TypeScript 3.0 – co nowego u webowego dziecka Microsoftu

Niecałe dwa lata od wydania wersji 2.0, a jednocześnie zaledwie dwa miesiące od (ostatniej) wersji 2.9, Microsoft wypuszcza nową wersję języka TypeScript. Czy zmiany są aż tak istotne, że należało przekręcić licznik? A może jest to tylko kolejna aktualizacja, a wersja wynika jedynie z tego, że nie chciano wydawać wersji 2.10? Przyjrzymy się, co nowego przygotowano dla programistów, i spróbujmy sobie odpowiedzieć na te pytania.

Wprowadzenie do TypeScript

Jeżeli już miałeś do czynienia z językiem TypeScript, możesz śmiało pominąć ten rozdział artykułu. W przypadku, gdy nie miałeś okazji się z nim zapoznać, tutaj znajdziesz szybkie wprowadzenie do tematu.

TypeScript to język programowania transkompilowalny do JavaScriptu, stworzony przez Microsoft i rozwijany przez społeczność open source [1]. Najważniejszą jego cechą, od której zresztą wywodzi się nazwa, jest możliwość nadawania typów. Stanowi on nadzbiór JavaScriptu, implementujący wszystkie nowości standardu ECMAScript, a także dodający własne elementy składni, takie jak interfejsy, typy enumeracyjne czy modyfikatory dostępu. To, że TypeScript jest nadzbiorem JavaScriptu, stanowi jego największą zaletę – dzięki temu przejście na ten język jest zdecydowanie prostsze niż na inne języki kompilowalne do JS (składnia przypomina mieszankę JavaScriptu z Javą bądź C#). W praktyce każdy kod napisany w JavaScript jest także poprawnym kodem TypeScriptowym. Dodatkowo możemy wykorzystywać biblioteki, które nie są pisane z myślą o TypeScript.

Dlaczego natomiast typowanie jest najważniejszą cechą? Spośród wielu zalet uważam, że najważniejszymi są zwiększenie czytelności kodu, wczesne wykrywanie błędów, a także lepsze wsparcie z poziomu IDE (celniejsze podpowiedzi). Pamiętajmy, że JavaScript sam w sobie jest językiem słabo typowanym, więc w każdej chwili możemy zmieniać typ zmiennej. Jest to na swój sposób wygodne, jednak może prowadzić do nieoczekiwanych błędów. Samo wprowadzenie TypeScripta, czy innych narzędzi do sprawdzania typów (jak np. Flow), nie naprawi automatycznie każdego projektu, jednak może wyeliminować część błędów na poziomie kompilacji. Typy wymuszają także czystość kodu - zmienne nie zmieniają w sposób nagły swojego typu, co zdarza się w kodzie napisanym w JavaScript.

Biorąc to wszystko pod uwagę, TypeScript staje się interesującym wyborem jako język programowania. W kontekście przeglądarkowej hegemonii JavaScriptu, który wciąż cierpi na niedoskonałości ze względu na nienajlepsze rozwiązania powstałe w dawnych wersjach standardu ECMAScript, warto rozejrzeć się za jego alternatywami. TypeScript jest alternatywą dojrzałą, biorącą to co najlepsze ze świata JavaScriptowego dodając do tego własne elementy tak, aby język był gotowy do bardziej zaawansowanych projektów i zarazem przyjazny zarówno dla dotyczasowych JavaScriptowców, jak i programistów silnie typowanych języków obiektowych.

Aby zobaczyć, jak wygląda TypeScript i jaki wynikowy kod JavaScript z niego powstaje, warto sprawdzić stronę http://www.typescriptlang.org/play/index.html, gdzie można znaleźć edytor online kompilujący TypeScript do JavaScript na bieżąco, a ponadto zawierający kilka przykładów pokazujących różne podstawowe aspekty języka. Warto poznać podstawy przed zapoznaniem się z

resztą artykułu, aby mieć wyobrażenie, jak TypeScript ma się do JavaScriptu i jak wpływają na to pokazane tutaj nowości.

Nowości wersji 3.0

TypeScript 3.0, mimo zmiany *major version*, nie wprowadza bardzo wielu zmian względem poprzedniej wersji (2.9 z maja 2018) z punktu widzenia użytkownika. Zmian w operowaniu językiem jest jedynie pięć, które wprowadzają zaledwie jedną *breaking change*, powodującą możliwą niekompatybilność starego kodu z nową wersją [2]. Niewiele zmian jednak nie oznacza, że zmiany te nie są istotne, wręcz przeciwnie. Przejrzyjmy je po kolei.

