Lekcja 34: Wiązanie typów ogólnych

Wróćmy do connectPreferences. Dużo rozmawialiśmy o tym, jakie argumenty chcemy przekazać do tej funkcji, że nie przyjrzeliśmy się temu, co zwraca nasza operacja.

```
function combinePreferences(

defaultP: UserPreferences,

userP: Partial<UserPreferences>

) {

return { ...defaultP, ...userP }

}

const prefs = combinePreferences(

defaultUP,

{ format: 'format720p' }

)
```

Po najechaniu kursorem na prefs możemy zobaczyć wynik tego, co TypeScript wywnioskuje z naszego przypisania:

```
const prefs: {

format: “format360p” | “format480p” |

“format720p” | “format1080p”;

subtitles: {

active: boolean;

language: “english” |

“german” | “french”;

};

theme: "dark" | "light";

}
```

To jest to samo, co UserPreferences i to, czego się spodziewaliśmy. Z jednym argumentem UserPreferences, a drugim Partial <UserPreferences>, kombinacja obu argumentów powinna być ponownie pełnym typem UserPreferences. Pobieranie UserPreferences w zamian z connectPreferences jest całkowicie dobrym zachowaniem i sprawi, że Twoja aplikacja będzie o wiele bardziej bezpieczna dla typów niż była. Potraktujmy to jako okazję do zbadania adnotacji typu, wnioskowania o typie i ogólnego powiązania typu oraz zobaczenia ich efektów.

Wnioskowanie o typie

Preferencje naszych użytkowników to format wideo 720p i ciemny motyw. Odpowiedni obiekt to:

```
{ format: 'format720p', theme: 'dark' }
```

Używamy tego literału jako dosłownego argumentu dla funkcji connectPreferences.

```
combinePreferences(
defaultUP,
{ format: 'format720p', theme: 'dark' }
)
```

W momencie, gdy przekazujemy literał, TypeScript wnioskuje, jaki typ naszego literału jest typem wartości. Dzieje się tak, ponieważ ta wartość, będąca argumentem funkcji, nie może się zmienić w wyniku operacji. Jedynym sposobem, w jaki możemy zmodyfikować tę wartość, jest edycja kodu źródłowego. To jest ostateczne. Kiedy przypisujemy tę wartość do zmiennej, sytuacja wygląda inaczej.

```
const userSettings = {
format: 'format720p', theme: 'dark'
}
combinePreferences(
defaultUP, userSettings
)
```

W momencie przypisania tej wartości do userSettings, TypeScript wywnioskuje swój typ na najbardziej rozsądnie najszerszy typ. W naszym przypadku string.

```
// typeof userSettings =
{
format: string,
theme: string
}
```

Ten typ jest znacznie szerszy niż to, czego oczekujemy w naszym typie UserPreferences. TypeScript wyrzuci do nas czerwone zawijasy, ponieważ nie możemy pobrać szerszego ciągu z „ciemnego” | „jasny”, ani dla wszystkich wymienionych przez nas formatów. A TypeScript ma rację! Nie ma zabezpieczenia przed zmianą wartości w pewnym momencie na coś całkowicie niezgodnego. Dziękuję, TypeScript! Jedną rzeczą, którą moglibyśmy zrobić, jest dodanie kontekstu const:

```
const userSettings = {
format: 'format720p', theme: 'dark'
} as const
```

Chroni to go przed zmianą w TypeScript i zawęża przypisanie do jego typu wartości, dzięki czemu jest zgodny z Partial <UserPreferences>, ponieważ jest to podtyp. Inną rzeczą, którą moglibyśmy zrobić, jest napisanie adnotacji typu.

Adnotacje typu

Chociaż zalecamy stosowanie jak największej ilości wnioskowania, adnotacje są magiczną rzeczą, której możemy używać, gdy nasze typy są bardzo wąskie, aby nie odpowiadać typom prymitywnym. Adnotacje typu sprawdzają typ w momencie przypisania wartości.

```
const userSettings:

Partial<UserPreferences> = {

format: 'format720p', theme: 'dark'

}
```

Z tą adnotacją typu userSettings zawsze będzie Partial <UserPreferences>, o ile przypisane przez nas wartości przejdą kontrolę typu. Jeśli tak, nigdy nie wrócimy do ich pierwotnych wartości podczas dalszego używania zmiennej. Te informacje są dla nas utracone.

