

# Wybrane elementy praktyki projektowania oprogramowania

## Wykład 06/15

## TypeScript

Wiktor Zychla 2023/2024

---

### 1 Spis treści

2	Wprowadzenie .....	3
3	Jak używać TypeScript .....	5
3.1	TypeScript Playground.....	5
3.2	node + tsc .....	5
3.3	ts-node.....	5
3.4	Deno / Bun.....	6
4	System typów .....	7
4.1	Informacje podstawowe.....	7
4.2	Typy literalne/proste/strukturalne. Typy jawne/niejawne. Typy a interfejsy .....	11
4.3	Inferencja typów.....	12
4.4	Unie i przecięcia typów .....	14
4.5	Wnioskowanie na ścieżkach wykonania, <i>type guards</i> .....	16
4.6	Przeciążanie funkcji .....	19
4.7	Przykład – unia z wariantami.....	20
4.8	Generyczność .....	21
4.9	Typy indeksowane .....	22
4.10	Typy funkcyjne, typy indeksowe .....	23
4.11	keyof, typeof.....	24
4.12	Typy mapowane, typy warunkowe .....	27
4.13	Wbudowane typy generyczne (typy użytkowe) .....	29
4.14	Infer .....	30
4.15	Klasy.....	30
4.16	TypeScript Type Challenge .....	32
4.17	Zupełność systemu typów w sensie Turinga .....	32
5	Migracja z JavaScript do TypeScript .....	34
6	Literatura .....	35



## 2 Wprowadzenie

TypeScript to język dodający do JavaScript statyczne typowanie. [Dokumentacja języka](#) obejmuje m.in. leksykon języka, podręczniki i playground.

O historii TypeScript mieliśmy już okazję powiedzieć. Warto wiedzieć, że TypeScript to nie jedyny pomysł na dodanie statycznego typowania do JavaScript. Inne podejścia to m.in.

- [Flow](#) od Facebook, warto zerknąć na [chart porównujący TypeScript i Flow](#)
- [Scala.js](#)

W dokumentacji TypeScript proszę zwrócić uwagę na rozdziały wprowadzające, np. [TS for Java/C# Programmers](#) oraz [handbook](#).

Fundamentalne założenia:

- Każdy kod JavaScript jest składniowo poprawnym kodem TypeScript, co najwyżej w pewnych przypadkach spowoduje błąd kompilacji
- Typy są w wielu miejscach opcjonalne i w wypadku ich braku kompilator używa mechanizmu [wnioskowania](#) (ang. *inference*) do określenia typu
- W JavaScript idiomatyczny styl programowania zakłada możliwość tworzenia obiektów za pomocą [składni literalnej](#) (oczywiście również: za pomocą **new** i wskazania funkcji konstruktorowej), stąd system typów nie może bazować na typach nominalnych (jak w klasycznych językach obiektowych gdzie **var foo = new Foo()** przypisuje ten konkretny typ do zmiennej). Zamiast tego, system typów TypeScript jest [strukturalny](#).
- Typy istnieją w kodzie do chwili kompilacji i są **wymazywane po kompilacji**. Wynikiem kompilacji TypeScript jest JavaScript, bez anotacji typowych. To trochę przypomina zasadę działania wczesnych kompilatorów C++, gdzie wynikiem kompilacji C++ z klasami był C bez klas.
- Oprócz konieczności zapewnienia poprawnego typowania strukturalnego, istnieje potrzeba generalizacji (uogólnień), stąd system typów posiada możliwość używania [typów generycznych](#)

Dawnej TypeScript rozwijał się niezależnie od JavaScript, ponieważ z uwagi na stagnację rozwojową JS był pozbawiony wielu istotnych mechanizmów (**class**, **async/await**). Do TS dodawano więc zarówno te elementy, o których wiadomo było że może w przyszłości pojawią się w JS, jak i takie, które swoich odpowiedników nie mają i wymagają nietrywialnego rozwinięcia w odpowiadający im kod JS.

Przykładem takiego mechanizmu jest konstrukcja **enum**

```
enum Direction {  
    Left,  
    Right,  
    Up,  
    Down  
}
```

która tłumaczy się na

```
"use strict";
var Direction;
(function (Direction) {
    Direction[Direction["Left"] = 0] = "Left";
    Direction[Direction["Right"] = 1] = "Right";
    Direction[Direction["Up"] = 2] = "Up";
    Direction[Direction["Down"] = 3] = "Down";
})(Direction || (Direction = {}));
```

Po uzupełnieniu JS o część mechanizmów które TS posiadał wcześniej, uznano że taka sytuacja, w której TS „wyprzedza” JS w obszarze konstrukcji języka, „wymuszając” niejako „dorównywanie” do TS w taki sposób żeby zamianami „wyrównującymi” w JS nie popsuć TS, jest niedopuszczalna i zagraża rozwojowi obu języków.

Komitet rozwojowy języka uznał więc, że przyszły rozwój języka TS powinien raczej skupiać się w obszarze **systemu typów**, a nie konstrukcji językowych.

Sporo interesujących informacji wnosi post [Ten Years of Typescript](#) z października 2022.

## 3 Jak używać TypeScript

### 3.1 TypeScript Playground

[TypeScript Playground](#) to najprostszy sposób żeby zacząć programować w TS. W piaskownicy można obejrzeć wynik kompilacji oraz nawet uruchomić kod.

### 3.2 node + tsc

Kompilator TypeScript najwygodniej zainstalować globalnie

**npm install typescript -g**

a następnie w folderze projektu utworzyć plik konfiguracyjny **tsconfig.json** lub poprosić kompilator o zainicjowanie domyślnego pliku

**tsc --init**

Plik konfiguracyjny określa szereg opcji samej kompilacji ale też to które pliki mają być kompilowane (sekcje **files/include/exclude**). W przypadku braku wskazania które pliki mają być kompilowane, kompilator skompiluje wszystkie pliki w bieżącym folderze i jego podfolderach (w przypadku w ogóle braku pliku konfiguracyjnego, będzie oczekiwał poprawnych argumentów wywołania i jest to znacznie mniej wygodne).

Kompilator uruchomiony w trybie

**tsc --watch**

„nasłuchuje” zmian w plikach źródłowych i inkrementacyjnie rekompiluje zmiany.

Domyślnie wynikiem kompilacji jest zbiór plików **\*.js** odpowiadających plikom **\*.ts**. Istnieje możliwość wskazania jednego pliku wyjściowego (parametr **outFile**) ale robi się tak rzadko ponieważ:

- w przypadku kompilacji kodu dla serwera (np. node.js), wiele plików dla aplikacji nie jest żadnym problemem
- w przypadku kompilacji kodu dla klienta (np. React) i tak używa się zewnętrznego bundlera (np. [webpack](#) i wbudowany w niego bundler [terser-webpack-plugin](#)) do łączenia plików (są inne bundlery, np. Rollup czy Parcel)

### 3.3 ts-node

[ts-node](#) to środowisko uruchomieniowe TypeScript, które rozszerza node o kompilację TS „w locie”. Do tego należy używać polecenia **ts-node** zamiast **node**.

