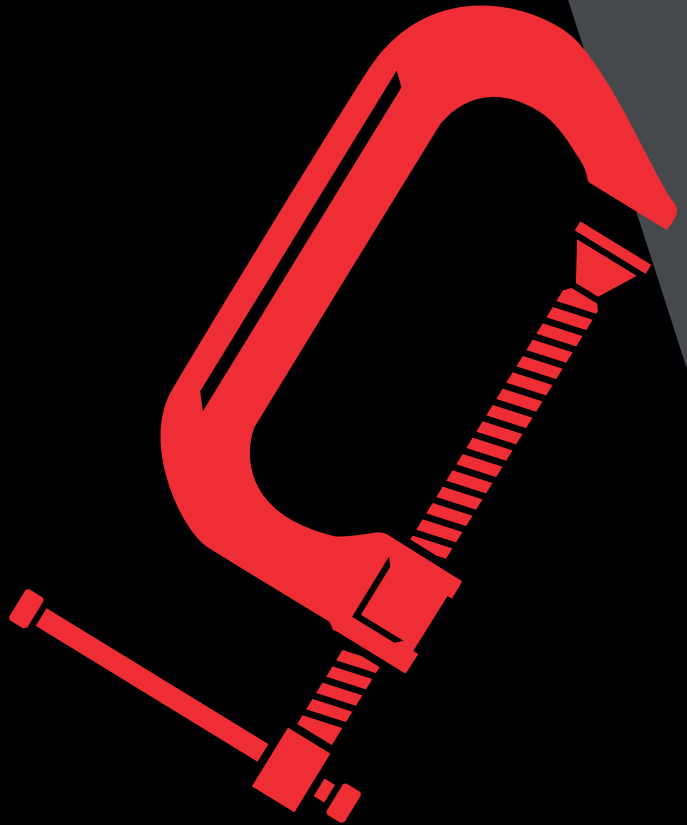


Microsoft Visual C# 2017 Krok po kroku



John Sharp

John Sharp

Microsoft Visual C# 2017

Krok po kroku

Przekład: Natalia Chounlamany, Janusz Machowski, Krzysztof Szkudlarek,
Marek Włodarz

APN Promise, Warszawa 2018

Microsoft Visual C# 2017 Krok po kroku

Authorized Polish translation of the English language edition entitled: Microsoft Visual C# Step by Step, 9th Edition, ISBN 978-1-5093-0776-0, by John Sharp, published by Pearson Education, Inc, publishing as Microsoft Press, A Division Of Microsoft Corporation.

Copyright © 2018 by Pearson Education, Inc.

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 Pearson Education, Inc.

Polish language edition published by APN PROMISE SA Copyright © 2018

Autoryzowany przekład z wydania w języku angielskim, zatytułowanego: Microsoft Visual C# Step by Step, 8th Edition, ISBN 978-1-5093-0104-1, by John Sharp, opublikowanego przez Pearson Education, Inc, publikującego jako Microsoft Press, oddział Microsoft Corporation.

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa, tel. +48 22 35 51 600, fax +48 22 35 51 699, e-mail: mSPress@promise.pl

Książka ta przedstawia poglądy i opinie autorów. Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń, chyba że zostanie jednoznacznie stwierdzone, że jest inaczej. Ewentualne podobieństwo do jakiegokolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

Nazwa Microsoft oraz znaki towarowe wymienione na stronie <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> są zastrzeżonymi znakami towarowymi grupy Microsoft. Wszystkie inne znaki towarowe są własnością ich odnośnych właścicieli.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji. APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiegokolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-372-4

Przekład: Natalia Chounlamany, Janusz Machowski, Krzysztof Szkudlarek,
Marek Włodarz

Redakcja: Marek Włodarz

Korekta: Ewa Swędrowska

Skład i łamanie: MAWart Marek Włodarz

Skrócony spis treści

Część I Wprowadzenie do języka Visual C# i Microsoft Visual Studio 2017

1	Wprowadzenie do języka C#	3
2	Zmienne, operatory i wyrażenia	41
3	Tworzenie metod i stosowanie zakresów zmiennych	71
4	Instrukcje wyboru	111
5	Złożone instrukcje przypisania oraz instrukcje iteracji	133
6	Obsługa błędów i wyjątków	153

Część II Model obiektowy języka C#

7	Tworzenie i zarządzanie klasami oraz obiektami	185
8	Wartości i referencje	213
9	Tworzenie typów wartościowych przy użyciu wyliczeń oraz struktur	243
10	Tablice	267
11	Tablice parametrów	297
12	Dziedziczenie	311
13	Tworzenie interfejsów oraz definiowanie klas abstrakcyjnych	337
14	Proces oczyszczania pamięci i zarządzanie zasobami	369

Część III Rozszerzalne typy danych w języku C#

15	Implementacja właściwości zapewniających dostęp do pól	397
16	Indeksatory i obsługa danych binarnych	423
17	Typy ogólne	443
18	Kolekcje	475
19	Wyliczanie kolekcji	501
20	Wydzielanie logiki aplikacji i obsługa zdarzeń	517
21	Odpytywanie danych w pamięci przy użyciu wyrażeń w języku zapytań	549
22	Przeciążanie operatorów	575

Część IV Tworzenie aplikacji Universal Windows Platform w języku C#

23	Przyspieszanie działania za pomocą zadań	601
24	Skracanie czasu reakcji za pomocą działań asynchronicznych.....	649
25	Implementowanie interfejsu użytkownika aplikacji Universal Windows Platform.....	699
26	Wyświetlanie i wyszukiwanie danych w aplikacjach Universal Windows Platform.....	749
27	Dostęp do zdalnej bazy danych z poziomu aplikacji Universal Windows Platform.....	799

Spis treści

<i>Podziękowania</i>	xv
<i>O autorze</i>	xvii
<i>Wstęp</i>	xix

Część V Wprowadzenie do języka Visual C# i Microsoft Visual Studio 2017

1 Wprowadzenie do języka C#	3
Rozpoczynamy programowanie przy użyciu środowiska Visual Studio 2017	3
Piszemy pierwszy program	9
Przestrzenie nazw	16
Tworzenie aplikacji graficznej	20
Analiza aplikacji Universal Windows Platform	31
Dodawanie kodu do aplikacji graficznej	35
Inne typy aplikacji graficznych	38
Podsumowanie	39
Krótki przegląd rozdziału 1	39
2 Zmienne, operatory i wyrażenia	41
Instrukcje	41
Identyfikatory	42
Słowa kluczowe	43
Zmienne	44
Nazywanie zmiennych	44
Deklarowanie zmiennych	45
Specyfikowanie wartości numerycznych	46
Podstawowe typy danych	47
Zmienne lokalne bez przypisanej wartości	47
Wyświetlanie wartości podstawowych typów danych	48
Posługiwanie się operatorami arytmetycznymi	55
Operatory i typy danych	55
Poznajemy operatory arytmetyczne	58
Kontrolowanie pierwszeństwa	64
Stosowanie zasad łączności przy wyznaczaniu wartości wyrażień	64
Zasady łączności a operator przypisania	65
Inkrementacja i dekrementacja wartości zmiennych	66
Formy przyrostkowe i przedrostkowe	67
Deklarowanie zmiennych lokalnych o niejawnie określonym typie danych	68
Podsumowanie	69

Krótki przegląd rozdziału 2	70
3 Tworzenie metod i stosowanie zakresów zmiennych	71
Tworzenie metod	71
Deklarowanie metody	72
Zwracanie danych przez metodę	73
Stosowanie metod wcielających wyrażenie	74
Wywoływanie metod	76
Zwracanie wielu wartości z metody	79
Stosowanie zakresu	82
Definiowanie zakresu lokalnego	83
Definiowanie zakresu klasy	83
Przeciążanie metod	84
Tworzenie metod	85
Zagnieżdżanie metod	95
Stosowanie parametrów opcjonalnych oraz nazwanych argumentów	98
Definiowanie parametrów opcjonalnych	100
Przekazywanie nazwanych argumentów	101
Rozwiązywanie niejednoznaczności związanych z parametrami opcjonalnymi i argumentami nazwanymi	102
Podsumowanie	108
Krótki przegląd rozdziału 3	108
4 Instrukcje wyboru	111
Deklarowanie zmiennych logicznych	111
Stosowanie operatorów logicznych	112
Operatory równościowe oraz operatory relacji	112
Warunkowe operatory logiczne	113
Skracanie działania	114
Podsumowanie informacji o pierwszeństwie oraz łączności operatorów	115
Podejmowanie decyzji przy użyciu instrukcji <i>if</i>	116
Składnia instrukcji <i>if</i>	116
Grupowanie instrukcji w bloki	118
Kaskadowe łączenie instrukcji <i>if</i>	119
Stosowanie instrukcji <i>switch</i>	125
Składnia instrukcji <i>switch</i>	125
Reguły stosowania instrukcji <i>switch</i>	127
Podsumowanie	131
Krótki przegląd rozdziału 4	131
5 Złożone instrukcje przypisania oraz instrukcje iteracji	133
Złożone operatory przypisania	133
Instrukcja <i>while</i>	135
Instrukcja <i>for</i>	140

Zakres instrukcji <i>for</i>	142
Instrukcja <i>do</i>	143
Podsumowanie	152
Krótki przegląd rozdziału 5	152
6 Obsługa błędów i wyjątków	153
Zmaganie się z błędami	154
Wypробowywanie kodu i przechwytywanie wyjątków	154
Nieobsłużone wyjątki	156
Stosowanie kilku bloków obsługi	157
Przechwytywanie wielu wyjątków	158
Filtry wyjątków	160
Propagowanie wyjątków	165
Wykonywanie operacji arytmetycznych z kontrolą lub bez kontroli	
przepełnienia	168
Pisanie instrukcji z kontrolą przepełnienia	169
Pisanie wyrażeń z kontrolą przepełnienia	170
Zgłaszanie wyjątków	172
Używanie wyjątków <i>throw</i>	178
Stosowanie bloku <i>finally</i>	179
Podsumowanie	181
Krótki przegląd rozdziału 6	181

Część VI Model obiektowy języka C#

7 Tworzenie i zarządzanie klasami oraz obiektami	185
Omówienie klasyfikacji	185
Cele hermetyzacji	186
Definiowanie i używanie klas	186
Kontrolowanie dostępności	188
Konstruktory	190
Przeciążanie konstruktorów	192
Dekonstrukcja obiektu	201
Metody i dane statyczne	203
Tworzenie pól współdzielonych	204
Tworzenie pól statycznych przy użyciu słowa kluczowego <i>const</i>	205
Klasy statyczne	205
Statyczne instrukcje <i>using</i>	206
Klasy anonimowe	209
Podsumowanie	210
Krótki przegląd rozdziału 7	211
8 Wartości i referencje	213
Kopiowanie klas oraz zmiennych typu wartościowego	213

Wartości null oraz typy danych dopuszczające stosowanie wartości null	220
Operatory warunkowe wartości <i>null</i>	222
Typy danych dopuszczające stosowanie wartości <i>null</i>	223
Właściwości typów nullable	224
Używanie parametrów typu <i>ref</i> i <i>out</i>	225
Tworzenie parametrów typu <i>ref</i>	226
Tworzenie parametrów typu <i>out</i>	227
Organizacja pamięci komputera	229
Korzystanie ze stosu i ze sterty	231
Klasa <i>System.Object</i>	232
Opakowywanie typów danych	233
Rozpakowywanie typów danych	234
Bezpieczne rzutowanie danych	236
Operator <i>is</i>	236
Operator <i>as</i>	237
Ponowne odwiedziny instrukcji <i>switch</i>	237
Podsumowanie	241
Krótki przegląd rozdziału 8	241
9 Tworzenie typów wartościowych przy użyciu wyliczeń oraz struktur	243
Wyliczeniowe typy danych	243
Deklarowanie wyliczeniowego typu danych	244
Stosowanie wyliczeniowych typów danych	244
Wybór wartości literałów wyliczeniowych	245
Wybór typu danych używanego do wewnętrznego reprezentowania wartości wyliczeniowych	246
Struktury	249
Deklarowanie struktury	251
Różnice pomiędzy strukturami i klasami	252
Deklarowanie zmiennych strukturalnych	254
Omówienie inicjowania struktur	254
Kopiowanie zmiennych strukturalnych	259
Podsumowanie	264
Krótki przegląd rozdziału 9	264
10 Tablice	267
Deklarowanie i tworzenie tablicy	267
Deklarowanie zmiennych tablicowych	268
Tworzenie instancji tablicy	268
Wypełnianie tablic danymi i ich używanie	270
Tworzenie tablic o niejawnie określonym typie elementów	271
Korzystanie z indywidualnych elementów tablicy	272
Wykonywanie iteracji poprzez elementy tablicy	272
Przekazywanie tablic jako parametrów i zwracanie ich jako wartości metod	274

Kopiowanie tablic	276
Tablice wielowymiarowe	278
Tworzenie tablic nieregularnych	279
Dostęp do tablic zawierających typy wartościowe	290
Podsumowanie	294
Krótki przegląd rozdziału 10	294
11 Tablice parametrów	297
Przeciążanie: krótkie przypomnienie faktów	297
Używanie argumentów będących tablicami	298
Deklarowanie tablicy parametrów typu <i>params</i>	300
Używanie parametru typu <i>params object[]</i>	302
Stosowanie tablicy parametrów typu <i>params</i>	304
Porównanie tablic parametrów z parametrami opcjonalnymi	307
Podsumowanie	310
Krótki przegląd rozdziału 11	310
12 Dziedziczenie	311
Czym jest dziedziczenie?	311
Korzystanie z mechanizmów dziedziczenia	312
Powtórka informacji na temat klasy <i>System.Object</i>	314
Wywoływanie konstruktora klasy bazowej	315
Przypisywanie klas	316
Deklarowanie metod z użyciem słowa kluczowego <i>new</i>	318
Deklarowanie metod wirtualnych	319
Deklarowanie metod z użyciem słowa kluczowego <i>override</i>	321
Dostęp chroniony	324
Metody rozszerzające	330
Podsumowanie	335
Krótki przegląd rozdziału 12	335
13 Tworzenie interfejsów oraz definiowanie klas abstrakcyjnych	337
Interfejsy	337
Definiowanie interfejsu	339
Implementowanie interfejsu	339
Odwoływanie się do klasy za pomocą jej interfejsu	341
Praca z wieloma interfejsami	342
Jawne implementowanie interfejsu	343
Ograniczenia interfejsów	345
Definiowanie i używanie interfejsów	346
Klasy abstrakcyjne	356
Metody abstrakcyjne	357
Klasy zamknięte	358
Metody zamknięte	358

Implementowanie i używanie klas abstrakcyjnych	359
Podsumowanie	366
Krótki przegląd rozdziału 13	367
14 Proces oczyszczania pamięci i zarządzanie zasobami	369
Żywot obiektów	369
Tworzenie destruktorów	371
Dlaczego trzeba używać kolektorów śmieci?	373
Działanie procesu oczyszczania pamięci?	375
Zalecenia	376
Zarządzanie zasobami	377
Metody sprzątające	377
Sprzątanie w sposób odporny na występowanie wyjątków	378
Instrukcja <i>using</i> oraz interfejs <i>IDisposable</i>	379
Wywoływanie metody <i>Dispose</i> z poziomu destruktora	380
Implementacja metody sprzątającej odpornej na występowanie wyjątków	383
Podsumowanie	393
Krótki przegląd rozdziału 14	393

Część VII Rozszerzalne typy danych w języku C#

15 Implementacja właściwości zapewniających dostęp do pól	397
Implementacja hermetyzacji przy użyciu metod	397
Co to są właściwości?	399
Używanie właściwości	402
Właściwości tylko do odczytu	403
Właściwości tylko do zapisu	403
Dostępność właściwości	404
Ograniczenia właściwości	405
Deklarowanie właściwości interfejsu	407
Zastępowanie metod właściwościami	408
Generowanie automatycznych właściwości	412
Inicjowanie obiektów przy użyciu właściwości	415
Podsumowanie	419
Krótki przegląd rozdziału 15	419
16 Indeksatory i obsługa danych binarnych	423
Co to jest indeksator?	423
Przechowywanie danych dwójkowych	424
Wyświetlanie wartości dwójkowych	425
Manipulowanie wartościami dwójkowymi	425
Ten sam przykład z wykorzystaniem indeksatorów	427
Akcesory indeksatora	429
Porównanie indeksatorów i tablic	430

Indeksatory w interfejsach	432
Stosowanie indeksatorów w aplikacjach Windows	433
Podsumowanie	440
Krótki przegląd rozdziału 16	440
17 Typy ogólne	443
Problem niewłaściwego użycia typu <i>object</i>	443
Rozwiązanie z użyciem typów ogólnych	447
Typy ogólne a klasy uogólnione	449
Typy ogólne i nakładanie ograniczeń	450
Tworzenie klasy ogólnej	450
Teoria drzew binarnych	450
Budowanie klasy drzewa binarnego przy użyciu typu ogólnego	454
Tworzenie metody ogólnej	463
Definiowanie metody ogólnej do budowy drzewa binarnego	464
Interfejsy ogólne i niezgodność typów	467
Interfejsy kowariantne	468
Interfejsy kontrawariantne	470
Podsumowanie	473
Krótki przegląd rozdziału 17	473
18 Kolekcje	475
Co to są klasy kolekcji?	475
Klasa kolekcji <i>List<T></i>	477
Klasa kolekcji <i>LinkedList<T></i>	480
Klasa kolekcji <i>Queue<T></i>	481
Klasa kolekcji <i>Stack<T></i>	482
Klasa kolekcji <i>Dictionary<TKey, TValue></i>	484
Klasa kolekcji <i>SortedList<TKey, TValue></i>	485
Klasa kolekcji <i>HashSet<T></i>	486
Inicjowanie kolekcji	488
Metody <i>Find</i> , predykaty i wyrażenia lambda	489
Różne formy wyrażen lambda	491
Porównanie tablic i kolekcji	493
Wykorzystanie klas kolekcji do gry w karty	494
Podsumowanie	498
Krótki przegląd rozdziału 18	499
19 Wyliczanie kolekcji	501
Wyliczanie elementów kolekcji	501
Ręczna implementacja modułu wyliczającego	503
Implementowanie interfejsu <i>IEnumerable</i>	508
Implementowanie modułu wyliczającego przy użyciu iteratora	510
Prosty iterator	510

Definiowanie modułu wyliczającego dla klasy <i>Tree<TItem></i> przy użyciu iteratora	512
Podsumowanie	514
Krótki przegląd rozdziału 19	515
20 Wydzielanie logiki aplikacji i obsługa zdarzeń	517
Co to są delegaty	518
Przykłady delegatów w bibliotece klas .NET Framework	519
Przykład zautomatyzowanej fabryki	520
Implementowanie systemu sterowania fabryką bez użycia delegatów	522
Implementowanie sterowania fabryką przy użyciu delegata	522
Deklarowanie i używanie delegatów	525
Delegaty i wyrażenia lambda	534
Tworzenie adaptera metody	534
Włączanie powiadomień za pomocą zdarzeń	535
Deklarowanie zdarzenia	535
Subskrypcja zdarzenia	536
Anulowanie subskrypcji zdarzenia	537
Zgłaszanie zdarzenia	537
Zdarzenia interfejsu użytkownika	538
Używanie zdarzeń	540
Podsumowanie	546
Krótki przegląd rozdziału 20	546
21 Odpytywanie danych w pamięci przy użyciu wyrażeń w języku zapytań	549
Co to jest LINQ?	549
Używanie LINQ w aplikacjach C#	550
Wybieranie danych	552
Filtrowanie danych	555
Porządkowanie, grupowanie i agregowanie danych	556
Złączanie danych	558
Operatory zapytań	559
Odpytywanie danych w obiektach <i>Tree<TItem></i>	562
LINQ i opóźnione przetwarzanie	568
Podsumowanie	572
Krótki przegląd rozdziału 21	572
22 Przeciążanie operatorów	575
Czym są operatory	575
Ograniczenia operatorów	576
Operatory przeciążone	577
Tworzenie operatorów symetrycznych	578
Złożone instrukcje przypisania	580

Deklarowanie operatorów inkrementujących i dekrementujących	581
Operatory porównań w strukturach i klasach	582
Definiowanie par operatorów	583
Implementowanie operatorów	584
Operatory konwersji	591
Wbudowane metody konwersji	591
Implementowanie własnych operatorów konwersji	592
Tworzenie operatorów symetrycznych – uzupełnienie	593
Zapisywanie operatorów konwersji	594
Podsumowanie	596
Krótki przegląd rozdziału 22	597

Część VIII Tworzenie aplikacji Universal Windows Platform w języku C#

23 Przyspieszanie działania za pomocą zadań	601
Po co stosować wielozadaniowość przy użyciu przetwarzania równoległego? ..	601
Narodziny procesora wielordzeniowego	602
Implementowanie wielozadaniowości w .NET Framework	604
Zadania, wątki i pula wątków	604
Tworzenie, uruchamianie i kontrolowanie zadań	606
Używanie klasy <i>Task</i> do implementacji równoległości	609
Tworzenie abstrakcji zadań za pomocą klasy <i>Parallel</i>	623
Kiedy nie używać klasy <i>Parallel</i>	627
Anulowanie zadań i obsługa wyjątków	630
Mechanizm anulowania kooperatywnego	630
Kontynuowanie w przypadku zadań anulowanych lub przerwanych z powodu wyjątku	644
Podsumowanie	645
Krótki przegląd rozdziału 23	645
24 Skracanie czasu reakcji za pomocą działań asynchronicznych	649
Implementowanie metod asynchronicznych	650
Definiowanie metod asynchronicznych: problem	651
Definiowanie metod asynchronicznych: rozwiązanie	654
Definiowanie metod asynchronicznych zwracających wartości	661
Wskazówki dotyczące metod asynchronicznych	662
Metody asynchroniczne i interfejsy API środowiska Windows Runtime	664
Zadania, alokacje pamięci i wydajność	666
Zrównoleglanie deklaratywnego dostępu do danych za pomocą PLINQ	670
Wykorzystanie PLINQ do poprawy wydajności przy wykonywaniu iteracji po elementach kolekcji	670
Anulowanie zapytania PLINQ	675
Synchronizowanie współbieżnych operacji dostępu do danych	676

Blokowanie danych	679
Elementarne narzędzia synchronizacji umożliwiające koordynowanie zadań	680
Anulowanie synchronizacji	683
Współbieżne klasy kolekcji	683
Wykorzystanie kolekcji współbieżnej i blokady do implementacji dostępu do danych przystosowanego do trybu wielowątkowego	684
Podsumowanie	695
Krótki przegląd rozdziału 24	696
25 Implementowanie interfejsu użytkownika aplikacji Universal Windows Platform	699
Cechy aplikacji Universal Windows Platform	701
Budowa aplikacji UWP przy użyciu szablonu Blank App	704
Implementowanie skalowalnego interfejsu użytkownika	707
Stosowanie stylów do interfejsu użytkownika	738
Podsumowanie	748
Krótki przegląd rozdziału 25	748
26 Wyświetlanie i wyszukiwanie danych w aplikacjach Universal Windows Platform	749
Implementowanie wzorca projektowego Model-View-ViewModel	749
Wyświetlanie danych przy użyciu mechanizmu wiązania danych	750
Modyfikowanie danych przy użyciu wiązania danych	757
Stosowanie wiązania danych do kontrolki <i>ComboBox</i>	762
Tworzenie składnika ViewModel	764
Dodawanie poleceń do składnika ViewModel	768
Wyszukiwanie danych przy użyciu Cortany	779
Dostarczanie odpowiedzi głosowej na polecenia głosowe	792
Podsumowanie	796
Krótki przegląd rozdziału 26	796
27 Dostęp do zdalnej bazy danych z poziomu aplikacji Universal Windows Platform	799
Pobieranie informacji z bazy danych	799
Tworzenie modelu encji	807
Tworzenie i korzystanie z usługi web typu REST	817
Wstawianie, aktualizacja i usuwanie danych za pośrednictwem usługi web typu REST	833
Raportowanie błędów i aktualizacja interfejsu użytkownika	844
Podsumowanie	852
Krótki przegląd rozdziału 27	852
Indeks	855

Podziękowania

No i znów nadeszło coś, co wydaje się stawać swoistym biennale – kolejne, dziewiąte już wydanie tej książki; takie jest bowiem tempo zmian w świecie projektowania oprogramowania! Gdy spoglądam na moje ukochane pierwsze wydanie Kernighana i Ritchie’ego *The C Programming Language*, chwyta mnie nostalgia za starymi dobrymi czasami. W tamtych dziewiczych czasach programowanie dodawało splendoru, a nawet – nie bójmy się tego słowa – było czymś mistycznym. Dziś umiejętność napisania choćby odrobiny kodu staje się, w tej czy innej formie, wymaganiem w wielu miejscach pracy, podobnie jak umiejętność czytania, pisania i dodawania. Romantyczny czas minął, zastąpiony „szarą codziennością”. Jednak choć niekiedy tęsknię za czasami, gdy jeszcze miałem włosy na głowie, a korporacyjny mainframe wymagał pełnoetatowego personelu technicznego umiejącego grzebać w jego bebeczach, zrozumiałem, że gdyby programowanie nadal było zastrzeżone dla wąskiej elity, rynek na książki o C# wyczerpałby się już po pierwszym, góra drugim wydaniu tego tomu. Podniesiony na duchu uruchomiłem zatem swój laptop, wyrzuciłem z myśli marzenia o minionej erze, gdy taka moc obliczeniowa pozwoliłaby na jednoczesne prowadzenie wielu setek statków Apollo na Księżyc i z powrotem, po czym siadłem do pracy nad następnym wydaniem tej książki!

