Lekcja 20: Przeciążanie funkcji

W poprzedniej Lekcji widzieliśmy kilka różnych implementacji funkcji search. Przejrzyjmy. Po pierwsze, tradycyjne. Wymaga warunku i kilku opcjonalnych tagów i zwraca obietnicę z wynikami:

declare function search(

term: string,

tags?: string[]

): Promise<Result[]>

Po drugie, mieliśmy funkcję wyszukiwania, która pobiera warunek, wywołanie zwrotne i opcjonalne tagi jako trzeci parametr. Nie zwracamy wartości, tylko void.

declare function search(

term: string,

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

tags?: string[]

): void

Może być więcej odmian, ale na razie skupmy się na tych dwóch. Te dwa nagłówki funkcji wyglądają zupełnie inaczej, z wyjątkiem pierwszego parametru. Zwracają różne rzeczy i przyjmują argumenty w innej kolejności lub całkowicie z różnymi typami. Jednak nie jest tak rzadkie w JavaScript, że mają funkcje, które przyjmują argumenty o różnym typie w różnych pozycjach. Przyjrzyj się funkcji write interfejsu API systemu plików w Node.js. Drugi argument może być buforem z danymi lub ciągiem znaków, który zapisujemy w pliku. Później jest wiele opcjonalnych parametrów. Jedyną stałą rzeczą jest to, że ostatnim argumentem jest zawsze wywołanie zwrotne. Ale to wywołanie zwrotne może znajdować się na pozycji trzeciej, czwartej, piątej lub nawet szóstej! W JavaScript argumenty funkcji są tym, co z nich robimy. To zasadniczo różni się od niektórych innych języków programowania, które pozwalają na wiele funkcji w tym samym zakresie. Inne języki programowania, takie jak C ++, Java lub C#, nazywają tę funkcję przeciążeniem: posiadanie więcej niż jednej implementacji, o ile lista argumentów jest inna. W JavaScript możemy mieć tylko jedną funkcję o określonej nazwie w określonym zakresie. Ale lista argumentów może być tak dynamiczna, jak jej potrzebujemy. Zatem funkcje w JavaScript mogą robić praktycznie wszystko. Ale jak to wpisujemy? W tym miejscu TypeScript zapożycza pojęcie przeciążania funkcji z innych języków programowania: jeśli twoja funkcja może mieć cokolwiek jako argumenty, możesz przynajmniej napisać definicje typów dla różnych przypadków użycia. Zamiast mieć wiele implementacji, masz jedną implementację, ale więcej typów. W TypeScript definiujemy takie typy, pisząc przeciążenia funkcji nad rzeczywistą implementacją. W przypadku naszej funkcji wyszukiwania wiemy już, jakich typów potrzebujemy; ogłosiliśmy je na początku tej lekcji:

function search(

term: string,

tags?: string[]

): Promise<Result[]>

function search(

term: string

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

tags?: string[]

): void

Po dwóch deklaracjach funkcji potrzebujemy rzeczywistej funkcji, która implementuje naszą funkcję wyszukiwania. Nagłówek funkcji musi być bardzo wyjątkowy: lista argumentów musi obejmować wszystkie listy argumentów ze wszystkich powyższych przeciążeń funkcji. Ponieważ większość argumentów jest różnych, do ich wpisania używamy any lub undefined.

function search(

term: string,

tags?: string[]

): Promise<Result[]>

function search(

term: string,

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

tags?: string[]

): void

// Here comes the implementation

function search(

term: string,

p2?: unknown,

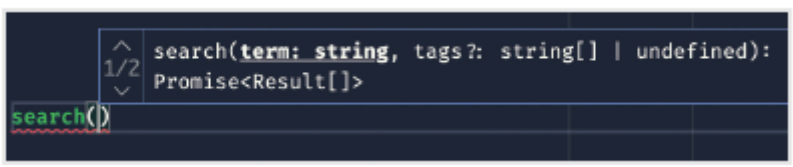
p3?: string[]

) {

// Now for the implementation

}

W momencie, gdy zdefiniujemy naszą funkcję wyszukiwania w ten sposób, otrzymamy autouzupełnianie dla dwóch przeciążeń funkcji.



Kod programu Visual Studio przedstawiający dwie możliwe funkcje wyszukiwania

Jak wymyśliliśmy ostateczny nagłówek funkcji?

1. Pierwszy argument, term, jest taki sam w obu nagłówkach funkcji. Przenosimy ten argument.

2. Drugi argument, p2, może być opcjonalnymi tags lub wywołaniem zwrotnym. Jest więc zdecydowanie opcjonalny i może mieć dwa typy. Dlatego na razie wybieramy unknown.