Rozbudowane typy krotkowe

Krotki w TypeScript to tak naprawdę JavaScriptowe tablice, tyle że z ograniczeniem liczby elementów zdefiniowanym na poziomie typów. Przynajmniej tak było do tej pory i wyglądało to tak jak w Listingu 1.

Listing 1. Krotki w TypeScript

```
type Triple = [string, number, boolean];
const a: Triple = ['a', 2, false];
```

Począwszy od TypeScript 3.0, będziemy mogli definiować krotki z niezdefiniowaną maksymalną liczbą elementów, a jedynie ich minimalną. Przykład możemy zobaczyć w Listingu 2.

Listing 2. Krotka bez zdefiniowanej maksymalnej liczby elementów

```
type StringAndNumbers = [string, ...number[]];
const b1: StringAndNumbers = ['a', 1, 2, 3];
const b2: StringAndNumbers = ['a'];
```

Pierwszy element typu `string` jest obowiązkowy, natomiast kolejne, które będą liczbami, już mogą być pominięte. Co więcej, możemy dzięki temu określić także krotkę bez minimalnego ograniczenia liczby elementów (*de facto* jest to typowa tablica), jak i pustą krotkę (patrz Listing 3).

Listing 3. Krotka pusta i bez ograniczenia liczby elementów

```
type Strings = [...string];
type Empty = [];
const c1: Strings = ['a', 'b'];
const c2: Strings = [];
const c3: Empty = [];
const c4: Empty = ['a', 'b']; // błędny typ
```

Takie krotki mają oczywiście zastosowania praktyczne, np. możemy narzucić na poziomie typów, że tablica ma posiadać co najmniej jeden element. Inne zastosowanie, w bardziej zaawansowanym przypadku, jest przedstawione w następnym rozdziale.

Rozwijanie list parametrów z użyciem typów krotkowych

Jedną z cech JavaScript jest możliwość korzystania z argumentów przekazanych do funkcji nawet wtedy, gdy jawnie nie zdefiniowaliśmy ich. Przykładowo kod JavaScriptowy z Listingu 4 zadziała i zwróci wynik „a,0,1,c”:

Listing 4. Funkcja zwracająca w postaci tekstowej podane argumenty

```
function hello() {
```

```
      return [...arguments].join(',');
}
hello('a', 0, 1, 'c');
```

Kod ten jednak nie zadziała przy domyślnych ustawieniach TypeScripta – kompilator widzi, że funkcja `hello()` oczekuje 0 argumentów, a otrzymuje 4, co jest dla niego błędem, i nieistotne jest, że wykorzystujemy je potem w kodzie. Można włączyć możliwość skompilowania takiego kodu odpowiednią flagą kompilatora, jednak nie jest to zalecane. Poprawny TypeScriptowy odpowiednik został przedstawiony w Listingu 5.

Listing 5. Funkcja z listingu 4 przerobiona na TypeScript

```
function hello(...args: any[]): string {
      return args.join(',');
}
```

W tym momencie wszystko jest w porządku dla kompilatora, jednak należy zwrócić uwagę na dwie rzeczy. Po pierwsze, wykorzystujemy typ `any`, który może pomieścić, co tylko chcemy, i w praktyce działa jako pominięcie sprawdzania typów. Jednak co zrobić, gdybyśmy chcieli zawęzić się tylko do konkretnych typów? Możemy wówczas zastosować konstrukcję przedstawioną w Listingu 6.

Listing 6 Narzucenie, że argumenty mogą być jedynie string lub number

```
function hello(...args: (string | number)[])
      : string {
            return args.join(',');
}
```

Teraz zawężamy, że nasze argumenty mogą być jedynie typu `string` bądź `number`. Jest to kod jak najbardziej w porządku. Twórcy języka jednak poszli o krok dalej – a co, jeżeli chcielibyśmy jednocześnie określić typy i zawęzić liczbę argumentów, jednak nie rozdzielając naszego `args` na poszczególne składowe? Tutaj pojawia się właśnie nowa konstrukcja, pozwalająca nam zdefiniować wprost typy dla takiego przypadku, korzystając z krotek (patrz Listing 7).

Listing 7. Jawne określenie typu każdego z argumentów po kolei na liście

```
function hello(...args:
      [string, number, number, ...string[]])
      : string {
    return args.join(',');
}
```

Naturalnie w tym przypadku nie zachodzi większa konieczność stosowania takiego rozwiązania i można by było oczywiście wypisać wszystkie te argumenty po kolei. Gdzie jednak ma to tak naprawdę zastosowanie? Rozważmy przykład z Listingu 8, który jest bardziej z życia wzięty.