Pomimo faktu, że na okładce figuruje moje nazwisko, to stworzenie tego rodzaju książki z pewnością nie jest zadaniem dla jednego człowieka. Chciałbym w tym miejscu podziękować wymienionym poniżej osobom za ich bezgraniczne wsparcie i pomoc w realizacji tego zadania.

Po pierwsze, Trina MacDonald z Pearson Education, która przyjęła rolę nadzorcy popychającego mnie do działania, i choć uprzejmie, to bezwzględnie przypominała mi dobrze zdefiniowane terminy realizacji poszczególnych etapów. Bez jej energii i uporu projekt ten nigdy by nie wystartował.

Następnie Rick Kughen, nieustrudzony redaktor, który zadbał o to, aby moja pisa-
nina była choć połowicznie zrozumiała, wyszukując brakujące słowa i bezsensowne frazy w tekście.

Następnie David Franson, który miał mało zachęcające zadanie przetestowania kodu i ćwiczeń. Wiem z własnego doświadczenia, jak frustrujące i niewdzięczne może być to zadanie, ale poświęcony przez niego czas i uwagi mogły uczynić tę książkę jedynie lepszą. Oczywiście, dowolne błędy, które pozostały, są wyłącznie moją winą i będę szczęśliwy, gdy czytelnicy powiadomią mnie o tych, które dostrzegą.

Jak zwykle muszę podziękować Dianie, mojej lepszej połowie, która dbała o stałe dostawy gorących napojów (złożonych głównie z kofeiny), gdy zbliżały się końcowe terminy. Diana wykazała się już wcześniej niebywałą cierpliwością, skoro przetrwała dziewięć wydań tej książki; nikomu innemu nie mógłbym zadedykować ani tego, ani żadnego z wcześniejszych wydań. Ostatnio zaczęła biegać. Przypuszczałem, że chodziło jej o poprawę figury, ale niewykluczone, że po prostu woli być poza domem, aby móc pośmiać się głośno, gdy jej nie słyszę!

Wreszcie na koniec muszę wspomnieć o Jamesie i Frankie, którzy oboje już wyfrunęli z rodzinnego gniazda. James stara się uniknąć nabrania akcentu Yorkshire, jednocześnie żyjąc i pracując w Sheffield, ale Frankie pozostała bliżej domu, tak że od czasu do czasu może wpaść do nas i przeczesać lodówkę.

O autorze

JOHN SHARP jest głównym technologiem w firmie Content Master Ltd. z Wielkiej Brytanii. Jest szeroko znanym programistą, konsultantem, projektantem i autorem wielu materiałów szkoleniowych, poradników i książek. Jego przeszło 35-letnie doświadczenie zawodowe obejmuje szeroki zakres różnych technologii, poczynając od programowania w Pascalu dla systemów CP/M i tworzenia aplikacji C/Oracle na różnych wersjach systemów UNIX, aż po projektowanie rozproszonych aplikacji w C# i JavaScript i programowanie dla Windows 10 i Microsoft Azure. Swój czas poświęca również na tworzenie kursów szkoleniowych dla firmy Microsoft, koncentrując się na takich obszarach, jak *Data Science* z wykorzystaniem języków R i Python, przetwarzanie Big Data przy użyciu Spark i CosmosDB oraz skalowalnych architekturach aplikacji.

Wstęp

Język Microsoft Visual C# jest bardzo potężnym, a jednocześnie prostym językiem programowania, przeznaczonym głównie dla programistów, którzy tworzą aplikacje oparte na platformie Microsoft .NET Framework. Visual C# odziedziczył wiele z najlepszych cech języka C++ oraz Microsoft Visual Basic i tylko kilka występujących w tych językach niespójności lub anachronizmów, co zaowocowało powstaniem bardziej przejrzystego i bardziej logicznego języka programowania.

- C# 1.0 miał swój publiczny debiut w roku 2001.
- C# 2.0 wraz z programem Visual Studio 2005 udostępniał kilka ważnych i nowych funkcji, takich jak ogólne typy wyliczeniowe i metody anonimowe.
- C# 3.0, która pojawiła się wraz z opublikowaniem programu Visual Studio 2008, dodał obsługę metod rozszerzających, wyrażeń lambda oraz najważniejszej ze wszystkich nowości – obsługi zapytań w języku LINQ (Language Integrated Query).
- Wprowadzona w roku 2010 wersja C# 4.0 zaoferowała kolejne udoskonalenia w zakresie polepszenia możliwości współdziałania z innymi technologiami i językami programowania. Funkcje te obejmowały obsługę argumentów nazwanych i opcjonalnych, typ *dynamic*, którego użycie wskazywało, że środowisko uruchomieniowe powinno zastosować dla danego obiektu tzw. późne wiązanie (ang. late binding). Bardzo ważną zmianą wprowadzoną do wersji platformy .NET Framework, opublikowanej w tym samym czasie co wersja C# 4.0, były klasy i typy danych składające się na nową bibliotekę równoległego realizowania zadań – TPL (Task Parallel Library). Korzystając z biblioteki TPL można tworzyć wysoko skalowalne aplikacje, które będą w pełni wykorzystywać możliwości wielordzeniowych procesorów.
- W C# 5.0 dodano natywną obsługę dla przetwarzania zadań w sposób asynchroniczny poprzez użycie modyfikatora metody *async* oraz operatora *await*.
- C# 6.0 była kolejnym ulepszeniem zaprojektowanym z myślą o ułatwianiu życia programistom. Te udoskonalenia to między innymi interpolacja łańcuchów (już nigdy nie będziemy musieli korzystać z *String.Format!*), ulepszone sposoby implementacji właściwości czy metody wcielające wyrażenia.
- C# 7.0 dodaje dalsze usprawnienia zwiększające produktywność, a jednocześnie w tej wersji usunięto część anachronizmów, które dotychczas występowały w C#. Na przykład można teraz implementować akcesory właściwości jako wyrażenia wcielające, metody mogą zwracać wiele wartości w postaci krotek, uproszczono użycie parametrów *out*, zaś instrukcje *switch* zostały rozszerzone o obsługę dopasowywania wzorców i typów. Istnieją też inne uaktualnienia, które również przedstawimy w tej książce.

Nie da się zaprzeczyć, że Microsoft Windows 10 jest ważną platformą dla aplikacji tworzonych w C#, ale teraz możemy uruchamiać kod zaprojektowany w C# również w innych systemach operacyjnych, takich jak Linux, dzięki środowisku .NET Runtime. Otwiera to możliwości tworzenia kodu, który może działać w wielu środowiskach. Dodatkowo Windows 10 wspiera aplikacje o wysokim stopniu interaktywności, które mogą współdzielić pomiędzy sobą dane i współpracować ze sobą nawzajem lub łączyć się z usługami działającymi w chmurze. Kluczowym elementem wersji Windows 10 są aplikacje Universal Windows Platform (UWP) – zaprojektowane tak, aby mogły być uruchamiane na dowolnym urządzeniu Windows 10, począwszy od bogato wyposażonego komputera, po laptop, tablet, smartfon, a nawet urządzenia IoT (Internet of Things) z ograniczonymi zasobami. Po opanowaniu podstawowych funkcji języka C#, ważne jest zdobycie umiejętności budowania aplikacji, które mogą być uruchamiane na wszystkich wspomnianych platformach.

Aktywacja głosowa to kolejna funkcja, która zyskała na popularności. System Windows 10 oferuje Cortanę, czyli osobistą asystentkę cyfrową, która może być aktywowana za pomocą głosu*. Możemy zintegrować swoje aplikacje z Cortaną, aby zapewnić im możliwość uczestniczenia w wyszukiwaniu danych i innych operacjach. Mimo komplikacji związanych zazwyczaj z analizą mowy, przygotowanie aplikacji do reagowania na żądania Cortany jest zaskakująco proste i zostało omówione w rozdziale 26. Ponadto chmura stała się tak istotnym elementem architektury wielu systemów, począwszy od dużych systemów korporacyjnych po aplikacje mobilne działające na smartfonach użytkowników, że poświęciliśmy temu aspektowi rozwoju oprogramowania ostatni rozdział książki.

Środowisko programowania oferowane przez pakiet Visual Studio 2017 sprawia, że korzystanie ze wszystkich tych nowych i potężnych funkcji jest bardzo łatwe, a wiele nowych kreatorów i ulepszeń wprowadzonych do najnowszej wersji Visual Studio pozwala znacząco podnieść produktywność programistów. Mamy nadzieję, że korzystanie z tej książki będzie równie przyjemne, jak jej pisanie!

Dla kogo przeznaczona jest ta książka

Książka ta powstała przy założeniu, że Czytelnik ma już pewne doświadczenie w programowaniu i pragnie poznać podstawy programowania w języku C# przy wykorzystaniu środowiska Visual Studio 2017 oraz platformy .NET Framework w wersji 4.6.1. Po przeczytaniu tej książki jej Czytelnicy powinni dysponować dobrą znajomością języka C# i powinni umieć używać tego języka do tworzenia szybkich i skalowalnych aplikacji dla systemu operacyjnego Windows 10.

* Funkcja Cortana nie jest jeszcze dostępna w polskiej wersji systemu Windows 10 (przyp. tłum.).

Dla kogo nie jest przeznaczona ta książka

Niniejsza książka skierowana jest do osób, które dopiero uczą się języka C#, ale nie są nowicjuszami w programowaniu jako takim. Dlatego koncentruje się ona głównie na kwestiach związanych z samym językiem C#. Celem tej książki nie jest dostarczenie wyczerpującego omówienia rozlicznych technologii pozwalających na tworzenie aplikacji przeznaczonych do użytku w dużych przedsiębiorstwach, takich jak ADO.NET, ASP.NET, Windows Communication Foundation lub Windows Workflow Foundation. Czytelnicy oczekujący większej ilości informacji na temat jednej z tych technologii powinni rozważyć lekturę kilku innych tytułów wydanych przez Microsoft Press.

Organizacja książki

Niniejsza książka została podzielona na następujące cztery części:

- Część I, zatytułowana „Wprowadzenie do języka Microsoft Visual C# oraz programu Microsoft Visual Studio 2017” stanowi wprowadzenie do podstawowej składni języka C# oraz środowiska programowania Visual Studio.
- Część II, zatytułowana „Omówienie modelu obiektowego języka C#”, przedstawia więcej szczegółów związanych z tworzeniem i zarządzaniem w języku C# nowymi typami danych, a także wyjaśnia, w jaki sposób należy zarządzać zasobami wskazywanymi przez te typy danych.
- Część III, zatytułowana „Tworzenie rozszerzalnych typów danych w języku C#”, zawiera poszerzone omówienie tych elementów oferowanych przez język C#, które można wykorzystywać do tworzenia typów danych nadających się do używania w wielu różnych aplikacjach.
- Część IV, zatytułowana „Tworzenie aplikacji Universal Windows Platform”, opisuje uniwersalny model programowania w systemie Windows 10 i wyjaśnia, jak używać języka C# do tworzenia interaktywnych aplikacji opartych na tym nowym modelu.

Określenie najlepszego miejsca, od którego należy rozpocząć lekturę tej książki

Niniejsza książka ma za zadanie ułatwić jej Czytelnikom podniesienie swoich umiejętności w kilku podstawowych obszarach. Z książki tej mogą korzystać zarówno początkujący programiści, jak również programiści mający już pewne doświadczenie w innych językach programowania, takich jak C, C++, Java lub Visual Basic. Zamieszczona dalej tabela powinna ułatwić każdemu określenie najlepszego miejsca, od którego należy rozpocząć lekturę tej książki.

Jeżeli jesteś	Wykonaj następujące kroki
Programistą początkującym w dziedzinie programowania zorientowanego obiektowo	1. Zainstaluj pliki używane w opisywanych w tej książce ćwiczeniach, zgodnie z opisem podanym w dalszej części, zatytułowanej „Przykładowe kody źródłowe”. 2. Przeczytaj po kolei wszystkie rozdziały z części I, II i III. 3. Przeczytaj odpowiednie rozdziały z części IV, stosownie do poziomu swojego doświadczenia oraz potrzeb.
Programistą dobrze obeznanym z proceduralnymi językami programowania, takimi jak np. język C, ale nowicjuszem w C#	1. Zainstaluj pliki używane w opisywanych w tej książce ćwiczeniach, zgodnie z opisem podanym w dalszej części, zatytułowanej „Przykładowe kody źródłowe”. 2. Zapoznaj się pobieżnie z treścią pierwszych pięciu rozdziałów, aby uzyskać ogólny obraz języka C# oraz możliwości programu Visual Studio 2017, a następnie skoncentruj się na rozdziałach od 6 do 22. 3. Przeczytaj odpowiednie rozdziały z części IV, stosownie do poziomu swojego doświadczenia oraz poziomu swoich potrzeb.
Programistą mającym już doświadczenie w innych zorientowanych obiektowo językach programowania, takich jak np. C++ lub Java, i pragnącym nauczyć się języka C#	1. Zainstaluj pliki używane w opisywanych w tej książce ćwiczeniach, zgodnie z opisem podanym w dalszej części, zatytułowanej „Przykładowe kody źródłowe”. 2. Zapoznaj się pobieżnie z treścią pierwszych siedmiu rozdziałów, aby uzyskać ogólny obraz języka C# oraz możliwości programu Visual Studio 2017, a następnie skoncentruj się na rozdziałach od 7 do 22. 3. Przeczytaj rozdziały z części IV, aby dowiedzieć się, w jaki sposób tworzyć aplikacje Universal Windows Platform.
Programistą znającym język Visual Basic i pragnącym nauczyć się języka C#	1. Zainstaluj pliki używane w opisywanych w tej książce ćwiczeniach, zgodnie z opisem podanym w dalszej części, zatytułowanej „Przykładowe kody źródłowe”. 2. Przeczytaj po kolei wszystkie rozdziały z części I, II i III. 3. Przeczytaj rozdziały z części IV, aby dowiedzieć się, w jaki sposób tworzyć aplikacje Universal Windows Platform. 4. Przeczytaj krótkie powtórzenia, znajdujące się na końcu każdego rozdziału, aby szybko uzyskać potrzebne informacje na temat konkretnych konstrukcji języka C# oraz funkcji programu Visual Studio 2017.

Jeżeli jesteś	Wykonaj następujące kroki
Zainteresowany znalezieniem potrzebnych informacji, związanych z prezentowanymi w tej książce ćwiczeniami	1. Skorzystaj z indeksu lub spisu treści, aby znaleźć potrzebne informacje na konkretny temat. 2. Przeczytaj krótkie powtórzenia, znajdujące się na końcu każdego rozdziału, aby szybko zapoznać się z omawianymi w danym rozdziale technikami oraz elementami składni.

Większość rozdziałów tej książki zawiera użyteczne przykłady ułatwiające zrozumienie omawianych koncepcji i pojęć. Każdy Czytelnik, niezależnie od tego, które części tej książki będą dla niego najbardziej interesujące, powinien koniecznie pobrać i zainstalować na swoim komputerze omawiane aplikacje przykładowe.

Konwencje i cechy charakterystyczne stosowane w tej książce

Zawarte w tej książce informacje są prezentowane przy użyciu konwencji poprawiających czytelność oraz ułatwiających śledzenie toku narracji.

- Każde ćwiczenie składa się z serii zadań, prezentowanych jako seria ponumerowanych kroków (1, 2, itd.) zawierających opis wszystkich działań niezbędnych do ukończenia danego ćwiczenia.
- Elementy umieszczone w ramkach z ikoną na marginesie i etykietą, taką jak np. „Uwaga”, zawierają dodatkowe informacje lub opis alternatywnego sposobu wykonania danego kroku.
- Tekst, który powinien zostać wpisany przez Czytelnika, wyróżniany jest wytłuszczoną czcionką.

Wymagania systemowe

Do wykonania prezentowanych w tej książce ćwiczeń potrzebny będzie komputer spełniający następujące wymagania sprzętowe i programowe:

- System operacyjny Windows 10 (Home, Professional Education lub Enterprise) wersja 1507 lub wyższa.
- Najnowsze wydanie Visual Studio Community 2017, Visual Studio Professional 2017 lub Visual Studio Enterprise 2017. Jako minimum należy podczas instalacji wybrać następujące środowiska:
 - ☐ Projektowanie dla Universal Windows Platform
 - ☐ Projektowanie .NET

- ❑ Projektowanie ASP.NET i Web
- ❑ Projektowanie Azure
- ❑ Przechowywanie i przetwarzanie danych
- ❑ Projektowanie międzyplatformowe .NET Core



UWAGA Wszystkie ćwiczenia i przykłady kodu w tej książce zostały opracowane i przetestowane przy użyciu Visual Studio Community 2017. Wszystkie powinny działać bez modyfikacji w Visual Studio Professional 2017 oraz Visual Studio Enterprise 2017.

- Komputer z procesorem 1.8 GHz lub szybszym (zalecany procesor dwurdzeniowy lub lepszy).
- 2 GB (zalecane 4 GB; w przypadku pracy na maszynie wirtualnej należy dodać 512 MB).
- 10 GB wolnego miejsca na dysku twardym.
- Dysk twardy o prędkości obrotowej 5400 RPM lub szybszy (zalecany dysk SSD).
- Karta graficzna kompatybilna ze standardem DirectX 9, o rozdzielczości 1024 × 768 lub wyższej.
- Połączenie z publiczną siecią Internet, wymagane do pobrania oprogramowania lub przykładów dla poszczególnych rozdziałów.

W zależności od konfiguracji używanego systemu Windows, zainstalowanie i skonfigurowanie oprogramowania Visual Studio 2017 może wymagać posiadania uprawnień lokalnego administratora.

Ponadto trzeba włączyć na komputerze tryb dewelopera, aby móc tworzyć i uruchamiać aplikacje UWP. Dodatkowe informacje dotyczące osiągnięcia tego celu znaleźć można w artykule „Enable Your Device for Development” (w jęz. angielskim) pod adresem <https://msdn.microsoft.com/library/windows/apps/dn706236.aspx>.

Przykładowe kody źródłowe

W większości rozdziałów tej książki zamieszczone zostały ćwiczenia pozwalające na interaktywne wypróbowanie nowych umiejętności, nabytych podczas lektury danego rozdziału. Wszystkie te przykładowe projekty można pobrać w wersji wyjściowej (tj. takiej, od której rozpoczyna się dane ćwiczenie) lub w wersji końcowej (tj. takiej, jaką otrzymalibyśmy po starannym wykonaniu całego ćwiczenia) z następującej strony internetowej:

<https://aka.ms/VisCSharp9e/downloads>

UWAGA Oprócz pobrania przykładowych kodów źródłowych należy pamiętać także o konieczności zainstalowania w systemie pakietu oprogramowania Visual Studio 2017. Należy także zainstalować najnowsze wersje pakietów serwisowych, które będą dostępne dla oprogramowania Visual Studio oraz dla systemu operacyjnego Windows.



Instalowanie przykładowych kodów źródłowych

Aby zainstalować na komputerze przykładowe kody źródłowe, które umożliwią praktyczne wykonywanie opisywanych w tej książce ćwiczeń, należy wykonać następujące kroki.

1. Rozpakuj plik Code_Files_Sharp_SBS_9E.zip, pobrany ze strony web powiązanej z książką do swojego katalogu Dokumenty.
2. Zaakceptuj postanowienia licencyjne w wyświetlonym monicie.

UWAGA Jeśli tekst umowy licencyjnej nie zostanie wyświetlony, z treścią tej umowy można zapoznać się na tej samej stronie webowej, z której pobrany został plik z przykładowymi kodami źródłowymi.



Korzystanie z przykładowych kodów źródłowych

Wszystkie rozdziały tej książki zawierają dokładne instrukcje wyjaśniające, kiedy i w jaki sposób należy korzystać z przykładowych kodów źródłowych dla danego rozdziału. Gdy nadejdzie pora użycia kolejnego przykładu, podane zostaną dokładne instrukcje, w jaki sposób należy otworzyć odpowiednie pliki.

WAŻNE Niektóre projekty są zależne od pakietów NuGet, które nie zostały dołączone do przykładów kodu. Te pakiety zostają automatycznie pobrane w trakcie pierwszej kompilacji projektu. W konsekwencji, jeśli Czytelnik otworzy projekt i zacznie go analizować przed kompilacją, Visual Studio może zgłaszać wiele błędów spowodowanych nierozwiązanymi odwołaniami. Po skompilowaniu projektu problemy powinny zostać rozwiązane i błędy powinny zniknąć.



Dla tych Czytelników, którzy chcieliby poznać więcej szczegółów, poniżej zamieszczona została lista wszystkich przykładowych projektów i rozwiązań programu Visual Studio 2017, pogrupowanych według katalogów, w których się one znajdują. W wielu przypadkach przykładowe projekty dostępne są w wersji z plikami w stanie początkowym, umożliwiającym samodzielne przeprowadzenie danego ćwiczenia zgodnie z podanym opisem oraz w wersji końcowej, której można używać jako punktu

odniesienia. Gotowe wersje projektów z każdego rozdziału znajdują się w folderach o nazwie uzupełnionej przyrostkiem „-Complete” (Gotowe).