Można też wymusić tryb ts-node w ramach „zwykłego” uruchomienia aplikacji (przez node ...):

- mieć w folderze plik **tsconfig.json**
- do parametrów uruchomieniowych w **launch.json** dodać

```
"runtimeArgs": ["-r", "ts-node/register"],
```

- jako plik startowy w **launch.json** wskazać plik \*.ts (zamiast \*.js)

W ten sposób uruchomiona aplikacja nie tworzy widocznych plików \*.js, a mimo to daje się normalnie debugować.

### 3.4 Deno / Bun

[Deno](#) to alternatywne środowisko uruchomieniowe dla JS/TS, które rozwija się niezależnie od node. Warto poczytać o planach rozwojowych i różnicach między deno a node. Uruchomienie kodu TS w deno przypomina uruchomienie w node pod nadzorem ts-node.

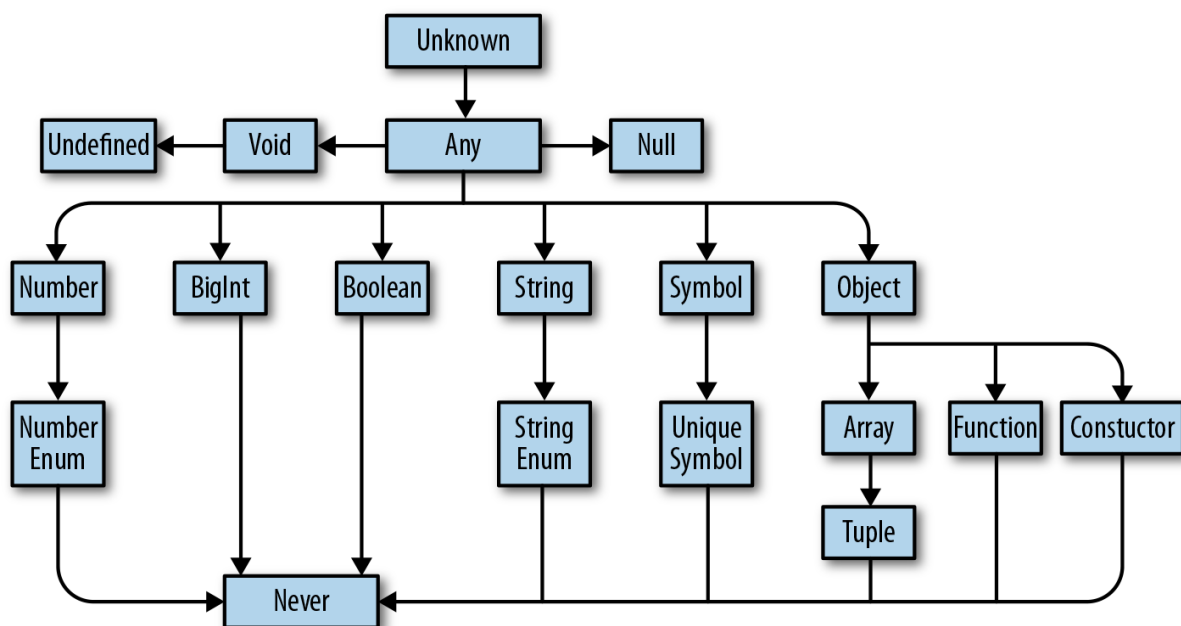
[Bun](#) to również stosunkowo młody projekt, warto śledzić jego rozwój.

## 4 System typów

### 4.1 Informacje podstawowe

Struktura kodu w TypeScript przypomina tę z JavaScript, stąd pominiemy oczywiste informacje o tym jak działają funkcje i klasy i jak wygląda kod wewnątrz ciał funkcji. TypeScript zachowuje wiele konwencji (np. **var** vs **let** vs **const** itd.).

Hierarchia typów TypeScript jest bardzo interesująca ...



Rysunek 1 Źródło: <https://www.oreilly.com/library/view/programming-typescript/9781492037644/ch03.html>

... i porównaniu z innymi statycznie typowanymi językami (np. C++, Java, C#) wyróżnia się następującymi właściwościami:

- Istnieje **najbardziej ogólny** typ, **unknown** (odpowiednik na przykład **object** z C#) i jego ogólny podtyp **any** (odpowiednik **dynamic** z C#). Każda zmienna każdego innego typu jest więc równocześnie typu **any** (i **unknown**)
- Istnieje **najmniej ogólny** typ, **never**, który wyróżnia się tym, że co prawda można nadać taki typ zmiennej, ale takiej zmiennej nie da się nadać żadnej wartości (sic!). Dopóki nie zobaczymy jak działa mechanizm typów warunkowych to w ogóle można zadać pytanie „po co jest taki typ”?

```
// typ unknown nie pozwala na dostęp do żadnej składowej
let u: unknown;

u.foo(); // błąd
u.bar;   // błąd
```

```
// typ any pozwala na dostęp do każdej składowej
let a: any;

a.foo(); // ok
a.bar;   // ok
```

- Na typ w TS należy patrzeć inaczej niż na typy (klasy) w językach z klasami – to nie jest tak, że *każdy obiekt ma swój typ*. Chodzi bardziej o to że *każdy obiekt ma strukturę pasującą do jednego lub wielu typów*
- Dla znających Java/C#: typy TypeScript są **bliższe interfejsom** niż klasom, opisują kształty obiektów a niekoniecznie wymuszają ich miejsce w hierarchii typów
- Obiekt może mieć strukturę statycznie (na etapie kompilacji) pasującą do wielu typów mimo tego, że w czasie uruchomienia kodu jego typ (prototyp) może być jakiś, jeden, konkretny (być może nawet **Object.prototype**)
- Typy są przypisywane statycznie, w trakcie kompilacji i w TS nie ma większości znanych z JS koercji. Jeżeli typ wynika z kontekstu (i jest przypisany automatycznie), to nie można go zmienić w „czasie życia” zmiennej, np.

```
let a = 1;

// nie można w TS!
a = 'foo';
```

```
let a = 1;
let b = true;

// nie można w TS!
let c = a + b;
```

