Lekcja 36: Jeśli This, Then That

Weź pod uwagę następującą strukturę danych, którą utworzyliśmy dla naszego sklepu internetowego. Mamy klientów, produkty i zamówienia. Klienci mają identyfikator, imię i nazwisko.

type Customer = {

customerId: number,

firstName: string,

lastName: string

}

const customer = {

id: 1,

firstName: ‘Stefan’,

lastName: ‘Baumgartner’

} // = type Customer

Produkt ma identyfikator produktu, tytuł i cenę.

type Product = {

productId: number,

title: string,

price: number

}

const product = {

id: 22,

title: ‘Form Design Patterns’,

price: 29

}

Zamówienie ma również identyfikator, klienta (typu Customer), listę produktów w zamówieniu oraz datę.

type Order = {

orderId: number,

customer: Customer,

products: Product[],

date: Date

}

To znacznie uproszczony, ale solidny początek naszej małej aplikacji. Wdrażamy interfejs administracyjny dla aplikacji e-commerce. Chcemy udostępnić funkcję fetchOrder, która działa w następujący sposób:

1. Jeśli przekazujemy klienta, otrzymujemy listę zamówień od tego klienta.

2. Jeśli przekażemy produkt, otrzymamy listę zamówień, które zawierają ten produkt.

3. Jeśli przekażemy identyfikator zamówienia, otrzymamy to konkretne zamówienie.

Naszym pierwszym pomysłem na wdrożenie tego byłoby przeciążenie funkcji.

function fetchOrder(customer: Customer): Order[]

function fetchOrder(product: Product): Order[]

unction fetchOrder(orderId: number): Order

function fetchOrder(param: any): any {

// Implementation to follow

}

Działa to dobrze w prostych przypadkach, w których mamy absolutną pewność, jakich parametrów oczekujemy:

fetchOrder(customer) // It's Order[]

fetchOrder(2) // It's Order

Ale robi się włochaty, gdy nasz wkład jest niejednoznaczny. Kiedy przekazujemy argument, który może być Customer lub number, wynik jest nieco nudny:

declare const ambiguous: Customer | number

fetchOrder(ambiguous) // It's any

Oczywiście moglibyśmy załatać typy funkcji implementacji, aby były nieco jaśniejsze

function fetchOrder(customer: Customer): Order[]

function fetchOrder(product: Product): Order[]

function fetchOrder(orderId: number): Order

function fetchOrder(

param: Customer | Product | number

): Order[] | Order {

// Implementation to follow

}

Ale wyraźne określanie wszystkich możliwych wyników jest bardzo szczegółowe bardzo szybko:

function fetchOrder(customer: Customer): Order[]

```
function fetchOrder(product: Product): Order[]

function fetchOrder(orderId: number): Order

function fetchOrder(

param: Customer | Product

): Order[]

function fetchOrder(

param: Customer | number

): Order[] | Order

function fetchOrder(

param: Product | number

): Order[] | Order

function fetchOrder(

param: Customer | Product | number

): Order[] | Order {

// I hope I didn't forget anything

}
```

Siedem przeciążeń dla trzech możliwych typów wejść i dwa możliwe typy wyjść. Teraz dodaj kolejny, to wyczerpujące!

Wprowadź typy warunkowe

To musi być łatwiejsze. Możemy zmapować każdy typ wejścia na typ wyjściowy

- Jeśli typ wejściowy to Customer, typ zwrotu to Order[]
- Jeśli typ wejściowy to Product, typ zwrotu to Order[]
- Jeśli typ wejściowy to number, typ zwracany to Order

Jeśli typ danych wejściowych jest kombinacją dostępnych typów danych wejściowych, typy zwracane są połączeniem odpowiednich typów danych wyjściowych. Możemy modelować to zachowanie za pomocą typów warunkowych. Składnia typów warunkowych jest oparta na typach ogólnych i wygląda następująco:

```
type Conditional<T> = T extends U ? A : B
```

Gdzie T jest parametrem typu ogólnego. U, A i B to inne typy. Możemy odczytać tę instrukcję jak operacje trójskładnikowe w JavaScript:

```
const x = (t > 0.5) ? true : false
```

Powyższe stwierdzenie brzmi, że jeśli t jest większe niż 0,5, to x jest prawdą, w przeciwnym razie jest fałszem. Możemy odczytać instrukcję typu warunkowego w ten sam sposób: jeśli typ T rozszerza typ

U, przypisany typ to A, w przeciwnym razie jest to B. Zobaczmy, jak to działa z funkcją fetchOrder. Najpierw tworzymy typ dla wszystkich możliwych danych wejściowych.

type FetchParams = number

| Customer

| Product;

Następnie tworzymy typ ogólny FetchReturn <T> z ogólnym ograniczeniem FetchParams.

type FetchReturn<Param extends FetchParams> =

Param extends Customer ? Order[] :

Param extends Product ? Order[] : Order

Ograniczenie typu <Param extends FetchParams> już ogranicza dostępne typy danych wejściowych do trzech możliwych typów, więc ten warunek jest już zaznaczony. Warunek następnie brzmi:

1. Jeśli typ Param rozszerza Customer, spodziewamy się tablicy Order [].

2. W przeciwnym razie, jeśli Param rozszerzy Product, spodziewamy się również tablicy Order [].

3. W przeciwnym razie, gdy zostanie tylko number, oczekujemy pojedynczego Order.

W żargonie TypeScript mówimy, że typ warunkowy jest rozpoznawany jako Order [].

Dostosujmy naszą funkcję do nowego typu warunkowego:

function fetchOrder<Param extends FetchParams>(

param: Param

): FetchReturn<Param> {

// Well, the implementation

}

To wszystko, czego potrzebujemy, aby uzyskać wymagane typy zwracane dla każdej kombinacji typów danych wejściowych.

fetchOrder(customer) // Order[] OK!

fetchOrder(product) // Order[] OK!

fetchOrder(2) // Order[] OK!

fetchOrder(ambiguous) // Order | Order[]

declare x: any

// any is not part of `FetchParams`

fetchOrder(x)

Typy warunkowe dobrze sprawdzają się również w przypadku warstwy tekstowej otaczającej zwykły JavaScript. Działają tylko w warstwie tekstowej i można je łatwo usunąć, a jednocześnie mogą opisywać wszystkie możliwe wyniki funkcji.