Lekcja 18: This i That

Nadszedł czas, aby zaimplementować funkcję displaySearch, aby pokazać nam inną stronę wywołań zwrotnych i funkcji, która jest unikalna dla języka TypeScript. Oto głowa naszej funkcji:

zadeklaruj funkcję displaySearch (

declare function displaySearch(

inputId: 'string',

outputId: 'string',

search: SearchFn

): void;

Chcemy wybrać elementy w naszym znaczniku za pośrednictwem document.getElementById i

1. Pobierz bieżącą wartość z pola wejściowego.

2. Pokaż wyniki w elemencie wyjściowym.

Chodźmy!

Implementacja

Załóżmy, że jest to niezbędny znacznik, który chcemy ulepszyć naszą małą funkcją:

<form action = "/ search" method = "POST">

<label for = "search"> Przeszukaj witrynę </label>

<input type = "search" id = "search">

<button type = "submit">

</form>

<div id = "output" hidden>

W prawdziwie progresywnym ulepszaniu to pole wyszukiwania działa doskonale i prowadzi użytkowników do strony wyszukiwania ze wszystkimi wynikami opartymi na haśle wpisanym w polu wyszukiwania. Chcemy jednak wzbogacić go o funkcje dynamiczne: ładowanie wyników wyszukiwania podczas pisania, wyświetlanie pierwszych pięciu w polu output. Dawanie ludziom wyobrażenia o tym, czego się spodziewać. Najpierw wybieramy element input, który przechodzi przez identyfikator przekazany w argumencie inputId. Chcemy słuchać wszystkich wyzwalanych zdarzeń change. Po wywołaniu zdarzenia change ustawiamy stan aktywny dla całego formularza, co oznacza, że dodajemy klasę o nazwie active do elementu nadrzędnego.

function displaySearch (

inputId: string,

outputId: string,

search: SearchFn

```
): void {

document.

getElementById(inputId)?.

addEventListener('change', function() {

this.

parentElement?.

classList.add('active')

})

}
```

Pomijając adnotacje typu w nagłówku funkcji, widzimy zwykły JavaScript. A TypeScript nie narzeka na nic. Żadnych czerwonych zawijasów. Wszystko kompiluje się tak, jak chcemy. Czy jest coś dziwnego w tak płynnej pracy? Nie? Dobrze! Jest to zwykły, działający kod JavaScript, tak jak napisalibyśmy go bez TypeScript, więc dobrze, że TypeScript nie rzuca na ciebie czerwonych zawijasów. Teraz dodajmy więcej kodu. Po ustawieniu naszego formularza na active pobieramy bieżącą wartość z pola wejściowego, aby przekazać ją do funkcji wyszukiwania.

```
function displaySearch (

inputId: string,

outputId: string,

search: SearchFn

): void {

document.

getElementById(inputId)?.

addEventListener('change', function() {

this.

parentElement?.

classList.add('active')

const searchTerm = this.value

})

}
```

Co to jest? Czerwony falisty? Jak to się stało? Cóż, porozmawiajmy o tym.

Wiązanie funkcji i elementy HTML

W JavaScript zwykłe funkcje są zawsze powiązane z obiektem. Ten obiekt staje się dostępny dzięki this w tych funkcjach. W naszym przykładzie funkcja zwrotna jest powiązana z elementem pobranym przez getElementById. W tym momencie jest to węzeł elementu, który jest typu HTMLElement. To

zachowanie jest zapewniane przez TypeScript. Celem getElementById jest pobranie elementów HTML. Każdy element jest określonego typu - możesz je wyszukać w MDN, gdzie zobaczysz szeroki zakres dla prawie każdego elementu HTML. Od HTMLAnchorElement (element a) do HTMLVideoElement (element wideo). Są to interfejsy dostępne w przeglądarce.

```
// Tworzenie elementu HTMLVideoElement przy użyciu

// metka

const x = document.createElement ('video')

console.log (x.toString ())

// Wyświetla „[obiekt HTMLVideoElement]”

// nazwa faktycznego interfejsu przeglądarki
```

Wiele z tych interfejsów jest generowanych automatycznie przez skrobanie standardowych dokumentów sieciowych i wyszukiwanie części, w których interfejsy są opisane w formacie języka definicji interfejsu internetowego (WIDL). TSJS Generator20 konwertuje pliki WIDL na pliki deklaracji TypeScript: wspaniałe okno na to, co dzieje się w wewnętrznych przeglądarkach!