- Nieszablonowe koercje (te z przykładowego zadania w którym wyrażenie złożone z `+ ! [ ]` dawało w wyniku napis „fail”) są w przeważającej większości przypadków również niemożliwe
- Należy wyraźnie rozgraniczyć więc TS jako statyczny system typów sprawdzanych na etapie kompilacji i JS jako nośnik typów uruchomieniowych, przypisywanych obiektom w trakcie wykonania kodu. System typów TS (na etapie kompilacji) jest nieporównanie bogatszy niż system typów JS (podczas uruchomienia kodu). Wynika to z tego że o ile część typów TS i JS pokrywa się (na przykład zmienna zadeklarowana w TS jako **string** będzie miała w JS też typ **string**) to sporo typów zadeklarowanych w TS po prostu **zniknie** po kompilacji do JS (np. wszystkie typy zadeklarowane przez programistę z użyciem **type/interface**)

```
// przykład 1. typ prosty w TS
let s: string = 's';
```



```
// po uruchomieniu, na poziomie JS, mamy ten sam typ, "string"
console.log( Object.getPrototypeOf(s) === String.prototype );
// > true

// przykład 2. typ złożony w TS
type Person = {
  name: string;
  surname: string;
}

// w czasie kompilacji przez TS, p ma typ Person
let p: Person = {
  name: 'jan',
  surname: 'jan'
}

// po uruchomieniu, na poziomie JS, prototypem p jest Object.prototype
// Person "znika" z pola widzenia systemu typów czasu wykonania
console.log( Object.getPrototypeOf(p) === Object.prototype );
// > true
```

- Większość typów umieszczonych w kodzie faktycznie TS znika po kompilacji do JS, wyjątkiem są albo konstrukcje lukrujące język (jak pokazany wcześniej **enum**) albo **klasy** (klasa w TS jest zarówno implementacją typu jak i samym typem)
- Dobrą intuicją dla pojęcia typu w TS jest **określenie nazwy (aliasu) dla zbioru możliwych wartości**. To myślenie kategoriami **zbiór (typ) vs element zbioru (obiekt)** ułatwia przyswojenie konwencji:
  - Jeden i ten sam obiekt może „pasować” do wielu typów (element może należeć do wielu zbiorów) mimo tego że w trakcie uruchomienia kodu obiekt ma jeden, określony typ/prototyp
  - Istnieją typy (zbiory) jednoelementowe (tzw. typy literałów) – co jest dość zaskakujące, aczkolwiek w innych językach mamy typ **boolean** który ma zasadniczo dwie wartości, więc typ jednoelementowy nie powinien nas dziwić
  - Istnieją operatory **sumy mnogościowej** i **części wspólnej** dla typów, których wynikiem są kolejne typy
  - Istnieje **operator warunkowy** (taki *if* na systemie typów) oraz **rekursja** – oba te mechanizmy łącznie powodują że system typów TS jest [zupełny w sensie Turinga](#) (to że język jest zupełny w sensie Turinga to dość oczywiste, natomiast to że system typów jest zupełny w sensie Turinga wymaga komentarza – w pierwszej chwili możemy bowiem nawet nie zdać sobie sprawy co to tak naprawdę znaczy; temat pojawi się na końcu wykładu)

Należy pamiętać że

- kontrola typów w trakcie kompilacji nie przeszkadza w świadomym pisaniu kodu niepoprawnego (nadużywającego systemu typów)

- typy na których rozumowanie prowadzi kompilator TS a typy jakie pojawiają się w czasie działania kodu (JS) mogą się różnić i prowadzić do błędów z powodów nieoczekiwanych przez programistę, np.

```
/**
 * TypeScript: Funkcja "oślepia" system typów
 */
function convert(s: string): number {
    return s as unknown as number;
}

let result = convert('123');
console.log( result.toFixed() );
```

To co programista uznał za „dobrą monetę” (`s as unknown as number`), po translacji na JS zamieniło się bowiem w

```
"use strict";
/**
 * Po kompilacji do JavaScript: Funkcja "oszukuje" system typów
 */
function convert(s) {
    return s;
}
let result = convert('123');
console.log(result.toFixed());
```

gdzie już śladu po żadnej „konwersji” nie ma i kod spowoduje wyjątek w trakcie działania (typ `string` nie ma metody `toFixed`)!.

Tego typu konstrukcje językowe

- `as unknown as XXX`
- `as any`

należy traktować jako broń obosieczną – można w ten sposób wymusić określony efekt na kompilatorze, ale otrzymany kod może nie działać w trakcie uruchomienia.

Na wczesnym etapie nauki TS można nie zawsze dobrze wyczuć który element języka znika po kompilacji do JS, a który nie znika i jakie są tego skutki dla kodu.

Trudno też zawsze dobrze wyczuć czy asercja typowa

**let bar = foo as Bar**

nie spowoduje błędu.

Tego typu niezamierzone błędy mogą się więc zdarzać.

Uwaga! Istnieją pewne **nietrywialne** przypadki w których poprawnie kompilujący się kod, nie używający tego typu wymuszonych rzutowań, również nie działa poprawnie w trakcie uruchomienia. Przykład takiej sytuacji można znaleźć w artykule [TypeScript puzzle No. 1](#).

## 4.2 Typy literalne/proste/strukturalne. Typy jawne/niejawne. Typy a interfejsy

Typy mogą być „anonimowe” (typu nie trzeba nazywać, przykład niżej) ale za pomocą słów kluczowych **type** i **interface** można tworzyć aliasy (nazwy) dla typów. Ta zasada – że typ nie musi być nazwany – jest uniwersalna i warto o niej pamiętać (nie bać się typów nienazwanych)

```
// typ literalny anonimowy: typem zmiennej jest 'x'

let _x1: 'x' = 'x';

// typ literalny został nazwany, X
type X = 'x';
let _x2: X = 'x';

// typ strukturalny anonimowy:
// typem zmiennej jest { name: string, surname: string }

let _person1: { name: string, surname: string } = { name: 'jan', surname: 'kowalski' };

// typ strukturalny został nazwany, Person
type Person = { name: string, surname: string };
let _person2: Person = { name: 'jan', surname: 'kowalski' }
```

Najwięźsze typy posiadające nietrywialne wartości to tzw. **typy literalne** (określające jedną, konkretną wartość, np. 'a') i ich zalety ujawniają się na przykład w połączeniu z **uniami typowymi** (za chwilę) – mogą służyć do reprezentacji skończonych zbiorów wartości (i stanowią alternatywę dla **enum**!). Typy szersze, opisujące struktury nazwiemy **typami strukturalnymi**.

```
// typy literalne ('jednoelementowe')
type A = 'a';
type _1 = 1;

// alias na typ prosty
type MyInt = number;

// typ strukturalny
```

```
type Person = {
  name: string,
  surname: string
}
```

Różnice między **type** a **interface** są nieznaczne i można powiedzieć że to synonimy, z zastrzeżeniem:

- W składni **type** pewne rzeczy da się napisać, a w składni **interface** nie (unie, przecięcia)
- W składni **interface** można powielać definicje, są one wtedy „składane” (to się lepiej nadaje do opisu kontraktu bibliotek zewnętrznych)

Typy proste (**number**, **string**, **boolean**) odpowiadają swoim wersjom z JavaScript – przykład już był wcześniej.

