BEN GRYNHAUS | JORDAN HUDGENS
RAYON HUNTE | MATT MORGAN
WEKOSLAV STEFANOVSKI

TYPESCRIPT
NA WARSZTACIE

PRAKTYCZNY PRZEWODNIK
PISANIA EFEKTYWNEGO KODU

Packt›

Helion

Tytuł oryginału:The TypeScript Workshop: A practical guide to confident, effective TypeScript programming

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-8951-9

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/typesc
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

6

ROZDZIAŁ

Typy zaawansowane

W TYM ROZDZIALE

Ten rozdział wprowadza typy zaawansowane. Najpierw zapoznasz się z elementami konstrukcyjnymi typów zaawansowanych: aliasami typów oraz literałami tekstowymi i liczbowymi. Pozwoli Ci to lepiej zrozumieć bardziej złożone koncepcje, takie jak typy unii. Dowiesz się również, jak łączyć typy, by budować bardziej złożone, takie jak typy części wspólnej. W tym rozdziale na bazie typów zaawansowanych nauczysz się pisać kod łatwiejszy do zrozumienia dla Ciebie i innych pracujących z Tobą osób, a także osób, które mogą odziedziczyć określony projekt. Po lekturze tego rozdziału będziesz w stanie budować typy zaawansowane przez łączenie z obiektami typów prostych, takich jak `string`, `number` czy `Boolean`.

Wprowadzenie

W poprzednim rozdziale omówiliśmy interfejsy i dziedziczenie. Dowiedziałeś się, że umożliwiają one rozszerzanie i modelowanie klas. Interfejsy zapewniają klasom strukturę, a dziedziczenie pozwala rozszerzać i rozbudowywać istniejący kod.

Ponieważ aplikacje internetowe stają się coraz bardziej złożone, konieczne jest modelowanie tej złożoności, a w TypeScripcie jest to łatwiejsze dzięki typom zaawansowanym. Typy zaawansowane umożliwiają modelowanie złożonych danych, z którymi będziesz pracować jako nowoczesny programista aplikacji internetowych. Będziesz mógł na bazie typów prostych tworzyć bardziej złożone typy, które będą warunkowe i elastyczne. W ten sposób będziesz pisać prosty do zrozumienia kod, z którym łatwiej będzie pracować. Jako aktywny zawodowo programista możesz natknąć się na dostarczane przez API zbiory danych, które będziesz musiał

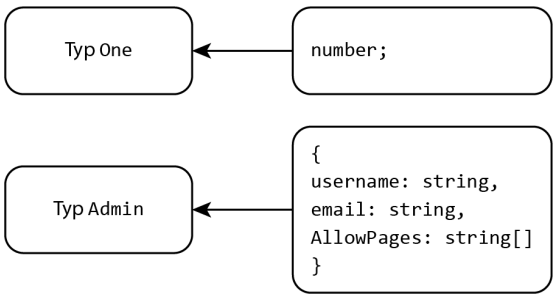
zintegrować ze swoją aplikacją. Mogą to być złożone zbiory danych. Przykładem może być Cloud Firestore firmy Google, czyli dokumentowa baza danych czasu rzeczywistego, która może zawierać obiekty zagnieżdżone w innych obiektach. Dzięki typom zaawansowanym możesz utworzyć typ, który będzie dokładną reprezentacją danych pochodzących z tego API. Zapewni to znacznie szerszy kontekst dla kodu, co z kolei ułatwi Tobie i Twojemu zespołowi pracę z tym kodem. Ponadto będziesz mógł układać złożoność w stosy — budować prostsze typy i układać je w stos, aby tworzyły typy bardziej złożone.

W tym rozdziale omówimy elementy konstrukcyjne typów zaawansowanych: aliasy typów i literały typów. Gdy nauczysz się budować typy, przejdziemy do bardziej zaawansowanych koncepcji, takich jak typy unii, typy części wspólnej oraz typy indeksowe. Wszystkie te koncepcje pomogą Ci nauczyć się, jak używać typów zaawansowanych, aby dodawać kontekst i tworzyć warstwę abstrakcji dla złożoności kodu.

Aliasy typów

Aliasy typów umożliwiają deklarowanie referencji do dowolnego typu — zaawansowanego lub prostego. Sprawiają, że kod staje się czytelniejszy, ponieważ można pisać go bardziej zwięźle. Aliasy pozwalają programistom na jednorazowe zadeklarowanie typu i ponowne wykorzystywanie go w całej aplikacji. Dzięki temu praca z typami złożonymi jest prostsza, a kod łatwiejszy w czytaniu i utrzymywaniu.

Załóżmy, że pracujemy nad aplikacją społecznościową i musimy zapewnić typ użytkownika administratora, aby użytkownicy mogli zarządzać utworzonymi przez siebie stronami. Ponadto musimy zdefiniować też użytkownika, który będzie administratorem strony. Na poziomie bazowym obaj użytkownicy będą administratorami, a zatem te typy będą miały między ze sobą pewne podobieństwa. Za pomocą aliasów typów moglibyśmy utworzyć typ administratora, jak pokazaliśmy na rysunku 6.1, z typowymi właściwościami, które posiadałby użytkownik administrator, i oprzeć się na tym podczas tworzenia typów administratora strony i użytkownika administratora. Aliasy pozwalają zamaskować złożoność kodu, co ułatwia jego zrozumienie. Na rysunku 6.1 możesz zobaczyć przypisanie aliasu `Admin` do typu administratora, który jest złożonym obiektem `type`. Zobaczysz również przykład aliasu `One` przypisanego do typu `number`, który jest typem prostym.



Rysunek 6.1. Przypisanie aliasu do złożonego typu administratora

Rozważmy poniższy fragment kodu:

```
// Przypisanie typu prostego
type One = number;
```

Utworzyliśmy w nim alias `One`, który może być używany jako typ dla dowolnej liczby, ponieważ został przypisany do typu `number`.

Zobaczmy teraz następujący fragment kodu:

```
// Przypisanie obiektu złożonego
type Admin = {
    username: string,
    email: string,
    userId: string,
    AllowedPages: string
};
```

W tym kodzie utworzyliśmy alias `Admin`, który przypisaliśmy do obiektu reprezentującego typowe właściwości administratora w kontekście tego przykładu. Jak widać, utworzyliśmy referencję do obiektu `type`, którego możemy teraz używać w naszym kodzie zamiast każdorazowego implementowania tego obiektu.

Jak pokazaliśmy na rysunku 6.1 i w powyższych fragmentach kodu, aliasy typów działają podobnie jak przypisania zmiennych, z tym wyjątkiem, że tworzona jest referencja do typu prostego i (lub) obiektu. Ta referencja może być następnie używana jako szablon dla danych. Umożliwia to skorzystanie ze wszystkich zalet języka silnie typowanego, takich jak uzupełnianie kodu i sprawdzanie poprawności danych.

Zanim przejdziesz do pierwszego ćwiczenia z aliasami typów, przyjrzymy się kilku przykładom przypisań prostych i złożonych.

Załóżmy, że pracujemy nad metodą klasową, która jako argumenty ma przyjmować wyłącznie liczby. Chcemy zapewnić, że gdy metoda będzie używana, jako argumenty będą przekazywane tylko liczby, a jeśli przekazany zostanie inny typ, użytkownik zobaczy odpowiednie komunikaty o błędach.

Najpierw musimy utworzyć alias typu liczbowego, korzystając z następującej składni:

```
type OnlyNumbers = number;
```

Po słowie kluczowym `type` następuje alias `OnlyNumbers`, a potem typ `number`.

Teraz możemy zbudować klasę z metodą, która jako argument przyjmuje tylko liczby, i do wyegzekwowania tej reguły możemy użyć aliasu typu:

```
// Instancja klasy NumbersOnly (tylko liczby)
class NumbersOnly {
    count: number

    SetNumber(someNumber: OnlyNumbers) {
```

```
            this.count = someNumber
        }
    }
```

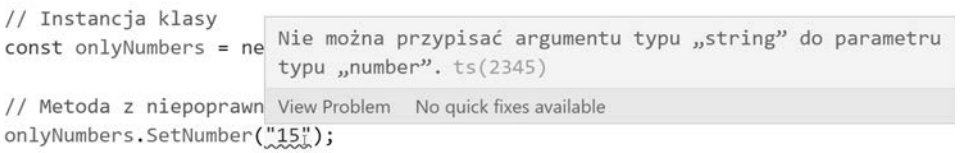
Utwórzmy instancję klasy i przekażmy do jej metody kilka argumentów, aby sprawdzić, czy kod działa.

