

Mark J. Price

C# 7.1 i .NET Core 2.0 dla programistów aplikacji wieloplatformowych

Helion 

Packt 

Tytuł oryginału: C# 7.1 and .NET Core 2.0 - Modern Cross-Platform Development

Tłumaczenie: Wojciech Moch

ISBN: 978-83-283-4450-1

Copyright © Packt Publishing 2017. First published in the English language under the title 'C# 7.1 and .NET Core 2.0 – Modern Cross-Platform Development – (9781788398077)'

Polish edition copyright © 2018 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 Wydawnictwo HELION 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/c7Inc2.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/c7Inc2>
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	18
O recenzentach	20
Wstęp	21
Rozdział 1. Cześć, C#! Witaj, .NET Core	27
Konfigurowanie środowiska programistycznego	28
Używanie innych IDE dla języka C#	29
Instalowanie na wielu platformach	30
Instalowanie Microsoft Visual Studio 2017	30
Instalowanie Microsoft Visual Studio Code	33
Instalowanie Visual Studio for Mac	37
Poznanie .NET	38
Poznanie .NET Framework	39
Poznanie projektów Mono i Xamarin	39
Poznanie .NET Core	40
Poznanie .NET Standard	42
Poznanie .NET Native	43
Porównanie technologii .NET	43
Pisanie i kompilowanie kodu przy użyciu narzędzi wiersza poleceń z .NET Core	44
Pisanie kodu za pomocą prostego edytora tekstu	44
Tworzenie i kompilowanie aplikacji za pomocą narzędzi wiersza poleceń .NET Core	46
Naprawianie błędów kompilacji	48
Poznanie języka pośredniego	49
Pisanie i kompilowanie kodu za pomocą Visual Studio 2017	49
Pisanie kodu w Visual Studio 2017	50
Kompilowanie kodu za pomocą Visual Studio 2017	53

Poprawianie pomyłek z listy błędów	54
Dodawanie istniejących projektów do Visual Studio 2017	55
Automatyczne formatowanie kodu	56
Eksperymentowanie z interaktywnym C#	57
Inne przydatne okna	59
Pisanie i kompilowanie kodu w Visual Studio Code	60
Pisanie kodu w Visual Studio Code	60
Kompilowanie kodu w Visual Studio Code	61
Automatyczne formatowanie kodu	62
Pisanie i kompilowanie kodu za pomocą Visual Studio for Mac	62
Następne kroki	65
Zarządzanie kodem źródłowym przy użyciu platformy GitHub	65
Używanie systemu Git w Visual Studio 2017	66
Używanie systemu Git w Visual Studio Code	68
Praktyka i ćwiczenia	69
Ćwiczenie 1.1 — sprawdź swoją wiedzę	69
Ćwiczenie 1.2 — ćwicz C# gdzie się da	70
Ćwiczenie 1.3 — dalsza lektura	70
Podsumowanie	70

Część I. C# 7.1 71

Rozdział 2. Mówimy w C# 73

Poznananie podstaw języka C#	73
Używanie Visual Studio 2017	74
Używanie Visual Studio Code w systemach macOS, Linux i Windows	76
Gramatyka języka C#	78
Słownictwo języka C#	79
Pomoc przy pisaniu kodu	81
Czasowniki jako metody	82
Rzeczowniki to typy, pola i zmienne	83
Ujawnienie wielkości słownika języka C#	84
Deklarowanie zmiennych	86
Nazywanie zmiennych	87
Literały	87
Przechowywanie tekstu	88
Przechowywanie liczb	88
Przechowywanie wartości logicznych	93
Typ object	93
Typ dynamic	94
Zmienne lokalne	94
Zapisywanie wartości null w typach wartości	96
Poznananie typów referencyjnych z wartością null	96
Zapisywanie wielu wartości w tablicy	98

Dokładniejsze poznawanie aplikacji konsoli	99
Wyświetlanie informacji dla użytkownika	99
Pobieranie danych od użytkownika	100
Importowanie przestrzeni nazw	100
Uprozczone korzystanie z konsoli	101
Odczytywanie parametrów i praca z tablicami	102
Działania na zmiennych	106
Eksperymenty z operatorami jednoargumentowymi	107
Eksperymentowanie z operatorami arytmetycznymi	108
Porównania i operatory logiczne	109
Praktyka i ćwiczenia	109
Ćwiczenie 2.1 — sprawdź swoją wiedzę	109
Ćwiczenie 2.2 — poznaj wielkości i zakresy liczb	109
Ćwiczenie 2.3 — dalsza lektura	110
Podsumowanie	111
Rozdział 3. Sterowanie przepływem i konwertowanie typów	113
<hr/>	
Instrukcje wyboru	113
Visual Studio 2017	113
Visual Studio Code w systemach macOS, Linux i Windows	114
Instrukcja if	114
Instrukcja switch	115
Instrukcje iteracji	118
Instrukcja while	118
Instrukcja do	119
Instrukcja for	119
Instrukcja foreach	120
Rzutowanie i konwertowanie między typami	121
Rzutowanie z liczby na liczbę	121
Używanie typu Convert	123
Zaokrąglanie liczb	123
Konwersja z dowolnego typu na ciąg znaków	124
Konwertowanie obiektu binarnego na ciąg znaków	124
Parsowanie ciągów znaków z liczbami, datami i czasem	126
Obsługa wyjątków podczas konwertowania typów	127
Instrukcja try	127
Przechwytywanie wszystkich wyjątków	128
Przechwytywanie wybranych wyjątków	129
Wykrywanie przepiętni	130
Instrukcja checked	130
Instrukcja unchecked	131
Gdzie znaleźć pomoc	132
Microsoft Docs i MSDN	133
Przejdź do definicji	133
Stack Overflow	134
Google	134

Subskrybowanie blogów	136
Wzorce projektowe	136
Praktyka i ćwiczenia	138
Ćwiczenie 3.1 — sprawdź swoją wiedzę	138
Ćwiczenie 3.2 — pętle i przepełnienia	138
Ćwiczenie 3.3 — pętle i operatory	139
Ćwiczenie 3.4 — obsługa wyjątków	139
Ćwiczenie 3.5 — dalsza lektura	140
Podsumowanie	140
Rozdział 4. Pisanie, debugowanie i testowanie funkcji	141
Tworzenie funkcji	141
Tworzenie funkcji wypisującej sekwencję mnożenia	142
Pisanie funkcji zwracającej wartość	144
Pisanie funkcji matematycznych	146
Debugowanie tworzonoego programu	149
Tworzenie aplikacji z celowym błędem	149
Tworzenie punktu przerwania	150
Pasek narzędzi debugowania	151
Okna debugowania	152
Krokowe wykonywanie kodu	154
Dostosowywanie punktów przerwania	155
Protokołowanie błędów	156
Używanie typów Debug i Trace	157
Przełączanie poziomów śledzenia	160
Testy jednostkowe	162
Tworzenie biblioteki klas wymagającej testowania w Visual Studio 2017	162
Tworzenie projektu testów jednostkowych w Visual Studio 2017	163
Tworzenie biblioteki klas wymagającej testowania w Visual Studio Code	164
Tworzenie testów jednostkowych	166
Uruchamianie testów jednostkowych w Visual Studio 2017	167
Uruchamianie testów jednostkowych w Visual Studio Code	168
Praktyka i ćwiczenia	170
Ćwiczenie 4.1 — sprawdź swoją wiedzę	170
Ćwiczenie 4.2 — tworzenie funkcji z wykorzystaniem debugowania i testów jednostkowych	170
Ćwiczenie 4.3 — dalsza lektura	171
Podsumowanie	171
Rozdział 5. Tworzenie własnych typów w programowaniu obiektowym	173
Rozmowy o programowaniu obiektowym	174
Tworzenie bibliotek klas	174
Tworzenie biblioteki klas w Visual Studio 2017	175
Tworzenie biblioteki klas w Visual Studio Code	175
Definiowanie klasy	176
Tworzenie obiektów	177

Zarządzanie wieloma projektami w Visual Studio Code	180
Dziedziczenie po System.Object	181
Przechowywanie danych w polach	182
Definiowanie pól	182
Zapisywanie wartości za pomocą słowa kluczowego enum	184
Zapisywanie wielu wartości za pomocą kolekcji	187
Tworzenie pól statycznych	188
Tworzenie stałych pól	189
Tworzenie pól tylko do odczytu	190
Inicjowanie pól w konstruktorach	190
Nadawanie polom domyślnego literału	191
Tworzenie i wywoływanie metod	193
Łączenie wielu wartości za pomocą krotki	194
Sterowanie przekazywaniem parametrów	197
Przeciążanie metod	198
Parametry opcjonalne i nazywane	198
Sposoby przekazywania parametrów	200
Dzielenie klas na części	202
Kontrola dostępu za pomocą właściwości i indeksów	203
Definiowanie właściwości tylko do odczytu	203
Definiowanie właściwości z możliwością przypisania	204
Definiowanie indeksów	205
Praktyka i ćwiczenia	206
Ćwiczenie 5.1 — sprawdź swoją wiedzę	206
Ćwiczenie 5.2 — dalsza lektura	207
Podsumowanie	207
Rozdział 6. Implementowanie interfejsów i dziedziczenie klas	209
Konfigurowanie biblioteki klas i aplikacji konsoli	210
Visual Studio 2017	210
Visual Studio Code	210
Definiowanie klas	212
Upraszczenie metod za pomocą operatorów	213
Implementowanie działań w metodzie	213
Implementowanie działań za pomocą operatora	214
Definiowanie funkcji lokalnych	216
Wywoływanie i obsługa zdarzeń	217
Wywoływanie metod za pomocą delegatów	217
Definiowanie zdarzeń	218
Implementowanie interfejsów	220
Typowe interfejsy	220
Porównywanie obiektów podczas sortowania	221
Wykorzystywanie typów generycznych	225
Tworzenie typu generycznego	225
Tworzenie generycznej metody	227

Zarządzanie pamięcią za pomocą typów referencyjnych i typów wartości	228
Definiowanie typu kategorii struct	229
Zwalnianie niezarządzanych zasobów	230
Wymuszanie wywołania metody Dispose	232
Dziedziczenie klas	233
Rozbudowywanie klasy	233
Ukrywanie elementów	234
Pokrywanie elementów klasy	235
Blokowanie dziedziczenia i pokrywania	236
Polimorfizm	237
Rzutowanie w ramach hierarchii dziedziczenia	238
Rzutowanie niejawne	238
Rzutowanie jawne	238
Obsługa wyjątków rzutowania	239
Dziedziczenie i rozbudowywanie typów .NET	240
Dziedziczenie po wyjątku	240
Rozszerzanie typów, po których nie można dziedziczyć	241
Praktyka i ćwiczenia	244
Ćwiczenie 6.1 — sprawdź swoją wiedzę	244
Ćwiczenie 6.2 — tworzenie hierarchii dziedziczenia	244
Ćwiczenie 6.3 — dalsza lektura	244
Podsumowanie	245

Część II. .NET Core 2.0 i .NET Standard 2.0

247

Rozdział 7. Poznawanie typów .NET Standard	249
Zestawy i przestrzenie nazw	249
Bazowe biblioteki klas i CoreFX	250
Poznawanie zależnych zestawów	251
Związki między zestawami i przestrzeniami nazw	252
Związki słów kluczowych języka C# z typami .NET	257
Wieloplatformowe współdzielenie kodu z bibliotekami klas .NET Standard	258
Tworzenie biblioteki klas .NET Standard 2.0	259
Pakiety NuGet	260
Metapakiety	261
Czym są frameworki?	263
Stabilizowanie zależności	263
Publikowanie własnych aplikacji	265
Tworzenie aplikacji konsoli do publikacji	265
Publikowanie za pomocą Visual Studio 2017 w systemie Windows	266
Publikowanie za pomocą Visual Studio Code w macOS	269
Przygotowywanie własnych pakietów NuGet	270
Poznawanie poleceń narzędzia dotnet	270
Dodawanie odwołania do pakietu	271

Tworzenie pakietu dla NuGet	273
Testowanie pakietu	277
Przenoszenie kodu z .NET Framework do .NET Core	278
Co można przenieść?	279
Co należy przenieść?	279
Różnice między .NET Framework i .NET Core	280
Korzystanie z programu .NET Portability Analyzer	280
Używanie bibliotek spoza .NET	280
Praktyka i ćwiczenia	282
Ćwiczenie 7.1 — sprawdź swoją wiedzę	282
Ćwiczenie 7.2 — dalsza lektura	282
Podsumowanie	283
Rozdział 8. Używanie typów biblioteki .NET Standard	285
Praca z liczbami	285
Praca z wielkimi liczbami całkowitymi	286
Praca z liczbami zespolonymi	286
Praca z tekstem	287
Odczytywanie długości ciągu znaków	287
Odczytywanie znaków z ciągu	288
Dzielenie ciągu znaków	288
Pobieranie części ciągu znaków	288
Poszukiwanie tekstu w ciągu	289
Inne elementy klasy string	289
Wydajne tworzenie ciągów znaków	290
Dopasowywanie wzorców za pomocą wyrażeń regularnych	291
Praca z kolekcjami	293
Wspólne funkcje wszystkich kolekcji	294
Poznawanie kolekcji	295
Praca z listami	297
Praca ze słownikami	298
Sortowanie kolekcji	299
Używanie specjalizowanych kolekcji	300
Używanie kolekcji niezmiennych	300
Praca z zasobami sieciowymi	301
Praca z adresami URI, serwerami DNS i adresami IP	301
Pingowanie serwera	302
Praca z typami i atrybutami	303
Numery wersji zestawów	304
Odczytywanie metadanych zestawu	304
Tworzenie własnych atrybutów	306
Inne możliwości refleksji	308
Internacjonalizacja kodu	308
Globalizacja aplikacji	308

Praktyka i ćwiczenia	310
Ćwiczenie 8.1 — sprawdź swoją wiedzę	310
Ćwiczenie 8.2 — wyrażenia regularne	311
Ćwiczenie 8.3 — metody rozszerzające	311
Ćwiczenie 8.4 — dalsza lektura	311
Podsumowanie	312
Rozdział 9. Praca z plikami, strumieniami i serializacją	313
Praca z systemem plików	313
Obsługa środowisk i systemów plików na wielu platformach	314
Obsługa napędów	316
Praca z katalogami	318
Praca z plikami	320
Praca ze ścieżkami	321
Odczytywanie informacji o pliku	322
Zarządzanie plikami	323
Odczytywanie i zapisywanie w strumieniach	324
Zapisywanie do strumieni tekstowych i XML	327
Zwalnianie zasobów plików	329
Kompresowanie strumieni	331
Kodowanie tekstu	333
Kodowanie ciągu znaków jako tablicy bajtów	333
Kodowanie i dekodowanie tekstu w plikach	336
Serializacja obiektów	337
Serializacja do formatu XML	337
Deserializacja danych z formatu XML	340
Dostosowywanie formatu XML	340
Serializowanie do formatu JSON	341
Serializowanie w innych formatach	342
Praktyka i ćwiczenia	342
Ćwiczenie 9.1 — sprawdź swoją wiedzę	342
Ćwiczenie 9.2 — serializowanie do formatu XML	343
Ćwiczenie 9.3 — dalsza lektura	344
Podsumowanie	344
Rozdział 10. Ochrona danych i aplikacji	345
Poznanie słownictwa związanego z ochroną	345
Klucze i wielkości kluczy	346
Wektor inicjujący i wielkość bloku	347
Sól	347
Generowanie kluczy i wektorów inicjujących	348
Szyfrowanie i rozszyfrowywanie danych	349
Szyfrowanie symetryczne algorytmem AES	350
Funkcje skrótu	354
Obliczanie skrótu za pomocą algorytmu SHA256	356

Podpisywanie danych	358
Podpisywanie za pomocą SHA256 i RSA	359
Testowanie i kontrolowanie podpisów	360
Generowanie liczb losowych	361
Generowanie liczb losowych na potrzeby gier	362
Generowanie liczb losowych dla kryptografii	363
Testowanie generatora losowego klucza lub wektora inicjującego	363
Uwierzytelnianie i autoryzacja użytkowników	364
Implementowanie uwierzytelniania i autoryzacji	366
Testowanie autoryzacji i uwierzytelniania	367
Ochrona funkcji aplikacji	369
Praktyka i ćwiczenia	370
Ćwiczenie 10.1 — sprawdź swoją wiedzę	370
Ćwiczenie 10.2 — ochrona danych za pomocą szyfrowania i funkcji skrótu	370
Ćwiczenie 10.3 — ochrona danych przez rozszyfrowywanie	371
Ćwiczenie 10.4 — dalsza lektura	371
Podsumowanie	371
Rozdział 11. Praca z bazami danych przy użyciu Entity Framework Core	373
Nowoczesne bazy danych	373
Używanie przykładowej relacyjnej bazy danych	374
Microsoft SQL Server	375
SQLite	379
Konfigurowanie EF Core	383
Wybieranie dostawcy danych EF Core	383
Łączenie z bazą danych	384
Definiowanie modeli EF Core	387
Konwencje w EF Core	387
Atrybuty EF Core	387
Płynne API EF Core	388
Tworzenie modelu w EF Core	388
Zapytania do modelu EF Core	393
Protokołowanie w EF Core	395
Dopasowywanie wzorców za pomocą instrukcji Like	399
Definiowanie globalnych filtrów	400
Wzorce ładowania w EF Core	401
Manipulowanie danymi w EF Core	404
Wstawianie encji	404
Aktualizowanie encji	405
Usuwanie encji	406
Grupowanie kontekstów baz danych	407
Transakcje	407
Jawne definiowanie transakcji	408

Praktyka i ćwiczenia	409
Ćwiczenie 11.1 — sprawdź swoją wiedzę	409
Ćwiczenie 11.2 — eksportowanie danych z wykorzystaniem różnych formatów serializacji	410
Ćwiczenie 11.3 — przeglądanie dokumentacji EF Core	410
Podsumowanie	410
Rozdział 12. Odczytywanie danych i manipulowanie nimi za pomocą LINQ	411
Tworzenie zapytań LINQ	411
Rozbudowa sekwencji za pomocą klas wyliczeniowych	412
Filtrowanie encji za pomocą metody Where	412
Sortowanie encji	417
Filtrowanie według typu	418
Praca ze zbiorami	420
Używanie LINQ z EF Core	421
Projekcje encji w metodzie Select	422
Tworzenie modelu danych EF Core	422
Łączenie i grupowanie	425
Agregowanie sekwencji	428
Upiększanie składni	429
Używanie wielu wątków w równoległych zapytaniach LINQ	430
Tworzenie własnych metod rozszerzających dla LINQ	433
Praca z LINQ to XML	436
Generowanie danych XML za pomocą LINQ to XML	437
Odczytywanie danych XML za pomocą LINQ to XML	437
Praktyka i ćwiczenia	438
Ćwiczenie 12.1 — sprawdź swoją wiedzę	438
Ćwiczenie 12.2 — zapytania LINQ	439
Ćwiczenie 12.3 — dalsza lektura	439
Podsumowanie	440
Rozdział 13. Poprawianie wydajności i skalowalności za pomocą wielozadaniowości	441
Monitorowanie wydajności i wykorzystania zasobów	442
Ocena wydajności typów	442
Monitorowanie wydajności i zużycia pamięci	443
Procesy, wątki i zadania	448
Asynchroniczne uruchamianie zadań	449
Synchroniczne uruchamianie wielu operacji	450
Asynchroniczne uruchamianie wielu operacji z wykorzystaniem zadań	451
Oczekiwanie na zadania	453
Kontynuowanie pracy w innym zadaniu	454
Zadania zagnieżdżone i potomne	455

Synchronizowanie dostępu do wspólnych zasobów	456
Używanie wspólnego zasobu w wielu wątkach	457
Nakładanie na zasoby wzajemnie wykluczającej blokady	459
Jak działa instrukcja lock	459
Tworzenie operacji atomowych	461
Stosowanie innych rodzajów synchronizacji	462
Słowa kluczowe async i await	462
Poprawianie reakcji aplikacji konsoli	463
Poprawianie reakcji aplikacji z interfejsem graficznym	464
Poprawianie skalowalności aplikacji i serwisów WWW	465
Często używane typy pozwalające na pracę wielowątkową	465
Instrukcja await w bloku catch	465
Praktyka i ćwiczenia	466
Ćwiczenie 13.1 — sprawdź swoją wiedzę	466
Ćwiczenie 13.2 — dalsza lektura	466
Podsumowanie	467

Część III. Modele aplikacji **469**

Rozdział 14. Tworzenie witryn WWW przy użyciu ASP.NET Core Razor Pages **473**

Tworzenie w sieci WWW	473
Protokół HTTP	473
Tworzenie oprogramowania dla sieci WWW po stronie klienta	477
ASP.NET Core	478
Klasyczna ASP.NET kontra ASP.NET Core	479
Tworzenie projektu ASP.NET Core w Visual Studio 2017	480
Tworzenie projektu ASP.NET Core w Visual Studio Code	480
Przeglądanie pustego szablonu projektu ASP.NET Core	480
Testowanie pustej witryny	483
Włączanie plików statycznych	484
Włączanie plików domyślnych	487
Technologia Razor Pages	487
Włączanie technologii Razor Pages	487
Definiowanie strony Razor	488
Używanie wspólnego układu w wielu stronach Razor	489
Używanie plików code-behind w stronach Razor	492
Używanie Entity Framework Core z ASP.NET Core	494
Tworzenie modelu encji dla firmy Northwind	494
Tworzenie bazy danych Northwind dla witryny	502
Konfigurowanie Entity Framework Core jako serwisu	502
Manipulowanie danymi	504
Praktyka i ćwiczenia	507
Ćwiczenie 14.1 — tworzenie witryny obsługującej dane	507
Ćwiczenie 14.2 — dalsza lektura	507
Podsumowanie	507

