Spis treści

Wstęp

Wiosną 2016 odwiedziłem dawnego współpracownika Evana Martina w siedzibie Google w San Francisco i zapytałem, co go ekscytuje. Zadawałem mu to samo pytanie od wielu lat, ponieważ jego odpowiedzi obejmowały szeroki zakres zagadnień, były nieprzewidywalne, lecz zawsze ciekawe: dotyczyły narzędzi do budowania C++, sterowników audio w systemie Linux, krzyżówek online, wtyczek emacs. Tym razem Evan był zainteresowany językiem TypeScript i programem Visual Studio Code.

To było zaskakujące! Słyszałem już o TypeScript, jednak wiedziałem, że został opracowany przez Microsoft i błędnie sądziłem, że był powiązany z .NET. Nie mogłem uwierzyć, że tak oddany użytkownik systemu Linux jak Evan przeszedł na stronę systemu Microsoft.

Wtedy Evan pokazał mi program vscode oraz narzędzie TypeScript Playground, czym natychmiast mnie przekonał. Wszystko działało tak szybko, a narzędzia do edycji kodu bardzo ułatwiały zbudowanie modelu myślowego systemu typowania. Po latach pisania adnotacji typów w komentarzach JSDoc dla narzędzia Closure Compiler, miałem poczucie tworzenia działającego typowanego kodu JavaScript. A firma Microsoft zbudowała wieloplatformowy edytor tekstu na bazie Chromium? Być może warto się nauczyć tego języka i narzędzi.

Ostatnio zacząłem pracę w Sidewalk Labs i pisałem nasz pierwszy kod JavaScript. Mieliśmy jeszcze tak mało kodu, że Evan i ja zdołaliśmy go przekształcić do TypeScript w ciągu kilku dni.

Wciągnęło mnie to. TypeScript jest czymś więcej niż tylko systemem typowania. Oferuje cały pakiet usług językowych, które są szybkie i łatwe w użyciu. W efekcie TypeScript nie tylko zwiększa bezpieczeństwo programów JavaScript, ale sprawia, że programowanie jest znacznie przyjemniejsze!

Dla kogo jest ta książka

Książki z serii *Effective programming* zostały opracowane jako „standardowe drugie podręczniki" do przedmiotu, który omawiają. Największe korzyści z książki *TypeScript. Skuteczne programowanie* odniosą osoby, które mają już praktyczne doświadczenie w pisaniu kodu JavaScript i TypeScript. Pisząc tę książkę nie miałem zamiaru uczyć nikogo programowania w TypeScript lub JavaScript, ale ułatwić przejście z poziomu początkującego lub średnio zaawansowanego programisty do poziomu eksperta. Możemy to osiągnąć dzięki elementom, na które podzielona jest ta książka. Ułatwią nam one tworzenie modeli myślowych działania języka i ekosystemu TypeScript, uczulą na pułapki, których

należy unikać i zaprezentują możliwości TypeScript, oraz pokażą, jak z nich korzystać w najbardziej skuteczny sposób. Podczas gdy typowe podręczniki przedstawiają kilka sposobów, jakimi język umożliwia wykonanie jakiegoś zadania, ta książka wyjaśnia, który z tych kilku sposobów należy wybrać i dlaczego.

TypeScript ewoluował szybko przez ostatnie lata, lecz mam nadzieję, że jest już na tyle ustabilizowany, że zawartość tej książki będzie aktualna przez kilka następnych lat. Ta książka koncentruje się przede wszystkim na samym języku, a nie na platformach czy narzędziach do budowania. Nie znajdziemy tu przykładów korzystania z platform React lub Angular z TypeScript, ani konfigurowania TypeScript w celu współpracy z webpack, Babel lub Rollup. Zalecenia zawarte w tej książce przydadzą się wszystkim użytkownikom TypeScript.

Dlaczego napisałem tę książkę

Gdy zacząłem pracować w Google, dostałem egzemplarz trzeciego wydania książki *Skuteczny nowoczesny C++*. Okazała się zupełnie inna niż wszystkie pozostałe książki programistyczne, które dotychczas czytałem. Jej adresatami nie byli początkujący programiści. Nie obejmowała też wszystkich funkcji języka. Zamiast przedstawiać działanie różnorodnych cech C++, informowała, których z nich używać, a których nie. Wszystkie informacje były podzielone na kilkadziesiąt krótkich specyficznych elementów poświęconych konkretnym przykładom.

Czytanie tych wszystkich przykładów podczas codziennego korzystania z języka było świetnym doświadczeniem. Pisałem już wcześniej programy C++, ale po raz pierwszy poczułem swobodę tworzenia w tym języku. Dowiedziałem się też jak postrzegać poszczególne opcje, które zostały przedstawione w książce. W kolejnych latach podobne wrażenie towarzyszyło mi podczas czytania książek *Java. Efektywne programowanie* i *Efektywny JavaScript*.

Jeśli ktoś swobodnie posługuje się kilkoma różnymi językami programistycznymi, wówczas nauka tajników nowego języka stanowi wyzwanie dla modeli myślowych i poznawania różnic między tymi językami. Pisząc tę książkę poznałem mnóstwo nowych aspektów TypeScript. Mam nadzieję, że czytelnicy odniosą takie same korzyści!

Organizacja książki

Ta książka zawiera kolekcję „elementów", z których każdy jest swego rodzaju technicznym esejem, oferującym konkretne porady dotyczące określonego aspektu TypeScript. Elementy są pogrupowane tematycznie w rozdziały, lecz zachęcam do swobodnego przeglądania i czytania najbardziej interesujących nas elementów.

Tytuł każdego elementu podkreśla kluczowe zalecenie. O tych sprawach powinniśmy pamiętać korzystając z TypeScript, a zatem warto przejrzeć spis treści, aby zachować je w pamięci. Jeśli na przykład piszemy dokumentację i wydaje się nam, że nie powinniśmy

uwzględniać informacji o typach, powinniśmy przeczytać Element 30: Nie powtarzajmy informacji o typie w dokumentacji.

Treść elementu omawia zalecenie podane w tytule, przedstawia odpowiednie przykłady i prezentuje argumenty techniczne. Niemal każdy punkt przedstawiony w tej książce jest poparty przykładowym kodem. Zwykle podczas czytania książek technicznych analizuję przykłady i przeglądam tekst. Podejrzewam, że tak samo postępują czytelnicy tej książki. Mam nadzieję, że przeczytają tekst i wyjaśnienia! Jednak nawet jeśli będziemy jedynie analizować przykłady, powinniśmy zrozumieć najważniejsze kwestie.

Po przeczytaniu każdego elementu czytelnicy powinni zrozumieć, dlaczego zawarte w nim zalecenie ułatwi im skuteczne korzystanie z TypeScript. Będą także wiedzieć, czy dane zalecenia należy stosować w określonych sytuacjach. Scott Meyers, autor książki *Skuteczny nowoczesny C++*, prezentuje zapadający w pamięć przykład. Kiedyś spotkał się z zespołem programistów, którzy napisali oprogramowanie sterujące pociskami. Wiedzieli, że mogą zignorować jego porady o zapobieganiu wycieków zasobów, ponieważ ich oprogramowanie zakończy działanie, gdy pocisk trafi w cel, a sprzęt ulegnie zniszczeniu. Nie znam żadnych pocisków sterowanych kodem JavaScript, jednak kod w tym języku obsługuje teleskop Kosmiczny Teleskop Jamesa Webba, a zatem wszystko jest możliwe!