Projekt	Opis
Chapter 1	
TextHello	Pierwszy projekt w języku C#. Projekt ten demonstruje kolejne kroki procesu tworzenia prostego programu wyświetlającego tekst pozdrowienia.
Hello	Ten projekt otwiera okno, które prosi użytkownika o podanie imienia, a następnie wyświetla spersonalizowane powitanie.
Chapter 2	
PrimitiveDataTypes	Projekt demonstrujący sposób deklarowania zmiennych każdego z podstawowych typów danych, sposób przypisywania tym zmiennym wartości oraz sposób wyświetlania wartości tych zmiennych w oknie programu.
MathsOperators	Projekt wprowadzający operatory arytmetyczne (+ – * / %).
Chapter 3	
Methods	Projekt polegający na ponownym przeanalizowaniu kodu poprzedniego projektu MathsOperators i wykorzystaniu metod do uporządkowania kodu źródłowego.
DailyRate	Projekt demonstrujący proces pisania własnych metod, ich uruchamiania oraz krokowego śledzenia przy pomocy debugera z programu Visual Studio 2017.
DailyRate Using Optional Parameters	Projekt demonstrujący sposób definiowania metod akceptujących parametry opcjonalne oraz wywoływania tych metod przy użyciu nazwanych argumentów.
Chapter 4	
Selection	Projekt demonstrujący kaskadowe użycie instrukcji if do zaimplementowania złożonej logiki, polegającej np. na porównywaniu dwóch dat.
SwitchStatement	Prosty program wykorzystujący instrukcję switch do przekształcania znaków na ich reprezentację w formacie XML.
Chapter 5	
WhileStatement	Projekt demonstrujący użycie instrukcji while do odczytywania kolejnych linii z pliku źródłowego i wyświetlania każdej z nich w umieszczonym na formularzu, osobnym polu tekstowym.
DoStatement	Projekt wykorzystujący instrukcję do do przekształcenia liczby dziesiętnej na jej reprezentację ósemkową.

Projekt	Opis
Chapter 6	
MathsOperators	Projekt pokazujący na przykładzie projektu MathsOperators z rozdziału 2, zatytułowanego „Zmienne, operatory i wyrażenia”, jak różne nieobsłużone wyjątki mogą doprowadzić do przerwania działania programu. Stabilność działania aplikacji zostaje poprawiona poprzez użycie słów kluczowych try i catch.
Chapter 7	
Classes	Projekt demonstrujący podstawy definiowania własnych klas, uzupełniania ich o publiczne konstruktory, metody oraz pola prywatne. Projekt ten demonstruje również sposób tworzenia nowych instancji klasy za pomocą słowa kluczowego new oraz sposób definiowania statycznych pól i metod.
Chapter 8	
Parameters	Program wyjaśniający różnicę pomiędzy parametrami typu wartościowego a parametrami typu referencyjnego. Program ten demonstruje także użycie słów kluczowych ref i out.
Chapter 9	
StructsAndEnums	Projekt definiujący strukturalny typ danych (przy użyciu słowa kluczowego struct), służący do reprezentowania daty kalendarzowej.
Chapter 10	
Cards	Projekt demonstrujący zastosowanie tablic do zamodelowania puli kart w grze karcianej.
Chapter 11	
ParamsArray	Projekt demonstrujący użycie słowa kluczowego params do tworzenia pojedynczej metody mogącej akceptować dowolną liczbę argumentów typu int.
Chapter 12	
Vehicles	Projekt wykorzystujący interfejsy do utworzenia prostej hierarchii klas pojazdów. Projekt ten demonstruje także sposób definiowania metod wirtualnych.
ExtensionMethod	Projekt demonstrujący sposób tworzenia metody rozszerzającej dla typu int, której zadaniem będzie przekształcanie dziesiętnej liczby całkowitej w jej reprezentację w innym systemie liczenia.
Chapter 13	
Drawing	Projekt implementujący część pakietu graficznego. Projekt ten wykorzystuje interfejsy do zdefiniowania metod udostępnianych i implementowanych przez różne figury geometryczne.
Drawing Using Interfaces	Projekt stanowi wstęp do rozszerzania projektu Drawing poprzez utworzenie klas abstrakcyjnych, oferujących funkcjonalność wspólną dla różnych figur geometrycznych.

Projekt	Opis
Chapter 14	
GarbageCollectionDemo	Projekt demonstrujący sposób bezpiecznego zwalniania zasobów w środowisku wielowątkowym poprzez odpowiednie używanie wzorca Dispose.
Chapter 15	
Drawing Using Properties	Projekt rozszerzający aplikację w projekcie Drawing opracowaną w rozdziale 13 poprzez hermetyzację wybranych danych wewnątrz klasy przy użyciu właściwości.
AutomaticProperties	Projekt demonstrujący sposób tworzenia automatycznych właściwości klas oraz wykorzystywania ich do inicjalizowania nowych instancji klasy.
Chapter 16	
Indexers	Projekt demonstrujący zastosowanie dwóch obiektów indeksujących (indeksatorów): jednego służącego do wyszukiwania numeru telefonu osoby o podanym nazwisku i drugiego służącego do wyszukiwania nazwiska osoby o podanym numerze telefonu.
Chapter 17	
BinaryTree	Rozwiązanie demonstrujące sposób używania ogólnych typów danych do utworzenia struktury bezpiecznej pod względem typu, mogącej zawierać elementy dowolnego typu.
BuildTree	Projekt demonstrujący sposób użycia ogólnych typów danych do zaimplementowania metody bezpiecznej pod względem typu, mogącej akceptować parametry dowolnego typu.
Chapter 18	
Cards	Projekt zawierający zmodyfikowaną wersję kodu z rozdziału 10, demonstrujący sposób użycia kolekcji do zamodelowania puli kart rozdanych pomiędzy graczy w grze karcianej.
Chapter 19	
BinaryTree	Projekt demonstrujący sposób zaimplementowania ogólnego interfejsu typu IEnumerator<T>, który umożliwia utworzenie obiektu wyczeniowego dla ogólnej klasy Tree.
IteratorBinaryTree	Rozwiązanie wykorzystujące iterator do wygenerowania typu wyczeniowego dla ogólnej klasy Tree.
Chapter 20	
Delegates	Projekt demonstrujący sposób wykorzystania delegacji do oddzielenia metody od logiki korzystającej z tej metody aplikacji. Następnie projekt zostaje rozszerzony, aby zademonstrować sposób wykorzystania zdarzeń do powiadamiania obiektów o ważnych wydarzeniach oraz sposób przechwytywania tego typu zdarzeń i wykonywania związanych z nimi akcji.

Projekt	Opis
Chapter 21	
QueryBinaryTree	Projekt demonstrujący sposób pobierania danych z obiektu typu drzewo binarne, przy użyciu zapytań w języku LINQ.
Chapter 22	
ComplexNumbers	Projekt definiujący nowy typ danych, modelujący liczby złożone i implementujący kilka typowych operatorów używanych dla tego typu danych.
Chapter 23	
GraphDemo	Projekt generujący skomplikowany wykres i wyświetlający go na formularzu UWP. W tym projekcie niezbędne obliczenia wykonywane są przez jeden wątek.
Parallel GraphDemo	Jest to wersja projektu GraphDemo, w którym do wyodrębnienia procesu tworzenia i zarządzania zadaniami użyta została klasa Parallel.
GraphDemo With Cancellation	Projekt pokazujący, jak zaimplementować możliwość zatrzymywania zadań w kontrolowany sposób, zanim same zakończą swoje działanie.
ParallelLoop	Przykładowa aplikacja pokazująca, kiedy nie należy używać klasy Parallel do tworzenia i uruchamiania kilku zadań równocześnie.
Chapter 24	
GraphDemo	Jest to wersja projektu GraphDemo z rozdziału 23, w której w celu asynchronicznego wykonania obliczeń potrzebnych do wygenerowania wykresu zastosowano słowo kluczowe async oraz operator await.
PLINQ	Projekt demonstrujący kilka przykładów wykorzystywania technologii PLINQ do pobierania danych, przy użyciu zadań równoległych.
CalculatePI	Projekt wykorzystujący algorytm próbkowania statystycznego do obliczenia przybliżonej wartości liczby pi. Projekt ten wykorzystuje zadania równoległe.
Chapter 25	
Customers	Ten projekt implementuje skalowalny interfejs użytkownika, który poddaje się skalowaniu dla różnych rozdzielczości ekranu i różnych kształtów formularza. Interfejs użytkownika wykorzystuje style XAML do zmiany czcionki oraz wyświetlanego przez aplikację obrazu tła.

Projekt	Opis
Chapter 26	
DataBinding	Ta wersja projektu Customers wyświetla informacje o klientach, pobierając je ze źródła danych przy użyciu mechanizmu wiązania danych. Projekt ten demonstruje również sposób implementacji interfejsu <i>INotifyPropertyChanged</i> , pozwalającego na aktualizowanie informacji o kliencie z poziomu interfejsu użytkownika i odsyłanie wykonanych zmian z powrotem do źródła danych.
ViewModel	W tej wersji projektu Customers zaimplementowano metodologię programowania MVVM (Model-View-ViewModel), co pozwoliło na oddzielenie interfejsu użytkownika od logiki pobierającej dane ze źródła danych.
Cortana	Ten projekt integruje aplikację Customers z funkcją Cortana. Użytkownik będzie mógł przy pomocy poleceń głosowych wyszukiwać klientów według ich nazwiska.
Chapter 27	
Web Service	Rozwiązanie obejmujące aplikację web dostarczającą usługi typu ASP.NET Web Service, używanej przez aplikację Customers do pobierania danych klientów z bazy danych serwera SQL Server. Podczas dostępu do bazy danych usługa web używa modelu encji utworzonego przy użyciu technologii Entity Framework.

Errata i wsparcie techniczne

Dołożono wszelkich starań, mających na celu zapewnienie dokładności tej książki oraz towarzyszących jej treści. Informacje o wszelkich błędach, które zostały dostrzeżone i zgłoszone już po opublikowaniu tej książki, dostępne są na stronie wydawnictwa Microsoft Press:

<https://aka.ms/VisCSharp9e/errata>

W przypadku wykrycia nowego błędu można go zgłosić za pomocą tej samej, wymienionej powyżej strony.

W razie potrzeby uzyskania dodatkowej pomocy technicznej związanej z tą książką prosimy o skontaktowanie się z działem pomocy technicznej wydawnictwa Microsoft Press Book Support poprzez wysłanie wiadomości email na adres:

mspinput@microsoft.com.

Prosimy pamiętać, że pod podanymi adresami nie jest oferowana pomoc techniczna dla oprogramowania firmy Microsoft.

CZĘŚĆ I

Wprowadzenie do języka Visual C# i Microsoft Visual Studio 2017

Wstępna część książki przedstawia kluczowe aspekty języka C# i demonstruje podstawowe techniki budowania aplikacji w Visual Studio 2017.

Z części I będzie się można dowiedzieć, jak tworzyć nowe projekty w Visual Studio oraz jak deklarować zmienne, wykorzystywać operatory do tworzenia wartości, wywoływać metody i pisać wiele instrukcji przydatnych w procesie implementowania programów C#. Będzie się można również dowiedzieć, jak obsługiwać wyjątki i stosować debugger Visual Studio do przechodzenia przez kod i wykrywania problemów, które mogą uniemożliwiać prawidłowe działanie aplikacji.

ROZDZIAŁ 1

Wprowadzenie do języka C#

Po ukończeniu tego rozdziału Czytelnik będzie potrafił:

- Korzystać ze środowiska programowania Microsoft Visual Studio 2017.
- Utworzyć aplikację konsolową w języku C#.
- Wyjaśnić cel stosowania przestrzeni nazw.
- Utworzyć prostą aplikację graficzną w języku C#.

Rozdział ten stanowi wprowadzenie do tematyki związanej z programem Visual Studio 2017, środowiskiem programowania oraz zestawem narzędzi stworzonych z myślą o ułatwieniu programiście tworzenia aplikacji przeznaczonych dla systemu Microsoft Windows. Program Visual Studio 2017 jest idealnym narzędziem do tworzenia kodu w języku C# i oferuje wiele różnorodnych funkcji, o których dowiemy się więcej w trakcie lektury tej książki. W tym rozdziale użyjemy programu Visual Studio 2017 do utworzenia kilku prostych aplikacji w języku C#, które będą stanowić wstęp do tworzenia wysoko funkcjonalnych rozwiązań dla systemu Windows.

Rozpoczynamy programowanie przy użyciu środowiska Visual Studio 2017

Visual Studio 2017 to rozbudowane środowisko programowania, oferujące funkcjonalność potrzebną przy tworzeniu zarówno małych, jak i dużych projektów w języku C#, przeznaczonych dla systemu Windows. Możliwe jest nawet konstruowanie projektów, które w gładki i bezproblemowy sposób łączą ze sobą moduły napisane przy użyciu różnych języków programowania, takich jak np. C++, Visual Basic lub F#. Nasze pierwsze ćwiczenie polegać będzie na uruchomieniu środowiska programowania Visual Studio 2017 i utworzeniu aplikacji konsolowej.

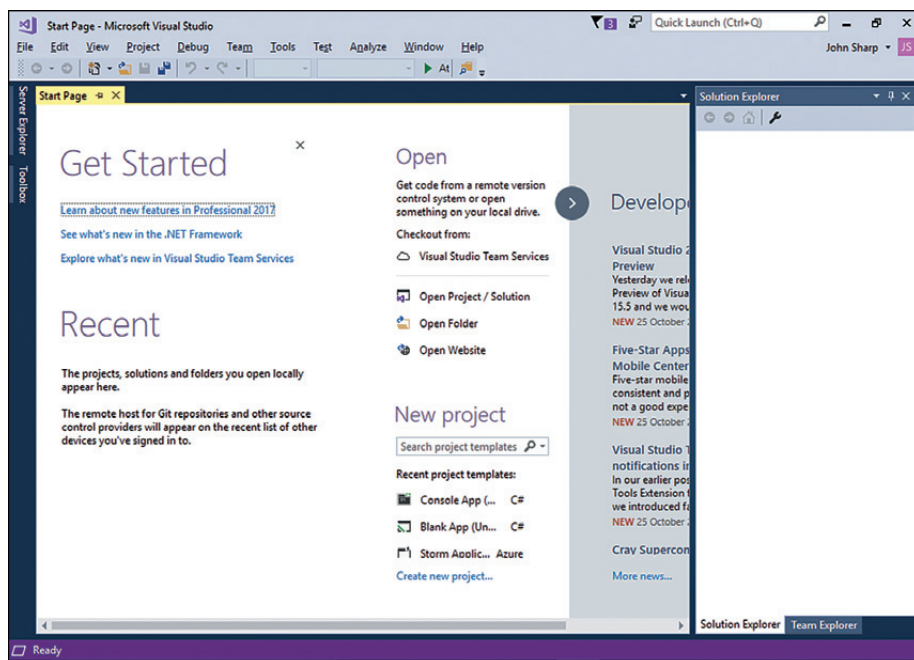
UWAGA Aplikacja konsolowa to aplikacja, która nie oferuje graficznego interfejsu użytkownika (GUI – ang. Graphical User Interface), lecz jest uruchamiana w oknie wiersza poleceń.



➔ Tworzenie aplikacji konsolowej przy użyciu Visual Studio 2017

1. W pasku zadań systemu Windows kliknij Start, wpisz Visual Studio 2017, a następnie naciśnij klawisz Enter.

Spowoduje to uruchomienie programu Visual Studio 2017 i wyświetlenie przez ten program strony Start, takiej jak ta pokazana poniżej (w zależności od używanej edycji programu Visual Studio 2017, strona Start na komputerze użytkownika może wyglądać nieco inaczej):

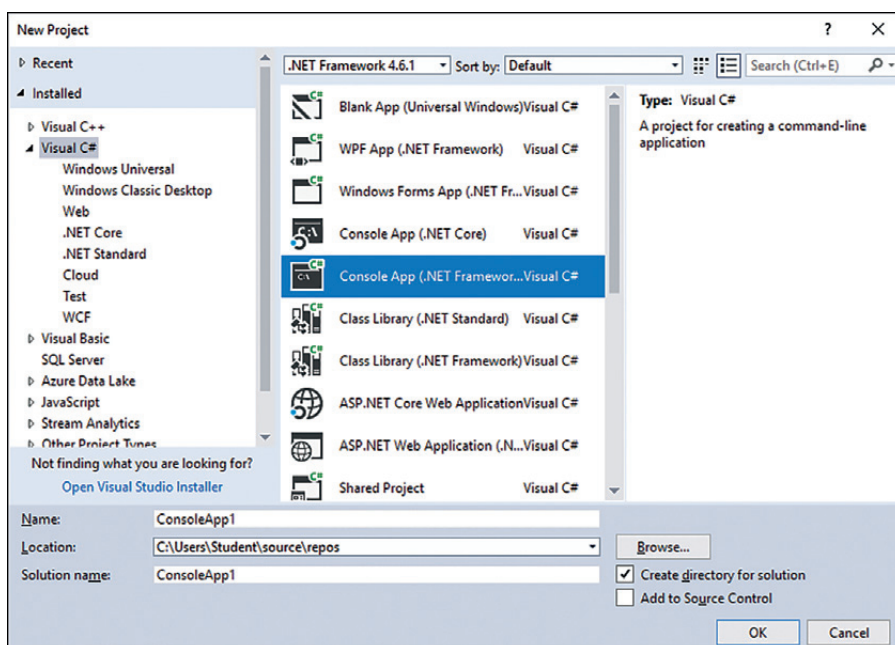


2. Wskaż w menu File (Plik) menu podrzędne New (Nowy), a następnie kliknij polecenie Project (Projekt).

Spowoduje to otwarcie okna dialogowego New Project (Nowy projekt). Okno to zawierać będzie listę szablonów, których można użyć jako punkt wyjściowy przy tworzeniu nowej aplikacji. Szablony widoczne w tym oknie dialogowym podzielone są na kategorie zależne od używanego języka programowania oraz rodzaju tworzonej aplikacji.

3. W panelu po lewej stronie rozwiń węzeł Installed (Zainstalowane), rozwiń węzeł Templates (Szablony), a następnie kliknij element Visual C#. Sprawdź, czy w polu rozwijanej listy, znajdującym się w górnej części środkowego panelu, wybrana jest opcja .NET Framework 4.6.1, a następnie kliknij Console Application (Aplikacja konsolowa).

UWAGA Upewnij się, że wybrałeś Console App (.NET Framework), a nie Console App (.NET Core). Szablonu .NET Core używa się do budowania aplikacji przenośnych, które działają również w innych systemach operacyjnych, takich jak Linux. Jednak aplikacje .NET Core nie udostępniają pełnego zakresu funkcjonalności osiągalnych w pełnym środowisku .NET Framework.



4. W polu Location (Lokalizacja) wpisz `C:\Users\TwojaNazwa\Documents\Microsoft Press\VCSBS\Chapter 1`. Frazę *TwojaNazwa* zastąp własną nazwą użytkownika w systemie Windows.

UWAGA Dla skrócenia zapisu, w dalszej części tej książki, zamiast odwoływać się do ścieżki `C:\Users\TwojaNazwa\Documents`, będziemy pisać po prostu o folderze Dokumenty użytkownika.



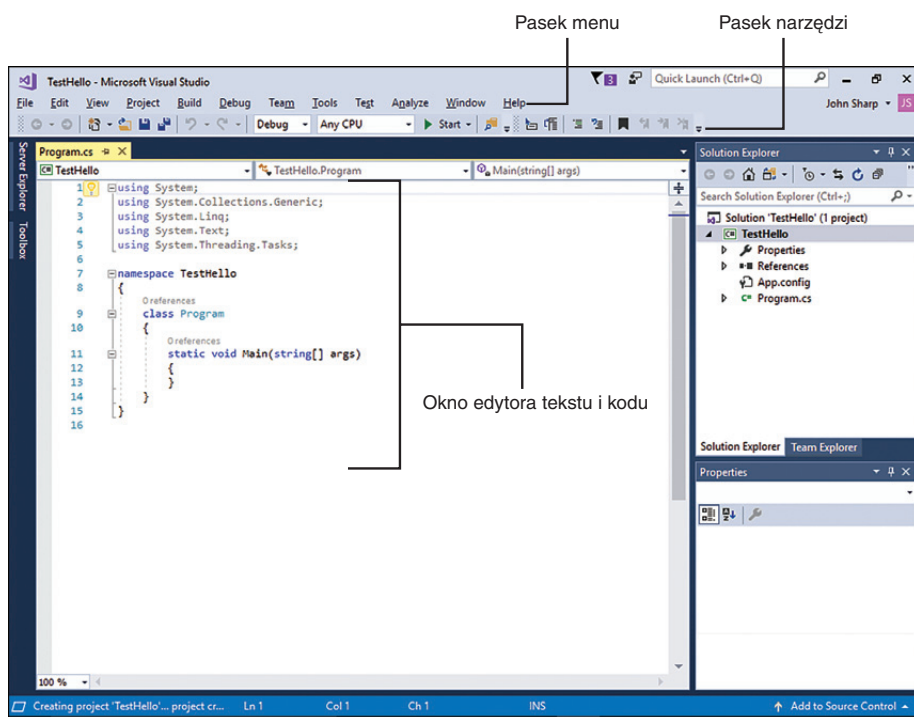
WSKAZÓWKA Jeśli określony folder nie istnieje, zostanie stworzony przez Visual Studio 2017.



5. W polu Name (Nazwa) wpisz tekst `TestHello`, nadpisując znajdującą się w tym polu nazwę `ConsoleApplication1`.

- Upewnij się, że pole wyboru opcji Create Directory for Solution (Utwórz katalog dla rozwiązania) jest zaznaczone oraz że pole wyboru Add To Source Control (Dodaj do kontroli źródła) jest odznaczone, a następnie kliknij przycisk OK.

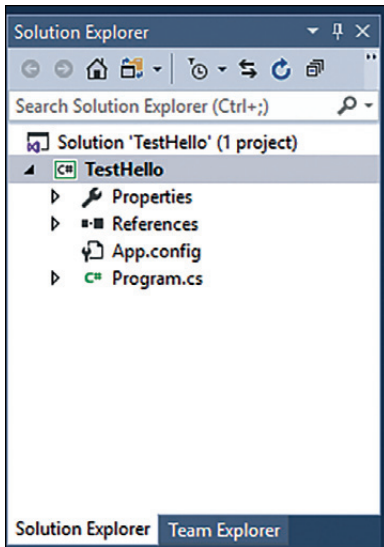
Program Visual Studio utworzy nowy projekt korzystając z szablonu Console Application. Następnie Visual Studio wyświetli początkowy kod aplikacji tak, jak to zostało pokazane na poniższym rysunku:



Pasek menu, znajdujący się w górnej części ekranu, zapewnia dostęp do różnych funkcjonalności używanych w środowisku programowania. Wszelkie polecenia oraz różne pozycje menu można uruchamiać przy pomocy klawiatury lub myszy, dokładnie w taki sam sposób, w jaki odbywa się to we wszystkich programach opartych na systemie Windows. Poniżej paska menu znajduje się pasek narzędziowy, który zawiera przyciski będące skrótami umożliwiającymi uruchamianie najczęściej używanych komend.

