Willkommen bei TypeScript

Getyptes JavaScript für Java-Entwickler

Johannes Dienst

Große JavaScript-Codebasen lassen sich nur schwer beherrschen. Selbst mit modernen Entwicklungsumgebungen werden die Grenzen schnell erreicht. Das liegt schon in der Natur von JavaScript selbst, welche eine dynamisch typisierte Skriptsprache ist. TypeScript hält Objektorientierung durch Interfaces und Klassen sowie optionale statische Typisierung bereit, wodurch nützliche Werkzeugunterstützung möglich wird. Weitere sinnvolle Sprachfeatures machen TypeScript zur idealen Sprache für JavaScript-Anwendungen mit umfangreichen Codebasen. Als echte Obermenge von JavaScript ist der Umstieg sogar fast schmerzfrei.

Getyptes JavaScript mit Zusatzfeatures

Bei der Entwicklung einer JavaScript-Applikation kommt man ab einer bestimmten Größe der Codebasis früher oder später an den Punkt, an dem nur mit höchster Disziplin und großer Konzentration der Überblick behalten werden kann. Oft führt der Weg in die Unwartbarkeit des Codes und wehe, der Verantwortliche soll einen Kollegen mit dem Projekt vertraut machen.

An dieser Weggabelung stehen viele Projekt und die Beteiligten suchen nach einem geeigneten Ausweg. Einerseits ist JavaScript eine flexible Sprache, die in den richtigen Händen sinnvolle Ergebnisse hervorbringt. Andererseits sprechen Konstrukte wie Pseudo-OOP dafür, dass es der Sprache an manchen Stellen an wichtigen Konzepten mangelt. Auf der Suche nach einer Alternative gibt es nicht sehr viel Auswahl; Dart und CoffeeScript sind interessant, aber auch mit einer deutlichen Lernkurve behaftet. Außerdem wird dann die Entwicklungssprache für immer festgelegt, da das aus ihnen generierte JavaScript nicht dem entspricht, was ein normaler Entwickler erzeugen würde. TypeScript tritt an, sich dieses Problems anzunehmen.

Von der Syntax her an Java mit einer Prise Scala erinnernd, verspricht es echte Objektorientierung und statische Typisierung, falls gewünscht. Außerdem ist der Compiler zu reinem JavaScript Open Source. Die Sprache überrascht nach einer kurzen Eingewöhnung mit einer angenehmen Lernkurve und einfach zu installierender Werkzeugunterstützung. Der inzwischen flotte Compiler generiert ansehnlichen Code, der den gängigen Formatierungsrichtlinien entspricht.

Entstehungsgeschichte

TypeScript wurde 2012 von Microsoft unter der Apache-Lizenz als Open Source entwickelt. Anders Hejlsberg erklärt im Einführungsvideo auf der Projektseite [Typ], dass die Entwicklung von JavaScript ab einer gewissen Größe nur mit speziellen Tools möglich sei, die nicht in das JavaScript-Ökosystem eingebunden seien. Dadurch verliere man den komfortablen Zugriff auf vorhandene Bibliotheken.

Deswegen wurde TypeScript entwickelt, das zu sauber formatiertem reinen JavaScript kompiliert wird und somit überall lauffähig ist. Durch die Abstraktion ist sinnvolles Tooling möglich. Ebenso optionale statische Typisierung und weitere nützliche Features, die große JavaScript-Codebasen verwaltbar werden lassen.

Viele Funktionen kennt man bereits aus ECMAScript 6. TypeScript nimmt hier eine Vorreiterstellung ein, indem es noch nicht vorhandene, aber von der Community gewünschte Konzepte schon jetzt zur Verfügung stellt. Dieser Trend setzt sich seit der Veröffentlichung kontinuierlich fort und zieht dadurch immer mehr Entwickler an.

