



TypeScript

Od początkującego
do profesjonalisty

—

Adam Freeman

Helion 

Apress®

Tytuł oryginału: Essential TypeScript: From Beginner to Pro

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-6531-5

First published in English under the title Essential TypeScript: From Beginner to Pro
by Adam Freeman, edition: 1
Copyright © 2019 by Adam Freeman

This edition has been translated and published under licence from APress Media, LLC,
part of Springer Nature.

APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the
accuracy of the translation.

Polish edition copyright © 2020 by Helion SA
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 jakiegokolwiek 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 Helion SA 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 Helion SA nie ponoszą również
żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/tspopr.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/tspopr>
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.....	15
	O recenzencie technicznym	17
Część I	Zaczynamy.....	19
Rozdział 1	Pierwsza aplikacja w TypeScriptie	21
	Przygotowanie systemu	21
	Krok 1. Instalowanie Node.js	21
	Krok 2. Instalowanie Gita	22
	Krok 3. Instalowanie TypeScriptu	22
	Krok 4. Instalowanie programistycznego edytora tekstu	22
	Utworzenie projektu	23
	Inicjalizacja projektu	24
	Utworzenie pliku konfiguracyjnego kompilatora	24
	Tworzenie pliku kodu TypeScriptu	24
	Kompilowanie i uruchamianie kodu	25
	Definiowanie modelu danych	26
	Dodawanie funkcji do klasy kolekcji	31
	Używanie pakietu zewnętrznego	37
	Dodawanie deklaracji typu dla pakietu JavaScriptu	40
	Dodawanie poleceń	41
	Filtrowanie elementów	41
	Dodawanie zadań	43
	Oznaczanie zadania jako wykonanego	44
	Trwałe przechowywanie danych	48
	Stosowanie klasy trwałego magazynu danych	50
	Podsumowanie	51
Rozdział 2	Poznajemy TypeScript	53
	Dlaczego powinieneś używać języka TypeScript?	54
	Funkcje języka TypeScript zwiększające produktywność programisty	54
	Poznanie wersji JavaScriptu	55

Co powinieneś wiedzieć?	56
Jak skonfigurować środowisko programistyczne?	56
Jaka jest struktura książki?	56
Czy w książce znajdziesz wiele przykładów?	57
Gdzie znajdziesz przykładowe fragmenty kodu?	58
Podsumowanie	59
Rozdział 3 Wprowadzenie do języka JavaScript — część I	61
Przygotowanie projektu	61
Zagmatwany JavaScript	62
Typy języka JavaScript	64
Praca z podstawowymi typami danych	64
Koercja typu	66
Praca z funkcją	69
Praca z tablicą	74
Używanie operatora rozwinięcia w tablicy	76
Praca z obiektem	77
Dodawanie, modyfikowanie i usuwanie właściwości obiektu	78
Używanie operatorów rozwinięcia i resztowego w obiekcie	80
Definiowanie funkcji typu getter i setter	81
Definiowanie metod	83
Słowo kluczowe this	84
Słowo kluczowe this w oddzielnych funkcjach	85
Słowo kluczowe this w metodach	87
Zmiana zachowania słowa kluczowego this	88
Słowo kluczowe this w funkcji strzałki	89
Powrót do problemu początkowego	90
Podsumowanie	91
Rozdział 4 Wprowadzenie do języka JavaScript — część II	93
Przygotowanie projektu	93
Dziedziczenie obiektu JavaScriptu	94
Analizowanie i modyfikowanie prototypu obiektu	95
Tworzenie własnych właściwości	97
Używanie funkcji konstruktora	98
Sprawdzanie typu prototypu	101
Definiowanie statycznych właściwości i metod	102
Używanie klas JavaScriptu	103
Używanie iteratorów i generatorów	106
Używanie generatora	107
Definiowanie obiektów pozwalających na iterację	108
Używanie kolekcji JavaScriptu	111
Sortowanie danych według klucza przy użyciu obiektu	111
Sortowanie danych według klucza przy użyciu obiektu Map	112
Przechowywanie danych według indeksu	114
Używanie modułów	115
Tworzenie modułu JavaScriptu	116
Używanie modułu JavaScriptu	116

	Eksportowanie funkcji z modułu	118
	Definiowanie w modelu wielu funkcjonalności nazwanych	119
	Podsumowanie	120
Rozdział 5	Używanie kompilatora TypeScriptu	121
	Przygotowanie projektu	121
	Struktura projektu	122
	Używanie menedżera pakietów Node	124
	Plik konfiguracyjny kompilatora TypeScriptu	127
	Kompilacja kodu TypeScriptu	128
	Błędy generowane przez kompilator	129
	Używanie trybu monitorowania i wykonywania skompilowanego kodu	131
	Używanie funkcjonalności wersjonowania celu	133
	Wybór plików biblioteki do kompilacji	135
	Wybór formatu modułu	138
	Użyteczne ustawienia konfiguracji kompilatora	141
	Podsumowanie	144
Rozdział 6	Testowanie i debugowanie kodu TypeScriptu	145
	Przygotowanie projektu	145
	Debugowanie kodu TypeScriptu	146
	Przygotowanie do debugowania	146
	Używanie Visual Studio Code do debugowania	147
	Używanie zintegrowanego debuggera Node.js	149
	Używanie funkcji zdalnego debugowania w Node.js	149
	Używanie lintera TypeScriptu	151
	Wyłączanie reguł lintowania	153
	Testy jednostkowe w TypeScriptie	156
	Konfigurowanie frameworka testów	157
	Tworzenie testów jednostkowych	157
	Uruchamianie frameworka testów	158
	Podsumowanie	160
Część II	Praca z językiem TypeScript	161
Rozdział 7	Typowanie statyczne	163
	Przygotowanie projektu	164
	Typy statyczne	166
	Tworzenie typu statycznego za pomocą adnotacji typu	168
	Używanie niejawnie zdefiniowanego typu statycznego	169
	Używanie typu any	171
	Używanie unii typów	175
	Używanie asercji typu	177
	Asercja typu nieoczekiwanego	178
	Używanie wartownika typu	179
	Używanie typu never	181
	Używanie typu unknown	181
	Używanie typów null	183
	Ograniczenie przypisywania wartości null	184
	Usunięcie null z unii za pomocą asercji	185

	Usuwanie wartości null z unii za pomocą wartownika typu	187
	Używanie asercji ostatecznego przypisania	187
	Podsumowanie	189
Rozdział 8	Używanie funkcji	191
	Przygotowanie projektu	192
	Definiowanie funkcji	193
	Ponowne definiowanie funkcji	193
	Parametry funkcji	195
	Wynik działania funkcji	201
	Przeciążanie typu funkcji	204
	Podsumowanie	205
Rozdział 9	Tablice, krotki i wyliczenia	207
	Przygotowanie projektu	208
	Praca z tablicami	209
	Używanie automatycznie ustalonego typu tablicy	211
	Unikanie problemów z automatycznie ustalonym typem tablicy	212
	Unikanie problemów z pustą tablicą	213
	Krotka	214
	Przetwarzanie krotki	215
	Używanie typów krotki	216
	Wyliczenie	217
	Sposób działania wyliczenia	218
	Używanie wyliczenia w postaci ciągu tekstowego	221
	Ograniczenia typu wyliczeniowego	222
	Używanie typu literału wartości	225
	Używanie w funkcji typu literałów wartości	226
	Łączenie typów wartości w typie literałów wartości	226
	Nadpisywanie za pomocą typu literałów wartości	227
	Używanie aliasu typu	229
	Podsumowanie	230
Rozdział 10	Praca z obiektami	231
	Przygotowanie projektu	232
	Praca z obiektami	233
	Używanie adnotacji kształtu typu obiektu	234
	Dopasowanie kształtu typu obiektu	235
	Używanie aliasu typu dla kształtu typu	239
	Radzenie sobie z nadmiarem właściwości	239
	Używanie unii kształtu typu	241
	Typy właściwości unii	242
	Używanie wartownika typu dla obiektu	242
	Używanie złączenia typów	247
	Używanie złączenia do korelacji danych	248
	Łączenie złączeń	250
	Podsumowanie	257

Rozdział 11	Praca z klasami i interfejsami	259
	Przygotowanie projektu	260
	Używanie funkcji konstruktora	261
	Używanie klas	264
	Używanie słów kluczowych kontroli dostępu	265
	Definiowanie właściwości tylko do odczytu	268
	Upraszczenie klasy konstruktora	269
	Używanie dziedziczenia klas	270
	Używanie klasy abstrakcyjnej	273
	Używanie interfejsu	276
	Implementowanie wielu interfejsów	278
	Rozszerzanie interfejsu	279
	Definiowanie opcjonalnych właściwości i metod interfejsu	281
	Definiowanie implementacji interfejsu abstrakcyjnego	283
	Wartownik typu interfejsu	284
	Dynamiczne tworzenie właściwości	285
	Podsumowanie	287
Rozdział 12	Używanie typów generycznych	289
	Przygotowanie projektu	290
	Zrozumienie problemu	291
	Dodawanie obsługi innego typu	292
	Tworzenie klasy generycznej	293
	Argumenty typu generycznego	295
	Używanie argumentów innego typu	295
	Ograniczanie wartości typu generycznego	296
	Definiowanie parametrów wielu typów	299
	Pozostawienie kompilatorowi zadania ustalenia typu argumentu	302
	Rozszerzanie klasy generycznej	303
	Wartownik typu generycznego	307
	Definiowanie metody statycznej w klasie generycznej	309
	Definiowanie interfejsu generycznego	311
	Rozszerzanie interfejsu generycznego	312
	Implementacja interfejsu generycznego	312
	Podsumowanie	316
Rozdział 13	Zaawansowane typy generyczne	317
	Przygotowanie projektu	317
	Używanie kolekcji generycznych	319
	Używanie iteratorów generycznych	321
	Łączenie iteratora i obiektu możliwego do iteracji	322
	Tworzenie klasy umożliwiającej iterację	324
	Używanie typów indeksu	325
	Używanie zapytania typu indeksu	325
	Jawne dostarczanie parametrów typu generycznego dla typów indeksu	326
	Używanie zindeksowanego operatora dostępu	327
	Używanie typu indeksu dla klasy <code>Collection<T></code>	329

Używanie mapowania typu	331
Używanie parametru typu generycznego z typem mapowanym	332
Zmiana modyfikowalności i opcjonalności właściwości	333
Mapowanie określonych właściwości	335
Łączenie transformacji w pojedyncze mapowanie	335
Tworzenie typu z użyciem mapowania	336
Używanie typów warunkowych	337
Zagnieżdżanie typów warunkowych	338
Używanie typu warunkowego w klasie generycznej	339
Używanie typów warunkowych z uniami typów	340
Używanie typów warunkowych podczas mapowania typów	342
Identyfikowanie właściwości określonego typu	343
Automatyczne ustalanie typów dodatkowych w warunkach	344
Podsumowanie	347
Rozdział 14 Praca z JavaScriptem	349
Przygotowanie projektu	350
Dodawanie kodu TypeScriptu do przykładowego projektu	351
Praca z JavaScriptem	354
Dołączanie kodu JavaScriptu w trakcie kompilacji	355
Sprawdzanie typu kodu JavaScriptu	356
Opisywanie typów używanych w kodzie JavaScriptu	357
Używanie komentarzy do opisywania typów	358
Używanie plików deklaracji typu	360
Opisywanie kodu JavaScriptu przygotowanego przez podmioty zewnętrzne	362
Używanie plików deklaracji pochodzących z projektu Definitely Typed	366
Używanie pakietów zawierających deklaracje typu	368
Generowanie plików deklaracji	370
Podsumowanie	372
Część III Tworzenie aplikacji internetowych	373
Rozdział 15 Tworzenie aplikacji internetowej TypeScriptu — część I	375
Przygotowanie projektu	375
Przygotowanie zestawu narzędzi	377
Dodawanie obsługi paczek	377
Dodawanie programistycznego serwera WWW	380
Utworzenie modelu danych	383
Utworzenie źródła danych	384
Generowanie treści HTML-a za pomocą API modelu DOM	387
Dodawanie obsługi stylów Bootstrap CSS	388
Używanie formatu JSX do tworzenia treści HTML-a	391
Sposób działania JSX	392
Konfigurowanie kompilatora TypeScriptu	
i procedury wczytującej pakiet webpack	393
Tworzenie funkcji fabryki	394
Używanie klasy JSX	395
Importowanie funkcji fabryki w klasie JSX	396