Te elementy mają bardzo płytką hierarchię dziedziczenia. W przypadku nadtypu najniższym wspólnym mianownikiem jest HTMLElement. Jest to również interfejs, do którego uzyskujemy dostęp przez to w naszej funkcji wywołania zwrotnego.

Jest to również najbardziej konkretny kontrakt, jaki mogą nam zapewnić predefiniowane typy TypeScript. W jaki sposób narzędzie do statycznej analizy kodu powinno wiedzieć, który element znajduje się pod inputId? Podczas korzystania z metod takich jak querySelector najniższy wspólny mianownik staje się jeszcze bardziej ogólny: Element, który nie pozwoliłby nam nawet uzyskać dostępu do elementu parentElement. Jeśli jednak zdecydujesz się na selektor elementów – np. querySeelctor(‘iput’) - TypeScript może zapewnić niewielką pomoc. W naszym przypadku musimy sprawdzić, co to za instancja podtypu this . Towarzyszy temu inna kontrola typu: instanceof.

```
function displaySearch(

inputId: string,

outputId: string,

search: SearchFn

): void {

document.

getElementById(inputId)?.

addEventListener('change', function() {

// This is of type HTMLElement because

// getElementById says so

this.

parentElement?.
```

```
classList.add('active')

if(this instanceof HTMLInputElement) {

// From here on, this is

// of type HTMLInputElement

const searchTerm = this.value // Works!

search(searchTerm)

.then(results => {

// TODO in another lesson

})

}

})

}
```

Dodatkową korzyścią jest to, że nasz kod jest dużo bezpieczniejszy. TypeScript ponownie wskazuje nam rzeczy, które mogą powodować pewne problemy: tak, możemy być pewni, że this jest to element HTMLElement, gdy dotrzemy do wywołania zwrotnego, ale nigdy nie możemy być całkowicie pewni, że jest to element typu HTMLInputElement. Co się stanie, jeśli identyfikatory zmienią się w naszych znacznikach? Nasz kod rozpadłby się na kawałki i wyrzucił błędy. Tutaj mamy strażnika, którego możemy sprawdzać i sterować w oparciu o wynik. Bezpieczeństwo wpisywania w naszym kodzie prowadzi do bardziej niezawodnego kodu po dostarczeniu.

Wyodrębnianie wywołania zwrotnego

Pozostaje nam jeden problem: co jeśli chcemy wyodrębnić wywołanie zwrotne do jego własnej funkcji? Nie jest to rzadkością podczas pisania JavaScript; ta sama funkcja może być używana w różnych miejscach. Ale w momencie, gdy wyodrębnimy funkcję i umieścimy ją w innym miejscu, również tracimy związek do this!

```
function inputChangeHandler() {

// We have no clue what this can be

// that's why we get red squigglies

this.

parentElement?.

classList.add('active')

}

function displaySearch(

inputId: string,

outputId: string,
```

```
search: SearchFn

): void {

document.

getElementById(inputId)?.

// Only here, inputChangeHandler's this

// becomes of type HTMLElement again

// but inputChangeHandler doesn't know about that

addEventListener('change', inputChangeHandler)

}
```

TypeScript radzi sobie z takimi sytuacjami: wolno nam wpisać this! Deklaracje funkcji mogą mieć inny dodatkowy parametr, który musi znajdować się na pierwszej pozycji: this.

```
// Definiujemy, że jest to typ HTMLElement

function inputChangeHandler (this: HTMLElement) {

to.

parentElement ?.

classList.add ('active')

}
```

Ten parametr zostanie usunięty, gdy skompilujemy TypeScript do JavaScript. Ponownie, są dodatkowe korzyści. Możemy używać inputChangeHandler tylko wtedy, gdy możemy być pewni, że będzie to (pod) typ HTMLElement. Gwarantuje to również, że nie wywołujemy metody inputChangeHandler na zewnątrz bez kontekstu:

```
// Kontekst „this” typu „void” nie jest możliwy do przypisania

// do metody „this” typu „HTMLElement”.

inputChangeHandler ()
```

I znowu, kierujemy się podstawową zasadą TypeScript: być tak prostym w użyciu, jak to tylko możliwe, ale zapewniamy wystarczające bezpieczeństwo czcionek, ponieważ musimy mieć pewność, że nie strzelimy sobie w stopę. TypeScript umieszcza czerwone zawijasy tam, gdzie musimy być bardziej wyraźni w naszym myśleniu, jeśli chcemy mieć pewność, że nie złamiemy się po uruchomieniu programu i zmianie ulegną subtelne rzeczy.