Ein kleines Beispiel

Ein erstes kleines Beispiel ist in Listing 1 zu finden. Es zeigt die Verwendung des Klassensystems und von statischer Typisierung. Die öffentliche Variable **`greeting`** ist vom Typ **`string`**. Unser Roboter grüßt uns, wenn wir die kompilierte JavaScript-Datei in ein gültiges HTML-Dokument einbinden.

```
class MyRobot {
  constructor(public greeting: string) { }
  greet() {
    return "<h1>" + this.greeting + "</h1>";
  }
}
var myRobot = new MyRobot("Hello, I'm your little robot!");
var str = myRobot.greet();
document.body.innerHTML = str;
```

Listing 1: TypeScript-Beispiel MyRobot

Bei der Instantiierung der Klasse **`MyRobot`** wird also ein String erwartet. In klassischem JavaScript würde eine Anweisung wie **`new MyRobot(2)`** keinen Fehler hervorrufen. In TypeScript wird bereits vor der Kompilierung ein Fehler angezeigt. Damit können eventuelle Fehler schon vor der Ausführung erkannt werden.

Natürlich stellt sich die Frage, wie man bei der Migration einer bestehenden Codebasis vorgeht. Dafür gibt es die Möglichkeit, die Typisierung ganz wegzulassen oder den Typ **`any`** zu verwenden. Listing 2 zeigt den vom TypeScript-Compiler generierten JavaScript-Code, der auch so in einer ts-Datei von ihm akzeptiert werden würde. Es wurden alle Typinformationen entfernt, sodass ein Aufruf wie **`new MyRobot(314)`** genauso gültig wäre.

```
var MyRobot = (function () {
  function MyRobot(greeting) {
    this.greeting = greeting;
  }
  MyRobot.prototype.greet = function () {
    return "<h1>" + this.greeting + "</h1>";
  };
  return MyRobot;
})();

var myRobot = new MyRobot("Hello, I'm your little robot!");
var str = myRobot.greet();
document.body.innerHTML = str;
```

Listing 2: Vom TypeScript-Compiler generierter JavaScript-Code

Werkzeugunterstützung

Beim Tooling hat sich seit der Veröffentlichung einiges getan. Microsoft Visual Studio unterstützt TypeScript inzwischen in vollem Umfang und wird als First-Class-Member über eine Erweiterung mit Stand-alone-Compiler gehandhabt (s. Abb. 1). Dadurch entfällt eine etwaige Node.js-Installation.

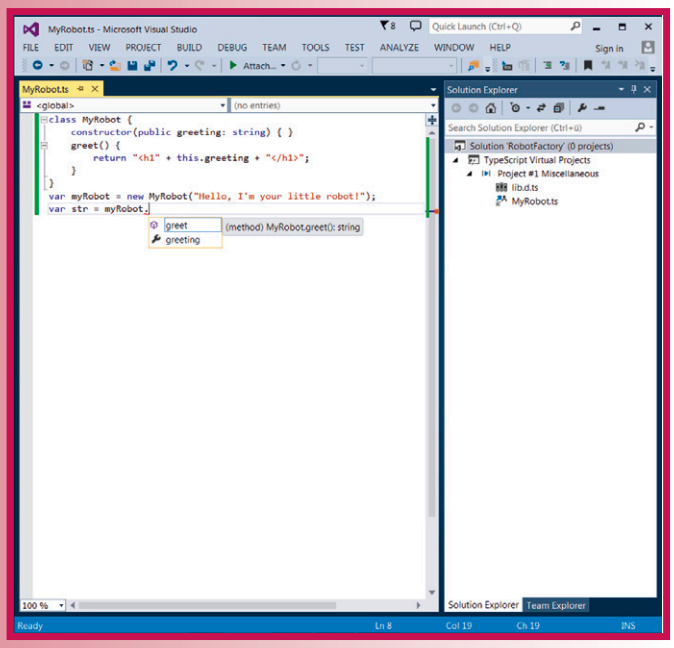

Abb. 1: Umfangreiches Tooling in Visual Studio 2013

Für Webstorm, Intellij und Eclipse wird eine Node.js-Installation benötigt. Zusätzlich sind noch die nötigen Plug-ins zu installieren. Das wären bei Eclipse lediglich TypEcs [TEcs] und bei Webstorm/Intellij JavaScript Support und Node.js [Jet].

In Eclipse muss anschließend einem Projekt noch die TypeScript-Umgebung hinzugefügt werden. Erst dann unterstützt die IDE sinnvolles Syntaxhighlighting, Code-Vervollständigung und das bequeme Autokompilieren von Dateien mit Endung `ts` (s. Abb. 2).

