Lekcja 35: Wartości domyślne typu ogólnego

Pokażemy wideo wewnątrz elementu video. Aby ułatwić sobie i naszym współpracownikom, decydujemy się na abstrakcyjne obchodzenie się z DOM i wybieramy do tego celu klasy. Klasa powinna zachowywać się następująco:

1. Możemy utworzyć dowolną liczbę instancji i przekazać do niej nasze preferencje użytkownika. Preferencje użytkownika są ważne, aby wybrać właściwy adres URL formatu wideo.

2. Możemy dołączyć do niego dowolny element HTML. Jeśli jest to element video, ładujemy źródło wideo bezpośrednio. Jeśli jest to jakikolwiek inny element, używamy go jako opakowania dla nowo utworzonego elementu video. elementy vwideo są jednak domyślne.

3. Element nie jest wymagany do tworzenia instancji; możemy ustawić to na późniejszym etapie. Oznacza to, że element może być niezdefiniowany w momencie wczytywania wideo.

Zaimplementujmy tę klasę.

Przechodzenie do Ogólnych

Najpierw tworzymy pomocniczy typ Nullable, który dodaje undefined do unii. To sprawia, że czytanie typów klas pól jest znacznie łatwiejsze.

```
type Nullable<G> = G | undefined
```

Następnie zaczynamy od zajęć. Ustawiamy typ elementu na HTMLElement, ponieważ jest to nadtyp wszystkich elementów HTML.

```
class Container {

#element: Nullable<HTMLElement>;

#prefs: UserPreferences

// We only require the user preferences

// to be set at instantiation

constructor(prefs: UserPreferences) {

this.#prefs = prefs

}

// We can set the element to an HTML element

set element(value: Nullable<HTMLElement>) {

this.#element = value

}

get element(): Nullable<HTMLElement> {

return this.#element

}

// We load the video inside a video element.
```

```
// If #element isn't an HTMLVideoElement, we
// create one and append it to #element
loadVideo(formats: VideoFormatURLs) {
const selectedFormat =
formats[this.#prefs.format].href
if(this.#element instanceof HTMLVideoElement) {
this.#element.src = selectedFormat
} else if(this.#element) {
const vid = document.createElement('video')
this.#element.appendChild(vid)
vid.src = selectedFormat
}
}
}
```

I to już działa cudownie:

```
const container = new Container(userPrefs)
container.element = document.createElement('video')
container.loadVideo(videos)
```

HTMLElement może być zbyt ogólny dla niektórych gustów. Zwłaszcza gdy mamy do czynienia z wideo, możemy chcieć pracować z funkcjami wideo HTMLVideoElement. Pracując z tym, potrzebujemy odpowiednich informacji typu. Pomóc mogą leki generyczne. Możemy dokładnie określić typ, z którym mamy do czynienia, a dzięki ograniczeniom typu możemy upewnić się, że jest to rozszerzenie naszego nadtypu HTMLElement.

```
class Container<GElement extends HTMLElement> {
#element: Nullable<GElement>;
// ...abridged...
set element(value: Nullable<GElement>) {
this.#element = value
}
get element(): Nullable<GElement> {
return this.#element
}
```

// ...abridged...

}

Tak jest lepiej, ale jeszcze nie jesteśmy z tego do końca zadowoleni.

Dodawanie ustawień domyślnych

Ponieważ w konstruktorze brakuje nam konkretnego elementu, TypeScript nie ma nic do wnioskowania, aby powiązać GElement z konkretnym typem. Wracamy do nadtypu HTMLElement bez wyraźnej adnotacji ogólnej:

// container accepts any HTML element

const container

= new Container(userPrefs)

// container accepts HTMLVideoElement

const vidcontainer

= new Container<HTMLVideoElement>(userPrefs)

I to jest złe, ponieważ naszym domyślnym zawsze powinien być HTML VideoElement. Inne elementy są wyjątkiem. W tym miejscu pojawiają się ogólne parametry domyślne. Jeśli nie udostępnimy adnotacji ogólnej, TypeScript użyje parametru domyślnego jako typu.

class Container<

GElement extends HTMLElement = HTMLVideoElement> {

// ...

}

// container accepts HTMLVideoElement

const container = new Container(userPrefs)

W porównaniu z ograniczeniami typu, ogólne parametry domyślne nie tworzą granicy, ale stanowią wartość domyślną na wypadek, gdybyśmy nie mogli wywnioskować lub nie opisywać. Jeśli ogólny parametr domyślny istnieje bez granic, ogólny może zaakceptować any. Podobnie jak parametry domyślne funkcji, ogólne parametry domyślne muszą znajdować się na końcu definicji ogólnej.

