Lekcja 17: Zastępowalność

W ostatniej Lekcji widzieliśmy, że możemy porzucić parametry funkcji, jeśli typ deklaruje, że ten argument jest opcjonalny. Jest w tym trochę więcej. Możemy również całkowicie porzucić argumenty, jeśli nie mamy z nich żadnego pożytku:

```
// This is a valid search Function

const dummyContentSearchFn: SearchFn = function() {

return Promise.resolve([{

title: 'Form Design Patterns',

url: '/form-design-patterns',

abstract: 'A practical book on accessible forms'

}])

}
```

Ta funkcja spełnia warunki kontraktu typu przez SearchFn, mimo że początkowo kształt funkcji nie wygląda na zgodny. Tyle że tak jest. Ma to coś wspólnego ze sposobem działania JavaScript.

Liczba parametrów

W JavaScript możemy wywołać funkcję z dowolną ilością parametrów, bez względu na to, ile zdefiniujemy w nagłówku funkcji. Może to prowadzić do dwóch skrajnych przypadków. Najpierw wywołujemy funkcję z brakującymi argumentami. Ponieważ najprawdopodobniej polegamy na pobieraniu wartości z treści funkcji, powoduje to żałosne niepowodzenie funkcji. Ale jest to objęte kontrolą typu TypeScript. Jeśli podpis funkcji wymaga od nas przekazania argumentów, TypeScript zgłosi błąd, jeśli tego nie zrobimy - i poda nam czerwone zawijasy.

```
// Calling our original search function.

// TypeScript tells us that we need to pass

// at least a query

search()
```

Dobrze! TypeScript upewnia się, że przekazujemy parametry wymagane przez typ funkcji. Więc wywołanie funkcji jest objęte. Jednak drugi przypadek krawędzi jest inny: możemy przekazać do funkcji zbyt wiele parametrów, a nadmiarowe parametry są po prostu ignorowane. Jest okej. Dlaczego mielibyśmy definiować parametry funkcji w głowie naszej funkcji, skoro nie mamy z nich żadnego zastosowania w ciele naszej funkcji? JavaScript nadal pozwala nam przekazywać parametry; po prostu nic z nimi nie robimy. To sprawia, że dummyContentSearchFn, bez parametrów, jest zgodny z typem SearchFn. Przyjemnym efektem ubocznym jest to, że ponieważ jawnie wpisaliśmy dummyContentSearchFn do SearchFn i przypisaliśmy funkcję anonimową, nie możemy wywołać dummyContentSearchFn bez odpowiedniej ilości parametrów zdefiniowanych przez SearchFn.

```
dummyContentSearchFn('Ember') // Good!

dummyContentSearchFn('Ember', ['JavaScript']) // Good!

// Not good, as an explicitly typed SearchFn requires
```

```
// us to pass parameters

dummyContentSearchFn()
```

Jeśli zmienimy atrybuty dummyContentSearchFn na funkcję nazwaną, a nie jawnie wpisaną, zachowanie jest zasadniczo inne:

```
function dummyContentSearchFn() {

return Promise.resolve([{

title: 'Form Design Patterns',

url: '/form-design-patterns',

abstract: 'A practical book on accessible forms'

}])

}

dummyContentSearchFn('Ember') // Nope!

dummyContentSearchFn('Ember', ['JavaScript']) // Nope!

// Good!

dummyContentSearchFn()
```

Dzieje się tak, ponieważ TypeScript ponownie sprawdza kontrakt typu funkcji. I tym razem umowa ma nie mieć żadnych argumentów. Nadal możemy przekazać dummyContentSearchFn do displaySearch. Wewnątrz displaySearch dummyContentSearchFn przyjmuje kształt SearchFn, mimo że nic nie robi z przekazanymi parametrami.

```
displaySearch ('wejście', 'wyjście', dummyContentSearchFn)
```

TypeScript nazywa to zachowanie zastępowalnością. Jeśli ma to sens, możemy zastąpić jedną sygnaturę funkcji inną. Pomijanie parametrów, jeśli nie mamy dla nich żadnego zastosowania w treści funkcji, jest w porządku. Kod będzie nadal działał. Jest to jeden z wielu sposobów, w jaki TypeScript jest mniej rygorystyczny i bardziej pragmatyczny, aby dostosować się do sposobu działania JavaScript.

