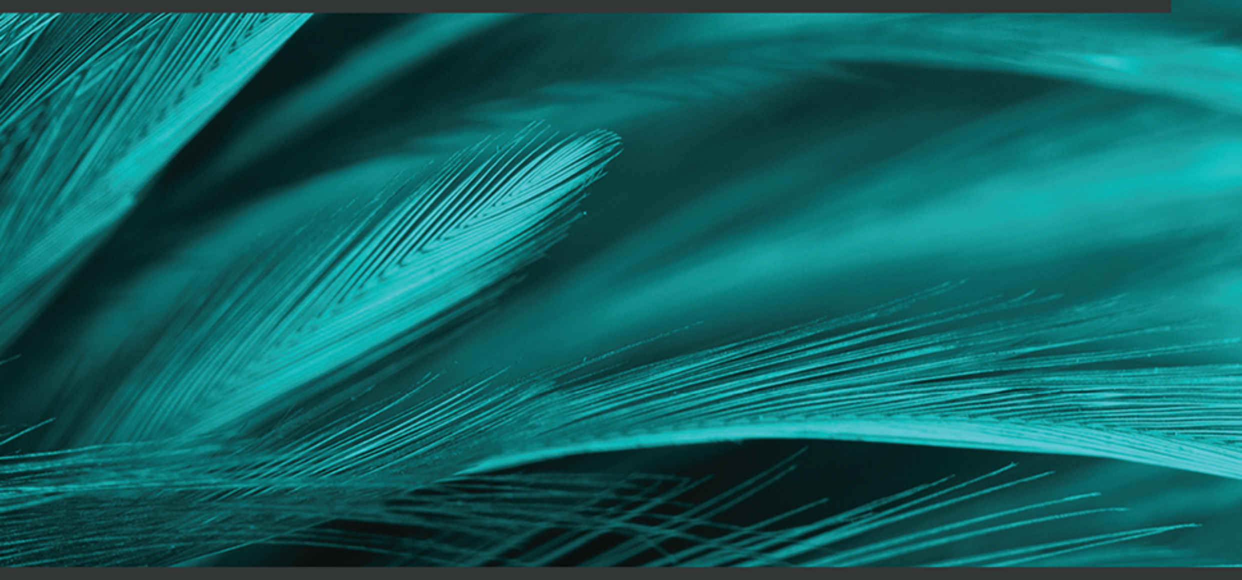
React, TypeScript i Node

Tworzenie aplikacji internetowych typu fullstack



David Choi

Helion

Tytuł oryginału: Full-Stack React, TypeScript, and Node: Build cloud-ready web applications using React 17 with Hooks and GraphQL

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-8392-0

Copyright © Packt Publishing 2020. First published in the English language under the title 'Full-Stack React, TypeScript, and Node – (9781839219931)'.

Polish edition copyright © 2022 by Helion S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/retyno.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/retyno
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorze 11

O recenzencie 11

Wstęp 12

Część I. Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript 17

Rozdział 1. Zrozumieć TypeScript 19

- **Wymagania techniczne** 20
- **Czym jest TypeScript?** 20
- **Dlaczego TypeScript jest niezbędny?** 21
  - Typowanie dynamiczne a statyczne 23
  - Programowanie obiektowe 28
- **Podsumowanie** 31

Rozdział 2. Prezentacja języka TypeScript 33

- **Wymagania techniczne** 34
- **Czym są typy?** 34
  - Jak działają typy? 35
- **Wprowadzenie do typów języka TypeScript** 36
  - Typ any 36
  - Typ unknown 37
  - Typy przecięć i unii 40
  - Typy literałowe 41
  - Nazwy zastępcze typów 42

Kup ksi k

Pole ksi k

Typy wyników funkcji 42
Funkcje jako typy 44
Typ never 45
Klasy i interfejsy 45
Klasy 46
Interfejsy 52
Dziedziczenie 54
Klasy abstrakcyjne 57
Interfejsy 59
Typy generyczne 61
Prezentacja najnowszych możliwości języka i konfigurowania kompilatora 64
Łączenie opcjonalne 64
Scalanie wartości pustych 65
Konfigurowanie TypeScriptu 66
Podsumowanie 67

Rozdział 3. Tworzenie lepszych aplikacji dzięki użyciu możliwości wersji ES6+ języka JavaScript 68

Wymagania techniczne 69
Poznawanie rodzajów zmiennych w ES6 oraz zasięgów w języku JavaScript 70
Poznawanie funkcji strzałkowych 72
Zmienianie kontekstu this 74
Rozproszenie, destrukturyzacja i reszta 76
Rozproszenie, Object.assign oraz Array.concat 77
Destrukturyzacja 79
Reszta 80
Prezentacja wybranych funkcji tablicowych 81
find 81
filter 82
map 83
reduce 84
some oraz every 85
Przedstawienie nowych typów kolekcji 86
Set 86
Map 87
Przedstawienie słów kluczowych async i await 88
Podsumowanie 93

Część II. Nauka tworzenia aplikacji jednostronicowych z użyciem frameworka React 95

Rozdział 4. Przedstawienie koncepcji aplikacji jednostronicowych oraz ich realizacja z użyciem frameworka React 97

Wymagania techniczne 98
Przedstawienie wcześniejszych sposobów tworzenia witryny WWW 98
Cechy i zalety aplikacji jednostronicowych 100

Kup ksi k   Pole ksi k

Jak React pomaga w tworzeniu aplikacji jednostronicowych 101
Atrybuty aplikacji Reacta 102
Podsumowanie 113

Rozdział 5. Tworzenie aplikacji Reacta z wykorzystaniem hooków 114

Wymagania techniczne 115
Wyjaśnienie ograniczeń i problemów związanych ze stosowaniem starych komponentów klasowych 115
Stan 116
Metody cyklu życia 117
Prezentacja hooków Reacta i wyjaśnienie, dlaczego w stosunku do komponentów klasowych są one usprawnieniem 132
Porównanie stosowania komponentów klasowych i hooków 144
Wielokrotne stosowanie kodu 145
Prostota 145
Podsumowanie 146

Rozdział 6. Przygotowywanie projektu za pomocą create-react-app i testowanie go przy użyciu Jest 147

Wymagania techniczne 148
Przedstawienie metod programowania aplikacji Reacta i systemu używanego do ich budowania 148
Narzędzia do zarządzania projektami 149
Transpilacja 156
Repozytoria kodu 158
Testowanie aplikacji Reacta po stronie klienta 160
Atrapy 172
Tworzenie atrap z wykorzystaniem jest.fn 173
Tworzenie atrap komponentów 178
Prezentacja najpopularniejszych narzędzi oraz praktyk tworzenia aplikacji Reacta 185
Visual Studio Code 185
Prettier 186
Debugger Chrome 187
Alternatywne zintegrowane środowiska programistyczne 190
Podsumowanie 191

Rozdział 7. Redux i React Router 192

Wymagania techniczne 192
Zarządzanie stanem przy użyciu Reduxa 193
Reduktory i akcje 195
React Context 205
Prezentacja frameworka React Router 212
Podsumowanie 221

Kup ksi k Pole ksi k

Część III. Tworzenie usług internetowych z użyciem Expressa i GraphQL-a 223

Rozdział 8. Prezentacja tworzenia aplikacji serwerowych z wykorzystaniem Node.js i Expressa 225

Wymagania techniczne 226
Wyjaśnienie sposobu działania środowiska Node 226
Pętla zdarzeń 228
Prezentacja możliwości środowiska Node 229
Instalowanie Node 229
Tworzenie prostego serwera Node 233
Żądania i odpowiedzi 236
Trasowanie 239
Debugowanie 241
Jak Express ułatwia pisanie rozwiązań przeznaczonych dla środowiska Node 248
Przedstawienie możliwości frameworka Express 250
Tworzenie internetowego API przy użyciu Expressa 256
Podsumowanie 259

Rozdział 9. Czym jest GraphQL? 260

Wymagania techniczne 260
Czym jest GraphQL? 261
Schematy GraphQL 263
Definicje typów i resolwery 264
Zapytania, mutacje oraz subskrypcje 270
Podsumowanie 277

Rozdział 10. Konfiguracja projektu Expressa z zależnościami od języków TypeScript i GraphQL 278

Wymagania techniczne 279
Tworzenie projektu Expressa tworzonego w języku TypeScript 279
Dodawanie do projektu GraphQL-a i jego zależności 283
Prezentacja pakietów pomocniczych 290
Podsumowanie 292

Rozdział 11. Czego się nauczysz — aplikacja internetowego forum 293

Analiza aplikacji, którą napiszemy — internetowego forum 294
Analiza uwierzytelniania użytkowników forum 295
Analiza zarządzania wątkami 296
Analiza systemu punktacji wątków 297
Podsumowanie 298

Kup ksi k Pole ksi k

Rozdział 12. Tworzenie klienta Reacta na potrzeby aplikacji internetowego forum 299

Wymagania techniczne 299
Tworzenie wstępnej wersji aplikacji Reacta 300
CSS Grid 302
Granice błędów 309
Warstwa usługi danych 311
Menu nawigacyjne 313
Komponenty związane z uwierzytelnianiem 318
Trasowanie i ekrany aplikacji 328
Ekran główny 329
Ekran wątku i jego wpisów 341
Podsumowanie 360

Rozdział 13. Przygotowywanie stanu sesji przy użyciu Expressa i Redisa 361

Wymagania techniczne 362
Czym jest stan sesji? 362
Przedstawienie magazynu danych Redis 363
Tworzenie stanu sesji z wykorzystaniem Expressa i Redisa 369
Podsumowanie 375

Rozdział 14. Przygotowywanie Postgresa oraz warstwy repozytorium przy wykorzystaniu TypeORM 376

Wymagania techniczne 377
Przygotowanie bazy danych Postgres 377
Przedstawienie mechanizmów odwzorowań obiektowo-relacyjnych na przykładzie TypeORM 382
Tworzenie warstwy repozytorium bazującej na Postgresie i TypeORM 383
Podsumowanie 413

Rozdział 15. Dodawanie schematu GraphQL-a — część 1. 414

Wymagania techniczne 414
Tworzenie definicji typów i resolwerów dla serwerowego kodu GraphQL 415
System punktacji wątków 424
Integracja mechanizmu uwierzytelniania z resolwerami GraphQL-a 428
Przygotowanie hooków Reacta do korzystania z serwera Apollo GraphQL 432
Ekran główny — komponent Main 435
Możliwości związane z uwierzytelnianiem 447
Ekran profilu użytkownika 454
Podsumowanie 460

Rozdział 16. Dodawanie schematu GraphQL-a — część 2. 461

Komponent Thread i jego trasa 461
System punktów 471
Podsumowanie 511

Kup ksi k
Pole ksi k

Rozdział 17. Wdrażanie w chmurze AWS **512**

Wymagania techniczne **513**
Konfiguracja Ubuntu w chmurze AWS **513**
Instalacja Redisa, Postgresa i Node w systemie Ubuntu **520**
Instalacja serwera Redis 521
Instalacja Postgresa 522
Instalacja Node 524
Konfiguracja i wdrażanie aplikacji na serwerze NGINX **525**
Konfigurowanie projektu super-forum-client 532
Konfiguracja serwera NGINX 534
Rozwiązywanie problemów 542
Podsumowanie **543**

Kup ksi k Pole ksi k

Prezentacja języka TypeScript

W tym rozdziale zajmiemy się dokładniejszym omówieniem języka TypeScript. Dowiesz się w nim jak wygląda jawna składnia deklarowania typów oraz poznasz wbudowane typy tego języka i ich przeznaczenie.

Znajdziesz tu także informacje o tym, jak można tworzyć własne typy oraz jak pisać aplikacje zgodne z wytycznymi programowania obiektowego. I w końcu przyjrzymy się także kilku najnowszym możliwościom dodanym do TypeScriptu w ostatnim czasie, takim jak łączenie opcjonalne (ang. *optional chaining*) oraz scalanie wartości pustych (ang. *nullish coalescing*).

Po przeczytaniu tego rozdziału będziesz dysponował dobrą znajomością TypeScriptu, która zapewni Ci możliwość bezproblemowego analizowania i rozumienia kodu pisanego w tym języku. Co więcej, Twoja znajomość języka pozwoli na pisanie w nim kodu o wysokiej jakości, który nie tylko będzie realizować cele stawiane przed aplikacją, lecz także będzie solidny i niezawodny.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- wyjaśnieniem czym są typy;
- przedstawieniem typów języka TypeScript;
- przedstawieniem klas i interfejsów;
- wyjaśnieniem czym jest dziedziczenie;

Kup ksi k Pole ksi k

- przedstawieniem typów generycznych;
- przedstawieniem najnowszych możliwości języka i opcji konfiguracji kompilatora.