Ogólne powiązanie typu

Proces podstawiania typu konkretnego do typu ogólnego nazywa się wiązaniem. Sprawdźmy, co się stanie, jeśli powiążemy parametr typu ogólnego z konkretnym typem. To jest connectPreferences z parametrem typu ogólnego.

```
function combinePreferences<

UserPref extends Partial<UserPreferences>

>(

defaultP: UserPreferences,

userP: UserPref

) {

return { ...defaultP, ...userP }

}

const prefs = combinePreferences(

defaultUP,

{ format: 'format720p', theme: 'dark' }

)
```

Kiedy wywołujemy connectPreferences z adnotowanym typem Partial <UserPreferences>, zastępujemy jego nadtyp UserPref. Zachowujemy się tak samo, jak pierwotnie. Kiedy wywołujemy connectPreferences z literałem lub zmienną w kontekście const, wiążemy typ wartości z UserPref.

1. {format: 'format720p', theme: 'dark'} jest traktowane jako dosłowne, dlatego patrzymy na typ wartości.

2. Typ wartości {format: 'format720p', theme: 'dark'} jest podtypem częściowej <UserPreferences>, więc sprawdza typ.

3. Wiążemy UserPref z {format: 'format720p', theme: 'dark'}, co oznacza, że teraz pracujemy z typem wartości zamiast Partial <UserPreferences>.

UserPref zmienił się teraz, co oznacza, że zmienił się również nasz typ wyniku. Jeśli najedziesz kursorem na prefs, otrzymamy następujące informacje o typie:

const p: {

format: “format360p” | “format480p” |

“format720p” | “format1080p”;

subtitles: {

active: boolean;

language: “english” |

“german” | “french”;

};

theme: “light” | “dark”;

} & {

format: “format720p”;

theme: “dark”;

}

Najpierw dowiadujemy się, co właściwie robi operacja {... defaultP, ... userP}. Tworzy kombinację dwóch obiektów, a wynikowy typ jest przecięciem. To ma sens! Widzimy również, czym UserPrefs stał się w momencie, gdy przekazaliśmy literał: typ wartości tego literału. To skrzyżowanie tworzy interesujące zachowanie. Mamy kilka typów unii, które są teraz przecinane z podtypami ich zbiorów. W takim scenariuszu zawsze wygrywa węższy zestaw:

('dark' | 'light') & 'dark' // type is 'dark

Co oznacza, że dokładnie wiemy, jakie wartości otrzymujemy, gdy pracujemy z prefs:

prefs.theme // jest typu „ciemny”

prefs.format // jest typu „format720p”

To sprawia, że niektóre kontrole w naszym kodzie są łatwiejsze. Uważaj jednak przy zbyt wielu typach wartości. Jeśli weźmiemy ten sam wzorzec dla domyślnych preferencji i przekażemy do niego obiekt kontekstu const, możemy uzyskać niepożądane efekty uboczne:

function combinePreferences<

Defaults extends UserPreferences,

UserPref extends Partial<UserPreferences>

>(

defaultP: Defaults,

userP: UserPref

) {

```
return { ...defaultP, ...userP }
}
const defaultUP = {
// wW know what we have here
} as const
const prefs = combinePreferences(
defaultUP,
{ format: 'format720p', theme: 'dark' }
)
```

Wynikowy typ wygląda następująco:

```
const prefs: {
readonly format: “format1080p”;
readonly subtitles: {
readonly active: false;
readonly language: “english”;
};
readonly theme: “light”;
} & {
format: “format720p”;
theme: “dark”;
}
```

Przecięcie dwóch różnych typów wartości zawsze powoduje, że nigdy, co oznacza, że zarówno motyw, jak i format stają się dla nas bezużyteczne.