Spróbujmy jako typ argumentu przypisać np. łańcuch znaków:

```
// Instancja klasy
const onlyNumbers = new NumbersOnly;

// Metoda z niepoprawnymi argumentami
onlyNumbers.SetNumber("15");
```

W powyższym fragmencie kodu podaliśmy niewłaściwy argument typu `string`, co spowoduje wyświetlenie ostrzeżenia, ponieważ metoda `SetNumber` oczekuje liczby. Ponadto definiowanie aliasów typów o znaczących nazwach, takich jak `onlyNumbers` (tylko liczby), ułatwia odczytywanie i debugowanie kodu. Na rysunku 6.2 pokazaliśmy, że po najechaniu kursorem na podkreślony nieprawidłowy fragment kodu zostaje wyświetlony bardzo pomocny komunikat o błędzie z informacją, na czym polega problem i jak można go rozwiązać.



Rysunek 6.2. Komunikat o błędzie w VS Code

Warunkiem wyświetlenia pokazanego na rysunku 6.2 komunikatu jest odpowiednie wsparcie ze strony IDE. Jeśli Twoje IDE nie zapewnia takiego wsparcia, otrzymasz komunikat o błędzie podczas kompilacji kodu.

Jest to prosty przypadek użycia, ale gdy aplikacje będą się rozrastać, upłynie trochę czasu od napisania kodu lub będziesz pracować w dużym zespole, tego rodzaju zabezpieczenia są niezbędne do pisania wolnego od błędów kodu.

Rozważmy kolejny przykład. Załóżmy, że pracujemy nad aplikacją sklepu internetowego i potrzebujemy użyć klasy produktu, która nie została utworzona przez nas. Jeśli twórca tej klasy korzystał z typów i stosował opisowe nazwy, łatwiej będzie nam pracować z tym kodem.

Wróćmy teraz do pierwszego przykładu i podajmy poprawny typ argumentu:

```
// Metoda z poprawnymi argumentami
onlyNumbers.SetNumber(15);
```

W powyższym fragmencie kodu podaliśmy właściwy typ argumentu `number`, a metoda klasy przyjęła ten argument bez żadnych problemów.

Przyjrzyjmy się złożonym przypisaniom aliasów.

Chcemy utworzyć np. nową funkcję, która jako argument typu przyjmuje obiekt użytkownika. Moglibyśmy zdefiniować ten obiekt wewnątrz kodu jako argument funkcji w następujący sposób:

```
// Definicja funkcji i typu
function badCode(user: {
    email: string,
    userName: string,
    token: string,
    lastLogin: number
}) {}
```

Powyższy fragment kodu powoduje utworzenie funkcji, która jako argument przyjmuje użytkownika, ale typ jest zdefiniowany w samej funkcji. Będzie to działać, ale jeśli mielibyśmy używać tego obiektu w kilku miejscach w kodzie, musielibyśmy definiować go za każdym razem. Jest to bardzo nieefektywne, a jako dobry programista nie chcesz powtarzać kodu. Ten sposób pracy prowadzi ponadto do wkradania się błędów. Utrudnia też pracę z kodem i jego aktualizację, ponieważ trzeba zmieniać każdą instancję typu `User` w całym kodzie. Aliasy typów rozwiązują ten problem, umożliwiając jednorazowe definiowanie typu, co zademonstrujemy w kolejnym fragmencie kodu.

Typ `User` zdefiniowaliśmy w podobny sposób, jak zrobiliśmy to w przypadku typu prostego. Użyliśmy słowa kluczowego `type`, ale tym razem dokonaliśmy zmapowania na obiekt, który jest szablonem typu `User`. Możemy teraz używać aliasu `User` zamiast ponownie deklarować ten obiekt za każdym razem, gdy musimy zdefiniować typ `User`:

```
// Obiekt — typ złożony User
type User = {
    email: string,
    userName: string,
    token: string,
    lastLogin: number
};
```

Jak widać, utworzyliśmy typ z aliasem `User`. Dzięki temu możemy jednorazowo odwołać się do tego typu obiektu i wykorzystywać go ponownie w całym kodzie. Gdybyśmy tego nie zrobili, musielibyśmy odwoływać się do tego typu bezpośrednio.

Możemy teraz zbudować nową funkcję, używając typu `User`:

```
// Funkcja z aliasem typu
function goodCode(user: User){}
```

Ten kod jest znacznie bardziej zwięzły i zrozumiały. Cały kod dotyczący typu `User` znajduje się w jednym miejscu, a w przypadku wprowadzenia zmian w obiekcie zostaną zaktualizowane wszystkie aliasy. W następnym ćwiczeniu zaimplementujesz to, co omówiliśmy do tej pory, aby zbudować własne aliasy typów.

Ćwiczenie 6.01. Implementowanie aliasów typów

W tym ćwiczeniu wykorzystasz swoją wiedzę o typach do zbudowania funkcji, która tworzy produkty. Załóżmy, że pracujesz nad aplikacją do robienia zakupów. Gdy menedżer magazynu doda do asortymentu jakiś produkt, musisz umieścić go w swojej tablicy produktów. To ćwiczenie pokazuje, że aliasy typów mogą być przydatne na kilka sposobów, umożliwiając jednokrotne zdefiniowanie modelu `Product` i ponowne wykorzystywanie go w całym kodzie.

W rzeczywistej aplikacji do zarządzania zapasami możesz mieć frontendową stronę, która będzie umożliwiać użytkownikowi ręczne wprowadzanie nazwy produktu i informacji pomocniczych. Na potrzeby tego ćwiczenia załóżmy, że produkty, które chcesz dodać, noszą nazwy od `Product_0` do `Product_5` i wszystkie mają cenę 100, a do zapasów ma być dodane po 15 sztuk każdego z tych produktów.

Może to nie odzwierciedlać dokładnie rzeczywistego scenariusza z aplikacji do zarządzania zapasami, ale pamiętaj, że Twoim głównym celem jest użycie aliasów typów. Na razie do wykonania powyższych zadań wystarczy prosta pętla `for`.

> Wszystkie pliki z tego rozdziału możesz uruchomić przez wykonanie w terminalu polecenia `npx ts-node` *`nazwa_pliku`*`.ts`.

1. Uruchom VS Code i utwórz nowy plik *Cwiczenie01.ts*.
2. Utwórz alias `Count` (licznik) typu prostego `number`. `Count` będzie używany do śledzenia liczby produktów:

```
// Typ prosty
type Count = number;
```

3. Utwórz alias `Product` dla typu obiektu `type`. Wykorzystaj ponownie `Count`, aby zdefiniować liczbę produktów. Alias typu `Product` będzie używany do definiowania każdego produktu dodawanego do zapasów. Te właściwości są wspólne dla wszystkich produktów:

```
// Typ obiektowy
type Product = {
    name: string,
    count: Count, // Ponowne wykorzystanie Count
    price: number,
    amount:number,
}
```

4. Zadeklaruj zmienną `products_list` tablicy typu `Product`:

```
// Tablica produktów
const products_list: Product[] = [];
```

   Abyś mógł korzystać z typu `Product`, został on w powyższym kodzie przypisany do zmiennej `products_list`, która jest tablicą obiektów typu `Product`.

5. Utwórz funkcję, która dodaje produkty do tablicy. Wykorzystaj ponownie aliasu typu `Product` do walidacji danych wejściowych argumentu:

```
// Dodawanie produktów do funkcji tablicy produktów
function makeProduct(p : Product ) {
    products_list.push(p); // Dodawanie produktu na końcu tablicy
}
```

6. Użyj pętli `for`, aby utworzyć obiekty produktów typu `Product` i dodać je do tablicy `products_list`:

```
// Użycie pętli for do utworzenia 5 produktów
for (let index = 0; index < 5; index++) {
    let p : Product = {
        name: "Produkt"+"_"+`${index}`,
        count: index,
        price: 100,
        amount: 15
    } // Tworzenie produktu
    makeProduct(p);
}

console.log(products_list);
```

7. Skompiluj i uruchom program, wykonując polecenie `npx ts-node Cwiczenie01.ts` z poziomu właściwego katalogu, w którym znajduje się ten plik. Powinieneś otrzymać następujące dane wyjściowe:

```
[
  { name: 'Produkt_0', count: 0, price: 100, amount: 15 },
  { name: 'Produkt_1', count: 1, price: 100, amount: 15 },
  { name: 'Produkt_2', count: 2, price: 100, amount: 15 },
  { name: 'Produkt_3', count: 3, price: 100, amount: 15 },
  { name: 'Produkt_4', count: 4, price: 100, amount: 15 }
]
```

W tym ćwiczeniu utworzyłeś dwa aliasy typów, które z kolei utworzyły referencje do rzeczywistych typów.