Na końcu każdego elementu znajduje się sekcja „Należy zapamiętać", która zawiera kilka punktów z podsumowaniem tego elementu. Jeśli ktoś tylko przegląda treść książki, może przeczytać te punkty, aby uzyskać ogólny pogląd na zagadnienia poruszane w tym elemencie i dowiedzieć się, czy chciałby przeczytać więcej. Warto jednak przeczytać treść elementu! W ostateczności wystarczy samo podsumowanie.

Konwencje wykorzystywane w przykładowym kodzie TypeScript

Wszystkie przykłady kodu są napisane w TypeScript z wyjątkiem sytuacji, gdy z kontekstu wynika, że są w formacie JSON, GraphQL lub w innym języku. Korzystanie z TypeScript w dużej mierze wymaga korzystania z edytora, co nie zawsze można dobrze zaprezentować w druku. W tym celu korzystam z kilku konwencji.

Większość edytorów wyświetla błędy za pomocą pofalowanego podkreślenia. Aby sprawdzić cały komunikat błędu, należy umieścić kursor myszy nad podkreślonym tekstem. Aby przedstawić błąd w przykładowym kodzie, umieściłem znaki tyldy w miejscu wystąpienia błędu:

```
let str = 'not a number';
let num: number = str;
 // ~~~ Typu 'string' nie można przypisać do typu 'number'
```

Czasem edytuję komunikaty błędów, aby zwiększyć czytelność, lecz nigdy nie usuwam błędu. Jeśli ktoś skopiuje przykład i wklei go do edytora, powinien uzyskać dokładnie te same błędy, ani więcej, ani mniej.

Aby podkreślić brak błędów, korzystam z zapisu *// OK:*

```
let str = 'not a number';
let num: number = str as any;  // OK
```

Każdy powinien móc przesunąć kursor nad symbolem w edytorze, aby sprawdzić typ ustalony przez TypeScript. Aby oznaczyć to w tekście, stosuję komentarze rozpoczynające się od frazy „typem jest":

```
let v = {str: 'hello', num: 42};  // Typem jest { str: string; num: number; }
```

Typ dotyczy pierwszego symbolu w wierszu (w tym przypadku `v`) lub wyniku wywołania funkcji:

```
'four score'.split(' ');  // Typem jest string[]
```

Dokładnie taki sam typ ujrzymy w swoim edytorze. Aby sprawdzić typ uzyskany poprzez wywołanie funkcji, może być konieczne przypisanie wyniku do zmiennej.

Czasem będę korzystał z instrukcji nie wykonujących żadnych działań, tylko w celu przedstawienia typu w określonym wierszu kodu:

```
function foo(x: string|string[]) {
  if (Array.isArray(x)) {
    x;  // Typem jest string[]
  } else {
    x;  // Typem jest string
  }
}
```

Wiersze `x;` służą jedynie do prezentacji typu w każdym odgałęzieniu instrukcji warunkowej. Nie musimy (i nie powinniśmy) umieszczać tego typu instrukcji w swoim kodzie.

O ile nie napisałem inaczej lub jeśli nie wynika to z kontekstu, przykładowy kod można zweryfikować z flagą `--strict`. Wszystkie przykład zostały zweryfikowane z użyciem wersji TypeScript 3.7.0-beta.

Typograficzne konwencje stosowane w tej książce

W tej książce stosowane są następujące konwencje typograficzne:

Kursywa
: Oznacza nowe terminy, adresy URL, adresy e-mail, nazwy plików i rozszerzenia plików.

`Krój o stałej szerokości`
: Wykorzystywany w listingach oraz w tekście do podkreślenia elementów programu, takich jak nazwy zmiennych lub funkcji, baz danych, typów danych, zmiennych środowiskowych i słów kluczowych.

`Pogrubiony krój o stałej szerokości`
: Prezentuje polecenia lub inny tekst, który użytkownik powinien wpisać.

`Krój o stałej szerokości z kursywą`
: Prezentuje tekst, który czytelnik powinien zastąpić wartością odpowiednią dla jego środowiska lub przez wartości wynikające z kontekstu.

Ten tekst oznacza wskazówkę lub sugestię.

Ten tekst oznacza ogólną uwagę.

Ten tekst oznacza ostrzeżenie.

Używanie przykładowego kodu

Ze strony *https://github.com/danvk/effective-typescript* można pobrać dodatkowe materiały (przykładowy kod, ćwiczenia, itd)..

Na adres *bookquestions@oreilly.com* można wysyłać pytania techniczne oraz zgłaszać wątpliwości dotyczące przykładowego kodu.

Ta książka ma nam pomóc w wykonaniu swoich zadań. Ogólnie rzecz biorąc, przykładowy kod zawarty w tej książce można wykorzystać w swoich programach i dokumentacji. Nie ma potrzeby wnioskowania o zezwolenie, o ile nie zamierzamy wykorzystać znacznych fragmentów kodu. Przykładowo, jeśli piszemy program zawierający kilka fragmentów kodu z tej książki, nie potrzebujemy pozwolenia. Jeśli zamierzamy sprzedać lub opublikować przykłady z książek wydawnictwa O'Reilly musimy uzyskać pozwolenie. Jeśli odpowiemy na pytanie cytując tę książkę i przykładowy kod, nie potrzebujemy pozwolenia. Jeśli umieścimy znaczną ilość przykładowego kodu do dokumentacji swojego produktu, musimy uzyskać pozwolenie.

Doceniamy, lecz nie wymagamy podawania źródeł pochodzenia kodu. Zwykle należy przy tym podać tytuł, autora, wydawnictwo oraz numer ISBN oryginalnego wydania angielskiego. Przykładowo: „*TypeScript. Skuteczne programowanie* autorstwa Dana Vanderkama (O'Reilly). Copyright 2020 Dan Vanderkam, 978-1-492-05374-3".

Jeśli ktoś sądzi, że wykorzystanie przykładowego kodu wymaga uzyskania pozwolenia zachęcamy do kontaktu pod adresem *permissions@oreilly.com*.

O'Reilly Online Learning

Przez ponad 40 lat *O'Reilly Media* oferowało szkolenia technologiczne i biznesowe, przekazywało wiedzę oraz wgląd w technologie, ułatwiając firmom osiągnięcie sukcesu.

Nasza unikalna sieć ekspertów i innowatorów dzieli się wiedzą w książkach, artykułach, na konferencjach oraz na naszej platformie szkoleniowej online. Platforma szkoleniowa online wydawnictwa O'Reilly oferuje dostęp na żądanie do szkoleń na żywo, ścieżek szkoleniowych, interaktywnych środowisk programistycznych oraz obszerną kolekcję zasobów tekstowych i wideo opracowanych przez O'Reilly oraz ponad 200 innych wydawców. Więcej informacji znajduje się pod adresem *http://oreilly.com*.

