Lekcja 13: Typowanie klas

TypeScript może być postrzegany jako język programowania, który wymazuje JavaScript. Oprócz kodu JavaScript, który istnieje i jest prawidłowy, dodajemy kolejną warstwę informacji o typie, która znika po uruchomieniu kompilatora. Rozróżniamy deklaracje, które tworzą typy i elementy tworzące wartości. Typy są dostępne tylko w języku TypeScript. Dodają nową nazwę do istniejącej przestrzeni typów, używamy ich do narzędzi i znikają w momencie, gdy kompilujemy do JavaScript. Deklaracje tworzące wartość nadal istnieją w JavaScript. Funkcje, zmienne - rzeczy, które pozostają po kompilacji. Rozdzielenie tych dwóch światów bardzo pomaga podczas pracy z TypeScript, ponieważ można usunąć deklaracje tworzące typ i nadal mieć poprawny JavaScript, który wygląda prawie tak samo. Jest jednak jedna rzecz, która wpływa zarówno na przestrzeń tworzącą typy, jak i przestrzeń kreującą wartości: klasy.

Klasy w JavaScript

Począwszy od standardu ECMAScript 2015, JavaScript zawiera klasy jako alternatywną formę składniową dla funkcji konstruktora i wzorca prototypu. Oto klasa w czystym języku JavaScript, która stosuje rabat do jednego z naszych artykułów:

```
class Discount {

isPercentage

amount

constructor(isPercentage, amount) {

this.isPercentage = isPercentage

this.amount = amount

}

apply(article) {

if(this.isPercentage) {

article.price = article.price

- (article.price * this.amount)

} else {

article.price = article.price - this.amount

}

}

}

// A discount that shaves off 10 EUR

const discount = new Discount(false, 10)

discount.apply({

price: 39,
```

```
vat: 0.2,

title: 'Form Design Patterns'

})
```

Dzięki kilku dodatkowym adnotacjom typu możemy mieć odpowiednie narzędzia i upewnić się, że konstruujemy poprawne obiekty:

```
class Discount {

isPercentage: boolean

amount: number

constructor(

isPercentage: boolean,

amount: number) {

this.isPercentage = isPercentage

this.amount = amount

}

apply(article: Article) {

if(this.isPercentage) {

article.price = article.price

- (article.price * this.amount)

} else {

article.price = article.price - this.amount

}

}

}
```

W chwili, gdy tworzymy klasę, jest ona również dostępna w przestrzeni tekstowej:

```
let discount: Discount = new Discount(true, 0.2)
```

W przypadku niestandardowych typów obiektów zawsze opisujemy kształt obiektu i upewniamy się, że wszystkie wartości przekazane jako parametry lub przypisane do zmiennych pasują do tego kształtu.

Typowanie strukturalne z klasami

Ponieważ TypeScript jest strukturalnym systemem typów, bardziej interesuje nas również kształt obiektów tworzonych przez klasę, a nie samą klasę. Jaki jest więc kształt obiektu wygenerowanego przez klasę? Klasy składają się z dwóch części:

1. Funkcja konstruktora. Zdefiniowaliśmy naszą funkcję konstruktora tak, aby przyjmowała wartość logiczną isPercentage oraz liczbę określającą kwotę, którą chcemy zgolić. W momencie wywołania new Discount (true, 0.2) wywołujemy funkcję konstruktora.

2. Część druga to prototyp. To wszystko wokół funkcji konstruktora: dwa pola (isPercentage, amount) i funkcja do zastosowania rabatu do artykułu.

Prototyp definiuje kształt obiektu, który jest zwracany przez wywołanie konstruktora. Teraz, gdy znamy już kształt, możemy nawet przypisać regularnie generowane obiekty do zmiennej typu Discount.

```
let allProductsTwentyBucks: Discount = {

isPercentage: false,

amount: 20,

apply(article) {

article.price = 20

}

}
```

Jest to poprawne Discount, ponieważ kształt jest nienaruszony. Ale to ogromnie zmienia semantykę klasy Discount. Działa to również odwrotnie. Możemy zdefiniować typ obiektu i stworzyć nowy obiekt Discount za pomocą konstruktora:

```
type DiscountType = {

isPercentage: boolean,

amount: number,

apply(article: Article): void

}

let disco: DiscountType = new Discount(true, 0.2)
```

W systemie typu strukturalnego ważny jest tylko kształt. Nazwy to podstawa.

Rozszerzanie klas

Jedną z głównych cech klas jest to, że są rozszerzalne. Możesz wziąć istniejącą klasę i rozszerzyć ją, nadpisując i dodając funkcje.

```
/**

* This class always gives 20 %, but only if

* the price is not higher than 40 EUR

*/

class TwentyPercentDiscount extends Discount {

// No special constructor

constructor() {
```

```
// But we call the super constructor of
// Discount
super(true, 0.2)
}
apply(article: Article) {
if(article.price <= 40) {
super.apply(article)
}
}
}
```

Stworzyliśmy klasę rabatową, która zawsze obejmuje 20%, ale tylko wtedy, gdy cena artykułu jest niższa niż 40 euro. W tym szczególnym przypadku TwentyPercentDiscount ma taki sam kształt jak Discount, co oznacza, że ich deklaracja typu jest wymienna:

```
let disco1: Discount
= new TwentyPercentDiscount() // OK
let disco2: TwentyPercentDiscount
= new Discount(true, 0.3) // OK! Semantics changed!
```

Ale możemy zmienić kształt. Utwórzmy funkcję weryfikacji TwentyPercentDiscount:

```
class TwentyPercentDiscount extends Discount {
constructor() {
super(true, 0.2)
}
apply(article: Article) {
if(this.isValidForDiscount(article)) {
super.apply(article)
}
}
isValidForDiscount(article: Article) {
return article.price <= 40
}
}
```

Kształt się zmienił, co oznacza, że obowiązują te same zasady, co dla typów obiektów: jeśli dostępnych jest więcej właściwości, kontrakt kształtu jest spełniony; jeśli brakuje właściwości, kontrakt kształtu nie jest spełniony:

let disco1: Discount

= new TwentyPercentDiscount() // Still OK!

// Error! We miss the `isValidForDiscount`

// method

let disco2: TwentyPercentDiscount

= new Discount(true, 0.3)

Do tej pory klasy stały się podstawą JavaScript, zwłaszcza że frameworki oparte na komponentach w dużym stopniu polegają na klasach do definiowania komponentów. Klasy pisania na maszynie mogą być nieco zagmatwane, ponieważ łączą dwa światy tworzenia typów i tworzenia wartości, ale jeśli będziemy pamiętać o głównych zasadach strukturalnych systemów typów, są one równie łatwe w użyciu. I lepiej, żeby tak było. Klasy były jedną z pierwszych zabójczych funkcji języka TypeScript, która przeniosła ludzi ze światów C # i Java do języka JavaScript. TypeScript przedstawił jedną z pierwszych implementacji klasy ECMAScript jako pierwszy dowód koncepcji, że klasy mogą działać w JavaScript. Od tamtej pory składnia niewiele się zmieniła.