Lekcja 11: Typowanie obiektów

W poprzednich przykładach dużo pracowaliśmy z typami prymitywnymi: łańcuchami, liczbami, wartościami logicznymi. Dowiedzieliśmy się również o dwóch prymitywnych topowych typach, które nie są dostępne w JavaScript, ale występują wyłącznie w systemie czcionek TypeScript i TypeScript: any i unknown. any jest zarówno beztroski, jak i nieostrożny, oddając bezpieczeństwo typu w ręce programisty; unknown wymaga dużo większej ostrożności i troski.

Typy złożone

Zarówno any, jak i unknown są najpopularniejszymi typami, które obejmują cały zestaw innych typów istniejących w JavaScript - takich jak typy złożone. Typy złożone są interesujące. Mogą to być praktycznie dowolne kombinacje nazw właściwości i innych typów, zarówno typów pierwotnych, jak i dodatkowych typów złożonych. To sprawia, że całkowita przestrzeń możliwych typów jest praktycznie nieograniczona. Obiekty są typami złożonymi. Weź na przykład ten artykuł z naszego sklepu internetowego:

```
const book = {

title: 'Form Design Patterns by Adam Silver',

price: 32.77,

vat: 0.19,

stock: 1000,

description: 'A practical book on accessibility and forms'

}
```

Aby zdefiniować typ dla tego obiektu, możemy użyć składni aliasu typu:

```
type Article = {

title: string,

price: number,

vat: number,

stock: number,

description: string

}
```

W ten sposób opisaliśmy właśnie kształt obiektu książki, który utworzyliśmy wcześniej. Obowiązują te same zasady dotyczące pisania leworęcznego i praworęcznego, jak w przypadku podstawowych:

```
const movie: Article = {

title: 'Helvetica',

price: 6.66,

vat: 0.19,

stock: 1000,
```

description: '90 minutes of gushing about Helvetica'

}

W tym miejscu opisujemy typ movie, który jest sprawdzany przy przypisywaniu wartości. Brak odpowiednich właściwości lub ich całkowity brak spowodowałby uszkodzenie:

// Property 'description' is missing

const movie: Article = {

title: 'Helvetica',

price: 6.66,

vat: 0.19,

stock: 1000,

}

Typy strukturalne i kontrole nadmiaru właściwości

Jeśli przypiszemy wartość z właściwościami spoza określonego typu, TypeScript wyświetli błąd:

// Property 'rating' is not allowed

const movie: Article = {

title: 'Helvetica',

price: 6.66,

vat: 0.19,

stock: 1000,

description: '90 minutes of gushing about Helvetica',

rating: 5

}

Jednak tak się nie dzieje, gdy definiujemy wartość w innym miejscu:

// Property 'rating' is not allowed

const movBackup = {

title: 'Helvetica',

price: 6.66,

vat: 0.19,

stock: 1000,

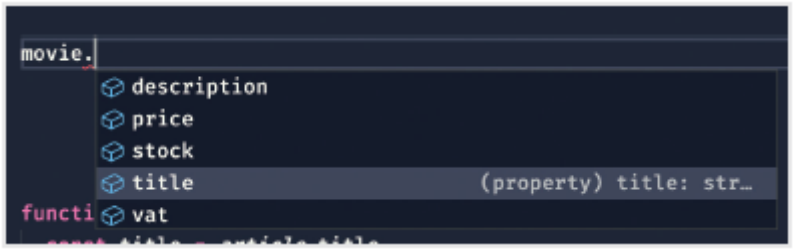
description: '90 minutes of gushing about Helvetica',

rating: 5

}

const movie: Article = movBackup // Totally OK!

Dlaczego? TypeScript to strukturalny system typów. Oznacza to, że tak długo, jak zdefiniowane właściwości typu są dostępne w obiekcie, kontrakt strukturalny jest spełniony. Jabłko ma kształt kuli. Nie jest to doskonała kula. Jest łodyga i wszędzie są nierówności, ale to wciąż kula. Przy przypisywaniu movBackup do filmu typu Article, wszystkie odpowiednie właściwości są zgodne: title, prie, vat, stok i descripts. Obca - lub nadmierna - właściwość ratingy jest zamiatana pod dywan. Dosłownie! Jeśli spojrzymy na funkcje autouzupełniania, które daje nam VS Code, gdy tylko przypiszemy movBackup do filmu, zobaczymy, że ocena nie jest już dostępna:



Kontrakt strukturalny jest spełniony. Wszystkie nadmiarowe właściwości nie są już dostępne.