Dodawanie funkcjonalności do aplikacji	397
Wyświetlanie filtrowanej listy produktów	397
Wyświetlanie treści i obsługa uaktualnień	401
Podsumowanie	403
Rozdział 16 Tworzenie aplikacji internetowej TypeScriptu — część II	405
Przygotowanie projektu	406
Dodawanie usługi sieciowej	408
Wykorzystanie źródła danych w aplikacji	409
Używanie dekoratorów	411
Używanie metadanych dekoratora	413
Dokończenie aplikacji	416
Dodawanie klasy Header	417
Dodawanie klasy obsługującej szczegóły zamówienia	417
Dodawanie klasy obsługującej potwierdzenie zamówienia	419
Zakończenie pracy nad aplikacją	419
Wdrażanie aplikacji	422
Dodawanie pakietu produkcyjnego serwera HTTP	422
Tworzenie pliku dla trwałego magazynu danych	423
Utworzenie serwera	424
Używanie względnych adresów URL do obsługi żądań danych	424
Kompilacja aplikacji	425
Testowanie gotowej aplikacji	426
Umieszczanie aplikacji w kontenerze	426
Instalowanie Dockera	427
Przygotowanie aplikacji	427
Tworzenie kontenera Dockera	428
Uruchamianie aplikacji	429
Podsumowanie	430
Rozdział 17 Tworzenie aplikacji internetowej Angulara — część I	431
Przygotowanie projektu	432
Konfigurowanie usługi sieciowej	433
Konfigurowanie pakietu Bootstrap CSS	434
Uruchomienie przykładowej aplikacji	435
Rola TypeScriptu w programowaniu z użyciem frameworka Angular	436
Rola TypeScriptu w łańcuchu narzędzi Angulara	436
Poznajemy dwa kompilatory Angulara	437
Utworzenie modelu danych	440
Utworzenie źródła danych	441
Utworzenie implementacji klasy źródła danych	443
Konfigurowanie źródła danych	445
Wyświetlenie filtrowanej listy produktów	446
Wyświetlanie przycisków kategorii	448
Utworzenie nagłówka	449
Połączenie komponentów produktu, kategorii i nagłówka	450
Konfigurowanie aplikacji	451
Podsumowanie	453

Rozdział 18	Tworzenie aplikacji internetowej Angulara — część II	455
	Przygotowanie projektu	456
	Dokończenie pracy nad funkcjonalnością aplikacji	456
	Dodawanie komponentu obsługującego podsumowanie zamówienia	459
	Tworzenie konfiguracji routingu	460
	Wdrażanie aplikacji	462
	Dodawanie pakietu produkcyjnego serwera HTTP	462
	Tworzenie pliku dla trwałego magazynu danych	463
	Utworzenie serwera	463
	Używanie względnych adresów URL do obsługi żądań danych	464
	Kompilacja aplikacji	465
	Testowanie gotowej aplikacji	466
	Umieszczanie aplikacji w kontenerze	467
	Przygotowanie aplikacji	467
	Tworzenie kontenera Dockera	467
	Uruchamianie aplikacji	468
	Podsumowanie	469
Rozdział 19	Tworzenie aplikacji internetowej React — część I	471
	Przygotowanie projektu	472
	Konfigurowanie usługi sieciowej	473
	Instalowanie pakietu Bootstrap CSS	474
	Uruchamianie przykładowej aplikacji	474
	TypeScript i programowanie React	475
	Definiowanie typów encji	478
	Wyświetlanie filtrowanej listy produktów	479
	Używanie zaczepów i komponentów funkcyjnych	481
	Wyświetlanie listy kategorii i nagłówka	483
	Przygotowanie i przetestowanie komponentów	484
	Utworzenie magazynu danych	486
	Utworzenie klasy żądania HTTP	490
	Połączenie komponentów z magazynem danych	491
	Podsumowanie	493
Rozdział 20	Tworzenie aplikacji internetowej React — część II	495
	Przygotowanie projektu	496
	Konfigurowanie routingu URL	496
	Dokończenie pracy nad funkcjonalnością aplikacji	498
	Dodawanie komponentu obsługującego podsumowanie zamówienia	500
	Dodawanie komponentu potwierdzającego złożenie zamówienia	501
	Dokończenie konfiguracji routingu	502
	Wdrażanie aplikacji	503
	Dodawanie pakietu produkcyjnego serwera HTTP	504
	Tworzenie pliku dla trwałego magazynu danych	504
	Utworzenie serwera	505
	Używanie względnych adresów URL do obsługi żądań danych	505
	Kompilacja aplikacji	506
	Testowanie gotowej aplikacji	507

Umieszczanie aplikacji w kontenerze	507
Przygotowanie aplikacji	508
Tworzenie kontenera Dockera	508
Uruchamianie aplikacji	509
Podsumowanie	510
Rozdział 21 Tworzenie aplikacji internetowej Vue.js — część I	511
Przygotowanie projektu	512
Konfigurowanie usługi sieciowej	513
Instalowanie pakietu Bootstrap CSS	514
Uruchamianie przykładowej aplikacji	515
TypeScript i programowanie w Vue.js	515
Zestaw narzędzi TypeScriptu podczas programowania z użyciem frameworka Vue.js	517
Utworzenie klas encji	518
Wyświetlanie filtrowanej listy produktów	519
Wyświetlanie listy kategorii i nagłówka	521
Tworzenie i testowanie komponentów	523
Utworzenie magazynu danych	526
Utworzenie dekoratorów magazynu danych	527
Połączenie komponentów z magazynem danych	528
Dodawanie obsługi usługi sieciowej	530
Podsumowanie	534
Rozdział 22 Tworzenie aplikacji internetowej Vue.js — część II	535
Przygotowanie projektu	536
Konfigurowanie routingu URL	536
Dokończenie pracy nad funkcjonalnością aplikacji	538
Dodawanie komponentu obsługującego podsumowanie zamówienia	540
Dodawanie komponentu potwierdzającego złożenie zamówienia	541
Dokończenie konfiguracji routingu	542
Wdrażanie aplikacji	543
Dodawanie pakietu produkcyjnego serwera HTTP	543
Tworzenie pliku dla trwałego magazynu danych	544
Utworzenie serwera	544
Używanie względnych adresów URL do obsługi żądań danych	545
Kompilacja aplikacji	546
Testowanie gotowej aplikacji	546
Umieszczanie aplikacji w kontenerze	547
Przygotowanie aplikacji	547
Tworzenie kontenera Dockera	547
Uruchamianie aplikacji	548
Podsumowanie	549

ROZDZIAŁ 1



Pierwsza aplikacja w TypeScriptie

Najlepszym sposobem na rozpoczęcie pracy z TypeScriptem jest zagłębienie się w ten język. W tym rozdziale przedstawię prosty proces programistyczny polegający na utworzeniu aplikacji pozwalającej na zdefiniowanie listy rzeczy do zrobienia. W kolejnych rozdziałach dokładniej poznasz sposób działania poszczególnych funkcji TypeScriptu, natomiast pokazany tutaj prosty przykład jest w zupełności wystarczający do zaprezentowania podstawowych funkcji tego języka. Nie przejmuj się, jeśli nie wszystko w rozdziale będzie dla Ciebie zrozumiałe. Moim celem jest jedynie pokazanie ogólnego trybu działania języka TypeScript i sposobu używania jego poszczególnych konstrukcji podczas tworzenia aplikacji.

Przygotowanie systemu

Aby móc wykonywać przykłady przedstawione w książce, konieczne jest zainstalowanie czterech pakietów. Przeprowadź instalacje zgodnie z informacjami zamieszczonymi w kolejnych sekcjach i upewnij się, że pakiety działają zgodnie z oczekiwaniami.

Krok 1. Instalowanie Node.js

Przede wszystkim musisz pobrać i zainstalować środowisko uruchomieniowe Node.js (znane również pod nazwą Node). Niezbędny pakiet instalacyjny znajdziesz na stronie <https://nodejs.org/dist/v12.0.0/>. Na podanej stronie znajdują się pakiety instalacyjne dla wszystkich platform obsługiwanych przez wydanie 12.0.0 Node.js, czyli w wersji, której użyłem w przykładach omówionych w książce. Podczas instalacji upewnij się o wybraniu opcji powodującej zainstalowanie również menedżera pakietów Node.js (ang. *node package manager*). Po zakończeniu procesu instalacji przejdź do powłoki lub wiersza poleceń, a następnie wydaj polecenie przedstawione na listingu 1.1, aby w ten sposób sprawdzić poprawność działania Node.js i menedżera pakietów (npm).

Listing 1.1. Sprawdzanie poprawności instalacji Node i menedżera pakietów (npm)

```
$ node --version  
$ npm --version
```

Wygenerowane dane wyjściowe pierwszego polecenia powinny mieć postać `v12.0.0` wskazującą na dostępność Node.js w wymaganej wersji. Natomiast dane wyjściowe drugiego polecenia powinny mieć postać `6.9.0` wskazującą na prawidłowe zainstalowanie menedżera pakietów Node.js.

Krok 2. Instalowanie Gita

Drugim krokiem jest pobranie i zainstalowanie narzędzia systemu kontroli wersji Git — znajdziesz je na stronie <https://git-scm.com/download/>. Wprawdzie system Git nie jest ściśle wymagany podczas programowania w języku TypeScript, ale jest niezbędny dla części pakietów najczęściej używanych z TypeScriptem. Po zakończeniu instalacji użyj powłoki (Linux) lub wiersza poleceń (Windows) w celu wydania polecenia przedstawionego na listingu 1.2 i tym samym sprawdzenia, czy Git faktycznie działa.

Listing 1.2. Sprawdzenie poprawności instalacji narzędzia Git

```
$ git --version
```

W chwili powstawania książki najnowszą dostępną wersją narzędzia Git na wszystkich platformach była wersja 2.21.0.

Krok 3. Instalowanie TypeScriptu

Trzecim krokiem jest zainstalowanie pakietu języka TypeScript. Powłokę lub wiersz poleceń wykorzystaj do wydania polecenia przedstawionego na listingu 1.3.

Listing 1.3. Instalowanie pakietu języka TypeScript

```
$ npm install --global typescript@3.5.1
```

Po zainstalowaniu pakietu wydaj polecenie przedstawione na listingu 1.4, aby mieć pewność o prawidłowym zainstalowaniu kompilatora.

