TypeScript

TABLICE INFORMATYCZNE • Marcin Felczerek

Helion

PODSTAWY

TypeScript jest **statycznie typowanym** wariantem języka ECMAScript, znanego dużo lepiej pod nazwą JavaScript. Pozwala tworzyć programy na dowolne platformy oparte na JavaScript, takie jak strony WWW, Node czy Electron. Programy w TypeScript są na ogół kompilowane („transpilowane") do jednej z dostępnych wersji języka ECMAScript, aczkolwiek istnieje również eksperymentalny kompilator generujący kod Web Assembly.

Niniejsza sekcja zawiera podstawowe informacje dotyczące składni TypeScript. Część z nich powinna być doskonale znana osobom programującym w JavaScript, jednak część występuje jedynie w najnowszych wersjach tego języka, a niektóre są unikatowe dla TypeScript.

Linie programu

Programy TypeScript, podobnie jak w innych językach wywodzących się z języka C, składają się z linii kończących się znakiem średnika (z wyjątkiem komentarzy i końca bloków). Podobnie jak w wypadku ECMAScript, znak średnika jest opcjonalny, ponieważ środowisko wykonania dopisuje średniki na końcu linii, gdy ma to sens. Istnieją argumenty zarówno za używaniem średnika na końcu linii, jak i za niestosowaniem go, więc ostateczna decyzja zależy od przyjętych w danym projekcie standardów kodowania. Ważne jest konsekwentne stosowanie jednego lub drugiego podejścia, aby uniknąć chaosu w kodzie i trudnych do znalezienia błędów składniowych.

W przykładach poniżej stosowany jest styl z wykorzystaniem średników.

Komentarze

Komentarze w TS, podobnie jak w innych wariantach ECMAScript, mogą być blokowe:

```
/*
 * Komentarz blokowy rozciąga się od znaków / i * do * i /.
 * Częstą praktyką jest rozpoczynanie każdej linii od gwiazdki.
 * Nie jest to element języka, a jedynie dekoracja.
 */
```

lub wierszowe:

```
// Wszystko po znaku dwóch ukośników do
// znaku końca linii jest komentarzem
```

W plikach TS można też spotkać dwa specjalne typy komentarzy:

```
/**
 * Komentarz dokumentacyjny jest komentarzem blokowym
 * zaczynającym się dodatkową gwiazdką.
 */
```

Komentarze dokumentacyjne nie są częścią języka TypeScript, lecz przeniesionym z JavaScript standardem **JSDoc**. Służą one do zamieszczania w kodzie dokumentacji, możliwej do wyeksportowania przy pomocy specjalistycznych narzędzi, w tym większości dostępnych środowisk programistycznych (IDE).

```
/// <Dyrektywa/>
```

Dyrektywy to specjalne komentarze wierszowe, zawierające specjalistyczny znacznik języka XML. Służą one do sterowania procesem kompilacji. Odgrywały istotną rolę w starszych wersjach TypeScript, jednak obecnie ich znaczenie maleje.

Deklaracje zmiennych

Zmienne w języku TypeScript muszą być zadeklarowane przed pierwszym użyciem, w przeciwnym wypadku zostaną potraktowane jako błąd składniowy.

Zmienne lokalne w TypeScript deklarowane są przy użyciu słowa kluczowego `let`:

```
let imie: string = 'Janek';
```

Podobnie jak w ECMAScript, zmienną można zainicjować w miejscu deklaracji, przypisując jej wartość. Jeśli zmienna nie ma nadanej wartości, próba jej odczytania traktowana jest jako błąd kompilacji (w odróżnieniu od ECMAScript, gdzie ma wartość `undefined`).

