Lekcja 23: Poruszanie się w przestrzeni typów

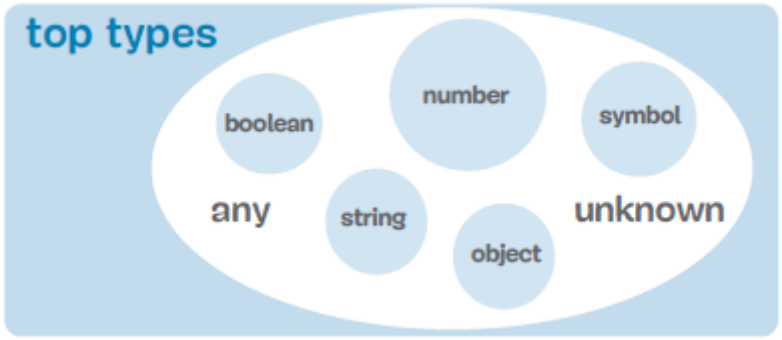
Zanim przejdziemy dalej, przejrzyjmy szybko to, czego się właśnie dowiedzieliśmy. Dowiedzieliśmy się o typach przecięć, sposobie łączenia dwóch lub więcej typów w jeden, podobnie jak w przypadku typu obiektu. Dowiedzieliśmy się też o typach związków, sposobie wyodrębnienia najniższego wspólnego mianownika zestawu typów. Ale dlaczego nazywamy je typami przecięć i związków?

Teoria zbiorów

Aby się tego dowiedzieć, musimy sprawdzić, jakie faktycznie są typy. W swojej książce Programming with Types Vlad Riscutia definiuje typ w następujący sposób:

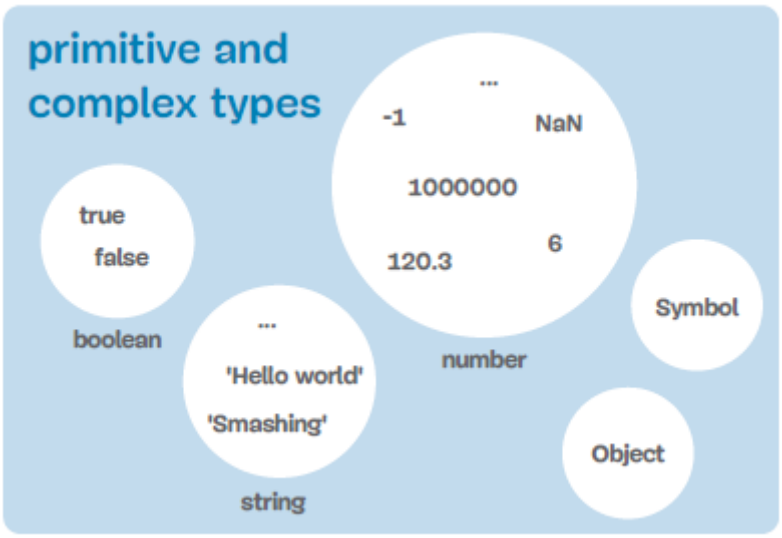
Typ to klasyfikacja danych, która definiuje operacje, które można wykonać na tych danych, znaczenie danych i zbiór dozwolonych wartości.

Część, na której chcemy się skupić, to „zbiór dozwolonych wartości”. To jest coś, czego już doświadczyliśmy podczas pracy z typami. Gdy zmienna ma adnotację określonego typu, TypeScript umożliwia przypisanie tylko określonego zestawu wartości. Typ string pozwala tylko na przypisanie łańcuchów znaków; number pozwala tylko na przypisanie liczb. Każdy typ dotyczy odrębnego zestawu wartości. Kiedy myślimy dalej, możemy ustawić te zestawy w hierarchii. Typy any i unknown obejmują cały zestaw wszystkich dostępnych wartości. Są one znane jako najwyższe typy, ponieważ znajdują się na samym szczycie hierarchii.