W oknie edytora kodu i tekstu, zajmującym główną część ekranu, wyświetlana jest zawartość plików źródłowych. Jeśli podczas pracy z projektami złożonymi z kilku plików edytowany jest więcej niż jeden plik źródłowy, to każdy z tych plików posiadać będzie swoją własną zakładkę oznaczoną nazwą danego pliku. W celu przywołania wybranego pliku źródłowego na pierwszy plan okna edytora kodu i tekstu wystarczy kliknąć zakładkę z nazwą tego pliku.

Po prawej stronie okna wyświetlany jest panel Solution Explorer (Eksplorator rozwiązań), obok edytora kodu i tekstu:



W panelu Solution Explorer wyświetlane są między innymi nazwy związanych z projektem plików. Przywołanie wybranego pliku źródłowego na pierwszy plan okna edytora kodu i tekstu jest również możliwe poprzez dwukrotne kliknięcie tego pliku źródłowego w panelu eksploratora rozwiązań.

Zanim przystąpimy do pisania kodu źródłowego, zapoznajmy się najpierw z plikami wyświetlanymi w panelu eksploratora rozwiązań, które zostały utworzone przez program Visual Studio 2017 jako część nowego projektu:

- ❑ **Solution (Rozwiązanie) 'TestHello'** Jest to plik leżący na najwyższym poziomie hierarchii rozwiązania. Każde rozwiązanie może zawierać jeden lub więcej projektów, a program Visual Studio 2017 tworzy plik rozwiązania, aby ułatwić organizowanie projektów. Jeśli użyjemy Eksploratora plików do sprawdzenia zawartości katalogu Dokumenty\Microsoft Press\VCSBS\Chapter 1\TestHello, to przekonamy się, że faktycznie plik ten nosi nazwę TestHello.sln.
- ❑ **TestHello** Jest to plik projektu w języku C#. Każdy plik projektu zawiera odwołania do jednego lub kilku plików zawierających kod źródłowy lub inne elementy projektu, np. takie jak obrazy graficzne. Całość kodu źródłowego używanego w jednym projekcie musi być napisana w tym samym języku programowania. W programie Eksplorator Windows plik ten jest widoczny pod swoją faktyczną nazwą TestHello.csproj i znajduje się w katalogu \Microsoft Press\VCSBS\Chapter 1\TestHello\TestHello.

- ❑ **Properties (Właściwości)** Jest to folder będący częścią projektu TestHello. Po rozwinięciu tego folderu (w tym celu należy kliknąć strzałkę znajdującą się obok nazwy Properties) przekonamy się, że zawiera on plik o nazwie AssemblyInfo.cs. Plik AssemblyInfo.cs to specjalny plik, który umożliwia dodawanie do programu różnych atrybutów, takich jak np. nazwa autora, data utworzenia programu itp. Możliwe jest także określenie dodatkowych atrybutów, zmieniających sposób działania programu. Omówienie sposobów korzystania z tych atrybutów wykracza jednak poza ramy tej książki.
- ❑ **References (Odwołania)** Ten folder zawiera odwołania do bibliotek skompilowanego kodu, które mogą być używane przez tworzoną aplikację. Podczas kompilacji kodu źródłowego napisanego w języku C# następuje konwersja tego kodu na bibliotekę, której zostanie nadana unikatowa nazwa. W środowisku Microsoft .NET Framework biblioteki te nazywane są *asemblacjami*. Programiści mogą używać asemlacji do umieszczania w nich napisanych przez siebie użytecznych fragmentów kodu w sposób umożliwiający ich dystrybuowanie i używanie przez innych programistów we własnych aplikacjach. Jeśli rozwiemy folder References, to przekonamy się, że zawiera on zestaw domyślnych bibliotek dodanych do projektu przez program Visual Studio 2017. Te asemlacje zapewniają dostęp do wielu powszechnie wykorzystywanych funkcji platformy .NET Framework i zostały dostarczone przez firmę Microsoft wraz z Visual Studio 2017. Podczas wykonywania zamieszczonych w tej książce ćwiczeń będziemy mieli okazję zapoznać się dokładniej z wieloma z tych asemlacji.
- ❑ **App.config** Jest to plik konfiguracyjny aplikacji. Plik ten jest plikiem opcjonalnym i nie zawsze musi występować. Plik konfiguracyjny umożliwia określanie ustawień, które będą mogły być wykorzystywane przez uruchomioną aplikację do modyfikowania sposobu jej działania, takich jak np. wersja platformy .NET Framework używana do uruchamiania danej aplikacji. Więcej informacji na temat tego pliku zostanie podanych w dalszych rozdziałach tej książki.
- ❑ **Program.cs** Jest to plik źródłowy w języku C#, który jest wyświetlany w oknie edytora kodu i tekstu bezpośrednio po utworzeniu projektu. W pliku tym będziemy zapisywać kod tworzonej aplikacji konsolowej. Plik ten zawiera także kod dodany do niego automatycznie przez program Visual Studio 2017, który wkrótce dokładniej przeanalizujemy.

Piszemy pierwszy program

Plik `Program.cs` definiuje klasę o nazwie *Program*, która zawiera metodę o nazwie *Main*. W języku C# cały kod wykonywalny musi być zdefiniowany wewnątrz metody, a wszystkie metody muszą należeć do pewnej klasy lub struktury. Więcej informacji na temat klas znajduje się w rozdziale 7, „Tworzenie i zarządzanie klasami oraz obiektami”, a więcej informacji na temat struktur znaleźć można w rozdziale 9, „Tworzenie typów wartościowych przy użyciu wyliczeń oraz struktur”.

Metoda *Main* określa tzw. punkt wejścia do programu. Metoda ta musi być zdefiniowana w sposób określony w klasie *Program*, tj. jako metoda statyczna, gdyż w przeciwnym razie środowisko .NET Framework mogłoby nie rozpoznać tej metody jako punktu wejścia do programu. Szczegóły budowy metod zostaną omówione w rozdziale 3, „Tworzenie metod i stosowanie zasięgów zmiennych”, a więcej informacji na temat metod statycznych znajduje się we wspomnianym już rozdziale 7.

WAŻNE W języku C# są rozróżniane małe i wielkie litery. Metoda *Main* musi mieć nazwę pisaną wielką literą.

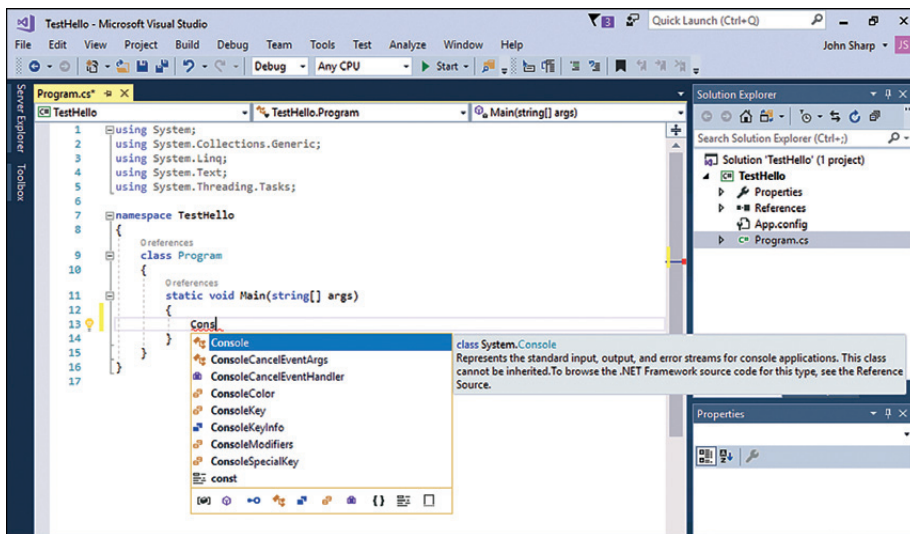


W kolejnych ćwiczeniach napiszemy kod wyświetlający w oknie konsoli komunikat „Hello World!” (Witaj świecie!), zbudujemy i uruchomimy naszą aplikację konsolową Hello World oraz poznamy sposób wykorzystywania przestrzeni nazw do dzielenia kodu na różne elementy.

➔ Pisanie kodu przy użyciu funkcji Microsoft IntelliSense

1. W oknie edytora kodu i tekstu, wyświetlającym zawartość pliku `Program.cs`, umieść kursor wewnątrz metody *Main*, bezpośrednio za otwierającym nawiasem klamrowym, {, a następnie naciśnij klawisz Enter, aby utworzyć nową linię.
2. W nowej linii wpisz słowo `Console`; jest to nazwa jeszcze jednej klasy zawartej w jednym z asembliacji dołączonych automatycznie do naszej aplikacji. Klasa ta oferuje metody pozwalające na wyświetlanie komuników w oknie konsoli oraz na odczytywanie danych wprowadzanych z klawiatury.

Po wpisaniu litery `C`, będącej pierwszą literą słowa *Console*, wyświetlona zostanie lista funkcji IntelliSense. Lista ta zawierać będzie wszystkie słowa kluczowe języka C# oraz typy danych, które są poprawne w danym kontekście. Możesz albo kontynuować wpisywanie słowa, albo odszukać je na liście i dwukrotnie kliknąć myszą. Po wpisaniu liter `Cons` funkcja IntelliSense automatycznie podświetli na liście element *Console* i wówczas do jego wybrania wystarczy wciśnięcie klawisza Tab lub Enter.



Metoda *Main* powinna teraz wyglądać następująco:

```
static void Main(string[] args)
{
    Console
}
```



UWAGA Klasa *Console* jest klasą wbudowaną.

3. Wpisz znak kropki, bezpośrednio po słowie *Console*.

Spowoduje to wyświetlenie nowej listy funkcji IntelliSense, na której wyświetlane będą nazwy metod, właściwości oraz pól klasy *Console*.

4. Przewiń w dół zawartość tej listy, zaznacz na niej metodę *WriteLine*, a następnie wciśnij klawisz Enter. Możesz również wpisywać kolejne litery, W, r, i, t, e, L, aż do zaznaczenia na liście metody *WriteLine*, a następnie wciśnięcie klawisza Enter. Listę funkcji IntelliSense zostanie zamknięta, a do pliku źródłowego zostanie dodane słowo *WriteLine*. Metoda *Main* powinna teraz wyglądać następująco:

```
static void Main(string[] args)
{
    Console.WriteLine
}
```

5. Wpisz znak nawiasu otwierającego, (. Spowoduje to wyświetlenie kolejnej podpowiedzi funkcji IntelliSense.

Podpowiedź ta zawierać będzie listę parametrów akceptowanych przez metodę *WriteLine*. W rzeczywistości metoda *WriteLine* jest tzw. *metodą przeciążoną*,

co oznacza, że klasa *Console* zawiera więcej niż jedną metodę o nazwie *WriteLine* – faktycznie klasa ta zawiera aż 19 różnych wersji tej metody. Każda z wersji metody *WriteLine* pozwala na wyświetlanie na ekranie różnych typów danych. (Metody przeciążone zostaną omówione dokładniej w rozdziale 3). Metoda *Main* powinna teraz wyglądać następująco:

```
static void Main(string[] args)
{
    Console.WriteLine(
}
```

Wskazówka Klikanie znajdujących się w okienku podpowiedzi strzałek skierowanych w górę i w dół pozwala na przewijanie listy pomiędzy różnymi przeciążonymi wersjami metody *WriteLine*.



6. Wpisz znak nawiasu zamykającego, `)`, a po nim znak średnika, `;`.

Metoda *Main* powinna teraz wyglądać następująco:

```
static void Main(string[] args)
{
    Console.WriteLine();
}
```

7. Przesuń kursor i wpisz pomiędzy znakami nawiasów występujących po nazwie metody *WriteLine*, tekst `"Hello World!"`, włącznie ze znakami cudzysłowów.

Metoda *Main* powinna teraz wyglądać następująco:

```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
}
```

Wskazówka Warto wyrobić sobie nawyk, by dopełniające pary znaków – takie jak nawiasy zwykłe (`()`) oraz klamrowe (`{}`) – wpisywać razem, jeszcze przed wypełnieniem ich treścią. Można łatwo zapomnieć w wpisaniu znaku zamykającego, jeśli odłożymy to do czasu po wprowadzeniu zawartości.



Ikony funkcji IntelliSense

Po wpisaniu kropki po nazwie klasy funkcja IntelliSense wyświetla nazwy wszystkich elementów składowych tej klasy (jej członków). Z lewej strony nazwy każdego takiego elementu znajduje się ikona określająca jego rodzaj. Poniżej pokazane zostały typowe ikony wraz z ich znaczeniem:

Ikona	Znaczenie
	Metoda (zostanie omówiona w rozdziale 3)
	Właściwość (zostanie omówiona w rozdziale 15)
	Klasa (zostanie omówiona w rozdziale 7)
	Struktura (zostanie omówiona w rozdziale 9)
	Zmienna wyliczeniowa (zostanie omówiona w rozdziale 9)
	Metoda rozszerzająca (zostanie omówiona w rozdziale 12)
	Interfejs (zostanie omówiony w rozdziale 13)
	Delegacja (zostanie omówiona w rozdziale 17)
	Zdarzenie (zostanie omówione w rozdziale 17)
	Przestrzeń nazw (zostanie omówiona w następnej części tego rozdziału)

Podczas wpisywania kodu w innym kontekście można spotkać się także z innymi ikonami funkcji IntelliSense.

W kodzie źródłowym często można spotkać linie zawierające dwa znaki ukośnika, //, po których następuje zwykły tekst. Są to komentarze – są one ignorowane przez kompilator, ale są bardzo użyteczne dla programistów, ponieważ ułatwiają dokumentowanie faktycznego sposobu działania programu. Przykładowo:

```
Console.ReadLine(); // Czekaj, aż użytkownik naciśnie klawisz Enter
```

Kompilator pomija cały tekst, począwszy od dwóch znaków ukośnika aż do końca danej linii. Możliwe jest także dodawanie komentarzy składających się z wielu linii, które rozpoczynają się od ukośnika i gwiazdki: /*. Po napotkaniu tych znaków kompilator pomija wszystko, aż do napotkania sekwencji gwiazdki i ukośnika */, która

może znajdować się w pliku źródłowym nawet o wiele linii dalej. Zdecydowanie zachęcamy do dokumentowania własnego kodu źródłowego przy użyciu niezbędnej liczby wyczerpujących komentarzy.

➔ Budowanie i uruchamianie aplikacji konsolowej

1. Wybierz z menu Build (Kompilowanie) polecenie Build Solution (Kompiluj rozwiązanie).

Wykonanie tej akcji spowoduje skompilowanie kodu w języku C# i utworzenie wykonywalnego programu. Poniżej okna edytora kodu i tekstu wyświetlone zostanie okno Output (Dane wyjściowe).

Wskazówka Wykonanie tej akcji spowoduje skompilowanie kodu w języku C# i utworzenie wykonywalnego programu. Poniżej okna edytora kodu i tekstu wyświetlone zostanie okno Output.



W oknie Output powinien wówczas zostać wyświetlony komunikat podobny pokazanego poniżej, informujący o przebiegu procesu kompilacji programu:

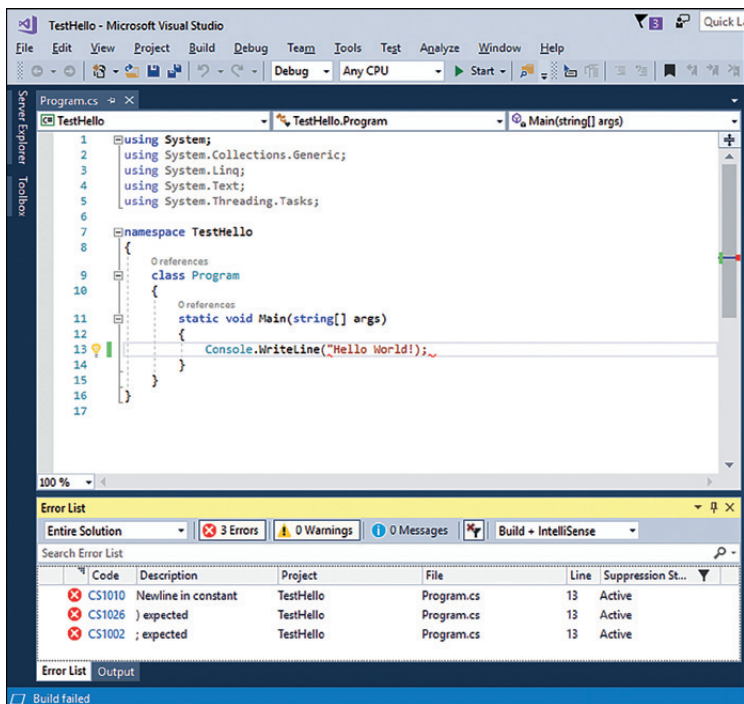
```
1>----- Build started: Project: TestHello, Configuration: Debug Any CPU -----
1> TestHello -> C:\Users\John\Dokumenty\Microsoft Press\Visual CSharp Step
By Step\Chapter
1\TestHello\TestHello\bin\Debug\TestHello.exe
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

Jeśli w kodzie źródłowym popełnione zostały jakieś błędy, to zostaną one wymienione w oknie Error List (Lista błędów). Zamieszczony na następnej stronie przykład pokazuje, co by się stało, gdybyśmy w instrukcji *WriteLine* zapomnieli wpisać zamykającego znaku cudzysłowu po tekście Hello World! Należy zwrócić uwagę na fakt, że czasami jedna pomyłka może prowadzić do wygenerowania kilku błędów kompilacji.

Wskazówka Dwukrotne kliknięcie wybranego elementu w oknie Error List spowoduje umieszczenie kursora w linii, która spowodowała dany błąd. Należy także zauważyć, że w kodzie źródłowym program Visual Studio podkreśla czerwoną falistą linią wszystkie wiersze, które nie będą mogły zostać poprawnie skompilowane.



Jeśli wszystkie poprzednie instrukcje zostały wykonane dokładnie i z należytą starannością, to nie powinniśmy otrzymać żadnych błędów ani ostrzeżeń, a proces tworzenia programu powinien zakończyć się sukcesem.



WSKAZÓWKA Nie ma potrzeby jawnego zapisywania pliku źródłowego przed rozpoczęciem procesu budowy/kompilacji, ponieważ polecenie Build Solution powoduje automatyczne zapisanie pliku źródłowego. Gwiazdka widniejąca obok nazwy pliku na zakładce okna edytora kodu i tekstu oznacza, że dany plik został zmodyfikowany od czasu jego ostatniego zapisania na dysku.

- Wybierz z menu Debug (Debugowanie) polecenie Start Without Debugging (Uruchom bez debugowania).

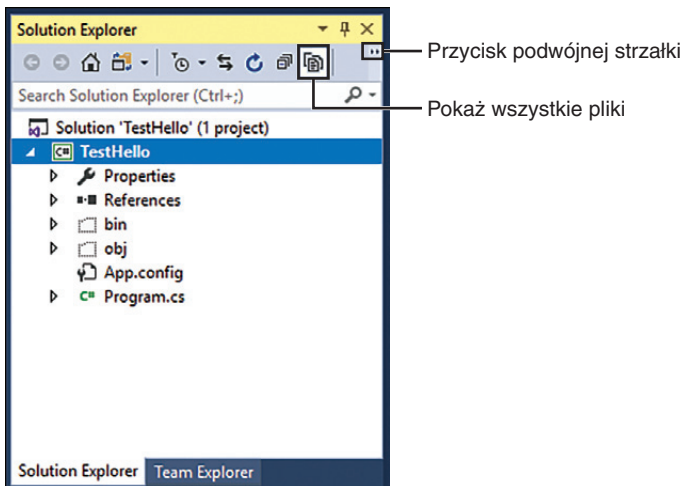
Spowoduje to otwarcie okna wiersza poleceń i uruchomienie programu. Uruchomiony program wyświetli komunikat Hello World! i będzie oczekiwać na wcisnięcie przez użytkownika dowolnego klawisza, tak jak to zostało pokazane na poniższym rysunku:



UWAGA Tekst monitu „Press any key to continue...” (Wciśnij dowolny klawisz, aby kontynuować...) został wygenerowany przez program Visual Studio; w naszym projekcie nie ma żadnego kodu powodującego wypisanie tego komunikatu. Jeśli program zostałby uruchomiony przy użyciu polecenia Start Debugging (Rozpocznij debugowanie) z menu Debug, to aplikacja również zostałaby uruchomiona, ale jej okno wyjściowe zostałoby natychmiast zamknięte, bez oczekiwania na wciśnięcie przez użytkownika dowolnego klawisza.



3. Upewnij się, że okno wiersza poleceń, w którym wyświetlane są rezultaty działania programu, jest aktywnym oknem (ma tzw. fokus), a następnie wciśnij klawisz Enter. Spowoduje to zamknięcie okna wiersza poleceń i powrót do środowiska programowania Visual Studio 2017.
4. W oknie Solution Explorer kliknij projekt *TestHello* (projekt, a nie rozwiązanie o tej samej nazwie), a następnie kliknij przycisk Show All Files (Pokaż wszystkie pliki) znajdujący się na pasku narzędziowym eksploratora rozwiązań. Zwróć uwagę, że aby wyświetlić ten przycisk, konieczne może być kliknięcie przycisku z podwójną strzałką >>, znajdującego się po prawej stronie paska narzędziowego eksploratora rozwiązań.



Ponad plikiem *Program.cs* zauważymy elementy o nazwach *bin* i *obj*. Wpisy te odpowiadają folderom *bin* i *obj* w folderze projektu (*Microsoft Press\Visual CSharp Step By Step\Chapter 1\TestHello\TestHello*). Foldery te są tworzone przez program Visual Studio podczas budowania aplikacji i zawierają wykonywalną wersję programu oraz pewne dodatkowe pliki, używane podczas procesu budowania aplikacji oraz podczas jej debugowania.

5. W oknie Solution Explorer rozwiń gałąź *bin*. Ukaze się wówczas kolejny folder o nazwie *Debug*.



UWAGA W gałęzi tej może również znajdować się folder o nazwie *Release*.

6. Korzystając z okna Solution Explorer rozwiń folder *Debug*.

Po rozwinięciu tego folderu pojawi się w nim kilka kolejnych elementów, wśród których znajdować się będzie plik o nazwie *TestHello.exe*. Plik ten to skompilowany program i to właśnie ten plik jest uruchamiany po wybraniu z menu *Debug* polecenia *Start Without Debugging*. Pozostałe trzy pliki zawierają informacje, które są używane przez program Visual Studio 2017 podczas uruchamiania programu w trybie debugowania – po wybraniu z menu *Debug* polecenia *Start Debugging*.

Przestrzenie nazw

Prezentowany dotychczas przykład to bardzo mały program. Małe programy mogą jednak bardzo szybko rozrosnąć się do dużo większych rozmiarów. Wraz z powiększaniem się rozmiarów programu pojawiają się dwa główne problemy. Po pierwsze, w przypadku dużych programów utrzymywanie ich kodu oraz zrozumienie sposobu działania staje się trudniejsze niż w przypadku małych programów. Po drugie, większa ilość kodu zwykle oznacza większą liczbę klas, zawierających większą liczbę metod, to z kolei oznacza konieczność posługiwania się większą liczbą nazw. Wraz ze wzrostem liczby nazw rośnie również prawdopodobieństwo niepowodzenia kompilowania projektu spowodowanego konfliktem dwóch lub więcej nazw. Przykładem może być próba utworzenia dwóch klas o takiej samej nazwie. Sytuacja komplikuje się jeszcze bardziej, gdy tworzony program odwołuje się do asemblacji stworzonych przez innych programistów, którzy również posługują się wieloma różnymi nazwami.

W przeszłości programiści starali się rozwiązywać problem konfliktów nazw, poprzedzając je pewnego rodzaju kwalifikatorem (lub korzystając ze zbioru takich kwalifikatorów). Takie rozwiązanie nie było jednak najlepsze, ponieważ nie było skalowalne. Nazwy stawały się coraz dłuższe, a programiści spędzali coraz więcej czasu na wpisywaniu kodu, zamiast na jego pisaniu (to nie to samo) oraz na ciągłym odczytywaniu coraz bardziej niezrozumiałych nazw.

Przestrzenie nazw pomagają w rozwiązaniu tego problemu poprzez stworzenie kontenera dla innych identyfikatorów, takich jak np. nazwy klas. Dwie klasy o takiej samej nazwie nie zostaną ze sobą pomyłone, jeśli będą należeć do różnych przestrzeni nazw. Przykładowo, utworzenie klasy *Pozdrowienia* w przestrzeni nazw *TestHello* przy użyciu słowa kluczowego *namespace* może wyglądać następująco:

```
namespace TestHello
{
```

```
class Pozdrowienia
{
    ...
}
```

Do utworzonej w ten sposób klasy *Pozdrowienia* możemy odwoływać się w swoich programach przy użyciu nazwy *TestHello.Pozdrowienia*. Jeśli inny programista również utworzy klasę *Pozdrowienia* w innej przestrzeni nazw, np. w przestrzeni *NowaPrzestrzenNazw* i asemblacja zawierająca tę klasę zostanie zainstalowana na naszym komputerze, to nasze programy nadal będą działać zgodnie z oczekiwaniami, ponieważ używają one klasy *TestHello.Pozdrowienia*. Jeśli zechcemy odwołać się do klasy *Pozdrowienia* utworzonej przez tego innego programistę, to będziemy musieli posłużyć się nazwą *NowaPrzestrzenNazw.Pozdrowienia*.

Dobre praktyki programowania wymagają, aby wszystkie tworzone klasy były definiowane przy użyciu przestrzeni nazw, do której należą i środowisko Visual Studio 2017 stosuje się do tego zalecenia, używając nazwy projektu jako nazwy przestrzeni nazw najwyższego poziomu. Do zaleceń tych stosuje się również biblioteka klas platformy .NET Framework; każda zdefiniowana przez tę platformę klasa istnieje w pewnej przestrzeni nazw. Przykładowo, klasa *Console* istnieje w przestrzeni nazw *System*. Oznacza to, że faktyczna pełna nazwa tej klasy to *System.Console*.

Oczywiście, jeśli korzystając z klasy musielibyśmy za każdym razem wpisywać jej pełną nazwę, to sytuacja nie byłaby wcale lepsza niż wówczas, gdybyśmy stosowali jedynie jakąś formę przedrostków lub po prostu używali globalnie unikatowych nazw, takich jak np. *SystemConsole*. Na szczęście problem ten można rozwiązać stosując w swoich programach dyrektywę *using*. Jeśli cofniemy się do projektu *TestHello* otwartego w Visual Studio 2017 i przyjrzymy się widocznej w oknie edytora kodu i tekstu zawartości pliku *Program.cs*, to zauważymy na początku tego pliku następujące linie:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

Linie te to dyrektywy *using*. Dyrektywa *using* powoduje włączenie wskazanej przestrzeni nazw do aktualnego zasięgu (ang. scope). W dalszej części kodu, znajdującej się w tym samym pliku źródłowym, nie ma już potrzeby jawnego kwalifikowania nazw obiektów nazwami przestrzeni nazw, z których pochodzą te obiekty. Pięć przestrzeni nazw pokazanych w tym przykładzie zawiera klasy, które są używane na tyle często, że program Visual Studio 2017 dodaje je automatycznie przy użyciu dyrektywy *using* do każdego nowo tworzonego projektu. Jeśli zachodzi potrzeba korzystania także z innych przestrzeni nazw, to oczywiście możliwe jest dodanie na początku pliku źródłowego kolejnych dyrektyw *using*.



UWAGA Można zauważyć, że niektóre z dyrektyw *using* jest wyszarzonych. Dyrektywy te odpowiadają przestrzeniom nazw, których aplikacja aktualnie nie używa. Jeśli nie będziemy ich potrzebować również po zakończeniu pisania kodu, można je będzie bezpiecznie usunąć. Jednak jeśli będziemy później potrzebować elementów zawartych w tych przestrzeniach nazw, konieczne będzie ich ponowne dodanie do projektu.

Koncepcja przestrzeni nazw zostanie zaprezentowana dokładniej w poniższym ćwiczeniu.

➔ Ręczne wpisywanie długich nazw

1. W wyświetlanym w oknie edytora kodu i tekstu pliku *Program.cs* oznacz jako komentarz znajdującą się na początku pliku pierwszą dyrektywę *using*, tak jak to zostało pokazane poniżej:

```
//using System;
```

2. Wybierz z menu Build polecenie Build Solution.
Proces kompilacji zakończy się niepowodzeniem, a w oknie Error List wyświetlony zostanie następujący komunikat błędu:

```
The name 'Console' does not exist in the current context.  
(Nazwa 'Console' nie istnieje w bieżącym kontekście).
```

3. Kliknij dwukrotnie komunikat błędu wyświetlany w oknie Error List.
Spowoduje to zaznaczenie w pliku źródłowym *Program.cs* identyfikatora, który spowodował wystąpienie tego błędu.
4. Popraw w oknie edytora kodu i tekstu treść metody *Main* tak, by używała ona w pełni kwalifikowanej nazwy metody *System.Console*.

Metoda *Main* powinna wyglądać następująco:

```
static void Main(string[] args)
{
    System.Console.WriteLine("Hello World!");
}
```



UWAGA Po wpisaniu znaku kropki po słowie *System* funkcja IntelliSense wyświetli nazwy wszystkich elementów istniejących w przestrzeni nazw *System*.

5. Wybierz z menu Build polecenie Build Solution.
Tym razem kompilacja projektu zakończy się powodzeniem. Jeśli tak się nie stanie, to należy sprawdzić, czy metoda *Main* wygląda dokładnie tak, jak w pokazanym wcześniej fragmencie kodu, a następnie ponowić próbę kompilacji kodu.

6. Uruchom aplikację, wybierając z menu Debug polecenie Start Without Debugging, aby przekonać się, że nadal działa ona poprawnie.
7. Po uruchomieniu programu i wypisaniu przez niego w oknie konsoli tekstu Hello World! wciśnij klawisz Enter, aby powrócić do Visual Studio 2017.

Przestrzenie nazw a wykonywalne pliki binarne

Dyrektywa *using* powoduje po prostu włączenie elementów ze wskazanej przestrzeni nazw do bieżącego zasięgu (ang. *scope*), uwalniając programistę od konieczności używania w swoim kodzie w pełni kwalifikowanych nazw klas. Klasy są kompilowane w *asemblacje* (ang. *assemblies*). Binarna asemlacja to plik, który zwykle ma rozszerzenie *.dll*, choć ściśle rzecz biorąc, są nimi również programy wykonywalne z rozszerzeniem *.exe*.

Asemlacja może zawierać wiele klas. Klasy składające się na bibliotekę klas platformy .NET Framework, takie jak np. *System.Console*, są dostarczane w asemlacjach instalowanych na komputerze razem z programem Visual Studio. Jak wkrótce się przekonamy, biblioteka klas .NET Framework zawiera tysiące różnych klas. Gdyby wszystkie te klasy znajdowały się w jednym zestawie binarnym, miałby on olbrzymie rozmiary i był trudny do utrzymania (gdyby Microsoft uaktualnił tylko jedną metodę pojedynczej klasy, musiałby na nowo rozesłać całą bibliotekę klas do wszystkich programistów!).

Z tego względu biblioteka klas platformy .NET Framework została podzielona na kilka mniejszych asemlacji, odpowiadających różnym obszarom funkcjonalnym, z którymi związane są zawarte w nich klasy. Istnieje więc „zasadnicza” asemlacja (zestaw ten nosi nazwę *mscorlib.dll*), który zawiera wszystkie powszechnie używane klasy, takie jak np. *System.Console*, a także inne asemlacje, zawierające klasy służące do manipulowania bazami danych, korzystania z usług webowych, tworzenia graficznego interfejsu użytkownika itd. Jeśli zamierzamy skorzystać z klasy zawartej w asemlacji, konieczne jest dodanie w projekcie odwołania do niej. Następnie można dodać w kodzie źródłowym dyrektywę *using*, która spowoduje włączenie do zasięgu elementów z przestrzeni nazw zawartych w danej asemlacji.

Należy w tym miejscu podkreślić, że relacja pomiędzy binarną asemlacją a przestrzenią nazw niekoniecznie musi być relacją typu 1:1. Pojedyncza asemlacja może zawierać klasy zdefiniowane w wielu różnych przestrzeniach nazw, a pojedyncza przestrzeń nazw może rozciągać się na kilka asemlacji. Przykładowo klasy oraz inne elementy z przestrzeni nazw *System* zostały faktycznie zaimplementowane jako kilka różnych asemlacji, między innymi *mscorlib.dll*, *System.dll* oraz *System.Core.dll*. Wszystko to może początkowo wydawać się bardzo zagnieżdżone, ale szybko można się do tego przyzwyczaić.

Ciąg dalszy na stronie następnej

Ciąg dalszy ze strony poprzedniej

Szablon wybrany podczas tworzenia nowej aplikacji za pomocą programu Visual Studio powoduje automatyczne dołączenie odwołań do właściwych zestawów binarnych. Jeśli dla przykładu rozwiniemy folder *References* w oknie Solution Explorer otwartego projektu *TestHello*, zobaczymy, że użycie szablonu *Console application* spowodowało dołączenie odwołań do asembacji o nazwach: *Microsoft.CSharp*, *System*, *System.Core*, *System.Data*, *System.Data.DataSetExtensions*, *System.Net.Http*, *System.Xml* oraz *System.Xml.Linq*. Pewnym zaskoczeniem może być brak na tej liście zestawu *mscorlib.dll*. Wynika to z faktu, że zestaw ten zawiera podstawowe funkcjonalności czasu wykonania i musi być używany przez wszystkie aplikacje korzystające z platformy .NET Framework. Folder *References* zawiera tylko opcjonalne asembacje i jeśli zachodzi taka potrzeba, to możliwe jest dodawanie nowych i usuwanie istniejących asembacji z tego folderu.

Dodatkowe odwołania do asembacji można dodać do projektu klikając prawym klawiszem myszy folder *References* i wybierając polecenie *Add Reference* (Dodaj odwołanie) – zadanie to zostanie wykonane podczas ćwiczeń zamieszczonych w dalszej części tego rozdziału. Operacja usunięcia asembacji polega na kliknięciu prawym klawiszem myszy wybranego zestawu wyświetlanego w folderze *References*, a następnie wybraniu z menu kontekstowego polecenia *Remove* (Usuń).

Tworzenie aplikacji graficznej

Dotychczas użyliśmy programu Visual Studio 2017 do utworzenia i uruchomienia prostej aplikacji konsolowej. Środowisko programowania Visual Studio 2017 zawiera także wszystko, czego będziemy potrzebować do tworzenia graficznych aplikacji dla systemu Windows 10. Szablony te noszą nazwę aplikacji Universal Windows Platform (UWP), ponieważ umożliwiają tworzenie programów, które można uruchamiać na dowolnym urządzeniu systemu Windows, takim jak komputer, tablet czy telefon. Graficzny interfejs użytkownika aplikacji Windows można projektować interaktywnie. Oprogramowanie Visual Studio 2017 wygeneruje następnie odpowiednie instrukcje programu, implementujące zaprojektowany interfejs użytkownika.

Visual Studio 2017 oferuje dwa rodzaje widoków dla aplikacji graficznych: *widok projektowy* (Design view) oraz *widok kodu* (Code view). Okno edytora kodu i tekstu pozwala na modyfikowanie i utrzymywanie kodu oraz logiki tworzonej aplikacji graficznej, a widok projektowy pozwala na graficzne rozmieszczanie elementów interfejsu użytkownika. Możliwe jest swobodne przełączenie się pomiędzy tymi dwoma widokami.

Następny zestaw ćwiczeń demonstruje tworzenie aplikacji graficznej przy użyciu Visual Studio 2017. Program ten wyświetlać będzie prosty formularz zawierający

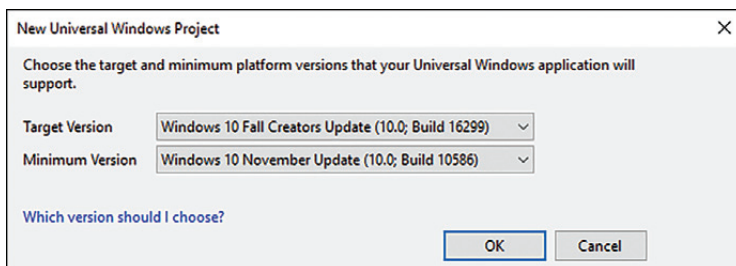
pole tekstowe, w którym można będzie wpisać swoje imię, oraz przycisk służący do wyświetlania spersonalizowanego tekstu pozdrowienia w oknie komunikatu.

DODATKOWE INFORMACJE Więcej wskazówek oraz informacji dotyczących specyfiki tworzenia aplikacji UWP znaleźć można w części IV tej książki.

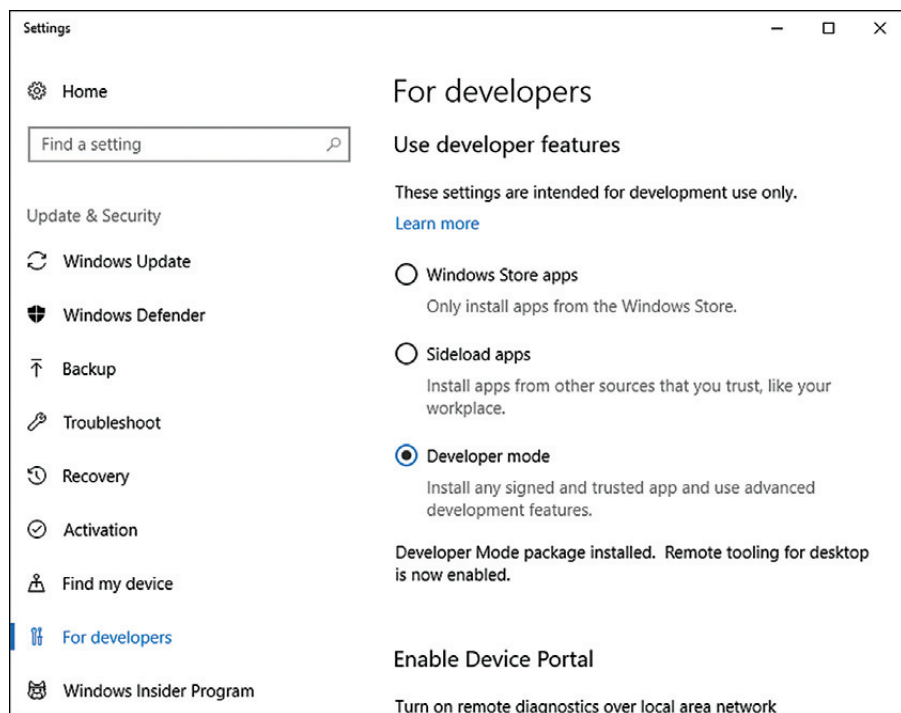


➔ Tworzenie aplikacji graficznej w środowisku Visual Studio 2017

1. Uruchom program Visual Studio 2017, jeśli nie jest on jeszcze uruchomiony.
2. W menu File wskaż menu podrzędne New, a następnie wybierz z niego polecenie Project (Projekt).
Spowoduje to otwarcie okna dialogowego New Project.
3. W lewym panelu rozwiń węzeł Installed (jeśli nie jest on już rozwinięty), rozwiń węzeł Templates, rozwiń folder Visual C#, a następnie kliknij Windows Universal.
4. Kliknij znajdującą się w środkowym panelu ikonę Blank App (Windows Universal) (Pusta aplikacja (aplikacja uniwersalna)).
5. Upewnij się, że ścieżka podana w polu Location wskazuje katalog \Microsoft Press\VCBSB\Chapter 1 w folderze Dokumenty.
6. Wpisz w polu Name tekst Hello.
7. Upewnij się, że zaznaczone zostało pole wyboru opcji Create Directory For Solution, a następnie kliknij przycisk OK.
8. W tym momencie pojawi się okno dialogowe z pytaniem, w której kompilacji (*build*) Windows 10 ma być uruchamiana tworzona aplikacja. Późniejsze kompilacje Windows 10 udostępniają więcej i nowszych funkcjonalności. Firma Microsoft zaleca wybieranie najnowszej wersji Windows 10 jako docelowej, ale jeśli projektujemy aplikację wewnętrzną przedsiębiorstwa, która musi również działać w starszych wersjach, powinniśmy wybrać najstarszą aktualnie używaną wersję Windows 10 jako minimalną. Nie należy jednak automatycznie zaznaczać najstarszej istniejącej wersji, gdyż może to ograniczyć dostępną funkcjonalność aplikacji:



Jeśli aplikacja UWP jest tworzona po raz pierwszy, to w tym miejscu może pojawić się także monit o włączenie trybu dewelopera dla systemu Windows 10 i zostanie wyświetlony ekran ustawień systemowych. Wybierz Developer Mode (Tryb dewelopera). Po dokonaniu wyboru pojawi się okno dialogowe potwierdzenia, gdyż działanie to powoduje pominięcie pewnych funkcji zabezpieczeń systemu Windows. Kliknij Yes (Tak). System pobierze i zainstaluje pakiet Developer Mode, udostępniający dodatkowe funkcjonalności debugowania aplikacji UWP:



UWAGA Zewnętrzne aplikacje, które nie są pobierane ze Sklepu Windows, mogą potencjalnie ujawniać dane osobowe i tworzyć inne zagrożenia bezpieczeństwa, ale włączenie trybu dewelopera jest konieczne, aby móc kompilować i testować własne, nie-standardowe aplikacje.

9. Powróć do Visual Studio. Po utworzeniu aplikacji przejrzyj zawartość okna Solution Explorer.

Nazwa użytego szablonu aplikacji jest nieco myląca – choć Blank App oznacza „pustą aplikację”, w rzeczywistości szablon ten dostarcza kilka gotowych plików, zawierających całkiem pokaźną ilość kodu źródłowego. Jeśli na przykład otworzymy folder MainPage.xaml, znajdziemy w nim plik z kodem w języku C# o nazwie

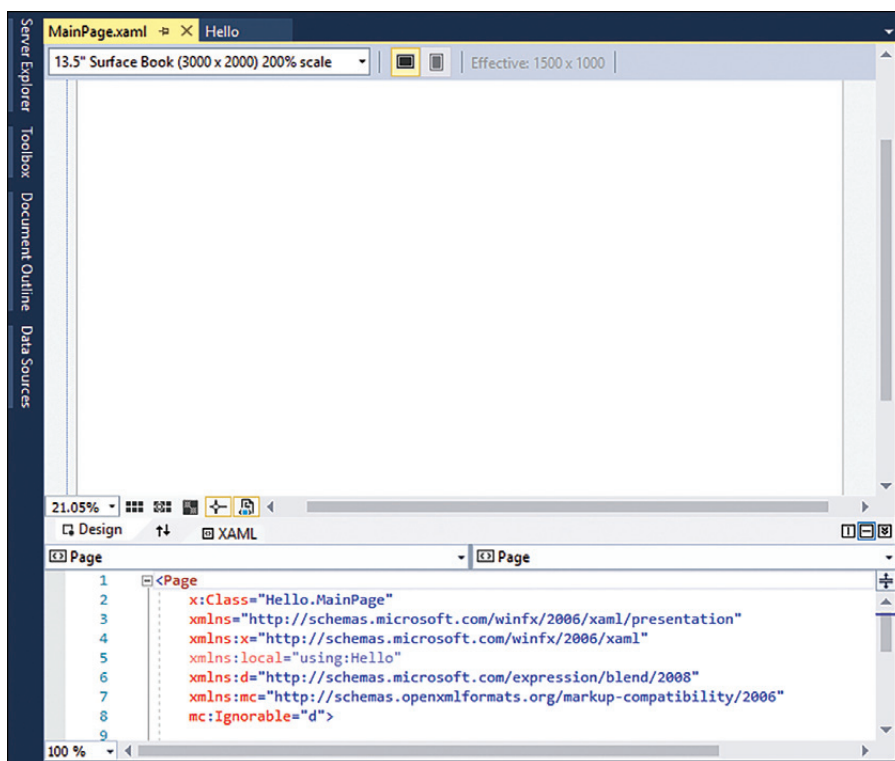
MainPage.xaml.cs. To właśnie w tym pliku zapisywany będzie nasz kod, który będzie wykonywany po wyświetleniu interfejsu użytkownika zdefiniowanego za pomocą pliku MainPage.xaml.

UWAGA Skrót XAML oznacza Extensible Application Markup Language (Rozszerzalny język znaczników aplikacji). Język XAML jest używany przez aplikacje Universal Windows Platform do definiowania wyglądu graficznego interfejsu użytkownika. Więcej informacji na temat języka XAML zawierać będą opisy zamieszczonych w tej książce ćwiczeń.



10. W Solution Explorer kliknij dwukrotnie plik MainPage.xaml.

Plik ten zawiera informacje o układzie interfejsu użytkownika. W oknie widoku projektowego wyświetlone zostaną dwie reprezentacje tego pliku:



W górnej części wyświetlany jest widok graficzny, domyślnie odzwierciedlający wygląd ekranu Surface Book, a w dolnej części znajduje się opis zawartości tego ekranu, zapisany w języku XAML. XAML jest podobny do języka XML i dla osób znających ten ostatni kod w języku XAML powinien wyglądać znajomo. W następnym ćwiczeniu posłużymy się oknem widoku projektowego do rozmieszczenia

elementów interfejsu użytkownika tworzonej aplikacji, a następnie zapoznamy się z wygenerowanym w ten sposób kodem w języku XAML.



Wskazówka Jeśli potrzebne będzie więcej miejsca do wyświetlania okna widoku projektowego, można zamknąć okna Output oraz Error List.



Uwaga Zanim przejdziemy dalej, warto wyjaśnić pewne kwestie związane z używaną terminologią. W tradycyjnych aplikacjach Windows interfejs użytkownika składa się z jednego lub z kilku *okien*, ale w aplikacjach Universal Windows Platform elementy odpowiadające oknom nazywane są *stronami*. Dla uproszczenia oba te elementy będę nazywać po prostu ogólnym terminem *formularz*. Słowo *okno* nadal jednak będzie używane, ilekroć mowa będzie o elementach środowiska IDE oprogramowania Visual Studio 2017, takich jak np. okno widoku projektowego.

W kolejnych ćwiczeniach posłużymy się oknem widoku projektowego, za pomocą którego dodamy do wyświetlanego przez aplikację formularza trzy nowe kontrolki oraz zapoznamy się z kodem implementującym funkcjonalność tych kontrolki w języku C#, wygenerowanym automatycznie przez program Visual Studio 2017.

➔ Tworzenie interfejsu użytkownika

1. Kliknij zakładkę Toolbox (Przybornik), znajdującą się w oknie widoku projektowego po lewej stronie formularza.

Spowoduje to wyświetlenie panelu narzędziowego Toolbox zawierającego różne komponenty oraz składniki, które można dodawać do tego formularza. Domyślnie wybrana jest sekcja General (Ogólne) przybornika, która (jeszcze) nie zawiera żadnych kontrolki.

2. Rozwiń sekcję o nazwie Common XAML Controls (Typowe kontrolki XAML).
W sekcji tej znajduje się lista kontrolki używanych przez większość aplikacji graficznych.



Wskazówka Bardziej obszerną listę kontrolki można znaleźć w sekcji All XAML Controls (Wszystkie kontrolki XAML).

3. Kliknij ikonę kontrolki *TextBlock* (Blok tekstu), znajdującą się w sekcji Common XAML Controls, a następnie przeciągnij tę kontrolkę do formularza wyświetlanego w oknie widoku projektowego.

WSKAZÓWKA Upewnij się, że zaznaczyłeś kontrolkę *TextBlock* (Blok tekstu), a nie *TextBox* (Pole tekstowe). Jeśli niechcący umieścisz na formularzu niewłaściwą kontrolkę, można ją łatwo usunąć, klikając ją i naciskając Delete.



Spowoduje to dodanie do formularza kontrolki *TextBlock* (za chwilę przeniesiemy ją we właściwe położenie) oraz ukrycie panelu Toolbox.

WSKAZÓWKA Jeśli chcesz, by panel Toolbox pozostał widoczny, lecz nie przesłaniał żadnej części formularza, kliknij przycisk Auto Hide (Autoukrywanie), znajdujący się po prawej stronie paska tytułowego panelu Toolbox. (Przycisk ten wygląda jak pinezka). Spowoduje to zakotwiczenie panelu Toolbox po lewej stronie okna programu Visual Studio 2017 oraz odpowiednie zmniejszenie rozmiarów okna widoku projektowego tak, by panel ten mógł zmieścić się na ekranie. (W przypadku korzystania z monitora o niskiej rozdzielczości może to oznaczać utratę bardzo dużego obszaru roboczego). Ponowne kliknięcie przycisku Auto Hide spowoduje ponowne ukrycie panelu Toolbox.



4. Umieszczona na formularzu kontrolka *TextBlock* prawdopodobnie nie znajduje się we właściwym miejscu. Położenie dodanych do formularza kontrolek możesz zmieniać poprzez ich kliknięcie i przeciągnięcie w żądane miejsce. Posługując się tą techniką przesuń kontrolkę *TextBlock* w okolice górnego, lewego narożnika formularza. (W tym konkretnym przypadku dokładne umiejscowienie tej kontrolki nie jest ważne). Można zauważyć, że przesunięcie kontrolki może wymagać uprzedniego kliknięcia w inne miejsce okna widoku projektowego, a następnie ponownego kliknięcia tej samej kontrolki.

Opis formularza w języku XAML, który jest wyświetlany w dolnym panelu, zawiera teraz opis kontrolki *TextBlock*, wraz z jej właściwościami, takimi jak np. położenie kontrolki w obrębie formularza, określane przez właściwość *Margin* (Margines), tekst wyświetlany domyślnie przez kontrolkę, zapisany we właściwości *Text*, sposób wyrównywania tekstu wyświetlanego przez kontrolkę, określane przez właściwości *HorizontalAlignment* (Wyrównanie w poziomie) i *VerticalAlignment* (Wyrównanie w pionie) oraz to, czy tekst wykraczający poza szerokość kontrolki powinien być zawijany czy nie.

Kod XAML dla kontrolki *TextBlock* będzie wyglądać podobnie jak w pokazanym poniżej przykładzie (konkretne wartości właściwości *Margin* mogą być nieco inne w zależności od miejsca, w którym umieszczona została na formularzu kontrolka *TextBlock*):