Zmienne w TypeScript powinny mieć nadany typ. Typ zmiennej oddziela się od nazwy dwukropkiem. Jeśli typ zmiennej wynika z wartości początkowej, można go nie podawać.

```
let count = 1; // ta zmienna ma typ number
```

Słowo kluczowe `const` deklaruje tzw. zmienną niemodyfikowalną. W odróżnieniu od `let` taka zmienna musi mieć wartość początkową i nie może ona być zmieniana w wyniku przypisania. Zmienne te mogą mieć wartość zależną od wykonania funkcji, lub nawet różną w różnych przebiegach pętli, o ile nie jest ona zmieniana w swoim zasięgu.

```
const PI = 3.1415;
```

Zmienne w TypeScript mają *zasięg leksykalny*, istnieją od miejsca deklaracji do końca najbliższego bloku. Jeżeli w zasięgu zewnętrznym znajduje się zmienna o tej samej nazwie, zostaje ona *zasłonięta* przez nową **w całym bloku, również w liniach poprzedzających deklarację zmiennej zasłaniającej**.

```
const numer = 4;
if (czyPrawda()) {
    // (1) tu "numer" jest zasłonięty,
    // dostęp do niego jest błędem
    const numer = 5; // (2)
    // (3) tu "numer" jest zasłonięty i ma wartość 5
}
// (4) tu "numer" ma wartość 4
```

W punkcie (1) powyższego przykładu zmienna `numer` jest już zasłonięta (bo w bloku znajduje się deklaracja zmiennej o tej samej nazwie), ale jeszcze nie jest zadeklarowana. Oznacza to, że nie mamy do niej dostępu. Jest to zachowanie różniące TypeScript od języków takich jak Java czy C.

```
var staraZmienna = false;
```

TypeScript dopuszcza też słowo kluczowe `var` do deklarowania zmiennych. Funkcjonuje ono tak samo jak `let`, jednak zmienna ma zasięg funkcji, nie bloku, a jej deklaracja jest automatycznie przenoszona na początek treści funkcji. Jest to zachowanie identyczne jak w przypadku deklaracji zmiennych w „starym" ECMAScript i ze względu na swoje nietypowe cechy nie powinno być używane w kodzie TypeScript.

Przypisania i wyrażenia

Przypisanie wartości do zmiennej w TypeScript odbywa się przy pomocy operatora `=` (znaku przypisania, nie mylić ze znakiem równości `==` i `===`), podobnie jak w języku ECMAScript:

```
zmienna = 'Nowa wartość';
```

Przypisanie w TS jest wyrażeniem, którego wartością jest wartość wyrażenia z prawej strony znaku przypisania, dzięki czemu może ono być wykorzystane w innych konstrukcjach:

```
while ((znak = nextChar()) != ';') ...
```

W powyższej instrukcji wynik funkcji `nextChar` zostaje przypisany do zmiennej `znak`; następuje też sprawdzenie, czy nie jest on równy znakowi średnika.

TypeScript jest językiem statycznie typowanym, co oznacza, że wartość wyrażenia po prawej stronie znaku przypisania musi mieć ten sam typ, co zadeklarowany typ zmiennej.

Wyrażenia w języku TypeScript konstruowane są tak samo jak w ECMAScript i mogą zawierać dokładnie te same operatory:

- arytmetyczne: `+`, `-`, `*`, `/`, `%` (reszta z dzielenia), `**` (potęgowanie);
- w stylu języka C: `++` (zwiększenie o 1) i `--` (zmniejszenie o 1);
- bitowe: `|` (alternatywa), `&` (koniunkcja), `^` (suma wyłączna, XOR), `~` (negacja);
- przesunięć bitowych: `<<`, `>>`, `>>>`;
- porównań: `==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`;
- logiczne (skracające): `&&` (AND), `||` (OR), `!` (NOT);
- łączenia łańcuchów tekstowych: `+`;
- wywołania funkcji: `(...)`;
- indeksowania: `[]`;
- dostępu do pola obiektu: `.`;
- sprawdzenia istnienia indeksu: `in`;
- odczytania typu: `typeof`;
- sprawdzenia typu: `instanceof`;
- stworzenia obiektu: `new`;
- warunkowy: `? :`.

Operatory te funkcjonują identycznie jak w innych wariantach ECMAScript. W szczególności, operatory `==` i `!=` dokonują konwersji typów przed porównaniem, dlatego w większości przypadków powinno się stosować operatory ścisłego porównania, `===` i `!==`. Operator `typeof` zwraca łańcuch tekstowy zawierający nazwę typu podstawowego wartości, czyli `"string"`, `"number"`, `"boolean"`, `"null"`, `"undefined"` dla typów prostych, `"function"` dla funkcji i klas oraz `"object"` dla pozostałych wartości, w tym tablic.