Listing 1.4. Testowanie poprawności instalacji kompilatora TypeScriptu

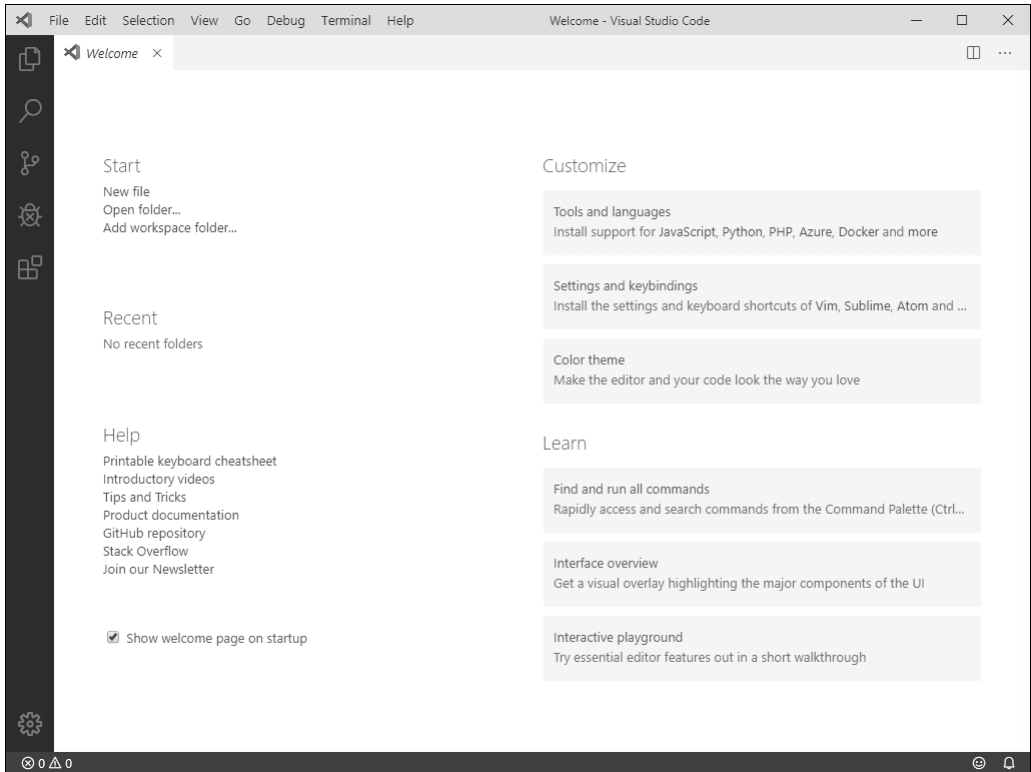
```
$ tsc --version
```

Kompilator języka TypeScript nosi nazwę `tsc`, a dane wyjściowe po wykonaniu polecenia przedstawionego na listingu 1.4 powinny mieć postać `Version 3.5.1`.

Krok 4. Instalowanie programistycznego edytora tekstu

Ostatnim krokiem jest zainstalowanie programistycznego edytora tekstu, który zapewnia obsługę języka TypeScript. Podczas programowania w TypeScriptie można skorzystać z większości popularnych edytorów tekstu. Jeżeli nie masz żadnego ulubionego, wówczas pobierz i zainstaluj Visual Studio Code ze strony <https://code.visualstudio.com/>. Visual Studio Code to edytor programistyczny rozprowadzany jako oprogramowanie typu *open source* dostępne dla wielu platform. Z tego edytora będę korzystał podczas tworzenia przykładów omówionych w książce.

Po zainstalowaniu Visual Studio Code polecenie `code` powoduje uruchomienie edytora. Ewentualnie możesz skorzystać z ikony programu dodanej podczas instalacji. Po uruchomieniu zobaczysz okno powitalne podobne do pokazanego na rysunku 1.1. (Konieczne może być zmodyfikowanie systemowej ścieżki dostępu, zanim będzie można korzystać z polecenia `code`).



Rysunek 1.1. Okno powitalne edytora Visual Studio Code

Utworzenie projektu

Aby rozpocząć pracę z TypeScriptem, zamierzam utworzyć prostą aplikację pozwalającą na zdefiniowanie listy rzeczy do zrobienia. Język TypeScript jest najczęściej używany do tworzenia aplikacji internetowych, co w trzeciej części książki pokażę na przykładzie kilku najpopularniejszych frameworków (Angular, React i Vue.js). Natomiast w tym rozdziale utworzę aplikację działającą w powłoce, co pozwoli skoncentrować się na samym języku TypeScript i uniknąć niepotrzebnego poziomu skomplikowania narzucanego przez framework aplikacji internetowej.

Działanie aplikacji budowanej w rozdziale polega na wyświetleniu listy rzeczy do zrobienia. Użytkownik będzie miał możliwość dodawania nowych zadań, a także oznaczania istniejących zadań jako już wykonanych. Dostępna będzie również opcja pozwalająca filtrować na liście wykonane zadania. Gdy podstawowa funkcjonalność aplikacji zostanie ukończona, przystąpię do implementacji obsługi trwałego magazynu danych, aby zmiany wprowadzone w aplikacji nie były tracone po zakończeniu jej działania.

Inicjalizacja projektu

Aby przygotować katalog projektu, w powłoce lub w wierszu poleceń przejdź do wybranego katalogu, a następnie utwórz w nim podkatalog o nazwie *todo*. Teraz wydaj polecenia przedstawione na listingu 1.5, które spowodują przejście do nowo utworzonego katalogu i inicjalizację nowego projektu.

Listing 1.5. Inicjalizacja katalogu projektu

```
$ cd todo
$ npm init --yes
```

Działanie polecenia `npm init` polega na utworzeniu pliku *package.json*, który jest używany do monitorowania pakietów wymaganych przez projekt, a także do konfiguracji narzędzi programistycznych.

Utworzenie pliku konfiguracyjnego kompilatora

Pakiet TypeScriptu zainstalowany po wykonaniu polecenia przedstawionego na listingu 1.3 w wcześniejszej części rozdziału obejmuje kompilator `tsc` odpowiedzialny za kompilację kodu TypeScriptu na postać czystego kodu JavaScriptu. W celu zdefiniowania konfiguracji kompilatora TypeScriptu należy w katalogu projektu (tutaj *todo*) utworzyć plik o nazwie *tconfig.json* z zawartością przedstawioną na listingu 1.6.

Listing 1.6. Zawartość pliku *tconfig.json* w katalogu *todo*

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "commonjs"
  }
}
```

Dokładne omówienie kompilatora TypeScriptu znajdziesz w rozdziale 5. Teraz wystarczy wiedzieć, że przedstawione tutaj polecenia nakazują kompilatorowi użycie najnowszej wersji JavaScriptu, wskazują mu katalog *src* jako zawierający pliki TypeScriptu projektu, określają katalog *dist* jako docelowy dla wygenerowanych danych wyjściowych, a także nakazują użycie standardu *commonjs* podczas wczytywania kodu z oddzielnych plików.

Tworzenie pliku kodu TypeScriptu

Plik kodu TypeScriptu ma rozszerzenie *.ts*. W celu dodania pierwszego pliku kodu do projektu zacznij od utworzenia w katalogu projektu *todo* podkatalogu *src*, a następnie umieść w nim plik o nazwie *index.ts* z kodem przedstawionym na listingu 1.7. Ten plik przedstawia popularną konwencję nadania plikowi głównemu aplikacji nazwy *index* z rozszerzeniem *.ts* wskazującym na to, że zawartością pliku jest kod JavaScriptu.

Listing 1.7. Zawartość pliku *index.ts* w katalogu *src*

```
console.clear();
console.log("Lista Adama");
```


Ten plik zawiera zwykle polecenia JavaScriptu używające obiektu `console` do usunięcia zawartości okna powłoki, a następnie do wyświetlenia prostego komunikatu. To wystarczająca funkcjonalność do sprawdzenia, czy wszystko na pewno działa, zanim przystąpimy do pracy nad rzeczywistą aplikacją.

Kompilowanie i uruchamianie kodu

Plik TypeScriptu musi zostać skompilowany — dopiero wtedy na jego podstawie jest generowany kod JavaScriptu, który później będzie mógł być wykonany przez przeglądarkę WWW lub środowisko uruchomieniowe Node.js przygotowane na początku rozdziału. Polecenie przedstawione na listingu 1.8 należy wydać w katalogu *todo*, aby w ten sposób uruchomić kompilator.

Listing 1.8. Uruchamianie kompilatora TypeScriptu

```
$ tsc
```

Kompilator odczytuje ustawienia konfiguracyjne zapisane w pliku *tsconfig.json*, a potem odszukuje pliki TypeScriptu w katalogu *src*. Następnie tworzy podkatalog o nazwie *dist* i umieszcza w nim wygenerowany kod JavaScriptu. Jeżeli przeanalizujesz zawartość podkatalogu *dist*, to znajdziesz w nim plik *index.js*. Rozszerzenie *.js* wskazuje, że plik zawiera kod JavaScriptu. Z kolei po przeanalizowaniu pliku *index.js* przekonasz się, że zawiera on następujące polecenia:

```
console.clear();
console.log("Lista Adama");
```

Pliki TypeScriptu i JavaScriptu zawierają te same polecenia, ponieważ jeszcze nie zostały użyte żadne funkcje oferowane przez język TypeScript. Gdy aplikacja zacznie nabierać kształtu, zawartość plików TypeScriptu będzie różniła się od plików JavaScriptu wygenerowanych przez kompilator.

-
- **Ostrzeżenie** Nie wprowadzaj zmian w plikach znajdujących się w katalogu *dist*, ponieważ będą one nadpisane w trakcie następnego uruchomienia kompilatora. Podczas programowania w języku TypeScript zmiany są wprowadzane w plikach z rozszerzeniem *.ts*, na podstawie których później są generowane pliki JavaScriptu z rozszerzeniem *.js*.
-

Aby uruchomić skompilowany kod, w powłoce przejdź do katalogu projektu, a następnie wydaj polecenie przedstawione na listingu 1.9.

Listing 1.9. Uruchomienie skompilowanego kodu

```
$ node dist/index.js
```

Polecenie `node` powoduje przejście do środowiska uruchomieniowego Node.js JavaScriptu, a argument wskazuje na plik, którego zawartość ma być wykonana. Jeżeli narzędzia programistyczne zostały zainstalowane prawidłowo, zawartość powłoki będzie usunięta i zobaczysz wyświetlone następujące dane wyjściowe:

```
Lista Adama
```

Definiowanie modelu danych

Działanie przykładowej aplikacji ma polegać na zarządzaniu listą rzeczy do zrobienia. Użytkownik będzie miał możliwość wyświetlenia listy, dodawania nowych elementów, oznaczania zadań jako wykonanych, a także filtrowania zawartości listy. W tej sekcji zacznę używać języka TypeScript do zdefiniowania modelu danych opisującego dane aplikacji i operacje, które mogą być na nich przeprowadzane. Pracę trzeba rozpocząć od utworzenia w katalogu `src` pliku o nazwie `todoItem.ts` i umieszczenia w nim kodu przedstawionego na listingu 1.10.

Listing 1.10. Zawartość pliku `todoItem.ts` w katalogu `src`