Weitere wichtige Sprachfeatures

Eine optionale statische Typisierung und Objektorientierung über Vererbung von Eigenschaften sowie hilfreiches Tooling sind noch nicht genug? Im Folgenden werden ausgewählte Sprachfeatures dargestellt. Bei richtigem Einsatz machen sie die Codebasis verständlicher und damit wartbarer.

Interfaces mit struktureller Typisierung

Interfaces funktionieren in TypeScript ziemlich ähnlich wie in Java. Natürlich gibt es Besonderheiten wie optionale Membervariablen oder Hybridtypen. Im normalen Entwickleralltag wird man diesen aber nicht sehr oft begegnen.

Größter Unterschied zu Java ist die strukturelle Typisierung, wie sie in Listing 3 zu sehen ist. Ein Interface beschreibt also die Struktur einer Klasse, was es unnötig macht, dass die jeweilige Klasse ihre implementierten Interfaces explizit benennt. Der Aufruf der Funktion `greetTheAudience` mit einer Instanz von `MyRobot` wird vom Compiler nicht bemängelt, da die Methode `greet` vorhanden ist.

```
interface Robot {
  greet(): string;
}

class MyRobot {
  constructor(public greeting: string) { }
  greet(): string {
    return "<h1>" + this.greeting + "</h1>";
  }
};
function greetTheAudience(robot: Robot) {
  var str = robot.greet();
  document.body.innerHTML = str;
}
var myRobot = new MyRobot("Hello, I'm your little robot!");
greetTheAudience(myRobot);
```

Listing 3: Strukturelle Typisierung

Mixins

Zusätzlich zu strukturellen Vererbungshierarchien gibt es noch ein weiteres mächtiges Werkzeug, das sich für Vererbung in Betracht ziehen lässt. Mit Mixins kann man Klassen auch ohne Ableitung mit Eigenschaften versehen. Komplizierte Vererbungshierarchien können damit manchmal radikal vereinfacht werden.

In Listing 4 werden zuerst die Mixins `Walk` und `Fly` definiert, die anschließend einem Roboter hinzufügt werden. Die Konsolenausgabe bestätigt, dass der erzeugte Roboter tatsächlich laufen und fliegen kann. Der Vorteil dieser Vorgehensweise ist klar. Soll ein Roboter erstellt werden, der nur eine der beiden Fortbewegungsmethoden beherrscht, kann einfach nur diese hinzufügt werden. Es ist keine Änderung an übergeordneten oder abgeleiteten Klassen dadurch vorzunehmen.

Leider ist an dieser Stelle schon ein großer Nachteil von Mixins erkennbar. Jede Methode (und auch Membervariable), die in eine Klasse eingewebt werden soll, muss noch einmal in der Klasse selbst als Stumpf deklariert werden. Dadurch ergibt sich ein nicht zu unterschätzender Overhead. Bis jetzt wurde dieser bekannte und von der Community schon bemängelte Umstand noch nicht angegangen. Hier sollte vielleicht ein spezielles Schlüsselwort eingefügt werden oder das