Pozwoliło to zmniejszyć złożoność kodu i uczynić go bardziej czytelnym, ponieważ teraz możesz podawać opisowe nazwy, takie jak `Product` i `products_list`, które mają dodatkowy kontekst. Gdybyśmy musieli pisać ten kod bez użycia aliasów, to w każdym miejscu, w którym w ćwiczeniu użyliśmy aliasów, musielibyśmy bezpośrednio definiować obiekt lub typ. W przypadku tej prostej funkcji być może nie stanowiłoby to większego problemu, ale pamiętaj, o ile więcej kodu potrzeba, aby zbudować klasę lub duży projekt.

Gdy przejdziemy do bardziej złożonych struktur typów, ta wiedza okaże się nieoceniona. W kolejnym podrozdziale poszerzysz swoją wiedzę, gdyż poznasz literały typów.

Literały typów

Literały typów umożliwiają tworzenie typów na podstawie określonego łańcucha znaków lub liczby. Samo w sobie nie jest to zbyt przydatne, ale gdy przejdziemy do bardziej złożonych typów, takich jak typy unii, ich zastosowanie stanie się oczywiste. Literały są

stosunkowo proste, więc nie będziemy poświęcać im dużo czasu, ale musisz zrozumieć tę koncepcję, zanim przejdziemy dalej.

Utwórzmy najpierw literały: tekstowy i liczbowy.

Zaczniemy od literału tekstowego.

Plik *Przyklad01.ts*

```
1  // Literał tekstowy
2  type Yes = "yes";
```

Powyższy kod powoduje utworzenie typu `Yes`, który jako dane wejściowe przyjmuje tylko określony łańcuch znaków `"yes"`.

W podobny sposób możemy utworzyć literał liczbowy:

```
3  // Literał liczbowy
4  type One = 1;
```

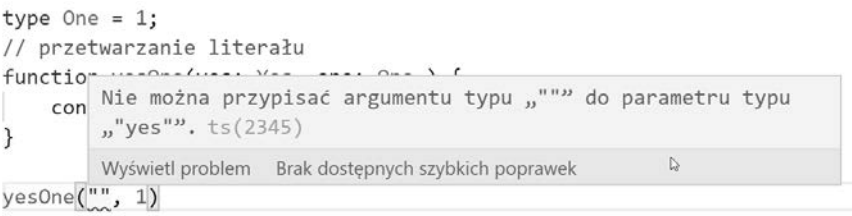
W tym kodzie utworzyliśmy typ literału liczbowego `One`, który jako dane wejściowe przyjmuje tylko liczbę `1`.

Podstawowa składnia zaprezentowana w tych przykładach jest dość prosta. Zaczynamy od słowa kluczowego `type`, po którym następuje nazwa (alias) nowego literału, a potem sam literał. Mamy teraz typy łańcucha znaków `yes` i liczby `1`.

Następnie zbudujemy funkcję, która wykorzysta nowe typy:

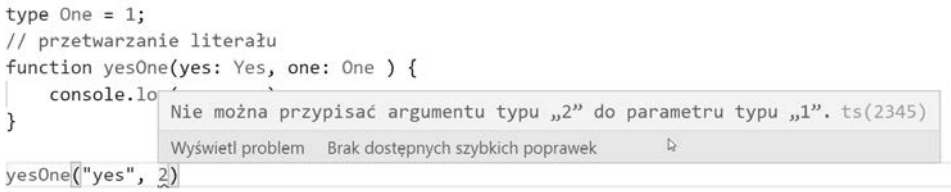
```
5  // Przetwarzanie literału
6  function yesOne(yes: Yes, one: One ) {
7      console.log(yes, one);
8  }
```

Przeprowadziliśmy rzutowanie (konwersję) argumentów funkcji na typy literałów, a ponieważ nasze typy są literałami, jako argumenty będą akceptowane tylko łańcuch znaków `"yes"` lub liczba `1`. Nasza funkcja nie będzie przyjmować innych argumentów. Załóżmy, że jako argumenty przekazaliśmy do tej funkcji `""` i `2` w następujący sposób: (`yesOne("", 2)`). W VS Code wyświetlone zostanie ostrzeżenie, które pokazaliśmy na rysunku 6.3.



Rysunek 6.3. Ostrzeżenie wyświetlane przez IDE w przypadku przekazania nieprawidłowych argumentów

Załóżmy, że teraz przekazaliśmy jako argumenty `"yes"` i `2`. Tym razem otrzymamy ostrzeżenie, które pokazaliśmy na rysunku 6.4.



Rysunek 6.4. Błędy wyświetlane po przekazaniu parametru, którego nie można przypisać

Zaprezentowaliśmy kilka przykładów komunikatów o błędach, których możesz się spodziewać, jeśli podasz nieprawidłowe argumenty. Te komunikaty są czytelne i dokładnie informują, co należy zrobić, aby usunąć błąd. Jak widać, mimo że przekazujemy łańcuch znaków i liczbę, nadal otrzymujemy błąd typu. Dzieje się tak, ponieważ te argumenty są literałami; muszą być całkowicie zgodne.

Spróbujmy przekazać teraz poprawne argumenty:

```
// Funkcja z poprawnymi argumentami
yesOne("yes", 1);
```

Po dostarczeniu poprawnych argumentów funkcja może zostać wywołana bez żadnych problemów, jak pokazaliśmy w poniższych danych wyjściowych:

```
yes 1
```

Zanim przejdziemy do typów części wspólnej, wykonasz proste ćwiczenie, które utrwali Twoją wiedzę na temat literałów tekstowych i liczbowych.

Ćwiczenie 6.02. Literały typów

Skoro znasz już literały typów, wykonaj krótkie ćwiczenie, aby utrwalić wiedzę. Utworzysz funkcję, która przyjmuje literał tekstowy i zwraca literał liczbowy.

1. Uruchom VS Code i utwórz nowy plik *Cwiczenie02.ts*.
2. Utwórz typ literał tekstowy `No` i przypisz mu jako wartość łańcuch znaków `"no"`. Utwórz także literał liczbowy i przypisz mu jako wartość liczbę `0`:

```
type No = "no"
type Zero = 0
```

3. Zbuduj funkcję, która przyjmuje literał `"No"` i wypisuje go w konsoli:

```
function onlyNo(no: No):Zero {
    return 0;
}
```

4. Wypisz w konsoli wynik wywołania funkcji:

```
console.log(
    onlyNo("no")
)
```

Spowoduje to wyświetlenie następujących danych wyjściowych:

```
0
```

Literały same w sobie nie są zbyt użyteczne, ale w połączeniu z bardziej złożonymi typami ich przydatność stanie się oczywista. Na razie musisz zrozumieć, jak tworzyć literały, aby móc z nich skorzystać dalej w tym rozdziale. W następnym podrozdziale przejdziemy do typów części wspólnej. Cała wykonana do tej pory praca będzie pomocna, ponieważ będziemy korzystać z aliasów i literałów typów.

Typy części wspólnej

Typy części wspólnej (ang. *intersection type*) umożliwiają łączenie typów w celu utworzenia nowego typu z właściwościami połączonych typów. Jest to przydatne, gdy mamy istniejący typ, który sam w sobie nie odnosi się do pewnych danych, które musimy zdefiniować, ale może to robić w połączeniu z innym istniejącym typem. Przypomina to dziedziczenie wielopoziomowe, ponieważ obiekt potomny może mieć więcej niż jeden obiekt nadrzędny, z którego wywodzi swoje właściwości.

Załóżmy, że mamy typ `A` z właściwościami `name` (nazwa) i `age` (wiek). Mamy również typ `B` z właściwościami `height` (wzrost) i `weight` (waga). Stwierdzamy, że w aplikacji potrzebny jest typ `Person` (osoba): chcemy śledzić nazwę, wiek, wzrost i wagę użytkownika. Możemy utworzyć część wspólną typów `A` i `B`, aby powstał typ `Person`. Możesz zapytać, dlaczego nie utworzyć po prostu nowego typu? Odpowiemy w ten sposób: chcemy być dobrymi programistami, a dobrzy programiści stosują się do zasady DRY — nie powtarzaj się. Jeśli dany typ nie jest naprawdę unikatowy w aplikacji, powinien wykorzystywać ponownie jak największą ilość kodu. Do tego dochodzi centralizacja.