Wyrażeniami mogą być też deklaracje funkcji i klas opisane dalej.

Instrukcje kontrolne

Instrukcje kontrolne służą do sterowania przebiegiem programu. Należą do nich instrukcje warunkowe i instrukcje pętli.

Instrukcja warunkowa if

Instrukcja warunkowa ma następującą składnię:

```
if (wyrażenie_warunkowe) {
    // blok instrukcji (1)
    ...
} else {
    // blok instrukcji (2)
}
```

Wyrażenie warunkowe jest konwertowane do wartości logicznej (prawda / fałsz), aczkolwiek TS jest stosunkowo tolerancyjny w kwestii *typu* wyrażenia. Wartości `0`, `null`, `undefined`, `NaN` i pusty tekst (`''`) traktowane są jako wartości fałszywe, a wszystkie inne jako prawdziwe.

Jeśli wyrażenie warunkowe ma wartość prawdy (`true`), to wykonywany jest blok instrukcji (1), zaś jeśli jest fałszywe, blok instrukcji (2). Jeżeli blok składa się z jednej instrukcji, TS pozwala nie używać nawiasów klamrowych, aczkolwiek taka składnia może prowadzić do niejasności i nie jest zalecana.

Klauzula `else` jest opcjonalna; można ją pominąć, jeśli nie planujemy żadnych instrukcji w bloku (2).

Bloki `if / else` można łączyć ze sobą, jeśli potrzebujemy sprawdzić więcej niż jeden warunek naraz:

```
if (warunek_1) {
    ...
} else if (warunek_2) {
    ...
} else if (warunek_3) {
    ...
} else {
    ...
}
```

Podobnie jak w wypadku prostego `if`, ostatnia klauzula `else` jest opcjonalna. Wykonanie bloku kończy się po znalezieniu pierwszego prawdziwego warunku lub na klauzuli `else`, jeśli żaden nie jest prawdziwy.

Instrukcja switch

Instrukcja `switch` służy do porównania zmiennej z serią stałych:

```
switch (zmienna) {
    case STAŁA_1:
        ...
        break;
    case STAŁA_2:
        ...
        break;
    default:
        ...
}
```

Instrukcja `switch` działa podobnie do połączonych instrukcji `if / else`, porównujących zmienną z kolejnymi wartościami stałymi. Jeśli zmienna nie jest równa żadnej z wartości stałych, wykonuje się blok oznaczony `default:`. Podobnie jak w wypadku `else` blok `default` jest opcjonalny.

Instrukcja `switch` posiada kilka nietypowych cech. Cała instrukcja stanowi jeden blok, w związku z tym deklarowanie zmiennej o tej samej nazwie w dwóch różnych sekcjach `case` jest nielegalne. Dodatkowo, jeśli znalezione zostanie dopasowanie zmiennej, wykonywany jest **cały** kod od początku tej sekcji `case` do końca bloku (włączając sekcję `default`). Słowo kluczowe `break` przerywa wykonanie bloku, stąd ważne jest umieszczenie go na końcu każdej sekcji. Obecnie większość narzędzi do sprawdzania kodu wyświetla ostrzeżenie, jeśli nie zakończymy sekcji słowem `break`.

Z drugiej strony, ta właściwość pozwala na stworzenie sekcji, która wykona się dla kilku różnych wartości zmiennej:

```
switch (zmienna) {
    ...
    case STAŁA_1:
    case STAŁA_2:
    case STAŁA_3:
        // kod
        ...
        break;
    case...
}
```

Ta wersja funkcji `case` **nie** jest uznawana za potencjalny błąd przez narzędzia sprawdzające.

Instrukcja pętli while

Pętla to blok kodu wykonujący się wiele razy. Instrukcja `while` jest najprostszą pętlą, wykonującą się, dopóki podany w niej warunek jest prawdziwy:

```
while (warunek) {
    // instrukcje...
    ...
}
```

Każdy „przebieg" pętli jest uznawany za samodzielny blok, stąd można wewnątrz pętli deklarować zmienne niemodyfikowalne (`const`). Jeżeli warunek zawsze jest prawdziwy, uzyskujemy pętlę nieskończoną:

```
while (true) {
    // ta pętla się nie kończy
}
```

Pętla z warunkiem zawsze fałszywym (np. `while (0) ...`) nie wykona się nigdy.