3. Trzeci argument, p3, musi być opcjonalny, ponieważ drugi argument może być opcjonalny. Oczywiście nie możemy wprowadzić wymaganych parametrów po parametrach opcjonalnych. Ale gdy pojawi się argument trzeci, z pewnością są to tags z przeciążenia drugiej funkcji. Stąd string [].

Celowo wybieramy nieopisowe nazwy, takie jak p2 lub p3, aby przedefiniować je do ich rzeczywistego przeznaczenia w treści funkcji. Możemy również nazwać p2 coś w rodzaju tagsOrCallback, a p3 to tagsAfterCallback, jeśli chcesz być bardziej celowy. Dobrą praktyką jest celowe potraktowanie przeładowanych argumentów na samym początku treści funkcji:

```
function search(

term: string,

tags?: string[]

): Promise<Result[]>

function search(

term: string,

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

tags?: string[]

): void

function search(

term: string,

p2?: unknown,

p3?: string[]

) {

// We only have a callback if 'p2' is a function

const callback =

typeof p2 === 'function' ? p2 : undefined

// We have tags if p2 is defined and an array, or if p3

// is defined and an array

const tags =

typeof p2 !== 'undefined' && Array.isArray(p2) ? p2 :

typeof p3 !== 'undefined' && Array.isArray(p3) ? p3 :

undefined;
```

```
let queryString = `?query=${term}`
if(tags && tags.length) {
// tags at this point has to be an array
queryString += `&tags=${tags.join()}`
}
// The actual fetching of results!
const results = fetch(`/search${queryString}`)
.then(response => response.json())
// callback is either undefined or a function, as
// seen above
if(callback) {
// Now it's definitely a function! So let's then()
// the results and call the callback!
// We don't return anything. This is equivalent to
// void
return void results.then(res => callback(res))
} else {
// Otherwise, we have to return a promise with
// results as described in the first function
// overload
return results
}
}
```

Jest wiele do zrobienia, jeśli musisz zadbać o zaimplementowanie dwóch różnych nagłówków funkcji jednocześnie. Im więcej masz przeciążeń, tym bardziej się komplikuje. Jeśli to możliwe, nie powinniśmy zbytnio przeciążać. Jak to kiedyś ujął Shawn Wang, jeśli jest to trudne do zaimplementowania w języku TypeScript, może być trudne w użyciu bez języka TypeScript. Mimo to, TypeScript rozpozna funkcję wyszukiwania jako kompatybilną ze wszystkim, co potrzebuje wersji opartej na obietnicy i wszystkim, co potrzebuje wariantu wywołania zwrotnego. Zauważ, że wewnątrz ciała funkcji tracimy wiele informacji o typie, ponieważ p2 jest nieznany. Możemy zawęzić to do zwykłej funkcji lub dowolnej tablicy. Jeśli chcemy dowiedzieć się więcej o typach w treści funkcji i nadal spełniać przeciążenia funkcji, możemy być bardziej jednoznaczni:

```
function search(
term: string,
```

```
tags?: string[]
): Promise<Result[]>
function search(
term: string,
callback: (results: Result[]) => void,
tags?: string[]
): void
function search(
term: string,
p2?: string[] | ((results: Result[]) => void,
p3?: string[]
) {
// All from above, but with better type info
}
```

Ta konstrukcja jest nazywana typem unii, w którym definiujemy ten parametr jako tablicę ciągów lub funkcję, która akceptuje tablicę wyników jako parametr.

Typy funkcji z przeciążeniami

W poprzednich Lekcjach dowiedzieliśmy się, że możemy tworzyć aliasy typów funkcji, aby zadeklarować kontrakt w jednym typie. Podobnie jak wcześniej SearchFn. Możemy zrobić to samo z przeciążonymi funkcjami. Szybkim sposobem jest ponowne pobranie typu za pomocą typeof search i może to wyglądać mniej więcej tak:

```
type SearchOverloadFn = {
// Function overload number 1
(
term: string,
tags?: string[] | undefined
) : Promise<Result[]>;
// Function overload number 2
(
term: string,
callback: (results: Result[]) => void,
tags?: string[] | undefined
```

): void;

}

Możemy ponownie użyć tego typu funkcji, aby wpisać funkcje strzałkowe, które reagują na przeciążenia:

const searchWithOverloads: SearchOverloadFn =

(

term: string,

p2?: string[] | (results: Result[]) => void,

p3?: string[]

) => {

// Do your magic

}

Zwróć uwagę, że wnioskowanie o typie jest trudne w przypadku wielu odmian. Dlatego musisz samodzielnie podać typy parametrów w funkcji strzałki.