Lekcja 49: Rozszerzanie obiektu, część 2

W JavaScript możemy definiować właściwości obiektu w locie za pomocą Object.defineProperty. Jest to przydatne, jeśli chcemy, aby Twoje właściwości były tylko do odczytu lub podobne. Wróćmy do pierwszego przykładu w tej książce: obiekt pamięci o maksymalnej wartości, której nie należy nadpisywać:

```
const storage = {

currentValue: 0

}

Object.defineProperty(storage, 'maxValue', {

value: 9001,

writable: false

})

console.log(storage.maxValue) // 9001

storage.maxValue = 2

console.log(storage.maxValue) // Still 9001
```

defineProperty i deskryptory właściwości są bardzo złożone. Pozwalają nam robić wszystko z właściwościami, które zwykle są zarezerwowane dla obiektów wbudowanych. Dlatego są powszechne w większych bazach kodu. TypeScript ma mały problem z defineProperty:

```
const storage = {

currentValue: 0

}

Object.defineProperty(storage, 'maxValue', {

value: 9001,

writable: false

})

// Property 'maxValue' does not exist on type...

console.log(storage.maxValue)
```

Jeśli nie wpiszemy jawnie rzutowania, nie otrzymamy wartości maxValue dołączonej do typu storage. Jednak w prostych przypadkach możemy pomóc!

Podpisy potwierdzeń

W języku TypeScript 3.7 zespół wprowadził sygnatury asercji. Pomyśl o funkcji assertIsNum, w której możemy upewnić się, że jakaś wartość jest typu number, w przeciwnym razie zgłasza błąd. Jest to podobne do funkcji assert w Node.js:

```
function assertIsNum(val: any) {
```

```
if (typeof val !== “number”) {
throw new AssertionError(“Not a number!”);
}
}
function multiply(x, y) {
assertIsNum(x);
assertIsNum(y);
// At this point I'm sure x and y are numbers.
// If one assert condition is not true, this
// position is never reached.
return x * y;
}
```

Aby zachować zgodność z takim zachowaniem, możemy dodać sygnaturę potwierdzenia, która mówi TypeScript, że wiemy więcej o typie po tej funkcji:

```
function assertIsNum(val: any): asserts val is number {
if (typeof val !== “number”) {
throw new AssertionError(“Not a number!”);
}
}
```

Działa to bardzo podobnie do predykatów typów, ale bez przepływu kontroli struktury opartej na warunkach, takiej jak if lub switch.

```
function multiply(x, y) {
assertIsNum(x);
assertIsNum(y);
// Now also TypeScript knows that both x and y are numbers
return x * y;
}
```

Jeśli przyjrzymy się temu uważnie, zobaczymy, że sygnatury potwierdzeń mogą zmieniać typ parametru lub zmiennej w locie. To jest właśnie to, co robi Object.defineProperty.

Niestandardowe defineProperty

Zastrzeżenie: Poniższy pomocnik nie ma na celu być w 100% dokładnym ani kompletnym. Może zawierać błędy i może nie uwzględniać każdego przypadku krawędzi specyfikacji defineProperty. Może jednak radzić sobie wystarczająco dobrze z wieloma przypadkami użycia. Więc używaj go na własne

ryzyko! Podobnie jak w przypadku hasOwnProperty w ostatniej lekcji, tworzymy funkcję pomocniczą, która naśladuje oryginalną sygnaturę funkcji:

function defineProperty<

Obj extends object,

Key extends PropertyKey,

PDesc extends PropertyDescriptor>

(obj: Obj, prop: Key, val: PDesc) {

Object.defineProperty(obj, prop, val);

}

Pracujemy z trzema ogólnymi:

1. Obiekt, który chcemy zmodyfikować, typu Obj, który jest podtypem obiektu.

2. Wpisz Key, który jest podtypem PropertyKey (wbudowanym), więc string | numer | symbol.

3. PDesc, podtyp PropertyDescriptor (wbudowany). To pozwala nam zdefiniować właściwość ze wszystkimi jej cechami (zapisywalność, wyliczalność, rekonfigurowalność).

Używamy typów ogólnych, ponieważ TypeScript może zawęzić je do bardzo konkretnego typu jednostki. Na przykład PropertyKey to wszystkie liczby, ciągi znaków i symbole. Ale jeśli użyjemy Key extends PropertyKey, możemy wskazać właściwość, aby była (na przykład) typu „maxValue". Jest to przydatne, jeśli chcemy zmienić oryginalny typ, dodając więcej właściwości. Funkcja Object.defineProperty zmienia obiekt lub zgłasza błąd, jeśli coś pójdzie nie tak. Dokładnie to, co robi funkcja asercji. Nasz niestandardowy pomocnik defineProperty robi to samo. Dodajmy podpis potwierdzenia. Po pomyślnym wykonaniu defineProperty nasz obiekt ma inną właściwość. W tym celu stworzymy kilka typów pomocników. Podpis pierwszy:

function defineProperty<

Obj extends object,

Key extends PropertyKey,

PDesc extends PropertyDescriptor>

- (obj: Obj, prop: Key, val: PDesc) {

+ (obj: Obj, prop: Key, val: PDesc):

