Lekcja 4: Dodawanie typów za pomocą JSDoc

Teraz, gdy wiemy, jakie są typy, możemy zacząć być nieco bardziej celowi z obiektami danych i funkcjami w naszym małym skrypcie. Dodawanie typów do istniejącego pliku JavaScript może odbywać się na wiele sposobów. Jednym z najłatwiejszych jest użycie małego narzędzia o nazwie JSDoc. JSDoc to sposób na dodawanie adnotacji do naszego kodu za pomocą komentarzy. Opisujemy sygnatury funkcji, właściwości obiektów i wiele więcej przy użyciu pewnych konwencji:

```
/**
* Adding two numbers. This annotation tells TypeScript
* which types to expect. Two parameters (params) of
* type number and a return type of number
*
* @param {number} numberOne
* @param {number} numberTwo
* @returns {number}
*/
function addNumbers(numberOne, numberTwo) {
return numberOne + numberTwo
}
```

Aplikacja JSDoc zwykle działa na naszym kodzie źródłowym z adnotacjami i tworzy dokumentację w formacie HTML. TypeScript używa tych samych adnotacji, aby uzyskać więcej informacji na temat zamierzonych typów.

```
// TypeScript throws an error here, because the JSDoc
// comments expect two numbers, not a number and
// a string
addNumbers(3, '2')
// TypeScript throws an error here, because addNumbers
// returns a number, and toUpperCase() is not available
// in number
addNumbers(3, 2).toUpperCase()
```

This is a great way to be very intentional about what types to expect. And with that intent comes safety when using functions and when implementing them.

```
/**
* @param {number} numberOne
* @param {number} numberTwo
```

```
* @returns {number}

*/

function addNumbers(numberOne, numberTwo) {

return numberOne.toUpperCase() + ''

// Wait, what? We are treating numberOne like a

// string, even though it's a number, and we return

// it as a string even though we expect a number in

// return, there's something wrong here!

}
```

Dodatkową korzyścią jest dokumentacja z naszymi typami. A może otrzymujemy typy z naszą dokumentacją?

Typy niestandardowe

JSDoc działa dobrze z typami pierwotnymi, takimi jak liczba, ciąg i wartość logiczna. Ale jesteśmy również w stanie zdefiniować typy złożone, takie jak obiekty i tablice, za pomocą JSDoc. Obiekty są trochę skomplikowane. W JavaScript możemy deklarować obiekty w trakcie pracy za pomocą dwóch małych nawiasów klamrowych: const x = {}. Ten pusty obiekt nie ma obecnie żadnych dodatkowych właściwości, ale może się to radykalnie zmienić wraz z rozwojem naszego programu. Gdy x.hey = 'Ho', nagle bez problemu dodajemy nową właściwość do x. Te swobodne obiekty w JavaScript są elastyczne, ale także bardzo nieprzewidywalne. Dlatego TypeScript nie generuje błędów, gdy właściwość obiektu, która nie została jeszcze zdefiniowana, pojawia się w naszym zwykłym kodzie JavaScript. Można to gdzieś zdefiniować! Za pomocą adnotacji typu JSDoc możemy zadeklarować, których właściwości obiektu oczekujemy, i upewnić się, że TypeScript wie, czego się spodziewać. Nagle obiekt ma określony typ - kontrakt - i upewniamy się, że zawsze odwołujemy się do tej umowy. Dodajmy adnotację JSDoc do naszego obiektu pamięci. Spójrz na jego obecny kształt:

```
const storage = {

max: undefined,

items: []

}
```

Istnieje wartość o nazwie max, która może być dowolna. undefined to poprawna wartość dowolnego typu! Mamy też szereg pozycji, w przypadku których nie wiemy jeszcze, jak powinny wyglądać nasze produkty. Skoncentrujmy się najpierw na elementach. Elementy mogą mieć wiele właściwości, ale według naszego małego skryptu potrzebujemy wagi, a waga powinna być typu numer.

```
/**

* @typedef {Object} StorageItem

 * @property {number} weight

*/
```

Dlatego w komentarzu definiujemy nowy typ o nazwie Storage Item, który jest obiektem. Ma jedną właściwość zwaną wagą, która jest liczbą. W ten sam sposób tworzymy również typ dla naszego obiektu pamięci masowej.

```
/**

* @typedef {Object} ShipStorage

* @property {number} max

* @property {StorageItem[]} items

*/
```

Tutaj wpisanie jest bardzo ważne, ponieważ znowu jesteśmy bardzo świadomi tego, czego oczekujemy. max jest na razie niezdefiniowane, ale po przypisaniu wartości rzeczywistych powinna to być liczba. A nasza tablica jest pełna StorageItems, niestandardowego typu, który zdefiniowaliśmy zaledwie kilka wierszy powyżej. Teraz, po zdefiniowaniu naszych typów, zastosujmy je do obiektu pamięci:

```
/** @type ShipStorage */

const storage = {

max: undefined,

items: []

}
```

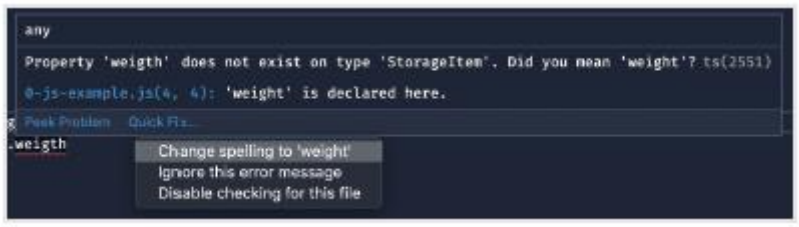
i dowiedz się, gdzie eksploduje nasz kod!

Bum!

Mamy pewne problemy z naszą funkcją storageUsed ():

```
function storageUsed () {

...

for (let i = 0; i <storage.length (); i ++) {

currentStorage + = storage.items [i] .weigth

}

...

}
```

Po pierwsze, właściwość .length() nie istnieje w typie Ship Storage. Jest to właściwość elementów, które otrzymujemy, ponieważ items to tablica. Ale to nie jest funkcja, to wartość. Również waga jest literówką. TypeScript sugeruje nam korektę.



Błąd TypeScript nie tylko pokazuje nam, co jest nie tak, ale także sugeruje, co chcieliśmy zrobić! Znajdowanie i naprawianie literówek to coś, w czym TypeScript jest naprawdę dobry.

Zamiar

Po prostu dodając prostą adnotację typu w komentarzu JSDoc, TypeScript wie znacznie więcej o semantyce naszego programu. A nasze zamiary dotyczące zmiennych, stałych i funkcji stają się dużo bardziej widoczne. Typy stają się narzędziem, którego możemy używać w całym naszym kodzie:

/ **

* Element @param {StorageItem}

* /

function add (item) {

if (storage.max - item.weight> = storageUsed ()) {

storage.items.push (element)

currentStorage + = item.weight

}

}

Typy sygnalizują zamiar, definiują kontrakt i upewniają się, że używamy naszego kodu programu tak, jak powinien. A to prawie nic nie kosztuje. Mała warstwa dokumentacji oprócz naszego zwykłego JavaScript i otrzymujemy znacznie więcej narzędzi, informacji i wiedzy. JSDoc jest bardzo potężny i może cię wiele przydać.