Lekcja 29: Nie wiem, czego chcę, ale wiem, jak to zdobyć

Niesławna linia Sex Pistols doskonale opisuje punk z lat 70., małe dzieci w napadach złości i leki generyczne. Weź pod uwagę następującą strukturę danych w przypadku filmu wideo, który istnieje w różnych formatach:

type VideoFormatURLs = {

format360p: URL,

format480p: URL,

format720p: URL,

format1080p: URL

}

URL to klasa adresów URL wbudowana w przeglądarkę. Chcemy zapewnić API, w którym programiści mogą załadować określony format (używając instrukcji declare dla zwięzłości):

declare const videos: VideoFormatURLs

declare function loadFormat(

format: string

): void

Aby upewnić się, że format przychodzący jest prawidłowym kluczem w naszej strukturze danych, tworzymy funkcję narzędziową z predykatem typu

function isFormatVailable(

obj: VideoFormatURLs,

key: string

): key is keyof VideoFormatURLs {

return key in obj

}

Funkcja działa zgodnie z przeznaczeniem, a w naszej przestrzeni typów możemy zawęzić zestaw wszystkich ciągów do samych kluczy VideoFormatURL:

if(isFormatAvailable(videos, format)) {

// format is now “format360p” | “format480p” |

// “format720p” | “format1080p”

// and index accessing perfectly works:

videos[format]

}

Teraz mamy podobną sytuację z ładowaniem napisów. Oto nasza struktura danych napisów:

```
type SubtitleURLs = {

english: URL,

german: URL,

french: URL

}
```

A to jest funkcja walidacji, która sprawdza, czy określony klucz jest dostępny w naszym obiekcie napisów:

```
function isSubtitleAvailable(

obj: SubtitleURLs,

key: string

): key is keyof SubtitleURLs {

return key in obj

}
```

Poczekaj minutę. To jest dokładnie ta sama funkcja! W JavaScript nie stworzylibyśmy dwóch z nich, ponieważ służą dokładnie temu samemu celowi, a nawet mają dokładnie taką samą implementację! Ale potrzebujemy dwóch implementacji, ponieważ chcemy mieć bezpieczeństwo typów, prawda? Cóż, oto zasada, według której należy postępować: jeśli zrobimy coś w języku TypeScript, czego nie chcielibyśmy zrobić w JavaScript, powinniśmy to przemyśleć. TypeScript został zaprojektowany, aby zapewnić bezpieczeństwo typów dla prawie wszystkich scenariuszy JavaScript. Prawidłowe wpisanie funkcji narzędzia jest zdecydowanie jedną z nich.

Wprowadź Ogólne

Cofnijmy się i zdefiniujmy funkcję narzędzia w taki sam sposób, jak w JavaScript, bez żadnych typów:

```
function isAvailable(

obj, key

) {

return key in obj

}
```

Teraz chcemy przygotować naszą funkcję na nieznane typy i nadal udzielić poprawnej odpowiedzi. W tym miejscu pojawiają się ogólne.

Programowanie ogólne to styl programowania komputerowego, w którym algorytmy są zapisywane w kategoriach typów, które mają zostać określone później, a następnie są tworzone w razie potrzeby dla określonych typów podanych jako parametry. (*Wikipedia*)

Ta definicja zawiera kilka kluczowych informacji: zamiast pracować z określonym typem, pracujemy z parametrem, który jest następnie zastępowany przez określony typ. Parametry typu są umieszczane w nawiasach ostrych na nagłówkach funkcji lub w deklaracjach klas. Dodajmy jeden do naszej funkcji isAvailable.

```
function isAvailable<Formats>(

obj, key

) {

return key in obj

}
```

Typ Formats nie istnieją w naszych deklaracjach typów, ale są parametrem, który jest zastępowany prawdziwym, takim jak VideoFormatURL lub SubtitleURLs. Możemy jednak użyć tego parametru typu ze zwykłymi typami w naszej deklaracji funkcji:

```
function isAvailable<Formats>(

obj: Formats, key

): key is keyof Formats {

return key in obj

}
```

Jeśli chcemy wpisać kley - który jest teraz niejawnie any - musielibyśmy użyć szerszego zestawu możliwych typów kluczy:

```
function isAvailable<Formats>(

obj: Formats,

key: string | number | symbol

): key is keyof Formats {

return key in obj

}
```

Dzieje się tak, ponieważ nasz ogólny typ Formats nie wiedzą, że mogą mieć tylko klucze w postaci łańcuchów; musi przygotować się na wszystkie możliwe klucze. W JavaScript liczby i symbole są prawidłowymi typami kluczy. Weźmy na przykład tablicę; tablice można postrzegać jako obiekty z klawiszami numerycznymi.

Adnotacje ogólne i wnioskowanie ogólne

Teraz, gdy zdefiniowaliśmy naszą funkcję jako funkcję ogólną, użyjmy jej. Mamy dwa różne sposoby korzystania z funkcji ogólnej. Po pierwsze, możemy jawnie oznaczyć typ, który chcemy zastąpić:

```
if(isFormatAvailable<VideoFormatURLs>(videos, format))

{

// ...

}
```

Podobnie jak jawne adnotacje typu w innych miejscach, TypeScript przyjmuje to jako dane i weryfikuje wszystko inne względem tego typu. Oznacza to, że w momencie, gdy określimy VideoFormatURLs jako

substytut typu parametru, musimy upewnić się, że argument obj, który przekazujemy do funkcji, jest zgodny z typem VideoFormatURLs. Jednak o wiele bardziej interesujące i wydajniejsze jest, gdy używamy wnioskowania o typie do podstawiania naszego parametru typu. TypeScript jest w stanie wywnioskować parametr typu na podstawie rzeczywistych argumentów przekazywanych do funkcji, co wydaje się znacznie bardziej naturalne:

```
// An object with video formats

declare const videoFormats: VideoFormatURLs

if(isAvailable(videoFormats, format)) {

// Inferred type 'VideoFormatURLs'

// format is now keyof VideoFormatURLs

}

// An object with video formats

declare const subtitles: SubtitleURLs

if(isAvailable(subtitles, language)) {

// Inferred type 'SubtitleURLs'

// language is now keyof SubtitleURls

}
```

To tylko pisanie JavaScript.

Typy ogólne na wolności

To była nasza pierwsza, napisana samodzielnie, ogólna funkcja. Być może już spotkałeś się z niektórymi rodzajami. Promise to typ ogólny, który pojawia się w momencie pisania kodu asynchronicznego. Argument w Promise podaje typ wyniku:

```
// randomNumber returns a Promise<number>

async function randomNumber() {

return Math.random()

}
```

Kolejnym jest Array. Możemy pisać typy tablicowe za pomocą literału tablicowego:

```
let anArray: number[]
```

Lub możemy użyć generycznego

```
let anotherArray: Array <number>
```

Obie robią to samo. Może się jednak okazać, że wygodniejsze będzie użycie typu ogólnego, gdy masz do czynienia z typami unii:

```
let aMixedArray: Array<number | string | boolean>
```

Pracując szczególnie z wieloma wbudowanymi interfejsami API JavaScript lub interfejsami API przeglądarki, napotkasz typy ogólne.