ilionx
Your partner in digital business

De kracht van TypeScript

Met TypeScript is programmeren voor het web eenvoudiger, leuker en volwassener geworden. Waar JavaScript voor veel Java-ontwikkelaars ondoorgrondelijk was, is TypeScript een intuïtieve taal met een verfijnder typesysteem dan Java. Nieuwsgierig geworden? Lees verder en zie wat je als Java-ontwikkelaar van TypeScript kunt leren. Al bekend met TypeScript? Maak dan kennis met misschien nog onbekende features, zoals intersection types, function types en generics.

Van JavaScript naar TypeScript

JavaScript heeft onder veel Java-ontwikkelaars een slechte naam. Het klassensysteem is lastig te doorgronden, het gedrag van diverse keywords is moeilijk voorspelbaar en tot overmaat van ramp ontdek je codeerfouten pas in je browser. Op de eerste twee punten is met JavaScript 6 (officieel ECMAScript 2015) al veel verbeterd. Sinds JavaScript 6 kun je klassen op een vergelijkbare manier declareren als in Java, kun je variabelen declareren met block-scope en doet het this-keyword wat je ervan verwacht. Het modulesysteem zorgt er bovendien voor dat je je code in duidelijke blokken kunt organiseren (*zie Listing 1*).

JavaScript 6 is echter nog steeds een dynamische taal, zonder enige vorm van compile time type checking. De kans op fouten is weliswaar kleiner, maar nog steeds ontdek je die fouten vaak pas in je browser. Daarin brengt TypeScript verandering. TypeScript is een uitbreiding op JavaScript 6 met een uitgebreid typesysteem. Daardoor kun jij duidelijkere code schrijven en is het voor IDE's en compilers veel gemakkelijker om tijdens het coderen al fouten te ontdekken en code completion toe te passen. Dat maakt het voor jou veel simpeler om aan grotere projecten te werken zonder de weg kwijt te raken.

TypeScript is ontwikkeld door Microsoft en is volledig open source. Goede editors zijn WebStorm/IntelliJ en Visual Studio Code (gratis).

Webframeworks als Angular maken dankbaar gebruik van TypeScript, maar eigenlijk is TypeScript voor alle webapplicaties heel geschikt.

```
// ieder broncodebestand vormt een module
// het export keyword maakt de klasse ook buiten de module beschikbaar
export class Country {

  // een klassedefinitie is vergelijkbaar met Java.
  name;
  languages;

  constructor(name, languages) {
  // this verwijst naar de instantie van de klasse, zoals je ook verwacht
    this.name = name;
    this.languages = languages;
  }

  speaks(aLanguage) {
  // let definieert een variabele met block-scope
  // language is buiten de for-loop niet beschikbaar
    for (let language of this.languages) {
      if (language === aLanguage) {
        console.log('YES!');
      }
    }
  }
}
```

Listing 1

Transpilatie

Je kunt TypeScript overal gebruiken waar je nu al JavaScript gebruikt. In de meeste gevallen is dat in de browser. Veel browsers ondersteunen echter de nieuwste versies van JavaScript (en dus ook TypeScript) nog niet volledig. Om toch in de nieuwste versies van JavaScript en TypeScript te kunnen programmeren, gebruik je een transpiler. Een transpiler vertaalt jouw code naar JavaScript 5. Omdat de transpiler ook mapping-files genereert, weet de browser met welk oorspronkelijk regelnummer de uitgevoerde code correspondeert. Daardoor kun jij in de browser gewoon breakpoints op je eigen code zetten en is het net alsof je TypeScript-code aan het draaien bent.

Om de voorbeelden in dit artikel te draaien, zorg je ervoor dat je node geïnstalleerd hebt

(https://nodejs.org). Daarna installeer je met npm install -g typescript de TypeScripttranspiler. Met tsc listingx.ts transpileer je een TypeScript-bestand naar JavaScript. Met node listingx.js voer je de code uit.

Type safety

TypeScript voegt type safety toe aan JavaScript. Code die iets probeert te doen wat volgens de types niet kan, veroorzaakt al tijdens het transpileren foutmeldingen. Goede editors (zoals Visual Studio Code en Webstorm/IntelliJ) markeren fouten zelfs al in de editor.

Je voegt types toe door achter een declaratie een dubbele punt te zetten en de naam van het type. In *Listing 2* leggen we nogmaals een klasse vast, maar nu met expliciete types. Types als number, string en boolean zijn net als in JavaScript ook in TypeScript geldig. En net als in JavaScript 6 kun je ook gebruik maken van tupels. Een tupel is een combinatie van waarden die in één keer aan een enkele variabele kan worden toegewezen.