```
<TextBlock x:Name="textBlock" HorizontalAlignment="Left" Margin="10,10,0,0"
TextWrapping="Wrap" Text="TextBlock" VerticalAlignment="Top"/>
```

Panel kodu XAML oraz okno widoku projektowego są ze sobą nawzajem powiązane. Możliwe jest edytowanie wartości w panelu XAML, a wszelkie wykonane

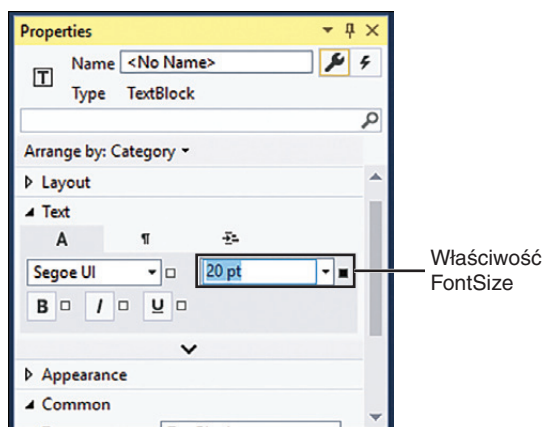
w tym panelu zmiany zostaną odzwierciedlone w oknie widoku projektowego. Możliwa jest na przykład zmiana położenia kontrolki *TextBlock* poprzez odpowiednią modyfikację wartości zapisanych we właściwości *Margin*.

5. Wybierz z menu View (Widok) polecenie Properties Window (Okno właściwości).

Polecenie to spowoduje wyświetlenie okna Properties (Właściwości) w dolnej prawej części ekranu, poniżej panelu Solution Explorer, o ile okno to nie było wyświetlane już wcześniej. Właściwości kontrolki można określać także za pomocą panelu XAML, znajdującego się poniżej okna z widokiem projektowym. Okno Properties (Właściwości) oferuje jednak wygodniejszy sposób modyfikowania właściwości umieszczonych na formularzu elementów, a także innych elementów projektu.

Zawartość okna Properties zależy od kontekstu, co oznacza, że wyświetlane są w nim właściwości aktualnie zaznaczonego elementu. Kliknięcie w oknie widoku projektowego dowolnego miejsca formularza poza kontrolką *TextBlock* spowoduje wyświetlenie w oknie Properties właściwości elementu typu *Grid* (Siatka). Jeśli sprawdzimy zawartość panelu XAML, to przekonamy się, że kontrolka *TextBlock* znajduje się wewnątrz elementu typu *Grid*. Wszystkie formularze zawierają element typu *Grid*, który zawiera rozkład wyświetlanych elementów; np. dodanie do elementu *Grid* nowych wierszy i kolumn pozwala na rozmieszczenie elementów formularza w sposób tabelaryczny.

6. Kliknij kontrolkę *TextBlock*, widoczną w oknie widoku projektowego. Spowoduje to ponowne wyświetlenie w oknie Properties właściwości kontrolki *TextBlock*.
7. Korzystając z okna Properties rozwiń gałąź właściwości *Text* (Tekst). Zmień wartość właściwości *FontSize* (Rozmiar czcionki) na 20 pt, a następnie wciśnij klawisz Enter. Właściwość ta znajduje się obok rozwijanej listy zawierającej nazwę czcionki, którą będzie Segoe UI:



UWAGA Przyrostek *pt* oznacza, że wielkość czcionki mierzona jest w *punktach*, gdzie 1 punkt to 1/72 cala.



8. Korzystając z panelu XAML, wyświetlanego poniżej okna widoku projektowego, przeanalizuj tekst definiujący kontrolkę *TextBlock*. Na końcu linii powinien znajdować się tekst `FontSize="26.667"`. Reprezentuje on przybliżoną konwersję rozmiaru czcionki z punktów do pikseli (3 punkty stanowią mniej więcej 4 piksele, choć dokładna konwersja zależy od rozmiaru i rozdzielczości ekranu). Wszelkie zmiany wykonywane za pomocą okna Properties są automatycznie odzwierciedlane w definicji zapisanej w języku XAML i na odwrót.

Zmień wartość atrybutu *FontSize*, nadpisując jej bieżącą wartość w panelu XAML wartością 24. Spowoduje to ponowną zmianę wielkości tekstu wyświetlanego przez kontrolkę *TextBlock* w oknie widoku projektowego, a także odpowiednią zmianę wartości wyświetlanej w oknie Properties.

9. Korzystając z okna Properties zapoznaj się z pozostałymi właściwościami kontrolki *TextBlock*. Wypróbuj, jakie efekty dają zmiany tych właściwości.

Zauważ, że zmiany wartości właściwości powodują dodawanie tych właściwości do definicji kontrolki *TextBlock* wyświetlanej w panelu XAML. Każda kontrolka dodana do formularza ma zestaw domyślnych wartości swoich właściwości i te domyślne wartości nie będą wyświetlane w panelu XAML, dopóki nie zostaną zmienione.

10. Zmień wartość właściwości *Text* dla kontrolki *TextBlock* z *TextBlock* na *Please enter your name* (Proszę podać swoje imię). Możesz to zrobić edytując element *Text* w panelu XAML lub zmieniając wartość właściwości w oknie Properties (ta właściwość znajduje się w sekcji Common [Wspólne] okna Properties).

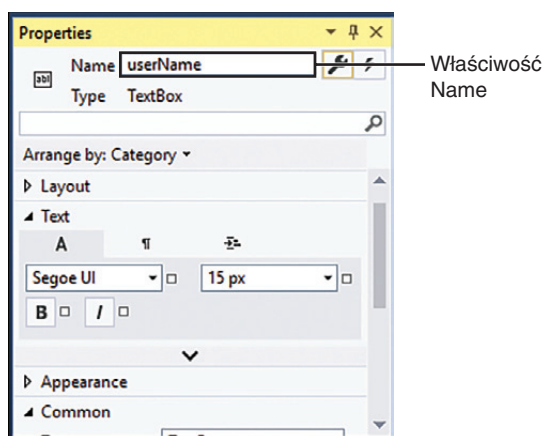
Należy zwrócić uwagę na fakt, że zmiana tej właściwości spowoduje także zmianę tekstu wyświetlanego przez kontrolkę *TextBlock* w oknie widoku projektowego.

11. Kliknij formularz znajdujący się w oknie widoku projektowego, a następnie wyświetl ponownie panel Toolbox.
12. Kliknij znajdującą się w panelu Toolbox kontrolkę *TextBox* i przeciągnij ją na formularz. Przesuń kontrolkę pola tekstowego tak, by znajdowała się bezpośrednio pod kontrolką *TextBlock*.

WSKAZÓWKA Jeśli podczas przeciągania kontrolki na formularzu jej chwilowe położenie staje się wyrównane w pionie lub w poziomie z innymi kontrolkami, następuje automatyczne wyświetlanie wskaźników wyrównywania. Wskaźniki te stanowią szybką, wizualną podpowiedź, ułatwiającą dokładne wyrównywanie względem siebie położenia różnych kontroltek.

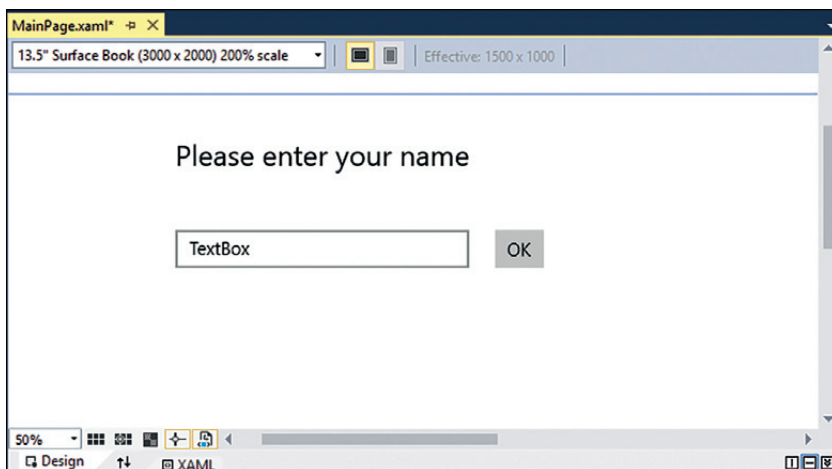


13. W oknie widoku projektowego umieść kursor myszy nad prawą krawędzią kontrolki *TextBox*. Kształt wskaźnika myszy powinien się wówczas zmienić na skierowaną w dwie strony strzałkę, oznaczającą możliwość zmiany rozmiaru kontrolki. Kliknij tę krawędź i przeciągnij ją w prawą stronę tak długo, aż wyrówna się ona z prawą krawędzią znajdującą się powyżej kontrolki *TextBlock*; w chwili, gdy obie krawędzie będą prawidłowo wyrównane, pojawią się wskaźniki wyrównywania.
14. Przy zaznaczonej kontrolce *TextBox* zmień wartość właściwości *Name*, wyświetlanej w górnej części okna *Properties*, z *textBox* na *userName*:



UWAGA Konwencje tworzenia nazw dla kontroltek i zmiennych zostaną omówione w rozdziale 2, „Zmienne, operatory i wyrażenia”.

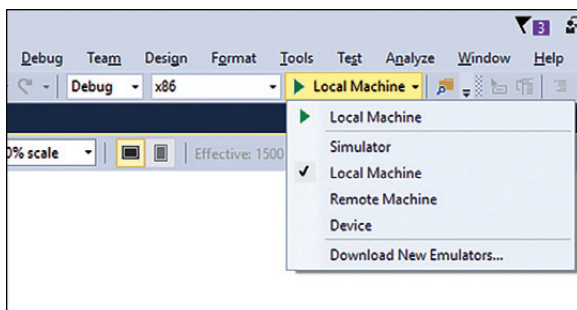
15. Wyświetl ponownie panel *Toolbox*, a następnie kliknij znajdującą się w tym panelu kontrolkę *Button* (Przycisk) i przeciągnij ją na formularz. Umieść kontrolkę przycisku po prawej stronie kontrolki pola tekstowego (*TextBox*) w taki sposób, by dolna krawędź przycisku była wyrównana w poziomie z dolną krawędzią pola tekstowego.
16. Korzystając z okna *Properties* zmień dla kontrolki *Button* wartość jej właściwości o nazwie *Name* na *ok*. Zmień również wartość właściwości *Content* (Zawartość) (właściwość ta znajduje się w sekcji *Common*) z *Button* na *OK*. Upewnij się, że działanie to spowodowało również zmianę tekstu wyświetlanego na znajdującej się na formularzu kontrolce przycisku.
Gotowy formularz powinien wyglądać podobnie to pokazanego na sąsiedniej stronie:



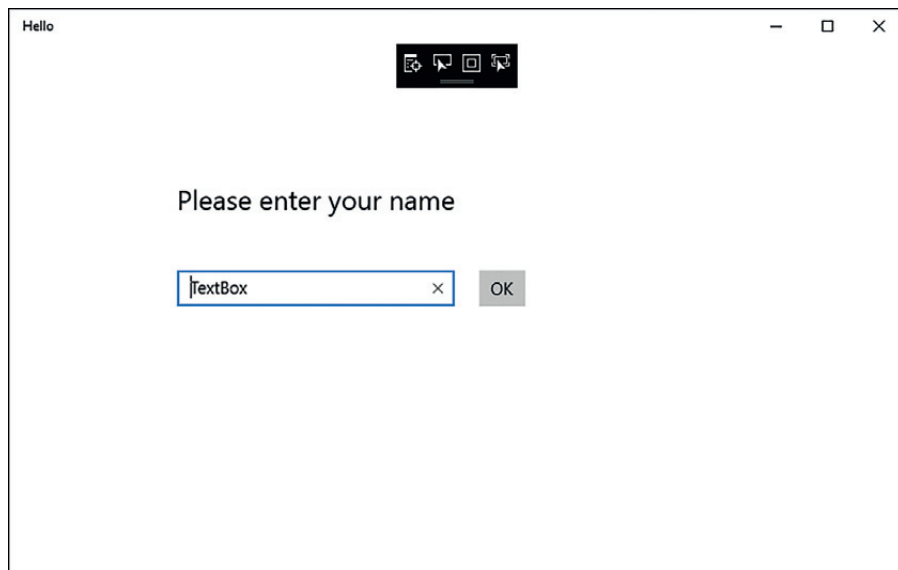
UWAGA Menu rozwijane w lewym górnym rogu okna Design View (Projektuj) umożliwia sprawdzanie, w jaki sposób formularz będzie wyglądał dla różnych rozmiarów i rozdzielczości ekranu. W tym przykładzie wybrany jest domyślny widok 13,5-calowego ekranu Surface Book o rozdzielczości 3000 x 2000. Po prawej stronie menu rozwijanego znajdują się dwa przyciski, które umożliwiają przełączanie się między orientacją pionową a poziomą. Prezentowane w kolejnych rozdziałach przykłady będą wykorzystywały widok projektowy 13,3-calowego ekranu Desktop, ale w tym przykładzie można pozostawić widok Surface Book.



17. Wybierz z menu Build polecenie Build Solution i upewnij się, czy kompilacja przebiegła bez błędów.
18. Sprawdź, czy na liście rozwijanej Debug Target wybrana jest opcja Local Machine (Maszyna lokalna), jak pokazano poniżej (pozostawienie domyślnej opcji Device mogłoby wiązać się z próbą nawiązania połączenia z telefonem Windows, co mogłoby skutkować niepowodzeniem kompilacji). Następnie z menu Debug wybierz opcję Start Debugging.



Powinno to spowodować uruchomienie aplikacji i wyświetlenie przez nią przygotowanego formularza. Formularz będzie wyglądał następująco:



UWAGA Uruchomienie aplikacji Universal Windows Platform w trybie debugowania powoduje wyświetlenie dwóch par liczb w lewym górnym rogu formularza. Liczby te podają częstotliwość odświeżania (wyświetlania klatek) i programiści mogą je wykorzystać do ustalenia, czy aplikacja nie reaguje wolniej niż powinna (co może sygnalizować problem z wydajnością). Wartości te są widoczne tylko w trybie debugowania. Pełne omówienie znaczenia poszczególnych liczb wykracza poza zakres tej książki, zatem obecnie możemy je zignorować.

W polu tekstowym możesz nadpisać istniejący tekst swoim imieniem, a następnie kliknąć przycisk OK, ale nic się jeszcze nie stanie. Konieczne będzie jeszcze dodanie kodu, który będzie określał, co ma się stać po kliknięciu przez użytkownika przycisku OK, co uczynimy w następnym kroku.

19. Powrót do programu Visual Studio 2017 i wybierz z menu Debug polecenie Stop Debugging (Zatrzymaj debugowanie).



UWAGA Można również kliknąć przycisk zamykania (symbol X znajdujący się w górnym prawym narożniku formularza), aby zamknąć formularz, zakończyć debugowanie i powrócić do programu Visual Studio.

Dotychczas zdołaliśmy utworzyć aplikację graficzną bez konieczności pisania nawet jednej linii kodu w języku C#. Na razie aplikacja ta nie robi jeszcze niczego szczególnego (aby to zmienić, będziemy musieli wpisać nieco kodu), ale w rzeczywistości program Visual Studio 2017 wygenerował dla nas całkiem sporo kodu zajmującego się obsługą rutynowych działań, które muszą być wykonywane przez każdą aplikację graficzną, takich jak uruchomienie się i wyświetlenie formularza. Zanim dodamy do aplikacji swój własny kod, dobrze będzie zaznajomić się z kodem wygenerowanym dla nas automatycznie przez program Visual Studio, który zostanie opisany w kolejnej części rozdziału.

Analiza aplikacji Universal Windows Platform

W panelu Solution Explorer rozwiń węzeł MainPage.xaml. Gdy pojawi się plik MainPage.xaml.cs, kliknij go dwukrotnie. W oknie edytora kodów i tekstu zostanie wówczas wyświetlony następujący kod formularza:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;
// Szablon elementu Blank Page jest udokumentowany pod adresem
// http://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409

namespace Hello
{
    /// <summary>
    /// Pusta strona, która może być używana samodzielnie
    /// lub do której można nawigować wewnątrz ramki.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

Oprócz sporej liczby dyrektyw *using*, powodujących włączenie do zasięgu przestrzeni nazw używanych przez większość aplikacji UWP, plik ten zawiera definicję klasy o nazwie *MainPage* i prawie nic poza tym. W pliku tym istnieje pewna niewielka ilość kodu dla klasy *MainPage*, będąca konstruktorem tej klasy i zawierająca wywołanie metody o nazwie *InitializeComponent*. Konstruktor to specjalna metoda o takiej samej nazwie jak klasa, której dotyczy. Metoda ta jest wykonywana podczas tworzenia nowego wystąpienia danej klasy i może zawierać kod służący do inicjowania nowej instancji klasy. Więcej informacji na temat konstruktorów podanych zostanie w rozdziale 7.

Klasa *MainPage* w rzeczywistości zawiera znacznie więcej kodu niż tych kilka linii widocznych w pliku *MainPage.xaml.cs*, ale większość tego kodu jest generowana automatycznie w oparciu o opis formularza w języku XAML i pozostaje ukryta. Ten ukryty kod wykonuje takie operacje, jak utworzenie i wyświetlenie formularza oraz utworzenie i umieszczenie na nim różnych kontrolek.



Wskazówka Wybranie z menu View polecenia Code, w czasie gdy aktywne jest okno widoku projektowego, pozwala także na wyświetlenie pliku z kodem w języku C# dla strony aplikacji UWP.

W tym miejscu część z Czytelników być może zaczyna się zastawiać, gdzie znajduje się metoda *Main* i w jaki sposób następuje wyświetlenie formularza po uruchomieniu aplikacji. Jak pamiętamy, w aplikacjach konsolowych to właśnie metoda *Main* definiuje punkt, od którego rozpoczyna się wykonywanie aplikacji. Aplikacje graficzne są nieco inne.

W panelu Solution Explorer powinien być widoczny jeszcze jeden plik źródłowy o nazwie *App.xaml*. Jeśli rozwiniemy węzeł tego pliku, to zobaczymy kolejny plik o nazwie *App.xaml.cs*. Plik *App.xaml* dostarcza punkt wejścia do uruchamianej aplikacji UWP. Jeśli klikniemy dwukrotnie plik *App.xaml.cs* w oknie panelu Solution Explorer, to spowoduje to wyświetlenie kodu, który powinien być podobny do tego zamieszczonego poniżej:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
```

```
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

namespace Hello
{
    /// <summary>
    /// Zapewnia zachowania specyficzna dla aplikacji, uzupełniające domyślną
    /// klasę Application.
    /// </summary>
    sealed partial class App : Application
    {
        /// <summary>
        /// Inicjuje pojedynczy obiekt aplikacji.
        /// Jest to pierwszy wiersz napisanego kodu wykonywanego i jest logicznym
        /// odpowiednikiem metod main() lub WinMain().
        /// </summary>
        public App()
        {
            this.InitializeComponent();
            this.Suspending += OnSuspending;
        }

        /// <summary>
        /// Wywoływane, gdy aplikacja jest uruchamiana normalnie
        /// przez użytkownika końcowego. Inne punkty wejścia
        /// będą używane, kiedy aplikacja zostanie uruchomiona
        /// w celu otworzenia określonego pliku.

        /// <param name="e">Szczegóły dotyczące żądania uruchomienia i procesu)
        /// </param>

        protected override void OnLaunched(LaunchActivatedEventArgs e)
        {
            Frame rootFrame = Window.Current.Content as Frame;

            // Nie powtarzaj inicjowania aplikacji, gdy w oknie znajduje się już
            // zawartość, upewnij się tylko, że okno jest aktywne.
            if (rootFrame == null)
            {
                // Utwórz ramkę, która będzie pełnić funkcję kontekstu nawigacji
                // i przejdź do pierwszej strony
                rootFrame = new Frame();

                rootFrame.NavigationFailed += OnNavigationFailed;

                if (e.PreviousExecutionState ==
                    ApplicationExecutionState.Terminated)
                {
                    //TODO: Załaduj stan z wstrzymanej wcześniej aplikacji
                }

                // Umieść ramkę w bieżącym oknie
                Window.Current.Content = rootFrame;
            }
        }
    }
}
```



```

        if (e.PrelaunchActivated == false)
        {
            if (rootFrame.Content == null)
            {
                // Kiedy stos nawigacji nie jest przywrócony, przejdź do pierwszej
                // strony, konfigurując nową stronę przez przekazanie
                // wymaganych informacji jako parametr
                rootFrame.Navigate(typeof(MainPage), e.Arguments);
            }
            // Upewnij się, że bieżące okno jest aktywne
            Window.Current.Activate();
        }
    }

    /// <summary>
    /// Wywoływane, gdy nawigacja do konkretnej strony nie powiedzie się
    /// </summary>
    /// <param name="sender">Ramka, do której nawigacja się nie powiodła</param>
    /// <param name="e">Szczegóły dotyczące niepowodzenia nawigacji</param>
    void OnNavigationFailed(object sender, NavigationFailedEventArgs e)
    {
        throw new Exception("Failed to load Page " + e.SourcePageType.FullName);
    }

    /// <summary>
    /// Wywoływane, gdy wykonanie aplikacji jest wstrzymywane.
    /// Stan aplikacji jest zapisywany bez wiedzy o tym, czy aplikacja zostanie
    /// zakończona, czy wznowiona z niezmienioną zawartością pamięci.
    /// </summary>
    /// <param name="sender">Źródło żądania wstrzymania.</param>
    /// <param name="e">Szczegóły żądania wstrzymania.</param>
    private void OnSuspending(object sender, SuspendingEventArgs e)
    {
        var deferral = e.SuspendingOperation.GetDeferral();
        //TODO: Zapisz stan aplikacji i zatrzymaj wszelkie aktywności w tle
        deferral.Complete();
    }
}