Rozdział 15. Tworzenie aplikacji WWW przy użyciu ASP.NET Core MVC	509
Konfigurowanie witryny ASP.NET Core MVC	509
Tworzenie witryny ASP.NET Core MVC	510
Przegląd szablonu projektu ASP.NET Core MVC	513
Migrowanie bazy danych	515
Testowanie witryny ASP.NET Core MVC	517
Mechanizm uwierzytelniania systemu ASP.NET Identity	519
Poznananie mechanizmów ASP.NET Core MVC	520
Rozruch ASP.NET Core	520
Czym jest domyślna ścieżka	522
Kontrolery ASP.NET Core MVC	522
Modele ASP.NET Core MVC	523
Widoki ASP.NET Core MVC	525
Przekazywanie parametrów przy użyciu wartości ścieżki	532
Przekazywanie parametrów za pomocą ciągu znaków zapytania	534
Praktyka i ćwiczenia	536
Ćwiczenie 15.1 — poprawianie skalowalności przez poznananie i implementowanie asynchronicznych metod akcji	536
Ćwiczenie 15.2 — dalsza lektura	537
Podsumowanie	538
Rozdział 16. Tworzenie usług i aplikacji WWW przy użyciu ASP.NET Core	539
Tworzenie serwisów w technologii ASP.NET Core Web API	539
Kontrolery ASP.NET Core	540
Tworzenie projektu ASP.NET Core Web API	540
Tworzenie serwisu internetowego dla bazy danych Northwind	544
Tworzenie repozytorium danych dla encji	545
Dokumentowanie i testowanie serwisów przy użyciu narzędzia Swagger	551
Testowanie żądań GET za pomocą przeglądarki	551
Testowanie żądań POST, PUT i DELETE w narzędziu Swagger	553
Instalowanie pakietu Swagger	553
Testowanie żądań GET w narzędziu Swagger	554
Testowanie żądań POST za pomocą narzędzia Swagger	557
Tworzenie aplikacji SPA przy użyciu biblioteki Angular	561
Szablon projektu Angular	561
Wywoływanie serwisu Northwind	565
Testowanie wywołań serwisu z komponentu Angular	568
Używanie innych szablonów projektów	569
Instalowanie dodatkowych pakietów z szablonami	570
Praktyka i ćwiczenia	570
Ćwiczenie 16.1 — aplikacje React i Redux	571
Ćwiczenie 16.2 — dalsza lektura	572
Podsumowanie	573

Rozdział 17. Tworzenie aplikacji dla Windowsa przy użyciu XAML i Fluent Design	575
Poznanie nowoczesnej platformy Windows	576
Universal Windows Platform	576
System Fluent Design	577
XAML Standard 1.0	578
Tworzenie nowoczesnej aplikacji dla Windowsa	579
Włączanie trybu dewelopera	580
Tworzenie projektu UWP	580
Przeglądanie typowych kontrolki i akrylowych pędzli	584
Oświetlenie ujawniające	586
Instalowanie dodatkowych kontrolki	589
Używanie zasobów i szablonów	590
Współdzielenie zasobów	591
Zmiana szablonu kontrolki	592
Wiązanie danych	594
Wiązanie do elementów	594
Wiązanie do źródła danych	595
Tworzenie aplikacji przy użyciu Windows Template Studio	606
Instalowanie Windows Template Studio	606
Wybieranie typu projektu, frameworka, stron i funkcji	607
Poprawianie ustawień projektu	610
Poprawianie wybranych widoków	610
Testowanie działania aplikacji	612
Praktyka i ćwiczenia	613
Ćwiczenie 17.1 — dalsza lektura	613
Podsumowanie	614
Rozdział 18. Tworzenie aplikacji mobilnych przy użyciu XAML i Xamarin.Forms	615
Xamarin i Xamarin.Forms	615
W jaki sposób Xamarin.Forms rozbudowuje platformę Xamarin	616
Najpierw mobilne, najpierw chmura	616
Tworzenie aplikacji mobilnej za pomocą Xamarin.Forms	617
Dodawanie SDK systemu Android	617
Tworzenie rozwiązania Xamarin.Forms	618
Tworzenie modelu	620
Tworzenie interfejsu do wybierania numerów telefonów	624
Tworzenie widoków listy klientów i szczegółowych danych klienta	627
Testowanie aplikacji mobilnej w systemie iOS	632
Dodawanie pakietów NuGet w celu wywołania serwisu REST	637
Pobieranie listy klientów z serwisu	638
Praktyka i ćwiczenia	640
Ćwiczenie 18.1 — dalsza lektura	640
Podsumowanie	640
Podsumowanie	641
Powodzenia!	641

Dodatek A. Odpowiedzi na pytania z testów	643
Rozdział 1. Cześć, C#! Witaj, .NET Core!	643
Rozdział 2. Mówimy w C#	644
Rozdział 3. Sterowanie przepływem i konwertowanie typów	645
Rozdział 4. Pisanie, debugowanie i testowanie funkcji	647
Rozdział 5. Tworzenie własnych typów w programowaniu obiektowym	648
Rozdział 6. Implementowanie interfejsów i dziedziczenie klas	650
Rozdział 7. Poznawanie typów .NET Standard	651
Rozdział 8. Używanie typów biblioteki .NET Standard	652
Rozdział 9. Praca z plikami, strumieniami i serializacją	654
Rozdział 10. Ochrona danych i aplikacji	655
Rozdział 11. Praca z bazami danych przy użyciu Entity Framework Core	656
Rozdział 12. Odczytywanie danych i manipulowanie nimi za pomocą LINQ	658
Rozdział 13. Poprawianie wydajności i skalowalności za pomocą wielozadaniowości	659
Skorowidz	661

Cześć, C#!

Witaj, .NET Core

W tym rozdziale zajmiemy się przygotowaniem środowiska programistycznego i przedstawimy podobieństwa i różnice pomiędzy .NET Core, .NET Framework, .NET Standard i .NET Native. Spróbujemy też użyć najróżniejszych narzędzi, żeby przygotować bardzo prostą aplikację z wykorzystaniem C# i .NET Core.

Większość ludzi uczy się złożonych rzeczy przez powtarzanie widzianych już akcji, a nie przez czytanie wyczerpujących opisów teoretycznych. Dlatego nie będę tu objaśniał każdego użytego słowa kluczowego i kroku. Chodzi o to, żeby nauczyć Cię pisania kodu, budowania aplikacji i uruchamiania jej. Na razie nie musisz znać każdego szczegółu ani wiedzieć dokładnie, jak to wszystko działa.

Przywołując słowa Samuela Johnsona, autora słownika języka angielskiego z 1755 r., najpewniej popełniłem tu „kilka błędów oraz sporych absurdów, od których nie jest wolna żadna praca tej wielkości”. Biorę na siebie pełną odpowiedzialność za te błędy, mając nadzieję, że docenisz moje starania podczas pisania książki o platformie .NET Core i związanych z nią narzędziach już w czasie jej narodzin w latach 2016 i 2017.

W rozdziale zostaną omówione następujące zagadnienia:

- konfigurowanie środowiska programistycznego,
- poznawanie .NET,
- pisanie i kompilowanie kodu za pomocą narzędzi wiersza poleceń z .NET Core,
- pisanie i kompilowanie kodu za pomocą Visual Studio 2017,
- pisanie i kompilowanie kodu za pomocą Visual Studio Core,
- pisanie i kompilowanie kodu za pomocą Visual Studio for Mac,
- obsługa kodu źródłowego z wykorzystaniem platformy GitHub.

Konfigurowanie środowiska programistycznego

Zanim zaczniemy programować, trzeba wybrać **zintegrowane środowisko programistyczne** (ang. *Integrated Development Environment* — IDE), którego częścią jest edytor kodu dla języka C#. Microsoft udostępnia małą rodzinę takich środowisk:

- Visual Studio 2017,
- Visual Studio for Mac,
- Visual Studio Code.

Najbardziej rozbudowanym wariantem, oferującym najwięcej funkcji, jest **Microsoft Visual Studio 2017**, ale działa on jedynie w systemach operacyjnych Windows.

Najnowocześniejsze i najlżejsze IDE, jakie mamy do wyboru, i jedyne, które w rodzinie Microsoftu jest wieloplatformowe, to **Microsoft Visual Studio Code**. Działa ono w wielu różnych systemach operacyjnych, takich jak Windows, macOS, oraz w wielu wariantach Linuksa, np. **Red Hat Enterprise Linux (RHEL)** albo Ubuntu.

W podjęciu decyzji, czy Visual Studio Code jest dla Ciebie właściwym wyborem, z pewnością pomoże obejrzenie wideo *Beginner's Guide to VS Code: Up and Running in Ten Minutes*: <https://channel9.msdn.com/Blogs/raw-tech/Beginners-Guide-to-VS-Code>.

IDE najlepiej dostosowane do rozwijania programów dla urządzeń mobilnych to **Visual Studio for Mac**. Aby stworzyć oprogramowanie dla systemów iOS (dla iPhone'a i iPada), tvOS, macOS i watchOS, trzeba mieć dostępny system macOS i środowisko Xcode. Co prawda można też skorzystać z Visual Studio 2017 z rozszerzeniem Xamarin do **napisania** wieloplatformowej aplikacji mobilnej, ale do jej **skompilowania** niezbędne będą macOS i Xcode.

W poniższej tabeli można sprawdzić, które IDE i który system operacyjny będziemy wykorzystywali w poszczególnych rozdziałach tej książki.

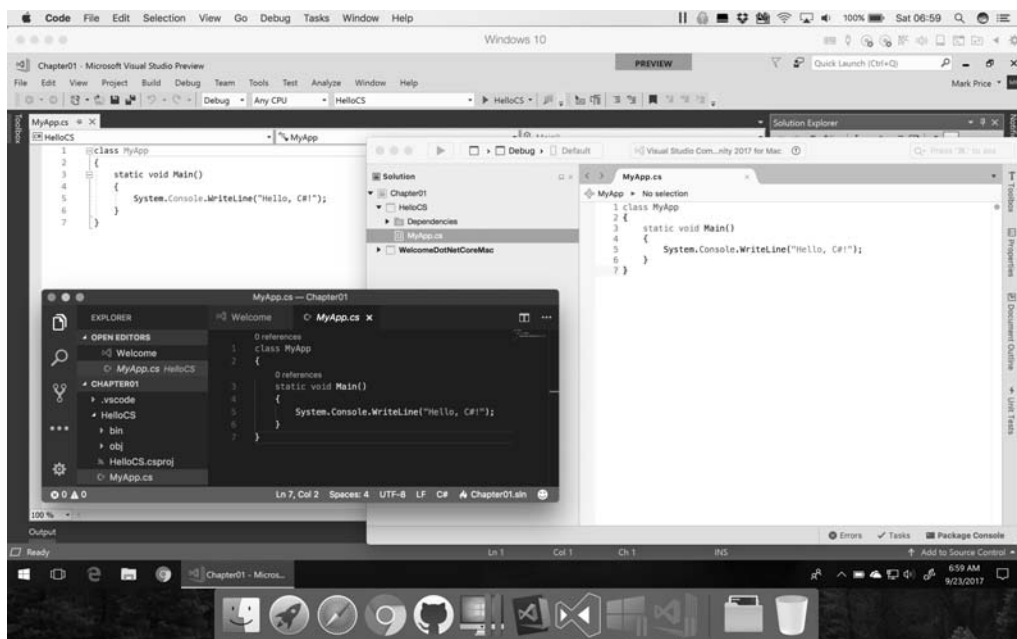
Rozdziały	IDE	Systemy operacyjne
od 1. do 16.	Visual Studio 2017	Windows 7 SP1 lub nowszy
od 1. do 16.	Visual Studio Code	Windows, macOS, Linux
od 1. do 16.	Visual Studio for Mac	macOS
17.	Visual Studio 2017	Windows 10
18.	Visual Studio for Mac	macOS

Dobra praktyka

Jeżeli masz taką możliwość, to zachęcam do wykonania wszystkich ćwiczeń w środowiskach Visual Studio 2017 dla Windowsa, Visual Studio Code w systemach macOS, Linux i Windows oraz Visual Studio for Mac. Dzięki temu zyskasz doświadczenie w pracy z językiem C# i platformą .NET Core w różnych systemach operacyjnych i z wykorzystaniem różnych narzędzi.

Pisząc trzecie wydanie tej książki, używałem oprogramowania wymienionego poniżej, przedstawionego na poniższym rysunku:

- Visual Studio 2017 w systemie Windows 10 w maszynie wirtualnej,
- Visual Studio for Mac w systemie macOS,
- Visual Studio Code w systemie macOS,
- Visual Studio Code w systemie RHEL (niewidoczne na rysunku).



Używanie innych IDE dla języka C#

Istnieją też inne IDE dla języka C#, np. **Mono Develop** albo **JetBrains Rider**. Możesz zainstalować dowolne z nich, korzystając z poniższych adresów URL:

- Środowisko MonoDevelop znajdziesz pod adresem <http://www.monodevelop.com/>.
- JetBrains Rider dostępne jest pod adresem <https://www.jetbrains.com/rider/>.

Instalowanie na wielu platformach

Wybór IDE i systemu operacyjnego na potrzeby tworzenia oprogramowania nie ogranicza nam możliwości instalowania go na innych platformach. .NET Core 2.0 pozwala zainstalować oprogramowanie na następujących platformach:

- Windows 7 SP1 i nowsze,
- Windows Server 2008 R2 SP1 i nowsze,
- Windows IoT 10 i nowsze,
- macOS Sierra (wersja 10.12) i nowsze,
- RHEL 7.3 i nowsze,
- Ubuntu 14.04 i nowsze,
- Fedora 25 i nowsze,
- Debian 8.7 i nowsze,
- openSUSE 42.2 i nowsze,
- Tizen 4 i nowsze.

Systemy linuksowe są bardzo popularne na platformach hostujących usługi serwerowe ze względu na ich lekkość i możliwość tańszego skalowania w porównaniu do takich platform jak Windows lub macOS.

W następnym podrozdziale zainstalujemy Visual Studio 2017 w systemie Windows. Jeżeli wolisz używać Visual Studio Code, to możesz od razu przejść do podrozdziału „Instalowanie Microsoft Visual Studio Code” dla systemu Windows, macOS lub Linux. Jeżeli chcesz korzystać z Microsoft Visual Studio for Mac, to przejdź do podrozdziału „Instalowanie Visual Studio for Mac”.

Instalowanie Microsoft Visual Studio 2017

Do wykonania zadań z większości rozdziałów tej książki wystarczy system Windows 7 SP1, ale znacznie lepszym rozwiązaniem jest zastosowanie systemu Windows 10 Fall Creators Update.

W październiku 2014 r. Microsoft udostępnił za darmo doskonale wyposażoną wersję Visual Studio, która otrzymała nazwę **Community Edition**.

Korzystając z poniższego adresu, możesz pobrać i zainstalować **Microsoft Visual Studio 2017 w wersji 15.4 lub nowszej**:

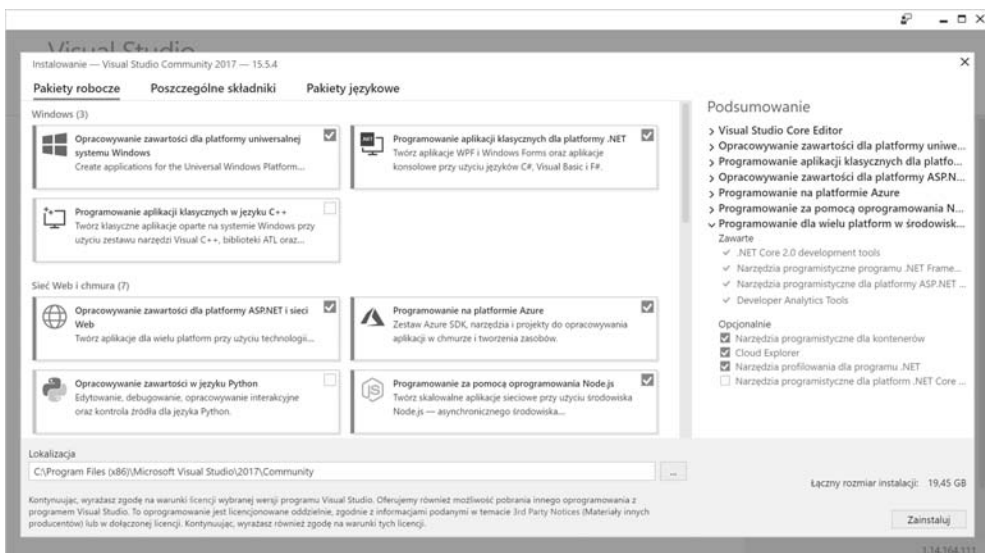
<https://www.visualstudio.com/downloads/>.

Żeby móc skorzystać z platformy .NET Core dla UWP, musisz zainstalować Visual Studio 2017 w wersji 15.4 albo nowszej. Wersja 15.3 lub nowsza wymagana jest do pracy z platformą .NET Core 2.0. Starsze wersje Visual Studio 2017 pozwalają jedynie na obsługę .NET Core 1.0 i 1.1.

Wybieranie pakietów

Na karcie *Pakiety robocze* zaznacz poniższe elementy, które są też częściowo widoczne na poniższym rysunku:

- *Opracowanie zawartości dla platformy uniwersalnej systemu Windows,*
- *Programowanie aplikacji klasycznych dla platformy .NET,*
- *Opracowywanie zawartości dla platformy ASP.NET i sieci Web,*
- *Programowanie na platformie Azure,*
- *Programowanie za pomocą oprogramowania Node.js,*
- *Programowanie dla wielu platform w środowisku .NET Core.*



Wybieranie dodatkowych komponentów

Na karcie *Individual components* zaznacz jeszcze poniższe komponenty:

- *Class Designer,*
- *GitHub extension for Visual Studio,*
- *PowerShell tools.*

Kliknij przycisk *Install* i czekaj, aż instalator pobierze wybrane pakiety i je zainstaluje. Po zakończeniu instalacji kliknij przycisk *Launch*.

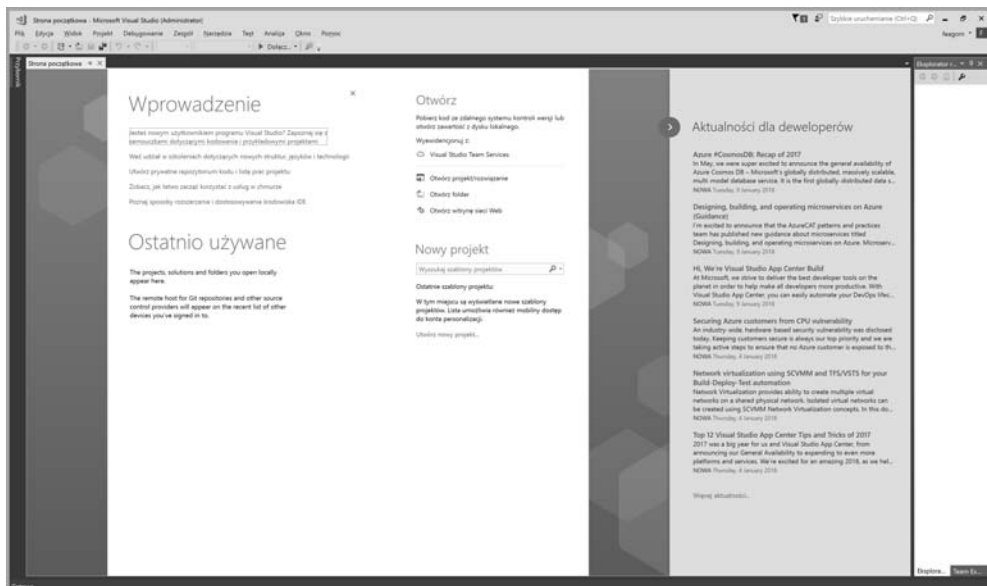
Czekając na zainstalowanie Visual Studio 2017, możesz zajrzeć do podrozdziału „Poznawanie .NET” w dalszej części tego rozdziału.

Przy pierwszym uruchomieniu Visual Studio 2017 pojawi się prośba o zalogowanie. Jeżeli masz konto Microsoftu, to użyj go do zalogowania się. Jeżeli nie masz jeszcze konta, to możesz utworzyć nowe, korzystając z poniższego adresu:

<http://signup.live.com/>.

Uruchamiając pierwszy raz Visual Studio 2017, zobaczysz ekran zachęcający do skonfigurowania środowiska. W sekcji *Ustawienia* zaznacz opcję *Visual C#*. Jeżeli chodzi o motyw kolorów, to sam wybieram niebieski, ale tutaj każdy może zdecydować według własnego uznania.

Później zobaczysz interfejs użytkownika Microsoft Visual Studio ze *Stroną początkową* otwartą w środkowej części. Podobnie jak w większości aplikacji dla systemu Windows, Visual Studio ma własny pasek menu, pasek narzędzi z najważniejszymi poleceniami oraz pasek stanu na dole. Po prawej znajduje się *Eksplorez rozwiązań*, który zawiera listę otwartych projektów:



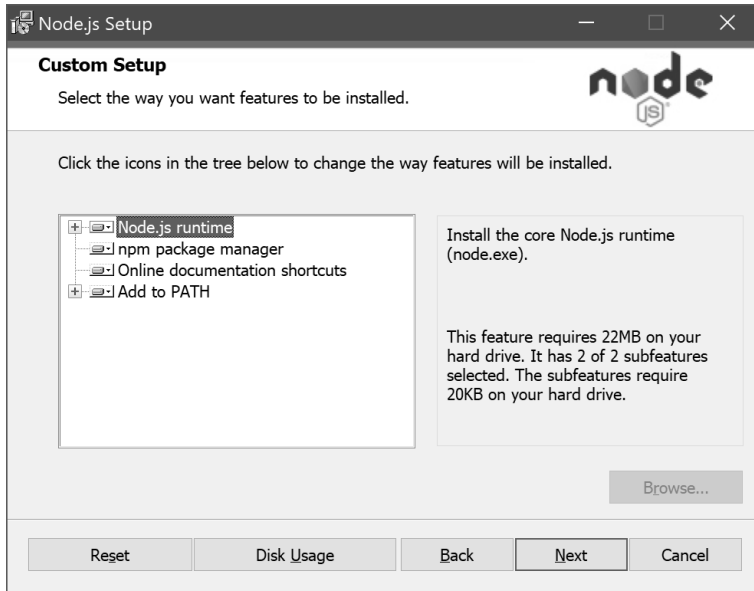
Jeśli chcesz mieć w przyszłości szybki dostęp do Visual Studio, możesz kliknąć prawym przyciskiem myszy jego ikonę na pasku zadań systemu Windows i wybrać opcję *Przypnij do paska zadań*.