Een tupel kan bijvoorbeeld gebruikt worden om gemakkelijk meerdere waarden uit een functie tegelijkertijd terug te geven. De info()-methode in het voorbeeld geeft een tupel [string, number] terug. Later roepen we de functie aan en maken we gebruik van een zogenaamde destructuring assignment om de waarden van de tupel aan afzonderlijke variabelen toe te wijzen.

```
export class Country {

  public name: string;
  private languages: string[];

  constructor(name: string, languages: string[]) {
    // ...
  }

  info(): [string, number] {
    return [this.name, this.languages.length];
  }
}

let country: Country = new Country('Nederland', ['nl', 'fy']);

// met een zogenaamde destructuring assignment kennen we de
// waarden uit de tuple toe aan afzonderlijke variabelen
let [name, numberOfLanguages]: [string, number] = country.info();

console.log(name);
console.log(numberOfLanguages);
```

Listing 2

Type inference

Dankzij de diamond operator kun je in Java generieke types vastleggen met minder code. De Java-compiler leidt zelf af welk generiek type je bedoelt. Dankzij de diamond operator heb je net zo veel type safety, maar hoef je er een stuk minder voor te typen. TypeScript gaat nog een stapje verder. TypeScript leidt niet alleen voor generieke types af welk type je bedoelt, maar kan dit doen voor alle types. Het expliciet vastleggen van types is daarmee optioneel. Zodra je de expliciete typering bij een declaratie weglaat, leidt de transpiler af welk type je bedoelt. Dit principe van type inference kennen we ook uit andere talen, zoals Scala en Kotlin.

In *Listing 3* wijzen we opnieuw een Country toe aan een variabele. Het ligt voor de hand dat die variabele van het type Country moet zijn. Daarom laten we het expliciete type weg en leidt de transpiler het type zelf af. De code is daarmee korter en bondiger dan in *Listing 2*. Toch kunnen we later niet ineens een waarde van een ander type aan de variabele toewijzen. Dat leidt namelijk tot een transpilatiefout. Als we wel willekeurige types aan een variabele willen toewijzen, kunnen we de variabele het type any geven. Doordat we expliciet een type opgeven, past de transpiler geen type inference toe en kunnen we aan dezelfde variabele eerst een Country en daarna een getal toewijzen.

```
import {Country} from './listing-2';

let country = new Country('...', []);
// transpileert niet; 3 is geen Country
country = 3;

let anything: any = new Country('...', []);
// transpileert wel; dankzij 'any' mogen we alles toewijzen
anything = 3;
```

Listing 3

Interfaces, excess property checks en duck typing

In JavaScript kun je niet alleen objecten maken door klassen te instantiëren, maar ook ad hoc, zonder dat er een definitie van de definitie aan voorafgaat. Dat maakt JavaScript heel flexibel, maar zonder type safety ook erg foutgevoelig. Ook in deze gevallen leidt TypeScript automatisch types af. In *Listing 4* is language van het type {code: string; name: string}.

Als we vervolgens een typo maken en de property namo proberen te zetten in plaats van name, merkt de transpiler de typo op. Dat noemen we een excess property check. Gaat het hier niet om een typo, maar om een bewuste actie? Dan kun je ook hier de variabele het type any geven. Je schakelt de excess property check daarmee uit.

language is van het type {code: string; name: string}. Zo'n type kunnen we ook expliciet maken door het vast te leggen in een interface. De Named interface in *Listing 4* legt vast dat een object een property name van het type string moet hebben. Elk object dat zo'n property heeft, kan worden meegegeven aan de print-functie. Merk op dat de Countryklasse niet expliciet de Named-interface implementeert. Je kunt weliswaar klassen expliciet interfaces laten implementeren, net als in Java, maar TypeScript vereist dat niet. In plaats daarvan maakt TypeScript gebruik van duck typing. Als een object toevallig een name-property van het type string heeft, voldoet het automatisch aan de interface. Dat hoeft niet zoals in Java van tevoren vastgelegd te zijn. Ook het Donald-Duck-object voldoet dus aan de interface.

```
import {Country} from './listing-2';

let language = {code: 'nl', name: 'Dutch'};
// Fout: Property 'namo' does not exist on type '{ code: string;
name: string; }'
language.namo = 'Nederlands';

export interface Named {
  name: string;
}

function print(named: Named) {
  console.log(named.name);
}

print(new Country('Nederland', []));
print({name: 'Donald Duck'});
```