### 4.3 Inferencja typów

Jako język typowany strukturalnie, TS nie wymaga obiektu który ma zadeklarowany **nominalnie** wymagany typ, tylko obiektu który **strukturalnie** spełnia narzucone wymagania:

```
type Person = {
  name: string,
  surname: string
}

function DoWork(p: Person): Person {
  // zwraca obiekt literalny - ale struktura jest zgodna!
  return {
    name: p.name + ' from doWork',
    surname: p.surname + " from doWork"
  };
}

// przekazuje jako argument obiekt literalny - ale struktura jest zgodna
const p = DoWork({ name: 'jan', surname: 'kowalski' });

console.log(JSON.stringify(p))
```

W powyższym przykładzie proszę zwrócić uwagę na to że ani obiekt zwracany z funkcji ani obiekt przekazywany do funkcji **nie ma** jawnie zadeklarowanego typu (choć mógłby!) i w obu przypadkach TS jest w stanie zweryfikować poprawność wywołania na podstawie wyłącznie zgodności pól w sygnaturze typu. Gdyby funkcja **DoWork** nie zadeklarowała w sygnaturze że zwraca obiekt typu **Person**, TypeScript uznałby że zwracany obiekt jest typu **{ name: string, surname: string }** i w innych językach byłby to nominalnie inny typ niż **Person**. **W TypeScript strukturalnie te typy są równoważne!**

Tu ciekawostka: obiekty literalne { **foo: 1, bar: 1, ...** } są szczególnie „chronione” przez kompilator kiedy pojawiają się bezpośrednio jako wartości zwracane z funkcji lub parametry przekazywane do funkcji. Nie skompiluje się więc ...

```
type Person = {
  name: string,
  surname: string
}

function DoWork(p: Person): Person {
  // zwraca obiekt literalny - ale struktura jest zgodna!
  return {
    name: p.name + ' from doWork',
    surname: p.surname + " from doWork"
  };
}

// próbuje przekazać obiekt literalny - ale dodatkowa składowa!
// błąd kompilacji!
const p = DoWork({ name: 'jan', surname: 'kowalski', foo: 1 });

console.log(JSON.stringify(p))
```

... ale już ...

```
type Person = {
  name: string,
  surname: string
}

function DoWork(p: Person): Person {
  // zwraca obiekt literalny - ale struktura jest zgodna!
  return {
    name: p.name + ' from doWork',
    surname: p.surname + " from doWork"
  };
}

// przekazuje obiekt literalny, jest dodatkowa składowa ale jest ok
var param = { name: 'jan', surname: 'kowalski', foo: 1 };

const p = DoWork( param );

console.log(JSON.stringify(p))
```

... tak!

Jedyna różnica między tymi przykładami sprowadza się do tego że parametr jest przekazany bezpośrednio lub z pomocniczej zmiennej.

Niemniej, ta zasada o wymaganiu wyłącznie zgodności strukturalnej, jest o wiele bardziej przydatna niż może się wydawać. Proszę sobie wyobrazić że pracujemy z obiektami przeglądarki i tam obiekt **window.document** ma właściwość **title**. I piszemy kod TS, który ma odczytać tę właściwość. W językach typowanych nominalnie musielibyśmy w sygnaturze metody wymusić typ taki jaki nominalnie ma **window.document** (jakiś specyficzny typ), bo zadeklarowanie własnego, nowego typu z jedną składową (title) nie pozwala wymusić implementacji takiego własnego typu przez istniejący obiekt. A TS naturalne jest ...

```
type ObjectWithTitle = {
  title: string
}

function ShowDocumentTitle( document: ObjectWithTitle ) {
  console.log( document.title );
}

ShowDocumentTitle( window.document );
```

Przez to że **window.document** ma właściwość **title** i że jest typowany strukturalnie – kompilator przepuści wywołanie metody **ShowDocumentTitle** jako poprawne.

Typowanie strukturalne w TypeScript jest więc bardzo naturalnym odpowiednikiem zachowania JavaScript, w którym odwołanie do składowej obiektu jest rozwiązywane w czasie działania: w TS to czy obiekt ma jakiś typ jest przez kompilator co prawda rozwiązywane w trakcie kompilacji, ale za to wynika to wyłącznie ze struktury obiektu, a nie z nominalnego osadzenia go gdzieś w hierarchii typów.

W językach z nominalnym systemem typów nie możemy ani swobodnie definiować sobie „równoważnych” interfejsów (interfejsy **Foo** i **Bar** nawet gdyby miały te same struktury byłyby dwoma różnymi interfejsami) ani też mieć takiego efektu że klasa niejawnie implementuje interfejs, tylko dlatego że interfejs ma zgodne składowe.

#### 4.4 Unie i przecięcia typów

Dwa dowolne typy (zbiory) można połączyć operatorami unii **|** albo przecięcia **&** dostając w ten sposób nowy typ mający intuicyjnie właściwości jednego **lub** drugiego albo jednego **i** drugiego. Działa to zarówno dla typów literalnych (choć uwaga na **&** w tym przypadku):

```
type AorB = 'a' | 'b';

// są dwa literały należące do tego typu, 'a' i 'b'
let aorb: AorB = 'a';
```

```
// czy coś może być równocześnie 'a' i 'b'?  
type AandB = 'a' & 'b';  
  
let aandb: AandB; // typ AandB jest równoważny typowi never (sic!)
```

jak i dla typów strukturalnych

```
type Person = {  
  name: string,  
  surname: string  
}  
  
type Vehicle = {  
  name: string,  
  maxSpeed: number  
}  
  
// {  
//   name: string  
// }  
type PersonOrVehicle = Person | Vehicle;  
  
// {  
//   name: string,  
//   surname: string  
//   maxSpeed: number  
// }  
type PersonAndVehicle = Person & Vehicle;
```

To w jaki sposób wyznaczane są składowe odpowiednich typów literalnych może być trochę zaskakujące i w pierwszej chwili – sprzeczne z intuicją. Wydawałoby się że operator `|` pozwala na *więcej*, można by więc zaryzykować że wnioskowana struktura będzie sumą mnogościową obu, a w przypadku operatora `&` że struktura będzie częścią wspólną obu.