Nie oznacza to, że tych właściwości nie ma w czasie wykonywania. One są! Ale podczas programowania nasze środowisko narzędziowe upewni się, że używamy tylko właściwości, które są zdefiniowane przez typ. TypeScript jest dla nas bardzo miły. Moglibyśmy uzyskać wartości z dowolnego miejsca, a te wartości mogą się zmieniać w czasie, ale nasza umowa nadal dba tylko o odpowiednie typy określonego zestawu właściwości. To sprawia, że nasza aplikacja jest nadal ważna i bezpieczna dla typów, ale pozwala nam być elastycznym w innych częściach naszej aplikacji. Dzieje się tak również wtedy, gdy mamy dwa różne typy o podobnej strukturze wystarczającej do realizacji zamówienia:

type ShopItem = {

title: string,

price: number,

vat: number,

stock: number,

description: string,

rating: number

}

const shopitem = {

title: 'Helvetica',

price: 6.66,

vat: 0.19,

stock: 1000,

description: '90 minutes of gushing about Helvetica',

```
rating: 5

}

const movie: Article = shopitem // Totally OK!
```

Ale dlaczego bezpośrednie przypisanie wartości po adnotacji typu powoduje błąd?

```
// Property 'rating' is not allowed

const movie: Article = {

title: 'Helvetica',

price: 6.66,

vat: 0.19,

stock: 1000,

description: '90 minutes of gushing about Helvetica',

rating : 5

}
```

Ta funkcja nosi nazwę kontroli nadmiaru właściwości. Ponieważ TypeScript jest dla nas miły, ponieważ struktury mogą się zmieniać w naszej aplikacji, wskaże nam rzeczy, które mogą być celowymi błędami. Przypisanie wartości tuż po adnotacji typu, która nie jest całkowicie zgodna, jest najprawdopodobniej niezamierzonym błędem. Dlaczego mielibyśmy dodawać adnotacje do określonego typu, a następnie przypisywać coś innego? Oczywiście posiadanie zbyt małej liczby właściwości w naszej wartości powoduje w każdym przypadku błąd:

```
const missingProperties = {

title: 'Helvetica',

price: 6.66

}

// Boom! This breaks

const anotherMovie: Article = missingProperties
```

Kontrakt strukturalny nie jest spełniony.

Obiekty jako parametry

Możemy również użyć naszych niestandardowych zdefiniowanych typów jako parametrów w funkcjach:

```
function createArticleElement(article: Article): string {

const title = article.title

const price = addVAT(article.price, article.vat)

return `<h2>Buy ${title} for ${price}</h2>`
```

}

I możemy przekazywać parametry bez wyraźnej adnotacji typu. Ponieważ kontrakt strukturalny jest spełniony, TypeScript będzie zadowolony:

```
const shopitem = {

title: 'Helvetica',

price: 6.66,

vat: 0.19,

stock: 1000,

description: '90 minutes of gushing about Helvetica',

rating: 5

}

createArticleElement(shopItem) // Totally OK!
```

Oznacza to również, że możemy być bardzo celowi z typem, który chcemy dla funkcji createArticleElement, i być może wprowadzić typ obiektu inline z tylko oczekiwanymi właściwościami:

```
function createArticleElement(

article: { title: string, price: number, vat: number }): string {

const title = article.title

const price = addVAT(article.price, article.vat)

return `<h2>Buy ${title} for ${price}</h2>`

}
```

Przekazywanie elementów typu Article nadal będzie działać:

```
const movie: Article = {

title: 'Helvetica',

price: 6.66,

vat: 0.19,

stock: 1000,

description: '90 minutes of gushing about Helvetica'

}

createArticleElement(movie)
```

Kontrakt strukturalny jest nadal wykonany. Ale tak jak w przypadku bezpośredniego przypisywania wartości, przekazanie obiektu ze zbyt dużą liczbą właściwości bezpośrednio do funkcji spowoduje nadmierne sprawdzanie właściwości:

```
createArticleElement({

title: 'Design Systems by Alla Kholmatova',

price: 20,

vat: 0.19,

rating: 5

}) // Boom! rating is one property too many
```

Typowanie strukturalne bardzo różni się od innych języków programowania, ale pasuje do JavaScript i tego, jak dobrze z nim pracujemy. Przekazywanie danych, które ewoluują - uzyskiwanie większej liczby właściwości niż wcześniej, a może nawet utrata niektórych właściwości - jest bardzo powszechne. Jedyne, o co powinniśmy dbać, to dostępność wszystkich potrzebnych nam nieruchomości. To jest umowa dotycząca naszych obiektów i funkcji.