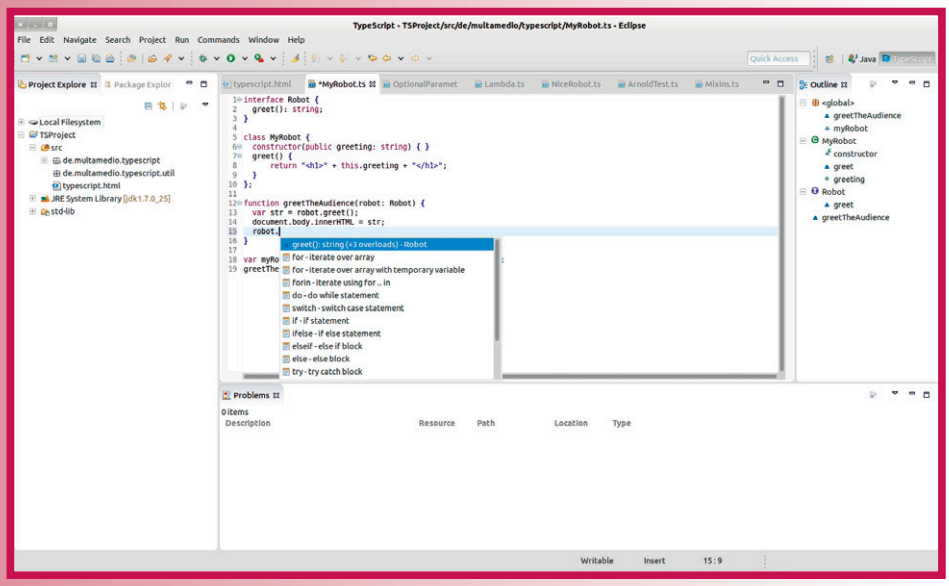

Abb. 2: TypeScript-Tooling in Eclipse

```
// Walk Mixin
class Walk {
  walk() {
    console.log("I walk");
  }
}
// Fly Mixin
class Fly {
  fly() {
    console.log("I fly");
  }
}
class MyRobot implements Walk, Fly {
  constructor() {
    setInterval(() => this.walk(), 1000);
  }
  walk: () => void;
  fly: () => void;
}
// Verwebung der Mixins mit der Klasse
applyMixins(MyRobot, [Walk, Fly])
var robot = new MyRobot();
setTimeout(() => robot.fly(), 1000);

// Funktion zur Verwebung der Klasse mit den Mixins
function applyMixins(derivedCtor: any, baseCtors: any[]) {
  baseCtors.forEach(baseCtor => {
    Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
      derivedCtor.prototype[name] = baseCtor.prototype[name];
    })
  });
}
```

Listing 4: Vererbung mit Mixins

Schlüsselwort **implements** zur Erkennung von Mixins überladen werden, wie bereits vorgeschlagen wurde (vgl. [Mix]). Dann würde der Compiler die duplizierte Angabe nicht benötigen.

Module

In TypeScript gibt es zur Modularisierung des Codes Module. Dabei wird zwischen internen und externen Modulen unterschieden (vgl. Kasten „Interne und externe Module“). Interne Module können sich dabei über mehrere Dateien erstrecken und dienen dem Zweck, den globalen Namespace nicht zu verstopfen. Die Wahrscheinlichkeit von Namenskollisionen wird damit minimiert. Sie können jedoch nicht zur Laufzeit geladen werden, da kein Mechanismus dafür zur Verfügung steht. Deswegen müssen entweder alle generierten Dateien per Hand eingebunden werden oder der Compiler erhält die Anweisung, alles in eine Datei zu verpacken.

Interne und externe Module

Interne Module bezeichnen im Grunde genommen nichts Anderes als einen separaten Namespace. Dass sie mit dem Schlüsselwort **module** deklariert werden, stiftet Verwirrung. Von einem Modul erwartet man eigentlich, dass es mit einer einzigen Zeile eingebunden werden kann. Zum Beispiel mit **require('modulname')**, wie bei AMD-Modulen. Das ist bei internen Modulen nicht der Fall. Dort muss unter Umständen jede Einzeldatei mit einer Referenz im Kopfbereich der TypeScript-Datei angegeben werden.

Tatsächlich gibt es für TypeScript 1.5 den Vorschlag, das Schlüsselwort **namespace** für interne Module einzuführen. Zum Zeitpunkt der Erstellung dieses Artikels war erst die Version 1.4 verfügbar.

Externe Module verwenden entweder CommonJS oder AMD als Laufzeitlademechanismus. Die Entscheidung über die Art des Mechanismus wird über das Compilerflag **-module** gefällt. Eine detaillierte Erörterung zu externen Modulen wird an dieser Stelle nicht gegeben.