Tak też jest w istocie, typ **PersonOrVehicle** zawiera w sobie zarówno wszystkie struktury zgodne z **Person** jak i te zgodne z **Vehicle**, więc jest *sumą mnogościową* obu swoich składowych, ale o pojedynczym obiekcie takiego zbioru – skoro może być jednym lub drugim – nie da się powiedzieć więcej:

```
type Person = {  
  name: string,  
  surname: string  
}  
  
type Vehicle = {
```

```

    name: string,
    maxSpeed: number
}

function DoWork( pv: Person | Vehicle ) {

    // jedyna składowa do której można odwołać się bez ograniczeń:
    let name = pv.name;

    // pozostałe składowe, surname i maxSpeed mogą być lub nie
    // jak się do nich dostać? O tym za chwilę
}

```

W praktyce:

- **Unie typów** wykorzystuje się tam, gdzie zmienna/parametr funkcji przyjmuje wartości różnych typów. Na przykład w sytuacji gdy funkcja ma kilka przeciążeń (kilka wariantów sygnatur) lub tam gdzie funkcja operuje na parametrach z hierarchii (przykłady za chwilę)
- **Przecięcia typów** to zwyczajnie składnia umożliwiające określenie **dziedziczenia** między typami – tam gdzie w składni klas byłoby ...

```

class Person {
    name!: string
}

class WorkingPerson extends Person {
    position!: string
}

```

(operator ! pozwala nam poinformować kompilator że sami na siebie bierzemy nadawanie wartości tej zmiennej i zrzekamy się kontroli przepływu wartości **null/undefined**)

... w składni dla typów jest po prostu

```

type Person = {
    name: string
}

type WorkingPerson = Person & { position: string }

```

#### 4.5 Wnioskowanie na ścieżkach wykonania, *type guards*



Kompilator TS wykonuje potężną pracę w sytuacji gdy zmienna ma typ opisany unią – potrafi [zawieźć typ zmiennej](#) na ścieżkach wykonania, pod warunkiem użycia jednej obsługiwanych składni ...

```
function doWork( p: string | number ) {  
  
    // p jest typu string lub typu number  
    // niewiele można więc z nim zrobić, ale ...  
  
}
```

... na przykład **typeof**

```
function doWork( p: string | number ) {  
  
    if ( typeof p === 'string' ) {  
  
        // w tym bloku p jest typu string  
  
    } else {  
  
        // w tym bloku p jest typu number  
  
    }  
  
}
```

Działa to całkiem sprawnie:

```
function doWork( p: string | number | boolean ) {  
  
    if ( typeof p === 'string' ) {  
  
        // w tym bloku p jest typu string  
  
    } else {  
  
        // w tym bloku p jest typu number lub boolean ale  
  
        if ( typeof p === 'number' ) {  
  
            // w tym bloku p jest typu number  
  
        } else {  
  
            // w tym boku p jest typu boolean  
  
        }  
  
    }  
  
}
```

```
}
```

Szczególnym przypadkiem zawężania są funkcje pełniące rolę [strażników typowych](#) – ich sygnatura jest wyjątkowa, bowiem o ile formalnie zwracają wartość typu **boolean**, sygnatura określa że zwracają informację o przynależności obiektu do jakiegoś typu. Te funkcje pozwalają na poziomie statycznej analizy poradzić sobie z następującym problemem:

```
type Person = {
  name: string
}

type WorkingPerson = Person & { position: string }

function doWork( p: Person | WorkingPerson ) {

  // jak odróżnić Person od WorkingPerson?
  // informacji o typie nie ma przecież w czasie wykonania!

  if ( ... ) {

    // tu p jest typu Person

  } else {

    // tu p jest typu WorkingPerson

  }
}
```

Otóż wystarczy **strażnik typowy**

```
type Person = {
  name: string
}

type WorkingPerson = Person & { position: string }

// strażnik typowy (type guard)
function isWorkingPerson( p: Person ): p is WorkingPerson {

  return 'position' in p;

}
```

```
function doWork( p: Person | WorkingPerson ) {

    if ( isWorkingPerson( p ) ) {

        // tu p jest typu WorkingPerson
        console.log( p.name, p.position );

    } else {

        // tu p jest typu Person
        console.log( p.name );

    }
}

doWork( { name: 'jan', position: 'lecturer' } );
```

Proszę zwrócić uwagę na użycie operatora **in** i w związku z tym pewne wymaganie – to odróżnienie **Person** od **WorkingPerson** zadziała tylko wtedy kiedy jest jakieś pole, tu: **position** i faktycznie występuje w obiekcie i można go użyć do rozróżnienia.

Niżej pokażemy inne podejście, charakterystyczne dla TypeScript, tzw. **unia z wariantami**.

#### 4.6 Przeciążanie funkcji

Wyżej opisany mechanizm zawężania typu na ścieżce wykonania uzupełnia się znakomicie z pomysłem na przeciążanie sygnatur funkcji. Przypomnijmy, że w JavaScript nie ma przeciążania, a jeżeli funkcja obsługuje kilka wariantów wywołania, to jest to obsługiwane w jednym, jedynym wariacie funkcji.

Na przykład funkcja **splice** tablicy w JS ma de facto dwa przeciążenia, jedno „zwykłe” a jedno z opcjonalną dodatkową tablicą elementów, które należy wstawić we wskazane miejsce. JS osiąga ten efekt tak że funkcja ma jedną implementację i sprawdza jak została wywołana (z jakimi argumentami).

TypeScript nie pozwala więc **przeciążyć implementacji** funkcji, ale zamiast tego pozwala na **przeciążenie samej sygnatury**

```
// sygnatury dla klienta
function sum( a: string, b: string ): string;
function sum( a: number, b: number ): number;

// implementacja
function sum( a: any, b: any ): any {
    return a + b;
}
```

W przypadku istnienia przeciążonych sygnatur, kompilator **ukrywa** sygnaturę implementacji (w przykładzie – tę z **any**) – klient ma możliwość użycia wyłącznie jeden z sygnatur zadeklarowanych (tych bez implementacji).

W powyższym przykładzie w sygnaturze implementacyjnej ukryto nietrywialny problem implementacyjny – operator **+** zachowuje się inaczej dla argumentów różnych typów i kompilator trochę „na wyrost” ufa programiście że sygnatury zadeklarowane są prawidłowe. Widać tu więc pole do nadużycia:

```
// sygnatury dla klienta
function sum( a: string, b: string ): string;
// to nieprawda, ale kompilator uwierzy programiście!
function sum( a: number, b: number ): string;

// implementacja
function sum( a: any, b: any ): any {
    . . .
}
```

#### 4.7 Przykład – unia z wariantami

Poniższy przykład do samodzielnego przestudiowania pokazuje jeszcze inny sposób uzyskania efektu zawężania typu na ścieżce wykonania – ten przykład jest idiomatyczny dla TS:

```
/* pola niewymagane */
type OsobaZPesel = {
    typ: 'pesel',
    nazwisko: string,
    pesel?: string,
}

type OsobaZDowodem = {
    typ: 'dowod',
    nazwisko: string,
    dowodOsobisty?: string
}

type OsobaZData = {
    typ: 'data',
    nazwisko: string,
    dataUrodzenia?: Date
}

type Osoba = OsobaZPesel | OsobaZDowodem | OsobaZData;

let o1: Osoba = {
    typ: 'pesel',
```

```

    nazwisko: 'kowalski',
    pesel: '12345'
}

function doSomething( o: Osoba ) {
    if ( o.typ === 'pesel' ) {

        // o jest typu OsobaZPesel

    } else if ( o.typ === 'dowod' ) {

        // o jest typu OsobaZDowodem

    } else {

        // o jest typu OsobaZData

    }
}

```

## 4.8 Generyczność

TypeScript pozwala na wprowadzenie zmiennej typowej w definicji typu lub funkcji, w ten sposób uzyskując znany z innych języków efekt [typów generycznych](#)

W każdym przypadku zmienna generyczna może być ukonkretniona przez programistę albo wnioskowana z wywołania.