+ asserts obj is Obj & DefineProperty<Key, PDesc> {

Object.defineProperty(obj, prop, val);

}

obj jest zatem typu Obj (zawężony do kategorii ogólnej) i jest naszą nowo zdefiniowaną właściwością. To jest typ pomocnika DefineProperty:

type DefineProperty<

Prop extends PropertyKey,

Desc extends PropertyDescriptor> =

Desc extends {

writable: any, set(val: any): any

} ? never :

Desc extends {

writable: any, get(): any

} ? never :

Desc extends {

writable: false

} ? Readonly<InferValue<Prop, Desc>> :

Desc extends {

writable: true

} ? InferValue<Prop, Desc> :

Readonly<InferValue<Prop, Desc>>

Najpierw zajmiemy się właściwością writeable PropertyDescriptor. Jest to zestaw warunków do zdefiniowania niektórych skrajnych przypadków i warunków działania oryginalnych deskryptorów właściwości:

1. Jeśli ustawimy writable i dowolny akcesor właściwości (get, set), nie powiedzie się. never mówi nam, że został zgłoszony błąd.

2. Jeśli ustawimy writable na false, właściwość jest tylko do odczytu. Opieramy się na typie pomocnika InferValue.

3. Jeśli ustawimy writable na true, właściwość nie jest tylko do odczytu. My też zwlekamy.

4. Ostatni, domyślny przypadek jest taki sam jak writeable: false, więc Readonly <InferValue <Prop, Desc >>. (Readonly <T> jest wbudowany).

To jest typ pomocnika InferValue, zajmujący się ustawioną właściwością set:

type InferValue<Prop extends PropertyKey, Desc> =

Desc extends { get(): any, value: any } ? never :

Desc extends { value: infer T } ? Record<Prop, T> :

Desc extends { get(): infer T } ? Record<Prop, T> :

never;

Ponownie zbiór warunków:

1. Jeśli mamy getter i ustawioną wartość, Object.define Property zgłasza błąd, więc never.

2. Jeśli ustawiliśmy wartość, wywnioskujmy typ tej wartości i stwórzmy obiekt z naszym zdefiniowanym kluczem właściwości i typem wartości.

3. Lub wywnioskujemy typ z zwracanego typu metody pobierającej.

4. Cokolwiek jeszcze zapomnieliśmy. TypeScript nie pozwoli nam pracować z obiektem, ponieważ nigdy się nie stanie.

Przenoszenie go do konstruktora obiektów

To już działa wspaniale w twoim kodzie, ale jeśli chcesz to wykorzystać w całej aplikacji, powinniśmy umieścić tę deklarację typu w ObjectConstructor. Przenieśmy naszych pomocników do mylib.d.ts i zmieńmy interfejs ObjectConstructor:

```
type InferValue<Prop extends PropertyKey, Desc> =

Desc extends { get(): any, value: any } ? never :

Desc extends { value: infer T } ? Record<Prop, T> :

Desc extends { get(): infer T } ? Record<Prop, T> :

never;

type DefineProperty<

Prop extends PropertyKey,

Desc extends PropertyDescriptor> =

Desc extends {

writable: any, set(val: any): any

} ? never :

Desc extends {

writable: any, get(): any

} ? never :

Desc extends {

writable: false

} ? Readonly<InferValue<Prop, Desc>> :

Desc extends {

writable: true

} ? InferValue<Prop, Desc> :

Readonly<InferValue<Prop, Desc>>

interface ObjectConstructor {

defineProperty<
```

```
Obj extends object,
Key extends PropertyKey,
PDesc extends PropertyDescriptor
>(obj: Obj, prop: Key, val: PDesc):
asserts obj is Obj & DefineProperty<Key, PDesc>;
}
```

Dzięki scalaniu deklaracji i przeciążaniu funkcji, dołączamy tę znacznie bardziej konkretną wersję defineProperty do Object. W użyciu TypeScript dąży do uzyskania najbardziej poprawnej wersji podczas wybierania przeciążenia. Więc zawsze kończymy z jednym przeciążeniem, w którym wiążemy typy generyczne poprzez wnioskowanie. Zobaczmy, co robi TypeScript:

```
const storage = {
currentValue: 0
}
Object.defineProperty(storage, 'maxValue', {
writable: false, value: 9001
})
storage.maxValue // It's a number
storage.maxValue = 2 // Error! It's read-only
const storageName = 'My Storage'
defineProperty(storage, 'name', {
get() {
return storageName
}
})
storage.name // It's a string!
// It's not possible to assign a value and a getter
Object.defineProperty(storage, 'broken', {
get() {
return storageName
},
value: 4000
})
```

// Storage is never because we have a malicious

// property descriptor

storage

Mamy już kilka świetnych dodatków do zwykłego pisania, których możemy ponownie użyć we wszystkich naszych aplikacjach. Zadbaj o ten plik i uczyń go częścią standardowej konfiguracji.