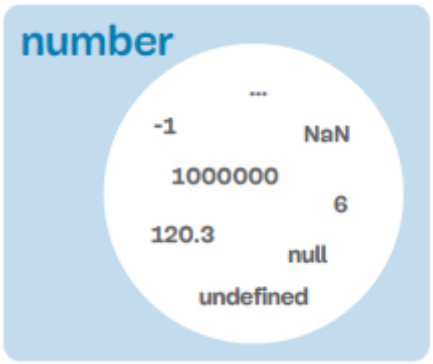
Lekcja 28: undefined i null

Zanim zamkniemy tą częśćł, musimy porozmawiać o dwóch specjalnych typach wartości, które wcześniej czy później złapiesz w swoich aplikacjach: null i undefined. Zarówno null, jak i undefined oznaczają brak wartości. undefined mówi nam, że zmienna lub właściwość została zadeklarowana, ale nie została przypisana żadna wartość. Z drugiej strony null to pusta wartość, którą można przypisać w celu wyczyszczenia zmiennej lub właściwości. Obie wartości są nazywane wartościami dolnymi, czyli wartościami, które nie mają rzeczywistej wartości. Douglas Crockford powiedział kiedyś, że w społeczności języków programowania toczy się wiele dyskusji na temat tego, czy język programowania powinien mieć w ogóle wartości dolne. Nikt nie uważa, że muszą być dwie. undefined i null w obszarze Type Space undefined i null są nieco wyjątkowe w TypeScript. Obie wartości są regularnie częścią każdego zbioru typów.



Numer typu z wartością undefined i null.

Dzieje się tak, ponieważ JavaScript zachowuje się w ten sposób. W momencie zadeklarowania zmiennej jest ona ustawiona na undefined. Programowo możemy ustawić zmienne na null lub undefined. Ale to rodzi pewne problemy. Spójrzmy na ten prosty przykład:

// Let's define a number variable

let age: number

// I'm getting one year older!

age = age + 1

To jest prawidłowy kod TypeScript. Deklarujemy liczbę i dodajemy do niej kolejną wartość liczbową. Problem w tym, że przynosi nam to wartości, których byśmy się nie spodziewali. Wynikiem tej operacji jest NaN, ponieważ dodajemy 1 do undefined. Technicznie rzecz biorąc, wynik jest ponownie numerem typu, ale nie taki, jakiego się spodziewaliśmy! Może się pogorszyć. Wróćmy do naszego przykładu wydarzenia technicznego. Chcemy stworzyć reprezentację HTML jednego z naszych wydarzeń i dołączyć ją do listy elementów. Tworzymy funkcję, która działa na typowych właściwościach i zwraca ciąg:

function getTeaserHTML (event: TechEvent) {

return `<h2> $ {event.title} </h2>

<p>

$ {event.description}

</p> `

}

Używamy tej funkcji do stworzenia elementu listy, który możemy dodać do naszej listy zdarzeń:

function getTeaserListElement (event: TechEvent) {

const content = getTeaserHTML (zdarzenie)

element const = document.createElement ('li')

element.classList.add ('karta-zwiastun')

element.innerHTML = treść

element zwrotny

}

Trochę szorstkie, ale to załatwia sprawę. Teraz dodajmy ten element do listy istniejących elementów:

function appendEventToList (event: TechEvent) {

const list = document.querySelector ('# lista-zdarzeń')

element const = getTeaserListElement (zdarzenie)

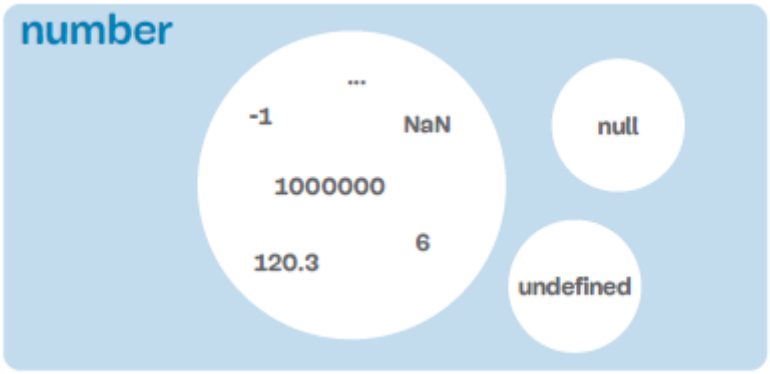
list.append (element)

}

I tu jest problem: musimy być bardzo pewni, że identyfikator exist-list istnieje w naszym kodzie HTML. W przeciwnym razie document.querySelector zwraca wartość null, a dołączenie listy spowoduje przerwanie aplikacji.