Wieder zurück zu den für TypeScript spezifischen internen Modulen. Dazu wird in Listing 5 eine kleine Fabrik für die Roboter aus den vorherigen Beispielen entwickelt. Die Umsetzung erfolgt als internes Modul. Das Schlüsselwort **module** leitet das Modul ein. Alles, was mit dem Schlüsselwort **export** beginnt, ist von außen sichtbar. So sollen das Interface **Robot** und die Factory-Methode zur Erzeugung neuer Roboter nicht abgeschottet werden. Die Membervariable **standardMessage** und die eigentliche Klasse **MyRobot** sind nur innerhalb des Moduls sichtbar.

```
module NiceRobot {
  var standardMessage = "I am a nice robot";
  export interface Robot {
    greet(): string;
  }
  class MyRobot implements Robot {
    constructor(public greeting: string) { }
    greet(): string {
      return this.greeting;
    }
  }
  export function constructRobot(message?: string ): Robot {
    if (message) { return new MyRobot(message); }
    else { return new MyRobot(standardMessage); }
  }
}
```

Listing 5: Verwendung eines internen Moduls

Um auf das Modul von einer anderen Datei aus zugreifen zu können, wird es zuerst mit einer Referenz eingebunden. Ist es über mehrere Dateien verteilt, dann müssen alle Dateien referenziert werden. Anschließend erzeugt der Aufruf **NiceRobot.constructRobot("Asta la Vista")** einen neuen Roboter. Listing 6 zeigt das komplette Beispiel, wobei davon ausgegangen wird, dass die Datei im selben Ordner abgelegt wurde wie die Moduldatei.

```
/// <reference path="NiceRobot.ts" />
var Arnold: NiceRobot.Robot =
  NiceRobot.constructRobot("Asta la vista");
alert(Arnold.greet());
```

Listing 6: Modulzugriff

Optionale Parameter mit Standardbelegung

In Java können optionale Parameter nur über das **varargs**-Konstrukt (zum Beispiel: **String... strings**) als Parameternamen übergeben werden. Dabei werden die Parameter in ein Array umgewandelt, was zusätzlichen Overhead innerhalb der Methode bedeutet. In JavaScript sind optionale Parameter keine Seltenheit. Beiden Sprachen gemeinsam ist die nicht vorhandene Standardbelegung, wenn ein Parameter beim Aufruf nicht angegeben wird. In Java ist das Array leer und in JavaScript ist die betreffende Variable **undefined**. TypeScript enthält sowohl optionale Parameter als auch eine mögliche Standardbelegung.

Listing 7 veranschaulicht deren Verwendung. Vor allem Default-Parameter können zur Lesbarkeit des Codes erheblich beitragen, da sie Evaluationen von bedingten Auswertungen am Anfang der Methode überflüssig machen.


```
// Funktion mit optionalen Parameter
function printGreetingMessageRobot(robot?: Robot) {
  var str = "<h1>I am no robot...</h1>";
  if (robot) {
    str = robot.greet();
  }
  document.body.innerHTML = str;
}
// Funktion mit default Parameter
function printGreetingMessage(message =
    "<h1>I am your function</h1>") {
  document.body.innerHTML = message;
}
var myRobot = new MyRobot("Hello, I'm your little robot!</h1>");
printGreetingMessageRobot(myRobot);
  // Ausgabe: Hello, I'm your little robot!
printGreetingMessageRobot(); // Ausgabe: I am no robot...
printGreetingMessage(); // Ausgabe: I am your function
```

Listing 7: Optionale Parameter

Lambdas und this

Als JavaScript-Neuling und vor allem als Java-Entwickler stiftet die Funktionsweise von `this` des Öfteren Verwirrung. Das liegt hauptsächlich daran, dass die Objektorientierung von JavaScript über Prototypen realisiert ist und damit `this` nicht die aktuelle Instanz einer Klasse bezeichnen kann. Das Schlüsselwort `this` bezeichnet vielmehr den aktuellen Aufrufkontext, was in JavaScript nicht unbedingt das beinhaltende Objekt sein muss.