Kontakt z nami

Uwagi i pytania dotyczące tej książki należy kierować na następujący adres wydawnictwa:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (w USA i Kanadzie)
- 707-829-0515 (połączenia międzynarodowe lub lokalne)
- 707-829-0104 (faks)

Pod adresem *https://oreil.ly/Effective_TypeScript* znajduje się strona WWW poświęcona tej książce. Dostępna jest tu errata, przykłady oraz dodatkowe informacje.

Aby skomentować lub zadać pytanie techniczne dotyczące tej książki, należy napisać na adres *bookquestions@oreilly.com*.

Więcej informacji o naszych książkach, kursach, konferencjach i nowościach można znaleźć na naszej stronie pod adresem *http://www.oreilly.com*.

Znajdź nas na Facebooku: *http://facebook.com/oreilly*

Śledź nas na Twitterze: *http://twitter.com/oreillymedia*

Oglądaj nas na YouTube: *http://www.youtube.com/oreillymedia*

Podziękowania

W pisaniu tej książki pomogło mi wiele osób. Dziękuję Evanowi Martinowi za wprowadzenie mnie do świata TypeScript i pokazanie jak o nim myśleć. Dziękuję Douwe Osinga, za skontaktowanie mnie z wydawnictwem O'Reilly i wsparcie w trakcie tego projektu. Brettowi Slatkinowi dziękuję za porady dotyczące struktury oraz za pokazanie mi, że ktoś kogo znam, mógłby napisać książkę z serii *Skuteczne programowanie*. Dziękuję też Scottowi Meyersowi za opracowanie tego formatu oraz za post „Effective *Effective* Books" na jego blogu, na którym się wzorowałem.

Dziękuję swoim recenzentom, Rickowi Battagline'owi, Ryanowi Cavanaughowi, Borisowi Cherny'emu, Yakovowi Fainowi, Jesse Hallettowi i Jasonowi Killianowi. Dziękuję wszystkim swoim współpracownikom w Sidewalk, którzy przez lata uczyli się ze mną TypeScript. Dziękuję wszystkim w wydawnictwie O'Reilly, którzy doprowadzili do wydania tej książki: Angeli Rufino, Jennifer Pollock, Deborah Baker, Nickowi Adamsowi i Jasmine Kwityn. Dziękuję też zespołowi grupy TypeScript NYC, Jasonowi, Orcie i Kiryłowi oraz wszystkim mówcom. Wiele elementów napisałem czerpiąc inspirację z wykładów wygłoszonych podczas spotkań, o czym wspominam na poniższej liście:

- Inspiracją do napisania Elementu 3 był post na blogu Evana Martina, który wydał się mi szczególnie ciekawy podczas nauki TypeScript.
- Element 7 napisałem zainspirowany wykładem Andersa, dotyczącym typowania strukturalnego oraz relacji `keyof`, który wygłosił na konferencji TSConf 2018, a także wykładem Jessego Halletta, wygłoszonym na spotkaniu grupy TypeScript NYC Meetup w kwietniu 2019.
- Zarówno wskazówki Basarata, jak i pomocne odpowiedzi udzielone przez DeeV i GPicazo w portalu Stack Overflow były podstawą do napisania Elementu 9.
- Element 10 opiera się na podobnych poradach udzielonych w Elemencie 4 książki *Efektywny JavaScript* (edycja polska wydawnictwo Helion).
- Do napisania Elementu 11 zainspirowało mnie masowe zamieszanie wokół tego zagadnienia podczas spotkania grupy TypeScript NYC Meetup w sierpniu 2019.
- Element 13 w dużej mierze opiera się na kilku pytaniach dotyczących różnic między `type` a `interface` zadanych w serwisie Stack Overflow. Jesse Hallett zasugerował sformułowanie dotyczące rozszerzalności.
- Jacob Baskin zachęcał do napisania Elementu 14 i przekazał mi swoje uwagi dotyczące początkowej wersji.
- Do napisania Elementu 19 zainspirowałem się kilkoma przykładami kodu przesłanymi do kategorii r/typescript w serwisie Reddit.
- Element 26 jest oparty na moich wpisach w serwisie Medium oraz na wykładzie, który wygłosiłem na spotkaniu grupy TypeScript NYC Meetup w październiku 2018.
- Element 28 opiera się na popularnej zasadzie stosowanej w języku Haskell („nieprawidłowe stany powinny być niemożliwe do odwzorowania"). Historia lotu Air France 447 została zainspirowana niesamowitym artykułem Jeffa Wise'a z 2011, opublikowanym w *Popular Mechanics*.
- Element 29 dotyczy problemu, z jakim zetknąłem się w przypadku deklaracji typów z biblioteki Mapbox. Jason Killian zasugerował tytuł.
- Porada dotycząca nazewnictwa z Elementu 36 jest popularna, lecz to konkretne sformułowanie zostało zainspirowane krótkim artykułem Dana Northa w książce *97 Things Every Programmer Should Know* (O'Reilly).
- Inspirację do napisania Elementu 37 zaczerpnąłem z wykładu Jasona Killiana na spotkaniu grupy TypeScript NYC Meetup we wrześniu 2017.

- Element 41 opiera się na informacjach o wersji TypeScript 2.1. Termin „ewoluujący typ any" nie jest szeroko rozpowszechniony poza samym kompilatorem TypeScript, jednak wydaje mi się, że warto nazwać ten nietypowy wzorzec.
- Inspirację do napisania Elementu 42 zaczerpnąłem z wpisu na blogu Jesse'go Halletta. Element 43 w dużej mierze wykorzystuje wpis Titiana Cernicova Dragomira w zgłoszeniu problemu nr 33128 w repozytorium TypeScript.
- Element 44 opiera się na pracy Yorka Yao nad narzędziem `type-coverage`. Potrzebowałem takiego narzędzia i okazało się, że istnieje!
- Element 46 opiera się na wykładzie, który wygłosiłem w grudniu 2017 na spotkaniu grupy TypeScript NYC Meetup.
- Element 50 zawdzięcza swoje istnienie wpisowi Davida Sheldricka na blogu *Artsy*, dotyczącemu typów warunkowych. Dzięki niemu zrozumiałem to zagadnienie.
- Do napisania Elementu 51 zainspirował mnie wykład Steve'a Faulknera, czyli southpolesteve, wygłoszony na spotkaniu grupy w lutym 2019.
- Element 52 opiera się na moich wpisach w serwisie Medium oraz pracy nad narzędziem typings-checker, która ostatecznie została uwzględniona w narzędziu dtslint.
- Element 53 powstał na podstawie wpisu Kat Busch w serwisie Medium, dotyczącym różnych typów wyliczeniowych w TypeScript, a także na podstawie tekstu Borisa Cherny'ego z książki *Programming TypeScript* (O'Reilly).
- Treść Elementu 54 jest wynikiem niezrozumienia tego zagadnienia przeze mnie oraz moich współpracowników. Wyjaśnienie znalazłem w komentarzu autorstwa Andersa do PR 12253 w repozytorium TypeScript.
- Dokumentacja MDN jest podstawą Elementu 55.
- Element 56 luźno opiera się na Elemencie 35 książki *Efektywny JavaScript* (Helion).
- Element 58 opiera się na moim doświadczeniu z migracją starzejącej się biblioteki dygraphs.