Wymagania techniczne

Wymagania techniczne, które musisz spełnić przed przystąpieniem do lektury tego rozdziału, są takie same jak w przypadku rozdziału 1., pt. „Jak rozumieć TypeScript i poprawić swoją znajomość języka JavaScript". Powinieneś dysponować podstawową znajomością języka JavaScript oraz technologii internetowych. Podobnie jak w poprzednim rozdziale, także tu będziemy używali środowiska Node oraz edytora **Visual Studio Code** (**VSCode**).

Kody źródłowe przykładów prezentowanych w tej książce można znaleźć na serwerze wydawnictwa Helion (*https://ftp.helion.pl/przyklady/retyno.zip*), a ich oryginalne, anglojęzyczne wersje — w serwisie GitHub, w repozytorium *https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node*.W tym rozdziale będziemy korzystali z kodów znajdujących się w podkatalogu *rozdzial02* (*Chap2*).

Zanim przystąpisz do lektury, przygotuj środowisko robocze:

1. Przejdź do katalogu *NaukaTypeScriptu* i utwórz w nim podkatalog *rozdzial02*.
2. Uruchom program VSCode i wybierz opcje *File/Open*, a następnie otwórz utworzony przed chwilą katalog *rozdzial02*. Następnie wybierz z menu opcje *View/Terminal* — w efekcie, wewnątrz okna VSCode zostanie wyświetlony panel terminala.
3. W panelu terminala wpisz polecenie `npm init` — użyłeś go już wcześniej w poprzednim rozdziale, by zainicjować nowy projekt `npm` — i zaakceptuj wszystkie ustawienia domyślne.
4. Wykonaj polecenie `npm install typescript` (analogicznie jak w rozdziale 1., pt. „Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript"), aby zainstalować język TypeScript.

Teraz już jesteś gotowy do dalszej lektury.

Czym są typy?

Typy są zestawami reguł nadającymi się do wielokrotnego stosowania. Typ może zawierać właściwości oraz funkcje (określające jego możliwości). Oprócz tego może być wielokrotnie współużytkowany. Kiedy ponownie używamy typu, tworzymy jego **instancję** (ang. *instance*). Oznacza to, że tworzymy egzemplarz danego typu, którego właściwości będą mieć konkretne wartości. W języku TypeScript, czego można było się spodziewać zważywszy na jego nazwę, typy odgrywają bardzo ważną rolę. Przyjrzyjmy się zatem, jak działają typy w języku TypeScript.

Kup ksi k Pole ksi k

Jak działają typy?

Jak już wcześniej wspominałem, typy występują także w języku JavaScript. Liczby, łańcuchy, wartości logiczne, tablice, itd., to wszystko typy dostępne w JavaScripcie. Niemniej jednak, typy te nie są jawnie określane w deklaracjach — środowisko wykonawcze określa je podczas wykonywania kodu. Natomiast w języku TypeScript, typy zazwyczaj określa się w deklaracjach, choć można także pozwolić kompilatorowi, by samemu je odgadywał. Trzeba jednak pamiętać, że typy odgadywane przez kompilator nie zawsze będą tymi, których chcielibyśmy używać, gdyż wskazanie odpowiedniego typu nie zawsze jest oczywiste. Oprócz typów obsługiwanych przez JavaScript, TypeScript udostępnia kilka swoich własnych, unikalnych typów, pozwala także nam tworzyć własne.

Pierwszą rzeczą, jaką musimy zapamiętać odnośnie do typów w języku TypeScript jest to, że są one obsługiwane na podstawie struktury (czy też kształtu, ang. *shape*), a nie na podstawie nazw. Oznacza to, że nazwa typu nie ma wielkiego znaczenia dla kompilatora TypeScriptu — liczą się właściwości, jakie dany typ zawiera.

Przyjrzyjmy się poniższemu przykładowi:

1. Utwórz plik o nazwie *shape.ts* i zapisz w nim następujący kod:

```
class Person {
    name: string;
}
const jill: { name: string } = {
    name: "Julka"
};
const person: Person = jill;
console.log(person);
```

   Przede wszystkim powinieneś zwrócić uwagę w tym kodzie na klasę `Person`, mającą jedną właściwość o nazwie `name`. **Poniżej klasy tworzymy zmienną o nazwie** `jill`, której typ określiliśmy jako `{ name: string }`. Rozwiązanie to wygląda nieco dziwacznie, gdyż zastosowana deklaracja typu nie jest nazwą, bardziej przypomina definicję typu. Jednak z punktu widzenia kompilatora nie stanowi to żadnego problemu, więc nie zgłosi on żadnych uwag. W języku TypeScript nic nie stoi na przeszkodzie, by w jednym miejscu zdefiniować i jednocześnie zadeklarować typ. W dalszej części kodu tworzymy zmienną `person` typu `Person`, w której zapisujemy wartość zmiennej `jill`. Także w tym przypadku kompilator nie zgłasza żadnych problemów, więc wydaje się, że wszystko jest w porządku.

2. Teraz skompiluj ten plik, uruchom go i przekonaj się, co się stanie. W tym celu w panelu terminala VSCode wpisz następujące polecenia:

```
tsc shape
node shape
```

   Wykonanie tych poleceń powinno wygenerować wyniki przedstawione na rysunku 2.1.

Kup ksi k Pole ksi k

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc shape

H:\NaukaTypeScriptu\rozdzial02>node shape
{ name: 'Julka' }

H:\NaukaTypeScriptu\rozdzial02>
```

Rysunek 2.1. Wyniki wykonania skryptu shape

Jak widać, kod można skompilować i wykonać bez najmniejszych problemów. Ten przykład pokazuje, że kompilator języka TypeScript zwraca uwagę na strukturę typów i całkowicie pomija ich nazwy. W następnych rozdziałach, kiedy będziemy dokładniej zajmować się typami TypeScriptu, przekonasz się, dlaczego pamiętanie o tej zasadzie działania typów w tym języku ma tak duże znaczenie.

Wprowadzenie do typów języka TypeScript

W tym podrozdziale zajmiemy się wybranymi z podstawowych typów dostępnych w języku TypeScript. Korzystanie z nich sprawi, że kompilator będzie kontrolował typy i wyświetlał komunikaty o błędach, które pozwolą nam na poprawianie pisanego kodu. Oprócz tego, stosowanie tych typów zapewni nam możliwość przekazania innym programistom w zespole informacji o naszych intencjach. Czytaj więc dalej, aby przekonać się, jak działają typy.

Typ any

`Any` jest typem dynamicznym, charakteryzuje się tym, że można go zastąpić dowolnym innym typem. A zatem, jeśli zadeklarujemy zmienną typu `any`, będziemy mogli przypisać jej dowolną wartość, a później dowolnie ją zmieniać. W efekcie oznacza to, że zmienna typu `any` nie ma żadnego typu, a kompilator nie będzie jej sprawdzał. I to jest najważniejsza informacja, którą należy zapamiętać odnośnie do typu `any` — kompilator nie będzie go sprawdzał ani ostrzegał nas o potencjalnych problemach związanych ze zmiennymi tego typu. Dlatego, jeśli to tylko możliwe, należy unikać jego stosowania. Można uznać, że to nieco dziwne, że język zaprojektowany pod kątem statycznej kontroli typów udostępnia taką możliwość, jednak okazuje się, że w niektórych okolicznościach jest ona niezbędna.

W dużych aplikacjach może się zdarzyć, że programista nie zawsze będzie miał kontrolę nad typami danych trafiających do jego kodu. Na przykład, jeśli programista pobiera dane korzystając z API jakiejś usługi internetowej, to typ zwracanych danych będzie określany przez jakiś inny zespół lub nawet przez programistów innej firmy. Dokładnie to samo dotyczy rozwiązań korzystających z mechanizmów współdziałania, w których nasz kod może być uzależniony do kodu napisanego w jakimś innym języku programowania — zdarza się

Kup ksi k Pole ksi k

tak choćby w sytuacjach, kiedy firma, podczas pisania nowego systemu w jakimś języku programowania, korzysta ze starego systemu, napisanego w innym języku.

Ważne jest, by nie nadużywać typu `any`. Powinieneś zwracać baczną uwagę, by stosować go jedynie w sytuacjach, kiedy nie ma żadnego innego rozwiązania — na przykład, kiedy informacje o typie nie są jasne lub kiedy mogą się zmieniać. Istnieje jednak kilka alternatyw dla stosowania typu `any`. W zależności od okoliczności możemy na przykład skorzystać z interfejsów, typów generycznych, unii, bądź też z typu `unknown`. W następnym punkcie zajmiemy się ostatnią z tych możliwości, typem `unknown`, a pozostałe opiszę w dalszej części rozdziału.

Typ unknown

Typ `unknown` (nieznany) został wprowadzony w wersji 3 języka TypeScript. Jest on podobny do typu `any` pod tym względem, że po zadeklarowaniu zmiennej tego typu można do niej przypisać wartość dowolnego typu. Tę wartość można następnie zmienić — i to także na wartość dowolnego innego typu. A zatem, w zmiennej moglibyśmy początkowo zapisać łańcuch znaków, a później zmienić jej wartość na liczbę. Jednak bez wcześniejszego sprawdzenia faktycznego typu takiej zmiennej nie można wywoływać żadnych jej składowych ani zapisywać jej jako wartości innej zmiennej. Jedyną sytuacją, kiedy możemy przypisać zmienną typu `unknown` innej zmiennej bez sprawdzania typu przypisywanej wartości, jest ta, gdy przypisujemy ją innej zmiennej typu `unknown` lub `any`.

Przeanalizujmy teraz przykład zastosowania typu `any`, a później pokażę, dlaczego stosowanie typu `unknown` jest lepsze od korzystania z typu `any` (w rzeczywistości jest to nawet zalecane przez zespół twórców języka TypeScript):

1. W pierwszej kolejności przyjrzymy się przykładowi problemowi związanemu z wykorzystaniem typu `any`. W edytorze VSCode utwórz plik *any.ts* i zapisz w nim następujący kod:

   ```
   let val: any = 22;
   val = "to jest łańcuch";
   val = new Array();
   val.push(33);

   console.log(val);
   ```

   Jeśli teraz skompilujesz i uruchomisz ten kod, wygeneruje on wyniki przedstawione na rysunku 2.2.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc any

H:\NaukaTypeScriptu\rozdzial02>node any
[ 33 ]
```

Rysunek 2.2. Wyniki wykonania skryptu any

Kup ksi k   Pole ksi k

2. Ponieważ zmienna `val` jest typu `any`, możemy zapisać w niej dowolną wartość, a następnie wywołać metodę `push`, gdyż jest to metoda typu `Array`. Niemniej jednak jest to oczywiste tylko dla nas, programistów, gdyż wiemy, że typ `Array` dysponuje metodą o nazwie `push`. A co by się stało, gdybyśmy przez przypadek spróbowali wywołać jakąś metodę, której typ `Array` nie udostępnia? Zmodyfikuj kod pliku *any.ts* zgodnie z kolejnym przykładem:

   ```
   let val: any = 22;
   val = "to jest łańcuch";
   val = new Array();
   val.nieistniejacametoda(33);

   console.log(val);
   ```

3. A teraz ponownie spróbuj skompilować plik *any.ts*:

   ```
   tsc any
   ```

   Przekonasz się, że — niestety — i tym razem kompilator nie zgłosił żadnych problemów, gdyż zadeklarowanie zmiennej typu `any` sprawia, że kompilator nie będzie sprawdzał jej typu. Oprócz tego straciliśmy także możliwości zapewniane przez mechanizm IntelliSense w VSCode, a konkretnie: kolorowanie kodu oraz sprawdzanie i wyświetlanie błędów w edytorze. Dopiero po wykonaniu kodu uzyskamy jakikolwiek sygnał, że występują w nim jakieś problemy; a to niestety nie jest to, o co nam chodziło. Jeśli teraz wykonamy skompilowany kod, natychmiast pojawią się komunikaty o błędzie, podobne do tych przedstawionych na rysunku 2.3.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc any.ts

H:\NaukaTypeScriptu\rozdzial02>node any.js
H:\NaukaTypeScriptu\rozdzial02\any.js:4
val.nieistniejacametoda(33);
    ^

TypeError: val.nieistniejacametoda is not a function
    at Object.<anonymous> (H:\NaukaTypeScriptu\rozdzial02\any.js:4:5)
    at Module._compile (node:internal/modules/cjs/loader:1101:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1153:10)
    at Module.load (node:internal/modules/cjs/loader:981:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:79:12)
    at node:internal/main/run_main_module:17:47
```

Rysunek 2.3. Błąd w skrypcie any

W przypadku prostego skryptu, takiego jak ten przedstawiony powyżej, popełnienie podobnego błędu jest raczej mało prawdopodobne; jednak w dużych aplikacjach taki błąd można popełnić bardzo łatwo — może to być choćby prosty błąd typograficzny.

Kup ksi k   Pole ksi k

A teraz przyjrzyjmy się podobnemu przykładowi, w którym tym razem zastosujemy typ `unknown`:

1. W pierwszej kolejności umieść w komentarzach cały dotychczasowy kod zapisany w pliku *any.ts* i usuń plik *any.js* (będziemy używać zmiennych o tych samych nazwach, więc gdybyś tego nie zrobił, kompilator mógłby zgłosić błąd związany z występowaniem konfliktów).

> W dalszej części książki poznasz tak zwane przestrzenie nazw. Pozwalają one unikać takich konfliktów, jednak jak na razie jest jeszcze trochę za wcześnie, byś ich używał.

2. Następnie utwórz nowy plik o nazwie *unknown.ts* i zapisz w nim następujący fragment kodu:

```
let val: unknown = 22;
val = "to jest łańcuch";
val = new Array();
val.push(33);

console.log(val);
```

   Po jego wpisaniu zauważysz zapewne, że VSCode od razu wyświetli błąd związany z wywołaniem funkcji `push`. To nieco dziwne, gdyż typ `Array` udostępnia przecież tę metodę. Jednak wyświetlenie tego błędu pokazuje, że typ `unknown` działa prawidłowo. Możesz sobie wyobrazić, że `unknown` to coś, co bardziej przypomina etykietę niż typ danych, oraz że pod tą etykietą jest ukryty faktyczny typ. Jednak kompilator nie jest w stanie samodzielnie określić tego typu, dlatego też sami musimy zadbać o to, by udowodnić kompilatorowi, że zmienna jest konkretnego typu.

3. Używamy strażników typów by upewnić się, że zmienna `val` jest konkretnego typu:

```
let val: unknown = 22;
val = "to jest łańcuch";
val = new Array();
if (val instanceof Array) {
    val.push(33);
}

console.log(val);
```

   Jak widać, w tej wersji kodu wywołanie metody `push` umieściliśmy wewnątrz instrukcji warunkowej, sprawdzającej, czy `val` jest instancją typu `Array`.

4. Kiedy już upewnimy się, że warunek będzie spełniony, wywołanie metody `push` zostanie wykonane bez wyświetlania żadnych komunikatów o błędach, jak pokazałem na rysunku 2.4.

Ten mechanizm jest dość niewygodny, gdyż zmusza nas do sprawdzania typu za każdym razem, gdy chcemy odwołać się do składowej obiektu. Jednak pomimo to jest on preferowany względem stosowania typu `any` i znacznie od niego bezpieczniejszy, gdyż pozwala na sprawdzanie kodu przez kompilator.

Kup ksi k
Pole ksi k

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc unknown

H:\NaukaTypeScriptu\rozdzial02>node unknown
[ 33 ]
```

Rysunek 2.4. Zastosowanie typu unknown

Typy przecięć i unii

Zapewne pamiętasz, że na początku tego podrozdziału zaznaczyłem, że kompilator TypeScriptu koncentruje się na strukturze typu, a nie na jego nazwie? Ten sposób działania sprawia, że język TypeScript udostępnia specyficzny rodzaj typów nazywanych **przecięciami** (ang. *intersection types*), określanych także czasami jako *intersekcje*. Oznacza to, że TypeScript pozwala programistom na „tworzenie typów” poprzez scalanie kilku odrębnych typów w jedną, nową całość. Ponieważ dość trudno to sobie wyobrazić, więc najlepiej będzie, jak przedstawię odpowiedni przykład. Jeśli spojrzysz na poniższy fragment kodu, zobaczysz w nim zmienną o nazwie `obj`, z którą są skojarzone dwa typy. Zapewne pamiętasz, że w TypeScripcie możemy nie tylko użyć nazwanego typu w deklaracji zmiennej, lecz także jednocześnie zdefiniować i zadeklarować zmienną określonego typu. W poniższym przykładzie, każdy z użytych typów jest odrębnym typem, jednak zastosowanie operatora `&` sprawia, że zostaną połączone w jeden, nowy typ:

```
let obj: { name: string } & { age: number } = {
    name: 'Tomek',
    age: 25
}
```

Spróbujmy teraz wykonać ten kod i wyświetlić wyniki w panelu terminala. Utwórz nowy plik o nazwie *intrsection.ts* i zapisz w nim poniższy fragment kodu:

```
let obj: { name: string } & { age: number } = {
    name: 'Tomek',
    age: 25
}

console.log(obj);
```

Kiedy skompilujesz i uruchomisz ten przykład, przekonasz się, że zostanie wyświetlony obiekt zawierający obie właściwości — `name` oraz `age` (jak pokazałem na rysunku 2.5).

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc intersection

H:\NaukaTypeScriptu\rozdzial02>node intersection
{ name: 'Tomek', age: 25 }
```

Rysunek 2.5. Wyniki zastosowania przecięcia typów

Jak widać, zarówno mechanizm IntelliSense, jak i kompilator prawidłowo obsługują powyższy kod, a wynikowy obiekt dysponuje obiema właściwościami. Tak właśnie działa przecięcie typów.

Kolejnym, dość podobnym rodzajem typu są tak zwane **unie** (ang. *union type*). W tym przypadku, zamiast scalać kilka typów w jeden, wybierany jest jeden z kilku dostępnych. Zobaczmy na przykładzie, jak to działa. Utwórz nowy plik o nazwie *union.ts* i zapisz w nim poniższy fragment kodu:

```
let unionObj: null | { name: string } = null;
unionObj = { name: 'Janek' };

console.log(unionObj);
```

Zastosowanie znaku | sprawiło, że zmienna `unionObj` została zadeklarowana jako typ `null` lub `{ name: string }`. Jeśli teraz skompilujesz i uruchomisz ten przykład, przekonasz się, że faktycznie w zmiennej można zapisywać wartości obu tych typów. Oznacza to, że w zmiennej będzie można zapisywać bądź to wartość `null`, bądź też obiekty typu `{ name: string }`.

Typy literałowe

Typy literałowe (ang. *literal types*) są podobne do unii, jednak korzystają ze zbioru z góry określonych łańcuchów lub liczb. Poniżej przedstawiłem przykład zastosowania tego rodzaju typu korzystającego z łańcuchów; jest on na tyle prosty, że nie wymaga dodatkowych wyjaśnień. Jak widać, typ składa się z grupy określonych łańcuchów. Oznacza to, że w zmiennej takiego typu będzie można zapisać wyłącznie jeden z łańcuchów wymienionych w typie.

```
let literal: "Tomek" | "Linda" | "Jarek" | "Sylwia" = "Linda";
literal = "Sylwia";

console.log(literal);
```

Jak widać, kompilator akceptuje dowolną z wymienionych wartości, jak również pozwala ją zmieniać na inną z dopuszczalnych. Jeśli jednak spróbujesz przypisać zmiennej wartość, która nie została wymieniona w typie, kompilator zgłosi błąd. Przyjrzyjmy się, jak to działa — zaktualizuj kod przykładu, przypisując zmiennej wartość `"Janek"`:

```
let literal: "Tomek" | "Linda" | "Jarek" | "Sylwia" = "Linda";
literal = "Sylwia";
literal = "Janek";
console.log(literal);
```

Tym razem próbujemy przypisać zmiennej typu literałowego wartość `"Janek"`, co sprawi, że podczas próby kompilacji kodu zostanie zgłoszony błąd (patrz rysunek 2.6).

Dostępne są także typy literałowe, których dopuszczalnymi wartościami są nie łańcuchy, lecz liczby; pomijając tę różnicę, działają one tak samo.

Kup ksi k Pole ksi k

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS                    1: cmd

H:\NaukaTypeScriptu\rozdzial02>tsc literal
literal.ts:3:1 - error TS2322: Type '"Janek"' is not assignable to type '"Tomek" | "Linda" | "Jarek" | "Sylwia"'.

3 literal = "Janek";
  ~~~~~~~

Found 1 error.
```

Rysunek 2.6. Błąd przypisania wartości do zmiennej typu literałowego

Nazwy zastępcze typów

Nazwy zastępcze typów (określane także jako *aliasy*), są bardzo często stosowane w języku TypeScript. Pozwalają one nadawać typom inne nazwy i w większości przypadków są stosowane w celu upraszczania długich i złożonych nazw typów. Poniższy przykład pokazuje sposób definiowania oraz użycia nazwy zastępczej typu:

```
type Points = 20 | 30 | 40 | 50;
let score: Points = 20;

console.log(score);
```

W tym przykładzie tworzymy długi typ literałowy określający kilka dopuszczalnych wartości liczbowych i przypisujemy mu krótszą nazwę `Points`. Następnie deklarujemy zmienną `score` typu `Points` i przypisujemy jej wartość `20`, czyli jedną z dopuszczalnych wartości typu `Points`. Oczywiście, jeśli spróbujemy przypisać zmiennej jakąkolwiek inną wartość, kompilator zgłosi błąd.

Kolejny przykład przedstawia określanie nazwy zastępczej dla typu literału obiektowego:

```
type ComplexPerson = {
    name: string,
    age: number,
    birthday: Date,
    married: boolean,
    address: string
}
```

Deklaracja tego typu jest bardzo długa, a sam typ nie określa nazwy, czym, między innymi, różni się od klasy, która by ją miała, dlatego też przypisujemy mu nazwę zastępczą. Nazwy zastępcze można określać dla dowolnych typów języka TypeScript, w tym także dla funkcji i typów generycznych, którym przyjrzymy się w dalszej części tego rozdziału.

Typy wyników funkcji

W celu uzupełnienia zagadnień związanych z typami chciałbym także przedstawić przykład deklarowania typu wartości zwracanej przez funkcję. Wygląda ona bardzo podobnie do typowej deklaracji zmiennej. Zacznij od utworzenia nowego pliku o nazwie *functionReturn.ts*, a następnie zapisz w nim poniższą funkcję:

Kup ksi k   Pole ksi k

```
function runMore(distance: number): number {
    return distance + 10;
}
```

Funkcja `runMore` pobiera jeden parametr typu `number` i zwraca wartość także typu `number`. Deklaracja parametru wygląda dokładnie tak samo, jak deklaracja każdej innej zmiennej; natomiast typ wartości zwracanej przez funkcję jest zapisywany za nawiasem zamykającym listę parametrów. Jeśli funkcja niczego nie zwraca, to możemy bądź to całkowicie pominąć deklarację typu wyniku, bądź też zadeklarować go jako `void`.

Przyjrzymy się teraz przykładowi funkcji zwracającej wynik typu `void`. Umieść zdefiniowaną wcześniej funkcję `runMore` w komentarzu, dodaj do pliku zamieszczony poniżej kod, a następnie skompiluj go i wykonaj:

```
function eat(calories: number) {
    console.log("Zjadłem " + calories + " kalorii.");
}
function sleepIn(hours: number): void {
    console.log("Spałem " + hours + " godzin.");
}

let ate = eat(100);
console.log(ate);
let slept = sleepIn(10);
console.log(slept);
```

Żadna z tych nowych funkcji nie zwraca wyniku, a ich działanie sprowadza się do wyświetlenia na konsoli wartości przekazanego parametru (patrz rysunek 2.7).

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc functionReturn

H:\NaukaTypeScriptu\rozdzial02>node functionReturn
Zjadłem 100 kalorii.
undefined
Spałem 10 godzin.
undefined

H:\NaukaTypeScriptu\rozdzial02>
```

Rysunek 2.7. Funkcje, które niczego nie zwracają

Jak pokazują wyniki wykonania skryptu, wywołania `console.log` umieszczone wewnątrz funkcji są wykonywane; jednak próba pobrania i użycia wartości wynikowej zwracanej przez każdą z tych funkcji kończy się wyświetleniem `undefined`, gdyż żadna z funkcji niczego nie zwraca.

A zatem, deklaracje typu wyniku funkcji są bardzo podobne do deklaracji zmiennych. A teraz przyjrzyjmy się bliżej kolejnemu zagadnieniu, a mianowicie: stosowaniu funkcji jako typów.

Kup ksi k Pole ksi k

Funkcje jako typy

Może się to wydawać nieco dziwne, jednak w języku TypeScript typem może być także cała sygnatura funkcji. W poprzednim punkcie pokazałem, w jaki sposób funkcja może pobierać parametry określonych typów oraz zwracać wynik określonego typu. Ta definicja jest właśnie określana sygnaturą funkcji. W języku TypeScript, taka sygnatura może być także używana jako typ właściwości obiektów.

Przeanalizujmy przykład takiego rozwiązania. Utwórz plik o nazwie *functionSignature.ts*, a następnie zapisz w nim poniższy kod:

```
type Run = (miles: number) => boolean;
let runner: Run = function (miles: number): boolean {
    if(miles > 10){
        return true;
    }
    return false;
}

console.log(runner(9));
```

W pierwszym wierszu tego przykładu znajduje się typ funkcyjny, którego użyjemy w dalszej części kodu. Nazwę zastępczą `Run` tego typu zdefiniowałem tylko po to, by uprościć stosowanie długiej sygnatury funkcji. Sam typ funkcyjny ma następującą postać: `(miles: number) => boolean`. Wygląda to dość dziwnie, jednak w rzeczywistości ten typ jest jedynie nieco skróconą sygnaturą funkcji. W tej deklaracji typu należy zwrócić uwagę na nawiasy, w których jest zapisana lista parametrów, symbol `=>` informujący, że mamy do czynienia z funkcją, oraz umieszczony za tym symbolem typ wyniku.