So einfach diese Definition klingt, so schwierig kann sie in der Praxis sein. Listing 8 zeigt ein Beispiel aus dem TypeScript-Handbuch [Typ]. Beim Aufruf von `cardPicker()` zeigt `this` nicht auf das übergeordnete Objekt `desk`, sondern laut Definition auf das aufrufende Element `window` (im strikten Modus sogar auf `undefined`). Wird der Code so ausgeführt, ist die Folge eine Fehlermeldung, da `this.suits` nicht definiert ist. Eine Lösung wäre, den gewünschten Kontext in einer Variablen `that` in `desk` abzulegen. Damit kann das eine Ebene tiefer gelegene Objekt `createCardPicker` darauf zugreifen. Aber schön und elegant ist so etwas natürlich nicht.

```
var deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function() {
    return function() {
      var pickedCard = Math.floor(Math.random() * 52);
      var pickedSuit = Math.floor(pickedCard / 13);
      return {suit: this.suits[pickedSuit], card: pickedCard % 13};
    }
  }
}
var cardPicker = deck.createCardPicker();
var pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Listing 8: Das Schlüsselwort this in JavaScript verhält sich unerwartet

```
...
createCardPicker: function() {
  // Lambdaausdruck legt this bei der Erzeugung fest
  return () => {
    var pickedCard = Math.floor(Math.random() * 52);
    var pickedSuit = Math.floor(pickedCard / 13);
    return {suit: this.suits[pickedSuit], card: pickedCard % 13};
  }
}
...
```

Listing 9: Lambdas legen den Scope von this bei der Erzeugung fest

Abhilfe schafft TypeScript mit einem sogenannten Lambdaausdruck. In Listing 9 sorgt die angepasste Funktion dafür, dass `this` bereits bei der Erzeugung der Funktion festgelegt wird. Dadurch verhält sich `this` so, wie man es intuitiv erwarten würde. Mit der Änderung von Listing 9 wird die gewünschte Alert-Box angezeigt.

Externe Bibliotheken

Ohne externe Libraries und Frameworks geht heutzutage in JavaScript nichts mehr. Da TypeScript ein Typsystem verwendet, werden Definitionsdateien mit der Endung *.d.ts benötigt. Ansonsten erkennt der Compiler zum Beispiel die Funktion $ aus jQuery nicht, da ihm der entsprechende Typ fehlt. Bei einer Zeile wie

```
$("#body").innerHtml("I am from jQuery")
```

würde er die Kompilierung schlicht und einfach verweigern.

Das Projekt stößt aber auf große Gegenliebe bei Entwicklern und Unternehmen, sodass sich mit dem Projekt DefinitelyTyped [Def] eine Anlaufstelle für Definitionsdateien gebildet hat. Dort können Definition Requests abgegeben werden und es befinden sich dort inzwischen Definitionsdateien für alle gängigen Bibliotheken. Das dazugehörige GitHub-Repository umfasst zur Zeit der Erstellung dieses Artikels mehr als 700 Definitionsdateien und es werden fast täglich mehr.

Eine d.ts-Datei wird als Referenz mit der bekannten TypeScript-Syntax in Dateien eingebunden, die die entsprechende Bibliothek verwenden wollen. Um bei jQuery zu bleiben, folgt an dieser Stelle die dazugehörige Zeile:

```
/// <reference path="jquery.d.ts" />
```

Typischerweise wird die Referenz im Kopfbereich einer TypeScript-Datei eingebunden. Dann ist die Funktion $ dort uneingeschränkt nutzbar.

Praxiserfahrung

Wie sieht die Arbeit mit TypeScript in der Praxis aus? In einem kleinen, aber dennoch komplexen Projekt, mit abschließend über 3000 Zeilen Code, hat sich TypeScript als sehr nützlich erwiesen. Dort wurde zuerst mit reinem JavaScript entwickelt, was sich zunehmend als unüberschaubar herausstellte. Die Herausforderung bestand darin, asynchrone Methodenaufrufe mit Promises aus der Bibliothek Q und DOM-Manipulationen mit jQuery zu implementieren. Durch das fehlende Tooling wurden die geschachtelten Aufrufhierarchien schwer nachvollziehbar. Änderungen waren schon in der fortgeschrittenen Anfangsphase mühsam und fehlerträchtig.

Der Einsatz von Interfaces und zu weiten Teilen statischer Typisierung ging leicht, aber nicht ohne Aufwand, von der Hand. Am Ende standen klar definierte Hierarchien sowie Funktionen. Die vom Compiler durchgeführte Typprüfung brachte so manchen schlummernden Fehler ans Tageslicht. Diese Spezialfälle wären erst sehr spät und zur Laufzeit aufgetreten. Schon alleine dafür lohnt sich der Aufwand.

Für die beiden eingesetzten Bibliotheken stehen vollständige Definitionsdateien zur Verfügung, die von hoher Qualität sind. Lediglich in Q konnte eine Funktion nicht wie gewohnt benutzt werden.

Insgesamt erinnert die Codebasis an Java-Code, sodass sie von großen Teilen der Entwickler im Unternehmen leicht verstanden werden kann. Als Bonus wurde an Stellen, an denen reines JavaScript ausdrucksstärker und mächtiger ist (Stichwort: funktionale Programmierung) einfach darauf zurückgegriffen.

Zusammenfassung und Ausblick

TypeScript ist für sein junges Alter eine sehr ausgewachsene Sprache mit stetig wachsender Community. Die Entwickler orientieren sich an den ECMAScript-Standards, was das erworbene Wissen sogar auf reines JavaScript übertragbar macht. Bis auf kleinere Schwächen der Sprache enthält sie eine ganze Reihe wichtiger Abstraktionen und Vereinfachungen, wie Vererbung mit Interfaces, Lambdaausdrücke, ein Modulkonzept und noch mehr.

Mit diesem Werkzeugkasten ist die Entwicklung von JavaScript-Anwendungen in großem Stil und ohne ins Chaos abzugleiten möglich. Das wird dadurch bestätigt, dass bereits größere Unternehmen und Entwickler von Frameworks auf TypeScript setzen. Erst im März hat AngularJS angekündigt, die Version 2.0 mit TypeScript zu entwickeln [Msd].

Auch das entstandene Ökosystem um TypeScript mit seiner aktiven Community zeigt, dass ein breites Interesse von Entwicklerseite besteht. Dem Projekt DefinitelyTyped werden wöchentlich neue Definitionsdateien hinzugefügt, mit denen sich JavaScript-Bibliotheken nahtlos in TypeScript einbinden lassen.

Die Roadmap bis zum Release 2.0 liest sich ebenfalls vielversprechend. So soll TypeScript an ECMAScript 6 angeglichen werden, um als Obermenge für die nächsten Versionen zur Verfügung zu stehen. Dazu werden noch fehlende Features, wie Destrukturierung, Promises oder Iteratoren, implementiert. Außerdem wird das Engagement innerhalb der JavaScript-Gemeinschaft weitergeführt. So sollen weitere Kooperationen entstehen, um TypeScript weiterzuentwickeln.

Die persönliche Praxiserfahrung des Autors hat gezeigt, dass TypeScript eine ernst zu nehmende und angenehm zu verwendende Sprache ist. Eine aus dem Ruder laufende JavaScript-Anwendung konnte damit wieder verwaltbar gemacht werden. Das Ziel, JavaScript im Applikationsumfang wartbar und möglich zu machen, hat Microsoft damit erreicht.

Links und Literatur

[Def] DefinitelyTyped, TypeScript-Definitionsdateien für JS-Libraries, `http://definitelytyped.org/`

[Jet] Jetbrains, TypeScript-Support in IntelliJ IDEA, `https://www.jetbrains.com/idea/help/typescript-support.html`

[Mix] Microsoft, Dokumentation Mixins, `https://typescript.codeplex.com/wikipage?title=Mixins%20in%20TypeScript`

[Msd] J. Turner, Angular 2: Built on TypeScript, 2015, `http://blogs.msdn.com/b/typescript/archive/2015/03/05/angular-2-0-built-on-typescript.aspx`

[TEcs] TypEcs, TypeScript IDE for Eclipse, `http://typecsdev.com/`

[Typ] TypeScript, Projektseite, `http://www.typescriptlang.org`

Johannes Dienst ist Softwareentwickler bei der MULTA MEDIO Informationssysteme AG in Würzburg. Er ist leidenschaftlicher Java-Entwickler, wobei ihm das Thema Clean Code sehr am Herzen liegt. In seiner Freizeit beschäftigt er sich gerne mit allen Themen rund um Webentwicklung.
E-Mail: jdienst@multamedio.de