```
export class TodoItem {
  public id: number;
  public task: string;
  public complete: boolean = false;

  public constructor(id: number, task: string, complete: boolean = false) {
    this.id = id;
    this.task = task;
    this.complete = complete;
  }

  public printDetails(): void {
    console.log(`${this.id}\t${this.task} ${this.complete
      ? "\t(wykonane)": ""}`);
  }
}
```

Klasa to szablon opisujący typ danych. Dokładne omówienie klas znajdziesz w rozdziale 4. Kod przedstawiony na listingu 1.10 będzie wyglądał znajomo dla każdego, kto ma doświadczenie w programowaniu w językach takich jak C# lub Java, nawet jeśli dokładny sposób działania tego kodu pozostaje niejasny.

Klasa przedstawiona na listingu 1.10 nosi nazwę `TodoItem`, definiuje właściwości `id`, `task` i `complete`, a także metodę `printDetails()` wyświetlającą w konsoli podsumowanie danego zadania. TypeScript jest zbudowany na bazie JavaScriptu, więc nic dziwnego, że kod znajdujący się na listingu 1.10 jest połączeniem standardowej funkcjonalności JavaScriptu z usprawnieniami oferowanymi przez TypeScript. JavaScript obsługuje klasy z konstruktorami, właściwościami i metodami, natomiast funkcje takie jak słowa kluczowe kontroli dostępu (np. `public`) są dostarczane przez TypeScript. Ważną cechą języka TypeScript jest statyczne typowanie, co pozwala na wyraźne zdefiniowanie typu każdej właściwości i każdego parametru klasy `TodoItem`:

```
...
public id: number;
...
```

To jest przykład adnotacji typu wskazującej kompilatorowi TypeScriptu, że właściwości `id` można przypisać jedynie wartości typu `number`. Jak wyjaśnię w rozdziale 3., JavaScript stosuje dość elastyczne podejście do typów. Największą zaletą oferowaną przez TypeScript jest zapewnienie większej zgodności typów danych z innymi językami programowania przy jednoczesnym zachowaniu dostępu do normalnego podejścia JavaScriptu, gdy zachodzi potrzeba.

- **Wskazówka** Nie przejmuj się, jeśli nie znasz sposobu, w jaki JavaScript obsługuje typy danych. W rozdziałach 3. i 4. znajdziesz więcej informacji o funkcjach JavaScriptu, które trzeba opanować, aby móc efektywnie programować w języku JavaScript.

Klasę przedstawioną na listingu 1.10 utworzyłem w celu podkreślenia podobieństwa zachodzącego między TypeScriptem a językami takimi jak C# i Java. Jednak to nie jest sposób, w jaki zwykle definiuje się klasę w kodzie TypeScriptu. Dlatego też na listingu 1.11 przedstawiłem zmodyfikowaną wersję klasy `TodoItem` wykorzystującą funkcje TypeScriptu pozwalające na zwięzłe definiowanie klas.

Listing 1.11. Zwięźlejsza postać kodu pliku `todoItem.ts` znajdującego się w katalogu `src`

```
export class TodoItem {
    constructor(public id: number,
                public task: string,
                public complete: boolean = false) {
        // Polecenia nie są wymagane.
    }

    printDetails(): void {
        console.log(`${this.id}\t${this.task} ${this.complete}
            ? "\t(wykonane)": ""`);
    }
}
```

Obsługa statycznych typów danych to tylko jeden z aspektów języka TypeScript mających na celu zapewnienie możliwości wygenerowania przewidywalnego i działającego bezpiecznie kodu JavaScriptu. Zwięzła składnia przedstawiona na listingu 1.11 pozwala klasie `TodoItem` otrzymywać parametry i używać ich do tworzenia właściwości egzemplarza w jednym kroku, unikając tym samym podatnego na błędy podejścia polegającego na definiowaniu właściwości i wyraźnego przypisywania jej wartości przekazywanych za pomocą parametru.

Zmiana w metodzie `printDetails()` polega na usunięciu kontrolującego sposób dostępu słowa kluczowego `public`, które tak naprawdę jest niepotrzebne — w języku TypeScript przyjęto założenie, że wszystkie metody i właściwości są domyślnie publiczne, o ile nie zostało użyte słowo kluczowe wskazujące na inny rodzaj dostępu. (W konstruktorze nadal znajduje się słowo kluczowe `public`, ponieważ w ten sposób kompilator TypeScriptu rozpoznaje, że ma do czynienia ze zwięzłą składnią konstruktora. Więcej informacji na ten temat znajdziesz w rozdziale 11.).

Tworzenie klasy kolekcji elementów listy rzeczy do zrobienia

Następnym krokiem jest utworzenie klasy zbierającej wszystkie elementy listy rzeczy do zrobienia, co pozwoli na znacznie łatwiejsze zarządzanie nimi. W katalogu `src` utwórz plik o nazwie `todoCollection.ts` i umieść w nim kod przedstawiony na listingu 1.12.

Listing 1.12. Zawartość pliku `todoCollection.ts` w katalogu `src`

```
import { TodoItem } from "./todoItem";

export class TodoCollection {
    private nextId: number = 1;
```

```

constructor(public userName: string, public todoItems: TodoItem[] = []) {
    // Polecenia nie są wymagane.
}

addTodo(task: string): number {
    while (this.getTodoById(this.nextId) {
        this.nextId++;
    }
    this.todoItems.push(new TodoItem(this.nextId, task));
    return this.nextId;
}

getTodoById(id: number) : TodoItem {
    return this.todoItems.find(item => item.id === id);
}

markComplete(id: number, complete: boolean) {
    const todoItem = this.getTodoById(id);
    if (todoItem) {
        todoItem.complete = complete;
    }
}
}

```

Implementowanie podstawowych funkcji modelu danych

Zanim przejdę dalej, chciałbym się upewnić o prawidłowym działaniu początkowo zaimplementowanych funkcji klasy `TodoCollection`. Temat przeprowadzania testów jednostkowych w projektach TypeScriptu zostanie omówiony w rozdziale 6., natomiast w tym miejscu wystarczające będzie utworzenie obiektów typu `TodoItem` i umieszczenie ich w obiekcie `TodoCollection`. Na listingu 1.13 przedstawiłem kod, którym należy zastąpić dotychczasową zawartość pliku `index.ts` utworzonego na początku rozdziału.

Listing 1.13. Testowanie modelu danych za pomocą kodu pliku `index.ts` znajdującego się w katalogu `src`

```

import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos = [
    new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
    new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwonić do Janka", true)];

let collection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}a`);

let newId = collection.addTodo("Iść pobić");
let todoItem = collection.getTodoById(newId);
console.log(JSON.stringify(todoItem));

```

Wszystkie polecenia przedstawione na listingu 1.13 to standardowe funkcje JavaScriptu. Polecenie `import` jest używane w celu zadeklarowania zależności od klas `TodoItem` i `TodoCollection`. To polecenie jest częścią funkcji modułów w JavaScriptcie pozwalającej na definiowanie kodu w wielu plikach (dokładnie omówię to w rozdziale 4.). Definiowanie tablicy i używanie słowa kluczowego `new`

w celu utworzenia egzemplarzy klas to również standardowe funkcje JavaScriptu, podobnie jak wywołania obiektu `console`.

-
- **Uwaga** Kod przedstawiony na listingu 1.13 używa funkcji będących dodatkami do języka JavaScript. Jak to wyjaśnię w rozdziale 5., kompilator TypeScriptu znacznie ułatwia korzystanie z nowoczesnych funkcji języka JavaScript, takich jak słowo kluczowe `let`, nawet jeśli nie są one obsługiwane przez środowisko uruchomieniowe przeznaczone do wykonania tego kodu, np. starsze wersje przeglądarek WWW. Funkcje JavaScriptu niezbędne do efektywnego programowania w języku TypeScript omówię w rozdziałach 3. i 5.
-

Kompilator TypeScriptu stara się być użyteczny dla programisty. W trakcie kompilacji następuje sprawdzenie używanych typów danych, a informacje podane w definicjach klas `TodoItem` i `TodoCollection` okazują się pomocne podczas ustalania typów danych wykorzystywanych w kodzie przedstawionym na listingu 1.13. Kod wygenerowany przez kompilator nie będzie zawierał żadnych statycznych informacji o typie, mimo to kompilator i tak może przeprowadzić sprawdzenie kodu pod kątem zapewnienia bezpieczeństwa w trakcie jego wykonywania. Aby przekonać się, jak to działa, dodaj do pliku `index.ts` polecenie przedstawione na listingu 1.14.

Listing 1.14. Dodawanie polecenia do pliku `index.ts` znajdującego się w katalogu `src`

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwońić do Janka", true)];

let collection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}`);

let newId = collection.addToDo("Iść pobiegać");
let todoItem = collection.getToDoById(newId);
todoItem.printDetails();

collection.addToDo(todoItem);
```

Nowe polecenie wywołuje metodę `collection.addToDo()` z argumentem w postaci obiektu typu `TodoItem`. Kompilator sprawdza definicję metody `addToDo()` w pliku `todoItem.ts`, a następnie ustala, że oczekuje ona innego typu danych:

```
...
addToDo(task: string): number {
  while (this.getToDoById(this.nextId)) {
    this.nextId++;
  }
  this.todoItems.push(new TodoItem(this.nextId, task));
  return this.nextId;
}
...
```

Informacje o typie metody `addTodo()` wskazują kompilatorowi, że parametr `task` musi być typu `string`, a wynik musi być typu `number`. (Typy `string` i `number` są typami wbudowanymi w JavaScript; więcej informacji na ich temat znajdziesz w rozdziale 3.). Wykonanie w katalogu `todo` polecenia przedstawionego na listingu 1.15 spowoduje kompilację kodu.

Listing 1.15. Uruchamianie kompilatora

```
$ tsc
```

Kompilator TypeScriptu przetworzy kod projektu i odkryje, że wartość parametru używanego do wywoływania metody `addTodo()` ma nieprawidłowy typ danych. W efekcie zostanie wygenerowany następujący komunikat błędu:

```
src/index.ts:17:20 - error TS2345: Argument of type 'TodoItem' is not assignable to
parameter of type 'string'.
17 collection.addTodo(todoItem);
                        ~~~~~~

Found 1 error.
```

TypeScript doskonale radzi sobie z wykrywaniem problemów i ustaleniem tego, co tak naprawdę się dzieje w kodzie. Dlatego też w projekcie możesz podać jedynie minimalną ilość informacji o typie. W książce będę podawał informacje o typie, aby listingi były łatwiejsze do zrozumienia, ponieważ wiele omówionych przykładów ma związek ze sposobem, w jaki kompilator TypeScriptu obsługuje typy danych. Na listingu 1.16 przedstawiłem dodanie do pliku `index.ts` poleceń definiujących typy i umieszczenie w komentarzu polecenia, które spowodowało wygenerowanie błędu.