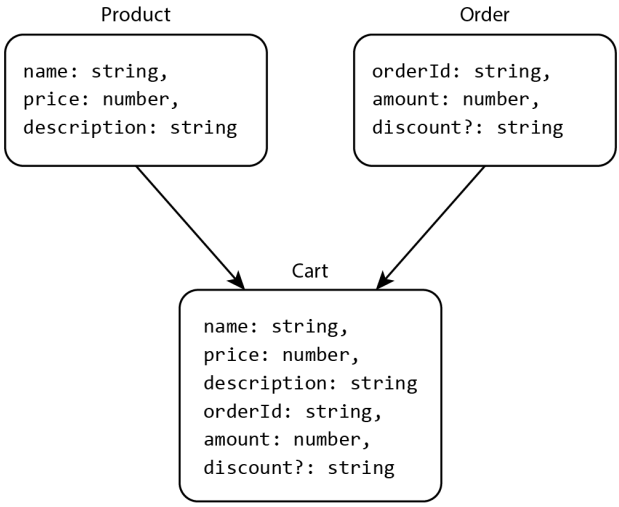
Jeżeli będziemy potrzebowali wprowadzić jakieś zmiany w kodzie typu `Person`, wystarczy wprowadzić zmiany w `A` lub `B`. Jest to również nieco ograniczające, ponieważ w niektórych przypadkach typ `A` może być używany przez więcej niż jeden obiekt i jeśli wprowadzimy zmiany, zepsuje to aplikację. Dzięki części wspólnej można utworzyć typ `C` ze zmianami i zaktualizować typ `Person`. Można także scalać typy ze wspólnymi właściwościami.

Rozważmy sytuację, w której mamy właściwość `name` zarówno w `A`, jak i w `B`. Po utworzeniu części wspólnej będziemy mieli tylko jedną właściwość `name`. Scalane właściwości muszą mieć jednak nie tylko taką samą nazwę, ale powinny być również tego samego typu. W przeciwnym razie typy nie zostaną scalone i spowoduje to błędy.

Jeśli nie jest to do końca jasne, przyjrzyjmy się właściwości `age`. W jednym typie może to być liczba, a w drugim łańcuch znaków. Jedynym sposobem na utworzenie części wspólnej tych typów byłoby ujednolicenie właściwości, aby obie były łańcuchem znaków lub liczbą.

Wyobraź sobie, że w ramach projektu e-commerce musisz zbudować obiekt koszyka (`Cart`), który wywodzi swoje właściwości z obiektów `Product` (produkt) i `Order` (zamówienie).

Na rysunku 6.5 przedstawiliśmy podstawowe właściwości każdego obiektu oraz właściwości nowego obiektu `Cart` utworzonego przy użyciu obiektów `Product` i `Order`.



Rysunek 6.5. Schemat przedstawiający właściwości obiektu Cart

Na rysunku 6.5 pokazaliśmy obiekty nadrzędne `Product` i `Order`, które łączą się, tworząc obiekt potomny `Cart` ze wszystkimi właściwościami obiektów nadrzędnych. Zwróć uwagę, że w części wspólnej możemy mieć więcej niż dwa obiekty nadrzędne, ale do celów tego wyjaśnienia będziemy trzymać się dwóch, ponieważ pozwoli to szybciej zrozumieć koncepcję. W kolejnym przykładzie omówimy proces tworzenia nowego typu `Cart` w kodzie oraz podstawowy przypadek użycia.

Wyobraź sobie, że pracujemy nad aplikacją zakupową. Musimy utworzyć obiekt do modelowania danych produktu, który będzie wysyłany do koszyka w celu zdjęcia ze stanu magazynowego. Mamy już typ `Product` dla danych produktu. Typ `Product` obejmuje już większość tego, co jest potrzebne, aby wyświetlać na stronie internetowej prawidłowe informacje dotyczące produktu. Brakuje jednak kilku rzeczy niezbędnych podczas zdejmowania produktu ze stanu magazynowego. Nie będziemy w tym celu tworzyć nowego typu produktu. Utworzymy natomiast typ `Order` zawierający tylko te właściwości, których potrzebujemy: `orderId` (identyfikator zamówienia), `amount` (kwota) i `discount` (rabat). Ta ostatnia będzie opcjonalna, ponieważ nie zawsze będzie miała zastosowanie.

Oto kod do zadeklarowania typu `Product`:

Plik *Przyklad02.ts*

```
// Typ produktu
type Product = {
    name: string,
    price: number,
    description: string
}

// Typ zamówienia
type Order = {
    orderId: string,
    amount: number,
    discount?: number
}
```

W powyższym fragmencie kodu utworzyliśmy typy nadrzędne o nazwach `Product` i `Order`. Teraz musimy je scalić. Spowoduje to utworzenie typu, którego potrzebujemy do modelowania danych koszyka:

```
// Alias Cart części wspólnej typów Product i Order
type Cart = Product & Order;
```

Obiekt koszyka budujemy przez przypisanie aliasu `Cart` do typów `Product` i `Order` oraz użycie między nimi operatora `&`, jak pokazaliśmy w powyższym fragmencie kodu. Mamy teraz nowy, scalony typ `Cart`, którego możemy użyć do modelowania danych koszyka:

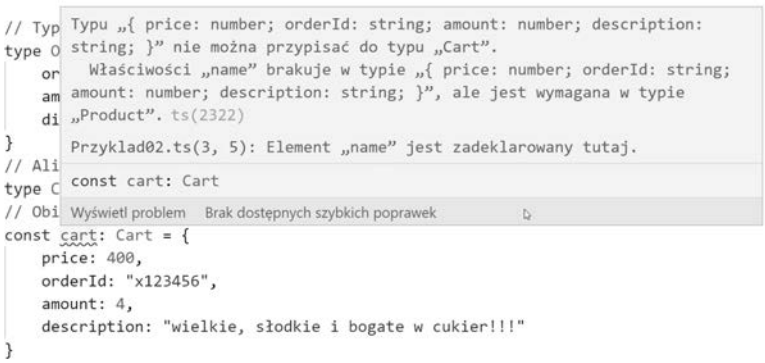
```
// Obiekt cart typu Cart
const cart: Cart = {
    name: "Mango",
    price: 400,
    orderId: "x123456",
    amount: 4,
    description: "wielkie, słodkie i bogate w cukier!!!"
}
```

Pokazaliśmy przykład zadeklarowania obiektu koszyka przy użyciu typu `Cart`. Jak widać, mamy dostęp do wszystkich właściwości i możemy pominąć opcjonalne, takie jak `discount`, które nie zawsze mogą mieć zastosowanie.

Jak pokazaliśmy na rysunku 6.6, jeśli nie określimy wszystkich wymaganych właściwości, IDE wyświetli bardzo pomocny komunikat o błędzie, który informuje, co trzeba zrobić w celu rozwiązania problemu.

Wypiszmy teraz w konsoli nowy obiekt `cart`. Spowoduje to wyświetlenie następujących danych wyjściowych:

```
{
  name: 'Mango',
  price: 400,
  orderId: 'x123456',
```



Rysunek 6.6. Komunikat o błędzie wyświetlany w przypadku braku wymaganych właściwości

```
  amount: 4,
  description: 'wielkie, słodkie i bogate w cukier!!!'
}
```

W następnym ćwiczeniu zdobędziesz praktyczne doświadczenie w tworzeniu typów części wspólnej, budując prototypowy system zarządzania użytkownikami.