Listing 8. Funkcja tworząca złożenie dwóch funkcji

```
function compose2<T1 extends any[], T1R, T2>
      (f1: (...args1: T1) => T1R,
       f2: (arg: T1R) => T2) {
            return (...a: T1) => f2(f1(...a));
}
const add = (x: number, y: number) => x + y;
const sqr = (x: number) => x * x;
const addAndSqr = compose2(add, sqr);
addAndSqr(1, 2); // poprawne
```

```
addAndSqr('a', 2); // błędny typ
```

Przy kodzie z Listingu 8 TypeScript sam pobierze typy i liczbę argumentów pierwszej z funkcji (poprzez typ generyczny `T1 extends any[]`) . Dzięki temu, mimo iż mamy tutaj zdefiniowane `any[]` (który teoretycznie powinien akceptować dowolny typ), tak naprawdę na ostatniej linijce kodu otrzymamy błąd kompilacji. `any[]` oznacza tutaj tablicę dowolnych elementów, jednak określenie, że coś rozszerza je, w najnowszym TypeScript oznacza tyle, że tworzona jest krotka. Może się to okazać przydatne w definiowaniu typów chociażby dla bibliotek umożliwiających programowanie funkcyjne. Na przykład Ramda (jedna z najpopularniejszych bibliotek umożliwiających programowanie funkcyjne) posiada 10 różnych generycznych definicji typów dla funkcji `curry`, a także ogólną bazującą na `any`. Całkiem możliwe, że dzięki tej możliwości liczbę tę będzie można zmniejszyć, bądź też typowanie będzie mogło być dokładniejsze.

Typ unknown

Jak wspomniałem w poprzednim rozdziale, TypeScript posiada typ `any`, pod którym może być trzymane cokolwiek. Nie należy tego mylić z C#owym `var` lub C++owym `auto`, gdyż nie jest to typ inferowany przez kompilator, lecz narzucenie JavaScriptowej słabej typizacji. Generalnie `any` zostało pozostawione jako furtka dająca kompatybilność z JavaScript, jednak jego używanie nie jest najlepszą praktyką i powinno się go unikać, jeżeli to tylko możliwe. Nawiasem mówiąc, w TypeScript kompilator automatycznie określi typ w przypadku, gdy go jawnie niezdefiniujemy.

Są jednak przypadki, kiedy typu faktycznie nie znamy i musimy go dopiero określić. Do tej pory używało się do tego `any`. Niestety, wówczas TypeScript nie będzie zwracał błędów na jakiejkolwiek operacji, jaką byśmy wykonywali z taką zmienną. Mimo wszystko nie jest to pożądane zachowanie. Chcielibyśmy, aby zmienna posiadała wartość, jednak dopóki nie określimy wprost, jaki ma typ, nie możemy nic z nią zrobić. Do takich przypadków wprowadzono w najnowszej wersji języka nowy typ: `unknown`. W Listingu 9 pokazana jest różnica w działaniu typów `unknown` oraz `any`. W komentarzu podano wynik działania, bądź też w którym momencie został rzucony błąd.

Listing 9. Różnice pomiędzy any i unknown

```
const a1: any = 'a';
const a2: unknown = 'a';
a1.length; // = 1
a2.length; // błąd kompilacji
a1(); // błąd wykonania
a2(); // błąd kompilacji
new a1(); // błąd wykonania
new a2(); // błąd kompilacji
const a3 = a1 + 3; // = 'a3'
const a4 = a2 + 3; // błąd kompilacji
```

Aby korzystać z wartości zmiennej, należy upewnić się co do jej typu, chociażby korzystając z `typeof`. Przykład można zobaczyć w Listingu 10.

Listing 10. Wykorzystanie sprawdzenia typów w celu korzystania ze zmiennej o typie unknown

```
const b: unknown = { a: 'b' };
console.log(b.a); // błąd kompilacji
function hasA(obj: any): obj is { a: any } {
    return !!obj
        && typeof obj === 'object'
        && 'a' in obj;
```

```
}
if (hasA(b)) {
    console.log(b.a); // = 'b'
}
```

Można także narzucić typ poprzez asercję typów (ang. *type assertion*), jednak wówczas tracimy sprawdzenie typu w trakcie wykonania kodu.