Listing 1.16. Dodawanie informacji o typach do pliku `index.ts` znajdującego się w katalogu `src`

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);
usunąćlet newId: number = collection.addTodo("Iść pobić");
let todoItem: TodoItem = collection.getTodoById(newId);
todoItem.printDetails();

//collection.addTodo(todoItem);
```

Wprowadź informacje dodane przez nowe polecenia przedstawione na listingu 1.16 nie zmieniają sposobu działania kodu, ale powodują wyraźne użycie typów danych, co może ułatwić zrozumienie przeznaczenia tego kodu, a kompilator nie musi już samodzielnie ustalać niezbędnych typów danych. W katalogu `todo` wydaj polecenia przedstawione na listingu 1.17, aby w ten sposób skompilować i uruchomić kod.

Listing 1.17. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Po wykonaniu kodu zostaną wygenerowane następujące dane wyjściowe:

```
Lista Adama
5      Iść pobiegać
```

Dodawanie funkcji do klasy kolekcji

Następnym krokiem jest dodanie nowych możliwości do klasy `TodoCollection`. Przede wszystkim trzeba zmienić sposób przechowywania obiektów `TodoItem` — teraz będzie użyty obiekt `Map` JavaScriptu, jak pokazałem na listingu 1.18.

Listing 1.18. Użycie obiektu typu `Map` w kodzie pliku `todoCollection.ts` znajdującym się w katalogu `src`

```
import { TodoItem } from "./todoItem";

export class TodoCollection {
  private nextId: number = 1;
  private itemMap = new Map<number, TodoItem>();

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
  }

  addTodo(task: string): number {
    while (this.getTodoById(this.nextId)) {
      this.nextId++;
    }
    this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
    return this.nextId;
  }

  getTodoById(id: number) : TodoItem {
    return this.itemMap.get(id);
  }

  markComplete(id: number, complete: boolean) {
    const todoItem = this.getTodoById(id);
    if (todoItem) {
      todoItem.complete = complete;
    }
  }
}
```

TypeScript obsługuje typy generyczne, czyli tak naprawdę miejsca zarezerwowane, które są wypełniane podczas tworzenia obiektu. Przykładowo obiekt `Map` w JavaScriptcie to ogólnego przeznaczenia kolekcja przechowująca pary klucz-wartość. Ponieważ JavaScript ma system typów dynamicznych, kolekcję `Map` można wykorzystać do przechowywania elementów różnych typów danych, używając do tego odmiennych typów kluczy. Aby ograniczyć typy możliwe do obsługi przez kolekcję `Map`, w kodzie przedstawionym na listingu 1.18 dostarczyłem ogólne argumenty typu wskazujące kompilatorowi TypeScriptu typy dozwolone do używania przez klucze i wartości:

```
...
private itemMap = new Map<number, TodoItem>();
...
```

Argumenty typu generycznego zostały umieszczone w nawiasie ostrym, a kolekcja `Map` na listingu 1.18 otrzymująca te argumenty wskazuje kompilatorowi, że będzie przechowywała obiekty typu `TodoItem` z kluczami w postaci wartości typu `number`. Kompilator wygeneruje komunikat błędu, jeśli polecenie spróbuje przechowywać w tym obiekcie inny typ danych lub jeśli klucz nie będzie wartością typu `number`. Typy generyczne to ważna cecha języka TypeScript, a ich dokładne omówienie znajdziesz w rozdziale 12.

Zapewnienie dostępu do elementów listy rzeczy do zrobienia

Klasa `TodoCollection` definiuje metodę `getTodoById()`, ale aplikacja musi mieć możliwość wyświetlenia listy elementów, które opcjonalnie będą filtrowane, aby wyeliminować już wykonane zadania. Na listingu 1.19 pokazałem dodanie metody zapewniającej dostęp do obiektów `TodoItem`, którymi zarządza `TodoCollection`.

Listing 1.19. Modyfikacja kodu pliku `todoCollection.ts` w katalogu `src` mająca na celu zapewnienie dostępu do elementów listy

```
import { TodoItem } from "./todoItem";

export class TodoCollection {
  private nextId: number = 1;
  private itemMap = new Map<number, TodoItem>();

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
  }

  addTodo(task: string): number {
    while (this.getTodoById(this.nextId)) {
      this.nextId++;
    }
    this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
    return this.nextId;
  }

  getTodoById(id: number) : TodoItem {
    return this.itemMap.get(id);
  }

  getTodoItems(includeComplete: boolean): TodoItem[] {
    return [...this.itemMap.values()]
      .filter(item => includeComplete || !item.complete);
  }

  markComplete(id: number, complete: boolean) {
    const todoItem = this.getTodoById(id);
    if (todoItem) {
      todoItem.complete = complete;
    }
  }
}
```


Metoda `getTodoItems()` pobiera obiekty z kolekcji `Map`, używając jej metody `values()`, a następnie na podstawie tych obiektów tworzy tablicę za pomocą operatora JavaScriptu w postaci trzech kropek. Obiekty są przetwarzane z użyciem metody `filter()` w celu pobrania wymaganych obiektów na podstawie parametru `includeComplete`.

Kompilator TypeScriptu wykorzystuje otrzymane informacje o typach do ich monitorowania na każdym kroku. Argumenty typu generycznego zostały użyte do utworzenia kolekcji `Map` wskazującej kompilatorowi, że zawiera ona obiekty `TodoItem`. Dlatego też kompilator wie, że wartością zwrótną metody `values()` są obiekty typu `TodoItem`, który jest jednocześnie typem obiektów w tablicy. Podążając za tym, kompilator wie, że funkcja przekazana metodzie `filter()` będzie przetwarzała obiekty `TodoItem`, z których każdy zdefiniuje właściwość `complete`. Jeżeli nastąpi próba odczytania właściwości lub metody niezdefiniowanej przez klasę `TodoItem`, kompilator wygeneruje komunikat błędu. Błąd zostanie zgłoszony również wtedy, gdy wynik zwrócony przez polecenie `return` nie będzie odpowiadał typowi wyniku zadeklarowanego przez metodę.

Na listingu 1.20 przedstawiłem uaktualniony kod pliku `index.ts`, który teraz używa nowej funkcji klasy `TodoCollection` i wyświetla użytkownikowi listę rzeczy do zrobienia.

Listing 1.20. Pobieranie elementów kolekcji przez kod pliku `index.ts` znajdującego się w katalogu `src`

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}`);

collection.getTodoItems(true).forEach(item => item.printDetails());
```

Nowe polecenie wywołuje metodę `getTodoItems()` zdefiniowaną na listingu 1.19 i używa standardowej metody JavaScriptu `forEach()` w celu wyświetlenia opisu poszczególnych obiektów `TodoItem` za pomocą obiektu `console`.

W katalogu `todo` wydaj polecenia przedstawione na listingu 1.21, aby w ten sposób skompilować i uruchomić kod.

Listing 1.21. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Po wykonaniu kodu zostaną wygenerowane następujące dane wyjściowe:

```
Lista Adama
1    Kupić kwiaty
2    Odebrać buty
3    Zamówić bilety
4    Zadzwoń do Janka      (wykonane)
```

Usuwanie wykonanych zadań

Dodawanie nowych zadań i później oznaczanie ich jako wykonanych będzie prowadziło do zwiększania się liczby elementów kolekcji, która w pewnym momencie stanie się trudna w zarządzaniu. Dlatego też na listingu 1.22 przedstawiłem metodę pozwalającą na usunięcie z kolekcji już wykonanych zadań.

Listing 1.22. Modyfikacja kodu w pliku `todoCollection.ts` znajdującym się w katalogu `src` pozwalająca na usuwanie wykonanych zadań z kolekcji

```
import { TodoItem } from "./todoItem";

export class TodoCollection {
  private nextId: number = 1;
  private itemMap = new Map<number, TodoItem>();

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
  }

  addTodo(task: string): number {
    while (this.getTodoById(this.nextId)) {
      this.nextId++;
    }
    this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
    return this.nextId;
  }

  getTodoById(id: number) : TodoItem {
    return this.itemMap.get(id);
  }

  getTodoItems(includeComplete: boolean): TodoItem[] {
    return [...this.itemMap.values()]
      .filter(item => includeComplete || !item.complete);
  }

  markComplete(id: number, complete: boolean) {
    const todoItem = this.getTodoById(id);
    if (todoItem) {
      todoItem.complete = complete;
    }
  }

  removeComplete() {
    this.itemMap.forEach(item => {
      if (item.complete) {
        this.itemMap.delete(item.id);
      }
    });
  }
}
```

Funkcja `removeComplete()` używa metody `Map.forEach()` do przeanalizowania przechowywanych w kolekcji `Map` elementów `TodoItem` i wywołuje metodę `delete()` dla tych, których właściwość `complete` ma przypisaną wartość `true`. Uaktualniony kod w pliku `index.ts` pozwalający na wywołanie metody `removeComplete()` przedstawiłem na listingu 1.23.

Listing 1.23. Sprawdzanie możliwości usunięcia z kolekcji już wykonanego zadania — zmianę należy wprowadzić w pliku `src/index.ts`

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}`);

collection.removeComplete();
collection.getTodoItems(true).forEach(item => item.printDetails());
```

W katalogu `todo` wydaj polecenia przedstawione na listingu 1.24, aby w ten sposób skompilować i uruchomić kod.

Listing 1.24. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Po wykonaniu kodu zostaną wygenerowane następujące dane wyjściowe potwierdzające usunięcie wykonanych zadań z kolekcji:

```
Lista Adama
1      Kupić kwiaty
2      Odebrać buty
3      Zamówić bilety
```

Obsługa licznika elementów

Ostatnią funkcjonalnością, którą chcę dodać do klasy `TodoCollection`, jest obsługa licznika całkowitej liczby elementów `TodoItem`, liczby zadań już wykonanych i liczby zadań wciąż oczekujących na wykonanie.

W kodzie przedstawionym na listingach we wcześniejszej części rozdziału skoncentrowałem się na klasach, ponieważ w taki właśnie sposób większość programistów tworzy typy danych. Obiekt JavaScriptu może zostać zdefiniowany za pomocą składni literału, a TypeScript ma możliwość sprawdzenia typu i wymuszenia typu statycznego, podobnie jak w przypadku tworzenia obiektu na podstawie klasy. Gdy pracujesz z literałami, kompilator TypeScriptu koncentruje się na połączeniu nazw właściwości i typów ich wartości, co jest określane mianem *kształtu* obiektu. Konkretnie połączenie nazw i typów nosi nazwę *kształtu typu*. Na listingu 1.25 pokazałem dodanie do klasy `TodoCollection` metody zwracającej obiekt opisujący liczbę elementów znajdujących się w kolekcji.

Listing 1.25. Pozwalająca na używanie kształtu typu modyfikacja kodu w pliku `todoCollection.ts` znajdującym się w katalogu `src`

```
import { TodoItem } from "./todoItem";

type ItemCounts = {
  total: number,
  incomplete: number
}

export class TodoCollection {
  private nextId: number = 1;
  private itemMap = new Map<number, TodoItem>();

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
  }

  addTodo(task: string): number {
    while (this.getTodoById(this.nextId)) {
      this.nextId++;
    }
    this.itemMap.set(this.nextId, new TodoItem(this.nextId, task));
    return this.nextId;
  }

  getTodoById(id: number) : TodoItem {
    return this.itemMap.get(id);
  }

  getTodoItems(includeComplete: boolean): TodoItem[] {
    return [...this.itemMap.values()]
      .filter(item => includeComplete || !item.complete);
  }

  markComplete(id: number, complete: boolean) {
    const todoItem = this.getTodoById(id);
    if (todoItem) {
      todoItem.complete = complete;
    }
  }

  removeComplete() {
    this.itemMap.forEach(item => {
      if (item.complete) {
        this.itemMap.delete(item.id);
      }
    })
  }

  getItemCounts(): ItemCounts {
    return {
      total: this.itemMap.size,
      incomplete: this.getTodoItems(false).length
    };
  }
}
```

Słowo kluczowe `type` zostało użyte do utworzenia tzw. *aliasu typu*, czyli wygodnego rozwiązania pozwalającego na przypisanie nazwy kształtowi typu. Alias typu na listingu 1.25 opisuje obiekty zawierające dwie właściwości typu `number`: o nazwach `total` i `incomplete`. Alias typu jest używany jako wynik działania metody `getItemCounts()`, która z kolei wykorzystuje składnię literału obiektu JavaScriptu do utworzenia obiektu o kształcie typu dopasowanym do aliasu typu. Na listingu 1.26 przedstawiłem zmodyfikowaną wersję pliku *index.ts*, która teraz wyświetla użytkownikowi liczbę jeszcze niewykonanych zadań.

Listing 1.26. Modyfikacja pliku *index.ts* w katalogu *src* pozwalająca na wyświetlanie liczby elementów na liście rzeczy do zrobienia

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

console.clear();
console.log(`Lista ${collection.userName}a
  + `(liczba zadań pozostałych do zrobienia: ${ collection.getItemCounts().incomplete
  })`);
collection.getTodoItems(true).forEach(item => item.printDetails());
```

W katalogu *todo* wydaj polecenia przedstawione na listingu 1.27, aby w ten sposób skompilować i uruchomić kod.