Ćwiczenie 6.03. Tworzenie typów części wspólnej

Pracujesz nad aplikacją e-commerce. Przydzielono Ci zadanie zbudowania systemu zarządzania użytkownikami. W wymaganiach aplikacji klient określił typy profilów użytkowników, które będą wchodzić w interakcję z systemem. Aby zbudować typy użytkowników, użyjesz typów części wspólnej. Umożliwi to separację zagadnień oraz budowanie prostych typów, które można łączyć w celu tworzenia typów bardziej złożonych. Dzięki temu kod będzie mniej podatny na błędy i otrzyma lepsze wsparcie. Oto typy użytkowników, które zbudujesz, i przegląd ich funkcji:

- *Użytkownik podstawowy* (typ `User`) — będzie miał właściwości `_id`, `email` i `token`.
- *Użytkownik administracyjny* (typ `Admin`) — będzie miał dostęp do stron niedostępnych dla zwykłego użytkownika. Ten użytkownik będzie miał właściwości `accessPages` i `lastLogin`. Właściwość `accessPages` będzie tekstową tablicą stron dostępnych dla tego użytkownika, a `lastLogin` pomoże rejestrować jego aktywności.
- *Użytkownik kopii zapasowej* (typ `Backup`) — jego zadaniem będzie tworzenie kopii zapasowej systemu. Ten użytkownik będzie miał właściwości `lastBackUp` i `backUpLocation`. Właściwość `lastBackUp` informuje, kiedy ostatnio wykonano kopię zapasową systemu, a `backUpLocation` wskazuje lokalizację zapisania plików kopii zapasowej.
- Typ `superUser` — będzie częścią wspólną typów `Admin` i `User`. Wszyscy użytkownicy wymagają właściwości użytkownika podstawowego, ale tylko administratorzy wymagają właściwości użytkownika `Admin`. Aby zbudować niezbędne właściwości, użyjesz części wspólnej typów.

- Typ `BackUpUser` — będzie częścią wspólną typów `Backup` i `User`. Niezbędną złożoność, której ten typ użytkownika wymaga do funkcjonowania, po raz kolejny możesz dodać do użytkownika podstawowego.

1. Uruchom VS Code i utwórz nowy plik *Przyklad03.ts*.
2. Utwórz typ podstawowy `User`:

```
// Tworzenie typu obiektowego użytkownika
type User = {
    _id: number;
    email: string;
    token: string;
}
```

Tego typu będziesz używać jako bazy dla innych typów użytkowników w aplikacji. Dlatego ma on wszystkie typowe właściwości, których będą wymagać wszyscy użytkownicy.

3. Utwórz typ `Admin` dla użytkowników, którzy muszą pełnić funkcje administratora:

```
// Tworzenie typu obiektowego administratora
type Admin = {
    accessPages: string[],
    lastLogin: Date
}
```

4. Utwórz typ `Backup` dla użytkowników odpowiedzialnych za tworzenie kopii zapasowych danych aplikacji:

```
// Tworzenie typu obiektowego użytkownika kopii zapasowej
type Backup = {
    lastBackUp: Date,
    backUpLocation: string
}
```

5. Zadeklaruj obiekt `superUser` o typie części wspólnej typów `User` i `Admin`. Dodaj wymagane właściwości. Aby utworzyć superużytkownika, musisz podać wartości dla właściwości typów `User` i `Admin`:

```
// Połączenie użytkownika i administratora w celu utworzenia superużytkownika
const superUser: User & Admin = {
    _id: 1,
    email: 'rayon.hunte@gmail.com',
    token: '12345',
    accessPages: [
        'profile', 'adminConsole', 'userReset'
    ],
    lastLogin: new Date()
};
```

W rzeczywistej aplikacji ten kod może się znajdować w funkcji logowania, a zwracane wartości mogą pochodzić z interfejsu API logowania.

6. Zbuduj typ `BackUpUser`, przypisując alias `BackUpUser` do części wspólnej typów `User` i `Backup`:

```
// Tworzenie typu BackUpUser
type BackUpUser = User & Backup
```

7. Zadeklaruj obiekt `backUpUser` typu `BackUpUser` i dodaj wymagane właściwości:

```
// Tworzenie obiektu backUpUser
const backUpUser: BackUpUser = {
    _id: 2,
    email: 'rayon.backup@gmail.com',
    token: '123456',
    lastBackUp: new Date(),
    backUpLocation: '~/backup'
};
```

8. Wypisz w konsoli obiekty `superUser` i `backUpUser`:

```
// Wypisanie w konsoli właściwości obiektu superUser
console.log(superUser);

// Wypisanie w konsoli właściwości obiektu backUpUser
console.log(backUpUser);
```

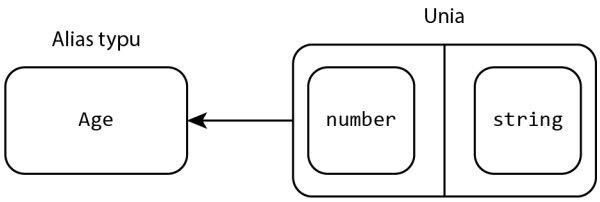
Spowoduje to wyświetlenie następujących danych wyjściowych:

```
{
  _id: 1,
  email: 'rayon.hunte@gmail.com',
  token: '12345',
  accessPages: [ 'profile', 'adminConsole', 'userReset' ],
  lastLogin: 2021-02-25T07:27:57.009Z
}
{
  _id: 2,
  email: 'rayon.backup@gmail.com',
  token: '123456',
  lastBackUp: 2021-02-25T07:27:57.009Z,
  backUpLocation: '~/backup'
}
```

W tym ćwiczeniu zbudowałeś dwa typy użytkowników przy użyciu części wspólnych `superUser` i `backUpUser`, które zostały oparte na typach `User`, `Admin` i `Backup`. Wykorzystanie przecięć pozwala zachować prostotę typu użytkownika podstawowego i modelować większość danych użytkownika. `Admin` i `Backup` utworzyły część wspólną z typem `User` tylko wtedy, gdy było to konieczne do wymodelowania konkretnego przypadku użytkownika. Na tym polega separacja zagadnień. Od tej pory wszelkie zmiany wprowadzane w typach `User`, `Backup` lub `Admin` będą odzwierciedlane we wszystkich typach potomnych. Przyjrzymy się teraz typom unii, które są funkcjonalnością typów. Jednak inaczej niż typy części wspólnej, typy unii podczas łączenia typów zapewniają funkcjonalność `OR`.

Typy unii

Typy unii (ang. *union type*) są podobne do części wspólnych, ponieważ są kombinacją typów, które tworzą pojedynczy typ. Różnią się od nich jednak tym, że nie scalają typów, lecz zapewniają funkcjonalność `OR` zamiast funkcjonalności `AND`. Przypomina to działanie operatora trójargumentowego w JavaScripcie, gdzie łączone typy są oddzielone znakiem potoku (|). Jeśli poczułeś się zdezorientowany, wszystko stanie się jasne, gdy przejdziemy do przykładu. Przyjrzymy się również strażnikom typów, czyli wzorcowi, który będzie odgrywał główną rolę w korzystaniu z typów unii w aplikacji. Najpierw przyjrzyjmy się wizualnej reprezentacji typu unii, którą pokazaliśmy na rysunku 6.7.



Rysunek 6.7. Ilustracja przypisania typu unii

Na rysunku 6.7 widać podstawowy schemat przypisania typu unii, gdzie `Age` (wiek) może być typem danych `number` lub `string`. Możemy mieć typy unii z więcej niż dwiema opcjami i typami innymi niż proste. Daje to możliwość pisania bardziej dynamicznego kodu. W kolejnym przykładzie rozszerzymy przykład `Age` i zbudujemy podstawowy typ unii.

Załóżmy, że pracujemy nad aplikacją, która musi sprawdzać czyjś wiek. Chcemy napisać jedną funkcję, która będzie przetwarzać wiek z bazy danych przechowywany jako liczba oraz wiek z interfejsu internetowego dostarczany w postaci łańcuch znaków. W takim przypadku moglibyśmy pokusić się o użycie `any` jako typu. Unie pozwalają nam jednak obsłużyć tego rodzaju scenariusz bez tworzenia wektora dla błędów w wyniku zastosowania `any`:

Plik *Przyklad03.ts*

```
1  // Podstawowy typ unii
2  type Age = number | string;
```

Najpierw tworzymy typ unii `Age`, który może być typem danych `number` lub `string`, jak pokazaliśmy w składni powyższego kodu. Przypisujemy alias `Age` do typów, które rozdzielamy znakiem potoku (|). Moglibyśmy mieć więcej niż dwie opcje, np. `number | string | object`.

Teraz tworzymy funkcję, która będzie korzystać z nowego typu `Age`:

```
3  function myAge(age: Age): Age {
4      if (typeof age === "number") {
5          return `mój wiek to ${age} i jest to typ number`;
6      } else if (typeof age === "string"){
7          return `mój wiek to ${age} i jest to typ string`;
```

```
 8      } else {
 9          return `niepoprawny typ" ${typeof(age)}`;
10      }
11  }
```

Funkcja `myAge` przyjmuje jako argument typ `Age` i przy użyciu pętli `if-else` zwraca sformatowany łańcuch znaków typu `Age`. Używamy również wzorca strażnika typów, `typeof`, który pozwala sprawdzać typ argumentu. Ten rodzaj sprawdzania typów jest konieczny podczas korzystania z typów unii, ponieważ mamy kilka dopuszczalnych typów argumentu — w przypadku powyższego fragmentu kodu może to być `number` lub `string`. Każdy typ będzie musiał być przetwarzany według innej logiki.