Rozważmy prostą funkcję filtrującą:

```

let filter = (t: number[], f: (n: number) => boolean) : number[] => {
    var result = new Array<number>();

    for ( var n of t ) {
        if ( f(n) )
            result.push( n );
    }

    return result;
}

```

Wygląda na to że można ją uogólnić, filtrowanie działałoby tak samo gdyby tablica była **jakiegoś** typu a predykat filtrujący mówił co zrobić z obiektem tego typu:

```

let filter = function <T>(t: T[], f: (n: T) => boolean): T[] {
    var result = [];
}

```

```

    for (var n of t) {
        if (f(n))
            result.push(n);
    }

    return result;
}

```

Kompilator zapewni zgodność typu tablicy i typu predykatu:

```

8
9     return result;
10 }
11
12 filter([1, 2, 3], )

```

filter(t: number[], f: (n: number) => boolean):  
number[]

Na powyższym rzucie ekranu z VS Code widać, że wywołanie funkcji tak że pierwszy argument jest inferowany jako **number[]** powoduje automatyczne wymuszenie typu drugiego argumentu jako **(n: number) => boolean**.

## 4.9 Typy indeksowane

Kolejny interesujący rodzaj typów to typy indeksowane – chodzi o możliwość określenia aliasu na część innego typu. Ten mechanizm pozwala inaczej patrzeć na to które typy są **źródłami wiedzy** dla systemu typów a które są typami wtórnymi (*pochodnymi*).

W klasycznym języku z klasami można zwykle wyłącznie komponować duże typy z małych:

```

type Address = {
    city: string,
    number: number
}

type Person = {
    name: string,
    surname: string,
    address: Address
}

```

W TS też tak można ale można też w drugą stronę, zadeklarować duży typ, a potem *wydzielić* z niego mniejsze:

```

type Person = {
    name: string,
    surname: string,

```

```

    address: {
      city: string,
      number: number
    }
  }

type Address = Person['address'];

```

#### 4.10 Typy funkcyjne, typy indeksowe

Funkcje mają swoje typy zarówno jeśli są samodzielnymi typami jak i wtedy kiedy są polami innych typów. Warto zwrócić uwagę na niuanse składni. W poniższym przykładzie typ funkcyjny zadeklarowany jest trzy razy, dwie z definicji są sobie równoważne (które?).

```

type functionType1 = (a: number) => number;

type functionType2 = { (a: number) : number };

type functionType3 = { a : (a: number) => number };

// które są równoważne a które nie?
let f1: functionType1;
let f2: functionType2;
let f3: functionType3;

```

Można opisać typ obiektu, mapujący klucze na wartości (czyli sytuację w której nie wiadomo jakie będą nazwy pól obiektu). W poniższym przykładzie typ **Settings** ma jedno pole, które wiadomo na pewno jak się nazywa (**timeout**) i dowolne inne pola o jakichś nazwach (składnia **[key: string] ...**)

```

type Settings = {
  [key: string]: string | number | boolean;
  timeout: number;
}

let settings: Settings = {

  timeout: 4,
  setting1: 'bar',
  anotherSetting: true
}

```

## 4.11 keyof, typeof

Dopiero w tym miejscu można powiedzieć o jednej z najważniejszych cech systemu typów – **typy literalne i typy strukturalne są ze sobą ściśle powiązane**. Typ literalny, a raczej unia typów literalnych może bowiem określać zestaw kluczy typu strukturalnego.

```
type Person = {
  name: string,
  surname: string,
}

// 'name' | 'surname'
type PersonKeys = keyof Person;
```

Ponieważ rozwiązywanie typów działa na etapie kompilacji, również w trakcie pisania kodu w VS otrzymujemy stosowne podpowiedzi.

**Rozważmy przykład.** Funkcja ma otrzymać obiekt z ustawieniami i klucz (nazwę ustawienia) i zwrócić wartość. Obiekt z ustawieniami opisany jest jako:

```
let settings = {

  setting1: true,
  anotherSetting: 'server=localhost;username=admin'

}

type SettingsType = typeof settings;

function readSetting( settings: SettingsType, settingName: string ) {

  // błąd, dowolny string nie może być użyty tu jako indeks!
  return settings[settingName]

}
```

Przy okazji zwróćmy uwagę że za pomocą operatora **typeof** można otrzymać statycznie typ odpowiadający jakiejś zmiennej (**typeof** w JavaScript nie służył do określenia typu w trakcie wykonania; to oznacza że w TypeScript są w rzeczywistości dwa operatory **typeof** – jeden działający w trakcie wykonania i drugi dostępny w trakcie kompilacji; o tym który jest aktualnie używany decyduje kontekst!).

Jak naprawić ten przykład?

Pierwsza wersja to dodanie ograniczenia na klucze:



```

let settings = {
    setting1: true,
    anotherSetting: 'server=localhost;username=admin'
}

type SettingsType = typeof settings;

function readSetting( settings: SettingsType, settingName: keyof SettingsType
) {
    // już lepiej, można indeksować
    return settings[settingName]
}

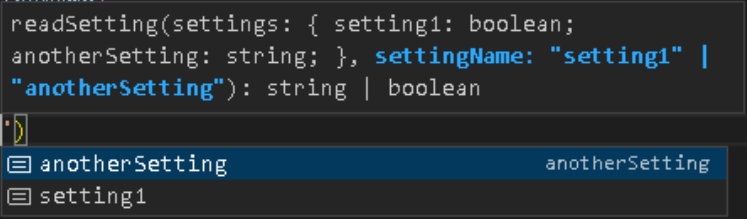
```

Jest lepiej, funkcja akceptuje już wyłącznie klucze rzeczywiście zdefiniowane w typie:

```

11
12 // błąd, dowolny string nie może być użyty tu jako indeks!
13 return settings[settingName]
14
15 }
16
17 readSetting( settings, '')

```



Ale co ta funkcja zwraca? Okazuje się że **string | boolean** (dlaczego?)

Można jeszcze lepiej, można podpowiedzieć kompilatorowi że wartość zwracana zależy od parametru. Wspomoże nas mechanizm typów generycznych i możliwość [narzucania ograniczeń na parametry generyczne](#)

```

let settings = {
    setting1: true,
    anotherSetting: 'server=localhost;username=admin'
}

type SettingsType = typeof settings;

function readSetting<K extends keyof SettingsType>(
    settings: SettingsType,
    settingName: K ): SettingsType[K] {

```

```

    // jest ok
    return settings[settingName]
}

readSetting( settings, 'setting1' ) // boolean
readSetting( settings, 'anotherSetting' ) // string

```

**Popatrzmy na inny przykład.** W tym przykładzie zmapujemy tablicę obiektów na tablicę zawierającą rzutowanie obiektów na którąś z ich właściwości:

```

function mapToProperty(tab: any[], prop: string): any[] {
    return tab.map( item => item[prop] );
}

type Person = {
    name: string,
    surname: string,
    age: number
}

let persons: Person[] = [
    { name: 'jan', surname: 'kowalski', age: 1 },
    { name: 'tomasz', surname: 'malinowski', age: 2 }
]

let mappedPersons = mapToProperty( persons, 'name' );

console.log( JSON.stringify( mappedPersons ) );

```

Ta wersja działa, ale nie ma żadnej kontroli nad drugim argumentem, przekazany do funkcji jako nazwa pola które ma być użyte do rzutowania.