W kolejnym wierszu znajduje się deklaracja zmiennej `runner` typu `Run`, która oczywiście będzie funkcją. Funkcja, którą przypisujemy tej zmiennej sprawdza, czy biegacz przebiegł więcej niż 10 mil i zwraca `true`, jeśli ten warunek został spełniony, lub `false` w przeciwnym przypadku. Na samym końcu kodu znajduje się wywołanie metody `console.log`, które wyświetla wynik wywołania funkcji. Zamieszczony poniżej rysunek 2.8 przedstawia wyniki kompilacji i wykonania tego przykładu:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc functionSignature

H:\NaukaTypeScriptu\rozdzial02>node functionSignature
false

H:\NaukaTypeScriptu\rozdzial02>
```

Rysunek 2.8. Zastosowanie typu funkcyjnego

Jak widać, wywołanie funkcji `runner` z argumentem `9` sprawi, że funkcja zwróci wartość `false`, co jest zgodne z oczekiwaniami. W przypadku języków korzystających z silnego typowania, ważne jest, by dla wszystkich sposobów zwracania danych w kodzie określać typy tych danych, a takimi sposobami zwracania danych mogą być nie tylko zmienne, lecz także funkcje.

Kup ksi k Pole ksi k

Typ never

Na pierwszy rzut oka ten typ będzie się wydawał dość dziwny. Typ `never` jest używany do oznaczania, że funkcja nigdy nie zwróci wyniku (nie uda się jej zakończyć), bądź też, że zmienna nigdy nie zostanie ustawiona (nie zostanie jej przypisana żadna wartość, nawet `null`). Z pozoru `never` bardzo przypomina typ `void`. Jednak w żadnym wypadku nie są one identyczne. W przypadku typu `void`, działanie funkcji się kończy (i to w dosłownym znaczeniu tego słowa), jednak funkcja nie zwraca żadnego wyniku (zwraca `undefined`, co nie jest żadną wartością). Może się zatem wydawać, że `never` jest całkowicie bezużytecznym typem, okazuje się jednak, że jest on bardzo przydatny do określania naszych intencji.

Przeanalizujmy poniższy przykład. Utwórz plik *never.ts* i zapisz w nim poniższy kod:

```
function oldEnough(age: number): never | boolean {
    if(age > 59) {
        throw Error("Jesteś za stary!");
    }
    if(age <=18){
        return false;
    }
    return true;
}
```

Jak widać, typem wyniku zwracanego przez funkcję zdefiniowaną w tym przykładzie jest unia — `never` lub `boolean`. Okazuje się, że mogliśmy zadeklarować typ wyniku jako `boolean` i powyższy kod i tak by działał prawidłowo. Niemniej jednak, jeśli wiek przekazany do funkcji będzie przekraczał pewną wartość, funkcja zgłosi błąd, informując w ten sposób, że przekazana wartość jest nieoczekiwana. A zatem, ponieważ hermetyzacja jest jedną z podstawowych zasad pisania kodu o wysokiej jakości, korzystne może być wyraźne oznaczenie, że w pewnych okolicznościach funkcja może zawieść i nie zwrócić wartości, bez zmuszania innych do zagłębiania się w szczegóły jej działania. Właśnie tę informację przekazuje użycie typu `never`.

W tym podrozdziale poznałeś wiele wbudowanych typów języka TypeScript. Przekonałeś się także, dlaczego ich stosowanie pozwala poprawić jakość kodu i ułatwia wczesne wykrywanie błędów — już na etapie pisania kodu. W następnym podrozdziale dowiesz się, w jaki sposób możemy używać TypeScriptu do tworzenia własnych typów oraz pisania kodu, który będzie zgodny z zasadami programowania obiektowego.

Klasy i interfejsy

O klasach i interfejsach wspominałem już pobieżnie we wcześniejszych częściach rozdziału. W tym podrozdziale przyjrzymy się im dokładniej i przekonamy, dlaczego pozwalają nam one na tworzenie lepszego kodu. Pod koniec tej części rozdziału będziesz znacznie lepiej przygotowany do pisania kodu bardziej czytelnego, nadającego się do wielokrotnego stosowania i mniej podatnego na występowanie błędów.

Kup ksi k Pole ksi k

Klasy

Na podstawowym poziomie, klasy w języku TypeScript wyglądają tak samo jak klasy w JavaScripcie. Są to pojemniki grupujące powiązane ze sobą pola i metody, których można używać do tworzenia obiektów i wielokrotnie używać. Jednak klasy języka TypeScript zapewniają większe możliwości hermetyzacji, które nie są dostępne w języku JavaScript. Przyjrzyjmy się im na przykładzie.

Utwórz nowy plik o nazwie *classes.ts* i zapisz w nim następujący kod:

```
class Person {
    constructor() {}
    msg: string;
    speak() {
        console.log(this.msg);
    }
}

const tom = new Person();
tom.msg = "cześć";
tom.speak();
```

Ten przykład przedstawia prostą klasę, która — poza statyczną kontrolą typów — niczym nie różni się od klas pisanych w języku JavaScript. W pierwszej kolejności podawana jest nazwa klasy, dzięki której będzie można jej wielokrotnie używać. Następnie deklarujemy konstruktor klasy, służący do inicjalizacji wszelkich pól, którymi klasa dysponuje, oraz wykonywania wszelkich innych czynności niezbędnych do przygotowania instancji danej klasy (przypomnę tylko, że instancja to konkretny egzemplarz klasy, dysponujący unikalnymi wartościami pól). W dalszej części kodu klasy deklarujemy zmienną o nazwie `msg` oraz funkcję o nazwie `speak`, która wyświetla na konsoli wartość zmiennej `msg`. W pozostałej części kodu tworzymy nową instancję naszej klasy, potem zapisujemy w jej polu `msg` łańcuch `"cześć"` i wywołujemy metodę `speak`. A teraz przyjrzyjmy się różnicom pomiędzy klasami w językach TypeScript i JavaScript.

Modyfikatory dostępu

Podkreślałem już wcześniej, że jedną z głównych zasad programowania obiektowego jest hermetyzacja, czyli ukrywanie informacji. Jeśli jeszcze raz przyjrzymy się klasie przedstawionej na ostatnim przykładzie, wyraźnie zauważymy, że nie ukrywamy zmiennej `msg`, a jej wartość jest dostępna dla kodu spoza klasy, który nawet może ją zmieniać. Przekonajmy się zatem, co TypeScript pozwala nam zrobić z tym problemem. Zmodyfikuj kod w pliku *classes.ts* do postaci przedstawionej poniżej:

```
class Person {
    constructor(private msg: string) {}

    speak() {
        console.log(this.msg);
    }
}
```


Kup ksi k Pole ksi k

```
const tom = new Person("cześć");
// tom.msg = "cześć";
tom.speak();
```

Jak widać, zmodyfikowaliśmy konstruktor, dodając do niego parametr poprzedzony słowem kluczowym `private`. Ten sposób deklarowania parametrów konstruktora, w którym są do nich dodawane modyfikatory dostępu, pozwala nam zrobić kilka rzeczy w jednym wierszu kodu. Przede wszystkim zapis ten informuje kompilator, że klasa dysponuje polem o nazwie `msg` typu `string`, które ma być polem prywatnym (co określa modyfikator `private`). Zazwyczaj deklaracje tego typu zapisuje się w wierszu poniżej lub powyżej konstruktora, co także jest prawidłowym rozwiązaniem, jednak TypeScript udostępnia zapis skrócony, polegający właśnie na dodaniu parametru do konstruktora. Co więcej, dodanie tego parametru zapewnia nam możliwość ustawienia wartości pola `msg` w czasie tworzenia instancji klasy — wystarczy użyć wywołania o postaci `new Person("cześć")`.

A właściwie co daje poprzedzenie pola klasy modyfikatorem `private`? Otóż użycie `private` sprawia, że pole nie będzie dostępne dla kodu spoza tej klasy. W efekcie, używana wcześniej instrukcja `tom.msg = "cześć"` przestanie działać, a próba jej skompilowania spowoduje zgłoszenie błędu. Możesz się o tym przekonać, usuwając znaki komentarza na początku tego wiersza kodu. Kiedy to zrobisz i spróbujesz skompilować plik, zostanie wyświetlony komunikat o błędzie przedstawiony na rysunku 2.9.

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc classes
classes.ts:10:5 - error TS2341: Property 'msg' is private and only accessible within class 'Person'.

10 tom.msg = "cześć";
       ~~~

Found 1 error.
```

Rysunek 2.9. Błąd odwołania do prywatnego pola klasy

Jak widać, komunikat informuje, że do składowej prywatnej `msg` nie można odwoływać się spoza kodu klasy. W powyższym przykładzie użyliśmy modyfikatora, by zmienić widoczność pola, jednak modyfikatorów dostępu można także używać do określania widoczności dowolnych składowych klas, czyli nie tylko pól, lecz także funkcji.

> Jak już wspominałem wcześniej, w języku ECMAScript 2020 będzie można tworzyć pola prywatne; będzie do tego służył symbol #. Jednak w ten sposób będzie można tworzyć jedynie pola, a nie metody prywatne; poza tym jest to tak nowy standard, że jego obsługa w przeglądarkach jest aktualnie jeszcze dość ograniczona.

Przyjrzyjmy się teraz kolejnemu modyfikatorowi dostępu: `readonly`. Jego działanie jest bardzo proste: sprawia, że pole, po początkowym ustawieniu wartości w konstruktorze, będzie przeznaczone tylko do odczytu. Wprowadź do naszego przykładu kolejną modyfikację, dodając do deklaracji pola `msg` modyfikator `readonly`:

Kup ksi k   Pole ksi k

```
class Person {
    constructor(private readonly msg: string) {}

    speak() {
        this.msg = "mówię: " + this.msg;
        console.log(this.msg);
    }
}

const tom = new Person("cześć");
// tom.msg = "cześć";
tom.speak();
```

Kiedy wpiszesz tę nową wersję kodu, mechanizm IntelliSense wyświetli błąd w kodzie funkcji `speak`, gdyż próbujemy w niej zmienić wartość pola `msg`, którego wartość została już raz ustawiona — w konstruktorze.

Modyfikatory `private` i `readonly` nie są jedynymi dostępnymi w TypeScripcie. Istnieje także kilka innych modyfikatorów dostępu, jednak lepiej będzie przedstawić je nieco później, w kontekście mechanizmów dziedziczenia.

Akcesory get i set

Kolejną możliwością klas, dostępną zarówno w języku TypeScript, jak i JavaScript, są akcesory *get* i *set*:

- **akcesor *get*:** to właściwość pozwalająca na zmodyfikowanie lub sprawdzenie poprawności powiązanego z nią pola przed zwróceniem jego wartości.
- **akcesor *set*:** to właściwość pozwalająca na zmodyfikowanie lub wyliczenie wartości przed jej zapisaniem w powiązanym polu.

W niektórych innych językach programowania takie właściwości są nazywane właściwościami złożonymi (ang. *compound properties*). Przyjrzyjmy się im na przykładzie. Utwórz nowy plik o nazwie *getSet.ts* i zapisz w nim następujący kod:

```
class Speaker {
    private message: string;
    constructor(private name: string) {}

    get Message() {
        if(!this.message.includes(this.name)){
            throw Error("W komunikacie brakuje imienia mówcy.");
        }
        return this.message;
    }

    set Message(val: string) {
        let tmpMessage = val;
        if(!val.includes(this.name)){
            tmpMessage = this.name + " " + val;
        }
        this.message = tmpMessage;
    }
```


Kup ksi k Pole ksi k

```
}

const speaker = new Speaker("Janek");
speaker.Message = "cześć";
console.log(speaker.Message);
```

W tym przykładzie dzieje się dosyć dużo, więc zanim go skompilujesz i uruchomisz, warto dokładniej go przeanalizować. W pierwszej kolejności zwróć uwagę na to, że pole `message` nie jest dostępne w konstruktorze, lecz zostało zadeklarowane jako pole prywatne (`private`), więc w kodzie, który nie należy do klasy, nie będzie można odwoływać się do niego bezpośrednio. Jedyną wartością inicjalizującą pobieraną przez konstruktor jest pole `name`. Poniżej konstruktora umieszczona jest właściwość `Message`; przed jej nazwą widoczne jest słowo kluczowe `get`, które sygnalizuje, że jest to akcesor *get*. W kodzie tej właściwości sprawdzamy, czy łańcuch zapisany we właściwości `message` zawiera nazwę mówcy, a jeśli jej nie zawiera, to zgłaszamy wyjątek, sygnalizując w ten sposób niepożądaną sytuację. Akcesor *set*, który także nosi nazwę `Message`, jest poprzedzony słowem kluczowym `set` i pobiera wartość — łańcuch, do którego, w razie konieczności, dodajemy nazwę mówcy i zapisujemy w polu `message`. Zwróć uwagę na to, że choć oba akcesory wyglądają jak funkcje, to jednak nimi nie są. Kiedy używamy ich w dalszej części kodu, robimy to w sposób charakterystyczny dla odwołań do pól, a nie wywołań metod, czyli bez nawiasów. W dolnej części kodu tworzymy obiekt `Speaker`, przekazując do konstruktora imię `"Janek"`. Następnie przypisujemy właściwości `Message` łańcuch `"cześć"`. W ostatniej instrukcji wyświetlamy komunikat na konsoli.