Typy unii mogą być także obiektami, jednak w takim przypadku `typeof` nie będzie zbyt przydatny, ponieważ zwróci tylko typ, którym zawsze będzie `object`. W takich sytuacjach możesz sprawdzać dowolne unikatowe właściwości obiektu i w ten sposób zastosować odpowiednią logikę. Przykłady zobaczysz podczas pracy nad ćwiczeniem w następnym punkcie.

Wróćmy teraz do naszego przykładu. Aby upewnić się, że funkcje działają tak, jak powinny, wypisujemy wyniki w konsoli, wywołując funkcje z różnymi typami argumentów (`number` i `string`):

```
console.log(myAge(45));
console.log(myAge("45"));
```

Spowoduje to wypisanie następujących danych wyjściowych:

```
mój wiek to 45 i jest to typ number
mój wiej to 45 i jest to typ string
```

Załóżmy, że przekazaliśmy niepoprawny argument:

```
console.log(myAge(false));
```

Wyświetlony zostanie następujący komunikat o błędzie:

```
error TS2345: Argument of type 'boolean' is not assignable to parameter
of type 'Age'.
```

Ćwiczenie 6.04. Aktualizowanie zapasów produktów za pomocą interfejsu API

W poniższym ćwiczeniu rozszerzysz przykład zarządzania zapasami z ćwiczenia 6.01, dodając interfejs API. Umożliwi to zdalnym użytkownikom dodawanie i aktualizowanie produktów za pomocą żądania `PUT` lub `POST` interfejsu API.

Ponieważ procesy aktualizowania i dodawania produktu są bardzo podobne, napiszesz jedną metodę do obsługi obu żądań i użyjesz typu unii, aby umożliwić metodzie przyjmowanie obu typów przy jednoczesnym zachowaniu bezpieczeństwa typów. Oznacza to również, że możesz napisać mniej kodu i zhermetyzować cały powiązany kod w jednej metodzie, co ułatwi Ci (lub każdemu innemu programiście pracującemu nad tą aplikacją) znajdowanie i usuwanie błędów.

Mógłbyś użyć typu `any`, ale wtedy kod nie zapewniałby bezpieczeństwa typów, co mogłoby prowadzić do powstawania błędów i niestabilności kodu.

1. Uruchom VS Code i utwórz nowy plik *Przyklad04.ts*.
2. Utwórz trzy typy, `Product`, `Post` i `Put`, wraz z bazowymi obiektami, których będziesz potrzebować. Pokazaliśmy to w poniższym kodzie:

```
type Product = {
    name: string,
    price: number,
    amount: number,
}

type Post = {
    header: string,
    method: string,
    product: Product
}

type Put = {
    header: string,
    method: string,
    product: Product,
    productId: number
}
```

   Najpierw utworzyłeś typ dla produktu (`Product`), który pomoże Ci określić, jaki format będą przyjmować dane produktu w ramach żądania `Put` lub `Post`. Zdefiniowałeś `Put` i `Post`, które nieco się różnią, ponieważ żądanie `Put` będzie wymagało aktualizacji rekordu, który już istnieje. Zwróć uwagę, że `Put` ma właściwość `productId`.

3. Utwórz typ unii `SomeRequest`, który może być typem `Put` lub `Post`:

```
type SomeRequest = Post | Put
```

   Dane dopasowywane do typu unii mogą być dowolnymi typami z unii. Zwróć uwagę, że unie nie łączą typów, lecz próbują dopasować dane do jednego z typów z unii, co zapewnia programiście większą elastyczność.

4. Utwórz instancję tablicy typu `Product`:

```
const products: Product[] = [];
```

5. Zbuduj funkcję procedury obsługi, która będzie przetwarzać żądanie typu `SomeRequest`:

```
function ProcessRequest(request: SomeRequest ) {
    if ("productId" in request) { products.forEach(
            (p: Product, i: number) => {
               products[request.productId] = {
                   ...request.product
               };});
    } else {
        products.push(request.product);
    }}
```

Ta funkcja będzie odbierać żądanie typu `Put` lub `Post` i dodawać załączony produkt do tablicy `products` lub go aktualizować. Aby funkcja mogła zdecydować, czy powinna zaktualizować czy dodać produkt, najpierw sprawdza, czy dany produkt ma argument `productId`. Jeśli tak, wykonujesz pętlę przez tablicę `products`, aż znajdziesz pasujący argument `productId`. Następnie korzystasz z operatora rozwinięcia, aby zaktualizować dane produktu przy użyciu danych z żądania. Jeżeli produkt nie ma argumentu `productId`, używasz dołączonej do tablicy funkcji `push`, aby dodać nowy produkt do tablicy.

6. Zadeklaruj obiekty `apple` i `mango` typu `Product`:

```
const apple: Product = {
    name: "jabłko",
    price: 12345,
    amount: 10
};
const mango: Product = {
    name: "mango",
    price: 66666,
    amount: 15
};
```

W prawdziwym interfejsie API dane byłyby dostarczane przez użytkownika wysyłającego je za pośrednictwem żądania, ale na potrzeby tego ćwiczenia zakodowaliśmy dane, z którymi możesz pracować.

7. Zadeklaruj obiekty `postAppleRequest` i `putMangoRequest` typu odpowiednio `Post` i `Put`:

```
const postAppleRequest : Post = {
    header: "zzzzz",
    method: 'new',
    product: apple,
};

const putMangoRequest : Put = {
    header:"ggggg",
    method: 'update',
    product: mango,
    productId: 2
};
```

W powyższym kodzie zdefiniowałeś obiekty `POST` i `PUT`. Załączyłeś obiekt produktu jako ładunek żądania. Pamiętaj, że funkcja nie sprawdza obiektu produktu, lecz typ żądania, który informuje ją, czy jest to żądania `POST` czy `PUT`.

8. Wywołaj funkcję procedury obsługi i przekaż jako argumenty `postAppleRequest` i `putMangoRequest`:

```
ProcessRequest(postAppleRequest);

ProcessRequest(putMangoRequest);
```

W normalnym API, gdy użytkownik wykonywałby żądanie `PUT` lub `POST`, wywoływana byłaby metoda `ProcessRequest`. Jednak Ty po prostu symulujesz API i sam wykonujesz wywołania.

9. Wypisz wyniki w konsoli:

```
console.log(products)
```

Zobaczysz następujące dane wyjściowe:

```
[
  { name: 'jabłko', price: 12345, amount: 10 },
  <1 pusty element>,
  { name: 'mango', price: 66666, amount: 15 }
]
```

W powyższych danych wyjściowych możesz zobaczyć teraz produkty, które przekazałeś do metod. Oznacza to, że symulowany kod API korzystający z unii działa zgodnie z oczekiwaniami.

Typy unii, podobnie jak typy części wspólnej, zapewniają programiście szerszą funkcjonalność i większą elastyczność podczas tworzenia aplikacji. W tym ćwiczeniu napisałeś funkcję, która przyjmuje jeden argument dwóch różnych typów i stosuje logikę opartą na wzorcach sprawdzania typów lub strażników typów. W następnym rozdziale będziemy kontynuować temat zwiększania elastyczności kodu przez stosowanie typów indeksowych.

Typy indeksowe

Typy indeksowe pozwalają tworzyć obiekty, które zapewniają elastyczność w zakresie liczby posiadanych przez nie właściwości. Załóżmy, że mamy typ definiujący komunikat o błędzie, który może być więcej niż jednego typu, i chcemy mieć możliwość dodawania z czasem kolejnych typów komunikatów. Ponieważ obiekty mają ustaloną liczbę właściwości, musielibyśmy wprowadzać zmiany w kodzie komunikatów za każdym razem, gdy pojawiałby się nowy typ wiadomości. Typy indeksowe umożliwiają zdefiniowanie sygnatury dla typu za pomocą interfejsu, dzięki czemu liczba właściwości może być elastyczna. W poniższym przykładzie rozwiniemy to w kodzie.