Listing 1.27. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Oto dane wyjściowe po wykonaniu przedstawionego kodu:

```
Lista Adama (liczba zadań pozostałych do zrobienia: 3)
1      Kupić kwiaty
2      Odebrać buty
3      Zamówić bilety
4      Zadzwoń do Janka      (wykonane)
```

Używanie pakietu zewnętrznego

Jedną z zalet tworzenia kodu JavaScriptu jest istnienie ekosystemu pakietów możliwych do wykorzystania w projektach. TypeScript pozwala na używanie pakietów JavaScriptu, ale po zapewnieniu obsługi typowania statycznego. W tym podrozdziale zamierzam skorzystać z doskonałego pakietu *Inquirer.js* (<https://github.com/SBoudrias/Inquirer.js>) przeznaczonego do pobierania informacji od użytkownika i przetwarzania odpowiedzi. Aby dodać ten pakiet do projektu, przejdź do katalogu *todo* i wydaj polecenie przedstawione na listingu 1.28.

Listing 1.28. Dodawanie pakietu do projektu

```
$ npm install inquirer@6.3.1
```

Dodawanie pakietów do projektów TypeScriptu odbywa się w dokładnie taki sam sposób, jak w przypadku zwykłego kodu JavaScriptu, czyli za pomocą polecenia `npm install`. Na listingu 1.29 przedstawiłem zmodyfikowaną wersję kodu w pliku `index.ts`, który teraz korzysta z nowo dodanego pakietu.

Listing 1.29. Zmodyfikowana wersja kodu znajdującego się w pliku index.ts w katalogu src i wykorzystującego nowy pakiet

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
import * as inquirer from 'inquirer';

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwonić do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);

function displayTodoList(): void {
  console.log(`Lista ${collection.userName}a`
    + `(liczba zadań pozostałych do zrobienia: ${
collection.getItemCounts().incomplete })`);
  collection.getTodoItems(true).forEach(item => item.printDetails());
}

enum Commands {
  Quit = "Koniec"
}

function promptUser(): void {
  console.clear();
  displayTodoList();
  inquirer.prompt({
    type: "list",
    name: "command",
    message: "Wybierz opcję",
    choices: Object.values(Commands)
  }).then(answers => {
    if (answers["command"] !== Commands.Quit) {
      promptUser();
    }
  })
}

promptUser();
```

TypeScript nie przeszkadza w korzystaniu z kodu JavaScriptu, a zmiany wprowadzone na listingu 1.29 pozwalają na używanie pakietu `Inquirer.js` do zaproponowania użytkownikowi zestawu dostępnych poleceń. Obecnie obsługiwane jest tylko jedno polecenie, *Koniec*, przy czym obsługa kolejnych zostanie wkrótce dodana.

- **Wskazówka** W książce nie zamierzam szczegółowo omawiać API pakietu `Inquirer.js`, ponieważ nie ma on bezpośredniego związku z językiem TypeScript. Jeżeli chcesz z tego pakietu korzystać we własnych projektach, więcej informacji na jego temat znajdziesz na stronie <https://github.com/SBoudrias/Inquirer.js>.

Metoda `inquirer.prompt()` jest używana do pobierania danych od użytkownika i została skonfigurowana do pracy z obiektem JavaScriptu. Wybrane przeze mnie opcje konfiguracyjne zostaną przedstawione użytkownikowi w postaci listy, po której może się poruszać za pomocą klawiszy kursora, a wybór opcji następuje po naciśnięciu klawisza *Enter*. Gdy użytkownik dokona wyboru, wywołana zostanie funkcja przekazana metodzie `then()`, a nazwa tej funkcji będzie dostępna za pomocą właściwości `answers.command`.

Na listingu 1.29 pokazałem, jak używać kodu TypeScriptu i JavaScriptu pakietu `Inquirer.js`. Słowo kluczowe `enum` to funkcjonalność oferowana przez TypeScript i pozwalająca na nadawanie nazw wartościom, jak to dokładnie omówię w rozdziale 9. W tym przypadku `enum` pozwala zdefiniować polecenia i odwoływać się do nich bez konieczności powielania w aplikacji wartości w postaci ciągów tekstowych. Wartości pochodzące z konstrukcji `enum` są używane razem z funkcjami `Inquirer.js`, np.:

```
...
if (answers["command"] !== Commands.Quit) {
...

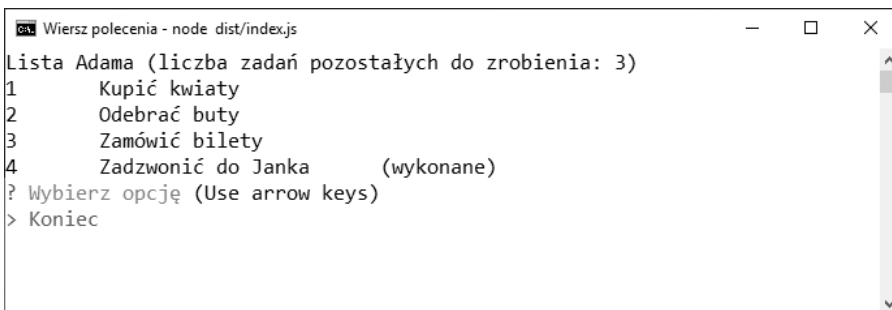
```

W katalogu `todo` wydaj polecenia przedstawione na listingu 1.30, aby w ten sposób skompilować i uruchomić kod.

Listing 1.30. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Po wykonaniu tego kodu zostanie wyświetlona lista rzeczy do zrobienia, a także lista dostępnych opcji, jak pokazałem na rysunku 1.2, choć w tym momencie to tylko jedna opcja.



Rysunek 1.2. Wyświetlenie użytkownikowi dostępnych opcji

Jeżeli naciśniesz klawisz *Enter*, zostanie wybrana opcja *Koniec* i program zakończy działanie.

Dodawanie deklaracji typu dla pakietu JavaScriptu

TypeScript pozwala na używanie kodu JavaScriptu, choć jednocześnie nie zapewnia żadnej pomocy w tym zakresie. Kompilator nie ma informacji o typach danych używanych przez pakiet `Inquirer.js` i musi polegać na tym, że podczas wyświetlenia opcji użytkownikowi zostały zastosowane odpowiednie typy argumentów, a otrzymane w odpowiedzi obiekty są przetwarzane w sposób bezpieczny.

Istnieją dwa sposoby na dostarczenie językowi TypeScript informacji wymaganych przez ten język w celu zastosowania statycznego typowania. Pierwszy polega na samodzielnym opisywaniu typów. Dostarczane przez TypeScript funkcje przeznaczone do opisywania kodu JavaScriptu przedstawiamy w rozdziale 14. Samodzielne opisywanie kodu JavaScriptu nie jest trudne, choć wymaga nieco czasu i dobrej znajomości opisywanego kodu.

Drugi sposób polega na wykorzystaniu przygotowanych przez kogoś innego definicji typów. Projekt `Definitely Typed` to repozytorium deklaracji typu JavaScriptu dla tysięcy pakietów JavaScriptu, w tym także dla `Inquirer.js`. Aby zainstalować deklaracje typu, należy w katalogu `todo` wydać polecenie przedstawione na listingu 1.31.

Listing 1.31. Instalowanie deklaracji typów

```
$ npm install --save-dev @types/inquirer
```

Deklaracje typów są instalowane za pomocą polecenia `install`, podobnie jak pakiety JavaScriptu. Argument `save-dev` jest stosowany w przypadku pakietów używanych podczas pracy nad aplikacją, ale niebędących jej częścią. Nazwa pakietu rozpoczyna się prefiksem `@types/`, a następnie znajduje się nazwa pakietu, dla którego są wymagane deklaracje typów. W przypadku pakietu `Inquirer.js` pakietem deklaracji typów jest `@types/inquirer`, ponieważ `inquirer` to nazwa użyta do zainstalowania pakietu JavaScriptu.

-
- **Uwaga** Więcej informacji o projekcie `Definitely Typed` oraz o dostępnych deklaracjach typów znajdziesz na stronie <https://github.com/DefinitelyTyped/DefinitelyTyped>.
-

Kompilator TypeScriptu automatycznie wykrywa deklaracje typów, więc polecenie przedstawione na listingu 1.31 pozwala kompilatorowi na sprawdzanie typów danych używanych przez API `Inquirer.js`. Aby pokazać efekt zastosowania deklaracji typów, na listingu 1.32 przedstawiłem konfigurację właściwości nieobsługiwanej przez pakiet `Inquirer.js`.

Listing 1.32. Dodawanie właściwości w pliku `index.ts` znajdującym się w katalogu `src`

```
...
function promptUser(): void {
  console.clear();
  inquirer.prompt({
    type: "list",
    name: "command",
    message: "Wybierz opcję",
    choices: Object.values(Commands),
    badProperty: true
  }).then(answers => {
    // Nie jest wymagane żadne działanie.
  });
}
```



```

        if (answers["command"] !== Commands.Quit) {
            promptUser();
        }
    })
}
...

```

API `Inquirer.js` nie zawiera właściwości konfiguracyjnej o nazwie `badProperty`. W katalogu `todo` wydaj polecenie przedstawione na listingu 1.33, aby w ten sposób skompilować kod.

Listing 1.33. Kompilowanie kodu przykładowej aplikacji

```
$ tsc
```

Kompilator wykorzysta informacje o typach zainstalowane po wykonaniu polecenia przedstawionego na listingu 1.31 i wygeneruje następujący komunikat błędu:

```

src/index.ts:30:13 - error TS2345: Argument of type '{ type: string; name: string;
message: string; choices: any[]; badProperty: boolean; }' is not assignable to parameter
of type 'Questions<{}>'.
  Object literal may only specify known properties, and 'badProperty' does not exist in type
'Questions<{}>'.
30           badProperty: true
                ~~~~~
Found 1 error.

```

Deklaracja typu pozwala TypeScriptowi na dostarczanie tego samego zestawu funkcjonalności w całej aplikacji, nawet mimo że pakiet `Inquirer.js` został utworzony w czystym kodzie JavaScriptu, a nie TypeScriptu.

Dodawanie poleceń

Przykładowa aplikacja na razie nie działa zbyt dobrze i wymaga zdefiniowania kolejnych poleceń. W tym podrozdziale zajmę się przygotowaniem nowych poleceń i dostarczeniem ich implementacji.

Filtrowanie elementów

Pierwsze dodawane polecenie pozwoli użytkownikowi na włączenie lub wyłączenie filtrowania wykonanych zadań. Modyfikacje konieczne do wprowadzenia w kodzie przedstawiłem na listingu 1.34.

Listing 1.34. Implementacja w kodzie pliku `index.ts` w katalogu `src` operacji filtrowania elementów listy rzeczy do zrobienia

```

import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
import * as inquirer from 'inquirer';

let todos: TodoItem[] = [
    new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
    new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwońić do Janka", true)];

```

