Lekcja 14: Interfejsy

Podczas pracy z typami w pewnym momencie natkniesz się na interfejsy. Kiedy klasy zostały wprowadzone do języka TypeScript, pojawiły się interfejsy, przeniesione z popularnych w tamtych czasach zorientowanych obiektowo języków programowania. Historycznie rzecz biorąc, klasy i interfejsy tworzą silny związek, a interfejsy opisują plan klasy: informacje strukturalne, które muszą zostać zaimplementowane przez odpowiednią klasę. Tutaj jest wprowadzane słowo kluczowe implements.

Brzmi to bardzo podobnie do relacji między niestandardowymi typami obiektów a obiektami. W rzeczywistości, wraz z ewolucją TypeScript, interfejsy stały się prawie nie do odróżnienia od niestandardowych typów obiektów. Jednak nadal istnieje kilka subtelnych różnic.

Opisywanie interfejsów

Załóżmy, że współpracujemy z innym zespołem, który pracuje nad inną częścią tego samego sklepu internetowego. Piszą własny kod, ale używają tych samych struktur danych. I wolą interfejsy. Typ Article w naszej bazie kodów staje się interfejsem ShopItem w ich bazie kodu.

```
// Our Article type

type Article = {

title: string,

price: number,

vat: number,

stock?: number,

description?: string

}

// Our friend’s ShopItem

interface ShopItem {

title: string;

price: number;

vat: number;

stock?: number;

description?: string;

} // And yes, the semicolons are optional
```

Różnice składniowe są tak subtelne, że zostanie ci wybaczone, że prawie ich nie zauważysz. Oczywiście zarówno Article, jak i ShopItem są kompatybilne, ponieważ ich kształt - ich struktura - jest taki sam:

```
const discount = new Discount(true, 0.2)

const shopItem: ShopItem = {

title: 'Inclusive components',
```

```
price: 30,
vat: 0.2
}
// Discount.apply is typed to take `Article`
// but also takes a `ShopItem`
discount.apply(shopItem)
```

Jeśli używasz klas, mogą to być zarówno interfejsy, jak i typy wdrożone:

```
// Implementing Interfaces
class DVD implements ShopItem {
title: string
price: number
vat: number
constructor(title: string) {
this.title = title
this.price = 9.99
this.vat = 0.2
}
}
// Implementing Types
class Book implements Article {
title: string
price: number
vat: number
constructor(title: string) {
this.title = title
this.price = 39
this.vat = 0.2
}
}
```

Dzięki temu upewniamy się, że klasa, którą tworzymy, przylega do pożądanego kształtu. Jeśli przegapimy wymagane właściwości, TypeScript ostrzega nas:

```
// Nope, we missed the property `title`!

class Book implements Article {

price: number

vat: number

constructor() {

this.price = 39

this.vat = 0.2

}

}
```

Oczywiście obiekty typu Book i DVD mają taki sam kształt jak Article lub ShopItem, więc nasza klasa Rabat również na nich działa:

```
let book = new Book('Art Direction on the Web')

discount.apply(book)

let dvd = new DVD('Contagion')

discount.apply(dvd)
```

Rabaty wszędzie!

Łączenie deklaracji

Na pierwszy rzut oka interfejsy i typy wydają się być całkowicie takie same. Pamiętaj o tym na później, kiedy zobaczymy zaawansowane techniki z typami obiektów. W dowolnym momencie można podstawiać interfejsy dla typów obiektów. Poza historycznymi, gdzie są różnice? Oprócz pewnych niuansów, największą różnicą jest łączenie deklaracji. Łączenie deklaracji dla interfejsów oznacza, że możemy zadeklarować interfejs w oddzielnych pozycjach w tym samym pliku, z różnymi właściwościami, a TypeScript łączy wszystkie deklaracje i łączy je w jedną. Możemy wziąć naszą deklarację ShopItem wcześniej i rozszerzyć ją o szereg recenzji z zupełnie innej pozycji:

```
interface ShopItem {

reviews: {

rating: number,

content: string

}[]

}
```

Dodanie tych kilku linii przerwie całe użycie ShopItem w naszym pliku, ponieważ właściwość reviews nie jest opcjonalna, a DVD i inne elementy będą musiały ją zaimplementować. Łączenie deklaracji w jeden plik, o ile jest to możliwe, może wydawać się trochę bezsensowne. Czy nie jest bardziej czytelne i zrozumiałe, jeśli mamy wszystkie właściwości w jednej deklaracji? Oczywiście, że jest! Ale jest

specjalny przypadek użycia, w którym scalanie deklaracji ma dużo sensu. Wróćmy do części 1, kiedy zdefiniowaliśmy zmienną globalną w pliku definicji typu otoczenia.