Listing 4

Intersection types

Doordat je met duck typing heel ad hoc typering kunt toepassen, is het ook heel eenvoudig om ad hoc types met elkaar te combineren. In *Listing 4* zagen we een print-functie, die gebruik maakte van de name-property van het argument. De Named-interface dwong af dat een argument een name-property had. In *Listing 5* voegen we een tweede interface toe. De Aged interface vereist dat een object een age-property heeft. In de functie printWithAge drukken we zowel de naam als de leeftijd van het argument af. Om af te dwingen dat het argument beide properties heeft, maken we gebruik van een intersection type. Het &-teken geeft aan dat subject zowel aan de Named-interface moet voldoen, als aan de Aged-interface. Het is dus niet nodig om hiervoor expliciet een nieuwe interface vast te leggen, die beide andere interfaces uitbreidt, zoals in Java. Met het &-teken kun je ter plekke meerdere interfaces met elkaar combineren, waar je dat maar nodig acht.

Binnen de printWithAge-functie maken we gebruik van een zogenaamde template literal. De template literal komt uit JavaScript 6 en zorgt ervoor dat je gemakkelijk strings kunt concateneren en formatteren. In plaats van alle losse string met plusjes achter elkaar te plakken, vullen we tussen accolades expressies in, die in de string ingevuld moeten worden. Deze techniek is vergelijkbaar met string interpolation in Scala of de GString in Groovy.

```
import {Named} from './listing-4';

interface Aged {
  age: number;
}

function printWithAge(subject: Named & Aged) {
  console.log(`${subject.name} (${subject.age})`);
}

// compileert niet: het argument voldoet niet aan beide interfaces
printWithAge({name: 'Donald Duck'});
```

Listing 5

Function types

TypeScript is net als JavaScript een functionele taal. Functies zijn in TypeScript first class citizens. Ze kunnen worden toegekend aan variabelen en ze kunnen worden meegegeven als parameters aan andere functies. Net als strings, arrays en tuples hebben ook functies een type. In Java 8 is hiervoor de functional interface geïntroduceerd. Een functional interface is een interface, die bestaat uit precies één methode. Iedere lambda-functie met dezelfde function signature als die ene methode voldoet aan de functional interface. De lambda-functie (String tekst) -> "Java Magazine".equals(tekst) voldoet aan de functional interface Predicate<String>, omdat er een String in gaat en een Boolean uitkomt. Je kunt in Java alleen een type van een functie vastleggen door er expliciet een functional interface voor te declareren.

TypeScript gaat veel flexibeler om met functietypes. In JavaScript 6 (en dus ook in TypeScript) kun je met een zogenaamde arrow function inline een functie declareren, vergelijkbaar met de lambda-functie uit Java 8. In *Listing 6* declareren we op die manier een functie en wijzen we die toe aan de variabele isLeuk. Ook bij functies past TypeScript automatisch type inference toe. In dit geval leidt de transpiler af dat isLeuk het type (string) => boolean heeft.

Net als in Java kunnen we functietypes ook expliciet vastleggen in een interface. Dat doen we in het type Predicate<T>. Iedere functie waar een type T in gaat en die een boolean teruggeeft, voldoet aan deze interface. T is een generic type parameter, zoals we die in Java ook kennen. IsLeuk voldoet aan de interface. Zodoende kunnen we isLeuk meegeven als parameter aan een functie die een Predicate<T> verwacht. Dankzij type inference en duck typing hoeven we isLeuk hier dus niet expliciet een interface te laten implementeren, zoals dat bij Java wel moet.

```
// isLeuk heeft inferred type (string) => boolean
let isLeuk = (tekst: string) => tekst === 'Java Magazine';

// iedere functie (T) => boolean voldoet aan de interface Predicate<T>
interface Predicate<T> {
  (subject: T): boolean;
}

// filter verwacht als tweede parameter een Predicate<T>
function filter<T>(objs: T[], predicate: Predicate<T>) {
  // ...
}

// isLeuk is een Predicate<String> en dus is onderstaande aanroep geldig
let objs = ['Java Magazine', '.NET Magazine'];
filter(objs, isLeuk);
```

Listing 5

En verder

In dit artikel heb ik geprobeerd om in een aantal voorbeelden de kracht van TypeScript te illustreren. Zaken als type inference, duck typing en function types maken TypeScript misschien wel krachtiger dan Java en maken het heel aangenaam om aan grotere webprojecten te werken. Ben je nieuwsgierig geworden en wil je ook meer weten over alle andere mogelijkheden van TypeScript? Kijk dan zeker even op www.typescriptlang.org, waar de taal volledig beschreven en uitgelegd staat.

Meer lezen?

Blog Jabba the Hut, the sexy parts

Michel Vollebregt
Senior Java Developer - ilionx
mvollebregt@ilionx.com