```

let collection: TodoCollection = new TodoCollection("Adam", todos);
let showCompleted = true;

function displayTodoList(): void {
    console.log(`Lista ${collection.userName}a `
        + `(liczba zadań pozostałych do zrobienia: ${
collection.getItemCounts().incomplete })`);
    collection.getTodoItems(showCompleted).forEach(item => item.printDetails());
}

enum Commands {
    Toggle = "Pokaż lub ukryj wykonane",
    Quit = "Koniec"
}

function promptUser(): void {
    console.clear();
    displayTodoList();
    inquirer.prompt({
        type: "list",
        name: "command",
        message: "Wybierz opcję",
        choices: Object.values(Commands),
        //badProperty: true
    }).then(answers => {
        switch (answers["command"]) {
            case Commands.Toggle:
                showCompleted = !showCompleted;
                promptUser();
                break;
        }
    })
}
promptUser();

```

Proces dodawania opcji sprowadza się do zdefiniowania nowej wartości dla wyliczenia `Commands` i określenia poleceń wykonywanych po wybraniu danej opcji. W omawianym przykładzie nową wartością jest `Toggle`, a po jej wybraniu wartość zmiennej `showCompleted` zostanie zmieniona, aby funkcja `displayTodoList()` uwzględniła (lub nie) wykonane zadania. W katalogu *todo* wydaj polecenia przedstawione na listingu 1.35, aby w ten sposób skompilować i uruchomić kod.

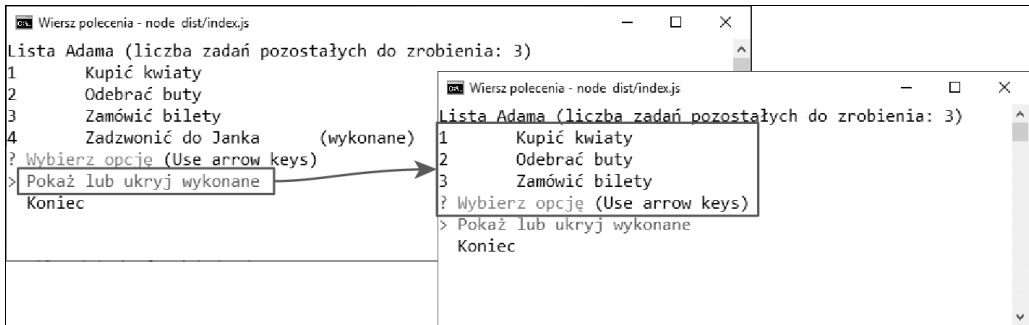
Listing 1.35. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```

$ tsc
$ node dist/index.js

```

Wybierz opcję *Pokaż lub ukryj wykonane* i naciśnij klawisz *Enter*, aby w ten sposób włączyć lub wyłączyć wyświetlanie na liście wykonanych zadań, jak pokazałem na rysunku 1.3.



Rysunek 1.3. Pokazywanie lub ukrywanie wykonanych zadań

Dodawanie zadań

Przykładowa aplikacja nie będzie za bardzo użyteczna, jeśli użytkownik nie otrzyma możliwości tworzenia nowych elementów na liście. Na listingu 1.36 przedstawiłem zmiany w kodzie pozwalające na tworzenie nowych obiektów typu `TodoItem`.

Listing 1.36. Zmiany w kodzie pliku `index.ts` w katalogu `src` pozwalające na dodawanie nowych zadań na liście

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
import * as inquirer from 'inquirer';

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwoń do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);
let showCompleted = true;

function displayTodoList(): void {
  console.log(`Lista ${collection.userName}a `
    + `~(liczba zadań pozostałych do zrobienia: ${
collection.getItemCounts().incomplete })~`);
  collection.getTodoItems(showCompleted).forEach(item => item.printDetails());
}

enum Commands {
  Add = "Dodaj nowe zadanie",
  Toggle = "Pokaż lub ukryj wykonane",
  Quit = "Koniec"
}

function promptAdd(): void {
  console.clear();
  inquirer.prompt({ type: "input", name: "add", message: "Podaj zadanie:"})
    .then(answers => {if (answers["add"] !== "") {
      collection.addToDo(answers["add"]);
    }
  })
}
```

```

        promptUser();
    })
}

function promptUser(): void {
    console.clear();
    displayTodoList();
    inquirer.prompt({
        type: "list",
        name: "command",
        message: "Wybierz opcję",
        choices: Object.values(Commands),
    }).then(answers => {
        switch (answers["command"]) {
            case Commands.Toggle:
                showCompleted = !showCompleted;
                promptUser();
                break;
            case Commands.Add:
                promptAdd();
                break;
        }
    })
}
promptUser();

```

Pakiet Inquirer.js może wyświetlać użytkownikowi różne rodzaje pytań. Gdy użytkownik wybierze polecenie Add, wówczas typ input będzie użyty w celu pobrania zadania, które następnie stanie się argumentem wywołania metody `TodoCollection.addTo()`. W katalogu *todo* wydaj polecenia przedstawione na listingu 1.37, aby w ten sposób skompilować i uruchomić kod.

Listing 1.37. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```

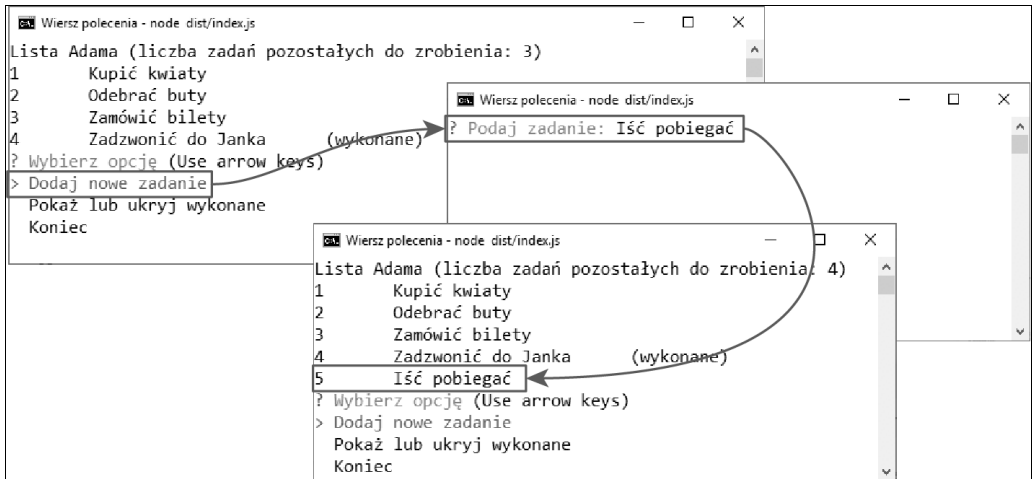
$ tsc
$ node dist/index.js

```

Wybierz teraz opcję *Dodaj nowe zadanie*, wpisz treść zadania i naciśnij klawisz *Enter*, aby dodać nowe zadanie, jak pokazałem na rysunku 1.4.

Oznaczanie zadania jako wykonanego

Wykonanie zadania to proces składający się z dwóch etapów i wymagający od użytkownika wskazania elementu listy, który ma być oznaczony jako wykonany. Na listingu 1.38 przedstawiłem zdefiniowanie dodatkowego polecenia i pytania, co pozwoli użytkownikowi na oznaczenie zadania jako wykonanego oraz na usuwanie już wykonanych zadań.



Rysunek 1.4. Dodawanie nowego zadania na liście rzeczy do zrobienia

Listing 1.38. Modyfikacje w kodzie pliku `index.ts` w katalogu `src` pozwalające na oznaczanie zadania jako już wykonanego

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
import * as inquirer from 'inquirer';

let todos: TodoItem[] = [
    new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
    new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwońić do Janka", true)];

let collection: TodoCollection = new TodoCollection("Adam", todos);
let showCompleted = true;

function displayTodoList(): void {
    console.log(`Lista ${collection.userName}a `
        + `(liczba zadań pozostałych do zrobienia: ${
collection.getItemCounts().incomplete })`);
    collection.getTodoItems(showCompleted).forEach(item => item.printDetails());
}

enum Commands {
    Add = "Dodaj nowe zadanie",
    Complete = "Wykonanie zadania",
    Toggle = "Pokaż lub ukryj wykonane",
    Purge = "Usuń wykonane zadania",
    Quit = "Koniec"
}

function promptAdd(): void {
    console.clear();
    inquirer.prompt({ type: "input", name: "add", message: "Podaj zadanie:"})
        .then(answers => {if (answers["add"] !== "") {
            collection.addToDo(answers["add"]);
        }
    });
}
```

```

    }
    promptUser();
  })
}

function promptComplete(): void {
  console.clear();
  inquirer.prompt({ type: "checkbox", name: "complete",
    message: "Oznaczenie zadań jako wykonanych",
    choices: collection.getTodoItems(showCompleted).map(item =>
      ({name: item.task, value: item.id, checked: item.complete}))
  }).then(answers => {
    let completedTasks = answers["complete"] as number[];
    collection.getTodoItems(true).forEach(item =>
      collection.markComplete(item.id,
        completedTasks.find(id => id === item.id) != undefined));
    promptUser();
  })
}

function promptUser(): void {
  console.clear();
  displayTodoList();
  inquirer.prompt({
    type: "list",
    name: "command",
    message: "Wybierz opcje",
    choices: Object.values(Commands),
  }).then(answers => {
    switch (answers["command"]) {
      case Commands.Toggle:
        showCompleted = !showCompleted;
        promptUser();
        break;
      case Commands.Add:
        promptAdd();
        break;
      case Commands.Complete:
        if (collection.getItemCounts().incomplete > 0) {
          promptComplete();
        } else {
          promptUser();
        }
        break;
      case Commands.Purge:
        collection.removeComplete();
        promptUser();
        break;
    }
  })
}
promptUser();

```

Wprowadzone zmiany powodują dodanie nowego polecenia wyświetlanego użytkownikowi z listą zadań i pozwalającego na zmianę ich stanu. Zmienna `showCompleted` jest używana do ustalenia, czy wykonane zadania mają się pojawiać na liście, oraz tworzy połączenie między poleceniami `Toggle` i `Complete`.

Jedyna nowa funkcjonalność TypeScriptu znalazła się w następującym poleceniu:

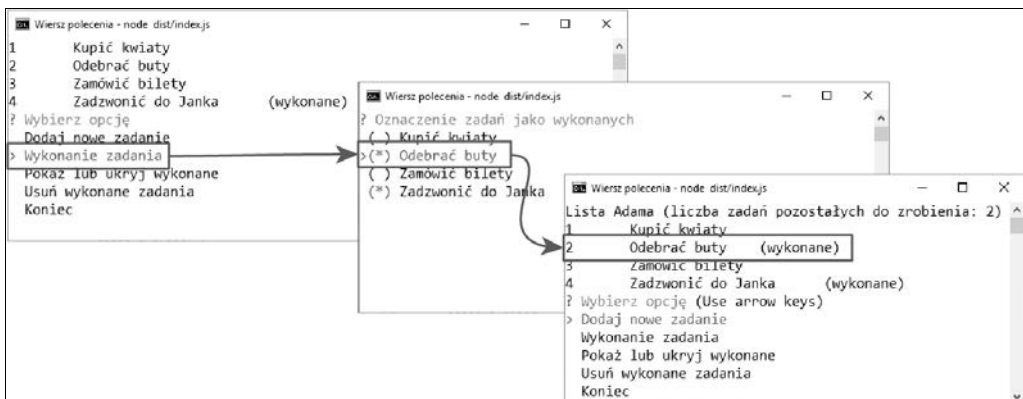
```
...
let completedTasks = answers["complete"] as number[];
...
```

Nawet pomimo istnienia definicji typów zdarzają się sytuacje, w których TypeScript nie potrafi prawidłowo ustalić używanego typu. W takim przypadku pakiet `Inquirer.js` pozwala na użycie dowolnego typu danych w pytaniu wyświetlonym użytkownikowi, a kompilator nie będzie w stanie ustalić, że wykorzystane są tylko wartości typu `number`, co oznacza możliwość otrzymania odpowiedzi jedynie w postaci wartości typu `number`. Do rozwiązania tego problemu wykorzystałem *asercję typu*, która pozwala wskazać kompilatorowi, że powinien używać podanego typu, nawet jeśli zidentyfikował inny typ danych (lub nie ustalił żadnego). Podczas stosowania asercji zachowania kompilatora są nadpisywane, co oznacza, że programista staje się odpowiedzialny za zagwarantowanie użycia prawidłowego typu. W katalogu `todo` wydaj polecenia przedstawione na listingu 1.39, aby w ten sposób skompilować i uruchomić kod.

Listing 1.39. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Wybierz opcję *Wykonanie zadania*, zaznacz jedno lub więcej zadań, których stan ma się zmienić, i naciśnij klawisz `Enter`. Stan wybranych zadań zostanie zmieniony, co będzie odzwierciedlone na liście, jak pokazałem na rysunku 1.5.



Rysunek 1.5. Oznaczenie zadania jako wykonanego

Trwałe przechowywanie danych

Aby zapewnić możliwość trwałego przechowywania danych, zamierzam skorzystać z kolejnego pakietu typu *open source*, ponieważ nie ma żadnego sensu tworzenie funkcjonalności, gdy dostępne są już gotowe i doskonale przetestowane alternatywy. W katalogu *todo* wydaj polecenia przedstawione na listingu 1.40, aby w ten sposób zainstalować pakiet Lowdb i definicje typów dostarczające TypeScriptowi opis API tego pakietu.

Listing 1.40. Dodawanie pakietu i definicji typu