Do prac z rozdziałów 14., 15. i 16. konieczne będzie zainstalowanie pakietów App Node.js oraz NPM.

Instalator pakietu Node.js dla systemu Windows możesz pobrać z poniższego adresu:

<https://nodejs.org/en/download/>.

Po uruchomieniu instalatora zastosuj ustawienia jak na poniższym rysunku:



Instalowanie Microsoft Visual Studio Code

Pomiędzy czerwcem 2015 r. a wrześniem 2017 r. Microsoft niemal co miesiąc wydawał nową wersję **Visual Studio Code**. Pakiet ten rozwija się bardzo szybko i jego popularność chyba trochę zaskoczyła Microsoft. Nawet jeżeli planujesz korzystać z Visual Studio 2017 lub Visual Studio for Mac jako głównego narzędzia do pracy, to i tak zalecam zapoznanie się również z Visual Studio Code i narzędziami wiersza poleceń .NET Core.

Visual Studio Code możesz pobrać z poniższego adresu:

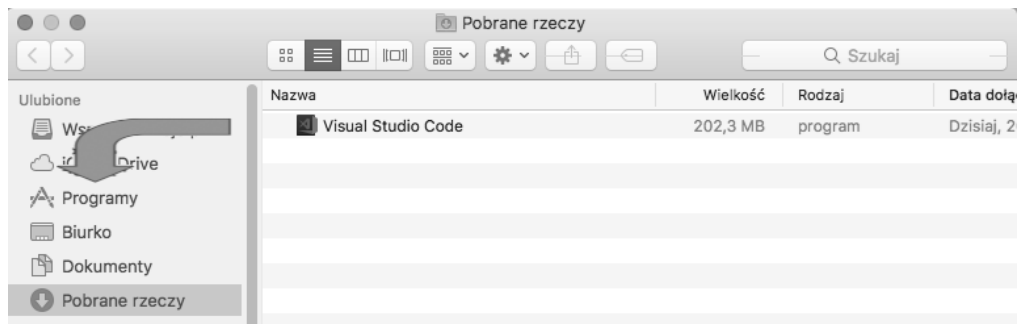
<https://code.visualstudio.com/>.

O planach rozwoju Visual Studio Code w 2018 r. możesz poczytać pod tym adresem: <https://github.com/Microsoft/vscode/wiki/Roadmap>.

Instalowanie Microsoft Visual Studio Code for macOS

W książce przedstawię przykłady i zrzuty ekranów pochodzących z Visual Studio Code w wersji dla macOS. Kroki niezbędne do wykonania tych samych operacji w systemach Windows i Linux są bardzo podobne, dlatego nie będę powtarzał tu instrukcji dla każdego z tych systemów z osobna.

Po pobraniu pakietu Visual Studio for macOS musisz przeciągnąć go i upuścić na folder *Programy*, tak jak pokazano na poniższym rysunku:



Teraz musisz jeszcze zainstalować .NET Code SDK for macOS. Dokładne instrukcje instalacji, wraz z filmem instruktażowym, są dostępne pod poniższym adresem, a w tej książce zawarłem jeszcze proste kroki wykonania tej instalacji:

<https://www.microsoft.com/net/core#macos>.

W pierwszym kroku należy zainstalować Homebrew (o ile nie jest już zainstalowane).

Następnie musisz uruchomić aplikację *Terminal*, a po pojawieniu się wiersza poleceń wpisać w nim:

```
/usr/bin/ruby -e "$(curl -fsSL
↳https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Aplikacja *Terminala* poprosi o wciśnięcie klawisza *Enter*, a następnie zapyta o hasło użytkownika.

Jeżeli używasz .NET Core w wersji 1.0 lub 1.1, to na tym etapie skorzystaj z Homebrew do zainstalowania pakietu OpenSSL, który jest wymagany przez wcześniejsze wersje .NET Core for macOS.

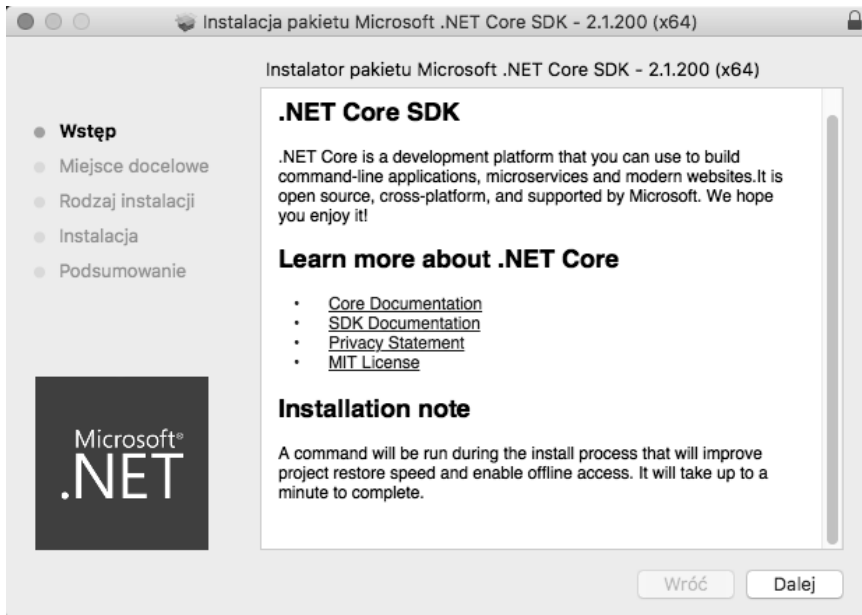
Instalowanie .NET Core SDK for macOS

W kolejnym kroku musisz pobrać instalator pakietu .NET Core SDK w wersji dla macOS (x64). Skorzystaj z poniższego adresu:

<https://www.microsoft.com/net/download/core>.

Uruchom instalator pakietu *dotnet-sdk-2.0.0-sdk-osx-x64.pkg*, tak jak pokazałem na rysunku na następnej stronie.

Kliknij przycisk *Dalej*, zaakceptuj umowę licencyjną, a potem kliknij przycisk *Install*. Po zakończeniu instalacji pozostaje już tylko kliknąć przycisk *Close*.



Instalowanie menedżera pakietów Node dla macOS

Do prac prowadzonych w rozdziałach 14., 15. i 16. konieczne będzie zainstalowanie pakietów Node.js oraz NPM.

W *Terminalu* wprowadź polecenia instalujące pakiety Node.js i NPM, a następnie skontroluj ich wersje. W czasie gdy pisałem tę książkę, Node.js był dostępny w wersji 8.4, a NPM w wersji 5.3, co potwierdza pierwszy rysunek na następnej stronie.

```
brew install node
node -v
npm -v
```

Instalowanie rozszerzenia C# for Visual Studio Code

Rozszerzenie *C# for Visual Studio Code* nie jest elementem wymaganym, ale umożliwia skorzystanie z funkcji IntelliSense w czasie pisania kodu, co jest wyjątkowo użyteczne.

Uruchom Visual Studio Code, a następnie kliknij ikonę *Extensions* albo wybierz z menu pozycję *View/Extensions*, albo naciśnij klawisze *Cmd+Shift+X*.

C# jest bardzo popularnym rozszerzeniem, dlatego powinno znaleźć się na samym szczycie listy, tak jak na drugim rysunku na następnej stronie.

```

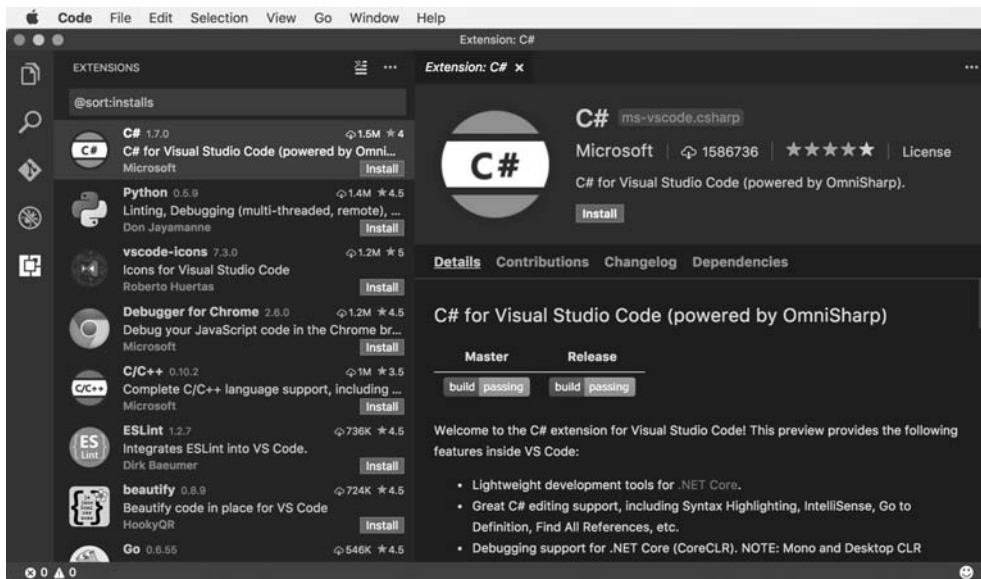
Terminal Powloka Edycja Widok Okno Pomoc
markjprice --bash -- 80x35

==> Installing dependencies for node: icu4c
==> Installing node dependency: icu4c
==> Downloading https://homebrew.bintray.com/bottles/icu4c-61.1.sierra.bottle.ta
==> Downloading from https://akamai.bintray.com/07/07bad03c12d39c9216caa94ff85a2
##### 100.0%
==> Pouring icu4c-61.1.sierra.bottle.tar.gz
==> Caveats
This formula is keg-only, which means it was not symlinked into /usr/local,
because macOS provides libicucore.dylib (but nothing else).

If you need to have this software first in your PATH run:
  echo 'export PATH="/usr/local/opt/icu4c/bin:$PATH"' >> ~/.bash_profile
  echo 'export PATH="/usr/local/opt/icu4c/sbin:$PATH"' >> ~/.bash_profile

For compilers to find this software you may need to set:
  LDFLAGS:  -L/usr/local/opt/icu4c/lib
  CPPFLAGS: -I/usr/local/opt/icu4c/include

==> Summary
  /usr/local/Cellar/icu4c/61.1: 249 files, 67.2MB
==> Installing node
==> Downloading https://homebrew.bintray.com/bottles/node-10.2.1.sierra.bottle.t
==> Downloading from https://akamai.bintray.com/e2/e2c3f4541c056fbad01625529bbf9
##### 100.0%
==> Pouring node-10.2.1.sierra.bottle.tar.gz
==> Caveats
Bash completion has been installed to:
  /usr/local/etc/bash_completion.d
==> Summary
  /usr/local/Cellar/node/10.2.1: 5,301 files, 51.9MB
marks-iMac:~ markjprice$ node -v
v10.2.1
marks-iMac:~ markjprice$ npm -v
5.6.0
marks-iMac:~ markjprice$
    