Jeśli chcemy najpierw wykonać jakiś kod, a dopiero potem sprawdzić warunek (czyli mieć gwarancję, że pętla wykona się co najmniej raz), używamy konstrukcji `do... while`:

```
do {
    // instrukcje...
    ...
} while (warunek);
```

Warto zauważyć średnik po klauzuli `while` w tej wersji.

Instrukcja pętli for

Pętla `for` jest wariantem pętli `while`, jednak najczęściej stosuje się ją do wykonania kodu określoną liczbę razy (np. do przetworzenia elementów tablicy).

```
for (<wyrażenie startowe>; <warunek>;
<wyrażenie pętli>) {
    // instrukcje
    ...
}
```

Średniki wewnątrz polecenia `for` są obowiązkowe, nawet jeśli nie stosujemy średników na końcu wierszy.

Wyrażenie startowe wykonuje się raz, na początku pętli. Z reguły są w nim deklarowane i inicjowane zmienne pętli.

Warunek sprawdzany jest przed każdym wykonaniem pętli, podobnie jak w pętli `while`.

Wyrażenie pętli wykonywane jest po wykonaniu instrukcji pętli, ale przed sprawdzeniem warunku dla kolejnego przejścia.

Każda z sekcji pętli `for` jest opcjonalna, jednak średnik na jej końcu nie. Pominięcie środkowego wyrażenia oznacza, że jest ono zawsze prawdziwe. Pętla `for(;;) {}` jest więc pętlą nieskończoną.

Pętla `for` jest logicznie odpowiednikiem:

```
{
<wyrażenie startowe>
    while (<warunek>) {
        instrukcje
        ....
        <wyrażenie pętli>
    }
}
```

Instrukcja pętli for in

Pętla `for in` jest bardziej wyspecjalizowaną wersją pętli `for`. Wygląda ona następująco:

```
for (const zmienna in obiekt) {
    // instrukcje
    ...
}
```

Pętla wykona się tyle razy, ile pól posiada obiekt, zaś zmienna będzie w każdym przebiegu przyjmować **nazwę** tego pola jako łańcuch tekstowy (co oznacza, że zmienna zawsze jest typu `string`, a dokładniej `keyof<obiekt>`).

Pętla ta jest wykorzystywana najczęściej z obiektami słownikowymi. Warto jednak pamiętać, że jeśli obiekt posiada jakieś prototypy, pętla będzie iterować również po nich. Dlatego jeśli chcemy tego uniknąć, powinniśmy za każdym razem sprawdzać, czy właściwość należy do tego konkretnego obiektu, a nie do jego prototypów:

```
for (const zmienna in obiekt) {
    if (obiekt.hasOwnProperty(zmienna)) {
        // instrukcje
        ...
    }
}
```

Dodatkowo, warto pamiętać, że TypeScript, podobnie jak inne wersje ECMAScript, nie rozróżnia pól i metod w obiektach, co oznacza, że jeśli obiekt, po którym iterujemy, ma metody, zostaną one również wyliczone. Możemy tego uniknąć dzięki operatorowi `typeof`:

```
for (const zmienna in obiekt) {
    if (obiekt.hasOwnProperty(zmienna) &&
typeof obiekt[zmienna] !== 'function') {
        // instrukcje
        ...
    }
}
```

Instrukcje break i continue

Instrukcje `break` i `continue` służą do kontroli nad wykonaniem pętli. Polecenie `break` przerywa wykonanie pętli i przekazuje sterowanie do pierwszej instrukcji po pętli:

```
while (warunek) {
    ...
    break;
    // ten kod się nie wykona
}
// tu zacznie wykonywać się kod po break
```

Polecenie `break` działa również w instrukcji `switch`, tak jak opisane jest to powyżej.

Polecenie `continue` przerywa wykonanie pętli, po czym przechodzi do wykonania następnego przebiegu