Ścisłe zerowe kontrole

Ponieważ null jest częścią wszystkich typów, powyższy kod jest zarówno prawidłowy, jak i wysoce toksyczny. Prosta zmiana w naszych znacznikach i cała aplikacja się psuje. Potrzebujemy sposobu, aby upewnić się, że wynik działania document.querySelector jest rzeczywiście dostępny i nie jest null. Oczywiście możemy sprawdzić wartość null lub użyć fantazyjnego operatora „Elvis" (?. Znanego również jako opcjonalne łańcuchowanie), ale czy nie byłoby wspaniale, gdyby TypeScript powiedział nam aktywnie, że powinniśmy to zrobić? Jest sposób. W twoim tsconfig.json możemy aktywować opcję strictNullChecks (która jest częścią trybu ścisłego). Po aktywowaniu tej opcji wszystkie wartości zerowe zostaną wykluczone z naszych typów.



Numer typu ze ścisłymi kontrolami null.

Z null i undefined nie będącymi częścią rzeczywistego zestawu typów, ten fragment kodu spowoduje błąd w czasie kompilacji:

let age: number

age = age + 1

age nie jest w końcu określony! Jednak metoda strictNullChecks nie zmienia sposobu działania document.querySelector. Wynik nadal może być zerowy. Ale typem zwracania selektora document.query jest Element | null, typ unii z wartością zerową! I to sprawia, że TypeScript natychmiast rzuca w nas czerwoną falistą:

function appendEventToList (event: TechEvent) {

const list = document.querySelector ('# lista-zdarzeń')

element const = getTeaserListElement (zdarzenie)

list.append (element)

}

list jest prawdopodobnie null. Jak dobry jest TypeScript. Szybkie sprawdzenie null (operator Elvisa tańczący przed nami) załatwia sprawę i sprawia, że nasz kod jest dużo bezpieczniejszy:

function appendEventToList(event: TechEvent) {

const list = document.querySelector('#event-list')

const element = getTeaserListElement(event)

list?.append(element) // Optional chaining / Null check

}

TypeScript idzie nawet trochę dalej. Przy włączonych ścisłych kontrolach NullCheck nie tylko musimy sprawdzać wartości zerowe, ale nie możemy również przypisywać wartości undefined lub null do zmiennych i właściwości. Obie wartości są usuwane ze wszystkich typów, więc przypisanie tego rodzaju jest zabronione. Są sytuacje, w których musimy pracować z wartością undefined lub null. Aby przywrócić jedną (lub obie) wartości z powrotem do miksu, musimy dodać je do unii; na przykład string | undefined. To sprawia, że dodawanie wartości zerowych jest jawne i musimy sprawdzić ich istnienie.

type Talk = {

title: string,

speaker: string,

abstract: string | undefined

}

Innym sposobem dodania undefined jest uczynienie właściwości obiektu opcjonalnymi. Należy również sprawdzić właściwości opcjonalne, ale bez utrzymywania zbyt wielu typów.

type Talk = {

title: string,

speaker: string,

abstract?: string

}

W każdym razie, jak powiedział Douglas Crockford, po co nam dwie zerowe wartości? Jeśli musisz użyć jednego, trzymaj się jednego z nich.

Podsumowanie

Ta część dotyczyła hierarchii typów, teorii zbiorów, typów górnych i dolnych oraz wartości zerowych, które mogą zepsuć nasze programy. Wszystko, czego nauczyliśmy się w zakresie typów związków i skrzyżowań, ma kluczowe znaczenie dla wszystkiego, co nadchodzi. Kiedy już nauczysz się poruszać w przestrzeni tekstowej, TypeScript ma wiele do zaoferowania.

1. Dowiedzieliśmy się o typach sum i przecięć oraz o tym, jak możemy modelować dane, które mogą przybierać różne kształty.

2. Dowiedzieliśmy się również, jak typy związków i przecięć działają w przestrzeni typów. Dowiedzieliśmy się również o rozróżnianiu związków i typów wartości.

3. Dowiedzieliśmy się o kontekście const i znaleźliśmy sposoby na dynamiczne tworzenie innych typów poprzez wyszukiwanie i typy mapowane.

4. Zbudowaliśmy własne predykaty typów jako osłony typu niestandardowego.

5. Typ dolny nigdy nie jest świetny do wyczerpujących sprawdzeń w ramach instrukcji switch lub if - else.

6. Na koniec poradziliśmy sobie z null i undefined i prawie się ich pozbyliśmy.

Jedną rzeczą, która jest teraz dla nas drugą naturą, są typy poszerzania i zawężania. Możemy przejść od wszystkiego, co obejmuje any, do typu bez wartości, never. W przestrzeni czcionek możemy swobodnie poruszać się po wszystkich znanych nam typach. Teraz dowiedzmy się, co zrobić z typami, których kształtów nie znamy