Lekcja 2: Polowanie na błędy

Po dodaniu // @ ts-check, TypeScript stał się aktywny w naszym pliku JavaScript i pokazał pierwszą listę problemów. Zwykle edytorzy kodu nie tylko przekazują wizualną informację zwrotną w postaci czerwonych falistych linii, ale także informują o problemie. Gdy najedziesz kursorem na jeden z problematycznych elementów, VS Code wyświetli wyjaśnienie w wyskakującym okienku. Ale możesz sam zauważyć problemy, gdy zostaną wyróżnione. Im więcej pracujesz z TypeScript, tym bardziej jest on intuicyjny.

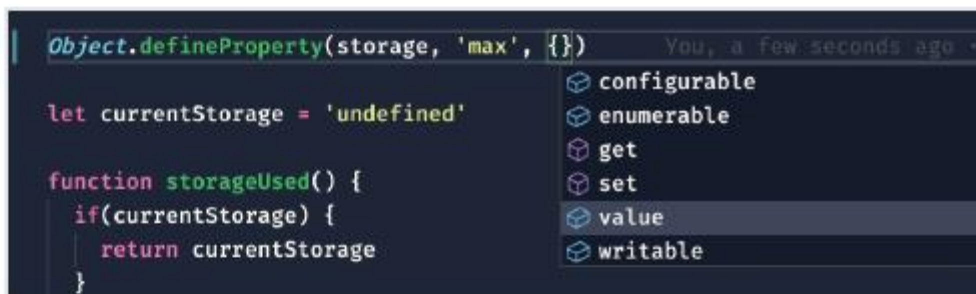
readonly nie jest przypisywalne

W linii 7. chcemy zdefiniować nową właściwość o nazwie max, ustawić wartość na 5000 i upewnić się, że nie możemy nadpisać wartości. TypeScript skarży się, że właściwość tylko do odczytu nie jest przypisywana do PropertyDescriptor. Fantazyjne słowa! Mają na myśli to, że pomieszaliśmy słowa. Deskryptory właściwości nie znają właściwości zwanej readonly; nazywa się to writable. Zamiast wartości readonly true potrzebujemy wartości writable false. Kiedy poprawimy nasz błąd, TypeScript powie nam również, że val nie istnieje - to value. Poprawiona linia wygląda następująco:

Object.defineProperty(storage, 'max', { writable:

false, value: 5000 })

To jest pierwsza cecha TypeScript: upewnienie się, że używasz poprawnych nazw dla rzeczy, których chcesz użyć. Koniec z literówkami, błędną pisownią lub pomieszanymi terminami. Nazywamy to sprawdzaniem typu: upewnianiem się, że spełniasz oczekiwania. Jaka jest różnica między readonly: true i writable: false? Jeden jest rozumiany przez JavaScript, a drugi nie. Ale skąd mamy wiedzieć, jakie właściwości ustawić? Ponieważ TypeScript wie, jakich właściwości się spodziewać i wyświetla błąd, gdy popełnisz błąd, możemy użyć tych samych informacji, aby uzyskać pomoc redaktora. Naciśnij Ctrl + Spacja bezpośrednio wewnątrz obiektu, a ponownie otrzymasz autouzupełnianie, ale w kontekście bieżącego wiersza:



Gdy TypeScript jest aktywny, natychmiast uzyskujemy szczegółowe informacje o tym, jakich informacji oczekujemy w tym konkretnym punkcie naszego kodu.

Nie tylko to, ale możemy również upewnić się, że nie dodamy czegoś innego niż true lub false podczas dodawania wartości do writable.

Wnioskowanie o typie

Przejdź do następnej czerwonej falistej linii. Gdy najedziemy kursorem na czerwone linie pod currentStorage, zobaczymy, że „Typu 0 nie można przypisać do typu ‘string’ ”. W JavaScript możesz przypisać różne wartości do zmiennych, w tym wartości zupełnie niepowiązane i wartości różnych typów. JavaScript nie dba o to, czy Twoje let foo to „Garfield” (ciąg znaków), 1337 (liczba) czy {heavy:

true}. To także moment, w którym pojawia się większość błędów. Po przypisaniu wartości do zmiennej najprawdopodobniej chcesz, aby zachowała ona określony typ. Kilka wierszy wcześniej utworzyliśmy zmienną currentStorage i ustawiliśmy ją na ‘undefined’. Jednak popełniliśmy tam mały błąd. Przypisaliśmy jej wartość ciągu „undefined”, a nie wartość programową undefined. To undefined mówi JavaScript, że nie ma jeszcze typu ani wartości.

```
// to powinno być currentStorage = undefined

let currentStorage = 'undefined'

function storageUsed() {

if(currentStorage) {

return currentStorage

}

currentStorage = 0

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

currentStorage += storage.items[i].weigth

}

return currentStorage

}
```