Na sam koniec dodam, że to jest właśnie wspomniany przeze mnie wcześniej *breaking change*. Jest nią dlatego, że począwszy od wersji 3.0, `unknown` staje się słowem kluczowym języka, co może powodować niekompatybilność starych projektów z najnowszą wersją TypeScriptu [3].

Wsparcie dla defaultProps w JSX

Kolejną ciekawą nowością w TypeScript 3.0 jest wsparcie na poziomie języka dla domyślnych wartości parametrów w Reactowych JSXach, znanych jako `defaultProps`. Dla osób, które nie miały do czynienia z tematem, pokrótce opiszę, w czym rzecz. `defaultProps` umożliwiają ustalenie domyślnych wartości parametrów komponentu Reactowego. Standardowo w JavaScript (ECMAScript 6) wygląda to następująco:

Listing 11. Użycie defaultProps w JavaScript

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
class Heading extends React.Component {
      render() {
            return <h1>{this.props.title.toUpperCase()}</h1>
      }
}
Heading.defaultProps = { title: 'Hello!' };
const elem = document.querySelector('#target');
ReactDOM.render(<Heading />, elem);
```

Jak możemy zauważyć, kod komponentu przy renderowaniu odwołuje się do `title`, którego nie definiujemy, tworząc komponent (w `ReactDOM.render`). Nie mamy żadnego zabezpieczenia przed niezdefiniowaną wartością, więc liczymy na to, że będzie on zawsze ustalony. Kod działa dzięki zdefiniowaniu statycznego obiektu `defaultProps`, gdzie ustaliliśmy domyślną wartość.
Dotychczas w TypeScript nie było to takie proste. Z racji odgórnie definiowanych typów, jeżeli coś jest opcjonalne, to musi też tak być otypowane. Oznacza to, że dla kompilatora „title” mogłoby być niezdefiniowane, więc nie można by wykonać na nim żadnej operacji. Dlatego powyższy komponent w TypeScript wyglądałby tak jak przedstawiono w Listingu 12 (pomijając importy i kod renderujący komponent w DOMie).

Listing 12. Komponent z listingu 11 przepisany na TypeScript

```
type Props = {
      title?: string;
};
class Heading extends React.Component<Props> {
      static defaultProps = {
            title: 'Hello!'
      };
      render() {
            const title = this.props.title;
            return <h1>{title && title.toUpperCase()}</h1>
      }
```

```
}
```

Jak widać, musieliśmy dołożyć dodatkowy fragment kodu odpowiadający za sprawdzenie, czy `title` istnieje, i dopiero wtedy wykonaliśmy akcję zamiany znaków na kapitaliki. Kod można by oczywiście uprościć, stosując operator `!.` (*non-null assertion operator*). Otrzymamy wtedy `title!.toUpperCase()`, jednak wciąż nie jest to idealne rozwiązanie, ponieważ kod po transpilacji do JavaScriptu otrzyma niepotrzebną operację sprawdzenia, czy zmienna nie jest nullem (`return React.createElement("h1", null, title && title.toUpperCase());`). TypeScript 3.0 rozwiązuje ten problem. Kompilator otrzymał dodatkowy helper zwany `LibraryManagedAttributes`, który odpowiada za wykrywanie takich mniej typowych z punktu widzenia języka przypadków. Stąd od najnowszej wersji kod pokazany w Listingu 13 nie będzie pokazywać błędów.

Listing 13. Uproszczony, działający kod komponentu w TypeScript 3.0

```
type Props = {
      title?: string;
};
class Heading extends React.Component<Props> {
      static defaultProps = {
            title: 'Hello!'
      };
      render() {
            const title = this.props.title;
            return <h1>{title.toUpperCase()}</h1>
      }
}
```

Jest to oczywiście ułatwienie dla programistów i wprowadzenie działania czegoś, co w JavaScript było czymś normalnym (aczkolwiek z racji natury języka w JavaScript wiele rzeczy po prostu działa). Na pewno niektórzy mogą zadać sobie pytanie, czy jest sens wprowadzać do języka programowania funkcjonalność typowo pod jedną bibliotekę. Należy w tym miejscu pamiętać, że sam format JSX powstał na potrzeby Reacta i jego wsparcie w TypeScript (jako TSX) jest wynikiem sporej popularności tej biblioteki.

Referencje do projektów

Na sam koniec postanowiłem pozostawić największą zmianę w TypeScript 3.0. Nie jest to zmiana *stricte* w składni języka (stąd pozostawiłem ją na koniec), jednak może nieść ze sobą spore konsekwencje w sposobie tworzenia projektów TypeScriptowych. Mianowicie od najnowszej wersji języka będzie można tworzyć referencje do innych projektów napisanych w TypeScript (bądź JavaScript z plikiem typingów). Aby to zrobić, należy w pliku *tsconfig.json* ustawić opcję kompilatora `composite`, a także dopisać ścieżki tsconfigów innych projektów, do których nasz się odwołuje. Przykładowy tsconfig został przedstawiony w Listingu 14, natomiast struktura przykładowego projektu składającego się z trzech innych podprojektów na Rysunku 1.

Listing 14. tsconfig.json z ustawioną flagą composite oraz referencją na zewnętrzny projekt

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "lib": ["es2015", "dom"],
    "outDir": "../../lib/client",
    "composite": true,
```

```
    "strict": true,
    "esModuleInterop": true
  },
  "references": [
    {
      "path": "../shared"
    }
  ]
}
```