Wiele wpisów na blogach oraz wykładów, na których oparłem tę książkę znalazłem za pośrednictwem wpisów w kategorii r/typescript w serwisie Reddit. Jestem szczególnie wdzięczny programistom, którzy udostępnili przykładowy kod ułatwiający zrozumienie typowych problemów początkującym użytkownikom TypeScript. Dziękuję Mariusowi Schulzowi za newsletter TypeScript Weekly. Chociaż tylko czasem jest on wysyłany co tydzień, zawsze stanowi doskonałe źródło materiałów i ułatwia poznawanie najnowszych funkcji TypeScript. Dziękuję Andersowi, Danielowi, Ryanowi i całemu zespołowi rozwijającemu TypeScript w firmie Microsoft za rozmowy i porady dotyczące problemów. Większość moich wątpliwości wynikała z niezrozumienia, lecz nic tak nie satysfakcjonuje jak wprowadzenie poprawki przez samego Andersa Hejlsberga od razu po zarejestrowaniu błędu! Na koniec dziękuję Alex za całe wsparcie podczas tego projektu oraz zrozumienie okazane podczas przepracowanych wakacji, poranków, wieczorów oraz weekendów, które poświęciłem na ukończenie tej książki.

Rozdział 1

Poznajemy TypeScript

Zanim przejdziemy do szczegółów, w tym rozdziale poznamy ogólny zarys języka TypeScript. Jakie rządzą nim zasady i jak należy go postrzegać? Jaki ma związek z językiem JavaScript? Czy jego typy mogą przyjmować wartości null czy nie? O co chodzi z tym `any`? Jakieś pomysły?

TypeScript jest dość niezwykłym językiem, ponieważ nie jest uruchamiany w interpreterze (jak Python i Ruby), ani nie kompiluje się do języka niższego poziomu (jak Java i C). Kompiluje się natomiast do innego języka wysokiego poziomu, czyli do kodu JavaScript. Wykonywany jest kod JavaScript, a nie napisany przez nas kod TypeScript. Zatem relacja między TypeScript a JavaScript jest zasadnicza, chociaż może też prowadzić do nieporozumień. Jeśli zrozumiemy zasady tej relacji, ułatwimy sobie drogę do efektywnej pracy z językiem TypeScript.

System typowania w TypeScript także niesie ze sobą nietypowe konsekwencje, które powinniśmy poznać. System typowania jest szczegółowo omówiony w kolejnych rozdziałach, natomiast w tym rozdziale poznamy niektóre niespodzianki z nim związane.

Element 1: Relacja między TypeScript a JavaScript

Jeśli od dawna korzystamy z TypeScript, z pewnością zetknęliśmy się ze stwierdzeniem, że „TypeScript jest nadzbiorem JavaScript" lub „TypeScript jest typowanym nadzbiorem JavaScript". Co to właściwie znaczy? Jak zdefiniować relację między TypeScript a JavaScript? Ze względu na ścisłe połączenie między tymi językami, zrozumienie zachodzących między nimi relacji jest kluczowe w poprawnym korzystaniu z TypeScript.

TypeScript jest nadzbiorem JavaScript pod względem składniowym: o ile nasz program JavaScript nie zawiera błędów składniowych, można go uznać także za program TypeScript. Bardzo możliwe, że podczas weryfikacji typów TypeScript zostaną wykryte pewne problemy w kodzie. Jednak będą one miały inny charakter. TypeScript nadal przetworzy nasz kod i wygeneruje kod JavaScript. (Jest to kolejny kluczowy aspekt relacji, który jest szczegółowo omówiony w Elemencie 3).

Pliki TypeScript mają rozszerzenie *.ts* (lub *.tsx*), zamiast rozszerzenia *.js* (lub *.jsx*) stosowanego w plikach JavaScript. Nie oznacza to, że TypeScript jest zupełnie innym językiem! Ponieważ TypeScript jest nadzbiorem JavaScript, kod z plików *.js* jest kodem języka TypeScript. Zmiana nazwy pliku *main.js* na *main.ts* tego nie zmieni.

Jest to niezwykle przydatne podczas przekształcania istniejącego kodu źródłowego JavaScript na kod TypeScript. Oznacza to, że nie musimy przepisywać kodu do innego języka, aby móc rozpocząć korzystanie z TypeScript i odnieść korzyści z jego stosowania. Inaczej musielibyśmy postąpić, gdybyśmy chcieli przepisać swój kod JavaScript w języku Java. Ten łatwy sposób migracji jest jedną z najlepszych cech języka TypeScript. Więcej informacji na ten temat zawiera Rozdział 8.

Wszystkie programy JavaScript są programami TypeScript, lecz odwrotne stwierdzenie nie jest prawdziwe: istnieją programy TypeScript, które nie są programami JavaScript. Otóż w TypeScript stosuje się dodatkową składnię określającą typy. (Oprócz tego składnia zawiera nieco innych dodatkowych elementów, głównie ze względów historycznych. Patrz Element 53).

Oto przykład poprawnego programu TypeScript:

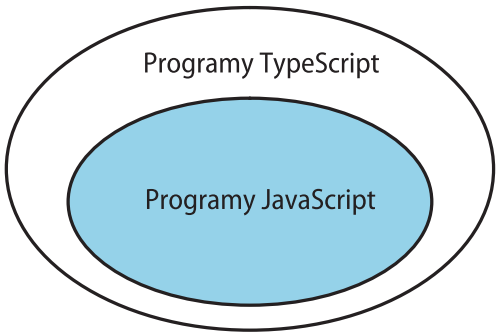
```
function greet(who: string) {
  console.log('Hello', who);
}
```

Jednak po uruchomieniu tego kodu w programie takim jak `node`, który wymaga składni JavaScript, uzyskamy błąd:

```
function greet(who: string) {
                   ^

SyntaxError: Unexpected token :
```

Fragment `: string` służy do oznaczenia typu i jest elementem składni TypeScript. Jeśli skorzystamy z tego typu elementu, wyjdziemy poza czysty kod JavaScript (patrz rysunek 1-1).



Rysunek 1-1 *Cały kod JavaScript jest kodem TypeScript, lecz nie cały kod TypeScript jest kodem JavaScript*

Nie oznacza to jednak, że nie możemy wykorzystać cech TypeScript w przetwarzaniu programów napisanych w czystym języku JavaScript. Możemy! Poniższy przykładowy program JavaScript:

```
let city = 'new york city';
console.log(city.toUppercase());
```

wygeneruje błąd po uruchomieniu:

```
TypeError: city.toUppercase is not a function
```

Ten program nie zawiera oznaczenia typów, lecz podczas weryfikacji typów TypeScript zostanie znaleziony problematyczny kod:

```
let city = 'new york city';
console.log(city.toUppercase());
              // ~~~~~~~~~~~ Property 'toUppercase' does not exist on type
              //             'string'. Did you mean 'toUpperCase'?
```

Nie musieliśmy informować TypeScript, że zmienna `city` jest typu `string`: zostało to wywnioskowane na podstawie początkowej wartości. Wnioskowanie typu jest kluczową cechą TypeScript, której poprawne użycie omawia Rozdział 3.