Pisząc JavaScript często napotykamy sytuacje, w których zmienne, funkcje lub klasy z zewnątrz stają się dostępne globalnie. Nie tylko flaga isDevelopment, ale może także skrypt analityczny, który pozwala uzyskać statystyki dotyczące wykorzystania witryny. Lub interfejs API YouTube, który umożliwia dołączanie i odtwarzanie różnych filmów z YouTube. Wszystkie te rzeczy zwykle wiszą na obiekcie globalnym. W przeglądarkach jest to obiekt okna. Obiekt okna jest bardzo mocno zdefiniowany za pomocą interfejsu Window. Czy nie byłoby wspaniale, gdybyśmy mogli rozszerzyć Window z dowolnego miejsca w naszym kodzie, udostępniając globalne flagi, interfejsy API i funkcje w dowolnym miejscu? To tylko kilka wierszy kodu:

```
declare global {

interface Window {

isDevelopment: boolean

}

}
```

Najpierw otwieramy przestrzeń nazw global. Przestrzenie nazw to cecha, która pojawiła się przed czasami odpowiedniego hermetyzacji modułów. Są one najczęściej używane, gdy chcemy rozłożyć deklaracje typu na pliki, takie jak wszystkie deklaracje, które są globalnie dostępne (window, document, navigator itd.). Deklaracje przestrzeni nazw można również scalać. Następnie otwieramy interfejs Window. Nie nadpisujemy całego typu; dołączamy do niego własne pole: isDevelopment typu boolean. Dzięki tej deklaracji w dowolnym miejscu naszego kodu możemy od razu sprawdzić, czy jesteśmy w trybie programistycznym:

```
class Discount {

...

apply(article: Article) {

...

// Here we check if we are in dev mode

if(window.isDevelopment) {

console.log('Another discount applied')

}

}

}
```

Interfejs Window jest zwykle bardzo świadomy bieżącego stanu przeglądarek, które obsługują określony target kompilacji w pliku tsconfig.json. Oznacza to, że możliwe jest, że w Twojej przeglądarce są dostępne funkcje, które nie mają jeszcze typów w języku TypeScript, więc mogą generować błędy. Dzieje się tak, ponieważ TypeScript wybiera najniższy, wspólny mianownik w Window. Jeśli chcesz korzystać z nowszych funkcji, które nie są dostępne od razu, i odpowiednio przetestować, czy istnieją, możesz użyć powyższej techniki, aby dodać odpowiednie typy.

Podsumowanie

Wiele omówiliśmy w ostatnich lekcjach. Mimo że TypeScript to tylko niewielka warstwa na wierzchu JavaScript, sam system czcionek jest tak różnorodny i elastyczny, że możemy z nim zrobić mnóstwo rzeczy. A to dopiero początek! Podsumujmy:

1. Dowiedzieliśmy się, jak działają adnotacje typów i jak możemy wymazać je do JavaScript za pomocą kompilatora TypeScript. Okazuje się, że TypeScript nie tylko dodaje warstwę tekstową, ale także kompiluje do różnych wersji ECMAScript.

2. Widzieliśmy, że TypeScript ma swoje własne prymitywne typy, takie jak każdy: symbol wieloznaczny, który pozwala oszukać bezpieczeństwo typów, ale zapewnia, że nie są blokowane przez typy, gdy wiesz lepiej.

3. To sprawia, że TypeScript jest stopniowym systemem typów. Używaj typów, kiedy masz na to ochotę i kiedy widzisz korzyści, a nie wtedy, gdy narzędzie ci to nakazuje.

4.Dowiedzieliśmy się o ochronie typów, przepływie sterowania i zawężaniu. Ponieważ funkcje JavaScript mogą przyjmować argumenty dowolnego typu, możemy sprawdzać typ w czasie wykonywania za pomocą typeof i wywnioskować z niego nowe typy.

5. Dowiedzieliśmy się, jak typować obiekty i jakie są kluczowe aspekty strukturalnego systemu typów.

6. Mamy też mnóstwo narzędzi ułatwiających tworzenie typów. Uczyń je w ruchu, pisząc zwykły JavaScript.

7. Zagłębiliśmy się także w kilka funkcji obiektowych, takich jak klasy. Klasy są różne, ponieważ przyczyniają się zarówno do tworzenia wartości, jak i tworzenia typów.

8. Zakończyliśmy poznając interfejsy (starsze rodzeństwo typów obiektów, wychowane inaczej, ale ostatecznie okazało się prawie nie do odróżnienia od swojego młodszego odpowiednika). Jedyną cechą, na którą warto zwrócić uwagę, jest scalanie deklaracji, co pozwala nam rozszerzać interfejsy, gdy widzimy taką potrzebę.

To jest czysta podstawa pracy z typami. Reszta tekstu skupi się na uzyskaniu najlepszych typów, najlepszych wrażeń z edytora i najbardziej niezawodnego, a jednocześnie elastycznego bezpieczeństwa typów dla twoich projektów.