```



Kliknij tu przycisk *Install*, a następnie przycisk *Reload*, żeby ponownie załadować zawartość okna i aktywować rozszerzenie.

Instalowanie Visual Studio for Mac

W listopadzie 2016 r. Microsoft udostępnił wstępną wersję środowiska Visual Studio for Mac. Początkowo pozwalało ono na tworzenie wyłącznie aplikacji mobilnych Xamarin, ponieważ była to specjalna wersja produktu Xamarin Studio. W ostatecznym wydaniu, które pojawiło się w maju 2017 r., dodana została możliwość tworzenia bibliotek klas .NET Standard 2.0, aplikacji ASP.NET Core oraz serwisów i aplikacji konsolowych. Oznacza to, że można go użyć do wykonania (prawie) wszystkich ćwiczeń z tej książki.

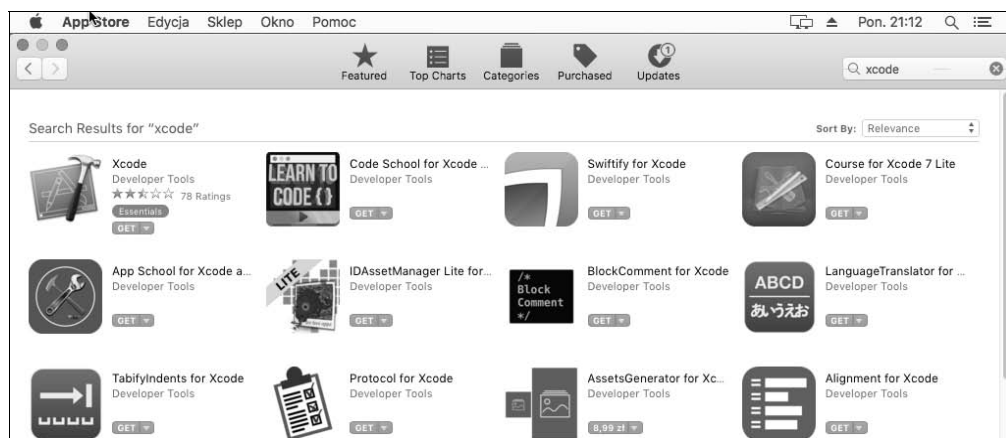
Co prawda Visual Studio 2017 dla Windowsa można wykorzystać do tworzenia aplikacji mobilnych dla systemów iOS i Android, ale do skompilowania wersji dla iOS potrzeba narzędzi Xcode działających w systemie macOS. Moim zdaniem można swobodnie skorzystać ze środowiska Visual Studio for Mac do tworzenia aplikacji mobilnych.

Instalowanie Xcode

Jeżeli jeszcze nie masz zainstalowanego Xcode na swoim komputerze mac, to skorzystaj z App Store, żeby go zainstalować.

Z menu *Apple* wybierz pozycję *App Store...*

W oknie *App Store* wpisz w polu wyszukiwania *xcode*, a jednym z pierwszych wyników będzie aplikacja *Xcode*, tak jak pokazano na poniższym rysunku:



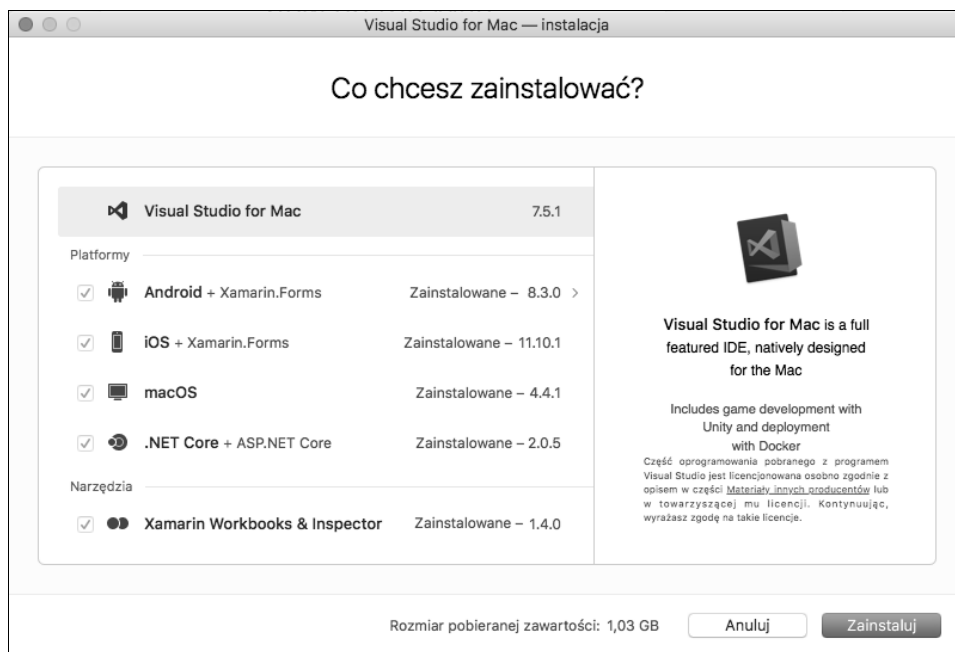
Kliknij przycisk *Get* i poczekaj na zakończenie instalacji.

Pobieranie i instalowanie Visual Studio for Mac

Visual Studio for Mac możesz pobrać z poniższego adresu URL:

<https://www.visualstudio.com/vs/visual-studio-mac/>.

Po uruchomieniu instalatora *Visual Studio for Mac* trzeba zaakceptować warunki licencji i informacje o zachowaniu prywatności, a następnie wybrać komponenty do zainstalowania, tak jak pokazano na poniższym rysunku. Na zakończenie kliknij jeszcze przycisk *Kontynuuj*.



W kolejnym oknie kliknij przycisk *Kontynuuj*, a w następnym — *Zainstaluj*.

Pozostaje jeszcze zaakceptować warunki licencji poszczególnych instalowanych komponentów, np. SDK systemu Android, kliknąc przycisk *Kontynuuj* i poczekać na zakończenie instalacji Visual Studio for Mac.

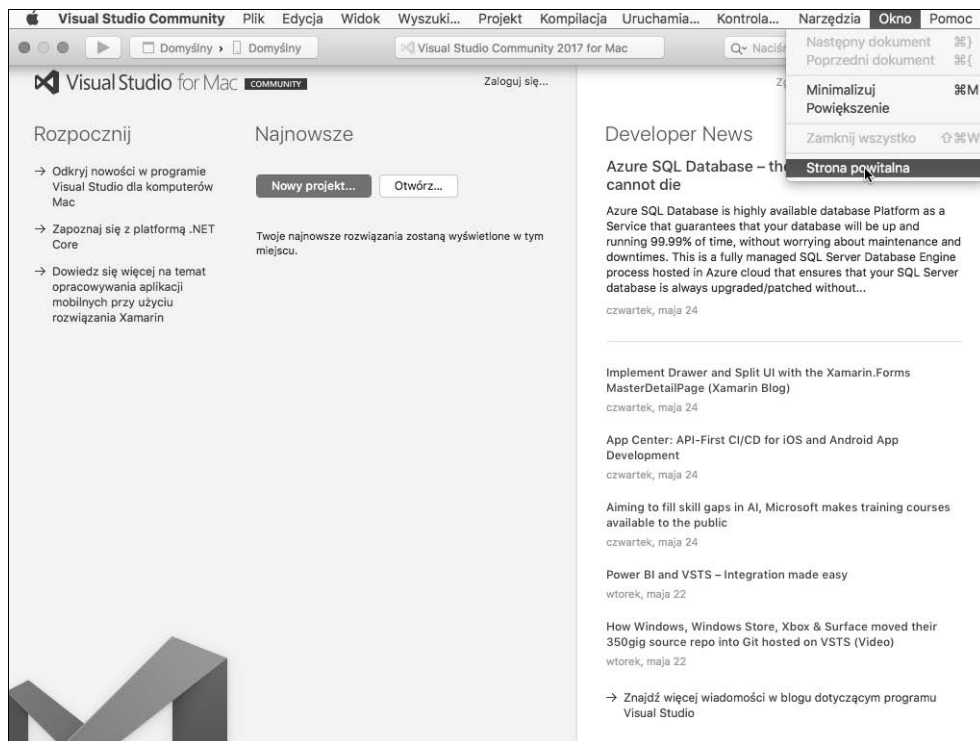
Po uruchomieniu Visual Studio for Mac zobaczysz stronę *Strona powitalna*, taką jak na rysunku na następnej stronie.

Jeżeli pojawi się prośba o aktualizację, zaakceptuj ją.

Skoro masz już zainstalowane i skonfigurowane środowisko programistyczne, możemy przystąpić do poznawania środowiska .NET i rozpocząć pisanie kodu.

Poznawanie .NET

.NET Framework, .NET Core, .NET Standard i .NET Native są powiązаныmi ze sobą platformami programistycznymi, których programiści mogą używać do tworzenia aplikacji i usług.



Poznanie .NET Framework

.NET Framework jest platformą programistyczną, w skład której wchodzi środowisko wykonawcze **CLR** (ang. *Common Language Runtime*) zajmujące się wykonywaniem kodu programów i udostępniające im bogatą bibliotekę klas.

Microsoft zaprojektował .NET Framework w taki sposób, żeby zachować możliwość pracy wieloplatformowej, ale jednocześnie całość swoich prac prowadził tak, żeby ta platforma najlepiej działała w systemie Windows.

To oznacza, że .NET Framework jest platformą wyłącznie dla Windowsa i jako taka jest powoli porzucana.

Poznanie projektów Mono i Xamarin

Powstała też niezależna implementacja platformy .NET, o której możesz przeczytać pod poniższym adresem:

<http://www.mono-project.com/>.

Mono jest implementacją wieloplatformową, ale pozostaje daleko w tyle za oficjalną implementacją platformy .NET Framework. Znalazła swoją niszę, stając się bazą dla platformy mobilnej **Xamarin**.

Microsoft wykupił platformę **Xamarin** w 2016 r. i teraz rozpowszechnia to, co było bardzo drogim pakietem, jako bezpłatny dodatek do Visual Studio 2017. Istniejące wcześniej **Xamarin Studio** Microsoft przemianował na **Visual Studio for Mac** i dodał do niego możliwość tworzenia serwisów ASP.NET Core dla sieci WWW. Pierwotnie platforma **Xamarin** specjalizowała się w tworzeniu aplikacji dla urządzeń mobilnych oraz budowaniu usług współdziałających z tymi aplikacjami.

Poznanie .NET Core

Żyjemy w świecie realizującym ideę wieloplatformowości. Nowoczesne metody tworzenia oprogramowania dla urządzeń mobilnych i dla chmury sprawiły, że Windows stracił na znaczeniu jako system operacyjny. W związku z tym Microsoft intensywnie pracuje nad tym, żeby poluzować związki łączące platformę .NET z systemami Windows.

Podczas ponownego projektowania środowiska .NET w sposób rzeczywiście wieloplatformowy Microsoft skorzystał z tej okazji, żeby przebudować platformę, usuwając z niej te ważne elementy, które dzisiaj nie są już uznawane za **podstawowe** (ang. *core*).

Nowy produkt otrzymał nazwę **.NET Core**, a jego podstawami stały się wieloplatformowa implementacja środowiska CLR o nazwie **CoreCLR** oraz odchudzona biblioteka klas, która otrzymała nazwę **CoreFX**.

Scott Hunter — odpowiedzialny za platformę .NET menedżer programu Microsoft Partner Director — powiedział: „Czterdzieści procent naszych klientów .NET Core to programiści wdrażający się dopiero w platformę. Tego właśnie tutaj chcemy. Chcemy pozyskać nowych ludzi”.

W poniższej tabeli podaję, kiedy zostały udostępnione poszczególne wersje .NET Core oraz plan Microsoftu opisujący przyszłe wydania platformy:

Wersja	Data wydania
.NET Core RC1	listopad 2015
.NET Core 1.0	czerwiec 2016
.NET Core 1.1	listopad 2016
.NET Core 1.0.4 i .NET Core 1.1.1	marzec 2017
.NET Core 2.0	sierpień 2017
.NET Core for UWP jako część aktualizacji Windows 10 Fall Creators Update	październik 2017
.NET Core 2.1	pierwszy kwartał 2018

Jeżeli musisz pracować z .NET Core 1.0 i .NET Core 1.1, to proponuję przeczytać ogłoszenie związane z .NET Core 1.1, ponieważ informacje zawarte pod poniższym adresem URL są przydatne dla każdego programisty używającego .NET Core:

<https://blogs.msdn.microsoft.com/dotnet/2016/11/16/announcing-net-core-1-1/>.

.NET Core jest znacznie mniejsza od aktualnej wersji .NET Framework, ponieważ zostało z niej usuniętych wiele różnych elementów.

Na przykład biblioteki **Windows Forms** oraz **Windows Presentation Foundation (WPF)** można wykorzystać do tworzenia aplikacji z graficznym interfejsem użytkownika (GUI), ale obie są ściśle związane z systemem Windows i dlatego nie znalazły miejsca w .NET Core. Najnowsza technologia do tworzenia aplikacji dla systemu Windows to **Universal Windows Platform (UWP)**, zbudowana na specjalnej wersji .NET Core. Więcej informacji na jej temat znajdziesz w rozdziale 17. „Tworzenie aplikacji dla Windowsa przy użyciu XAML i Fluent Design”.

Biblioteki **ASP.NET Web Forms** i **Windows Communication Foundation (WCF)** to stare technologie tworzenia aplikacji WWW i serwisów, których dzisiaj używa już coraz mniej programistów, dlatego obie zostały usunięte z .NET Core. Programiści znacznie chętniej używają bibliotek **ASP.NET MVC** i **ASP.NET Web API**, dlatego obie zostały przebudowane i połączone w jeden produkt działający w ramach .NET Core pod nazwą **ASP.NET Core**. Więcej informacji na temat biblioteki **ASP.NET Core MVC** znajdziesz w rozdziale 15., „Tworzenie aplikacji WWW przy użyciu ASP.NET Core MVC”. Rozdział 14., „Tworzenie witryn WWW przy użyciu ASP.NET Core Razor Pages”, został poświęcony technologii **ASP.NET Core Razor Pages**. Z kolei technologie **ASP.NET Core Web API** i **Single Page Applications (SPAs)** będą opisywać w rozdziale 16., „Tworzenie usług i aplikacji WWW przy użyciu ASP.NET Core”.

Entity Framework 6 (EF) to technologia mapowania relacji obiektów, współpracująca z relacyjnymi bazami danych, takimi jak Oracle albo Microsoft SQL Server. Przez lata nabierała wiele obciążeń, dlatego jej nowa, wieloplatformowa wersja została odchudzona i otrzymała nazwę **Entity Framework Core**. Więcej informacji na ten temat znajdziesz w rozdziale 11., „Praca z bazami danych przy użyciu Entity Framework Core”.

Po usunięciu sporych elementów .NET Framework w celu utworzenia .NET Core Microsoft podzielił całość na komponenty i stworzył z nich pakiety NuGet. Powstały zatem małe wycinki z funkcjami, które można wykorzystywać niezależnie od siebie.

Ostatecznym celem Microsoftu nie jest zmniejszenie .NET Core względem .NET Framework. Jego celem jest podzielenie .NET Core na komponenty, które pozwalałyby na obsługę nowoczesnych technologii i tworzenie mniejszej liczby zależności. Dzięki temu instalowanie aplikacji wymagać będzie skopiowania tylko tych pakietów, z których faktycznie korzystają programiści.

Poznawanie .NET Standard

Obecnie platforma .NET została podzielona na trzy osobne platformy, a każda z nich jest kontrolowana przez Microsoft:

- .NET Framework,
- .NET Core,
- Xamarin.

Każdy z tych wariantów ma swoje mocne i słabe strony, ponieważ były one projektowane dla różnych zastosowań. W związku z tym każdy programista ma problem, ponieważ musi się uczyć zasad działania trzech platform i poznawać związane z każdą z nich ograniczenia.

W związku z tym Microsoft zdefiniował specyfikację .NET Standard 2.0, która jest zbiorem różnych API, jakie musi implementować każda platforma .NET. Nie da się nigdzie zainstalować .NET Standard 2.0, podobnie jak nie da się zainstalować HTML5. Aby skorzystać z HTML5, musisz zainstalować przeglądarkę, która będzie implementowała specyfikację HTML5. Podobnie aby użyć .NET Standard 2.0, musisz zainstalować platformę .NET, która będzie implementowała specyfikację .NET Standard 2.0.

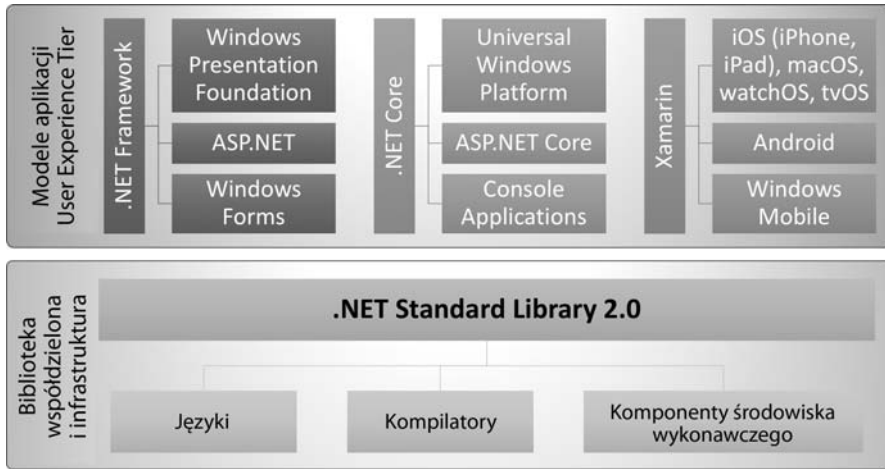
Specyfikacja .NET Standard 2.0 implementowana jest przez najnowsze wersje .NET Framework, .NET Core i Xamarin. Dzięki tej specyfikacji programiści znacznie łatwiej mogą przenosić kod pomiędzy poszczególnymi wariantami platformy .NET.

W przypadku .NET Core 2.0 spowodowało to dodanie wielu brakujących API, które programiści mogą wykorzystać do przenoszenia kodu przygotowanego dla .NET Framework na wieloplatformowy wariant .NET Core. Niestety, część z tych API została co prawda **zaimplementowana**, ale i tak rzuca wyjątki informujące programistę, że nie powinien z nich korzystać! Zazwyczaj wynika to z różnic w systemie operacyjnym, na którym uruchamiana jest .NET Core. W rozdziale 2., „Mówimy w C#”, dowiesz się, jak obsługiwać takie wyjątki.

Na poniższym rysunku widać, w jaki sposób wszystkie trzy warianty platformy .NET (czasami nazywane też modelami aplikacji) współdzielą ze sobą specyfikację .NET Standard 2.0 oraz całą infrastrukturę — zobacz rysunek na następnej stronie.

W pierwszym wydaniu tej książki koncentrowałem się na .NET Core, ale używałem też .NET Framework w przypadku, gdy ważne lub przydatne funkcje nie zostały jeszcze zaimplementowane w .NET Core. W większości przypadków używałem Visual Studio 2015, a Visual Studio Code prezentowałem jedynie pobieżnie.

W drugim wydaniu niemal całkowicie usunąłem wszystkie przykłady kodu dla .NET Framework.



Trzecie wydanie jest dopełnieniem całości. Większość książki została napisana od nowa, dzięki czemu całość kodu jest przystosowana do .NET Core i może być pisana w Visual Studio 2017, Visual Studio for Mac albo Visual Studio Code, i to niezależnie od systemu operacyjnego. Jedynymi wyjątkami są dwa ostatnie rozdziały. W rozdziale 17., „Tworzenie aplikacji dla Windowsa przy użyciu XAML i Fluent Design”, skorzystamy z .NET Core for UWP, co wymusi zastosowanie Visual Studio 2017 działającego w systemie Windows 10. Z kolei w rozdziale 18., „Tworzenie aplikacji mobilnych przy użyciu XAML i Xamarin.Forms”, zamiast .NET Core użyjemy wariantu Xamarin.

Poznanie .NET Native

Inną inicjatywą rozwijania platformy .NET jest .NET Native. Pozwala ona na skompilowanie kodu języka C# bezpośrednio do postaci instrukcji procesora, a nie do instrukcji języka pośredniego (ang. *Intermediate Language* — IL), które dopiero w czasie wykonywania przekładane są przez środowisko wykonawcze CLR na kod procesora.

.NET Native poprawia czasy wykonania kodu i zmniejsza zapotrzebowanie na pamięć aplikacji. Pozwala obsłużyć następujące technologie:

- aplikacje UWP dla Windows 10, Windows 10 Mobile, Xbox One, HoloLens oraz urządzeń **internetu rzeczy**, takich jak Raspberry Pi;
- aplikacje WWW po stronie serwera przy użyciu ASP.NET Core;
- aplikacje konsolowe używane w wierszu poleceń.

Porównanie technologii .NET

Poniższa tabela podsumowuje informacje o poszczególnych technologiach .NET:

Technologia	Zbiór funkcji	Kompilowane do	Systemy operacyjne
.NET Framework	nowoczesne i tradycyjne	kod IL	tylko Windows
Xamarin	wyłącznie mobilne	kod IL	iOS, Android, Windows Mobile
.NET Core	wyłącznie nowoczesne	kod IL	Windows, macOS, Linux
.NET Native	wyłącznie nowoczesne	kod procesora	Windows, macOS, Linux

Pisanie i kompilowanie kodu przy użyciu narzędzi wiersza poleceń z .NET Core

Po zainstalowaniu Visual Studio 2017, Visual Studio for Mac albo .NET Core SDK dostępne jest narzędzie wiersza poleceń o nazwie `dotnet` oraz środowisko wykonawcze .NET Core.

Zanim jednak skorzystamy z takich narzędzi jak `dotnet`, musimy napisać choć trochę kodu!

Pisanie kodu za pomocą prostego edytora tekstu

Jeżeli używasz systemu Windows, uruchom Notatnik.

Jeżeli używasz systemu macOS, uruchom TextEdit. W menu wybierz pozycję *TextEdit/Preferencje*, usuń zaznaczenie opcji *Cudzysłowy inteligentne*, a następnie zamknij okno dialogowe. W menu wybierz pozycję *Format/Zamień na tekst zwykły*.

Oczywiście zawsze możesz skorzystać ze swojego ulubionego edytora tekstu.

Wpisz poniższy kod:

```
class MyApp { static void Main() {
    System.Console.WriteLine("Witaj, C#!"); } }
```

Język C# rozróżnia wielkości liter, a to oznacza, że wielkie i małe litery musisz wpisać dokładnie tak, jak podałę w poprzednim przykładowym kodzie. Z drugiej strony, języka C# nie interesują znaki białe, a to oznacza, że nie ma znaczenia, czy użyjesz spacji, tabulacji czy znaków końca wiersza, żeby wprowadzić do kodu strukturę.

Całość kodu możesz wpisać do jednego wiersza albo rozdzielić go na kilka wierszy, każdemu z nich nadając odpowiednie wcięcie. Na przykład poniższy kod tak samo będzie można skompilować i da te same wyniki co poprzedni:

```
class
    MojaAplikacja    {
```

```

static void
Main() { System.Console.
    WriteLine("Witaj, C#!"); }

```

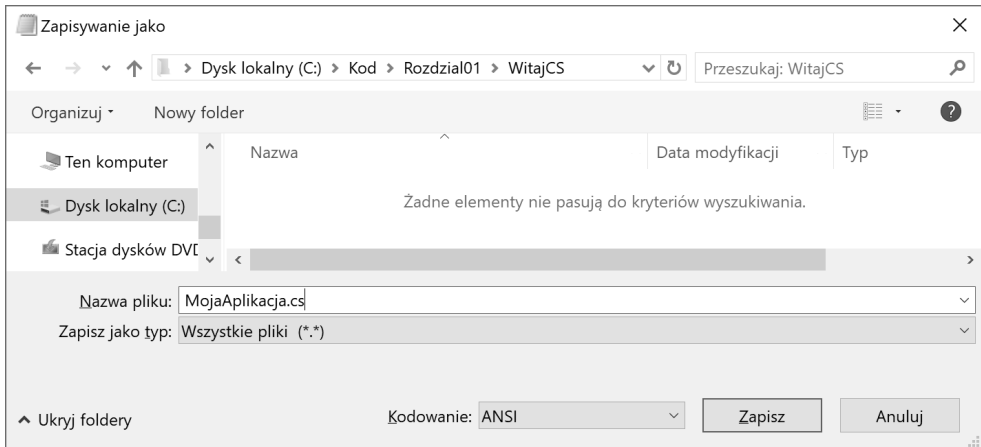
Oczywiście najlepiej jest tak pisać kod, żeby każdy inny programista (również Ty) mógł nawet po miesiącach lub latach bez problemu go odczytać!

Notatnik Windows

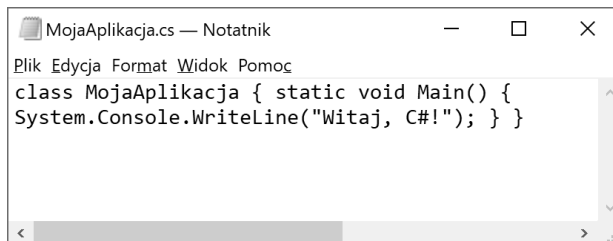
W Notatniku wybierz z menu pozycję *Plik/Zapisz jako...*

W oknie dialogowym *Zapisz jako* przejdź na dysk C: (albo dowolny inny, na którym chcesz zapisywać swoje projekty), kliknij przycisk *Nowy folder* i nadaj utworzonemu folderowi nazwę *Kod*. Otwórz utworzony właśnie folder i ponownie kliknij przycisk *Nowy folder*, a utworzonemu folderowi nadaj nazwę *Rozdzial01*. Otwórz ten folder, jeszcze raz kliknij przycisk *Nowy folder* i nazwij utworzony folder *WitajCS*. Na zakończenie otwórz folder *WitajCS*.

Z listy rozwijanej *Zapisz jako typ* wybierz opcję *Wszystkie pliki*, aby uniknąć dodania rozszerzenia *.txt* do tworzonego pliku. Wprowadź nazwę pliku *MojaAplikacja.cs*, tak jak pokazałem na poniższym rysunku:



Kod zapisany w Notatniku powinien wyglądać tak jak na poniższym rysunku:



Edytor TextEdit w systemie macOS

W edytorze TextEdit wybierz z menu *Plik/Zachowaj* albo naciśnij klawisze *Cmd+S*.

W oknie dialogowym *Zachowaj* przejdź do folderu użytkownika (mój nazywa się *markprice*) albo dowolnego innego, w którym chcesz zapisywać swoje projekty. Kliknij przycisk *Nowy katalog* i nadaj utworzonemu folderowi nazwę *Kod*. Otwórz utworzony właśnie folder i ponownie kliknij przycisk *Nowy katalog*, a utworzonemu folderowi nadaj nazwę *Rozdział01*. Otwórz ten folder, jeszcze raz kliknij przycisk *Nowy katalog* i nazwij utworzony folder *WitajCS*. Na zakończenie otwórz folder *WitajCS*.

W polu *Kodowanie tekstu zwykłego* wybierz z listy rozwijanej pozycję *Unicode (UTF-8)* i usuń zaznaczenie z opcji *W przypadku braku rozszerzenia, użyj ".txt"*, aby uniknąć dodania rozszerzenia *.txt* do tworzonego pliku. Wpisz nazwę pliku *MyApp.cs* i kliknij przycisk *Zachowaj*.

Tworzenie i kompilowanie aplikacji za pomocą narzędzi wiersza poleceń .NET Core

Jeżeli używasz systemu Windows, uruchom *Wiersz poleceń*.

Jeżeli używasz systemu macOS, uruchom *Terminal*.

W otwartym oknie wpisz polecenie *dotnet* i przyjrzyj się tekstom wypisanym na ekranie, pokazanym na poniższym rysunku:

```

markjprice -- -bash -- 54x11
[Marks-MBP-13:~ markjprice$ dotnet
Usage: dotnet [options]
Usage: dotnet [path-to-application]

Options:
  -h|--help           Display help.
  --version           Display version.

path-to-application:
  The path to an application .dll file to execute.

```

Tekst wyświetlany przez polecenie *dotnet* będzie taki sam w systemach Windows, macOS i Linux.

Tworzenie aplikacji konsolowej z wiersza poleceń

W wierszu poleceń wpisz poniższe polecenia, aby wykonać następujące operacje:

- Przejść do folderu z projektem.
- Utworzyć w tym folderze nową aplikację konsolową.

- Wypisać pliki, jakie utworzyło polecenie dotnet.

Jeżeli używasz systemu Windows, to w *Wierszu poleceń* wpisz następujące polecenia:

```
cd C:\Kod\Rozdzial01\WitajCS
dotnet new console
dir
```

Jeżeli używasz systemu macOS, to w *Terminalu* wprowadź następujące polecenia:

```
cd Kod/Rozdzial01/WitajCS
dotnet new console
ls
```

Polecenie dotnet powinno utworzyć dwa nowe pliki, takie jak widoczne na rysunku ze zrzutem ekranu z systemu Windows:

- *Program.cs* — kod źródłowy prostej aplikacji konsolowej.
- *WitajCS.csproj* — plik projektu zawierający listę zależności różnych ustawień samego projektu.