Jednym z celów systemu typowania TypeScript jest wykrywanie kodu, który wygeneruje wyjątek w czasie wykonywania, bez konieczności uruchomienia kodu. Kiedy ktoś mówi, że TypeScript jest „statycznym" systemem typowania, ma na myśli wspomnianą cechę. Podczas weryfikacji typów nie zawsze uda się znaleźć kod, który rzuci wyjątek, ale przynajmniej podejmuje się taką próbę.

Nawet jeśli nasz kod nie rzuci wyjątku, może zwrócić niezamierzone wyniki. TypeScript próbuje wychwycić również niektóre z podobnych problemów. Poniższy przykładowy program JavaScript:

```
const states = [
  {name: 'Alabama', capital: 'Montgomery'},
  {name: 'Alaska',  capital: 'Juneau'},
  {name: 'Arizona', capital: 'Phoenix'},
  // ...
];
for (const state of states) {
  console.log(state.capitol);
}
```

zwróci następujące wyniki:

```
undefined
undefined
undefined
```

Ups! Co się stało? Jest to poprawny program JavaScript (a zatem również TypeScript). Ponadto po jego uruchomieniu nie pojawiły się żadne błędy. Lecz z pewnością spodziewaliśmy się innych wyników. Nawet bez dodawania oznaczeń typów narzędzie do sprawdzania typów w TypeScript wychwyci błąd (i zasugeruje rozwiązanie):

```
for (const state of states) {
  console.log(state.capitol);
                // ~~~~~~~ Property 'capitol' does not exist on type
                //         '{ name: string; capital: string; }'.
                //         Did you mean 'capital'?
}
```

TypeScript może przechwycić błędy, nawet jeśli nie oznaczymy typu. Natomiast jeśli zadbamy o oznaczenia typu, możemy się spodziewać większych korzyści. Otóż oznaczenia typu informują TypeScript o naszych *intencjach*, dzięki czemu można wychwycić sytuacje, gdy kod nie spełnia naszych zamierzeń. Co by się stało, gdybyśmy odwrócili literówkę `capital`/`capitol` z poprzedniego przykładu?

```
const states = [
  {name: 'Alabama', capitol: 'Montgomery'},
  {name: 'Alaska',  capitol: 'Juneau'},
  {name: 'Arizona', capitol: 'Phoenix'},
  // ...
];
for (const state of states) {
  console.log(state.capital);
                // ~~~~~~~ Property 'capital' does not exist on type
                //         '{ name: string; capitol: string; }'.
                //         Did you mean 'capitol'?
}
```

Wcześniej przydatny komunikat błędu teraz jest zupełnie bezużyteczny! Problem polega na tym, że tę samą zmienną zapisaliśmy w dwóch miejscach inaczej, a TypeScript nie wie, który zapis jest poprawny. Może to odgadnąć, ale nie zawsze poprawnie. Rozwiązanie polega na określeniu swoich intencji poprzez jawną deklarację typu zmiennej `states`:

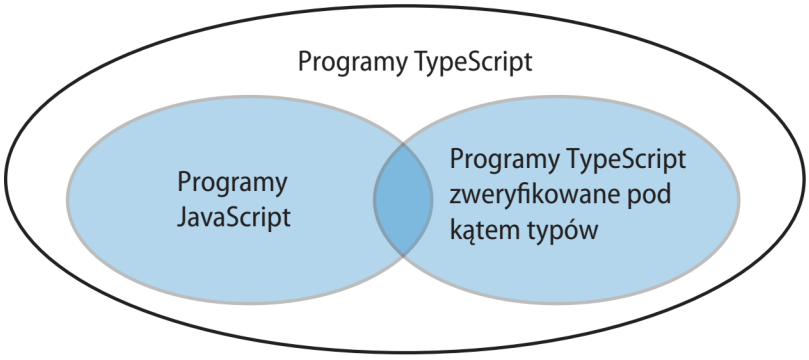
```
interface State {
  name: string;
  capital: string;
}
const states: State[] = [
  {name: 'Alabama', capitol: 'Montgomery'},
                // ~~~~~~~~~~~~~~~~~~~
```

```
  {name: 'Alaska',  capitol: 'Juneau'},
                // ~~~~~~~~~~~~~~~~~
  {name: 'Arizona', capitol: 'Phoenix'},
                // ~~~~~~~~~~~~~~~~~~ Object literal may only specify known
                //     properties, but 'capitol' does not exist in type
                //     'State'.  Did you mean to write 'capital'?
  // ...
];
for (const state of states) {
  console.log(state.capital);
}
```

Teraz błędy poprawnie identyfikują problem, a sugerowana poprawka jest właściwa. Sygnalizując swoje intencje ułatwiliśmy TypeScript wychwycenie innego potencjalnego problemu. Gdybyśmy pomylili się w zapisie zmiennej `capitol` tylko w jednym miejscu tablicy, nie uzyskalibyśmy wcześniej błędu. Jednak po wprowadzeniu oznaczeń typu, zauważymy błąd:

```
const states: State[] = [
  {name: 'Alabama', capital: 'Montgomery'},
  {name: 'Alaska',  capitol: 'Juneau'},
                // ~~~ Did you mean to write 'capital'?
  {name: 'Arizona', capital: 'Phoenix'},
  // ...
];
```

Możemy dodać do diagramu Venna nową grupę programów: programy TypeScript, pomyślnie zweryfikowane pod kątem poprawności typów (patrz rysunek 1-2).



Rysunek 1-2 *Wszystkie programy JavaScript są programami TypeScript. Jednak tylko niektóre programy JavaScript (i TypeScript) przechodzą weryfikację typów.*

Jeśli nie zgadzamy się ze stwierdzeniem, że „TypeScript jest nadzbiorem JavaScript", być może mamy na myśli ten trzeci zbiór programów z diagramu. W praktyce jest to najbardziej odpowiedni sposób codziennego używania języka TypeScript. Zasadniczo podczas

pisania programów TypeScript, staramy się, aby nasz kod został pomyślnie zweryfikowany pod kątem poprawności typów.

System typowania w TypeScript odzwierciedla zachowanie w czasie wykonywania programu JavaScript. Jeśli wcześniej programowaliśmy w języku o bardziej rygorystycznej weryfikacji typów, to może być dla nas zaskoczeniem. Oto przykład:

```
const x = 2 + '3';  // OK, typem jest string
const y = '2' + 3;  // OK, typem jest string
```

Obydwie instrukcje zostaną pomyślnie zweryfikowane pod kątem typów, chociaż wydaje się to dyskusyjne i w wielu innych językach doprowadziłoby do błędów w czasie wykonywania. Jednak te przykłady odzwierciedlają zachowanie w czasie wykonywania kodu JavaScript, kiedy to obydwa wyrażenia zostaną przetworzone do łańcucha „23”.