A teraz chcielibyśmy skompilować ten kod i go wykonać. W tym celu musimy postąpić nieco inaczej niż robiliśmy do tej pory. Kompilator języka TypeScript udostępnia opcje, których można używać w celu modyfikowania jego działania. W naszym przykładzie, zastosowane w nim akcesory oraz funkcja `includes` są możliwościami dostępnymi odpowiednio w wersjach ES5 oraz ES6 języka JavaScript. Jeśli jeszcze nie spotkałeś się z funkcją `includes`, to zapamiętaj, że sprawdza ona, czy jeden łańcuch znaków zawiera w sobie inny łańcuch. Poinformujmy zatem kompilator TypeScriptu, że wynikowy kod JavaScript musi być zgodny z wersjami języka nowszymi od ES3, a właśnie ta wersja kodu JavaScript jest generowana domyślnie.

Poniżej przedstawiłem nową postać polecenia uruchamiającego kompilator TypeScriptu, której będziesz musiał użyć (szczegółowe informacje dotyczące stosowania kompilatora, w tym także korzystanie z plików konfiguracyjnych, znajdziesz w dalszej części książki):

```
tsc --target "ES6" getSet
```

Po jego wykonaniu będziesz już mógł w standardowy sposób uruchomić skrypt:

```
node getSet
```

Wyniki wykonania tego skryptu przedstawiłem na rysunku 2.10.

```
H:\NaukaTypeScriptu\rozdzial02>node getSet
Janek cześć
```

Rysunek 2.10. Wyniki wykonania skryptu getSet

Kup ksi k

Pole ksi k

Aby dokładniej poznać działanie akcesorów, spróbujmy zmienić wiersz `speaker.Message = "cześć"` na `speaker.message = "cześć"`. Jeśli teraz spróbujemy skompilować kod przykładu, to kompilator zgłosi błąd przedstawiony na rysunku 2.11.

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc --target "ES6" getSet
getSet.ts:22:9 - error TS2341: Property 'message' is private and only accessible within class 'Speaker'.

22 speaker.message = "cześć";
           ~~~~~~~

Found 1 error.
```

Rysunek 2.11. Błąd dostępu do pola message

Czy potrafisz wyjaśnić, dlaczego nie udało się skompilować tego kodu? Owszem, wynika to z faktu, że pole `message` jest prywatne i nie można odwoływać się do niego spoza kodu klasy `Speaker`.

Być może zastanawiasz się, dlaczego wspominam tu o akcesorach *get* i *set*, skoro są one dostępne także w języku JavaScript. Jeśli przyjrzysz się przedstawionemu przykładowi, zapewne zauważysz, że pole `message` zostało zadeklarowane jako prywatne (`private`), natomiast właściwości akcesorów *get* i *set* są publiczne (zwróć uwagę, że brak jawnie podanego modyfikatora dostępu oznacza, że składowa jest publiczna — `public`). A zatem, w celu zapewnienia odpowiedniej hermetyzacji, dobra praktyka nakazuje ukrycie pola i udostępnianie go wyłącznie w razie konieczności, właśnie przy wykorzystaniu akcesorów *get* oraz *set*, bądź jakiejś funkcji, która umożliwi modyfikowanie jego wartości. Pamiętaj także, że podczas określania poziomu dostępu do składowych klas, należy zaczynać od najbardziej restrykcyjnych ustawień, a dopiero potem, w razie konieczności, minimalizować ograniczenia. Oprócz tego, dzięki odwoływaniu się do pól przy użyciu akcesorów zyskujemy możliwość wykonywania wszelkiego rodzaju testów i modyfikacji, tak jak robiliśmy to w przedstawionym przykładzie, i dysponujemy pełną kontrolą nad tym, jakie dane będą zapisywane w naszej klasie i przez nią zwracane.

Właściwości i metody statyczne

W tym podpunkcie zajmiemy się właściwościami i metodami **statycznymi**. Jeśli jakąś składową klasy zadeklarujemy jako statyczną (używając w tym celu modyfikatora `static`), zaznaczamy przez to, że jest to składowa klasy, a nie jej poszczególnych instancji. Oznacza to, że do takiej składowej można odwoływać się bez konieczności tworzenia instancji danej klasy, poprzedzając jej nazwę nazwą samej klasy.

Przyjrzyjmy się składowym statycznym na przykładzie. Utwórz nowy plik o nazwie *staticMember.ts* i zapisz w nim następujący kod:

```
class ClassA {
    static typeName: string;
    constructor(){}
    static getFullName() {
```


Kup ksi k   Pole ksi k

```
        return "ClassA " + ClassA.typeName;
    }
}

const a = new ClassA();
console.log(a.typeName);
```

Próba skompilowania takiego kodu zakończy się niepowodzeniem i wyświetleniem komunikatu z informacją, że właściwość `typeName` nie istnieje i pytaniem, czy chodziło nam o odwołanie do składowej klasowej `ClassA.typeName`. Przypominam, że w odwołaniach do składowych klasowych należy używać nazwy klas. Poniżej przedstawiłem poprawioną wersję kodu:

```
class ClassA {
    static typeName: string;
    constructor(){}
    static getFullName() {
        return "ClassA " + ClassA.typeName;
    }
}

const a = new ClassA();
console.log(ClassA.typeName);
```

Jak widać na powyższym przykładzie, do składowej statycznej należy odwoływać się używając nazwy klasy. Powstaje zatem pytanie, dlaczego moglibyśmy chcieć korzystać ze składowych statycznych, a nie instancyjnych? Otóż w pewnych okolicznościach może być przydatne współdzielenie pewnych danych pomiędzy wszystkimi instancjami klasy. Przykład takiego rozwiązania przedstawiłem na poniższym listingu:

```
class Runner {
    static lastRunTypeName: string;

    constructor(private typeName: string) {}

    run() {
        Runner.lastRunTypeName = this.typeName;
    }
}

const a = new Runner("a");
const b = new Runner("b");
b.run();
a.run();
console.log(Runner.lastRunTypeName);
```

W przypadku tego przykładu zależy nam na określeniu ostatniej instancji klasy, która w dowolnym momencie jako ostatnia wywołała funkcję `run`. Dzięki zastosowaniu składowej statycznej, przechowywanie takiej informacji może być trywialnie proste. Kolejnym zagadnieniem związanym ze składowymi statycznymi, o którym trzeba pamiętać, jest to, że wewnątrz kodu klasy mogą się do nich odwoływać zarówno składowe statyczne, jak i instancyjne. Z drugiej strony, składowe statyczne nie mogą odwoływać się do składowych instancyjnych.

Kup ksi k Pole ksi k

W tym punkcie rozdziału poznałeś klasy oraz ich możliwości. Ta wiedza pozwoli Ci projektować kod korzystający z zasady hermetyzacji, co z kolei przyczyni się do poprawy jego jakości. W następnym punkcie rozdziału zajmiemy się interfejsami oraz programowaniem w oparciu o kontrakty.

Interfejsy

Kolejną ważną regułą projektowania oprogramowania obiektowego jest abstrakcja. Jej celem jest redukcja złożoności oraz powiązań pomiędzy fragmentami kodu poprzez ukrywanie ich wewnętrznej implementacji (o abstrakcji wspominałem już w rozdziale 1., pt. „Jak zrozumieć TypeScript i poprawić swoją znajomość języka JavaScript". Jednym ze sposobów wprowadzania abstrakcji jest korzystanie z **interfejsów** (ang. *interfaces*) w celu ujawniania jedynie sygnatury typu, a nie jego wewnętrznego sposobu działania. Interfejsy są także czasami nazywane kontraktami, gdyż określanie konkretnych typów parametrów i wyników zwracanych przez funkcje wymusza pewne oczekiwania — i to zarówno na użytkownikach, jak i na twórcy interfejsu. A zatem, innym sposobem pojmowania, czym są interfejsy, jest wyobrażenie ich sobie jako ścisłych reguł określających postać danych przekazywanych do instancji danego typu oraz informacji, które można z takiej instancji pobierać.

A zatem, interfejsy są jedynie zestawami reguł. Aby przekształcić je w działający kod, musimy te reguły zaimplementować w formie kodu, który coś robi. Abyśmy mogli rozpocząć poznawanie interfejsów, przedstawię przykład prostego interfejsu wraz z jego implementacją. Utwórz nowy plik o nazwie *interfaces.ts* i zapisz w nim następujący kod:

```
interface Employee {
    name: string;
    id: number;
    isManager: boolean;
    getUniqueId: () => string;
}
```

Ten interfejs definiuje typ `Employee`; z następnego przykładu dowiesz się, jak utworzyć instancję tego typu. Jak widać, interfejs zawiera jedynie sygnaturę funkcji `getUniqueId`, a nie jej implementację — tę utworzymy podczas definiowania instancji tego typu.

Teraz dodaj do pliku *interfaces.ts* implementację interfejsu `Employee`. Poniższy kod tworzy dwie instancje tego interfejsu:

```
const linda: Employee = {
    name: "Linda",
    id: 2,
    isManager: false,
    getUniqueId: (): string => {
        let uniqueId = linda.id + "-" + linda.name;
        if(!linda.isManager) {
            return "prc-" + uniqueId;
        }
        return uniqueId;
    }
}
```

```
console.log(linda.getUniqueId());
const pam: Employee = {
    name: "Patrycja",
    id: 1,
    isManager: true,
    getUniqueId: (): string => {
        let uniqueId = pam.id + "-" + pam.name;
        if(pam.isManager) {
            return "kier-" + uniqueId;
        }
        return uniqueId;
    }
}
console.log(pam.getUniqueId());
```

A zatem, instancję naszego interfejsu tworzymy przygotowując literał obiektowy o nazwie `linda`, określając wartości jego dwóch pól — `name` oraz `id` — i implementując funkcję `getUniqueId`. W kolejnym kroku wyświetlamy na konsoli wynik zwrócony przez wywołanie `linda.getUniqueId`. Następnie tworzymy kolejny obiekt, `pam`, implementujący ten sam interfejs. Zwróć jednak uwagę, że różni się on od obiektu `linda` nie tylko wartościami pól, lecz także implementacją funkcji `getUniqueId`. I właśnie ten przykład pokazuje podstawowe zastosowanie interfejsów: mają one zagwarantować, że tworzone obiekty będą mieć taką samą strukturę, a jednocześnie zapewnić możliwość stosowania w nich różnych implementacji. W ten sposób możemy narzucać ściśle reguły na strukturę typu, a jednocześnie zapewnić mu pewną elastyczność pod względem sposobu działania jego funkcji. Na rysunku 2.12 przedstawiłem wyniki wykonania ostatniego przykładu.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc interfaces

H:\NaukaTypeScriptu\rozdzial02>node interfaces
prc-2-Linda
kier-1-Patrycja
```

Rysunek 2.12. Efekty zastosowania interfejsu Employee

Kolejną możliwością zastosowania interfejsów jest korzystanie z interfejsów programowania aplikacji (API) przygotowanych przez innych. Czasami podczas korzystania z takich narzędzi nie dysponujemy dokładną dokumentacją dotyczącą używanych typów, a jedynym, co mamy do dyspozycji, jest kod JSON, pozbawiony jakichkolwiek informacji o typach, bądź też ogromny obiekt typu, zawierający bardzo wiele pól, których nigdy nie będziemy musieli używać. W takich sytuacjach można ulec pokusie, by zastosować typ `any` i niczym więcej się nie przejmować. Niemniej jednak, należy starać się podawać deklaracje typów zawsze, kiedy tylko to jest możliwe.

W takich okolicznościach możemy utworzyć interfejs, który będzie uwzględniał wyłącznie te pola, które znamy i na których nam zależy. Następnie możemy zadeklarować, że używane dane są właśnie tego typu. W trakcie pisania kodu TypeScript nie będzie w stanie sprawdzać typu danych, gdyż sieciowe wywołania API będą je zwracały dopiero podczas wykonywania

kodu. Jednak nie ma to wielkiego znaczenia, gdyż TypeScript, który — jak wiemy — zwraca uwagę jedynie na strukturę typów i tak będzie ignorował pola, które nie zostały wymienione w deklaracji, i o ile tylko dane będą zawierać pola wymieniona w interfejsie, środowisko uruchomieniowe nie będzie zgłaszać problemów, a my będziemy mogli cieszyć się większym bezpieczeństwem podczas pisania kodu. Korzystając z takiego rozwiązania trzeba jednak zachować dużą ostrożność i zwracać baczną uwagę na właściwe obsługiwanie pól mogących zawierać wartości `null` lub `undefined` — na przykład należy w nich używać unii lub testować typy w kodzie.