Rysunek 1. Struktura projektu złożonego z innych projektów

Pokazany w Listingu 14 tsconfig należał do projektu `client` i odwołuje się on projektu `shared`. Aby korzystać z kodu zdefiniowanego w innym projekcie, importujemy go tak, jakbyśmy odwoływali się do innych części tego samego projektu, na przykład `import { User } from '../shared';` aby zaimportować klasę User wyeksportowaną w shared/index.ts. Aby skompilować projekt, korzystamy z nowej flagi kompilatora – `--build` (skrótowo: `-b`). Następnie podajemy ścieżkę do tsconfiga, z którego chcemy skorzystać. Na przykład dla projektu z Rysunku 1 skompilowanie aplikacji klienckiej zrealizujemy następującym poleceniem: tsc -b composite/client . Po skompilowaniu projektów `client` oraz `server` otrzymamy strukturę przedstawioną na Rysunku 2.

Rysunek 2. Struktura skompilowanego projektu z Rysunku 1

Warto tu zauważyć, że „shared” faktycznie jest współdzielone. Gdybyśmy podobne operacje wykonali na starszej wersji TypeScript, kod współdzielony byłby powielony w `client` i `server`. Innymi przykładami wykorzystania współdzielenia kodu mogą być projekty testów jednostkowych, które od teraz nie musiałyby kompilować własnej wersji kodu, który testują. Do tego będzie można teraz w TypeScript tworzyć bezproblemowo tzw. monorepo, czyli wiele projektów wykorzystujących wspólne node_modules (biblioteki zewnętrzne) i zwykle mające powiązania między sobą. Twórcy nowej funkcjonalności wskazują, że będzie ona działać bezproblemowo z obecnymi JavaScriptowymi rozwiązaniami do zarządzania wieloma projektami, takimi jak Lerna czy Yarn Workspaces. Aby zobaczyć przykład wykorzystujący Lernę, warto zajrzeć do repozytorium prowadzonego przez jednego z developerów TypeScriptu:

https://github.com/RyanCavanaugh/learn-a. Jest to jednak zbyt duża funkcjonalność, by móc przedstawić ją pokrótce w artykule wraz z resztą nowości. Jednocześnie jest to także zbyt młoda funkcjonalność, by móc już teraz ocenić, jak wpłynie na projekty i pracę programistów.

Podsumowując…

Jak sami mogliśmy zobaczyć, zmian nie było dużo. Pozornie mogą nie wydawać się rewolucyjne. Te, które zostały opisane w artykule, w dużej mierze dotyczyły rozbudowania możliwości nadawania typów, aby pokryć statycznymi typami coraz więcej przypadków spotykanych w JavaScript. Wbrew pozorom, jest to bardzo istotne - JavaScript pozwala na bardzo wiele w kwestii typów, a ideą TypeScriptu jest jawne ich określenie. Nie pojawiły się nowości składniowe, a jedyną większą zmianą niezwiązaną z typami jest wprowadzenie referencji do projektów. Na ile te zmiany są znaczące w praktyce i ułatwią życie programistom, przekonamy się zapewne w niedługim czasie, wraz z przechodzeniem coraz to kolejnych projektów na nową wersję i coraz szersze stosowanie jej możliwości.

Bibliografia

[1] https://en.wikipedia.org/wiki/TypeScript
[2] https://github.com/Microsoft/TypeScript/wiki/What%27s-new-in-TypeScript#typescript-30
[3] https://github.com/Microsoft/TypeScript/wiki/Breaking-Changes#typescript-30