Plik *Przyklad04.ts*

```
interface ErrorMessage {
    // Może być tylko typem string | number | symbol
    [msg: number ]: string;
    // Można dodawać kolejne właściwości, o ile będą tego samego typu
    apiId: number
}
```

Jak pokazaliśmy w powyższym fragmencie kodu, najpierw tworzymy sygnaturę typu. Mamy tutaj nazwę i typ właściwości, czyli indeks `[msg: number]`, po którym następuje typ wartości. Nazwa argumentu `msg` może być dowolna, ale jako dobry programista powinieneś zastosować nazwę, która ma sens w kontekście danego typu. Pamiętaj, że indeks może być tylko liczbą, łańcuchem znaków lub symbolem.

Do tego indeksu możemy dodać również inne właściwości, ale muszą one być tego samego typu co indeks — jak pokazaliśmy w powyższym fragmencie kodu, `apiId: number`. Następnie wykorzystujemy nasz typ, rzutując (przekształcając) go na `errorMessage`. Możemy mieć teraz obiekt komunikatu o błędzie z dowolną liczbą właściwości. Nie ma potrzeby modyfikowania typu wraz z wydłużaniem się listy komunikatów. Zachowujemy elastyczność, utrzymując jednocześnie typowanie kodu, co ułatwia skalowanie i zapewnianie wsparcia:

```
// Obiekt komunikatu typu indeksowego ErrorMessage
const errorMessage: ErrorMessage = {
    0: "błąd systemowy",
    1: "przeciążenie",
    apiId: 12345
};
```

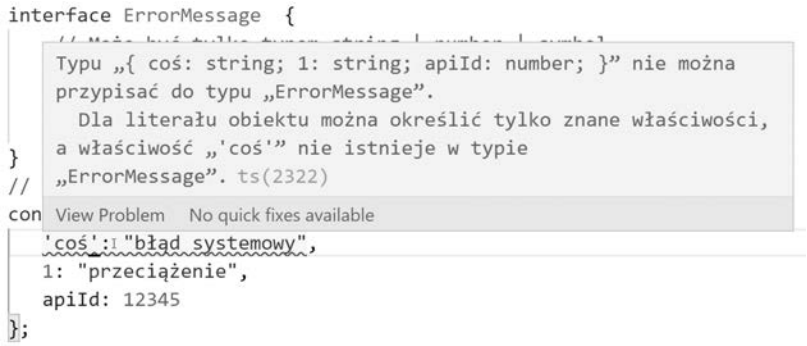
Wypisujemy nowy obiekt w konsoli, aby upewnić się, że wszystko działa:

```
// Wypisanie obiektu w konsoli
console.log(
    errorMessage
);
```

Po uruchomieniu tego pliku otrzymamy następujące dane wyjściowe:

```
{ '0': 'błąd systemowy', '1': 'przeciążenie', apiId: 12345 }
```

Jeśli spróbujemy podać nazwę właściwości niepoprawnego typu, takiego jak łańcuch znaków, otrzymamy spodziewany komunikat o błędzie, który pokazaliśmy na rysunku 6.8.



Rysunek 6.8. Wyświetlanie błędu typu

Możesz jednak użyć np. łańcuchów znaków, które są liczbami, a kod będzie działał jak poprzednio i wynik będzie taki sam:

```
// Obiekt komunikatu typu indeksowego ErrorMessage
const errorMessage: ErrorMessage = {
   '0': "błąd systemowy",
    1: "przeciążenie",
    apiId: 12345 };
```

Możesz pomyśleć, że to nie zadziała, biorąc pod uwagę, że wartość jest łańcuchem znaków, ale zostanie przekonwertowana na literał liczbowy. Będzie to również działać w drugą stronę, gdy użyjemy literału liczbowego, który zostanie przekonwertowany na łańcuch znaków. W następnym ćwiczeniu zasymulujesz praktyczne użycie typu indeksowego, budując prosty system do przetwarzania komunikatów o błędach.

Ćwiczenie 6.05. Wyświetlanie komunikatów o błędach

W tym ćwiczeniu zbudujesz system do przetwarzania komunikatów o błędach. Ponadto wykorzystasz ponownie typ indeksowy `ErrorMessage`, który utworzyliśmy w poprzednim przykładzie. Kod tego ćwiczenia jest nieco naciągany, ale pomoże Ci lepiej zrozumieć typy indeksowe.

1. Uruchom VS Code i utwórz nowy plik *Przyklad05.ts*.
2. Jeśli jeszcze tego nie zrobiłeś, utwórz interfejs typu `ErrorMessage` z poprzedniego przykładu:

```
interface ErrorMessage {
    // Może być tylko typu string | number | symbol
    [msg: number ]: string;
    // Można dodawać kolejne właściwości, o ile będą tego samego typu
    apiId: number
}
```

3. Zbuduj obiekt `errorCodes` jako typ `ErrorMessage`:

```
const errorMessage : ErrorMessage = {
    400:"złe żądanie",
    401:"brak autoryzacji",
    403:"niedozwolone", apiId: 123456,
};
```

4. Utwórz tablicę kodów błędów jako `errorCodes`:

```
const errorCodes: number [] = [
    400,401,403
];
```

5. Wykonaj pętlę przez tablicę `errorCodes` i wypisz w konsoli komunikaty o błędach:

```
errorCodes.forEach(
    (code: number) => {
        console.log(
            errorMessage[code]
        );
    }
);
```

Po uruchomieniu tego pliku otrzymasz następujące dane wyjściowe:

```
złe żądanie
brak autoryzacji
niedozwolone
```

Typy indeksowe zapewniają elastyczność w definiowaniu typów, o czym przekonałeś się w tym ćwiczeniu. Jeśli zechcesz dodać nowe kody, nie będziesz musiał zmieniać definicji typu — po prostu dodasz do obiektu `errorCode` nową właściwość kodu. Typy indeksowe sprawdzają się tutaj, ponieważ pomimo tego, że właściwości obiektu są różne, wszystkie mają ten sam podstawowy układ — właściwość liczbową (klucz), po której następuje wartość tekstowa.

Skoro masz już elementy konstrukcyjne dla typów zaawansowanych, możesz wykonać kolejne zadania. W tych zadaniach musisz wykorzystać wszystkie umiejętności nabyte podczas lektury tego rozdziału.

Zadanie 6.01. Typ części wspólnej

Wyobraź sobie, że jesteś programistą pracującym nad funkcjonalnością konfiguratora ciężarówek dla niestandardowej strony internetowej z ciężarówkami. Będziesz musiał umożliwić klientom strony konfigurowanie różnych typów ciężarówek. W tym celu musisz utworzyć własny typ części wspólnej `PickUpTruck`, łącząc dwa typy: `Motor` i `Truck`. Będziesz mógł wtedy skorzystać z nowego typu `PickUpTruck` z funkcją, która zwraca ten typ i przeprowadza przy jego użyciu walidację danych wejściowych.

Oto kilka kroków, które pomogą Ci wykonać to zadanie:

1. Utwórz typ `Motor` zawierający kilka standardowych właściwości, które będziesz mógł wykorzystać ponownie same lub w połączeniu z innymi typami, aby opisać obiekt pojazdu. Jako punkt wyjścia możesz użyć następujących właściwości: `color` (kolor), `doors` (drzwi), `wheels` (koła) i `fourWheelDrive` (napęd na cztery koła).
2. Utwórz typ `Truck` o właściwościach charakterystycznych dla ciężarówki, np. `doubleCab` (podwójna kabina) i `winch` (wyciągarka).
3. Utwórz część wspólną dwóch typów, aby powstał typ `PickUpTruck`.
4. Zbuduj funkcję `TruckBuilder`, która zwraca typ `PickUpTruck`, a także przyjmuje `PickUpTruck` jako argument.
5. Wypisz w konsoli wartość zwracaną funkcji.
6. Po wykonaniu zadania powinieneś otrzymać następujące dane wyjściowe:

```
{
  color: 'czerwony',
  doors: 4,
  doubleCab: true,
  wheels: 4,
  fourWheelDrive: true,
  winch: true
}
```

Rozwiązanie tego zadania znajdziesz w „Dodatku" na końcu książki.