```

C:\Kod\Rozdzial01\WitajCS>dir
Volume in drive C has no label.
Volume Serial Number is F6B8-1D3C

Directory of C:\Kod\Rozdzial01\WitajCS

28.04.2018  13:27    <DIR>          .
28.04.2018  13:27    <DIR>          ..
28.04.2018  13:27                87 MojaAplikacja.cs
28.04.2018  13:27    <DIR>          obj
28.04.2018  13:27                189 Program.cs
28.04.2018  13:27                178 WitajCS.csproj
                3 File(s)              454 bytes
                3 Dir(s)  19 364 573 184 bytes free

C:\Kod\Rozdzial01\WitajCS>

```

W tym przykładzie musimy usunąć plik o nazwie *Program.cs*, ponieważ już wcześniej utworzyliśmy własny plik o nazwie *MojaAplikacja.cs*.

Jeżeli używasz systemu Windows, to w *Wierszu poleceń* wpisz następujące polecenie:

```
del Program.cs
```

Jeżeli używasz systemu macOS, to w *Terminalu* wpisz następujące polecenie:

```
rm Program.cs
```

We wszystkich kolejnych przykładach zamiast tworzyć własne pliki z kodem, będziemy korzystać z pliku *Program.cs* stworzonego przez polecenie `dotnet`.

Odtwarzanie pakietów, kompilowanie kodu i uruchamianie aplikacji

W wierszu poleceń wpisz polecenie `dotnet run`.

Po kilku sekundach wszystkie pakiety wymagane przez nasz kod zostaną pobrane, kod źródłowy zostanie skompilowany, a sama aplikacja zostanie uruchomiona. Wszystko to można zobaczyć na poniższym rysunku:

```

WitajCS — -bash — 80x24
[marks-iMac:WitajCS markjprice$ ls
MojaAplikacja.cs      WitajCS.csproj
Program.cs             obj
[marks-iMac:WitajCS markjprice$ rm Program.cs
[marks-iMac:WitajCS markjprice$ dotnet run
Witaj, C#!
marks-iMac:WitajCS markjprice$
  
```

Nasz kod źródłowy, czyli plik *MojaAplikacja.cs*, został skompilowany do pliku o nazwie *WitajCS.dll*, który umieszczony jest w podkatalogu *bin/Debug/netcoreapp2.0*. (Jeżeli masz ochotę, możesz przejrzeć zawartość struktury katalogów. Ja tutaj poczekam, aż skończysz).

Na razie taki plik może zostać uruchomiony wyłącznie przez polecenie `dotnet run`. W rozdziale 7., „Poznanie typów .NET Standard”, dowiesz się więcej na temat przygotowywania skompilowanego kodu do wykonania w dowolnym systemie operacyjnym współpracującym z .NET Core.

Naprawianie błędów kompilacji

Jeżeli kompilator wyświetli jakiegokolwiek błąd, to dokładnie przeczytaj ich treść, a następnie popraw je w edytorze tekstu. Zapisz wprowadzone zmiany i spróbuj ponownie.

W wierszu poleceń możesz skorzystać ze strzałek w górę i w dół, żeby przywoływać poprzednio wprowadzone polecenia.

Często występującymi błędami jest wpisanie litery o niewłaściwej wielkości, pominięcie średnika na końcu instrukcji albo niedopasowanie do siebie nawiasów klamrowych. Na przykład jeżeli zdarzy Ci się wpisać małą literę `m` w nazwie metody `Main()`, to zobaczysz poniższy komunikat o błędzie:

```
error CS5001: Program nie zawiera statycznej metody "Main" odpowiedniej jako punkt wejścia
```

Poznanie języka pośredniego

Kompilator języka C# (nazywany **Roslyn**) używany przez polecenie `dotnet` zajmuje się przekształcaniem kodu źródłowego C# na kod języka pośredniego (IL), który jest zapisywany w pliku o rozszerzeniu `.dll` lub `.exe`.

Polecenia języka IL przypominają instrukcje języka asemblera, ale są wykonywane przez maszynę wirtualną .NET Core, nazywaną **CoreCLR**.

W czasie pracy CoreCLR ładuje kod IL z pliku wykonywalnego, a następnie kompiluje go do postaci instrukcji procesora, na którym działa. Dopiero te instrukcje przekazywane są procesorowi do wykonania.

Zaletą takiej dwuetapowej kompilacji jest to, że Microsoft może przygotować maszyny wirtualne CLR nie tylko dla systemów Windows, ale i dla Linuksa oraz systemu macOS. Ten sam kod IL może działać wszędzie, ponieważ dopiero drugi etap kompilacji generuje kod właściwy dla danego systemu operacyjnego i procesora.

Niezależnie od tego, w jakim języku napisany jest kod źródłowy (czy będzie to C#, czy też F#), wszystkie aplikacje działające w środowisku .NET mają zapisane w plikach wykonywalnych instrukcje języka IL. Microsoft i inni producenci udostępniają deasemblery, które odczytują pliki wykonywalne i podają zawarty w nich kod IL.

Tak naprawdę nie wszystkie aplikacje .NET używają kodu IL! Niektóre z nich używają kompilatora .NET Native, który generuje bezpośrednio kod procesora, a nie kod IL. Poprawia to wydajność aplikacji i zmniejsza zapotrzebowanie na pamięć, ale za cenę przenośności kodu.

Pisanie i kompilowanie kodu za pomocą Visual Studio 2017

Przygotujemy teraz podobną aplikację, wykorzystując Visual Studio 2017. Nawet jeżeli wolisz używać Visual Studio for Mac albo Visual Studio Code, to i tak zachęcam do przejrzania tego podrozdziału i zapoznania się z rysunkami, ponieważ oba warianty środowiska programistycznego mają bardzo podobne, choć nie aż tak rozbudowane funkcje.

Od ponad dekady nauczam ludzi stosowania Visual Studio przy pracy i nadal zaskakuje mnie, że programiści nie umieją wykorzystać potencjału tego narzędzia.

Na kolejnych stronach nauczę Cię wprowadzania kodu wiersz po wierszu. Może się to wydawać niepotrzebne, ale z pewnością warto się dowiedzieć, jaką pomoc i jakie informacje udostępnia

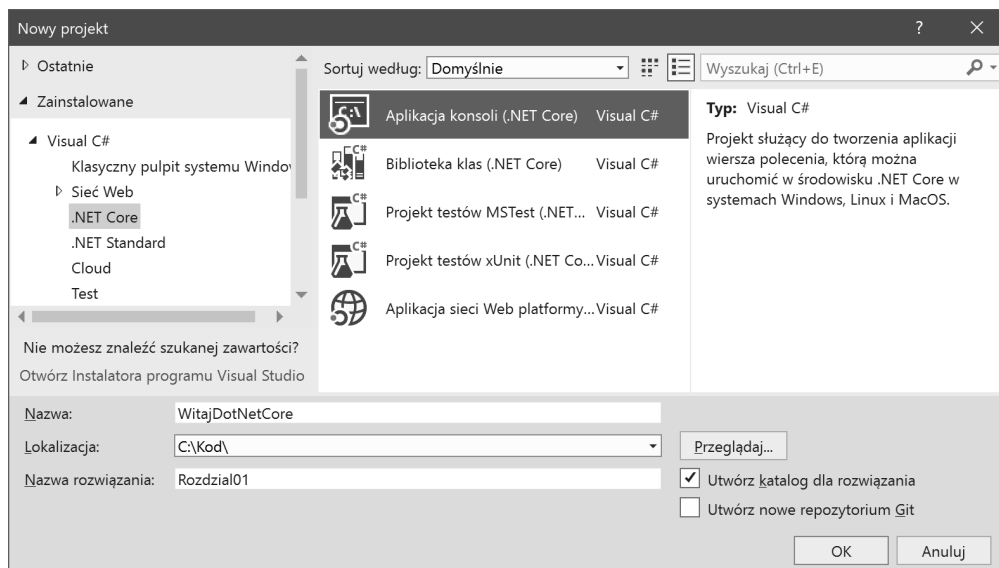
nam Visual Studio w czasie, gdy wprowadzamy kod. Jeżeli chcesz się nauczyć szybko i bezbłędnie wprowadzać swój kod, to bardzo wiele zyskasz, pozwalając Visual Studio na samodzielne napisanie większości kodu.

Pisanie kodu w Visual Studio 2017

Uruchom Visual Studio 2017.

Wybierz z menu pozycję *Plik/Nowy/Projekt...* albo naciśnij klawisze *Ctrl+Shift+N*.

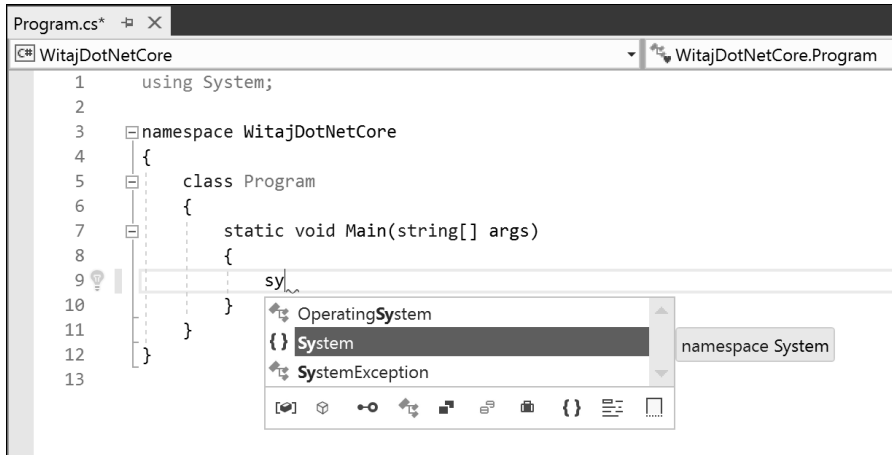
Ze znajdującej się po lewej stronie listy *Zainstalowane* rozwiń pozycję *Visual C#* i wybierz opcję *.NET Core*. Z listy na środku wybierz opcję *Aplikacja konsoli (.NET Core)*. Wprowadź nazwę projektu — *WitajDotNetCore*, jako lokalizację wybierz folder *C:\Kod*, a jako nazwę rozwiązania wybierz *Rozdział01*. Na zakończenie kliknij przycisk *OK* albo naciśnij klawisz *Enter*, tak jak na poniższym rysunku:



W edytorze kodu usuń znajdującą się w wierszu 9. instrukcję o treści: `Console.WriteLine("Witaj, świecie!");`

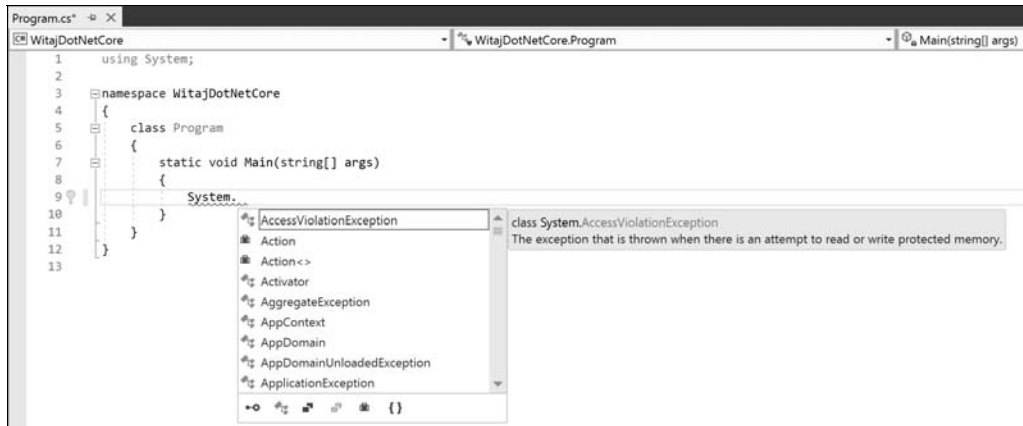
W metodzie `Main` wpisz litery `sy`, tak jak na rysunku na następnej stronie i zwróć uwagę na to, że pojawi się okienko usługi IntelliSense:

Usługa IntelliSense wyświetla filtrowaną listę **słów kluczowych**, **przestrzeni nazw** oraz **typów**, które zawierają litery `sy`, i wyróżnia na niej pozycję zaczynającą się od tych liter. W tym przypadku jest to przestrzeń nazw, której poszukujemy — `System`.



Wpisz kropkę.

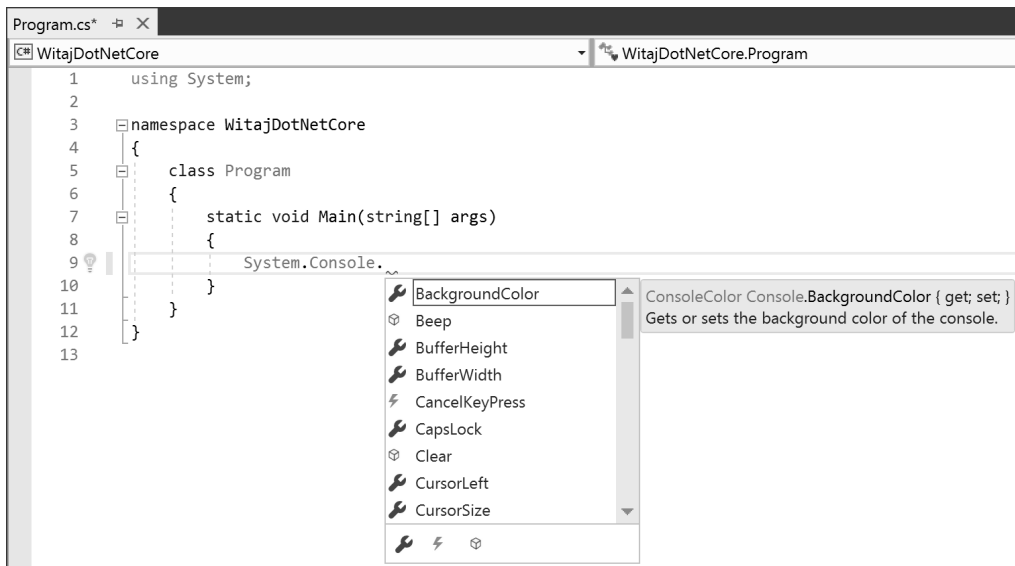
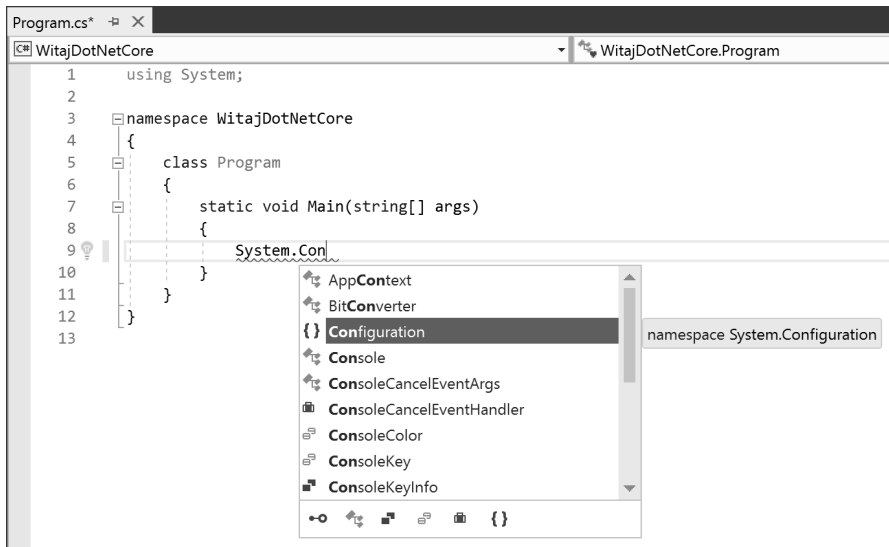
IntelliSense automatycznie uzupełni w kodzie słowo System, wprowadzi kropkę i wyświetli listę typów, takich jak AggregateException lub Action, które są dostępne w przestrzeni nazw System, tak jak na poniższym rysunku:



Wpisz litery con, a IntelliSense wyświetli listę pasujących typów i przestrzeni nazw, tak jak na pierwszym rysunku na następnym stronie.

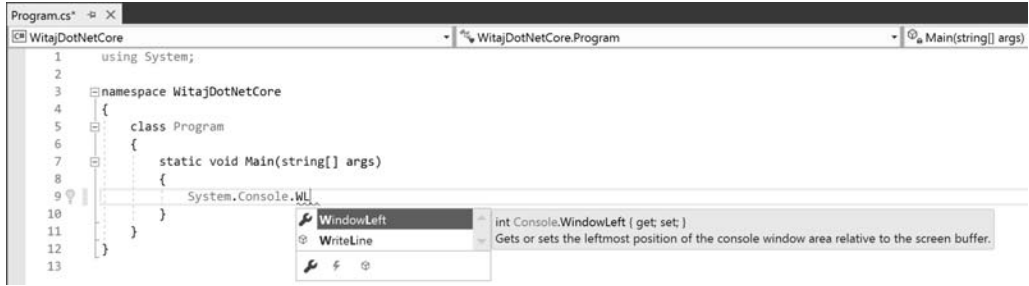
Poszukujemy tu słowa Console. Wciśnij na klawiaturze strzałkę w dół, żeby je wybrać. Po zaznaczeniu na liście słowa Console wpisz kropkę.

IntelliSense wyświetli teraz listę elementów klasy Console, taką jak na drugim rysunku na następnej stronie.



Elementami klasy są **właściwości** (atrybuty obiektu, takie jak `BackgroundColor`), **metody** (akcje, jakie może wykonać obiekt, takie jak `Beep`), **zdarzenia** i inne.

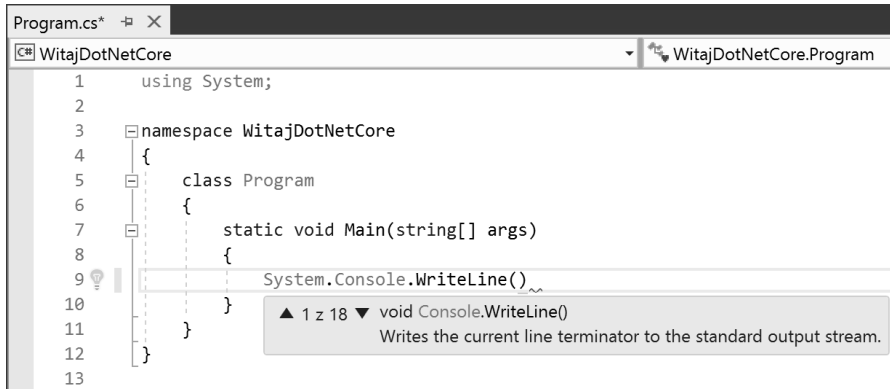
Wpisz litery `wl`. IntelliSense wyświetli teraz pasujące elementy klasy, czyli takie, w których nazwie słowa zaczynają się od podanych liter. W tym przypadku będą to `WindowsLeft` i `WriteLine`, co można zobaczyć na rysunku na następnej stronie.



Użyj strzałki w dół, żeby zaznaczyć na liście pozycję `WriteLine`, i wprowadź nawias otwierający (`(`).

IntelliSense samodzielnie wpisze nazwę metody `WriteLine` i doda za nią nawiasy okrągłe.

Pojawi się też pole wskazówki informujące, że metoda `WriteLine` ma 18 różnych wariantów, takie jak na poniższym rysunku:



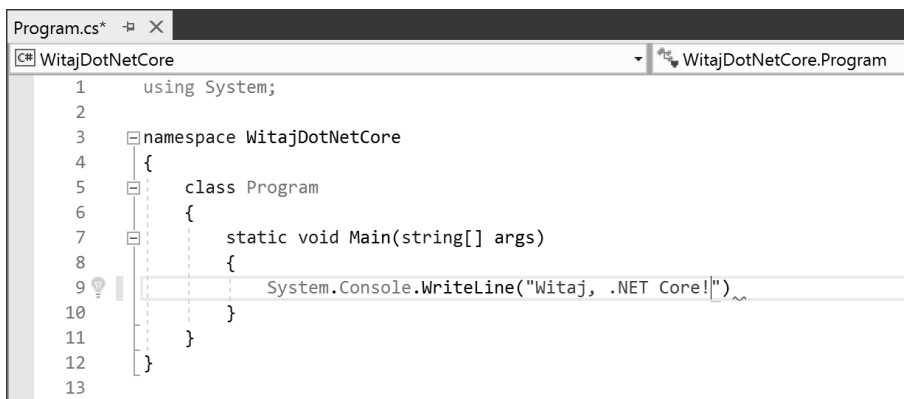
Wprowadź znak cudzysłowu (`"`). IntelliSense wpisze do programu dwa takie znaki i umieści kursor pomiędzy nimi.

Wpisz tekst `Witaj, .NET Core!`, tak jak na rysunku na następnej stronie.

Czerwone podkreślenie na końcu wiersza oznacza błąd, ponieważ w języku C# każda instrukcja musi zostać zakończona znakiem średnika. Przenieś kursor na koniec wiersza i wpisz znak średnika, żeby naprawić ten błąd.

Kompilowanie kodu za pomocą Visual Studio 2017

Wybierz z menu pozycję *Debugowanie/Uruchom bez debugowania* albo naciśnij klawisze `Ctrl+F5`.