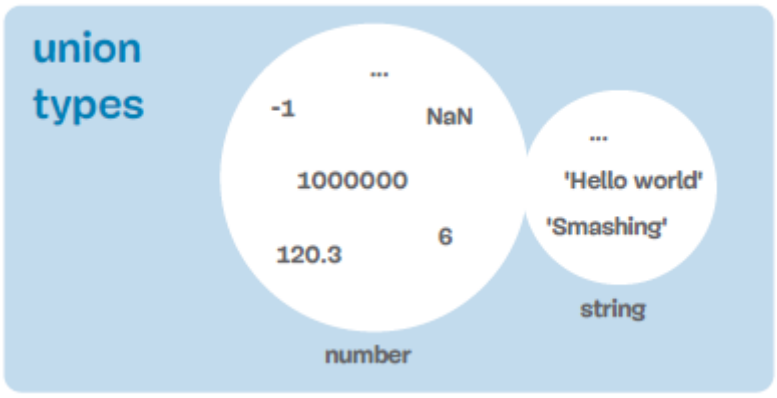
Najpopularniejsze typy

Typy pierwotne, takie jak boolean, number lub string, są o jeden poziom poniżej any i unknown. Grupują zbiór wszystkich dostępnych wartości w odrębne zbiory określonych wartości: wszystkie wartości logiczne, wszystkie liczby, wszystkie ciągi.



Zbiory typów pierwotnych i złożonych.

Te ziory są różne. Nie wyznają żadnych wspólnych wartości. Jeśli teraz utworzymy typu unii string | number, dopuszczamy wszystkie wartości, które pochodzą albo z ustalonego string, albo z ustalonej number, co oznacza, że otrzymujemy sumę możliwych wartości.



Gdybyśmy mieli utworzyć string & number typu przecięcia, mielibyśmy ustawione puste przecięcie, ponieważ nie mają one wspólnych wartości. Stąd też pochodzi określenie zawężenie. Chcemy mieć węższy zestaw wartości. Jeśli nasz typ to any, możemy wykonać sprawdzenie typeof, aby zawęzić do określonego zestawu w przestrzeni typów. Przechodzimy od najwyższego typu do węższego zestawu wartości.

Zbiory obiektów

W przypadku typów prymitywnych jest to proste, ale staje się o wiele przyjemniejsze, jeśli weźmiemy pod uwagę typy obiektów. Weźmy na przykład te dwa typy:

type Name = {

name: string

}

type Age = {

age: number

}

Ponieważ mamy system typów strukturalnych, obiekt taki jak

const person = {

name: 'Jam Kowalski',

city: 'Lincz'

}

jest prawidłową wartością typu Person. Ten obiekt

// In my midlife crisis, I don't use semicolons

// ... just like the cool kids

const midlifeCrisis = {

age: 38,

usesSemicolons: false

}

jest prawidłową wartością typu Age. Ten obiekt

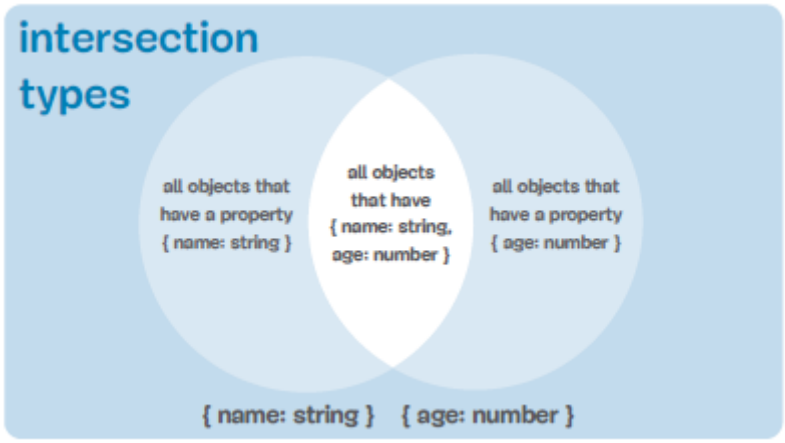
const me = {

name: Jam Kowalski',

age: 38

}

jest zgodny zarówno z Age , jak i Name. Nie możemy jednak przypisać każdej wartości typu Age do typu Name, ponieważ zbiory są na tyle różne, że nie mają żadnych wspólnych wartości. Po zdefiniowaniu typu unii Age | Name, zarówno midlifeCrisis, jak i person są zgodne z nowo utworzonym typem. Zbiór jest coraz szerszy, liczba zgodnych wartości rośnie. Ale tracimy też jasność. I odwrotnie, przecięcie type Person = Age & Name łączy oba zestawy. Teraz potrzebujemy wszystkich właściwości z typu Age i typu Name.