```

Większość pokazanego powyżej kodu to komentarze (linie rozpoczynające się od znaków `///`)* oraz inne instrukcje, których na razie nie musimy jeszcze rozumieć, ale kluczowe elementy znajdują się w ciele metody `OnLaunched` i zostały wyróżnione poprzez użycie pogrubionej czcionki. Metoda ta jest wykonywana po uruchomieniu aplikacji, a kod tej metody powoduje utworzenie przez aplikację nowego obiektu klasy *Frame* (Ramka), wyświetlenie w tej ramce formularza *MainPage*, a następnie jego aktywowanie. Na tym etapie nie ma potrzeby, by w pełni rozumieć sposób działania tego kodu

* W oryginalnym pliku występują komentarze tylko w języku angielskim. Zdecydowaliśmy się na przetłumaczenie ich w tym miejscu, gdyż zawierają wiele wartościowych informacji. Trzeba jednak pamiętać, że gdy Czytelnik wyświetli ten plik, znajdzie w nim oryginalne, angielskie komentarze (przyp. tłum.).

oraz składnię użytych w nim instrukcji. Wystarczy nam po prostu wiedzieć, że to właśnie w ten sposób następuje wyświetlenie formularza aplikacji po jej uruchomieniu.

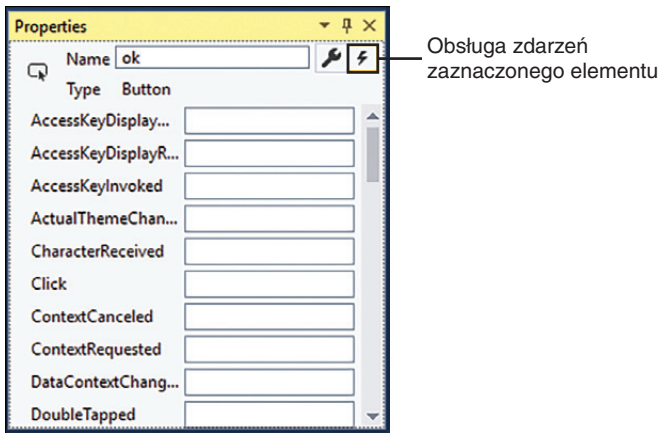
Dodawanie kodu do aplikacji graficznej

Teraz, gdy poznaliśmy już trochę strukturę aplikacji graficznej, nadszedł czas na napisanie kodu, który sprawi, że nasza aplikacja zacznie faktycznie coś robić.

➔ Pisanie kodu do obsługi przycisku OK

1. Otwórz w oknie widoku projektowego plik MainPage.xaml (poprzez dwukrotne kliknięcie pliku MainPage.xaml w oknie panelu Solution Explorer).
2. Kliknij przycisk OK znajdujący się na formularzu wyświetlanym w oknie widoku projektowego, aby go zaznaczyć.
3. W oknie Properties kliknij przycisk Event Handlers (Procedury obsługi zdarzeń) dla wybranego elementu.

Przycisk ten jest oznaczony ikoną wyglądającą jak mała błyskawica:



Okno Properties wyświetla listę z nazwami zdarzeń dla kontrolki typu *Button*. Zdarzenie oznacza istotną akcję, która zwykle wymaga jakiejś odpowiedzi i programista ma możliwość napisania własnego kodu, który zrealizuje tę odpowiedź.

4. W polu znajdującym się obok zdarzenia typu *Click* wpisz nazwę `okClick`, a następnie wciśnij klawisz Enter.

Spowoduje to wyświetlenie pliku w oknie edytora kodu i tekstu oraz dodanie nowej metody o nazwie `okClick` do klasy `MainPage`. Metoda ta będzie wyglądać następująco:

```
private void okClick(object sender, RoutedEventArgs e)
{
}
}
```

Nie należy przykładzać zbyt dużej uwagi do składni tego kodu, która na razie może być niezrozumiała – metody i ich budowa zostaną dokładnie omówione w rozdziale 3.

5. Dodaj do listy znajdującej się na początku pliku dyrektywę *using* wyróżnioną poniżej za pogrubioną czcionką (znak wielokropka oznacza, że część wierszy została pominięta dla zachowania zwięzłości wydruku):

```
using System;
...
using Windows.UI.Xaml.Navigation;
using Windows.UI.Popups;
```

6. Dodaj do metody *okClick* kod wyróżniony poniżej pogrubioną czcionką:

```
private void okClick(object sender, RoutedEventArgs e)
{
    MessageDialog msg = new MessageDialog("Hello " + userName.Text);
    msg.ShowAsync();
}
```

Kod ten będzie wykonywany po kliknięciu przez użytkownika przycisku OK. Również w tym przypadku nie należy na razie przejmować się składnią tego kodu. Należy jedynie zadbać o to, by pokazany kod został przepisany dokładnie tak, jak to zostało pokazane powyżej. Na razie wystarczy, jeśli będziemy wiedzieć, że pierwsza z tych instrukcji tworzy obiekt klasy *MessageDialog*, zawierający tekst komunikatu „Hello <TwojeImię>”, gdzie <TwojeImię> oznaczać będzie imię wpisane do znajdującej się na formularzu kontrolki typu *TextBox*. Druga z tych instrukcji wyświetla obiekt *MessageDialog*, powodując pojawienie się komunikatu na ekranie. Klasa *MessageDialog* zdefiniowana jest w przestrzeni nazw *Windows.UI.Popups* i dlatego konieczne było dodanie tej przestrzeni w kroku 5.

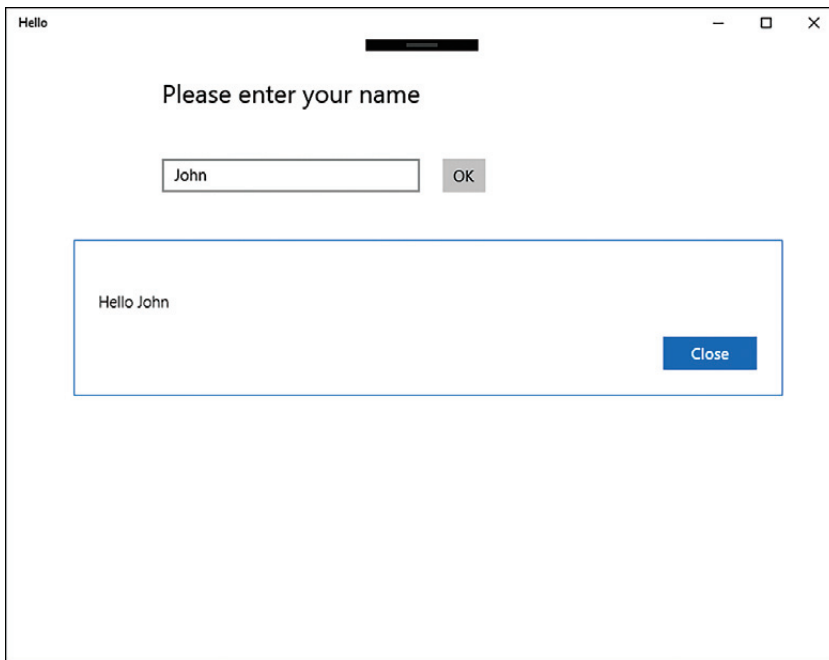


UWAGA Jak można zauważyć, Visual Studio 2017 dodaje zieloną falistą linię pod ostatnim wpisanym wierszem kodu. Jeśli umieścimy kursor nad tym wierszem kodu, program Visual Studio wyświetli ostrzeżenie “Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the ‘await’ operator to the result of the call” (Ponieważ to wywołanie nie jest oczekiwane, wykonywanie bieżącej metody będzie kontynuowane bez oczekiwania na ukończenie wywołania. Rozważ możliwość zastosowania operatora *await* do wyniku wywołania). Zasadniczo ostrzeżenie to sygnalizuje, że nie wykorzystujemy pełnych możliwości funkcji asynchroniczności oferowanych przez .NET Framework. Można zignorować to ostrzeżenie.

7. Kliknij zakładkę MainPage.xaml znajdującą się nad oknem edytora kodu i tekstu, aby ponownie wyświetlić formularz w oknie widoku projektowego.
8. Korzystając z dolnego panelu wyświetlającego opis formularza w języku XAML, zapoznaj się z elementem o nazwie *Button*, ale zachowaj ostrożność, by niczego nie zmienić w tym kodzie. Zwróć uwagę, że teraz element ten zawiera atrybut o nazwie *Click*, który wskazuje na metodę *okClick*.

```
<Button x:Name="ok" ... Click="okClick" />
```

9. Wybierz z menu Debug polecenie Start Debugging.
10. Po wyświetleniu formularza zastąp istniejący tekst znajdujący się w polu tekstowym, wpisując w jego miejsce swoje imię, a następnie wciśnij klawisz OK. Spowoduje to wyświetlenie komunikatu pozdrawiającego Cię po imieniu:



11. Kliknij przycisk Close (Zamknij) znajdujący się w oknie komunikatu.
12. Powrót do programu Visual Studio 2017 i wybierz z menu Debug polecenie Stop Debugging.

Inne typy aplikacji graficznych

Oprócz aplikacji Universal Windows Platform Visual Studio 2017 pozwala tworzyć także inne rodzaje aplikacji graficznych. Aplikacje te przeznaczone są dla specyficznych środowisk i nie zawierają zdolności adaptacyjnych, które pozwalałyby im działać na wielu platformach bez modyfikacji.

Te inne typy dostępnych aplikacji graficznych obejmują:

- **Aplikacje WPF** Szablon WPF App znajdziemy na liście Windows Classic Desktop w Visual Studio 2017. Akronim WPF oznacza „Windows Presentation Foundation”. WPF ukierunkowany jest na aplikacje, które działają na pulpicie Windows, a nie takie, które dostosowują się do szerokiego zakresu różnych urządzeń i proporcji ekranu. Zapewnia niezwykle wydajne środowisko bazujące na grafice wektorowej, która pozwala na gładkie skalowanie interfejsu użytkownika pomiędzy różnymi rozdzielczościami ekranu. Wiele kluczowych funkcjonalności WPF dostępnych jest w aplikacjach UWP, choć WPF oferuje dodatkowe funkcje, które są właściwe tylko dla aplikacji uruchamianych na wydajnych komputerach biurowych.
- **Aplikacje Windows Forms** Jest to starsza biblioteka graficzna sięgająca czasów początków platformy .NET Framework. Ten szablon znajdziemy na liście Class Desktop w Visual Studio 2017. Zgodnie z nazwą, biblioteka Windows Forms przeznaczona jest do projektowania bardziej klasycznych aplikacji opartych na formularzach, wykorzystujących biblioteki Graphics Device Interface (GDI) dostarczanych wraz z systemem Windows w tamtym czasie. Choć platforma ta jest łatwa w użyciu, nie zapewnia ani funkcjonalności i skalowalności WPF, ani przenośności UWP.

Przy projektowaniu aplikacji graficznych zalecane jest wybranie szablonu UWP, chyba że istnieją uzasadnione powody, aby tego nie robić.

Podsumowanie

W tym rozdziale pokazaliśmy, jak użyć programu Visual Studio 2017 do utworzenia, skompilowania i uruchamiania aplikacji. Utworzyliśmy prostą aplikację konsolową, która wyświetlała wyniki w oknie konsoli oraz aplikację Universal Windows Platform z prostym graficznym interfejsem użytkownika.

- Jeśli zamierzasz przejść do kolejnego rozdziału, pozostaw uruchomiony program Visual Studio 2017 i przejdź do rozdziału 2.
- Jeśli chcesz na tym zakończyć pracę z programem Visual Studio 2017, wybierz z menu File polecenie Exit (Zakończ). Jeśli spowoduje to wyświetlenie okna dialogowego z pytaniem o zapisanie otwartego projektu, należy zapisać ten projekt, klikając przycisk Yes (Tak).

Krótki przegląd rozdziału 1

W celu	Wykonaj następujące czynności
Utworzenia nowej aplikacji konsolowej przy użyciu programu Visual Studio 2017	W menu File wskaż menu podrzędne New, a następnie kliknij polecenie Project, co spowoduje otwarcie okna dialogowego New Project. W lewym panelu rozwiń Installed, rozwiń Templates, a następnie kliknij element Visual C#. W środkowym panelu kliknij ikonę Console Application. Określ w polu Location katalog, w którym należy umieścić pliki projektu. Wpisz nazwę projektu i kliknij przycisk OK.
Utworzenia nowej aplikacji UWP przy użyciu programu Visual Studio 2017	W menu File wskaż menu podrzędne New, a następnie kliknij polecenie Project, co spowoduje otwarcie okna dialogowego New Project. W lewym panelu rozwiń Installed, rozwiń Templates, rozwiń Visual C#, rozwiń Windows, a następnie kliknij element Universal (Aplikacja uniwersalna). W środkowym panelu kliknij ikonę Blank App (Windows Universal) (Pusta aplikacja). Określ w polu Location katalog, w którym należy umieścić pliki projektu. Wpisz nazwę projektu i kliknij przycisk OK.
Skompilowania aplikacji	Wybierz z menu Build polecenie Build Solution.
Uruchomienia aplikacji w trybie debugowania	Wybierz z menu Debug polecenie Start Debugging.
Uruchomienia aplikacji bez debugowania	Wybierz z menu Debug polecenie Start Without Debugging.

ROZDZIAŁ 2

Zmienne, operatory i wyrażenia

Po ukończeniu tego rozdziału Czytelnik będzie potrafił:

- Wyjaśnić, co to są instrukcje, identyfikatory i słowa kluczowe.
- Używać zmiennych do przechowywania informacji.
- Posługiwać się prostymi typami danych.
- Używać operatorów arytmetycznych, takich jak znak plus (+) i minus (-).
- Dokonywać inkrementacji i dekrementacji zmiennych.

W rozdziale 1, „Wprowadzenie do języka C#” pokazaliśmy, jak przy użyciu środowiska programowania Microsoft Visual Studio 2017 można zbudować/skompilować i uruchomić program konsolowy oraz aplikację graficzną. W tym rozdziale przedstawione zostaną elementy składni i semantyka języka Microsoft Visual C#, obejmująca instrukcje, słowa kluczowe i identyfikatory. Omówione zostaną podstawowe typy danych wbudowane w język C# (tzw. typy prymitywne) a także charakterystyki wartości, które mogą być przechowywane za pomocą każdego z tych typów danych. Pokażemy także, w jaki sposób można deklarować i używać zmiennych lokalnych (zmiennych istniejących tylko wewnątrz określonej metody lub innej, niewielkiej sekcji kodu), zapoznamy się z oferowanymi przez język C# operatorami arytmetycznymi, dowiemy się, w jaki sposób używać tych operatorów do manipulowania wartościami, a także poznamy sposób kontrolowania wyrażeń zawierających dwa lub więcej operatorów.

Instrukcje

Instrukcja to polecenie wykonujące określone działanie, takie jak np. obliczenie pewnej wartości i zapisanie rezultatu lub wyświetlenie komunikatu dla użytkownika. Instrukcje można łączyć ze sobą, tworząc metody. Więcej informacji na temat metod zostanie podanych w rozdziale 3, „Tworzenie metod i stosowanie zakresów zmiennych”, ale na razie możemy traktować metody jako nazwane sekwencje instrukcji. Przykładem metody może być metoda *Main*, z którą zetknęliśmy się już w poprzednim rozdziale.

Instrukcje języka C# podlegają dobrze określonej zbiorowi reguł, definiujących ich format oraz konstrukcję. Reguły te noszą zbiorczą nazwę *składni* (dla porównania, specyfikacja opisująca, *co robią* poszczególne instrukcje, określana jest mianem *semantyki*). Jedną z najprostszych, a zarazem najważniejszych reguł składni języka C# jest reguła nakazująca kończenie wszystkich instrukcji znakiem średnika. Przykładowo, bez umieszczonego na końcu średnika pokazana poniżej instrukcja nie zostanie skompilowana:

```
Console.WriteLine(„Hello, World!");
```



Wskazówka Język C# należy do języków o tzw. „swobodnym formatowaniu”, co oznacza, że wszelkie tzw. białe znaki, takie jak spacja lub znak nowej linii, traktowane są jako separator i nie mają żadnego innego znaczenia. Inaczej mówiąc, programista ma pełną dowolność w zakresie stylu zapisywania instrukcji. Należy jednak przyjąć jakiś prosty styl i konsekwentnie stosować go podczas pisania własnych programów, co znacznie ułatwi ich odczytywanie i zrozumienie działania.

Kluczem do dobrego programowania w dowolnym języku jest poznanie jego składni i semantyki, a następnie opanowanie sztuki posługiwania się tym językiem w naturalny i idiomatyczny sposób. Stosowanie takiego podejścia ułatwi utrzymywanie i pielęgnację kodu tworzonych programów. W kolejnych rozdziałach tej książki prezentowane będą przykłady używania najważniejszych instrukcji języka C#.

Identyfikatory

Identyfikatory to nazwy używane do identyfikowania różnych elementów programu, takich jak przestrzeń nazw, klasy, metody lub zmienne (więcej na temat zmiennych dowiemy się wkrótce). W języku C# identyfikatory muszą spełniać następujące reguły:

- Identyfikatory mogą składać się wyłącznie z liter (małych i dużych), cyfr oraz znaków podkreślenia.
- Identyfikator musi rozpoczynać się od litery lub od znaku podkreślenia.

Przykładowo, *result*, *_score*, *footballTeam* i *plan9* to przykłady poprawnych identyfikatorów, natomiast nie są nimi *result%*, *footballTeam\$* i *9plan*.



WAŻNE W języku C# są rozróżniane małe i duże litery: np. *footballTeam* i *FootballTeam* nie są tymi samymi identyfikatorami.

Słowa kluczowe

Język C# rezerwuje na swoje własne potrzeby 77 identyfikatorów, które nie mogą być używane przez programistów do swoich własnych celów. Te zarezerwowane identyfikatory nazywane są *słowa kluczowymi* i każdy z nich ma określone znaczenie. Przykłady słów kluczowych to *class*, *namespace* oraz *using*. W trakcie lektury tej książki zapoznamy się ze znaczeniem większości słów kluczowych języka C#. Poniżej przedstawiona została pełna lista słów kluczowych języka C#:

<i>abstract</i>	<i>do</i>	<i>in</i>	<i>protected</i>	<i>true</i>
<i>as</i>	<i>double</i>	<i>int</i>	<i>public</i>	<i>try</i>
<i>base</i>	<i>else</i>	<i>interface</i>	<i>readonly</i>	<i>typeof</i>
<i>bool</i>	<i>enum</i>	<i>internal</i>	<i>ref</i>	<i>uint</i>
<i>break</i>	<i>event</i>	<i>is</i>	<i>return</i>	<i>ulong</i>
<i>byte</i>	<i>explicit</i>	<i>lock</i>	<i>sbyte</i>	<i>unchecked</i>
<i>case</i>	<i>extern</i>	<i>long</i>	<i>sealed</i>	<i>unsafe</i>
<i>catch</i>	<i>false</i>	<i>namespace</i>	<i>short</i>	<i>ushort</i>
<i>char</i>	<i>finally</i>	<i>new</i>	<i>sizeof</i>	<i>using</i>
<i>checked</i>	<i>fixed</i>	<i>null</i>	<i>stackalloc</i>	<i>virtual</i>
<i>class</i>	<i>float</i>	<i>object</i>	<i>static</i>	<i>void</i>
<i>const</i>	<i>for</i>	<i>operator</i>	<i>string</i>	<i>volatile</i>
<i>continue</i>	<i>foreach</i>	<i>out</i>	<i>struct</i>	<i>while</i>
<i>decimal</i>	<i>goto</i>	<i>override</i>	<i>switch</i>	
<i>default</i>	<i>if</i>	<i>params</i>	<i>this</i>	
<i>delegate</i>	<i>implicit</i>	<i>private</i>	<i>throw</i>	

Wskazówka W oknie edytora kodu i tekstu Visual Studio 2017 słowa kluczowe wyróżniane są kolorem niebieskim.



Język C# wykorzystuje także wymienione poniżej identyfikatory. Nie są one zarezerwowane przez język C#, co oznacza, że można ich używać jako nazw swoich własnych metod, zmiennych lub klas, ale jeśli to tylko możliwe, należy tego unikać.

<i>add</i>	<i>get</i>	<i>remove</i>	<i>alias</i>	<i>global</i>
<i>select</i>	<i>ascending</i>	<i>group</i>	<i>set</i>	<i>async</i>
<i>into</i>	<i>value</i>	<i>await</i>	<i>join</i>	<i>var</i>
<i>descending</i>	<i>let</i>	<i>where</i>	<i>dynamic</i>	<i>orderby</i>
<i>yield</i>	<i>from</i>	<i>partial</i>		

Zmienne

Zmienna to miejsce służące do przechowywania wartości. Zmienne można postrzegać jako wycinek pamięci komputera, służący do przechowywania tymczasowych informacji. Każda istniejąca w programie zmienna musi posiadać własną, unikalną nazwę, jednoznacznie identyfikującą tę zmienną w kontekście, w którym została użyta. Nazwa zmiennej służy do odwoływania się do wartości przechowywanej przez tę zmienną. Przykładowo, jeśli zechcemy zapamiętać wartość pewnego elementu znajdującego się w magazynie, możemy utworzyć zmienną o nazwie *koszt* i zapisać w niej koszt tego elementu. Jeśli później odwołamy się do zmiennej *koszt*, to odczytaną wartością tej zmiennej będzie zapisany w niej wcześniej koszt elementu.