TypeScript jednak wyznacza pewną granicę. Podczas weryfikacji typów we wszystkich poniższych instrukcjach zostanie wykryty błąd, chociaż w trakcie działania programu nie zostaną zgłoszone wyjątki:

```
const a = null + 7;  // W JS wartość wyniesie 7
       // ~~~~ Operator '+' cannot be applied to types ...
const b = [] + 12;  // W JS zmienna będzie miała wartość '12'
       // ~~~~~~~~ Operator '+' cannot be applied to types ...
alert('Hello', 'TypeScript');  // wyświetli alert "Hello"
            // ~~~~~~~~~~~~ Expected 0-1 arguments, but got 2
```

Wiodącą zasadą systemu typowania TypeScript jest modelowanie zachowania programu JavaScript w trakcie działania. Jednak TypeScript traktuje dziwną składnię powyższych przykładów jak błąd w intencji programisty, a zatem wykracza poza prostą zasadę modelowania zachowania w czasie wykonywania. Podobny mechanizm zaobserwowaliśmy w przykładzie ze zmienną `capital`/`capitol`, w którym program nie wygenerował wyjątku (zwrócił wynik `undefined`) lecz podczas weryfikacji typów znaleziono błąd.

Jak TypeScript określa, kiedy należy modelować zachowanie w trakcie działania programu JavaScript, a kiedy zastosować szersze podejście? Jest to tak naprawdę kwestia gustu. Decydując się na TypeScript ufamy osądowi jego twórców. Jeśli lubimy sumować wartości `null` i `7` lub `[]` i `12`, albo wywoływać funkcje z nadmiarem argumentów, być może powinniśmy zrezygnować z TypeScript!

Jeśli nasz program zostanie pozytywnie zweryfikowany pod kątem typów, to czy mogą się w nim pojawić błędy w czasie wykonywania? Odpowiedź brzmi „tak”. Oto przykład:

```
const names = ['Alice', 'Bob'];
console.log(names[2].toUpperCase());
```

Po uruchomieniu tego kodu, ujrzymy następujący błąd:

```
TypeError: Cannot read property 'toUpperCase' of undefined
```

TypeScript zakłada, że podczas dostępu do tablicy uwzględnimy jej zakres. Ponieważ w tym przypadku nie przestrzegamy tej zasady, pojawi się błąd.

Nieprzechwycone błędy pojawiają się często podczas korzystania z typu `any`, co omawiamy w Elemencie 5 i bardziej szczegółowo w Rozdziale 5.

Przyczyną występowania tych wyjątków jest różnica między rzeczywistym typem wartości, a postrzeganiem tego typu przez TypeScript. System typowania, który może zagwarantować dokładność wyznaczania typów statycznych, wydaje się logiczny. System typowania w TypeScript zdecydowanie nie jest logiczny i nie został w taki sposób zaprojektowany. Jeśli logika jest dla nas istotna, powinniśmy rozważyć inne języki, takie jak Reason lub Elm. Chociaż oferują one większe bezpieczeństwo w czasie wykonywania, to jednak wymagają pewnego poświęcenia: żaden z tych języków nie jest nadzbiorem JavaScript, a zatem migracja okaże się trudniejsza.

Do zapamiętania

- TypeScript jest nadzbiorem JavaScript. Innymi słowy, wszystkie programy JavaScript są programami TypeScript. TypeScript ma dodatkowe elementy składni, a zatem programy TypeScript, ogólnie rzecz biorąc, nie są poprawnymi programami JavaScript.
- TypeScript wprowadza system typowania, który modeluje zachowanie w czasie wykonywania programu JavaScript i próbuje znaleźć kod, który w czasie wykonywania zwróci wyjątki. Nie powinniśmy jednak oczekiwać wychwycenia wszystkich wyjątków. Kod może zostać pozytywnie zweryfikowany pod kątem poprawności typów, lecz rzucić wyjątek w czasie wykonywania.
- Chociaż system typowania TypeScript w dużej mierze modeluje zachowanie JavaScript, niektóre konstrukcje możliwe do zastosowania w JavaScript są zabronione w TypeScript. Przykładem jest wywoływanie funkcji z niewłaściwą liczbą argumentów. Jest to w dużej mierze kwestia gustu.

Element 2: Które opcje TypeScript wykorzystujemy

Czy poniższy kod przejdzie weryfikację pod kątem typów?

```
function add(a, b) {
  return a + b;
}
add(10, null);
```

Jeśli nie wiemy, z których opcji korzystamy, nie odpowiemy na to pytanie! Kompilator TypeScript dysponuje ogromnym zbiorem opcji, sięgającym 100 w czasie pisania tej książki.

Można je ustawić korzystając z wiersza poleceń:

```
$ tsc --noImplicitAny program.ts
```

lub definiując plik konfiguracyjny *tsconfig.json*:

```
{
  "compilerOptions": {
    "noImplicitAny": true
  }
}
```

Powinniśmy raczej korzystać z pliku konfiguracyjnego. Dzięki temu możemy zagwarantować, że współpracownicy i narzędzia poprawnie zrozumieją, w jaki sposób zamierzamy korzystać z TypeScript. Plik konfiguracyjny możemy wygenerować poleceniem `tsc --init`.

Wiele ustawień konfiguracyjnych TypeScript kontroluje miejsce wyszukiwania plików źródłowych oraz rodzaj generowanych wyników. Niektóre kontrolują także podstawowe aspekty samego języka. Są to wysokopoziomowe cechy, na które programiści większości innych języków nie mają wpływu. W zależności od konfiguracji TypeScript może się wydawać zupełnie innym językiem. Aby nasza praca była efektywna, powinniśmy poznać działanie najważniejszych ustawień: `noImplicitAny` i `strictNullChecks`.

`noImplicitAny` określa, czy zmienne muszą mieć przypisane znane typy. Poniższy kod jest poprawny, jeśli opcja `noImplicitAny` jest wyłączona:

```
function add(a, b) {
  return a + b;
}
```

Jeśli w swoim edytorze umieścimy kursor myszy nad symbolem `add`, okaże się, że TypeScript wywnioskował następujący typ tej funkcji:

```
function add(a: any, b: any): any
```

Typy `any` wyłączają weryfikację typu w kodzie korzystającym z tych parametrów. `any` jest przydatnym narzędziem, lecz powinno się go używać ostrożnie. Więcej informacji o typie `any` znajduje się w Elemencie 5 i w rozdziale 3.

Takie typy nazywamy *niejawnymi* typami `any`, ponieważ słowo „any” nie pojawia się w kodzie, a mimo to mamy do czynienia z tymi typami. Po włączeniu opcji `noImplicitAny` w poniższym przykładzie pojawi się błąd:

```
function add(a, b) {
          // ~    Parameter 'a' implicitly has an 'any' type
          //    ~ Parameter 'b' implicitly has an 'any' type
  return a + b;
}
```

Te błędy można poprawić za pomocą jawnych deklaracji typu w postaci zapisu: `any` lub podając konkretny typ:

```
function add(a: number, b: number) {
  return a + b;
}
```

TypeScript jest najbardziej pomocny, gdy zapewnimy informacje o typie, dlatego powinniśmy stosować opcję `noImplicitAny` zawsze, gdy jest to możliwe. Gdy przyzwyczaimy się do deklarowania typów wszystkich zmiennych, TypeScript z wyłączoną opcją `noImplicitAny` będzie się wydawał zupełnie innym językiem.