A tak? ...

```

type Person = {
    name: string,
    surname: string,
    age: number
}

let persons: Person[] = [
    { name: 'jan', surname: 'kowalski', age: 1 },
    { name: 'tomasz', surname: 'malinowski', age: 2 }
]

```

```

]

function mapToProperty2<T>(tab: T[], prop: keyof T): T[keyof T][] {
    return tab.map( item => item[prop] );
}

let mappedPersons2 = mapToProperty2( persons, 'name' );
console.log( JSON.stringify( mappedPersons2 ) );

```

Jest lepiej, ale przez zastosowanie typu indeksowanego ( `T[keyof T][]` ) wynik jest typu ( `string | number` )[], trochę podobnie jak w poprzednim przykładzie.

Można więc jeszcze lepiej!

```

type Person = {
    name: string,
    surname: string,
    age: number
}

let persons: Person[] = [
    { name: 'jan', surname: 'kowalski', age: 1 },
    { name: 'tomasz', surname: 'malinowski', age: 2 }
]

function mapToProperty3<T, K extends keyof T>(tab: T[], prop: K): T[K][] {
    return tab.map( item => item[prop] );
}

let mappedPersons3 = mapToProperty3( persons, 'age' );
console.log( JSON.stringify( mappedPersons3 ) );

```

## 4.12 Typy mapowane, typy warunkowe

Jeśli **keyof** działa tak że z typu strukturalnego wybiera klucze, to czy istnieje operacja odwrotna? Taka, że dla zadanej unii kluczy tworzy typ strukturalny z tymi kluczami?

Tak. Do tego służą **typy mapowane**. Popatrzmy na przykład

```

type CustomRecord<K extends string | number | symbol, T> = { [k in K]: T };

// {
//   name: string
//   surname: string

```

```
//    position: string
//}
type PersonRecord = CustomRecord<'name' | 'surname' | 'position', string>
```

Wydaje się że typy mapowane to prosty mechanizm. Popatrzmy jednak na inny mechanizm, **typy warunkowe**, gdzie za pomocą operatora `?` : można statycznie sprawdzić czy typ spełnia jakiś warunek i w razie potrzeby „terminować” wyliczanie typu:

```
type AcceptOnly<T> = T extends 'name' ? T : never;

// 'name'
type Name = AcceptOnly<'name'>

// never
type NoName = AcceptOnly<'noname'>
```

Oba mechanizmy rozważane niezależnie, to po prostu – jakieś mechanizmy na systemie typów. Ale popatrzmy co potrafią razem:

```
type PersonKeys = 'name' | 'surname' | 'age' | 'position';

type ExceptKeys<T, U> = T extends U ? never : T;

// 'age' | 'position'
type SomePersonKeys = ExceptKeys<PersonKeys, 'name' | 'surname'>
```

Typ pomocniczny **ExceptKeys** potrafi “przefiltrować” typ określający zbiór kluczy przez swój podzbiór (dlaczego? Jak działa operator `?` : kiedy argument jest unią typów?)

I idąc krok dalej:

```
type Person = {
  name: string,
  surname: string,
  age: number,
  position: number
}

// pomiń klucze
type ExceptKeys<T, U> = T extends U ? never : T;

// mapuj obiekt na obiekt z pominiętymi kluczami
type OmitKeys<T, K extends keyof T> = { [k in ExceptKeys<keyof T, K>]: T[k] }
```

```
// {  
//   age: number.  
//   position: number  
//}  
type WorkingPerson = OmitKeys<Person, 'name' | 'surname'>
```

**To bardzo ważny przykład.** Pokazaliśmy że na poziomie systemu typów można mieć zaawansowane operacje „przykrawające” typ do innego (w drugą stronę – to że typ można rozszerzyć to już wiemy, bo do tego służy operator **&** ). Proszę zestawić tę nową wiedzę z uwagą w podrozdziale o typach indeksowanych – o tym które typy są źródłami wiedzy a które typy są *pochodne*.

Operatory takie jak zdefiniowany wyżej **OmitKeys** pozwalają na bezpieczne projektowanie własnych typów. Za chwilę pokażemy że biblioteka standardowa zawiera tego typu typy użytkowe!

#### 4.13 Wbudowane typy generyczne (typy użytkowe)

Wymieńmy kilka podstawowych [typów użytkowych](#) z biblioteki standardowej TypeScript:

##### **Partial<Type>**

Konstruuje z zadanego typu typ, który ma wszystkie składowe opcjonalne

##### **Required<Type>**

Konstruuje z zadanego typu typ, który ma wszystkie składowe wymagane

##### **Readonly<Type>**

Konstruuje z zadanego typu typ, który ma wszystkie składowe tylko-do-odczytu

##### **Record<Keys, Type>**

Konstruuje typ z zadanymi kluczami i typem wartości – chwilę temu definiowaliśmy taki (**CustomRecord**)

##### **Extract<Keys, Keys>**

Z zadanej unii typowej zwraca wskazane elementy

##### **Exclude<Keys, Keys>**

Z zadanej unii typowej wyrzuca wskazane elementy

##### **Pick<Type, Keys>**

Z zadanego typu struktury zwraca podtyp ze wskazanymi kluczami

##### **Omit<Type, Keys>**

Z zadanego typu struktury zwraca podtyp z wyrzuconymi wskazanymi kluczami

#### 4.14 Infer

Ostatnią nietrywialną właściwością systemu typów jaką pokażemy jest operator **infer**, który pozwala wydobyć część większego typu i „przekazać dalej” we wnioskowaniu warunkowym, na przykład:

```
type GetReturnType<T> = T extends (...args: never[]) => infer Return
  ? Return
  : never;

// number
type Num = GetReturnType<() => number>;
```

W ten sposób można wyciągnąć typ konkretnego parametru funkcji, np.:

```
type GetSecondArgType<Type> =
  Type extends (first: any, second: infer SecondType, ...args: never[])
    => any
  ? SecondType
  : never;

// number
type Num = GetSecondArgType<(a: string, b: number, c: boolean) => number>;
```

albo wręcz dowolnego parametru funkcji, np.:

```
type GetAnyArgType<Type, N extends number> =
  Type extends (...args: infer C) => any
  ? C[N]
  : never;

// number
type Num = GetAnyArgType<(a: string, b: number, c: boolean) => number, 1>;
```

Proszę spróbować zrobić coś takiego w C# czy Javie – zadeklarować typ, który z podanego typu funkcyjnego wydobywa typ któregoś konkretnego parametru. Nie w trakcie działania programu tylko właśnie – w trakcie jego kompilacji. Czy w ogóle jest to tam wykonalne?