Zadanie 6.02. Typ unii

Firma logistyczna poprosiła Cię o opracowanie dla jej strony internetowej funkcjonalności, która pozwoli klientom wybierać sposób wysyłki paczek: drogą lądową lub powietrzną. W tym celu zdecydowałeś się użyć typów unii. Możesz zbudować własny typ unii `ComboPack`, który może być typu `LandPack` (paczka lądowa) lub `AirPack` (paczka lotnicza). Do typów paczek możesz dodać dowolne właściwości, które Twoim zdaniem będą charakterystyczne dla paczki. Rozważ również użycie jednego literału typu do identyfikacji paczki jako powietrzna lub lądowa oraz zastosowanie opcjonalnej właściwości etykiety. Następnie musisz zbudować klasę do przetwarzania paczek. Twoja klasa powinna mieć metodę identyfikowania typu paczki. Metoda będzie przyjmować argumenty typu `ComboPack` i używać literału właściwości do identyfikacji typu paczki i dodania prawidłowej etykiety: `air cargo` (ładunek lotniczy) lub `land cargo` (ładunek lądowy).

Oto kilka kroków, które pomogą Ci wykonać to zadanie:

1. Zbuduj typy `LandPack` i `AirPack`. Upewnij się, że masz literał identyfikujący typ paczki.
2. Skonstruuj typ unii `ComboPack`, który może być typu `LandPack` lub `AirPack`.
3. Utwórz klasę `Shipping` (wysyłka) do przetwarzania paczek. Upewnij się, że używasz literału do identyfikowania typów paczek i modyfikowania paczki za pomocą etykiety odpowiedniej dla jej typu.
4. Utwórz dwa obiekty paczki: typu `AirPack` i typu `LandPack`.
5. Utwórz instancję klasy `Shipping`, przetwórz nowe obiekty i zmodyfikowane obiekty wypisz w konsoli.

> Rozwiązanie tego zadania znajdziesz w „Dodatku" na końcu książki.

Zadanie 6.03. Typ indeksowy

Ponieważ wykonałeś kawał dobrej roboty, uruchamiając na stronie internetowej opcje wysyłki, firma potrzebuje teraz, abyś dodał funkcjonalność, która pozwoli klientom śledzić status ich paczek. Ważne jest, aby w miarę rozwoju firmy istniała możliwość dodawania nowych statusów przesyłek, a ponieważ metody wysyłki się zmieniają, przydałaby się elastyczność w tym zakresie.

Zdecydowałeś więc, że zbudujesz typ indeksowy `PackageStatus`, używając sygnatury interfejsu o właściwości `status` typu `string` oraz wartości typu `Boolean`. Następnie utworzysz typ `Package` z właściwościami charakterystycznymi dla paczki. Uwzględnisz również właściwość `packageStatus` typu `PackageStatus`. Typu `PackageStatus` będziesz używać do śledzenia trzech statusów paczki: `shipped` (wysłana), `packed` (spakowana) i `delivered` (dostarczona), które mogą mieć ustawione wartości jako `true` (prawda) lub `false` (fałsz). Potem zbudujesz klasę, która podczas inicjowania będzie przyjmować obiekt typu `Package`, a ponadto będzie

zawierała metodę zwracającą właściwość `status` oraz metodę aktualizującą właściwość `status`, która będzie przyjmować `status` jako łańcuch znaków i wartość `Boolean` jako stan. Metoda aktualizująca paczkę powinna również zwracać właściwość `packageStatus`.

Oto kilka kroków, które pomogą Ci wykonać to zadanie:

1. Zbuduj typ indeksowy `PackageStatus` przy użyciu interfejsu z właściwością `status` typu `string` i wartością typu `Boolean`.
2. Utwórz typ `Package`, który zawiera właściwość typu `PackageStatus` i kilka właściwości standardowej paczki.
3. Utwórz klasę do przetwarzania typu `Package`, która przyjmuje typ `Package` podczas inicjowania oraz ma metodę zwracającą właściwość `packageStatus` i metodę, która aktualizuje i zwraca właściwość `packageStatus`.
4. Utwórz obiekt `Package` o nazwie `pack`.
5. Utwórz instancję klasy `PackageProcess` przy użyciu nowego obiektu `pack`.
6. Wypisz w konsoli status `pack`.
7. Zaktualizuj status `pack` i wypisz w konsoli nowy status `pack`.

   Oczekiwane dane wyjściowe są następujące:

```
{ wysłana: false, spakowana: true, dostarczona: true }
{ wysłana: true, spakowana: true, dostarczona: true }
```

Rozwiązanie tego zadania znajdziesz w „Dodatku" na końcu książki.

Podsumowanie

W tym rozdziale omówiliśmy typy zaawansowane, które pozwalają wyjść poza typy podstawowe. W miarę jak aplikacje stają się coraz bardziej złożone, a frontend poszerza swoją funkcjonalność, złożoność modeli danych również wzrasta. W tym rozdziale pokazaliśmy, że typy zaawansowane TypeScriptu dają możliwość zaimplementowania silnego typowania, co pomaga w tworzeniu czystszych i bardziej niezawodnych aplikacji. Omówiliśmy elementy konstrukcyjne typów zaawansowanych — aliasy i literały typów — a następnie przeszliśmy do typów części wspólnej, typów unii i typów indeksowych. Zobaczyłeś kilka praktycznych przykładów, a potem wykonałeś ćwiczenia i zadania.

Masz teraz możliwość tworzenia złożonych typów, które pozwolą Ci budować typy dla nowoczesnych aplikacji i pisać kod, który będzie miał dobre wsparcie i będzie skalowalny. Dotarłeś do etapu, na którym masz już narzędzia do wykorzystania frameworków internetowych, takich jak Angular2 i React. Możesz nawet używać TypeScriptu po stronie serwera za pomocą Node.js. Nie zamyka to oczywiście kwestii typów zaawansowanych, gdyż ten temat jest dosyć szeroki, złożony i abstrakcyjny w implementacjach. Ten rozdział wyposażył Cię jednak w umiejętności potrzebne do rozpoczęcia tworzenia aplikacji z użyciem typów zaawansowanych.

TYPESCRIPT: SPRAWDŹ, CZY UMIESZ NAPISAĆ LEPSZY KOD!

TypeScript szybko stał się ulubionym językiem programowania zawodowców. Pozwala na tworzenie czystego, efektywnego i łatwego w utrzymaniu kodu, a także zastosowanie zaawansowanych konstrukcji programistycznych. Co więcej, pracę w tym języku uprzyjemniają liczne frameworki i biblioteki, które pozwalają na sprawną budowę dojrzałych aplikacji. Jednak z punktu widzenia początkującego programisty TypeScript ma inną ogromną zaletę: jest świetnym punktem startu dla każdego, kto chce pisać czytelny, łatwy do zrozumienia i mniej podatny na błędy kod.

Dzięki tej książce Twoja nauka programowania w TypeScripcie przebiegnie w maksymalnie sprawny i efektywny sposób. Położono w niej nacisk na praktykę, a objaśnienia teoretyczne ograniczono do faktycznie ważnych i przydatnych zagadnień, które ułatwią ugruntowanie najlepszych podstaw programowania. W licznych ćwiczeniach pokazano, jak stosować kluczowe koncepcje w aplikacjach produkcyjnych, używanych w rzeczywistości. Każdy rozdział kończy się zadaniem do samodzielnego wykonania, opracowanym tak, by umożliwić wypróbowanie poznanych treści w praktyce. Jeśli chcesz pisać kod w TypeScripcie na profesjonalnym poziomie, a równocześnie odczuwasz onieśmielenie na myśl o nauce nowego języka, dzięki temu podręcznikowi szybko osiągniesz swój cel!

W KSIĄŻCE MIĘDZY INNYMI:

- gruntowne podstawy języka TypeScript i pliki deklaracji
- funkcje, klasy i obiekty w TypeScripcie
- wstrzykiwanie zależności i zachowania asynchroniczne
- obietnice i programowanie asynchroniczne
- biblioteka React i jej zastosowanie

Ben Grynhaus od lat tworzy strony internetowe z wykorzystaniem różnych stosów technologicznych. Jakiś czas temu pracował nad kilkoma produktami w firmie Microsoft. **Jordan Hudgens** jest wszechstronnym programistą i założycielem DevCampu. Specjalizuje się w Ruby on Rails, bibliotece React, Vue.js i TypeScripcie. **Rayon Hunte** budował złożone aplikacje internetowe, takie jak system zarządzania pojazdami. W języku TypeScript tworzył złożone, skalowalne aplikacje internetowe. **Matt Morgan** jest inżynierem oprogramowania i architektem. Pracował z wieloma technologiami, takimi jak RDBMS, Java czy Node.js. **Wekoslav Stefanovski** od lat jest zawodowym programistą. Jego pasja to budowanie lepszych programów i kształtowanie lepszych programistów.

ISBN 978-83-283-8951-9