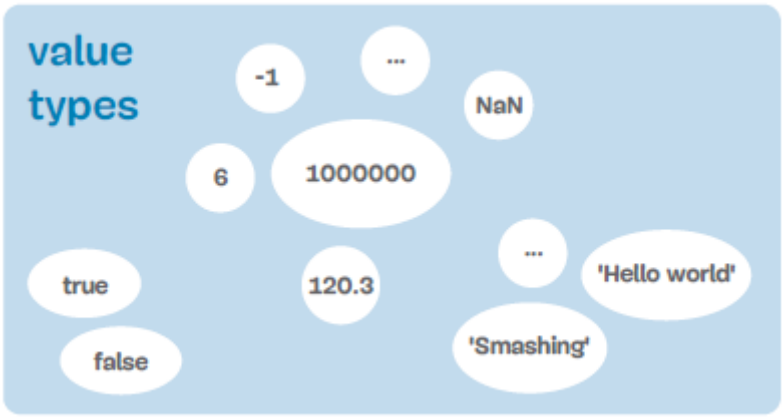
Skrzyżowanie imienia i wieku

Dzięki temu tylko zmienna me staje się zgodna z nowo wygenerowanym typem. Skrzyżowanie jest podzbiorem zarówno zbioru Age , jak i Name - mniejszych, węższych i musimy bardziej precyzyjnie określić nasze wartości. Formalnie rzecz biorąc, wszystkie wartości typu A są zgodne z typem A | B, a wszystkie wartości z typu A i B są zgodne z typem B.

Typy wartości

Przejdźmy jeszcze dalej do koncepcji zawężania i poszerzania zbiorów. Teraz wiemy, że możemy mieć wszystkie dostępne wartości i zawęzić je do ich pierwotnych typów. Możemy zawęzić typy złożone, takie jak zbiór wszystkich dostępnych obiektów, do mniejszych zbiorów możliwych obiektów zdefiniowanych w ich kluczach właściwości. Czy możemy być jeszcze mniejsi?

Możemy! Możemy zawęzić typy pierwotne do wartości. Okazuje się, że każda określona wartość zbioru jest swoim własnym typem: typem wartości.



I wreszcie typy wartości.

Przyjrzyjmy się na przykład ciągowi „conference”.

let conf = 'conference'

Nasza zmienna conf jest kompatybilna z kilkoma typami:

let withTypeAny: any = 'conference' // OK!

let withTypeString: string = 'conference' // OK!

// But also:

let withValueType: 'conference' = 'conference'

// OK!

Widzisz, że zbiór staje się węższy i węższy. Typ any wybiera wszystkie możliwe wartości, typ string wszystkie możliwe ciągi. Ale wpisanie „conference” wybiera określony ciąg znaków „conference”. Żadne inne ciągi nie są kompatybilne. TypeScript rozpoznaje typy wartości podczas przypisywania wartości pierwotnych:

// Type is string, because the value can change

let conference = 'conference'

// Type is 'conference', because the value can't

// change anymore.

const conf = 'conference'

Teraz, gdy zawęziliśmy zestaw do typów wartości, możemy ponownie tworzyć szersze zestawy niestandardowe. Wróćmy do naszego przykładu wydarzeń technicznych. Mamy trzy różne rodzaje wydarzeń technicznych: konferencje, seminaria internetowe i spotkania. Kiedy nasz zaplecze wysyła szczegółowe informacje o rodzajach zdarzeń, z którymi mamy do czynienia, możemy utworzyć niestandardowy typ unii:

type EventKind = 'webinar' | 'conference' | 'meetup'

Dzięki temu możemy mieć pewność, że nie przypisujemy wartości, które nie są zamierzone, a także wykluczamy literówki i inne błędy.

// Super, ale niemożliwe

let tomorrowsEvent: EventKind = 'concert'

Zbiory wartości typów pierwotnych są technicznie nieskończone. Nigdy nie bylibyśmy w stanie w rozsądny sposób wyrazić pełnego spektrum string lub number w niestandardowym typie. Ale możemy wyciągnąć z niego bardzo specyficzne wycinki, jeśli jest zgodny z naszymi danymi. Kiedy zagłębiamy się w system typów TypeScript, często poszerzamy i zawężamy zbiory. Poruszanie się po zbiorach możliwych wartości jest kluczem do definiowania jasnych, ale elastycznych typów, które dają nam narzędzia pierwszej klasy.