Ogólne parametry domyślne są niezwykle przydatne w przypadku klas, które muszą wiązać typ ogólny, ale nie mają informacji podczas tworzenia wystąpienia. We wszystkich innych przypadkach ograniczenia typu działają najlepiej.

Ogólne parametry domyślne i wnioskowanie o typie

Chociaż ogólne parametry domyślne mogą być niezwykle potężne, musimy być również bardzo ostrożni. Weźmy tę funkcję, która robi coś podobnego do klasy Container. Wczytuje wideo w elemencie i rozróżnia następujące przypadki:

1. Jeśli nie dostarczymy elementu, tworzymy element video

2. Jeśli udostępniamy element video, wczytujemy video w tym elemencie

3. Jeśli udostępnimy jakikolwiek inny element, używamy go jako opakowania dla nowego elementu video.

Funkcja zwraca element, który przekazaliśmy jako argument do dalszych operacji. Dzięki ogólnym parametrom domyślnym możemy pięknie zdefiniować to zachowanie i polegać tylko na wnioskach o typie:

```
declare function createVid<

GElement extends HTMLElement = HTMLVideoElement

>(

prefs: UserPreferences,

formats: VideoFormatURLs,

element?: GElement

)
```

Jeśli spróbujemy, otrzymamy:

```
declare const userPrefs: UserPreferences

declare const formats: VideoFormatURLs

// a is HTMLVideoElement, the default!

const a = createVid(userPrefs, formats)

// b is HTMLDivElement

const b = createVid(

userPrefs, formats,

document.createElement('div'))

// c is HTMLVideoElement

const c = createVid(

userPrefs, formats,

document.createElement('video'))
```

Jednak działa to tylko wtedy, gdy polegamy wyłącznie na wnioskowaniu o typie. Typy generyczne pozwalają nam również jawnie wiązać typ.

```
const a = createVid<HTMLAudioElement>(userPrefs, formats)
```

a jest typu HTMLAudioElement, mimo że nasza implementacja zwróci HTMLVideoElement. Ponadto, ponieważ jesteśmy na poziomie typu, implementacja nie ma pojęcia, że chcemy mieć HTMLAudioElement. Dlatego musimy zachować ostrożność, używając ogólnych parametrów domyślnych. Mamy też znacznie lepsze narzędzie do takich przypadków, o czym przekonamy się w następnym rozdziale.

Podsumowanie

Ogólne pozwalają nam przygotować się na typy, których nie znamy z góry. Dzięki temu możemy projektować niezawodne interfejsy API z lepszymi informacjami o typie i upewnić się, że przekazujemy tylko wartości, w których nasze typy spełniają określone kryteria.

1. Dzięki rodzajom ogólnym upewniliśmy się, że nie musimy tworzyć więcej funkcji tylko po to, aby zadowolić system typów. Ogólne pozwalają nam uogólniać funkcje do szerszego zastosowania.

2. Ogólne ograniczenia pozwalają nam tworzyć granice. Zamiast akceptować cokolwiek dla naszych typów ogólnych, możemy ustawić pewne kryteria, takie jak istnienie określonych kluczy lub typów właściwości.

3. Ogólne pozwalają nam również lepiej pracować z kluczami obiektów. W zależności od tego, co przekażemy jako argument funkcji, możemy wywnioskować właściwe klucze i pozwolić, aby TypeScript rzucił na nas czerwone zawijasy, jeśli nie podamy poprawnych argumentów.

4. Ogólne działają wyjątkowo dobrze z typami mapowanymi. Dzięki mapom kluczy unii, typom dostępu do indeksu i pomocnikowi rekordu jesteśmy w stanie utworzyć typ, który pozwala nam podzielić typ obiektu na zestaw unii.

5. Modyfikatory typu zmapowanego pozwalają nam kopiować typ obiektu, ale ustawiają wszystkie właściwości jako opcjonalne, wymagane lub tylko do odczytu.

6. Dowiedzieliśmy się dużo o wiążących rodzajach. Moment, w którym podstawiamy typ ogólny na prawdziwy, ma kluczowe znaczenie dla zrozumienia systemu typów TypeScript.

Widzieliśmy również, jak działają klasy ogólne i jak używamy domyślnych typów ogólnych, aby nieco ułatwić nam życie.

Praca z typami ogólnymi jest kluczem do maksymalnego wykorzystania systemu czcionek TypeScript. Typy generyczne zostały zaprojektowane tak, aby były zgodne z większością rzeczywistych scenariuszy JavaScript i otwierały drzwi do jeszcze lepszych i bardziej niezawodnych informacji o typach. Następny rozdział zabierze nas jeszcze dalej!