Lekcja 48: Rozszerzanie obiektu, część 1

Analiza przepływu sterowania TypeScript pozwala zawęzić typ z szerszego do węższego:

```
function print(msg: any) {
if(typeof msg === 'string') {
// We know msg is a string
console.log(msg.toUpperCase()) // thumbs up!
} else if (typeof msg === 'number') {
// I know msg is a number
console.log(msg.toFixed(2)) // thumbs up!
}
}
```

To jest sprawdzanie bezpieczeństwa typów w JavaScript, a TypeScript z tego korzysta. Istnieją jednak przypadki, w których TypeScript wymaga od nas nieco więcej pomocy.

Sprawdzanie właściwości obiektu

Załóżmy, że masz obiekt JavaScript i nie wiesz, czy istnieje określona właściwość. Obiekt może być any lub unknown. W JavaScript powinieneś sprawdzić takie właściwości:

```
if(typeof obj === 'object'
&& 'prop' in obj) {
// It's safe to access obj.prop
console.assert(typeof obj.prop !== 'undefined')
// But TS doesn't know :-(
}
if(typeof obj === 'object'
&& obj.hasOwnProperty('prop')) {
// It's safe to access obj.prop
console.assert(typeof obj.prop !== 'undefined')
// But TS doesn't know :-(
}
```

W tej chwili TypeScript nie jest w stanie rozszerzyć typu obj za pomocą prop, mimo że działa to z JavaScriptem. Możemy jednak napisać małą funkcję pomocniczą, aby uzyskać poprawne wpisy:

```
function hasOwnProperty<
X extends {}, Y extends PropertyKey
```

```
>(
obj: X, prop: Y
): obj is X & Record<Y, unknown> {
return obj.hasOwnProperty(prop)
}
```

Zobaczmy, co się dzieje:

1. Nasza funkcja hasOwnProperty ma dwa rodzaje:

a. X extends {} zapewnia, że używamy tej metody tylko na obiektach.

b. Y esxtends PropertyKey upewnia się, że klucz albo string | number | symbo.PropertyKey jest typem wbudowanym.

2. Nie ma potrzeby jawnego definiowania typów ogólnych, ponieważ są one wywnioskowane przez użycie.

3. (obj: X, prop: Y): chcemy sprawdzić, czy prop jest kluczem właściwości obiektu.

4. Typ zwracany jest predykatem typu. Jeśli metoda zwróci wartość true, możemy ponownie wpisać any z naszych parametrów. W tym przypadku mówimy, że nasz obj jest oryginalnym obiektem z typem przecięcia Record <Y, unknown>. Ostatni element dodaje nowo znalezioną właściwość do obj i ustawia ją na unknown. W użyciu hasOwnProperty działa w następujący sposób:

```
// person is an object
if(typeof person === 'object'
// person = { } & Record<'name', unknown>
// = { } & { name: 'unknown'}
&& hasOwnProperty(person, 'name')
// Yes! name now exists in person
&& typeof person.name === 'string'
) {
// Do something with person.name,
// which is a string
}
```

Rozszerzanie lib.d.ts

Pisanie funkcji pomocniczej jest dokładne. Po co pisać funkcję pomocniczą, która otacza niektóre wbudowane funkcje tylko po to, aby uzyskać lepsze typy? Powinniśmy być w stanie tworzyć te typowania bezpośrednio tam, gdzie występują. Na szczęście przy deklaracji scalania interfejsów możemy są w stanie to zrobić. Utwórz własny plik deklaracji typu otoczenia i upewnij się, że kompilator TypeScript wie, gdzie je znaleźć (typeRoots i typy w tsconfig.json to dobry początek). W tym pliku, który możemy nazwać mylib.d.ts, możemy dodać własne deklaracje otoczenia i rozszerzyć istniejące

deklaracje. Możemy to zrobić za pomocą interfejsu Object. To jest wbudowany interfejs dla wszystkich Objects.

interface Object {

hasOwnProperty<

X extends {},

Y extends PropertyKey

>(this: X, prop: Y): this is X & Record<Y, unknown>

}

Jeśli uważasz, że TypeScript powinien mieć coś takiego po wyjęciu z pudełka, to masz rację. Może istnieć dobry powód, dla którego definicje typów nie są jeszcze dostarczane w ten sposób, ale dobrze, że możemy je rozszerzyć, aby zaspokoić własne potrzeby.

Rozszerzanie konstruktora obiektów

Do podobnego scenariusza dochodzimy podczas pracy z innymi częściami Object. Jednym ze wzorców, z którym możesz się często spotkać, jest iteracja po tablicy kluczy obiektów, a następnie dostęp do tych właściwości, aby zrobić coś z wartościami.

const obj = {

name: 'Stefan',

age: 38

}

Object.keys(obj).map(key => {

console.log(obj[key])

})

W trybie ścisłym TypeScript chce wyraźnie wiedzieć, jaki typ key ma, aby mieć pewność, że ten dostęp do indeksu działa. Rzucamy więc w nas czerwonymi zawijasami. Cóż, powinniśmy znać typ key! To jest keyof obj! Jest to dobra okazja, aby rozszerzyć wpisywanie dla Object.

Tak powinno zachowywać się jak Object.keys:

1. Jeśli przekażemy liczbę, zwracamy pustą tablicę.

2. Jeśli przekażemy tablicę lub łańcuch, otrzymamy w zamian tablicę ciągów. Ta tablica ciągów zawiera uszeregowane indeksy wartości wejściowej.

3. Jeśli przekażemy obiekt, otrzymamy w zamian rzeczywiste klucze tego obiektu.

Interfejs do rozszerzenia nazywa się ObjectConstructor. W przypadku klas lub struktur podobnych do klas język TypeScript wymaga dwóch różnych interfejsów. Jednym z nich jest interfejs konstruktora, który zawiera funkcję konstruktora i wszystkie informacje statyczne. Drugi to interfejs instancji, który zawiera wszystkie dynamiczne informacje na instancję. Ten podział pochodzi ze starego JavaScript, kiedy klasy były zdefiniowane jako funkcja konstruktora i prototyp, na przykład:

```
// Static parts --> constructor interface

function Person(name, age) {

this.name = name

this.age = age;

}

Person.create(name, age) {

return new Person(name, age)

}

// Dynamic parts --> instance interface

Person.prototype.toString() {

return `My name is ${this.name} and I'm

${age} years old`

}
```

W naszym przypadku Object jest interfejsem instancji, a Object Constructor jest interfejsem konstruktora. Uczyńmy Object.keys silniejszym:

```
// A utility type

type ReturnKeys<O> =

O extends number ? [] :

O extends Array<any> | string ? string[] :

O extends object ? Array<keyof O> : never

// Extending the interface

interface ObjectConstructor {

keys<O>(obj: O) : ReturnKeys<O>

}
```

Umieśćmy to w naszym pliku deklaracji typu otoczenia, a Object.keys natychmiast uzyska lepsze wnioskowanie o typie.