#### 4.15 Klasy

Do tej pory w rozważaniach o systemie typów pomijaliśmy całkowicie klasy – zdefiniowane za pomocą składni **class Foo ...**

One oczywiście w TypeScript istnieją, pytanie jak się mają do typów definiowanych za pomocą type/interface? Przecież w przeciwieństwie do type/interface, klasy nie są wymazywane podczas kompilacji.

Zasada jest prosta – każda zdefiniowana klasa określa również typ, który funkcjonuje jak każdy inny typ (można go używać w połączeniu z innymi typami). Można również powiedzieć że klasa implementuje jakiś istniejący tylko w trakcie kompilacji typ/interfacejs.

Proszę we własnym zakresie sprawdzić które elementy z poniższego przykładu ulegają zatarciu przy kompilacji a które nie.

```
// klasa bazowa
class NamedObject {

    constructor( public name: string ) {

    }

    doWork() {
        return this.name;
    }

}

// jakiś typ
type PersonType = {
    name: string,
    surname: string
}

// podklasa
class Person extends NamedObject implements PersonType {

    constructor( public name: string, public surname: string ) {
        super(name);
    }

    doWork() {
        return super.doWork() + this.surname;
    }

}

let person = new Person('jan', 'kowalski');
console.log( person.doWork() );
```

```
// podtyp
type PersonWithAge = Person & { age: number }
```

## 4.16 TypeScript Type Challenge

Dla chętnych poznać system typów TypeScript przygotowano [interaktywny zbiór zadań](#), w których na kilkudziesięciu zadaniach o rosnącym stopniu trudności zademonstrowano różne właściwości systemu typów.

Zadania można rozwiązywać albo w VS Code albo w edytorze TypeScript playground.

High-quality types can help improve projects' maintainability while avoiding potential bugs.

There are a bunch of awesome type utility libraries that may boost your works on types, like [ts-toolbelt](#), [utility-types](#), [SimplyTyped](#), etc, which you can already use.

This project is aimed at helping you better understand how the type system works, writing your own utilities, or just having fun with the challenges. We are also trying to form a community where you can ask questions and get answers you have faced in the real world - they may become part of the challenges!

### Challenges

Click the following badges to see details of the challenges.

warm-up 1

13 · Hello World

easy 13

4 · Pick 7 · Readonly 11 · Tuple to Object 14 · First of Array 18 · Length of Tuple 43 · Exclude 189 · Awaited 268 · If 533 · Concat 898 · Includes 3057 · Push 3060 · Unshift 3312 · Parameters

medium 72

2 · Get Return Type 3 · Omit 8 · Readonly 2 9 · Deep Readonly 10 · Tuple to Union 12 · Chainable Options 15 · Last of Array 16 · Pop 20 · Promise.all 62 · Type Lookup 106 · Trim Left 108 · Trim 110 · Capitalize 116 · Replace 119 · ReplaceAll 191 · Append Argument 296 · Permutation 298 · Length of String 459 · Flatten 527 · Append to object 529 · Absolute 531 · String to Union 599 · Merge 612 · KebabCase 645 · Diff 949 · AnyOf 1042 · IsNever 1097 · IsUnion 1130 · ReplaceKeys 1367 · Remove Index Signature 1978 · Percentage Parser 2070 · Drop Char 2257 · MinusOne 2595 · PickByType 2688 · StartsWith 2693 · EndsWith 2757 · PartialByKeys 2759 · RequiredByKeys 2793 · Mutable 2852 · OmitByType 2946 · ObjectEntries 3062 · Shift 3188 · Tuple to Nested Object 3192 · Reverse 3196 · Flip Arguments 3243 · FlattenDepth 3326 · BEM style string 3376 · InorderTraversal 4179 · Flip 4182 · Fibonacci Sequence 4260 · AllCombinations 4425 · Greater Than 4471 · Zip 4484 · IsTuple 4499 · Chunk 4518 · Fill 4803 · Trim Right 5117 · Without 5140 · Trunc 5153 · IndexOf 5310 · Join 5317 · LastIndexOf 5360 · Unique 5821 · MapTypes 7544 · Construct Tuple 8640 · Number Range 8767 · Combination 8987 · Subsequence 9896 · GetMiddleElement 10969 · Integer 16259 · ToPrimitive 17973 · DeepMutable

hard 40

6 · Simple Vue 17 · Currying 1 55 · Union to Intersection 57 · Get Required 59 · Get Optional 89 · Required Keys 90 · Optional Keys

## 4.17 Zupełność systemu typów w sensie Turinga



Przykłady do samodzielnego przestudiowania:

- [Pierwsza dyskusja o tym czy system typów jest zupełny](#)
- [arytmetyka](#)
- [Maszyna Turinga](#)
- [Zbiór przykładów](#), m.in. interpreter SQL, kompilator TypeScript na systemie typów TypeScript, przygodowa gra tekstowa

## 5 Migracja z JavaScript do TypeScript

Pierwszy krok migracji to zmiana rozszerzeń \*.js na \*.ts i praca na konfiguracji

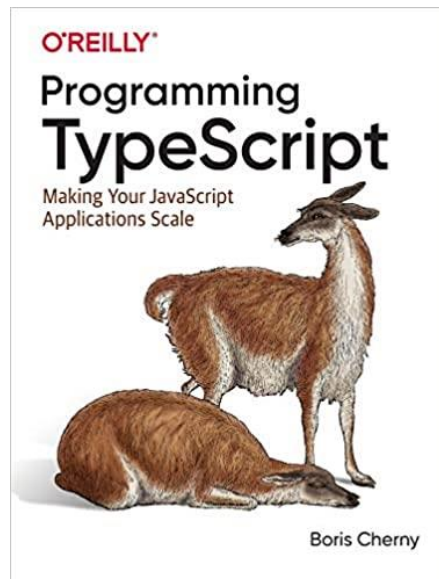
```
{
  "compilerOptions": {
    "outDir": "./built",
    "allowJs": true,
    "target": "es5"
  },
  "include": ["./*src/**/*"]
}
```

która dopuszcza JS bezpośrednio w kodzie źródłowym oraz pomija rozliczne weryfikacje. Po zweryfikowaniu początkowych problemów można stopniowo aktywować kolejne ważne przełączniki:

- [noImplicitAny](#)
- [strictNullChecks](#)
- [noImplicitThis](#)

## 6 Literatura

B. Cherny, Programming TypeScript: Making your JavaScript Applications Scale



D. Vanderkam, Effective TypeScript, 62 Specific Ways to Improve your TypeScript