```
$ npm install lowdb@1.0.0
$ npm install --save-dev @types/lowdb
```

Lowdb to doskonały pakiet bazy danych przechowujący dane w pliku JSON i używany jako komponent magazynu danych w pakiecie *json-server*, który w trzeciej części książki będzie wykorzystany podczas tworzenia usług sieciowych HTTP.

-
- **Wskazówka** W książce nie zamierzam szczegółowo omawiać API pakietu Lowdb, ponieważ nie ma on bezpośredniego związku z językiem TypeScript. Jeżeli chcesz z tego pakietu korzystać we własnych projektach, więcej informacji na jego temat znajdziesz na stronie <https://github.com/typicode/lowdb>.
-

Implementacja trwałego magazynu danych będzie się odbywała poprzez klasę `TodoCollection`. Na etapie przygotowań trzeba zmienić używane przez tę klasę słowo kluczowe kontroli dostępu, aby jej podklasy mogły uzyskiwać dostęp do kolekcji `Map` zawierającej obiekty `TodoItem`. Zmianę konieczną do wprowadzenia przedstawiłem na listingu 1.41.

Listing 1.41. Zmiana ustawień kontroli dostępu w pliku `todoCollection.ts` znajdującym się w katalogu `src`

```
import { TodoItem } from "./todoItem";

type ItemCounts = {
  total: number,
  incomplete: number
}

export class TodoCollection {
  private nextId: number = 1;
  protected itemMap = new Map<number, TodoItem>();

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    todoItems.forEach(item => this.itemMap.set(item.id, item));
  }

  // Pozostałe metody zostały pominięte.
}
```


Słowo kluczowe `protected` wskazuje kompilatorowi TypeScriptu, że właściwość jest dostępna jedynie dla klasy i jej podklas. W celu utworzenia podklasy należy dodać do katalogu `src` plik o nazwie `jsonTodoCollection.ts` z kodem przedstawionym na listingu 1.42.

Listing 1.42. Zawartość pliku `jsonTodoCollection.ts` w katalogu `src`

```
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
import * as lowdb from "lowdb";
import * as FileSync from "lowdb/adapters/FileSync";

type schemaType = {
  tasks: { id: number; task: string; complete: boolean; }[]
};

export class JsonTodoCollection extends TodoCollection {
  private database: lowdb.LowdbSync<schemaType>;

  constructor(public userName: string, todoItems: TodoItem[] = []) {
    super(userName, []);
    this.database = lowdb(new FileSync("Todos.json"));
    if (this.database.has("tasks").value()) {
      let dbItems = this.database.get("tasks").value();
      dbItems.forEach(item => this.itemMap.set(item.id,
        new TodoItem(item.id, item.task, item.complete)));
    } else {
      this.database.set("tasks", todoItems).write();
      todoItems.forEach(item => this.itemMap.set(item.id, item));
    }
  }

  addTodo(task: string): number {
    let result = super.addTodo(task);
    this.storeTasks();
    return result;
  }

  markComplete(id: number, complete: boolean): void {
    super.markComplete(id, complete);
    this.storeTasks();
  }

  removeComplete(): void {
    super.removeComplete();
    this.storeTasks();
  }

  private storeTasks() {
    this.database.set("tasks", [...this.itemMap.values()]).write();
  }
}
```

Definicje typu dla pakietu `Lowdb` używają schematu do opisanego struktury przechowywanych danych. Ten schemat jest stosowany za pomocą argumentów typu generycznego, aby kompilator TypeScriptu mógł sprawdzić używane typy danych. W omawianej tutaj aplikacji trzeba przechowywać dane tylko jednego typu, który zostanie opisany za pomocą aliasu typu.

```
...
type schemaType = {
  tasks: { id: number; task: string; complete: boolean; }[]
};
...
```

Typ schematu jest stosowany podczas tworzenia bazy danych Lowdb, a kompilator ma możliwość sprawdzenia sposobu używania danych w trakcie ich odczytywania z bazy danych, jak w przedstawionym tutaj poleceniu:

```
...
let dbItems = this.database.get("tasks").value();
...
```

Kompilator wie, że argument `tasks` odpowiada właściwości `tasks` w typie schematu, więc wynikiem operacji `get()` jest tablica obiektów z właściwościami `id`, `task` i `complete`.

Stosowanie klasy trwałego magazynu danych

Na listingu 1.43 przedstawiłem przykład użycia klasy `JsonTodoCollection` w pliku `index.ts`, aby umożliwić przykładowej aplikacji trwałe przechowywanie danych.

Listing 1.43. Używanie trwałej kolekcji danych w pliku `index.ts` w katalogu `src`

```
...
import { TodoItem } from "./todoItem";
import { TodoCollection } from "./todoCollection";
import * as inquirer from 'inquirer';
import { JsonTodoCollection } from "./jsonTodoCollection";

let todos: TodoItem[] = [
  new TodoItem(1, "Kupić kwiaty"), new TodoItem(2, "Odebrać buty"),
  new TodoItem(3, "Zamówić bilety"), new TodoItem(4, "Zadzwonić do Janka", true)];

let collection: TodoCollection = new JsonTodoCollection("Adam", todos);
let showCompleted = true;
...
```

W katalogu `todo` wydaj polecenia przedstawione na listingu 1.44, aby w ten sposób po raz ostatni w rozdziale skompilować i uruchomić kod.

Listing 1.44. Kompilowanie i wykonywanie kodu przykładowej aplikacji

```
$ tsc
$ node dist/index.js
```

Po uruchomieniu aplikacji w katalogu `todo` zostanie utworzony plik o nazwie `Todos.json`, który będzie zawierał reprezentację JSON obiektów `TodoItem`. Dzięki temu zmiany wprowadzone w aplikacji nie zostaną utracone po zakończeniu jej działania.

Podsumowanie

W tym rozdziale utworzyłem prostą aplikację, aby przedstawić wprowadzenie do programowania w języku TypeScript oraz zaprezentować kilka najważniejszych koncepcji tego języka. Przekonałeś się, że TypeScript dostarcza funkcje uzupełniające język JavaScript, skoncentrowane na zapewnieniu bezpieczeństwa typu i pomagające w unikaniu błędów najczęściej popełnianych przez programistów, zwłaszcza tych, którzy wcześniej mieli doświadczenie w pracy z językami takimi jak C# i Java.

Dowiedziałeś się, że TypeScript nie jest używany oddzielnie, a środowisko uruchomieniowe JavaScriptu jest niezbędne do wykonania kodu JavaScriptu wygenerowanego przez kompilator TypeScriptu. Zaletą przedstawionego podejścia jest to, że projekty utworzone za pomocą TypeScriptu mają pełny dostęp do szerokiej gamy dostępnych pakietów JavaScriptu, z których wiele zawiera definicje typów ułatwiające wykorzystanie tych pakietów w języku TypeScript.

Aplikacja utworzona w rozdziale stosuje kilka najważniejszych funkcji TypeScriptu. Jak możesz się domyślać na podstawie wielkości książki, tych funkcji jest znacznie więcej. W następnym rozdziale dowiesz się nieco więcej na temat powiązania języków TypeScript i JavaScript oraz poznasz strukturę i treść książki.

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>

GRUPA
Helion 

TypeScript: programuj jak zawodowiec i twórz bogate aplikacje!

JavaScript dojrzał i stał się pełnowartościowym językiem programowania. Jest wszechstronny, elastyczny i pozwala na tworzenie znakomitego kodu, jednak uzyskiwanie naprawdę dobrych efektów wymaga sporych umiejętności. Z tego powodu warto zainteresować się TypeScriptem, który w porównaniu z JavaScriptem o wiele lepiej spisuje się jako język programowania profesjonalnych aplikacji internetowych. Ich projektant, programujący w TypeScriptie, może przy tym łatwo skorzystać z wielu popularnych frameworków. W ten stosunkowo prosty sposób w pełni wykorzystuje możliwości nowoczesnych przeglądarek i urządzeń mobilnych.

Ta książka jest przystępnym podręcznikiem, dzięki któremu, poza uzyskaniem ważnych umiejętności, odkryjesz najcenniejsze aspekty TypeScriptu. Rozpoczniesz od zdobycia solidnych podstaw, a po przeanalizowaniu przejrzystych przykładów poznasz korzyści wynikające z używania TypeScriptu. Stopniowo nauczysz się stosować w praktyce najbardziej zaawansowane funkcje. Dowiesz się, jak stworzyć bezpieczniejsze i bardziej produktywnie środowisko do tworzenia aplikacji internetowych, a także poznasz kilka popularnych frameworków, takich jak Node.js, Angular, React i Vue.js. Znajdziesz tu również informacje o najczęściej występujących problemach oraz sposobach ich rozwiązywania.

W książce między innymi:

- przygotowanie środowiska pracy i potrzebne narzędzia
- solidne podstawy TypeScriptu
- tworzenie kodu TypeScriptu działającego po stronie klienta i po stronie serwera
- rozbudowa i modyfikowanie aplikacji napisanych w TypeScriptie
- testowanie, debugowanie i wdrażanie kodu

Adam Freeman — jest doświadczonym programistą. Napisał wiele świetnie przyjętych książek o programowaniu. Tworzył również duże systemy rozproszone (platformy e-commerce). Zajmował stanowiska kierownicze w wielu firmach, wśród których są Netscape, Sun Microsystems, giełda NASDAQ i banki o międzynarodowym zasięgu. Dziś jest na emeryturze, swój czas przeznacza na pisanie i bieganie na długich dystansach.

 Helion	<i>Sprawdź nasze szkolenia!</i> SZKOLENIA  AKADEMIA IT & BUSINESS HELIONSZKOLENIA.PL	KOD KORZYŚCI <i>Sięgnij po więcej!</i>   ISBN 978-83-283-6531-5  9 788328 365315
 helion.pl		
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl		
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 89,00 zł

Apress®