```

1  using System;
2
3  namespace WitajDotNetCore
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              System.Console.WriteLine("Witaj, .NET Core!");
10         }
11     }
12 }
13

```

Na pasku statusu Visual Studio pojawi się komunikat *Kompilacja rozpoczęta...*, a chwilę później — *Kompilacja zakończona sukcesem*. Zaraz potem uruchomiona zostanie aplikacja i otworzone okno konsoli, takie jak na poniższym rysunku:



```

C:\Program Files\dotnet\dotnet.exe
Witaj, .NET Core!

```

Aby oszczędzać miejsce w tej książce i poprawić czytelność, nie będę już zamieszczał zrzutów ekranu aplikacji konsolowych, tak jak zrobiłem to powyżej. Zamiast tego będę prezentował wyniki ich pracy w ten sposób:

```
Witaj, .NET Core!
```

Poprawianie pomyłek z listy błędów

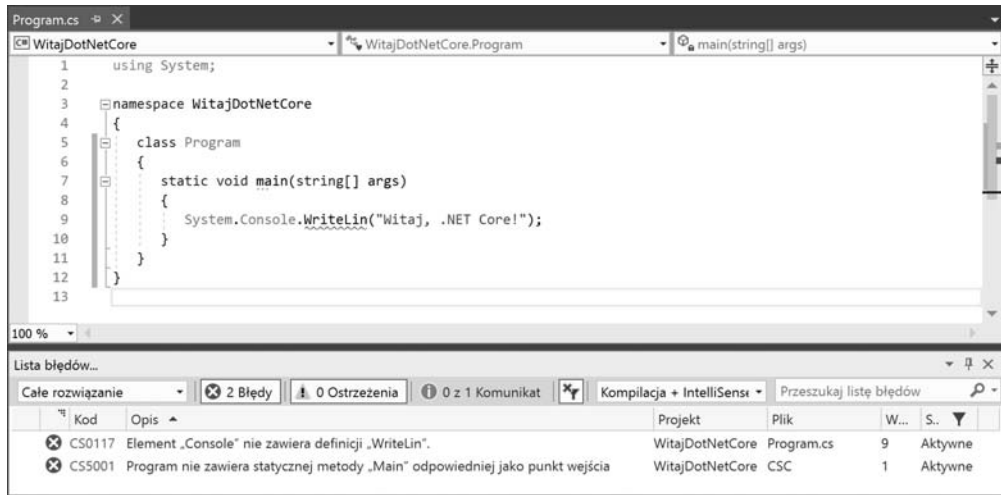
Wprowadźmy teraz do kodu dwa błędy:

- Zmień wielką literę *M* w nazwie funkcji *Main* na małą literę *m*.
- Usuń literę *e* z końca nazwy funkcji *WriteLine*.

Wybierz z menu pozycję *Debugowanie/Uruchom bez debugowania* albo naciśnij klawisze *Ctrl+F5*.

Po kilku sekundach na pasku stanu zostanie wyświetlony komunikat *Kompilacja nie powiodła się...* i pojawi się okienko z informacją o błędzie, w którym należy kliknąć przycisk *Nie*.

Pojawi się wtedy panel *Lista błędów*, taki jak na poniższym rysunku:



Listę można filtrować za pomocą przycisków w górnej części panelu *Lista błędów*, tak żeby pojawiały się na niej lub znikaly z niej *Błędy*, *Ostrzeżenia* oraz *Komunikaty*.

Jeżeli przy komunikacie o błędzie pojawia się nazwa pliku i numer wiersza, np. *Plik: Program.cs* i *Wiersz: 9*, to możesz kliknąć dwukrotnie ten błąd, aby od razu przejść do odpowiedniego miejsca w programie.

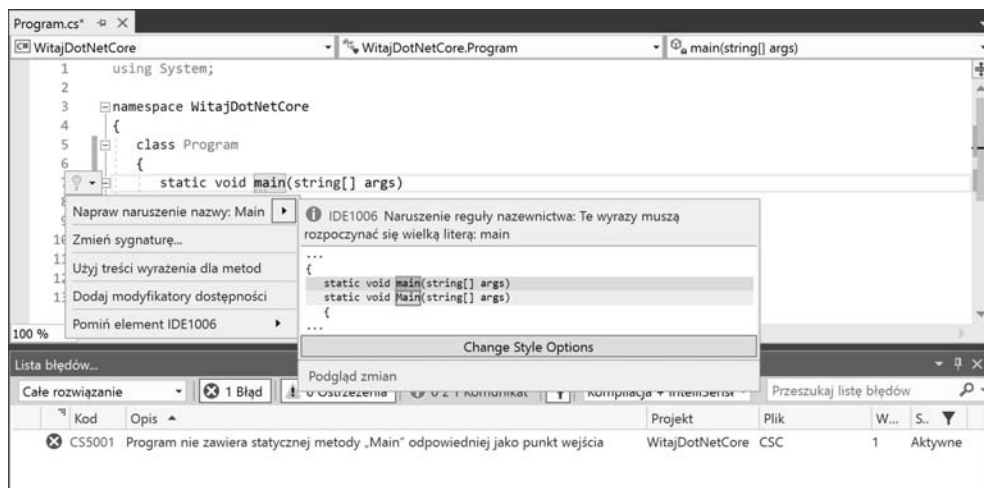
Jeżeli jest to bardziej ogólny błąd, taki jak brakująca metoda `Main`, to kompilator nie jest w stanie podać konkretnego numeru wiersza. W swoim programie możesz przecież chcieć mieć metodę `main`, która będzie całkowicie niezależna od metody `Main`. Pamiętaj, że język `C#` rozróżnia wielkość liter, więc takie nazwy metod są dozwolone.

Visual Studio może jednak analizować tworzony kod i podawać różne propozycje, prezentowane jako trzy szare kropki w miejscu wykrzyknika potencjalnych problemów. Po kliknięciu tak wyróżnionej instrukcji pojawi się ikona z żarówką, a po kliknięciu tej ikony wyświetlona zostanie lista propozycji. Mogą to być wskazówki mówiące, że nazwy metod powinny się zaczynać od wielkiej litery, takie jak na rysunku na następnej stronie.

Popraw oba błędy tak jak na powyższym rysunku i ponownie uruchom aplikację, upewniając się, że działa ona prawidłowo. Zauważ, że *Lista błędów* jest teraz pusta.

Dodawanie istniejących projektów do Visual Studio 2017

Wcześniej w tym rozdziale utworzyliśmy już jeden projekt, używając do tego narzędzia `dotnet`. Teraz mamy otwarte rozwiązanie w Visual Studio 2017, do którego możemy dodać utworzony wcześniej projekt.



Wybierz z menu pozycję *Plik/Dodaj/Istniejący projekt...*, w otwartym oknie dialogowym przejdź do folderu *C:\Kod\Rozdzial01\WitajCS*, a w nim wybierz plik *WitajCS.csproj*.

Aby uruchomić ten projekt, musisz w *Eksploratorze rozwiązań* kliknąć prawym przyciskiem myszy pozycję *Rozwiązanie „Rozdzial01” (2 projekty)* i z menu kontekstowego wybrać pozycję *Właściwości* albo nacisnąć klawisze *Alt+Enter*.

Dla opcji *Projekt startowy* wybierz wartość *Bieżące zaznaczenie* i kliknij przycisk *OK*.

W *Eksploratorze rozwiązań* kliknij dowolny plik poza projektem *WitajCS*, a następnie naciśnij klawisze *Ctrl+F5* albo wybierz z menu pozycję *Debugowanie/Uruchom bez debugowania*.

Automatyczne formatowanie kodu

Kod daje się znacznie lepiej czytać i jest bardziej zrozumiały, jeżeli zastosujesz w nim odpowiednie wcięcia i odstępy.

Jeżeli kod daje się skompilować, to Visual Studio 2017 zacznie go automatycznie formatować, wprowadzając jednolite wcięcia i odstępy.

W *Eksploratorze rozwiązań* kliknij dwukrotnie plik *MojaAplikacja.cs*, tak jak na pierwszym rysunku na następnej stronie.

Z menu wybierz pozycję *Kompilowanie/Kompiluj WitajCS* albo naciśnij klawisze *Shift+F6* i poczekaj na zakończenie kompilacji. Następnie wybierz z menu pozycję *Edycja/Zaawansowane/Formuj dokument* albo naciśnij klawisze *Ctrl+E, D*. Twój kod zostanie automatycznie sformatowany, tak jak na drugim rysunku na następnej stronie.



Eksperymentowanie z interaktywnym C#

Co prawda Visual Studio od zawsze dysponowało oknem *Bezpośrednie* z obsługą funkcji **REPL** (ang. *Read-Eval-Print-Loop* — pętla czytaj-wykonaj-wypisz), ale w Visual Studio 2017 dodano bardziej rozbudowane okno *C# Interactive*, w którym działa IntelliSense oraz kolorowanie składni.

W Visual Studio 2017 wybierz z menu pozycję *Widok/Inne okna/C# Interactive*.

Napišemy teraz krótki interaktywny kod pobierający stronę *About* z witryny WWW Microsoftu.

To jest tylko przykład. Nie musisz jeszcze w szczegółach rozumieć tego kodu.

W oknie *C# Interactive* będziemy wprowadzać instrukcje wykonujące następujące operacje:

- Odwołanie do zestawu `System.Net.Http`.
- Zaimportowanie przestrzeni nazw `System.Net.Http`.
- Zadeklarowanie i utworzenie zmiennej klienta HTTP.
- Ustalenie podstawowego adresu strony WWW Microsoftu.
- Asynchroniczne oczekiwanie na odpowiedź na żądanie typu GET zawierające kod strony *About*.
- Odczytanie kodu stanu przesłanego przez serwer WWW.

- Odczytanie zawartości nagłówka content-type.
- Odczytanie zawartości strony HTML jako ciągu znaków.

Wpisz poniższe polecenia w wierszu ze znakiem > i naciśnij klawisz *Enter*:

```
> #r "System.Net.Http"
> using System.Net.Http;
> var client = new HttpClient();
> client.BaseAddress = new Uri("http://www.microsoft.com/");
> var response = await client.GetAsync("about");
> response.StatusCode
OK
> response.Content.Headers.GetValues("Content-Type")
string[1] { "text/html" }
> await response.Content.ReadAsStringAsync()
"<!DOCTYPE html ><html
xmlns:mscom="http://schemas.microsoft.com/CMSvNext"
xmlns:md="http://schemas.microsoft.com/mscom-data" lang="en"
xmlns="http://www.w3.org/1999/xhtml"><head><meta http-equiv="X-UA-Compatible"
content="IE=edge" /><meta charset="utf-8" /><meta name="viewport"
↳content="width=device-width, initial-scale=1.0"/><link rel="shortcut icon"
↳href="//www.microsoft.com/favicon.ico?v2" /><script type="text/javascript"
↳src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.7.2.min.js">rn // Third party
↳scripts and code linked to or referenced from this website are licensed to you by
↳the parties that own such code, not by Microsoft. See ASP.NET Ajax CDN Terms of
↳Use - http://www.asp.net/ajaxlibrary/CN.DN.ashx.rn </script><script type="
↳text/javascript" language="javascript">/*! [CDATA[*if($ (document).bind
↳("mobileinit",function(){$.mobile.autoInitializePage=!1}),navigator.userAgent.
↳match(/IEMobile\\10\\.0/)){$varmsViewportStyle=document.createElement("style ...
```

Na poniższym rysunku widać, jak powinno wyglądać Visual Studio 2017 po wprowadzeniu powyższych poleceń w oknie *C# Interactive*:

```
C# Interactive
> #r "System.Net.Http"
> using System.Net.Http;
> var client = new HttpClient();
> client.BaseAddress = new Uri("http://www.microsoft.com/");
> var response = await client.GetAsync("about");
> response.StatusCode
OK
> response.Content.Headers.GetValues("Content-Type")
string[1] { "text/html" }
> await response.Content.ReadAsStringAsync()
"<!DOCTYPE html ><html xmlns:mscom="http://schemas.microsoft.com/CMSvNext\" xml
> |
```














Roslyn to nazwa kompilatora języka C#. Roslyn w wersji 1.0 obsługiwał język C# 6, w wersji 2.0 związany był z językiem C# 7, natomiast wersja 2.3 i wyższe to język C# 7.1.

Inne przydatne okna

W Visual Studio 2017 znajdziesz też wiele innych przydatnych okien, np. takie:

- *Eksplorator rozwiązań* pozwala na zarządzanie projektami i plikami.
- *Team Explorer* umożliwia pracę z narzędziami do zarządzania kodem źródłowym.
- *Eksplorator serwera* daje możliwość obsługiwanienia połączeń z bazami danych i zarządzania zasobami w usłudze Microsoft Azure.

Jeżeli nie widzisz potrzebnego Ci okna, to otwórz menu *Widok* i wybierz je z listy menu. Spróbuj zapamiętać przypisany mu skrót klawiszowy. Przykłady takich skrótów zobaczysz na poniższym rysunku:

Widok	Projekt	Kompilowanie	Debugowanie	Zespół
	Eksplorator rozwiązań			Ctrl+W, S
	Team Explorer			Ctrl+\, Ctrl+M
	Eksplorator serwera			Ctrl+W, L
	Cloud Explorer			Ctrl+\, Ctrl+X
	SQL Server Object Explorer			Ctrl+\, Ctrl+S
	Eksplorator programu cookiecutter			
	Hierarchia wywołań			Ctrl+W, K
	Widok klas			Ctrl+W, C
	Okno definicji kodu			Ctrl+W, D
	Przeglądarka obiektów			Ctrl+W, J
	Lista błędów			Ctrl+W, E
	Dane wyjściowe			Ctrl+W, O

Jeżeli skróty klawiszowe w Twoim systemie są inne niż przedstawione na powyższym rysunku, to znaczy, że w czasie instalowania Visual Studio został wybrany inny zestaw ustawień. Wszystkie skróty klawiszowe możesz zmienić tak, żeby były zgodne z prezentowanymi w tej książce. Wystarczy wybrać z menu pozycję *Narzędzia/Import i eksport ustawień...*, w wyświetlonym oknie dialogowym zaznaczyć opcję *Resetuj wszystkie ustawienia*, a w następnym kroku wybrać zbiór ustawień *Visual C#*.

Pisanie i kompilowanie kodu w Visual Studio Code

Instrukcje i zrzuty ekranów z tego podrozdziału pochodzą z systemu macOS, ale te same operacje można wykonywać w Visual Studio Code działającym w systemach Windows i Linux. Główne różnice będą polegały na wykonywaniu działań w wierszu poleceń systemu operacyjnego, np. usuwaniu plików. W takich przypadkach zarówno polecenia, jak i ścieżki z całą pewnością będą się różniły. Z kolei polecenie `dotnet` działa tak samo w wierszu poleceń wszystkich systemów operacyjnych.

Pisanie kodu w Visual Studio Code

Uruchom Visual Studio Code.

W menu wybierz pozycję *File/Open...* albo naciśnij klawisze *Cmd+O*.

W oknie dialogowym otwórz folder *Kod*, następnie wybierz folder *Rozdzial01* i kliknij przycisk *New Folder*, wpisz nazwę folderu — *WitajDotNetCore* i kliknij przycisk *Create*. Wybierz folder *WitajDotNetCore* i kliknij przycisk *Open* albo naciśnij klawisz *Enter*.

W Visual Studio Code wybierz z menu pozycję *View/Integrated Terminal* albo naciśnij klawisze *Ctrl+`*.

W oknie *Terminal* wpisz następujące polecenie:

```
dotnet new console
```

Zobaczysz, że polecenie `dotnet` tworzy w aktualnym folderze nowy projekt aplikacji konsoli, a w oknie *Explorer* pojawia się lista utworzonych plików. Całość wygląda tak jak na pierwszym rysunku na następnej stronie.

W oknie *EXPLORER* kliknij plik *Program.cs*, aby otworzyć go w oknie edytora.

Jeżeli zobaczysz komunikat mówiący o brakujących elementach, taki jak na drugim rysunku na następnej stronie, to kliknij przycisk *Yes*.

Zmień tekst, który ma zostać wypisany w konsoli, na *Witaj, .NET Core!*

W menu wybierz pozycję *File/Auto Save*. Dzięki temu nie trzeba będzie pamiętać o zapisywaniu kodu programu przed uruchomieniem kompilacji.

```

Welcome - WitajDotNetCore - Visual Studio Code
File Edit Selection View Go Debug Tasks Help

EXPLORER
  OPEN EDITORS
    Welcome
  WITAJDOTNETCORE
    bin
    obj
    Program.cs
    WitajDotNetCore.csproj

Start
New file
Open folder...
Add workspace folder...

Customize
Tools and languages
Install support for JavaScript, TypeScript, Python, PH...

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: powershell + - ^ □ ×

PS C:\Kod\Rozdzial01\WitajDotNetCore> dotnet new console
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\Kod\Rozdzial01\WitajDotNetCore\WitajDotNetCore.csproj...
  Restoring packages for C:\Kod\Rozdzial01\WitajDotNetCore\WitajDotNetCore.csproj...
    Generating MSBuild file C:\Kod\Rozdzial01\WitajDotNetCore\obj\WitajDotNetCore.csproj.nuget.g.props.
    Generating MSBuild file C:\Kod\Rozdzial01\WitajDotNetCore\obj\WitajDotNetCore.csproj.nuget.targets.
  Restore completed in 263.19 ms for C:\Kod\Rozdzial01\WitajDotNetCore\WitajDotNetCore.csproj.

Restore succeeded.

PS C:\Kod\Rozdzial01\WitajDotNetCore>

```

```

Program.cs - WitajDotNetCore - Visual Studio Code
File Edit Selection View Go Debug Tasks Help

EXPLORER
  OPEN EDITORS
    Welcome
    Program.cs
  WITAJDOTNETCORE
    bin
    obj
    Program.cs
    WitajDotNetCore.csproj

Warn Required assets to build and debug are missing from 'WitajDotNetCore'. Add them. Don't Ask Again Not Now Yes

1 using System;
2
3 namespace WitajDotNetCore
4 {
5     0 references
6     class Program
7     {
8         0 references
9         static void Main(string[] args)
10        {
11            Console.WriteLine("Hello World!");
12        }
13    }
14 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: powershell + - ^ □ ×

Generating MSBuild file C:\Kod\Rozdzial01\WitajDotNetCore\obj\WitajDotNetCore.csproj.nuget.g.targets.
Restore completed in 263.19 ms for C:\Kod\Rozdzial01\WitajDotNetCore\WitajDotNetCore.csproj.

Restore succeeded.

PS C:\Kod\Rozdzial01\WitajDotNetCore>
Ln 1, Col 1 Spaces: 4 UTF-8 with BOM CRLF C# WitajDotNetCore

```

Kompilowanie kodu w Visual Studio Code

Wybierz z menu pozycję *View/Integrated Terminal* albo naciśnij klawisze *Ctrl+`*, a następnie wprowadź poniższe polecenie:

```
dotnet run
```

Tekst, który pojawi się w oknie *TERMINAL*, będzie przedstawiał wyniki działania uruchomionej aplikacji.

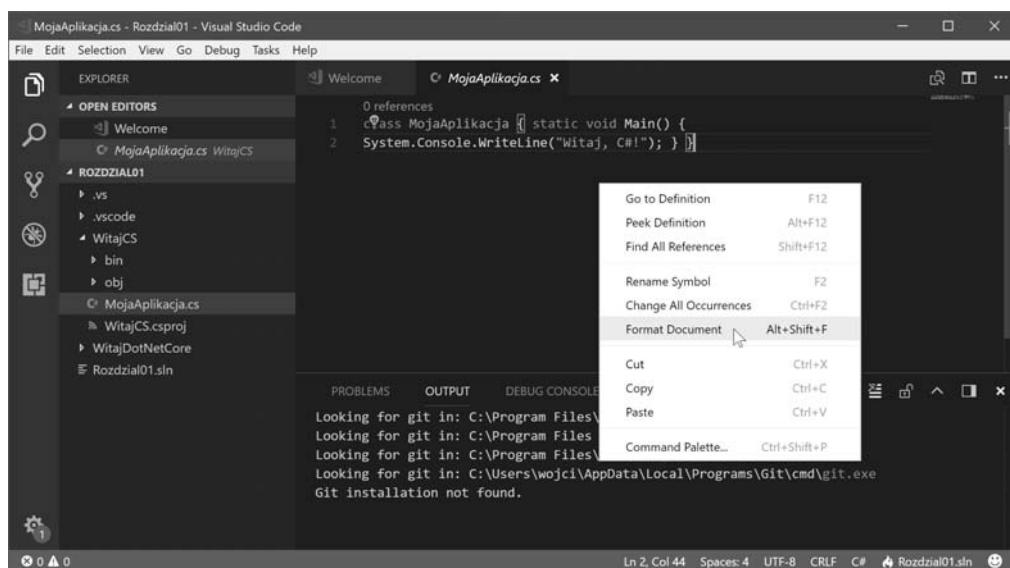
Automatyczne formatowanie kodu

W Visual Studio Code wybierz z menu pozycję *File/Open*, a następnie otwórz folder *Rozdzial01*.

W oknie *Explorer* rozwiń projekt *WitajCS* i zaznacz plik *MyApp.cs*.

Kliknij przycisk *Yes*, jeżeli pojawi się komunikat o konieczności dołączenia wymaganych elementów.

W oknie Visual Studio Code kliknij prawym przyciskiem myszy okno z kodem i wybierz z menu kontekstowego pozycję *Format Document* albo naciśnij klawisze *Alt+Shift+F*, tak jak na poniższym rysunku:



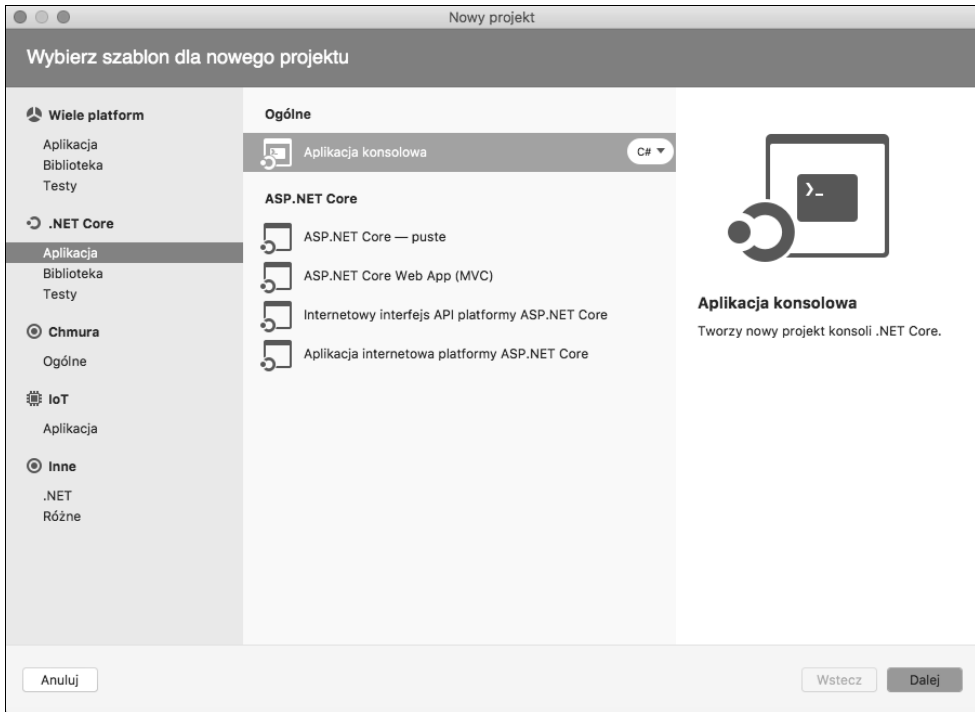
Visual Studio Code jest szybko rozwijającą się alternatywą dla Visual Studio 2017 dla Windowsa.

Pisanie i kompilowanie kodu za pomocą Visual Studio for Mac

Uruchom Visual Studio for Mac i wybierz z menu pozycję *Plik/Nowy projekt*.

Z listy po lewej stronie w sekcji *.NET Core* wybierz pozycję *Aplikacja*.

Z listy szablonów projektów znajdującej się na środku okna wybierz pozycję *Aplikacja konsolowa*, a następnie kliknij przycisk *Dalej*, tak jak na poniższym rysunku:



W kroku *Konfiguruj nowy element Aplikacja konsolowa* z listy rozwijanej *Framework* wybierz opcję *.NET Core 2.0* i kliknij przycisk *Dalej*.

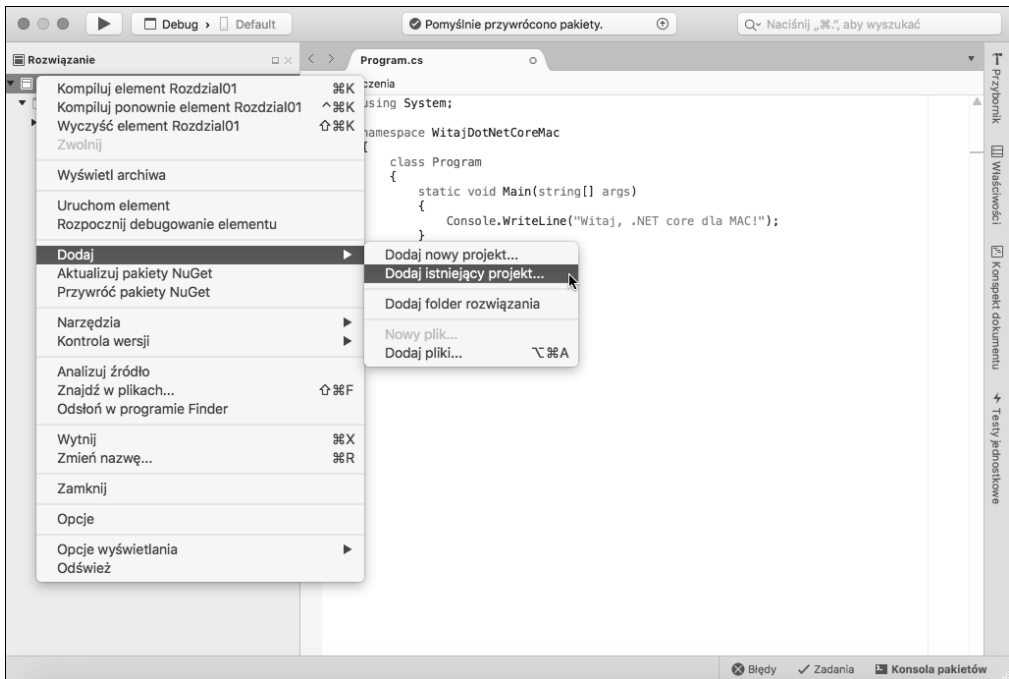
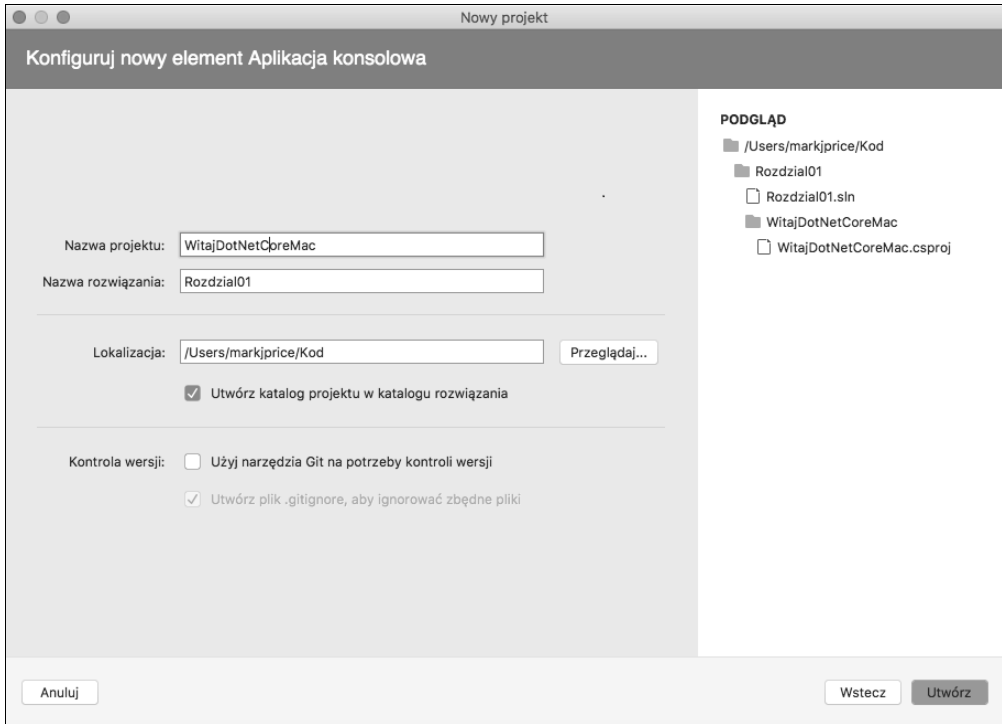
W następnym kroku *Konfiguruj nowy element Aplikacja konsolowa* wprowadź nazwę projektu — *WitajDotNetCoreMac*, jako *Nazwa rozwiązania* wpisz *Rozdział01*, a w polu *Lokalizacja* podaj folder *Kod*, tak jak na pierwszym rysunku na następnej stronie. Na koniec kliknij przycisk *Utwórz*.

Zmień tekst, który ma zostać wypisany w konsoli, np. na *Witaj, .NET Core dla Mac!*

W Visual Studio for Mac wybierz z menu pozycję *Uruchamianie/Uruchom bez debugowania* albo naciśnij klawisze *Cmd + Option + Enter*.

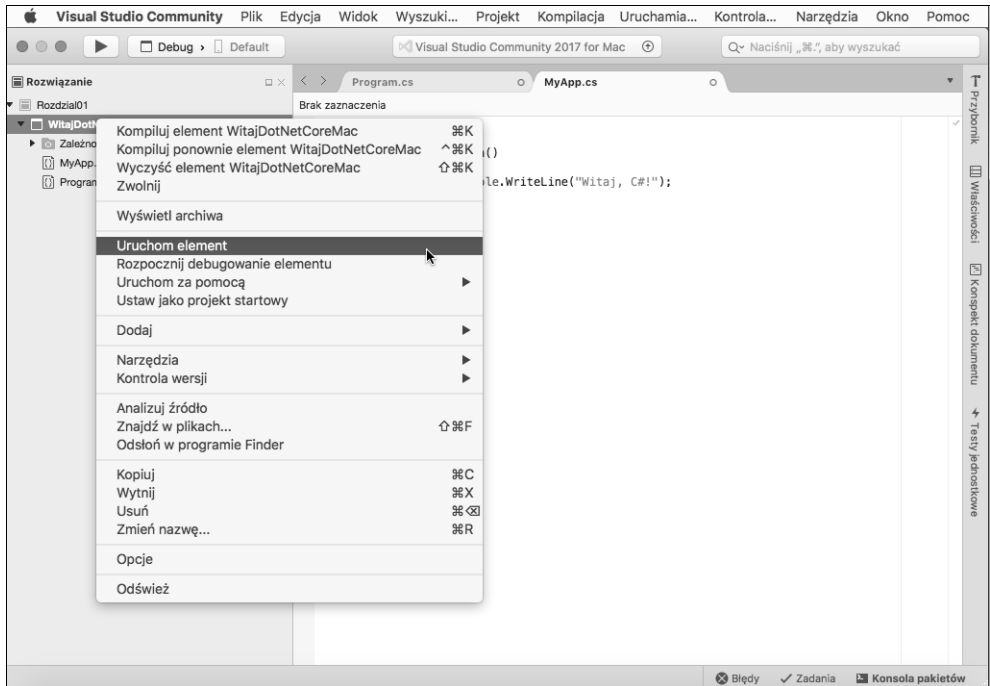
Tekst, który pojawi się w oknie *Terminal*, będzie wynikiem działania naszej aplikacji.

W panelu *Rozwiązanie* kliknij prawym przyciskiem myszy pozycję *Rozdział01*, a następnie wybierz z menu kontekstowego pozycję *Dodaj/Dodaj istniejący projekt...*, tak jak na drugim rysunku na następnej stronie.



W folderze *WitajCS* wybierz plik *WitajCS.cs.proj*.

W panelu *Rozwiązanie* kliknij prawym przyciskiem myszy projekt *WitajCS*, a z menu kontekstowego wybierz pozycję *Uruchom element*, tak jak na poniższym rysunku:



Następne kroki

Teraz już wiesz, jak tworzyć i kompilować proste aplikacje środowiska .NET Core w systemach Windows i macOS (praca w Linuksie jest równie prosta).

Będziesz w stanie wykonać prawie wszystkie zadania z kolejnych rozdziałów tej książki, korzystając z Visual Studio 2017 w systemie Windows, z Visual Studio for Mac albo z Visual Studio Code w systemach Windows, macOS lub Linux.

Zarządzanie kodem źródłowym przy użyciu platformy GitHub

Git to powszechnie stosowany system zarządzania kodem źródłowym. **GitHub** to firma, witryna WWW oraz aplikacja, która ułatwia korzystanie z systemu Git.

Używałem platformy GitHub do przechowywania rozwiązań wszystkich praktycznych ćwiczeń, jakie zamieszczam na końcu każdego z rozdziałów. Możesz je przejrzeć i pobrać pod poniższym adresem:

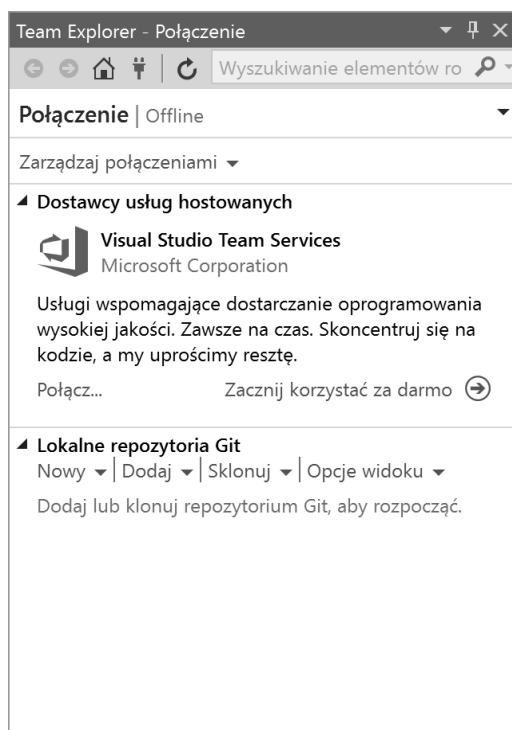
<https://github.com/markjprice/cs7dotnetcore2>.

Używanie systemu Git w Visual Studio 2017

Visual Studio 2017 ma wbudowane mechanizmy obsługi systemu Git i platformy GitHub, jak również własnego systemu Microsoftu o nazwie **Visual Studio Team Services (VSTS)**.

Używanie okna Team Explorer

W Visual Studio 2017 wybierz z menu pozycję *Widok/Team Explorer*, żeby wyświetlić okno przedstawione na poniższym rysunku:



Oczywiście zachęcam Cię do utworzenia własnego konta w sieciowym systemie zarządzania kodem źródłowym, choć możesz też pobrać zawartość mojego repozytorium GitHub bez zakładania konta.

Klonowanie repozytorium GitHub

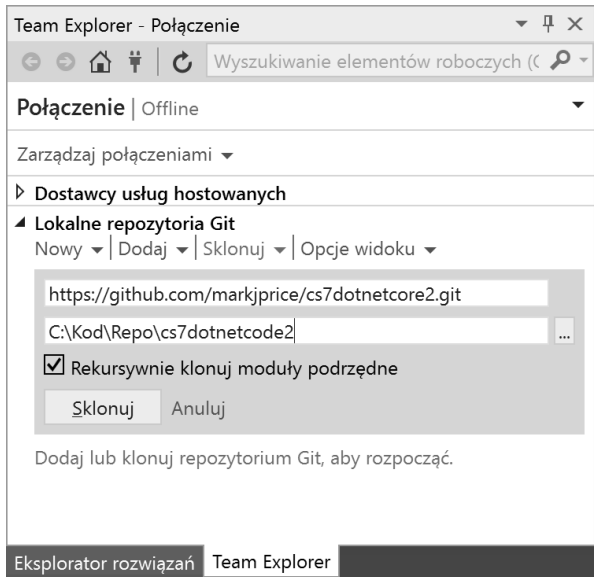
W oknie *Team Explorer* rozwiń sekcję *Lokalne repozytoria Git*, a następnie kliknij menu *Sklonuj* i wprowadź poniższy adres URL, wskazujący na repozytorium Git do sklonowania:

`https://github.com/markjprice/cs7dotnetcore2.git`.

Wprowadź też ścieżkę do sklonowanego repozytorium:

`C:\Kod\Repo\cs7dotnetcore2`.

Jeżeli wszystko wygląda tak jak na poniższym rysunku, to kliknij przycisk *Sklonuj*.



Poczekaj, aż repozytorium zostanie sklonowane.

Teraz masz na swoim dysku lokalną kopię rozwiązań wszystkich praktycznych zadań, jakie zamieściłem w tej książce.

Zarządzanie repozytorium GitHub

Kliknij dwukrotnie repozytorium *cs7dotnetcore2*, żeby otworzyć widok szczegółowy.

Możesz kliknąć poszczególne opcje w sekcji *Projekt*, żeby przejrzeć listę *Przychodzących zatwierdzeń* albo listę problemów lub inne informacje o repozytorium.

Możesz też kliknąć dwukrotnie dowolną pozycję w sekcji *Rozwiązania*, żeby otworzyć ją w *Ekspoloratorze rozwiązań*.

Używanie systemu Git w Visual Studio Code

Visual Studio Code może używać systemu Git, ale wykorzystuje przy tym zainstalowany w systemie operacyjnym pakiet Git. Musisz zatem sam zainstalować pakiet Git 2.0 albo nowszy. Pod poniższym adresem możesz pobrać niezbędny instalator:

<https://git-scm.com/download>.

Jeżeli wolisz korzystać z programów z interfejsem graficznym, to pod poniższym adresem możesz pobrać program GitHub Desktop:

<https://desktop.github.com>.

Konfigurowanie systemu Git z wiersza poleceń

Uruchom *Terminal* i wprowadź poniższe polecenie, żeby sprawdzić aktualną konfigurację:

```
git config --list
```

W wypisanych w odpowiedzi danych powinny się znaleźć Twoja nazwa użytkownika oraz adres e-mail, ponieważ będą one używane przy każdym wykonywanym zatwierdzeniu:

```
...pozostałe opcje konfiguracji...
user.name=Mark J. Price
user.email=markjprice@gmail.com
```

Jeżeli Twoja nazwa użytkownika i adres e-mail nie zostały jeszcze zapisane, możesz to zrobić, wprowadzając poniższe polecenia. Pamiętaj o podaniu w tych instrukcjach własnych danych, nie moich:

```
git config --global user.name "Mark J. Price"
git config --global user.email markjprice@gmail.com
```

Poszczególne opcje konfiguracji możesz skontrolować za pomocą takich instrukcji jak poniższa:

```
git config user.name
```

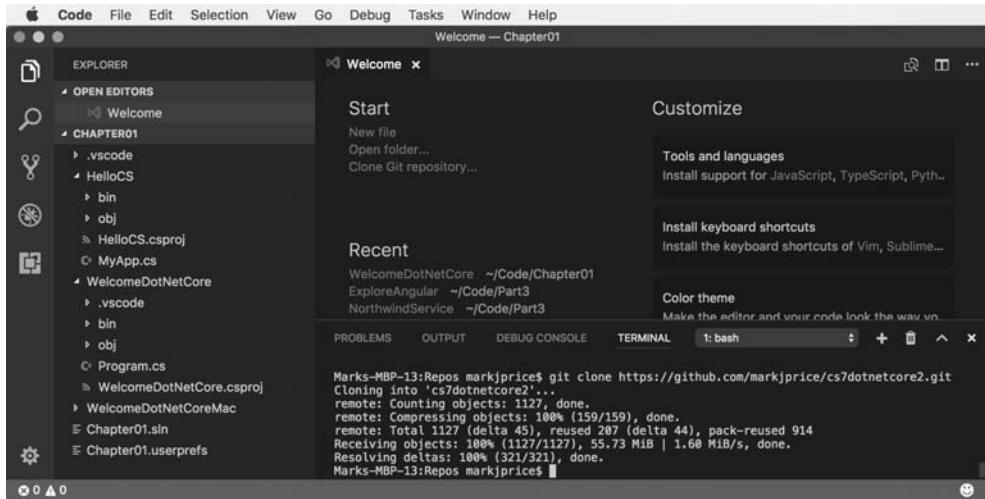
Zarządzanie systemem Git w Visual Studio Code

Uruchom Visual Studio Code i otwórz folder *Kod*.

Wybierz z menu pozycję *View/Integrated Terminal* albo naciśnij klawisze *Ctrl+`*, a następnie wprowadź poniższe polecenia:

```
mkdir Repo
cd Repo
git clone https://github.com/markjprice/cs7dotnetcore2.git
```

Klonowanie wszystkich rozwiązań dla poszczególnych rozdziałów na lokalny dysk twardy może zająć kilka minut, a całość będzie wyglądać tak jak na rysunku na następnej stronie.



Więcej informacji na temat systemów kontroli wersji kodu źródłowego w Visual Studio Code znajdziesz pod poniższym adresem:

<https://code.visualstudio.com/Docs/editor/versioncontrol>.

Praktyka i ćwiczenia

Sprawdź swoją wiedzę i wiadomości, odpowiadając na kilka prostych pytań. Zyskaj trochę doświadczenia w zakresie tematów omawianych w tym rozdziale.

Ćwiczenie 1.1 — sprawdź swoją wiedzę

Odpowiedz na poniższe pytania:

1. Dlaczego programista może używać różnych języków (np. C# i F#) podczas pisania aplikacji działających w .NET Core?
2. Co należy wpisać w wierszu poleceń, żeby skompilować i uruchomić kod źródłowy w C#?
3. Jakie skróty klawiszowe zdefiniowane są w zbiorze Visual C# dla operacji zapisywania, kompilowania i uruchamiania aplikacji bez podłączania debugera?
4. Jaki skrót klawiszowy w Visual Studio Code otwiera okno *Integrated Terminal*?
5. Czy Visual Studio 2017 jest lepsze od Visual Studio Code?
6. Czy .NET Core jest lepsza od .NET Framework?
7. Czym różni się .NET Native od .NET Core?
8. Czym jest .NET Standard i dlaczego jest to tak ważne?

9. Jaka jest różnica między systemem Git a platformą GitHub?
10. Jak się nazywa metoda startowa w aplikacjach konsoli środowiska .NET i jak należy ją deklarować?

Ćwiczenie 1.2 — ćwicz C# gdzie się da

Do ćwiczenia języka C# nie potrzebujesz Visual Studio 2017, Visual Studio for Mac ani Visual Studio Code.

Wystarczy, że odwiedzisz jedną z poniższych stron WWW i zaczniesz pisać kod.

- **.NET Fiddle:** <https://dotnetfiddle.net/>.
- **Cloud9:** <https://c9.io/web/sign-up/free>.

Ćwiczenie 1.3 — dalsza lektura

Skorzystaj z poniższych adresów, żeby dowiedzieć się więcej o zagadnieniach poruszanych w tym rozdziale.

- **Witaj, .NET Core:** <http://dotnet.github.io>.
- **Narzędzia wiersza poleceń .NET Core:** <https://github.com/dotnet/cli>.
- **.NET Core, CoreCLR:** <https://github.com/dotnet/coreclr/>.
- **Plany rozwoju .NET Core:**
<https://github.com/dotnet/core/blob/master/roadmap.md>.
- **Często zadawane pytania dotyczące .NET Standard:**
<https://github.com/dotnet/standard/blob/master/docs/faq.md>.
- **Dokumentacja Visual Studio:** <https://docs.microsoft.com/en-us/visualstudio/>.
- **Blog Visual Studio:** <https://blogs.msdn.microsoft.com/visualstudio/>.
- **Git i Team Services:** <https://www.visualstudio.com/en-us/docs/git/overview>.
- **Najprostszy sposób na podłączenie repozytoriów GitHub do Visual Studio:**
<https://visualstudio.github.com/>.

Podsumowanie

W tym rozdziale przygotowaliśmy środowisko programistyczne, skorzystaliśmy z wiersza poleceń systemu Windows oraz terminala systemu macOS, żeby uruchomić aplikację konsoli. Użyliśmy również Visual Studio 2017, Visual Studio for Mac oraz Visual Studio Code, żeby przygotować podobne aplikacje. Omówiliśmy też różnice pomiędzy .NET Framework, .NET Core, .NET Standard i .NET Native.

W następnym rozdziale nauczymy się mówić w języku C#.

Skorowidz

.NET Core, 40
.NET Framework, 39
.NET Native, 43
.NET Standard, 42, 249

A

abstrakcja, 174
adres URL, 474
agent użytkownika, 473
agregacja, 174
 sekwencji, 428
aktualizowanie encji, 405
algorytm
 AES, 350
 SHA256, 356
Angular, 477, 561
animacja, 577
API EF Core, 388
aplikacja Northwind, 597
aplikacje
 dla Windowsa, 470, 575, 579
 konsoli, 99, 463
 konsoli do publikacji, 265
 mobilne, 471, 615–617
 SPA, 561
 WWW, 470, 509, 539
 z interfejsem graficznym, 464
ASP.NET Core, 478, 539
 kontrolery, 540
 szablon projektu, 480
 tworzenie projektu, 480

ASP.NET Core MVC, 509
 domyślna ścieżka, 522
 kontrolery, 522
 mechanizmy, 520
 modele, 523
 szablon projektu, 513
 testowanie witryny, 517
 widoki, 525
ASP.NET Core Razor Pages
 tworzenie witryn, 473
ASP.NET Core Web API, 539
ASP.NET Identity, 519
asynchroniczne uruchamianie
 operacji, 451
 zadań, 449
atomowe operacje, 461
atrybuty EF Core, 387
automatyczne formatowanie kodu, 56, 62
autoryzacja, 346, 364

B

baza danych Northwind, 375, 379, 502, 544
bazowe biblioteki klas, 250
bazy danych, 373
 definiowanie klasy kontekstu, 500
 grupowanie kontekstów, 407
 migracja, 515
biblioteka, 174, 280
 Angular, 477, 561
 Bootstrap, 477
 Entity Framework, 383

biblioteka
 jQuery, 477
 Node.js, 477
 React, 477
 Redux, 477
 biblioteki bazowe, 250
 blok catch, 465
 blokady, 408, 459
 błędy, 156
 kompilacji, 48
 Bootstrap, 477
 Bower, 477

C

C#, 73
 bloki, 79
 deklarowanie zmiennych, 86
 instrukcje, 78
 komentarze, 78
 metody, 82
 pola, 83
 pomoc, 81
 słownictwo, 79
 słownik języka, 84
 typy, 83
 zmienne, 83
 chmura, 616
 CLR, Common Language Runtime, 250
 CoreFX, 250
 CORS, Cross-Origin Resource Sharing, 566
 CSS3, 477

D

debugowanie, 149
 dostosowywanie punktów przerwania, 155
 krokowe wykonywanie kodu, 154
 okna, 152
 pasek narzędzi, 151
 poziomy śledzenia, 160
 typ Debug, 157
 typ Trace, 157
 definiowanie
 funkcji lokalnych, 216
 indeksów, 205
 klas, 176, 212
 klas encji, 390, 495

klasy kontekstu bazy danych, 391, 500
 pól, 182
 wspólnego układu, 489, 490
 zdarzeń, 218
 deklarowanie zmiennych, 86
 dekodowanie tekstu, 336
 delegat, 217
 deserializacja danych, 340
 dodawanie typów, 85
 dokumentowanie serwisów, 551
 dopasowywanie wzorców, 399
 dostawca danych
 EF Core, 383
 LINQ, 412
 dostęp, 203
 do wspólnych zasobów, 456
 DRY, Don't Repeat Yourself, 141
 dziedziczenie, 174
 klas, 233
 po System.Object, 181
 dzielenie klas, 202

E

edytor tekstu, 44
 TextEdit, 46
 EF Core, *Patrz* Entity Framework Core
 eksplorator serwera, 376
 encje, 389
 aktualizowanie, 405
 filtrowanie, 412
 grupowanie, 425
 łączenie, 425
 sortowanie, 417
 usuwanie, 406
 wstawianie, 404
 Entity Framework Core, 373, 383
 ASP.NET Core, 494
 atrybuty, 387
 dopasowywanie wzorców, 399
 konwencje, 387
 manipulowanie danymi, 404, 504
 płynne API, 388
 protokolowanie, 395
 serwis, 502
 tworzenie modelu danych, 388, 422
 wzorce ładowania, 401
 zapytania do modelu, 393

F

filtrowanie
 encji, 412
 według typu, 418
 Fluent Design, 575, 577
 folder
 Controllers, 513
 Data, 513
 Extensions, 515
 Models, 513
 Services, 515
 Views, 513
 wwwroot, 513
 format XML, 436
 formatowanie kodu, 56
 frameworki, 263
 funkcje, 141
 agregujące, 428
 matematyczne, 146
 skrótów, 345, 354
 tworzenie, 141

G

generowanie
 danych XML, 437
 kluczy, 348
 liczb losowych, 361
 GitHub, 65
 klonowanie repozytorium, 67
 Visual Studio Code, 68
 zarządzanie repozytorium, 67
 graficzny interfejs użytkownika, 590
 grupowanie
 encji, 425
 kontekstów, 407
 Gulp, 477

H

hermetyzacja, 174
 HTML5, 477
 HTTP, Hypertext Transfer Protocol, 473

I

IDE, Integrated Development Environment, 28
 implementowanie
 autoryzacji, 366
 działań w metodzie, 213
 działań za pomocą operatora, 214
 interfejsów, 220
 uwierzytelniania, 366
 w systemie Android, 625
 w systemie iOS, 625
 importowanie przestrzeni nazw, 100, 179, 256
 indekser, 177, 203, 205
 informacje o pliku, 322
 inicjowanie pól, 190
 instalowanie
 .NET Core SDK for macOS, 34
 kontrolek, 589
 menedżera pakietów Node, 35
 Microsoft Visual Studio 2017, 30
 Microsoft Visual Studio Code, 33
 Microsoft Visual Studio Code for macOS,
 33, 37
 na wielu platformach, 30
 pakietu Swagger, 553
 rozszerzenia C#, 35
 Windows Template Studio, 606
 Xcode, 37
 instrukcja
 async, 462
 await, 462, 465
 checked, 130
 do, 119
 for, 119
 foreach, 120
 if, 114
 like, 399
 lock, 459
 switch, 115
 try, 127, 329
 unchecked, 131
 using, 331
 while, 118
 instrukcje
 iteracji, 118
 wyboru, 113

interfejs
 do wybierania numerów telefonów, 624
 IComparable, 220
 IComparer, 221
 IDisposable, 221, 232
 IFormatProvider, 221
 IFormattable, 221
 IFormatter, 221
 interfejsy
 implementowanie, 220
 użytkownika, 577
 iteracja, 118

J

JavaScript, 477
 jawne
 definiowanie transakcji, 408
 tworzenie delegata, 416
 język
 Roslyn, 49
 TypeScript, 477
 XAML, 578
 jQuery, 477
 JSON, 544

K

katalog, 318
 klasa, 176
 Category, 389
 Console, 416
 Nagrywanie, 444
 Northwind, 391
 Product, 390
 Thread, 451
 klasy
 definiowanie, 212
 dziedziczenie, 233
 dzielenie, 202
 kontekstu bazy danych, 499
 wyliczeniowe, 412
 klawisz F12, 133
 klient, 477
 klucze, 346
 kod
 automatyczne formatowanie, 56
 przenoszenie, 278
 statusu, Status Code, 476

kodowanie tekstu, 333
 kolekcje, 187
 silnie typowane, 187
 komparator, 223
 kompilatory języków, 249
 kompilowanie kodu, 48
 Visual Studio Code, 60
 kompozycja, 174
 kompresowanie strumieni, 331
 konfigurowanie
 biblioteki klas, 210
 EF Core, 383
 Entity Framework Core, 502
 obiektów nasłuchujących, 159
 repozytorium, 548
 środowiska programistycznego, 28
 konsola, 101
 konstruktor, 176
 kontrola dostępu, 203
 kontrolery
 ASP.NET Core, 540
 ASP.NET Core MVC, 522
 Web API, 549
 kontrolki, 579, 584
 instalowanie, 589
 zmiana szablonu, 592
 konwertowanie, 127
 między typami, 121
 na ciąg znaków, 124
 krotka, 194

L

liczby losowe, 361
 LINQ, Language Integrated Query, 411
 EF Core, 421
 rozszerzona składnia, 429
 równoległe zapytania, 430
 tworzenie zapytań, 411
 własne metody rozszerzające, 433
 wyrażenia lambda, 417
 LINQ to XML, 436
 generowanie danych XML, 437
 odczytywanie danych XML, 437
 lista klientów, 638
 literały, 87

Ł

ładowanie encji
 jawne, 402
 leniwe, 401
 łączenie encji, 425

M

manipulowanie danymi, 404
 menedżer pakietów NPM, 477
 metapakiety, 261
 metoda, 142, 176, 193, 217
 GroupJoin, 425
 Join, 425
 OrderBy, 417
 Restart, 446
 Select, 422
 Stop, 446
 t.Wait(), 453
 Task.WaitAll(), 453
 Task.WaitAny(), 453
 ThenBy, 418
 Where, 412, 418
 żądania, Request Method, 476
 metody
 przeciążanie, 198
 przekazywanie parametrów, 197, 200
 rozszerzające LINQ, 412
 tworzenie, 193
 wywoływanie, 193
 Microsoft
 Docs, 133
 SQL Server, 375
 Visual Studio 2017, 30
 Visual Studio Code, 33
 migrowanie bazy danych, 515
 modele
 aplikacji, 469
 ASP.NET Core MVC, 523
 danych EF Core, 422
 EF Core, 387
 modyfikatory dostępu, 176, 183
 monitorowanie
 wydajności, 442, 443
 zużycia pamięci, 443
 Mono, 39
 MSDN, Microsoft Developer Network, 133

N

nagłówek akceptowanych kodowań, 476
 nagłówki żądania, Request Headers, 476
 narzędzia wiersza poleceń, 44, 46
 narzędzie
 .NET Portability Analyzer, 280
 dotnet, 270
 Swagger, 551
 nazywanie zmiennych, 87
 Node.js, 477
 notatnik Windows, 45
 NPM, 477

O

obiekt, 177
 obiekty nasłuchujące, 159
 obsługa
 napędów, 316
 wyjątków, 127
 wyjątków rzutowania, 239
 zdarzeń, 217
 ochrona funkcji aplikacji, 369
 oczekiwanie na zadania, 453
 odwołanie do pakietu, 271
 okna debugowania, 152
 operacje
 atomowe, 461
 uruchamianie asynchroniczne, 451
 uruchamianie synchroniczne, 450
 operator rzutowania, 122
 operatory, 106, 177, 213
 arytmetyczne, 108
 jednoargumentowe, 107
 logiczne, 109
 oświetlenie ujawniające, 578, 586

P

pakiety
 NuGet, 250, 270, 637
 testowanie, 277
 tworzenie, 273
 z szablonami, 570
 parametry
 nazywane, 198
 opcjonalne, 198

- parsowanie ciągów znaków, 126
 - pasek narzędzi debugowania, 151
 - pędzle akrylowe, 584
 - platforma
 - GitHub, 65
 - Windows, 576
 - Xamaria, 616
 - plik, 320
 - appsettings.json, 515
 - bower.json, 515
 - NorthwindMvc.csproj, 513
 - Program.cs, 515
 - Startup.cs, 515
 - pliki
 - .cshtml, 487
 - code-behind, 492
 - domyślne, 487
 - statyczne, 484
 - plynne API, 388
 - pobieranie
 - danych, 100
 - listy klientów, 638
 - podpisy, 346
 - podpisywanie danych, 358
 - poła, 176, 182
 - definiowanie, 182
 - domyślny literal, 191
 - inicjowanie, 190
 - przechowywanie danych, 182
 - stałe, 189
 - statyczne, 188
 - tylko do odczytu, 190
 - polimorfizm, 174
 - połączenie z bazą danych, 384
 - pomiar wydajności, 446
 - pomoc, 132
 - klawisz F12, 133
 - Microsoft Docs, 133
 - MSDN, 133
 - Stack Overflow, 134
 - wzorce projektowe, 136
 - poprawianie widoków, 610
 - poziom izolacji, 408
 - procesy, 448
 - programowanie
 - obiektywne, 173
 - sterowane testami, TDD, 162
 - projekt UWP, 580
 - projekty responsywne, 582
 - protokołowanie błędów, 156
 - protokół HTTP, 473
 - przechowywanie
 - danych, 182
 - liczb, 88
 - tekstu, 88
 - wartości logicznych, 93
 - przechwytywanie wyjątków, 128
 - przeciążanie metod, 198
 - przekazywanie parametrów, 197, 200, 532, 534
 - przenoszenie kodu, 278
 - przepelnienia, 130
 - przestrzeń nazw, 100, 179, 249–255
 - System.Diagnostics, 443
 - publikowanie aplikacji, 265
 - za pomocą Visual Studio 2017, 266
 - za pomocą Visual Studio Code, 269
 - punkt przzerwania, 150, 155
- ## R
- Razor Pages, 487
 - definiowanie strony, 488
 - pliki code-behind, 492
 - wspólny układ, 489
 - React, 477
 - Redux, 477
 - rejestrowanie repozytorium, 548
 - rekurencja, 147
 - relacyjne bazy danych, 374
 - repozytorium danych, 545
 - responsywny układ, 582, 604
 - REST, Representational State Transfer, 539, 637
 - rodzaje synchronizacji, 462
 - Roslyn, 49
 - rozszerzanie
 - typów, 241
 - C# for Visual Studio Code, 35
 - RSA, 359
 - rzutowanie, 121
 - jawne, 122, 238
 - niejawne, 121
 - obsługa wyjątków, 239
- ## S
- serializacja obiektów, 337
 - serializowanie, 342
 - do formatu JSON, 341
 - do formatu XML, 337, 340

- serwer SQL Server, 375
- serwis internetowy, 544
 - dokumentowanie, 551
 - Northwind, 565
 - REST, 637
 - SerwisNorthwind, 567
 - testowanie, 551
- SHA256, 359
- silnia, 147
- skalowalność, 441
- składnia
 - rozszerzona zapytań LINQ, 429
 - zapytań LINQ, 412
- słowo kluczowe, 257
 - async, 462
 - await, 462, 604
 - enum, 184
- sortowanie
 - encji, 417
 - obiektów, 221
- sól, 347
- SQL Server, 375
- SQLite, 379
- SQLiteStudio, 380
- strumień, 324
 - tekstowy, 327
 - XML, 328
- Swagger, 551
 - instalowanie pakietu, 553
 - testowanie API HTTP, 553
 - testowanie żądań GET, 554
 - testowanie żądań POST, 557
- synchroniczne uruchamianie operacji, 450
- synchronizacja, 462
- synchronizowanie dostępu, 456
- system
 - Fluent Design, 577
 - plików, 313
 - zarządzania pakietami, 477
- szablon, 590
 - kontrolki, 592
 - projektu, 561, 569
- szyfrowanie, 345, 349
 - symetryczne, 350

Ś

- ścieżka, 321
- śledzenie, 160

- środowisko
 - CLR, 250
 - programistyczne, 28

T

- tablice, 102
- TDD, Test Driven Development, 162
- tekst
 - dekodowanie, 336
 - kodowanie, 333
- testowanie, 612
 - API HTTP, 553
 - aplikacji mobilnej, 632
 - autoryzacji, 367
 - pakietu, 277
 - podpisów, 360
 - pustej witryny, 483
 - serwisów, 551
 - uwierzytelniania, 367
 - w Visual Studio 2017, 162, 167
 - w Visual Studio Code, 164, 168
 - witryny, 517
 - wywołań serwisu, 568
 - żądań GET, 551, 554
 - żądań POST, 557
- testy jednostkowe, 162
- transakcje, 407
 - jawne definiowanie, 408
- tryb dewelopera, 580
- tworzenie
 - aplikacji
 - dla Windowsa, 579
 - konsoli, 265
 - mobilnych, 615, 617
 - Northwind, 597
 - SPA, 561
 - WWW, 509, 539
 - bazy danych Northwind, 502
 - biblioteki klas, 174, 259
 - w Visual Studio 2017, 175
 - w Visual Studio Code, 175
 - funkcji, 141
 - generycznej metody, 227
 - interfejsu, 624
 - kontrolera Web API, 549
 - metod, 193
 - metod rozszerzających, 433

tworzenie

- modelu, 620
 - danych EF Core, 422
 - encji, 494
- obiektów, 177
- operacji atomowych, 461
- pakietu, 273
- pól statycznych, 188
- pól tylko do odczytu, 190
- projektu ASP.NET Core, 480
- projektu UWP, 580
- punktu przerwania, 150
- repozytorium, 545
- rozwiązania Xamarin.Forms, 618
- serwisów, 539, 544
- stałych pól, 189
- testów jednostkowych, 166
- typu generycznego, 225
- usług, 539
- widoku listy klientów, 627
- widoku szczegółowego, 630
- witryny ASP.NET Core MVC, 510
- własnych typów, 173
- zapytań LINQ, 411

typ

- .NET, 258
- Convert, 123
- DbContext<T>, 465
- DbSet<T>, 465
- Debug, 157
- dynamic, 94
- HttpClient, 465
- object, 93
- StreamReader, 465
- StreamWriter, 465
- struct, 229
- Trace, 157

TypeScript, 477

typowanie statyczne, 187

typy

- .NET Standard, 249
- generyczne, 225
- referencyjne, 96, 228
- wartości, 228
- własne, 173
- zdefiniowane, 249

U

- Universal Windows Platform, 576
- upiększanie składni, 429
- URL, Uniform Resource Locator, 474
- uruchamianie aplikacji, 85
- usługi, 539
- ustawienia projektu, 610
- usuwanie encji, 406
- uwierzytelnianie, 346, 364
 - systemu, 519
- UWP, Universal Windows Platform, 575, 580

V

- Visual Studio 2017, 49, 50, 74
 - dodawanie projektów, 55
 - GitHub, 65
 - kompilowanie kodu, 53
 - łączenie z bazą danych, 385
 - tworzenie witryny ASP.NET Core MVC, 510
 - używanie systemu Git, 66
- Visual Studio Code, 60, 76
 - GitHub, 68
 - kompilowanie kodu, 60, 61
 - łączenie z bazą danych, 385
 - pisanie kodu, 60
 - tworzenie witryny ASP.NET Core MVC, 511
- Visual Studio for Mac, 37, 62

W

- wartość null, 96
- wątek, 448, 457
 - pierwotny, 453
- Webpack, 477
- wektor inicjujący, 347
- wiązanie
 - danych, 594
 - do elementów, 594
 - do źródła danych, 595
 - modelu, 532
- widok
 - listy klientów, 627
 - paralaksy, 578
- widoki ASP.NET Core MVC, 525
- wielowątkowość, 465
- wielozadaniowość, 441

wiersz poleceń, 46
 kompilowanie aplikacji, 46
 Windows Template Studio, 606
 własne
 metody rozszerzające, 433
 pakiety, 270
 właściwości, 177, 203
 tylko do odczytu, 203
 z możliwością przypisania, 204
 właściwość
 Elapsed, 446
 ElapsedMilliseconds, 446
 współdzielenie
 kodu, 258
 zasobów, 591
 wstawianie encji, 404
 wybieranie
 komponentów, 31
 pakietów, 31
 typu projektu, 607
 wydajność, 441
 typów, 442
 wyjątek, 127
 wykrywanie przepełnień, 130
 wyrażenia lambda, 412
 wyświetlanie informacji, 99
 wywoływanie metod, 193
 wzorce, 399
 ładowania, 401
 projektowe, 136

X

Xamarin, 39, 615
 Xamarin.Forms, 615
 XAML, 575, 615
 XAML Standard 1.0, 578

Z

zadania, 448
 asynchroniczne uruchamianie, 449
 kontynuowanie pracy, 454
 potomne, 455
 zagnieżdżone, 455
 zaokrąglanie liczb, 123
 zapytania
 LINQ, 411
 równoległe LINQ, 430
 zarządzanie
 kodem źródłowym, 65
 pamięcią, 228
 plikami, 323
 projektami, 180
 repozytorium GitHub, 67
 zasoby, 442, 590
 wspólne, 457
 zbiór, 420
 zdarzenia, 217
 definiowanie, 218
 zestaw, 250, 252, 255
 zintegrowane środowisko programistyczne, IDE, 28
 zmiana szablonu kontrolki, 592
 zmienne, 86, 106
 lokalne, 94
 zużycie pamięci, 443

Ż

żądanie
 DELETE, 553
 GET, 551, 554
 POST, 553, 557
 PUT, 553

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

Wieloplatformowa aplikacja — profesjonalny deweloper!

Powszechnie uważa się, że C# jest świetnym językiem ogólnego zastosowania, który nadaje się do tworzenia każdego rodzaju oprogramowania — od serwisów internetowych, przez aplikacje biznesowe, aż po gry. Oprogramowanie takie działa na komputerach biurowych, serwerach, urządzeniach mobilnych, a także na specjalizowanych systemach do gier. Z kolei .NET Core pozwala na tworzenie rozwiązań serwerowych w chmurze oraz na pracę z urządzeniami wirtualnej lub rozszerzonej rzeczywistości. To wszystko sprawia, że technologie C# i .NET Core umożliwiają tworzenie nowoczesnych systemów, które będą wydajnie działać na wielu różnych platformach.

Ta książka jest znakomitym, spójnym i bardzo praktycznym podręcznikiem do nauki języka C#. Pokazuje również najlepsze praktyki stosowane na platformie .NET Core. Z publikacji skorzysta każdy, kto pragnie zdobyć dobre przygotowanie do pracy z językiem i platformą. Znalazły się tu przystępnie wyłożone podstawy języka C#, a także sposoby debugowania kodu i zasady programowania obiektowego. Nie zabrakło informacji o najnowszych elementach języka C# 7.1, takich jak domyślne literały, krotki, dopasowywanie wzorców, zmienne typu out i inne. Dokładnie opisano biblioteki klas .NET Standard 2.0. Zaprezentowano najważniejsze rodzaje aplikacji, takie jak witryny i aplikacje internetowe, serwisy sieciowe, aplikacje UWP oraz aplikacje mobilne.

W książce między innymi:

- Solidne podstawy C# 7.1 i .NET Core 2.0
- Profesjonalne aplikacje w ASP.NET Core 2.0 i wielozadaniowość
- Stosowanie Entity Framework Core oraz LINQ do pracy nad danymi
- Korzystanie ze środowiska UWP oraz z Xamarin.Forms
- Szyfrowanie, strumienie i serializacja
- Planowane funkcje języka C# w wersji 8

Mark J. Price specjalizuje się w programowaniu w języku C#. Ma certyfikat MCSD oraz Episerver. Na początku XXI wieku pisał oficjalne kursy programowania dla Microsoftu, włączając w to kursy języka C#. Poza doświadczeniem w rozwijaniu aplikacji i pisaniu poprawnego kodu ma duże umiejętności edukacyjne: potrafi efektywnie przekazać wiedzę zarówno profesjonalistom z wieloletnim doświadczeniem, jak i 16-latkom stawiającym pierwsze kroki w pisaniu kodu. Obecnie tworzy i prowadzi kursy dotyczące usług Digital Experience Cloud firmy Episerver.

 helion.pl	<i>Sprawdź nasze szkolenia!</i> SZKOLENIA  AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	KOD KORZYŚCI <i>Sięgnij po więcej!</i>   ISBN 978-83-283-4450-1  9 788328 344501
 helion.pl		
 HELION SA ul. Kościuszki 1c 44-100 Gilwice tel.: 32 230 98 63 helion@helion.pl		
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 99,00 zł

Packt