Opcję `noImplicitAny` należy włączyć w nowych projektach, aby pamiętać o deklarowaniu typów w trakcie pisania kodu. Ułatwi to wykrywanie błędów przez TypeScript, zwiększy czytelność kodu i wzbogaci nasze doświadczenie programistyczne (patrz Element 6). Wyłączenie opcji `noImplicitAny` warto rozważyć tylko podczas przekształcania projektu JavaScript w TypeScript (patrz Rozdział 8).

`strictNullChecks` kontroluje, czy do zmiennej każdego typu można przypisać wartość `null` i `undefined`.

Poniższy kod jest poprawny po wyłączeniu opcji `strictNullChecks`:

```
const x: number = null;  // OK, null jest poprawną wartością typu number
```

lecz wygeneruje błąd po włączeniu opcji `strictNullChecks`:

```
const x: number = null;
//    ~ Type 'null' is not assignable to type 'number'
```

Podobny błąd pojawi się, jeśli wpiszemy `undefined` zamiast `null`.

Jeśli chcemy umożliwić stosowanie `null`, możemy naprawić ten błąd w jawny sposób deklarując swoją intencję:

```
const x: number | null = null;
```

Jeśli nie chcemy zezwalać na wartości `null`, musimy sprawdzić, skąd ta wartość pochodzi i zadbać o jej weryfikację lub dodać asercję:

```
const el = document.getElementById('status');
el.textContent = 'Ready';
// ~~ Object is possibly 'null'

if (el) {
  el.textContent = 'Ready';  // OK, wykluczono wartości null
}
el!.textContent = 'Ready';  // OK, zagwarantowaliśmy, że el nie ma wartości null
```

`strictNullChecks` jest niesłychanie przydatną opcją podczas wychwytywania błędów związanych z wartościami `null` i `undefined`, lecz utrudnia korzystanie z języka. Jeśli zaczynamy nowy projekt, spróbujmy włączyć opcję `strictNullChecks`. Jeśli jednak dopiero uczymy się języka lub migrujemy kod źródłowy JavaScript możemy ją wyłączyć. Zdecydowanie powinniśmy włączyć opcję `noImplicitAny`, zanim włączymy `strictNullChecks`.

Jeśli zdecydujemy się wyłączyć opcję `strictNullChecks`, powinniśmy zwrócić uwagę na niepożądane błędy czasu wykonywania głoszące „undefined is not an object". Każdy z tych błędów przypomina nam, że powinniśmy rozważyć włączenie bardziej rygorystycznej weryfikacji. Zmiana tej opcji będzie coraz trudniejsza w miarę rozrastania się projektu, dlatego nie należy zbyt długo zwlekać z jej włączeniem.

Na semantykę języka wpływa wiele innych opcji (np. `noImplicitThis` i `strictFunctionTypes`), lecz nie mają one tak poważnych skutków jak `noImplicitAny` i `strictNullChecks`. Aby włączyć wszystkie z tych opcji, należy ustawić opcję `strict`. TypeScript po włączeniu opcji `strict` może wychwycić większość błędów, zatem powinniśmy dążyć do takiej konfiguracji.

Musimy wiedzieć, z których opcji korzystamy! Jeśli współpracownik udostępni nam przykładowy kod w TypeScript i nie uda się nam odtworzyć tych samych błędów, upewnijmy się, że opcje naszych kompilatorów są identyczne.

Do zapamiętania

- Kompilator TypeScript korzysta z kilku ustawień wpływających na główne aspekty języka.
- TypeScript należy konfigurować za pomocą pliku *tsconfig.json* zamiast opcji wiersza poleceń.
- Włączmy opcję `noImplicitAny`, chyba że przenosimy projekt JavaScript do TypeScript.
- Włączmy opcję `strictNullChecks`, aby zapobiec występowaniu błędów czasu wykonywania „undefined is not an object".
- Powinniśmy dążyć do włączenia opcji `strict`, aby uzyskać najbardziej rygorystyczną weryfikację typów dostępną w TypeScript.

Element 3: Generowanie kodu jest niezależne od typów

Na wysokim poziomie narzędzie `tsc` (kompilator TypeScript) wykonuje dwa zadania:

- Przekształca kod TypeScript/JavaScript następnej generacji do starszej wersji kodu JavaScript działającej w przeglądarkach („transpilacja").
- Weryfikuje kod pod kątem błędów dotyczących typów.

Zaskakujące jest, że powyższe dwa zadania są od siebie całkowicie niezależne. Innymi słowy, typy zdefiniowane w naszym kodzie nie mają wpływu na kod JavaScript generowany na podstawie kodu TypeScript. Ponieważ wykonywany będzie uzyskany kod JavaScript, oznacza to, że zdefiniowane typy nie mają wpływu na działanie kodu.

Ma to dość nieoczekiwane konsekwencje, których znajomość powinna wpłynąć na nasze oczekiwania wobec możliwości wykorzystania TypeScript.

Kod z błędami typów może wygenerować wyniki

Ponieważ wynik działania kodu nie zależy od weryfikacji typów, okazuje się, że kod zawierający błędy dotyczące typów może zwrócić wyniki!

```
$ cat test.ts
let x = 'hello';
x = 1234;
$ tsc test.ts
test.ts:2:1 - error TS2322: Type '1234' is not assignable to type 'string'

2 x = 1234;
  ~

$ cat test.js
var x = 'hello';
x = 1234;
```

Może to być dość zaskakujące dla programistów znających takie języki jak C lub Java, w których weryfikacja typów i generowanie wyników są ze sobą ściśle powiązane. Wszystkie błędy TypeScript można traktować jak ostrzeżenia występujące we wspomnianych językach: najprawdopodobniej sygnalizują problem i warto je zweryfikować, lecz nie wstrzymają kompilacji.

Kompilacja i weryfikacja typów

Jest to prawdopodobnie źródłem dość nieprecyzyjnych określeń, jakie narosły wokół TypeScript. Często można się zetknąć z określeniami, że kod TypeScript „się nie kompiluje", podczas gdy w rzeczywistości oznacza to występowanie błędów. Jednak z technicznego punktu widzenia nie jest to poprawne stwierdzenie! „Kompilowanie" dotyczy tylko generowania kodu. Zatem zawsze, gdy nasz kod TypeScript jest poprawnym kodem JavaScript (a często nawet gdy nim nie jest), kompilator TypeScript wygeneruje wynik. Lepiej mówić, że w naszym kodzie są błędy, lub, że „nie przeszedł weryfikacji typów".

W praktyce, generowanie kodu mimo występujących w nim błędów jest przydatne. Jeśli tworzymy aplikację internetową, możemy mieć świadomość istnienia błędów w pewnych jej elementach. Jednak ponieważ TypeScript nadal będzie generować kod w obecności błędów, będziemy mogli najpierw przetestować inne elementy aplikacji, zanim zabierzemy się za eliminację błędów.

