9      pages: 42
10 }; // Okay, "foo" deklariert alle benötigten Eigenschaften
11
12 interface Comic { title: string; pages: number; }
13
14 let baz: Comic = foo; // Gültig, da Struktur von "Comic" und "Book" identisch
```

Listing 4: Offene Interfaces und strukturierte Typen

Listing 4 zeigt auch, dass alle Typen miteinander kompatibel sind, die die gleiche Struktur aufweisen. Dies wird durch die gültige Zuweisung von `foo` zu `baz` deutlich. `baz` wird hier explizit als `Comic` deklariert, zugewiesen wird der Variable allerdings ein `Book`. Da im resultierenden JavaScript-Code keine Typen mehr vorhanden sind, ergibt dies auch Sinn: Methoden, die die Eigenschaften eines `Comics` verwenden, können gar nicht den Unterschied zu einem `Book` bemerken. Das anfängiche JavaScript-Beispiel aus Listing 2 würde hier bspw. für den Parameter `item` als Typ `Book | Person` verwenden (der `Person`-Typ müsste dafür noch definiert werden). So würde der Compiler schon zur Entwicklungszeit einen Fehler melden.

TypeScript bietet noch weitere, fortgeschrittene Features für die Typisierung. Durch das strukturelle

Typsystem ermöglichen sich Szenarien, die mit nominellen Typen – wie sie bspw. in Java realisiert sind – nicht abbildbar sind. Insgesamt ist das Typsystem von TypeScript Turing-vollständig[4].

3.4 Type Declaration Files

Das JavaScript-Ökosystem ist über die Jahre gewachsen. Die JavaScript-Paketverwaltung Node Package Manager (NPM), welche Bibliotheken für sowohl server- als auch clientseitiges JavaScript verwaltet, beinhaltet zur Zeit über 800.000 Bibliotheken[5]. Ein Großteil dieser Bibliotheken ist in JavaScript implementiert. Hier ergibt sich ein Problem: TypeScript ist zwar eine Obermenge von JavaScript, allerdings hat der TypeScript-Compiler keine Typinformationen für diese Bibliotheken, da diese nicht in TypeScript implementiert sind. Um nicht die gesamte Menge an JavaScript-Bibliotheken neu in TypeScript implementieren zu müssen, wurden „Type Declaration Files“ eingeführt. Die Grundidee ist ähnlich zu Header-Dateien in C (`.h`). Für eine existierende JavaScript-Bibliothek kann eine TypeScript-Declaration-File geschrieben werden, die lediglich die öffentliche Schnittstelle der Bibliothek beschreibt. Eine solche Deklarationsdatei beinhaltet bspw. die in der Bibliothek bereitgestellten Funktionen, Typen, Interfaces, Klassen und Konstanten. Die konkrete Implementierung liegt weiterhin bei der bereits bestehenden JavaScript-Bibliothek.

Auf diese Weise können Entwickler etablierte, in JavaScript implementierte, Bibliotheken verwenden. Ein weiterer Vorteil ist, dass sich die Migration nach TypeScript einfacher gestaltet. Entwickelt und gewartet werden diese Deklarationen entweder von den Entwicklern der jeweiligen Bibliothek oder von der Community in einem zentralen Open-Source-Repository [4].

4 Zusammenfassung

JavaScript wird in der heutigen Zeit für Aufgaben verwendet, für die es anfangs nicht ausgelegt war. Dies zieht Skalierungsprobleme mit sich, was in großen Projekten zu schlecht wartbarer Software führt. TypeScript ist eine Obermenge von JavaScript, welche zusätzlich optionale, statische Typisierung beinhaltet. Das statische Typsystem von TypeScript orientiert sich an dem Typsystem, welches JavaScript zur Laufzeit hat. Dabei wird TypeScript üblicherweise nicht direkt interpretiert, sondern zu normalem JavaScript kompiliert, welches ausgeliefert wird. Durch die statische Typisierung werden Fehler erkannt, bevor das Programm ausgeführt wird. Konzepte wie das der Union Types oder der Non-Nullable-Types helfen dabei, Typ- oder Verweisfehler zur Laufzeit zu vermeiden. Der Typechecker des Compilers berücksichtigt bei der Überprüfung des Programms auch den Kontrollfluss, also Verzweigungen innerhalb des Programms. Mit Hilfe von Deklarationsdateien können bestehende JavaScript-Bibliotheken sowie das JavaScript-Ökosystem weiterverwendet werden.

TypeScript gewinnt aufgrund der immer größer werdenden Anforderungen an Webanwendungen immer mehr an Popularität[6]. Die Sprache bietet insgesamt viele Features, die das Entwickeln von JavaScript-basierten Anwendungen skalierbarer machen.

Literatur

[1] M. Haverbeke, “Eloquent JavaScript, 3rd edition.” `https://eloquentjavascript.net/`, 2018. [Online; aufgerufen am 3. Nov. 2018].

[2] Microsoft, “TypeScript Documentation.” `https://www.typescriptlang.org/docs/`, 2018. [Online; aufgerufen am 4. Nov. 2018].

[3] B. A. Syed, “TypeScript Deep Dive.” `https://basarat.gitbooks.io/typescript/`, 2018. [Online; aufgerufen am 5. Nov. 2018].

[4] DefinitelyTyped Community, “DefinitelyTyped.” `https://github.com/DefinitelyTyped/DefinitelyTyped/`, 2018. [Online; aufgerufen am 4. Nov. 2018].

[4] Der Beweis ist verfügbar auf https://github.com/Microsoft/TypeScript/issues/14833

[5] Aktuelle Zahlen verfügbar auf npmjs.com

[6] Vergleiche „2018 Node.js User Survey Report“, verfügbar auf https://nodejs.org/en/user-survey-report/