W tym podrozdziale poznałeś interfejsy i dowiedziałeś się, czym różnią się one od klas. Interfejsów będziemy używali, aby hermetyzować szczegóły klas i wprowadzać luźne powiązania pomiędzy różnymi fragmentami kodu, poprawiając tym samym jego jakość. W następnym podrozdziale pokażę, w jaki sposób klasy i interfejsy pozwalają na stosowanie dziedziczenia, a co za tym idzie, na wielokrotne używanie kodu.

Dziedziczenie

W tym podrozdziale zajmiemy się **dziedziczeniem** (ang. *inheritance*). W programowaniu obiektowym dziedziczenie jest sposobem na wielokrotne wykorzystywanie kodu. Dziedziczenie pozwala na skrócenie kodu aplikacji i poprawienie jego przejrzystości. Oprócz tego, ogólnie rzecz ujmując, krótszy kod będzie zawierał mniej błędów. A zatem, wszystkie te czynniki sprawią, że kiedy zaczniemy już korzystać z dziedziczenia, jakość naszego kodu się poprawi.

Jak już zaznaczyłem, dziedziczenie wiąże się przede wszystkim z wielokrotnym stosowaniem kodu. Oprócz tego, pod względem koncepcyjnym, dziedziczenie w programowaniu obiektowym zaprojektowano w taki sposób, by przypominało to występujące w realnym świecie — dzięki temu logiczny tok relacji dziedziczenia można łatwo i intuicyjnie zrozumieć. Przeanalizujmy zatem przykład dziedziczenia. Zacznij od utworzenia nowego pliku o nazwie *classInheritance.ts* i zapisania w nim następującego kodu:

```
class Vehicle {
    constructor(private wheelCount: number) {}

    showNumberOfWheels() {
        console.log(`Liczba kół w pojeździe: ${this.wheelCount} `);
    }
}
class Motorcycle extends Vehicle {
    constructor() {
        super(2);
    }
}
class Automobile extends Vehicle {
    constructor() {
        super(4);
    }
}
const motorCycle = new Motorcycle();
```


Kup ksi k Pole ksi k

```
motorCycle.showNumberOfWheels();
const autoMobile = new Automobile();
autoMobile.showNumberOfWheels();
```

Krótka uwaga dla czytelników, którzy jeszcze nigdy wcześniej nie spotkali się z użyciem znaków odwrotnego apostrofu `` oraz sekwencji ${}. Pozwalają one na korzystanie z mechanizmu wstawiania łańcuchów (ang. *string interpolation*), czyli umieszczania jednego łańcucha wewnątrz innego przy wykorzystaniu odpowiednich odwołań.

Jak widać, na początku kodu zdefiniowaliśmy klasę bazową, nazywaną także nadrzędną; w naszym przykładzie jest to klasa `Vehicle`. Ta klasa pełni rolę głównego pojemnika kodu źródłowego, który będzie używany przez wszelkie klasy dziedziczące po tej klasie bazowej, nazywane także klasami pochodnymi. W klasach pochodnych wskazujemy klasę, po której dziedziczą, dzięki podaniu nazwy klasy bazowej po słowie kluczowym `extends`. Ważną rzeczą, na jaką trzeba tu zwrócić uwagę, są konstruktory wszystkich klas pochodnych. Jak widać, w pierwszym wierszu kodu w ich konstruktorach zostało umieszczone wywołanie `super`. Metoda `super()` zwraca instancję klasy nadrzędnej, po której dana klasa dziedziczy. A zatem, w naszym przykładzie będzie to klasa `Vehicle`. Jak widać na przykładzie, każda z klas pochodnych przekazuje do wywołania konstruktora klasy bazowej różną liczbę, która zostanie zapisana jako wartość właściwości `wheelCount`. Pod koniec kodu tworzymy instancje obu klas pochodnych, `Motorcycle` oraz `Automobile`, i dla każdej z nich wywołujemy funkcję `showNumberOfWheels`. Na rysunku 2.13 pokazałem wyniki, które zostaną wyświetlone, kiedy skompilujemy i wykonamy ten kod.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc classInheritance

H:\NaukaTypeScriptu\rozdzial02>node classInheritance
Liczba kół w pojeździe: 2
Liczba kół w pojeździe: 4
```

Rysunek 2.13. Wyniki wykonania skryptu classInheritance.ts

A zatem, każda klasa pochodna określa własną liczbę kół, zapisywaną we właściwości `wheelCount` klasy nadrzędnej, a jednocześnie żadna z nich nie ma do tej wartości bezpośredniego dostępu. A teraz załóżmy, że istnieje jakaś ważka przyczyna, dla której klasa pochodna chciałaby pobrać wartość właściwości `wheelCount` klasy nadrzędnej. Na przykład załóżmy, że konieczna będzie aktualizacja liczby kół w przypadku przebicia dętki. Co możemy zrobić by zapewnić sobie tę możliwość? Moglibyśmy na przykład utworzyć w każdej z klas pochodnych unikalną funkcję, która będzie próbować zaktualizować wartość właściwości `wheelCount`. Zróbmy tak i sprawdźmy, co się stanie. Do klasy `Motorcycle` dodaj nową funkcję o nazwie `updateWheelCount`:

```
class Vehicle {
    constructor(private wheelCount: number) {}

    showNumberOfWheels() {
        console.log(`Liczba kół w pojeździe: ${this.wheelCount}`);
```


Kup ksi k   Pole ksi k

```
    }
}
class Motorcycle extends Vehicle {
    constructor() {
        super(2);
    }
    updateWheelCount(newWheelCount: number){
        this.wheelCount = newWheelCount;
    }
}
class Automobile extends Vehicle {
    constructor() {
        super(4);
    }
}
const motorCycle = new Motorcycle();
motorCycle.showNumberOfWheels();
const autoMobile = new Automobile();
autoMobile.showNumberOfWheels();
```

Okazuje się, że dodanie do klasy `Motorcycle` metody `updateWheelCount` przedstawionej na powyższym przykładzie powoduje wystąpienie błędu. Czy potrafisz wskazać jego przyczynę? Otóż błąd występuje dlatego, że próbujemy odwołać się do prywatnej składowej klasy nadrzędnej. A zatem, choć klasy pochodne dziedziczą składowe po swojej klasie nadrzędnej, to nie dysponują dostępem do ich składowych prywatnych. Ten sposób działania jest całkowicie prawidłowy i ma na celu wzmocnienie hermetyzacji. A jak możemy obejść ten problem? Spróbuj wprowadzić w kodzie niewielką zmianę, przedstawioną na kolejnym listingu:

```
class Vehicle {
    constructor(protected wheelCount: number) {}

    showNumberOfWheels() {
        console.log(`Liczba kół w pojeździe: ${this.wheelCount}`);
    }
}
class Motorcycle extends Vehicle {
    constructor() {
        super(2);
    }
    updateWheelCount(newWheelCount: number){
        this.wheelCount = newWheelCount;
    }
}
class Automobile extends Vehicle {
    constructor() {
        super(4);
    }
}
const motorCycle = new Motorcycle();
motorCycle.showNumberOfWheels();
const autoMobile = new Automobile();
autoMobile.showNumberOfWheels();
```


Kup ksi k Pole ksi k

Czy w ogóle zauważyłeś, co zmieniliśmy? Masz rację: parametr `wheelCount` w konstruktorze klasy `Vehicle` został teraz zadeklarowany jako chroniony (przy użyciu modyfikatora dostępu `protected`). Użycie tego modyfikatora sprawia, że dostęp do składowej będzie mieć kod tej samej klasy oraz wszystkich jej klas pochodnych.

Zanim przejdziemy do następnych zagadnień, chciałbym przedstawić pojęcie przestrzeni nazw (ang. *namespace*). Przestrzenie nazw pozwolą nam tworzyć swoiste pojemniki na kod, dysponujące wyznaczonym zasięgiem, a dzięki temu oddzielać jedne fragmenty kodu od innych. Wyznaczanie zasięgów przy użyciu przestrzeni nazw pozwala ukrywać wszystko, co znajduje się wewnątrz danej przestrzeni, od kodu znajdującego się poza nią. Pod tym względem przestrzenie nazw przypominają nieco klasy, jednak w przestrzeni nazw może się znaleźć dowolna liczba klas, funkcji, zmiennych oraz wszelkich innych typów. Poniżej przedstawiłem bardzo prosty przykład zastosowania przestrzeni nazw. Utwórz nowy plik o nazwie *namesapces.ts* i zapisz w nim następujący kod:

```
namespace A {
    class FirstClass {}
}

namespace B {
    class SecondClass {}
    const test = new FirstClass();
}
```

Kiedy będziesz wpisywać ten kod, jeszcze przed jego skompilowaniem, zauważysz, że mechanizm IntelliSense VSCode zasygnalizuje problem ze znalezieniem klasy `FirstClass`. Klasa ta nie jest widoczna w przestrzeni nazw `B`, gdyż została zdefiniowana w przestrzeni nazw `A`. Właśnie takie jest przeznaczenie przestrzeni nazw — służą one do ukrywania informacji pozostających w zasięgu tylko danej przestrzeni nazw, tak, by nie były widoczne w innych przestrzeniach.

W tym podrozdziale poznałeś zagadnienia związane z dziedziczeniem klas. Dziedziczenie jest niezwykle ważnym narzędziem, wykorzystywanym do wielokrotnego stosowania kodu. W następnym punkcie zajmiemy się klasami abstrakcyjnymi, stanowiącymi jeszcze bardziej elastyczne narzędzie dziedziczenia.

Klasy abstrakcyjne

Wcześniej wspominałem, że interfejsy przydają się do definiowania kontraktów, jednak nie dysponują implementacjami, czyli działającym kodem. Z kolei klasy dysponują działającym kodem, lecz czasami będziemy potrzebować jedynie sygnatur, a nie całych implementacji. Może się jednak zdarzyć, że w niektórych sytuacjach będziemy chcieli połączyć obie te cechy w jednym typie. W takich sytuacjach, zamiast klas bądź interfejsów przydadzą się nam **klasy abstrakcyjne** (ang. *abstract classes*). Utwórz nowy plik o nazwie *abstractClass.ts* i skopiuj do niego całą zawartość pliku *classInheritance.ts*. Kiedy to zrobisz, VSCode wyświetli kilka błędów, gdyż pojawiły się dwa pliki zawierające klasy i zmienne o tych samych nazwach.

Kup ksi k

Pole ksi k

Dlatego zmienimy plik *abstractClass.ts*, a konkretnie: dodamy do niego przestrzeń nazw i zmienimy klasę `Vehicle` na klasę abstrakcyjną. Dodaj zatem do pliku przestrzeń nazw i zmień klasę `Vehicle` zgodnie z przykładem zamieszczonym poniżej:

```
namespace AbstractNamespace {
    abstract class Vehicle {
        constructor(protected wheelCount: number) {}

        abstract updateWheelCount(newWheelCount: number): void;

        showNumberOfWheels() {
            console.log(`Liczba kół w pojeździe: ${this.wheelCount}`);
        }
    }
```

Zacznijmy od przestrzeni nazw… Jak widać, cały skopiowany kod umieściliśmy w nawiasach klamrowych wyznaczających zasięg przestrzeni nazw zdefiniowanej przy użyciu zapisu `namespace AbstractNamespace` (zwróć uwagę na to, że nazwa przestrzeni nazw może być dowolna i nie musi zawierać w sobie słowa „namespace"). Jak już wspominałem, przestrzeń nazw to jedynie pojemnik pozwalający nam kontrolować zakres widoczności. Dzięki jej zastosowaniu składowe zdefiniowane w pliku *abstractClass.ts* nie będą należeć do zasięgu globalnego, a tym samym nie będą wpływać na kod umieszczony w innych plikach. Kiedy przyjrzysz się nowemu kodowi klasy `Vehicle`, zauważysz zapewne, że w jej definicji zastosowaliśmy nowe słowo kluczowe — `abstract`. Oznacza ono, że tworzona klasa będzie klasą abstrakcyjną. Oprócz tego dodaliśmy do niej także nową funkcję o nazwie `updateWheelCount`. Na samym początku deklaracji tej funkcji umieściliśmy słowo kluczowe `abstract`; oznacza ono, że funkcja ta nie będzie mieć żadnej implementacji w klasie `Vehicle` i będzie musiała zostać zaimplementowana w klasach pochodnych.

Poniżej naszej nowej, abstrakcyjnej klasy `Vehicle` znajdują się jej dwie klasy pochodne — `Motorcycle` oraz `Automobile`. Zmodyfikuj je tak, by były zgodne z kodem przedstawionym poniżej:

```
    class Motorcycle extends Vehicle {
        constructor() {
            super(2);
        }
        updateWheelCount(newWheelCount: number){
            this.wheelCount = newWheelCount;
            console.log(`Motocykl ma ${this.wheelCount} koła.`);
        }
    }
    class Automobile extends Vehicle {
        constructor() {
            super(4);
        }
        updateWheelCount(newWheelCount: number){
            this.wheelCount = newWheelCount;
            console.log(`Motocykl ma ${this.wheelCount} koła.`);
        }
        showNumberOfWheels() {
            console.log(`Liczba kół w pojeździe: ${this.wheelCount} `);
        }
    }
```


Kup ksi k Pole ksi k

W ostatnim fragmencie kodu tworzymy instancje obu klas pochodnych i wywołujemy ich metody `updateWheelCount`:

```
        const motorCycle = new Motorcycle();
        motorCycle.showNumberOfWheels(1);
        const autoMobile = new Automobile();
        autoMobile.showNumberOfWheels(3);
    }
```

Jak widać, implementację abstrakcyjnej składowej `updateWheelCount` umieściliśmy w klasach pochodnych. Tę możliwość zapewniają nam klasy abstrakcyjne, które mogą działać jak zwyczajne klasy, czyli określać implementacje składowych, oraz jak interfejsy, czyli podawać jedynie reguły przyszłych implementacji, które zostaną określone w klasach pochodnych. Trzeba zauważyć, że ze względu na to, że klasy abstrakcyjne zawierają składowe abstrakcyjne, nie można tworzyć instancji tych klas.

Co więcej, jeśli przyjrzysz się teraz klasie `Automobile`, zauważysz zapewne, że dysponuje ona własną implementacją funkcji `showNumberOfWheels`, choć nie jest to funkcja abstrakcyjna. Zdefiniowanie tej funkcji w klasie `Automobile` stanowi przykład tak zwanego **przesłaniania** (ang. *overriding*), czyli możliwości zdefiniowania w klasie pochodnej unikalnej implementacji składowej zdefiniowanej w klasie nadrzędnej.

W tym punkcie rozdziału opisałem różne rodzaje dziedziczenia, jakie zapewniają klasy. Opanowanie zasad dziedziczenia pozwoli na wielokrotne wykorzystanie kodu oraz zmniejszenie nie tylko jego wielkości, lecz także liczby występujących w nim błędów. W następnym punkcie przedstawię możliwości dziedziczenia, jakie zapewniają interfejsy, wyjaśnię także, czym różnią się one od tych, które dają klasy.

Interfejsy

Jak już wyjaśniłem wcześniej, **interfejsy** są sposobem określania reguł odnoszących się do typu danych. Pozwalają one oddzielać implementację od definicji, umożliwiając tym samym wprowadzanie abstrakcji, która z kolei jest jedną z podstawowych zasad programowania obiektowego, mającą wpływ na jakość kodu. Przekonajmy się zatem, w jaki sposób można używać interfejsów do jawnego dziedziczenia i zapewnienia odpowiedniej struktury kodu.

Interfejsy w języku TypeScript pozwalają na określanie sygnatur typów składowych interfejsów, jednak nie pozwalają definiować żadnych implementacji. Wcześniej przedstawiłem już sposoby używania interfejsów do tworzenia niezależnych obiektów, jednak tym razem skoncentrujemy się na zastosowaniu interfejsów jako mechanizmu dziedziczenia i wielokrotnego stosowania kodu. Utwórz nowy plik o nazwie *interfaceInheritance.ts* i zapisz w nim następujący kod:

```
namespace InterfaceNamespace {
    interface Thing {
        name: string;
        getFullName: () => string;
    }
```


Kup ksi   k   Pole   ksi   k

```
    interface Vehicle extends Thing {
        wheelCount: number;
        updateWheelCount: (newWheelCount: number) => void;
        showNumberOfWheels: () => void;
    }
```

Jak widać, po deklaracji przestrzeni nazw definiujemy interfejs o nazwie `Thing`, a po nim kolejny interfejs o nazwie `Vehicle`, który, dzięki zastosowaniu słowa kluczowego `extends`, dziedziczy po interfejsie `Thing`. Wprowadziłem to rozwiązanie do przykładu celowo, aby pokazać, że interfejsy mogą dziedziczyć po innych interfejsach. Interfejs `Thing` ma dwie składowe, `name` oraz `getFullName`. Jednak, jak widać na przykładzie, choć interfejs `Vehicle` dziedziczy po `Thing`, w jego kodzie nie ma żadnej wzmianki o tych dwóch składowych. Wynika to z faktu, że `Vehicle` jest interfejsem, a — jak wiadomo — interfejsy nie mogą zawierać żadnych implementacji. Jeśli jednak przyjrzymy się dokładniej zamieszczonemu poniżej kodowi klasy `Motorcycle`, to zauważymy, że implementuje on interfejs `Vehicle` i zawiera wszystkie niezbędne implementacje składowych tego interfejsu:

```
    class Motorcycle implements Vehicle {
        name: string;
        wheelCount: number;
        constructor(name: string) {
            // W przypadku implementacji interfejsów nie trzeba
            // wywoływać konstruktora klasy bazowej
            this.name = name;
        }
        updateWheelCount(newWheelCount: number){
            this.wheelCount = newWheelCount;
            console.log(`Pojazd ma ${this.wheelCount} kół.`);
        }
        showNumberOfWheels() {
            console.log(`Liczba kół w pojeździe ${this.wheelCount}`);
        }
        getFullName() {
            return "MC-" + this.name;
        }
    }
    const moto = new Motorcycle("moto-dla-początkujących");
    console.log(moto.getFullName());
}
```

Kiedy skompilujemy ten kod i wykonamy go, uzyskamy takie wyniki, jak te przedstawione na rysunku 2.14.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc interfaceInheritance

H:\NaukaTypeScriptu\rozdzial02>node interfaceInheritance
MC-moto-dla-początkujących
```

Rysunek 2.14. Wyniki wykonania skryptu interfaceInheritance.ts

Kup ksi k

Pole ksi k

Interfejsy nie zapewniają możliwości wielokrotnego wykorzystywania kodu w sposób bezpośredni, gdyż nie pozwalają na określanie implementacji. Niemniej jednak, z punktu widzenia wielokrotnego stosowania kodu, korzystanie z interfejsów i tak jest korzystne, gdyż w jawny sposób określają one oczekiwania odnośnie do tego, jakie dane kod ma pobierać i zwracać. Ukrywanie implementacji przy użyciu interfejsu jest także korzystne z punktu widzenia wprowadzania hermetyzacji i abstrakcji, a jak wiemy, są to kolejne dwie ważne zasady programowania obiektowego.

> W przypadku programowania w języku TypeScript, warto korzystać ze wszelkich dostępnych w nim możliwości modelu dziedziczenia. Interfejsów można używać, by zapewniać abstrakcję szczegółów implementacyjnych. Z kolei modyfikatory dostępu `private` i `protected` pozwalają na hermetyzację danych. Pamiętaj, że kiedy nadejdzie czas na skompilowanie kodu i przekształcenie go do postaci kodu JavaScript, kompilator TypeScriptu wykona wszystkie niezbędne czynności konieczne do wygenerowania kodu korzystającego z modelu dziedziczenia przez prototyp. Niemniej jednak, podczas pisania kodu, powinieneś używać wszystkich usprawnień, jakie zapewnia język TypeScript, gdyż tylko w ten sposób będziesz mógł skorzystać ze wzbogaconych możliwości programowania.

Typy generyczne

Typy generyczne (ang. *generics*), nazywane także *typami ogólnymi*, zapewniają możliwość umieszczania w definicji typu innego, skojarzonego typu wybieranego przez jego użytkownika, a nie narzuconego przez twórcę. Dzięki temu typ ma swoją strukturę oraz reguły, a jednocześnie zapewnia pewną elastyczność. Typy generyczne nabiorą znacznie większego znaczenia później, kiedy zaczniemy używać Reacta, dlatego też warto je poznać.

Typów generycznych można używać w funkcjach, klasach oraz interfejsach. Przeanalizujmy przykład zastosowania typu generycznego w funkcji. Utwórz plik o nazwie *functionGeneric.ts* i zapisz w nim następujący kod:

```
function getLength<T>(arg: T): number {
    if(arg.hasOwnProperty("length")) {
        return arg["length"];
    }
    return 0;
}

console.log(getLength<number>(22));
console.log(getLength("Witaj, świecie!"));
```

Na samym początku kodu znajduje się definicja funkcji `getLength<T>`. Ta funkcja używa typu generycznego, który informuje kompilator, że w każdym miejscu, gdzie występuje typ `T`, należy oczekiwać dowolnego możliwego typu. Wewnętrzy sposób działania tej funkcji polega na sprawdzeniu, czy parametr `arg` dysponuje polem o nazwie `length`, a jeśli tak, to funkcja zwraca jego wartość. Jeśli pole `length` nie jest dostępne, to funkcja zwraca wartość `0`. Na końcu przykładu dwukrotnie wywołujemy funkcję `getLength`: za pierwszym razem

Kup ksi k | Pole ksi k

przekazujemy do niej liczbę, a za drugim łańcuch. Jak widać, w przypadku pierwszego wywołania, w którym przekazywana jest liczba, jawnie podajemy oznaczenie typu, natomiast w drugim wywołaniu, w którym przekazywany jest argument typu `string`, takiego oznaczenia typu nie ma. W pierwszym przypadku podałem typ tylko po to, by pokazać, że można go określić jawnie, jednak zazwyczaj kompilator może określić go samodzielnie, na podstawie kodu.

Jednak przykład ten powoduje pewien problem; otóż sprawdzenie, czy dana przekazana do funkcji dysponuje polem `length` wymaga użycia dodatkowego kodu. Z tego względu kod jest dłuższy niż to konieczne, a jego wykonywanie zajmuje więcej czasu. Spróbujmy zatem zaktualizować ten kod w taki sposób, by ta dodatkowa funkcja nie była wykonywana, jeśli argument nie dysponuje właściwością `length`. W pierwszej kolejności umieść całą dotychczasową zawartość pliku w komentarzu, a następnie poniżej zapisz następujący fragment kodu:

```
interface HasLength {
    length: number;
}

function getLength<T extends HasLength>(arg: T): number {
    return arg.length;
}

console.log(getLength<number>(22));
console.log(getLength("Witaj, świecie!"));
```

Ta nowa wersja kodu jest dość podobna do poprzedniej, jednak zastosowaliśmy w niej interfejs `HasLength`, który ogranicza typy, jakie można przekazywać w wywołaniu funkcji `getLength`. Zastosowany zapis, `T extends HasLength`, informuje kompilator, że niezależnie od tego, czym jest typ `T`, musi on dziedziczyć po typie `HasLength`, bądź też być tym typem, a to z kolei w praktyce oznacza, że przekazany typ musi dysponować właściwością `length`. Dlatego tym razem pierwsze z dwóch wywołań umieszczonych na końcu kodu, to, w którym przekazujemy argument typu `number`, spowoduje zgłoszenie błędu, gdyż liczby nie mają właściwości `length`. Drugie wywołanie, w którym używany typu `string`, działa bez problemów.

A teraz przyjrzyjmy się kolejnemu przykładowi, w którym zastosujemy interfejsy i klasy. Utwórz nowy plik o nazwie *classGeneric.ts* i zapisz w nim następujący kod:

```
namespace GenericNamespace {
    interface Wheels {
        count: number;
        diameter: number;
    }

    interface Vehicle<T> {
        getName(): string;
        getWheelCount: () => T;
    }
```

Jak widać, na samym początku definiujemy dwa interfejsy. Pierwszy z nich, `Wheels`, określa informacje o kołach. Drugi, `Vehicle<T>`, jest interfejsem generycznym typu `T`, przy czym parametr typu `T` może oznaczać dowolny inny typ.

Kup ksi k	Pole ksi k

Dalszą część kodu pliku *classGeneric.ts* stanowi definicja klasy `Automobile`, która implementuje interfejs `Vehicle` z parametrem typu `Wheel`, co kojarzy typ `Wheel` z klasą `Automobile`. I w końcu ostatnią definiowaną klasą jest `Chevy`, która dziedziczy po `Automobile` i określa potrzebne wartości domyślne:

```
    class Automobile implements Vehicle<Wheels> {
        constructor(private name: string, private wheels: Wheels){}

        getName(): string {
            return this.name;
        }

        getWheelCount(): Wheels {
            return this.wheels;
        }
    }

    class Chevy extends Automobile {
        constructor() {
            super("Chevy", { count: 4, diameter: 18 });
        }
    }
```

I w końcu, po tych wszystkich definicjach interfejsów i klas, umieść ostatni fragment kodu, w którym tworzymy instancję klasy `Chevy` i wyświetlamy kilka informacji o niej:

```
    const chevy = new Chevy();
    console.log("Nazwa auta: ", chevy.getName());
    console.log("Informacje o kołach: ", chevy.getWheelCount());
}
```

Ten kod można skompilować, a jego wykonanie zwróci wyniki przedstawione na rysunku 2.15.