Powoduje to kaskadę błędów! Na przykład, if(current Storage) ma wartość true, zanim moglibyśmy nawet ustawić rzeczywistą bieżącą ilość pamięci. Jeśli ustawimy currentStorage = undefined, warunek ten przyjmuje wartość false. To z kolei prowadzi do kodu, w którym po raz pierwszy podsumowujemy bieżącą ilość pamięci (nie martw się, ta część również zawiera mnóstwo błędów). TypeScript ostrzega nas, że mieszamy typy. Podczas inicjowania currentStorage przypisujemy wartość ciągu. Później chcemy, żeby była to liczba. Zwykle jest to zachowanie, którego nie chcemy, więc TypeScript generuje błędy. TypeScript używa koncepcji zwanej inferencją typów. W momencie, gdy przypisujemy wartość do zmiennej, TypeScript próbuje wywnioskować typ z przypisania. Na przykład currentStorage = 0 mówi TypeScript, że oczekuje się, że currentStorage jest liczbą. Od tego momentu możemy tylko ponownie przypisywać liczby lub wykonywać czynności oparte na liczbach (na przykład operacje matematyczne). W momencie przypisania undefined, currentStorage może mieć dowolną wartość, dopóki nie uzyska odrębnego typu. Aby rozwiązać nasz problem, zmieniamy ‘undefined’ na undefined:

```
let currentStorage = undefined

function storageUsed() {

// Suddenly this evaluates to false with the

first call

if(currentStorage) {

return currentStorage

}
```

```
// From now on, currentStorage is a number

currentStorage = 0

...

// and storageUsed() returns a number

return currentStorage

}
```

Wnioskowanie o typie działa również w metodach. W powyższym przykładzie zwracamy currentStorage na końcu storageUsed(). Ponieważ wiemy, że currentStorage staje się liczbą, funkcja storageUsed() również musi zwrócić liczbę. W const x = storageUsed(), x będzie liczbą.

Weryfikacje semantyczne

W linii 16. mamy pętlę for, w której przeglądamy wszystkie nasze pozycje w magazynie i dokonujemy sumy.

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

currentStorage + = storage.items [i] .weigth

}
```

Niestety ta pętla ulegnie awarii. Powód: zadeklarowaliśmy zmienną inicjalizacyjną i jako stałą. Stałych nie można ponownie przypisać; dlatego ten kod nie zadziała. Zmiana, która pozwoli rozwiązać ten problem:

```
for (let i = 0; i <storage.length (); i ++) {// OK!

currentStorage + = storage.items [i] .weigth

}
```

Jest to jedna z wielu kontroli semantycznych. TypeScript nie tylko mówi nam, co jest nie tak. Przy prawidłowej integracji edytora może również sugerować szybkie poprawki, które rozwiązują Twój problem. Ponownie, głównym celem TypeScript jest zapewnienie najlepszego możliwego narzędzia. Chce lepiej zrozumieć Twój kod niż Ty.

TypeScript zna typowe błędy i sugeruje automatyczne poprawki.

Ostatnie bity

Powinieneś już znać korzyści, jakie daje nam TypeScript: analizuje kod, informuje nas, co jest nie tak i zwraca sugestie, jak możemy zapobiegać potencjalnym błędom lub je naprawiać. Czyni to poprzez porównanie kształtu obiektów i zmiennych na podstawie tego, co może wywnioskować z tym, co powinno być. Kilka ostatnich bitów należy do tych samych kategorii. W linii 23. TypeScript mówi nam, że „Operator„> = ”nie może być zastosowany do typów„ liczba ”i„ () => liczba’ ”. Warunek jest nie tylko bardzo trudny do odczytania, ale także porównujemy liczby do funkcji! Zapomnieliśmy wywołać storageUsed, więc naprawmy to:

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

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

Następna linia mówi nam, że add nie jest poprawną metodą na tablicach; powinien być wciśnięty:

-storage.items.add (pozycja)

+ storage.items.push (przedmiot)

Na koniec mamy paskudną literówkę. Skąd się bierze ten element, kiedy powinien to być przedmiot?

-currentStorage + = iten.weight

+ currentStorage + = item.weight

Dzięki prostemu dodaniu wiersza komentarza na początku naszego pliku JavaScript byliśmy w stanie wykryć mnóstwo potencjalnych problemów i pułapek, które spowodowałyby awarię naszego programu. TypeScript robi to, porównując typy tytułowe. Ale jakie są typy? Czy możemy użyć typów, aby rozwiązać jeszcze więcej problemów? Możemy, możemy!