Lekcja 12: Pas narzędziowy typu obiektu

Pisząc TypeScript, dużo pracujemy z typami obiektów. Prawie wszystko w JavaScript jest albo funkcją, albo obiektem, więc warto poświęcić im trochę czasu! W tej lekcji przyjrzymy się kilku dodatkom, które mogą Ci pomóc podczas pracy z typami obiektów.

typeof

Typy obiektów mogą być bardzo długie. Czasami pracujemy ze strukturami danych, które są głęboko zagnieżdżone i mają mnóstwo właściwości. Spójrz na obiekt, który definiuje domyślne zamówienie w naszym sklepie internetowym:

```
const defaultOrder = {
articles: [
{
price: 1200.50,
vat: 0.2,
title: 'Macbook Air Refurbished - 2013'
},
{
price: 9,
vat: 0,
title: 'I feel smashing subscription'
}
],
customer: {
name: 'Fritz Furball',
address: {
city: 'Smashing Hill',
zip: '90210',
street: 'Whisker-ia Lane',
number: '1337'
},
dateOfBirth: new Date(2006, 9, 1)
}
}
```

Ten obiekt jest nieco skomplikowany! Moglibyśmy zdefiniować typ na jednym posiedzeniu:

```
type Order = {
articles: {
price: number,
vat: number,
title: number
}[],
customer: {
name: string,
address: {
city: string,
zip: string,
street: string,
number: string
},
dateOfBirth: Date
}
}
```

Lub moglibyśmy stworzyć wiele mniejszych typów:

```
type ArticleStub = {
price: number,
vat: number,
title: string
}
type Address = {
city: string,
zip: string,
street: string,
number: string,
}
type Customer = {
```

```
name: string,

address: Address,

dateOfBirth: date

}

type Order = {

articles: ArticleStub[],

customer: Customer

}
```

Lub połączenie obu. W obu przypadkach albo utrzymujemy wiele typów, albo tworzymy nieporęczne typy. Chcieliśmy tylko uzyskać szybki typ struktury danych, aby mieć lepsze autouzupełnianie i bezpieczeństwo typów w naszych metodach. Pamiętasz operatora typeof, którego spotkaliśmy w Lekcji 10? Dzięki operatorowi typeof mogliśmy sprawdzać typy w czasie wykonywania. W systemie czcionek TypeScript operator typeof pobiera dowolny obiekt (lub funkcję lub stałą) i wyodrębnia jego kształt:

```
type Order = typeof defaultOrder
```

To daje nam typ, którego możemy użyć w dowolnym miejscu w naszym kodzie:

```
/**

* Checks if all our orders have articles

*/

function checkOrders(orders: Order[]) {

let valid = true;

for(let order of orders) {

valid = valid && order.articles.length > 0

}

return valid
```

W momencie aktualizacji obiektu defaultOrder aktualizowany jest również typ Order!

Właściwości opcjonalne

W poprzednim przykładzie użyliśmy formy Article, która pomija dwie właściwości, które pierwotnie zdefiniowaliśmy: stock i description. To już drugi raz, kiedy nie korzystaliśmy z obu właściwości. Pamiętasz createArticleElement? Wygląda na to, że częściej potrzebujemy prostego typu Article, niż nam się wydaje. Więc co powinniśmy zrobić? Utworzyć dwa typy, Article i ArticleStub? Czy ustawić wartości fikcyjne dla właściwości, które nie są potrzebne? Czy celowo ustawić właściwości na niezdefiniowane?

Każde z tych brzmi trochę podejrzanie i niezbyt przypomina JavaScript. Podobnie jak sposoby na zaspokojenie systemu typów, który generuje tylko więcej kodu. A TypeScript nie powinien taki być. Nie powinno ci przeszkadzać. Nie powinno cię to zmuszać do dbania o typy - powinieneś się tym

przejmować, ponieważ chcesz się tym przejmować. Najlepszym sposobem byłoby dostosowanie oryginalnego typu Article i ustawienie dwóch opcjonalnych właściwości:

```
type Article = {
title: string,
price: number,
vat: number,
stock?: number,
description?: string
}
```

Znak zapytania po nazwie właściwości deklaruje, że właściwość jest opcjonalna. Co to oznacza, kiedy kodujemy? Cóż, parametry opcjonalne są… opcjonalne. Oznacza to, że mogą być dostępne, ale może też ich brakować. Musimy sprawdzić, czy są dostępne:

```
function isArticleInStock(article: Article) {
// this check is necessary to make sure
// the optional property exists
if(article.stock) {
return article.stock > 0
}
return false
}
```

Nasz typ staje się znacznie bardziej elastyczny i może być używany w wielu innych scenariuszach.

Eksportowanie i importowanie typów

Teraz, gdy nasz typ Article jest tak elastyczny, chcemy udostępnić go innym częściom naszej aplikacji. Podczas pracy z komentarzami JSDoc czasami importowaliśmy typy. Możemy zrobić to samo, pisząc czysty TypeScript. Najpierw udostępniamy typ, eksportując go:

```
export type Article = {
title: string,
price: number,
vat: number,
stock?: number,
description?: string
}
```

Następnie importujemy Article używając tej samej funkcji, której używaliśmy w Części 1. Ale tym razem importujemy typy poprzez zwykłe importy ECMAScript:

```
import { Article } from './example-two'

const book: Article = {

price: 29,

vat: 0.2,

title: 'Another book by Smashing Books'

}
```

Podobnie jak wszystkie adnotacje typu, jest on usuwany podczas kompilacji. Ta sama składnia importu jest używana do importowania innych elementów (obiektów, funkcji) również z pliku example-two. Jeśli interesują nas tylko typy, używamy niewielkiej odmiany zwykłego importu ECMAScript: import typów.

```
import type { Article } from './example-two'

const book: Article = {

price: 29,

vat: 0.2,

title: 'Another book by Smashing Books'

}
```

Jest to szczególnie przydatne, gdy masz do czynienia z wieloma importami z określonego pliku - zarówno typów, jak i innych elementów - i chcesz oddzielić informacje o typie od reszty aplikacji.