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

H:\NaukaTypeScriptu\rozdzial02>tsc classGeneric.ts

H:\NaukaTypeScriptu\rozdzial02>node classGeneric.js
Nazwa auta:  Chevy
Informacje o kołach:  { count: 4, diameter: 18 }
```

Rysunek 2.15. Wyniki wykonania skryptu classGenerics.ts

Jak widać, hierarchia dziedziczenia w tym przykładzie ma kilka poziomów głębokości, jednak nasz kod i tak może pomyślnie zwrócić prawidłowy wynik. Choć konkretne szczegóły kodu, z którym będziesz się spotykał w rzeczywistości będą zapewne inne, to jednak w programowaniu obiektowym takie wielopoziomowe hierarchie dziedziczenia spotyka się bardzo często.

W tym podrozdziale poznałeś sposoby używania typów generycznych w funkcjach oraz w klasach i interfejsach. Typy generyczne są powszechnie używane podczas tworzenia aplikacji Reacta, jak również w niektórych pakietach środowiska Node. Dlatego też ich znajomość na pewno Ci się przyda podczas lektury dalszych rozdziałów książki. Na zakończenie

tego rozdziału, w jego ostatniej części, zajmiemy się kilkoma dodatkowymi zagadnieniami, takimi jak stosowanie najnowszych możliwości języka TypeScript, czy konfigurowanie kompilatora.

Prezentacja najnowszych możliwości języka i konfigurowania kompilatora

W tym podrozdziale przedstawię wybrane spośród nowszych możliwości języka TypeScript oraz opiszę możliwości konfigurowania kompilatora TypeScriptu. Zdobycie tych informacji pozwoli Ci na pisanie bardziej przejrzystego, łatwiejszego do zrozumienia kodu, co oczywiście będzie bardzo korzystne podczas pracy w zespole. Z kolei wykorzystanie opcji konfiguracyjnych kompilatora TypeScriptu pozwoli zapewnić, że będzie on działał w sposób, który uznamy za optymalny na potrzeby tworzonego projektu.

Łączenie opcjonalne

Zacznę od przedstawienia **łączenia opcjonalnego** (ang. *optional chaining*). Ta możliwość pozwala na pisanie prostszego kodu, a jednocześnie chroni przed pewną niewielką klasą błędów związanych ze stosowaniem wartości `null`. Utwórz plik *optionalChaining.ts* i zapisz w nim następujący kod:

```
namespace OptionalChainingNS {
    interface Wheels {
        count?: number;
    }

    interface Vehicle {
        wheels?: Wheels;
    }

    class Automobile implements Vehicle {
        constructor(public wheels?: Wheels) {}
    }

    const car: Automobile | null = new Automobile({
        count: undefined
    });
    console.log("Auto: ", car);
    console.log("Informacje o kołach: ", car?.wheels);
    console.log("Liczba kół: ", car?.wheels?.count);
}
```

Jak widać, w tym przykładzie używanych jest jednocześnie kilka typów. Zmienna `car` dysponuje właściwością `wheels`, która z kolei dysponuje właściwością `count`. W końcowej części kodu, w której wyświetlamy na konsoli informacje o tym obiekcie widać, że odwołania do tych właściwości są łączone ze sobą. Na przykład w ostatnim wywołaniu funkcji `console.log`

Kup ksi k Pole ksi k

używamy wyrażenia o postaci `car?.wheels?.count`. Właśnie taki zapis, wykorzystujący operator `?.`, jest nazywany łączeniem opcjonalnym. Zastosowanie znaku zapytania oznacza istnienie możliwości, że obiekt będzie mieć wartość `null` lub `undefined`. Jeśli faktycznie okaże się, że obiekt ma wartość `null` lub `undefined`, to przetwarzanie wyrażenia zakończy się na tym obiekcie i zostanie zwrócona jego wartość — dalsza część wyrażenia zostanie pominięta, lecz nie spowoduje to zgłoszenia żadnego błędu.

A zatem, gdybyśmy chcieli napisać ostatnie wywołanie funkcji `console.log` w starym stylu, musielibyśmy zastosować rozbudowany kod warunkowy, aby upewnić się, że nie doprowadzimy do zgłoszenia błędu odwołując się do właściwości obiektu, który może przyjąć wartość `undefined`. Zapewne zastosowalibyśmy do tego celu operator trójargumentowy, a cały kod mógłby przypominać instrukcję przedstawioną poniżej:

```
const count = !car ? 0
    : !car.wheels ? 0
    : !car.wheels.count ? 0
    : car.wheels.count;
```

Nie ma wątpliwości, że taki kod jest trudny zarówno do napisania, jak i do zrozumienia. Zastosowanie operatora łączenia opcjonalnego sprawia, że pozwalamy kompilatorowi zatrzymać przetwarzanie wyrażenia w momencie napotkania wartości `null` lub `undefined` i zwrócenia jej jako wyniku. Możemy w ten sposób uniknąć pisania kodu bardzo rozbudowanego i potencjalnie podatnego na błędy.

Scalanie wartości pustych

Scalanie wartości pustych (ang. *nullish coalescing*) jest po prostu uproszczonym zapisem operatora trójargumentowego. Z tego względu ta nowa możliwość jest bardzo prosta. Poniżej przedstawiłem przykład jej użycia:

```
const val1 = undefined;
const val2 = 10;
const result = val1 ?? val2;
console.log(result);
```

Całe wyrażenie jest przetwarzane od lewej do prawej i oznacza, że jeśli `val1` jest różne od `null` lub `undefined` i ma jakąś faktyczną wartość, to zostanie ona zwrócona jako wartość wyrażenia. Jeśli jednak `val1` nie ma wartości, to jako wartość wyrażenia zostanie zwrócona wartość `val2`. A zatem, po skompilowaniu i wykonaniu powyższego fragmentu kodu, na konsoli zostałaby wyświetlona liczba `10`.

Można się zastanawiać, czy scalanie wartości pustych jest tym samym co operator `||`? Faktycznie, między tymi rozwiązaniami istnieją pewne podobieństwa, jednak działanie operatora scalania wartości pustych podlega większym ograniczeniom. W przypadku zastosowania operatora alternatywy logicznej sprawdzana jest jedynie „prawdziwość". W języku JavaScript koncepcja „prawdziwości" wiąże się z traktowaniem różnych wartości jako „prawdziwe" lub „fałszywe". Na przykład każda z wartości — `0`, `true`, `false`, `undefined` oraz `""` — ma w języku JavaScript swój logiczny odpowiednik prawdy lub fałszu. Natomiast w przypadku scalania wartości pustych sprawdzane jest jedynie występowanie wartości `null` lub `undefined`.

Kup ksi k

Pole ksi k

Konfigurowanie TypeScriptu

Informacje konfiguracyjne określające sposób działania kompilatora TypeScriptu można przekazywać w wierszu poleceń, bądź też, co jest znacznie częściej stosownym rozwiązaniem, można je zapisywać w pliku *tsconfig.json*. W przypadku podawania ich w wierszu poleceń, wywołanie kompilatora będzie wyglądać podobnie, jak na poniższym przykładzie:

```
tsc plikts.ts -lib 'es5, dom'
```

To polecenie informuje kompilator, że należy zignorować plik *tsconfig.json* i zastosować wyłącznie opcje podane w wierszu polecenia, czyli konkretnie opcję `-lib` określającą używaną wersję języka JavaScript; dodatkowo polecenie nakazuje skompilowanie tylko jednego, podanego pliku *.ts*. Jeśli polecenie będzie zawierało wyłącznie nazwę kompilatora, *tsc*, to TypeScript poszuka pliku konfiguracyjnego *tsconfig.json* i użyje zapisanych w nim ustawień, a dodatkowo skompiluje wszystkie pliki *.ts*, które uda mu się odnaleźć.

Kompilator języka TypeScript udostępnia bardzo wiele opcji, dlatego też nie opiszę tutaj ich wszystkich — ograniczę się do przedstawienia jedynie kilku najważniejszych (kiedy zaczniemy tworzyć aplikacje Reacta, przedstawię plik *tsconfig.json*, którego będziemy używać):

- `--lib` — ta opcja służy do określania wersji języka JavaScript używanej podczas tworzenia projektu;
- `--target` — ta opcja określa wersję języka JavaScript, z którą ma być zgodny generowany kod;
- `--noImplicytAny` — użycie tej opcji nie zezwala na stosowanie typu `any`, jeśli nie został on jawnie zadeklarowany;
- `--outDir` — ta opcja określa katalog, w którym będą zapisywane generowane pliki JavaScript;
- `--outFile` — ta opcja określa nazwę końcowego, generowanego pliku JavaScript;
- `--rootDirs` — ta tablica określa nazwy katalogów zawierających źródłowe pliki *.ts*;
- `--excludes` — ta tablica zawiera nazwy katalogów i plików, których nie należy kompilować;
- `--includes` — ta tablica zawiera nazwy katalogów i plików, które należy skompilować.

W tym punkcie przedstawiłem jedynie pobieżną prezentację wybranych możliwości języka TypeScript, jak również kilka opcji konfiguracyjnych kompilatora TypeScriptu. Niemniej jednak, przedstawione tu najnowsze możliwości języka oraz opcje konfiguracyjne są bardzo istotne i w kolejnych rozdziałach, kiedy już zaczniemy pisać kod aplikacji, będziemy ich bardzo często używać.

Kup ksi k Pole ksi k

Podsumowanie

W tym rozdziale nieco dokładniej nauczyłeś się języka TypeScript. Poznałeś różne typy, które są w nim dostępne, jak również nauczyłeś się definiować własne. Dowiedziałeś się także, jak pisać kod obiektowy w TypeScripcie. Ten rozdział był długi i całkiem trudny, jednak zamieszczone w nim informacje stanowią nieodzowny fundament, na którym będziemy bazować w dalszej części książki podczas pisania aplikacji.

W następnym rozdziale przedstawię najważniejsze cechy tradycyjnego języka JavaScript, jak również wybrane spośród najnowszych możliwości wprowadzonych w jego najnowszych wersjach. Ponieważ TypeScript jest w rzeczywistości nadzbiorem JavaScriptu, bardzo ważnym jest, by znać i rozumieć aktualne możliwości JavaScriptu, gdyż jest to niezbędne, by móc w pełni korzystać z możliwości, jakie zapewnia TypeScript.

Kup ksi k
Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

Nowy wymiar programowania: pełny stos najnowszych technologii!

JavaScript i jego następca, TypeScript, od lat są ulubionymi językami programowania zawodowców. Mogą oni korzystać z całej gamy wspaniałych narzędzi i frameworków, takich jak React, Node.js czy też Redux, Express i GraphQL. Dają one możliwość pisania całych aplikacji, zarówno części klienckich, jak i serwerowych, w jednym języku. Programiści coraz częściej doceniają korzyści, jakie płyną z tworzenia rozwiązań obejmujących pełny stos technologiczny. Jest to o wiele efektywniejszy i bardziej satysfakcjonujący sposób pracy niż tworzenie klasycznych aplikacji internetowych.

To książka przeznaczona dla osób, które posługują się językiem JavaScript i chcą wykorzystać jego możliwości do zbudowania kompletnej aplikacji internetowej. Prezentuje język TypeScript i opisuje jego najlepsze cechy, pokazuje także, w jaki sposób za pomocą takich frameworków jak React, Redux, Node, Express i GraphQL zbudować złożoną aplikację internetową o pełnej funkcjonalności. Wyjaśniono tu tajniki pracy z poszczególnymi elementami całego stosu technologicznego, a przy tym omówiono przydatne narzędzia, techniki i biblioteki. Przedstawiono również sposoby używania bazy danych na potrzeby aplikacji. Ważnym elementem jest dokładny opis wdrażania gotowej aplikacji w chmurze AWS.

W książce między innymi:

- najważniejsze możliwości języka TypeScript
- stosowanie *hooków* Reacta i magazynu Redux
- wdrażanie funkcjonalnych aplikacji za pomocą Reacta i GraphQL
- mechanizm uwierzytelniania z użyciem Redisa
- praca z bazą danych Postgres przy użyciu TypeORM

David Choi od ponad dziesięciu lat tworzy aplikacje korporacyjne. Zdobył doświadczenie w pracy z wieloma frameworkami i językami programowania. Zajmował się zagadnieniami finansowymi w takich firmach jak JPMorgan, CSFB i Franklin Templeton. Aktualnie pracuje nad własnym startupem. Rozwija aplikację DzHaven, która ma pomagać programistom we wspieraniu innych programistów.