Powinniśmy dążyć do całkowitej eliminacji błędów podczas zatwierdzania kodu, w przeciwnym razie będziemy musieli pamiętać, które błędy są akceptowalne, a które nie. Jeśli chcemy wyłączyć generowanie kodu w obecności błędów, możemy skorzystać z opcji `noEmitOnError` w pliku *tsconfig.json*, lub z jej odpowiednika w narzędziu do budowania.

Nie można zweryfikować typów TypeScript w czasie wykonywania

Możemy czuć pokusę napisania takiego kodu:

```
interface Square {
  width: number;
}
interface Rectangle extends Square {
  height: number;
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape instanceof Rectangle) {
                    // ~~~~~~~~~ 'Rectangle' only refers to a type,
                    //           but is being used as a value here
    return shape.width * shape.height;
                    //         ~~~~~~ Property 'height' does not exist
                    //                on type 'Shape'
  } else {
    return shape.width * shape.width;
  }
}
```

Warunek `instanceof` jest sprawdzany w czasie wykonywania, lecz `Rectangle` jest typem i nie ma wpływu na zachowanie kodu w czasie wykonywania. Typy TypeScript są „usuwalne": podczas kompilacji do kodu JavaScript wszystkie elementy `interface`, `type` i oznaczenia typów są usuwane z kodu.

Aby ustalić, z jakim typem kształtu mamy do czynienia, musimy znaleźć sposób odtworzenia typu w czasie wykonywania. W tym przypadku możemy sprawdzić istnienie właściwości `height`:

```
function calculateArea(shape: Shape) {
  if ('height' in shape) {
    shape;  // Typem jest Rectangle
    return shape.width * shape.height;
  } else {
    shape;  // Typem jest Square
    return shape.width * shape.width;
  }
}
```

Ten sposób działa, ponieważ testowanie właściwości sprawdza tylko wartości dostępne w czasie wykonywania, lecz nadal umożliwia ustalenie, że zmienna `shape` jest typu `Rectangle`.

Inna metoda polega na zastosowaniu „znacznika", jawnie przechowującego typ w sposób umożliwiający jego odczytanie w czasie wykonywania:

```
interface Square {
  kind: 'square';
  width: number;
}
interface Rectangle {
  kind: 'rectangle';
  height: number;
  width: number;
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape.kind === 'rectangle') {
    shape;  // Typem jest Rectangle
    return shape.width * shape.height;
  } else {
    shape;  // Typem jest Square
    return shape.width * shape.width;
  }
}
```

Typ `Shape` w tym przykładzie jest przykładem „unii z wariantami". Ponieważ znacząco ułatwiają uzyskanie informacji o typie w czasie wykonywania, unie z wariantami są powszechnie wykorzystywane w TypeScript.

Niektóre konstrukcje wprowadzają zarówno typ (który nie jest dostępny w czasie wykonywania) oraz wartość (która jest dostępna). Przykładem jest słowo kluczowe `class`. Innym sposobem poprawy błędu jest zadeklarowanie klas `Square` i `Rectangle`:

```
class Square {
  constructor(public width: number) {}
}
class Rectangle extends Square {
  constructor(public width: number, public height: number) {
    super(width);
  }
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape instanceof Rectangle) {
    shape;  // Typem jest Rectangle
```

```
    return shape.width * shape.height;
  } else {
    shape;  // Typem jest Square
    return shape.width * shape.width;  // OK
  }
}
```

Ten sposób działa, ponieważ składnia `class Rectangle` wprowadza zarówno typ, jak i wartość, zaś składnia `interface` deklaruje jedynie typ.

Zapis `Rectangle` w `type Shape = Square | Rectangle` dotyczy *typu*, natomiast `Rectangle` w `shape instanceof Rectangle` dotyczy *wartości*. Koniecznie należy zrozumieć tę różnicę, która może być dość subtelna. Patrz Element 8.

Operacje na typie nie mogą wpływać na wartości w czasie wykonywania

Załóżmy, że mamy wartość, która może być napisem lub liczbą i chcielibyśmy dokonać normalizacji, aby zawsze uzyskać typ liczbowy. Oto niepoprawna próba, która zostanie pozytywnie zweryfikowana pod kątem typów:

```
function asNumber(val: number | string): number {
  return val as number;
}
```

Gdy spojrzymy na wygenerowany kod JavaScript, zrozumiemy, co właściwie wykonuje ta funkcja:

```
function asNumber(val) {
  return val;
}
```

Nie wykonuje żadnego przekształcenia. Składnia `as number` jest operacją dotyczącą typu, a zatem nie wpływa na zachowanie kodu w czasie wykonywania. Aby znormalizować wartość, musimy sprawdzić typ w czasie wykonywania i dokonać przekształcenia z wykorzystaniem konstrukcji języka JavaScript:

```
function asNumber(val: number | string): number {
  return typeof(val) === 'string' ? Number(val) : val;
}
```

(`as number` jest *asercją typu*. Więcej informacji o poprawnym użyciu tego typu składni znajdziemy w Elemencie 9).

Typy w czasie wykonywania mogą się różnić od zadeklarowanych typów

Czy ta funkcja kiedykolwiek wykona końcową instrukcję `console.log`?

```
function setLightSwitch(value: boolean) {
  switch (value) {
    case true:
      turnLightOn();
      break;
    case false:
      turnLightOff();
      break;
    default:
      console.log(`Obawiam się, że nie mogę tego zrobić.`);
  }
}
```

TypeScript zwykle wychwytuje nieużywany kod, lecz w tym przypadku nie zgłosi zastrzeżeń, nawet po włączeniu opcji `strict`. Kiedy zostanie wykonana ta instrukcja?

Kluczem do tej zagadki jest fakt, że `boolean` jest *zadeklarowanym* typem. Ponieważ jest to typ języka TypeScript, nie zostanie uwzględniony w czasie wykonywania. W kodzie JavaScript użytkownik mógłby przez nieuwagę wywołać funkcję `setLightSwitch` z wartością „`ON`”.

W czystym TypeScript również możemy wywołać tę instrukcję. Być może funkcja zostanie wywołana z wykorzystaniem wartości pochodzącej z wywołania sieciowego:

```
interface LightApiResponse {
  lightSwitchValue: boolean;
}
async function setLight() {
  const response = await fetch('/light');
  const result: LightApiResponse = await response.json();
  setLightSwitch(result.lightSwitchValue);
}
```

Zadeklarowaliśmy, że wynikiem zapytania `/light` jest `LightApiResponse`, lecz nie zadbaliśmy o odpowiednie wymuszenie tego wyniku. Jeśli niepoprawnie skorzystaliśmy z API i `lightSwitchValue` w rzeczywistości jest typu `string`, wówczas w czasie wykonywania do funkcji `setLightSwitch` zostanie przekazana wartość typu `string`. Może też dojść do zmian w API już po wdrożeniu naszego kodu.

TypeScript może być dość trudny w użyciu, jeśli typy czasu wykonywania różnią się od typów zadeklarowanych. W miarę możliwości powinniśmy unikać podobnych sytuacji. Jednak musimy pamiętać, że wartość może mieć inny typ niż zadeklarowany.