Lekcja 30: Ogólne ograniczenia

Nasza ogólna funkcja jest już całkiem dobra. Możemy przekazać wszystko z szerokiej gamy dostępnych typów i możemy wskazać konkretne typy po podstawieniu. Kiedy myślimy o zbiorah, otwieramy typ any ; a kiedy już dokonamy zamiany, wybieramy dużo węższy zbiór. Może to niestety prowadzić do niepożądanych zachowań. Nasz typ isAvailable z ostatniej lekcji działa bardzo dobrze z typami obiektów, które zdefiniowaliśmy:

```
function isAvailable<FormatList>(

obj: FormatList,

key: string | number | symbol

): key is keyof FormatList {

return key in obj

}

// 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

}
```

A także ze wszystkimi innymi dostępnymi typami obiektów - nawet bez deklaracji konkretnego typu:

```
if(isAvailable({ name: 'Stefan', age: 38}, key)) {

// key is now “name” | “age”

}
```

Ale działa również ze wszystkimi typami niebędącymi obiektami:

```
if (isAvailable ('A string', 'length')) {

// Również łańcuchy mają metody,

// jak length, indexOf, ...
```

}

if (isAvailable (1337, aKey)) {

// Również liczby mają metody

// aKey to teraz wszystko, co liczba ma do zaoferowania

}

Działa również z tablicami, tak że kluczem może być cały zestaw liczb, a także funkcje tablicowe, takie jak map, forEach i tak dalej. Chociaż jest to fajne, ponieważ sprawia, że nasze typy są jeszcze bardziej kompatybilne, może prowadzić do niepożądanego zachowania, jeśli chcemy tylko sprawdzać obiekty. Na szczęście TypeScript radzi sobie z takimi sytuacjami.

Definiowanie granic

Jak wyjaśniliśmy na początku, parametry typów ogólnych obejmują cały zbiór typów: any. Zastępując określonym typem, zbiór typów staje się węższy i wyraźniejszy. Istnieje jednak możliwość zdefiniowania obwiedni lub podzbiorów przestrzeni tekstowej. To sprawia, że parametry typu ogólnego są nieco węższe, zanim zostaną zastąpione typami rzeczywistymi. Otrzymujemy informacje z góry, jeśli przekażemy obiekt, który nie powinien zostać przekazany. Aby zdefiniować podzbiory ogólne, TypeScript używa słowa kluczowego extends. Sprawdzamy, czy parametr typu ogólnego rozszerza określony podzbiór typów. Jeśli chcemy tylko przekazywać obiekty, możemy rozszerzyć z obiektu typu:

function isAvailable<FormatList extends object>(

obj: FormatList,

key: string

): key is keyof FormatList {

return key in obj

}

Za pomocą <FormatList extends object> mówimy TypeScript, że argument, który przekazujemy, musi być co najmniej obiektem. Wszystkie typy pierwotne, a nawet tablice są wykluczone. IsAvailable ('A string', 'length') Czerwone zawijasy tam, gdzie powinny.

Typy indeksów

Zdefiniujmy funkcję, która wczytuje plik, albo wideo w określonym formacie, albo napisy w określonym języku. Ponownie zaczynamy od surowej funkcji JavaScript (po prostu nagłówek dla zwięzłości):

function loadFile(fileFormats, format) {

// Implement

}

Kiedy dodajemy typy, robimy coś podobnego, jak w przypadku funkcji isAvailable:

function loadFile<Formats extends object>(

fileFormats: Formats,

```
format: string

) {

// You know

}
```

Możemy pójść jeszcze dalej. Kiedy spojrzysz na obie definicje formatów, zauważysz inną wspólną cechę.

```
type VideoFormatURLs = {

format360p: URL,

format480p: URL,

format720p: URL,

format1080p: URL

}

type SubtitleURLs = {

english: URL,

german: URL,

french: URL

}
```

Zgadza się: wszystkie właściwości są typu URL. Inny format najprawdopodobniej spowodowałby błąd w przypadku użycia z funkcją loadFiles. Potrzebowalibyśmy ograniczenia, aby zapewnić, że przekazujemy tylko zgodne obiekty, w przypadku których nie znamy samych właściwości, ale wiemy tylko, że każda właściwość jest typu URL. Typy indeksów, jak pokrótce widzieliśmy wcześniej, są do tego idealne. Poniżej znajduje się typ indeksu, z którym spotkaliśmy się wcześniej, iterujący po zbiorze unii, w tym przypadku dopuszczający dowolną wartość dla każdej właściwości:

```
type PossibleKeys = 'meetup' | 'conference'

'hackathon' | 'webinar'

type Groups = {

[k in PossibleKeys]: any

}
```

Typy indeksów nie definiują konkretnych kluczy właściwości. Po prostu definiują zestaw kluczy, po których iterują. Jako klucze możemy również zaakceptować cały zestaw ciągów.

```
type AnyObject = {

[k: string]: any

}
```

Teraz, gdy akceptujemy wszystkie klucze właściwości typu string, możemy wyraźnie powiedzieć, że typem każdej właściwości musi być URL:

```
type URLList = {

[k: string]: URL

}
```

Idealny kształt zawierający adresy URL VideoFormatURL i SubtitleURL. I właściwie każda inna lista adresów URL! Dlatego też idealne ograniczenie dla naszego parametru typu ogólnego:

```
type URLList = {

[k: string]: URL

}

function loadFile<Formats extends URLList>(

fileFormats: Formats,

format: string

) {

// The real work ahead

}
```

Dzięki temu każdy przekazany przez nas obiekt, który nie daje obiektu z adresami URL, utworzy piękne, czerwone, faliste linie w naszym edytorze.