void

Zastępowalność działa, ponieważ typy zwracanych wartości pozostają takie same. W obu dummyContentSearchFn i testSearchFn zwracamy obietnicę z tablicą wyników. Przekazane parametry znikają, gdy już ich nie potrzebujemy. Są sytuacje, w których możemy również podstawić zwracany typ funkcji: kiedy zwracanym typem jest void. void jako typ jest ciekawą konstrukcją w TypeScript, ponieważ próbuje odzwierciedlić zachowanie void w innych językach programowania, ale ma dużo więcej wspólnego z void w JavaScript. W JavaScript wszystkie funkcje mają wartość zwracaną. Jeśli nie włączymy go ponownie, wartość zwracana jest domyślnie niezdefiniowana. W TypeScript każda funkcja ma typ zwracany. Jeśli nie wpiszemy jawnie ani nie wywnioskujemy, typem zwracanym jest domyślnie void. void w TypeScript to inny sposób na powiedzenie undefined. Typ void może mieć tylko jedną wartość, która jest niezdefiniowana, ale ma kilka interesujących funkcji. Na razie refaktoryzujmy funkcję wyszukiwania, aby nie zwracała obietnicy, ale przekazywała wyniki do wywołania zwrotnego.

```
// We add a callback as second parameter, as
```

```
// optional parameters always have to be last
function search(
query: string,
callback: (results: Result[]) => void,
tags?: string[]
) {
let queryString = `?query=${query}`
if(tags && tags.length) {
queryString += `&tags=${tags.join()}`
}
fetch(`/search${queryString}`)
.then(res => res.json() as Promise<Result[]>)
// Here, we pass the results to our callback
.then(results => callback(results))
}
```

Funkcja podobna do oryginalnej, ale drugi parametr jest teraz wywołaniem zwrotnym, które pobiera tablicę wyników i zwraca wartość void. Możemy skorzystać z nowej funkcji wyszukiwania w ten sposób:

```
// logs all results to the console
search('Ember', function(results) {
console.log(results)
})
```

I, jak przyzwyczailiśmy się do wywołań zwrotnych, możemy przekazać dowolną funkcję, która przypomina jej kształt:

```
function searchHandler(results: Result[]) {
console.log(results)
}
search('Ember', searchHandler)
```

Ale o to chodzi: możemy również przekazywać funkcje, które mają inny typ zwracania.

```
// Search handler now returns a number
function searchHandler(results: Result[]): number {
return results.length
}
```

```
// Totally OK!

search('Ember', searchHandler)
```

Możemy zastąpić dowolny typ void. Wewnątrz funkcji wywołującej zwracany typ będzie traktowany jako niezdefiniowany, co oznacza, że nie możesz z nim zrobić niczego, co nie pozwoliłoby TypeScript krzyczeć na ciebie czerwonymi falistymi liniami:

```
function search(

query: string,

callback: (results: Result[]) => void,

tags?: string[]

) {

...

fetch(`/search${queryString}`)

.then(res => res.json() as Promise<Result[]>)

.then(results => {

const didItWork = callback(results)

// didItWork is undefined! This causes an error

didItWork += 2

})

}
```

Ma to również być zgodne ze sposobem działania JavaScript. Są sytuacje, w których przekazujemy funkcje zwrotne, które zwracają coś, nawet jeśli wywoływana funkcja nic z tym nie robi, zwłaszcza jeśli chcesz wielokrotnie używać funkcji w różnych scenariuszach.

```
// This function shows results in an HTML element

// but also returns the container element that

// has been filled

function showResults(results: Result[]) {

const container

= document.getElementById('results')

if(container) {

container.innerHTML = `<ul>

${results.map(el => `<li>${el.title}</li>`)}

<ul>`;
```

```
}

return container;

}

// Somewhere in our app, we show a list of

// pages on click

button.addEventListener('click', function() {

const el = showResults(storedResults)

if(el) {

el.style.display = 'block'

}

})

// But hey, this function also makes a good

// search handler

search('Ember', showResults)
```

Jeśli naprawdę chcesz się upewnić, że żadna wartość nie zostanie zwrócona, możesz umieścić void przed callback w zwykłym JavaScript:

```
function search(

query: string,

callback: (results: Result[]) => void,

tags?: string[]

) {

...

fetch(`/search${queryString}`)

.then(res => res.json() as Promise<Result[]>)

// void is a keyword in JavaScript returning

// undefined

.then(results => void callback(results))

}
```

lub użyj undefined jako typu:

```
// We change the return type of callback to

// undefined
```

```
function search(
query: string,
callback: (results: Result[]) => undefined,
tags?: string[]
) {
...
}
function searchHandler(results: Result[]): number {
return results.length
}
// This breaks now!
search('Ember', searchHandler)
```

Ponieważ nie możemy zastąpić undefined dla number. Zastępowalność to koncepcja w TypeScript, na którą się natkniesz, jeśli dużo pracujesz z funkcjami. Zamiast być zbyt rygorystycznym, jeśli chodzi o dokładne kształty funkcji, uzupełnia sposób, w jaki JavaScript działa z funkcjami: prosi tylko o parametry, których faktycznie potrzebujesz. Istnieją jednak sytuacje, w których deklaracja funkcji TypeScript może mieć więcej argumentów niż jej odpowiednik w JavaScript. I to dosłownie ma coś wspólnego z this.