Nazywanie zmiennych

Dobrze jest przyjąć pewną konwencję nazewnictwa, która eliminować będzie niejasności związane z definiowanymi w programie zmiennymi. Jest to szczególnie ważne w przypadku osób pracujących w większych zespołach, w których kilku programistów pracuje nad różnymi częściami aplikacji. Zachowanie spójnej konwencji nazewnictwa pomaga wówczas w unikaniu pomyłek i ograniczenie zakresu ewentualnych błędów. Poniższa lista zawiera pewne ogólne rekomendacje w tym zakresie:

- Nie należy rozpoczynać identyfikatorów od znaku podkreślenia. Wprawdzie jest to dozwolone w języku C#, ale może ograniczać możliwości współdziałania tworzonego kodu z aplikacjami tworzonymi przy użyciu innych języków programowania, takich jak np. Microsoft Visual Basic.
- Nie należy tworzyć identyfikatorów różniących się jedynie wielkością liter. Przykładowo, nie należy tworzyć jednej zmiennej o nazwie *mojaZmienna* i drugiej, o nazwie *MojaZmienna*, jeśli zmienne te miałyby być używane jednocześnie, ponieważ można je łatwo ze sobą pomylić. Ponadto, stosowanie identyfikatorów różniących się jedynie wielkością liter może ograniczać możliwości używania we własnych aplikacjach klas utworzonych w innych językach programowania, w których małe i duże litery nie są rozróżniane, takich np. jak Microsoft Visual Basic.
- Nazwy powinny rozpoczynać się od małej litery.
- W przypadku identyfikatorów będących złożeniem kilku słów, drugie i każde kolejne słowo powinno rozpoczynać się z dużej litery (taka notacja nazywana jest notacją wielbłądzą – ang. *camelCase*).
- Nie należy używać notacji węgierskiej (notacja ta jest zapewne dobrze znana programistom używającym języka C lub C++; jeśli jednak ktoś nie zna notacji węgierskiej, nie ma się czym przejmować).

Przykładowo, *score*, *footballTeam*, *_score* i *FootballTeam* są poprawnymi nazwami zmiennych, ale tylko dwie pierwsze są zgodne z podanymi zaleceniami.

Deklarowanie zmiennych

Zmienne służą do przechowywania wartości. W języku C# istnieje wiele różnych typów wartości, które można przechowywać i przetwarzać – liczby całkowite, liczby zmiennoprzecinkowe lub łańcuchy znakowe, by wymienić tylko trzy z nich. Deklarując zmienną, trzeba określić typ danych, jaki przechowywać będzie ta zmienna.

Typ i nazwę zmiennej deklaruje się w instrukcji deklaracji. Przykładowo, pokazana poniżej instrukcja deklaruje zmienną o nazwie *wiek*, która przechowywać będzie wartości typu *int* (liczby całkowite). Jak zawsze, taka instrukcja musi być zakończona znakiem średnika.

```
int wiek;
```

Typ zmiennej *int* to nazwa jednego z podstawowych (*primitive*) typów danych języka C#, oznaczająca *liczby całkowite*. (W dalszej części tego rozdziału poznamy więcej podstawowych typów danych).

UWAGA Programiści używający języka Microsoft Visual Basic powinni pamiętać, że język C# nie dopuszcza niejawnych deklaracji zmiennych. Wszystkie zmienne muszą zostać jawnie zadeklarowane, zanim będą mogły zostać użyte.



Po zadeklarowaniu zmiennej możliwe jest przypisanie jej pewnej wartości. Pokazana poniżej instrukcja przypisuje zmiennej *wiek* wartość 42. Ponownie przypominam o konieczności zakończenia instrukcji znakiem średnika.

```
wiek = 42;
```

Występujący w tej instrukcji znak równości (=) to operator *przypisania*, który przypisuje zmiennej wartość znajdującą się z prawej strony tego operatora. Po przypisaniu wartości, zmiennej *wiek* można używać w kodzie do odwoływania się do przechowywanej w niej wartości. Kolejna instrukcja demonstrowuje sposób wypisania w oknie konsoli wartości zmiennej *wiek*, czyli liczby 42:

```
Console.WriteLine(wiek);
```

Wskazówka Zatrzymanie wskaźnika myszy nad nazwą zmiennej w oknie edytora kodu i tekstu programu Visual Studio 2017 powoduje wyświetlenie podpowiedzi informującej o typie danej zmiennej.



Specyfikowanie wartości numerycznych

Ważne jest dobre zrozumienie wpływu typu zmiennej na to, jakiego rodzaju dane może przechować ta zmienna i jak te dane będą obsługiwane. Przykładowo powinno być oczywiste, że zmienna numeryczna nie może przechować wartości tekstowej (łańcucha), takiej jak „Hello”. Jednak w niektórych przypadkach typ wartości przypisywanej do zmiennej nie zawsze jest tak oczywisty.

Weźmy dla przykładu literalną wartość 42. Jest numeryczna. Co więcej, jest to liczba całkowita i można ją bezpośrednio przypisać zmiennej typu całkowitego (*int*). Ale co się stanie, jeśli spróbujemy przypisać tę wartość zmiennej typu niecałkowitego, na przykład zmiennoprzecinkowego (*float*)? Odpowiedź brzmi: C# w tle przekonwertuje wartość całkowitą na zmiennoprzecinkową. Jest to stosunkowo mało szkodliwa, ale niekoniecznie dobra praktyka. Powinniśmy naprawdę określić, że zamierzamy traktować literalną wartość 42 jako liczbę zmiennoprzecinkową i nie przypisywać jej przez pomyłkę niewłaściwego typu. Można to osiągnąć, dodając sufix „F” do literału numerycznego, jak poniżej:

```
float myVar; // deklaracja zmiennej zmiennoprzecinkowej
myVar = 42F; // przypisanie wartości zmiennoprzecinkowej
```

A co się stanie z taką liczbą, jak 0.42? Jaki jest typ tego wyrażenia? Odpowiedź brzmi, że tak jak wszystkie inne literały numeryczne zawierające znak dziesiętny, jest to liczba zmiennoprzecinkowa o podwójnej precyzji (*double-precision*), określana skrótowo terminem *double*. W kolejnym podpunkcie zobaczymy, że typ *double* ma większy zakres wartości i większą dokładność, niż *float*. Jeśli chcemy przypisać wartość 0.42 zmiennej typu *float*, należy dołączyć sufix „F” (w istocie kompilator C# będzie tego wymagał):

```
myVar = 0.42F;
```

C# zawiera wiele innych typów numerycznych; mamy tu „długie” liczby całkowite (*long*), oferujące większy zakres wartości niż zwykłe liczby całkowite, oraz liczby dziesiętne (*decimal*) przechowujące zadaną liczbę miejsc dziesiętnych (typy zmiennoprzecinkowe przechowują określoną liczbę *cyfr znaczących* i zależnie od wartości bezwzględnej mogą zawierać kilka lub więcej miejsc dziesiętnych, ale mogą też nie zawierać ich wcale). Dołączenie sufiksu „L” do literału numerycznego wymusza użycie typu *long*, zaś sufix „M” powoduje traktowanie podanej liczby jako dziesiętnej.

Wszystkie te rozważania mogą wydawać się oczywiste, żeby nie powiedzieć trywialne, ale zadziwiające jest, jak wiele subtelnych błędów może wkraść się do programu poprzez przypadkowe przypisanie zmiennej wartości niewłaściwego typu. Zastanówmy się na przykład, co mogłoby się wydarzyć, gdybyśmy próbowali wykonać obliczenia wymagające dużej liczby znaczących miejsc dziesiętnych (po przecinku) i zapisalibyśmy wynik w zmiennej typu *float*. W najgorszym razie doprowadzi to do obciążenia danych, co wywoła błędy w obliczeniach i nasza aplikacja sterująca sondą kosmiczną nie trafi w Marsa i zamiast tego uda się w przestrzeń międzygwiazdną!

Podstawowe typy danych

Język C# oferuje wiele wbudowanych typów danych, które nazywane są *podstawowymi typami danych* (lub typami prymitywnymi). Poniższa tabela zawiera zestawienie najczęściej stosowanych, podstawowych typów danych języka C#, wraz z informacją o zakresie wartości, jakie mogą być przechowywane za pomocą każdego z tych typów.

Typ danych	Opis	Rozmiar (w bitach)	Zakres wartości	Przykładowe zastosowanie
<i>int</i>	Liczby całkowite	32	-2^{31} do $2^{31} - 1$	<code>int count;</code> <code>count = 42;</code>
<i>long</i>	Liczby całkowite (o większym zakresie)	64	-2^{63} do $2^{63} - 1$	<code>long wait;</code> <code>wait = 42L;</code>
<i>float</i>	Liczby zmiennoprzecinkowe	32	$\pm 3.4 \times 10^{-38}$ do $\pm 3.4 \times 10^{38}$	<code>float away;</code> <code>away = 0.42F;</code>
<i>double</i>	Liczby zmiennoprzecinkowe o podwójnej precyzji (bardziej dokładne)	64	$\pm 5.0 \times 10^{-324}$ do $\pm 1.7 \times 10^{308}$	<code>double trouble;</code> <code>trouble = 0.42;</code>
<i>decimal</i>	Wartości finansowe	128	28 cyfr znaczących	<code>decimal coin;</code> <code>coin = 0.42M;</code>
<i>string</i>	Sekwencja znaków	16 bitów na każdy znak	Nie dotyczy	<code>string vest</code> <code>vest = "forty two";</code>
<i>char</i>	Pojedynczy znak	16	0 do $2^{16} - 1$	<code>char grill;</code> <code>grill = 'x';</code>
<i>bool</i>	Wartość logiczna	8	true lub false	<code>bool teeth;</code> <code>teeth = false;</code>

Zmienne lokalne bez przypisanej wartości

Po zadeklarowaniu zmiennej, dopóki nie zostanie jej przypisana jakaś wartość, zmienna zawiera wartość przypadkową. Takie zachowanie było źródłem wielu błędów w programach napisanych w języku C i C++, gdy utworzona zmienna została przypadkowo użyta jako źródło informacji, zanim jeszcze została jej przypisana jakakolwiek wartość. Język C# nie pozwala na używanie zmiennych bez przypisania im wartości. Każdej zmiennej, zanim będzie ona mogła być użyta, musi zostać przypisana wartość – w przeciwnym razie program nie zostanie skompilowany. Wymóg ten nosi nazwę *reguły przypisania określonej wartości* (ang. *definite assignment rule*). Przykładowo, ponieważ w pokazanym poniżej przykładzie zmiennej *wiek* nie przypisano żadnej wartości, to podczas próby jego skompilowania otrzymamy następujący komunikat błędu

„Use of unassigned local variable 'wiek' ” (Użycie zmiennej 'wiek' bez przypisania jej wartości):

```
int wiek;  
Console.WriteLine(wiek); // błąd kompilacji
```

Wyświetlanie wartości podstawowych typów danych

W poniższym ćwiczeniu zademonstrujemy sposób postępowania z kilkoma podstawowymi typami danych, korzystając w tym celu z programu w języku C# o nazwie *PrimitiveDataTypes*.

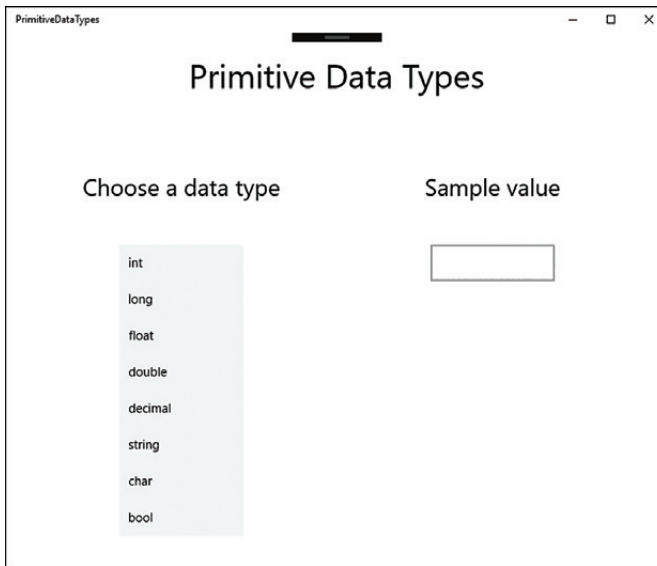
➔ Wyświetlanie wartości podstawowych typów danych

1. Jeśli program Visual Studio 2017 nie został uruchomiony już wcześniej, to uruchom go teraz.
2. Wskaż w menu File menu podrzędne Open, a następnie wybierz z niego polecenie Project/Solution (Projekt/Rozwiązanie).
Spowoduje to otwarcie okna dialogowego Open Project (Otwórz projekt).
3. Przejdź do katalogu \Microsoft Press\VCSBS\Chapter 2\PrimitiveDataTypes w folderze Dokumenty.
4. Zaznacz plik rozwiązania *PrimitiveDataTypes* i naciśnij przycisk Open.
Nastąpi wówczas załadowanie rozwiązania, a w panelu Solution Explorer wyświetlony zostanie projekt *PrimitiveDataTypes*.



UWAGA Pliki rozwiązań mają rozszerzenie .sln, np. *PrimitiveDataTypes.sln*. Rozwiązanie może zawierać jeden lub więcej projektów. Pliki projektów Visual C# mają rozszerzenia .csproj. Jeśli zamiast pliku rozwiązania otwarty zostanie plik projektu, to program Visual Studio 2017 automatycznie utworzy dla niego nowy plik rozwiązania. Taka sytuacja może być dezorientująca dla osób nieświadomych istnienia tej funkcjonalności, ponieważ może ona prowadzić do niezamierzonego wygenerowania wielu rozwiązań dla tego samego projektu.

5. Wybierz z menu Debug polecenie Start Debugging (Rozpocznij debugowanie).
W tym miejscu program Visual Studio może wyświetlić kilka ostrzeżeń, które na razie można jednak zignorować. (Ich przyczyny zostaną poprawione w następnym ćwiczeniu).



6. Wybierz z listy Choose a Data Type (Wybierz typ danych) typ danych *string*.
W polu Sample value (Przykładowa wartość) zostanie wówczas wyświetlony tekst „forty two” (czterdzieści dwa).
7. Wybierz z listy typ danych *int*.
W polu Sample value zostanie wówczas wyświetlony tekst "to do" (do zrobienia) oznaczający, że instrukcja wyświetlająca wartość typu *int* musi dopiero zostać napisana.
8. Wybierz po kolei każdy, dostępny na liście typ danych. Upewnij się, że kod wyświetlający wartości typu *double* oraz *bool* również nie został jeszcze zaimplementowany.
9. Powróć do programu Visual Studio 2017 i wybierz z menu Debug polecenie Stop Debugging (Zatrzymaj debugowanie).
Możesz również zatrzymać debugowanie, zamykając okno.

➔ Używanie podstawowych typów danych w kodzie źródłowym

1. Korzystając z okna Solution Explorer (Eksplorator rozwiązań), rozwiń węzeł projektu *PrimitiveDataTypes* (o ile nie został on rozwinięty już wcześniej), a następnie kliknij dwukrotnie plik *MainPage.xaml*.
Spowoduje to wyświetlenie formularza aplikacji w oknie widoku projektowego.



Wskazówka Jeśli ekran posiadanego urządzenia nie jest wystarczająco duży, by wyświetlić cały formularz, możliwe jest powiększanie i pomniejszanie skali używanej w oknie widoku projektowego za pomocą kombinacji klawiszy Ctrl+Alt+= i Ctrl+Alt+-, albo poprzez wybranieżądanego rozmiaru z rozwijanej listy skali powiększenia, znajdującej się w dolnym lewym rogu okna widoku projektowego.

- Przewiń w dół zawartość panelu XAML i odszukaj w nim znacznik dla kontrolki *ListBox*. Kontrolka ta wyświetla po lewej stronie formularza listę typów danych, a jej opis w języku XAML wygląda następująco (z poniższego wydruku usunięto niektóre z właściwości):

```
<ListBox x:Name="type" ... SelectionChanged="typeSelectionChanged">
  <ListBoxItem>int</ListBoxItem>
  <ListBoxItem>long</ListBoxItem>
  <ListBoxItem>float</ListBoxItem>
  <ListBoxItem>double</ListBoxItem>
  <ListBoxItem>decimal</ListBoxItem>
  <ListBoxItem>string</ListBoxItem>
  <ListBoxItem>char</ListBoxItem>
  <ListBoxItem>bool</ListBoxItem>
</ListBox>
```

Kontrolka *ListBox* wyświetla każdy typ danych jako osobny element typu *ListBoxItem*. Po uruchomieniu aplikacji kliknięcie przez użytkownika wybranego elementu z tej listy spowoduje wygenerowanie zdarzenia typu *SelectionChanged* (zdarzenie to jest trochę podobne do zdarzenia typu *Click*, z którym zetknęliśmy się już w rozdziale 1). Jak widać, wystąpienie tego zdarzenia powoduje wywołanie przez kontrolkę *ListBox* metody *typeSelectionChanged*. Metoda ta zdefiniowana jest w pliku *MainPage.xaml.cs*.

- Z menu View (Widok) wybierz polecenie Code (Kod).

Spowoduje to otwarcie okna edytora kodu i tekstu i wyświetlenie w nim pliku *MainPage.xaml.cs*.

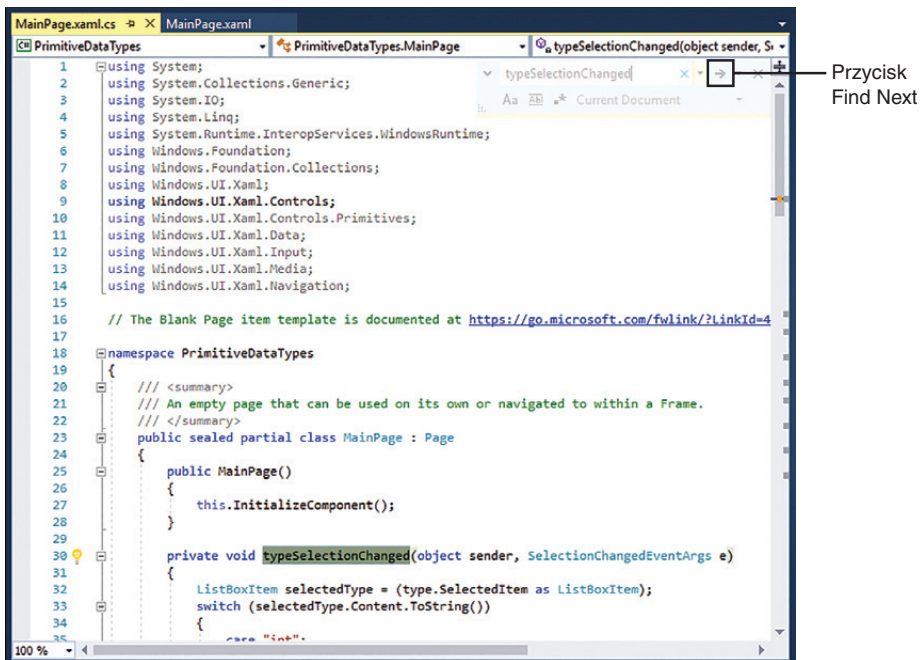


Uwaga Należy pamiętać, że do wyświetlenia kodu źródłowego można wykorzystać także Eksplorator rozwiązań. W tym celu należy rozwinąć węzeł pliku *MainPage.xaml* poprzez kliknięcie symbolu strzałki znajdującego się po lewej stronie tego pliku, a następnie kliknąć dwukrotnie plik *MainPage.xaml.cs*.

- W oknie edytora kodu i tekstu odszukaj metodę *typeSelectionChanged*.



WSKAZÓWKA Chcąc zlokalizować określony element projektu, można wskazać w menu Edit menu podrzędne Find and Replace (Znajdź i zamień), a następnie wybrać z niego polecenie Quick Find (Szybkie wyszukiwanie). Spowoduje to otwarcie menu wyszukiwania w górnym prawym narożniku okna edytora kodów i tekstów. Należy wpisać wyszukiwany tekst w polu tekstowym tego okna, a następnie kliknąć przycisk Find Next (Znajdź następny) (przycisk ten znajduje się po prawej stronie pola wyszukiwania i jest oznaczony strzałką skierowaną w prawą stronę):

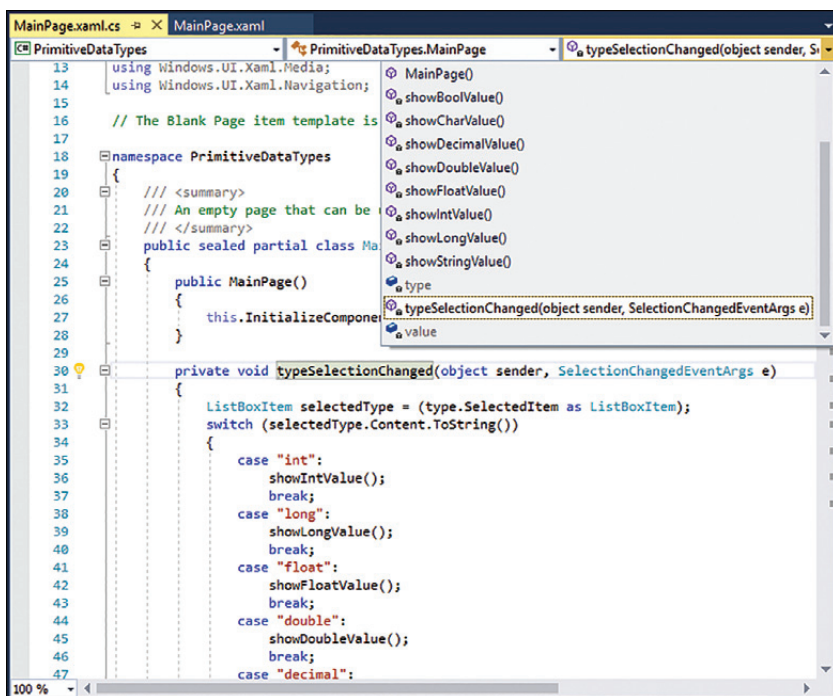


Domyślnie funkcja wyszukiwania nie rozróżnia małych i wielkich liter. Jeśli chcemy, by małe i wielkie litery były rozróżniane, to należy kliknąć przycisk Aa (Match Case, Uwzględnij wielkość liter) pod tekstem do wyszukania.

Zamiast korzystać z menu Edit można również wcisnąć kombinację klawiszy Ctrl+F, aby wyświetlić okno dialogowe Quick Find. Podobnie, wciśnięcie kombinacji klawiszy Ctrl+H spowoduje otwarcie okna dialogowego Quick Replace (Szybkie zamienianie).

Oprócz korzystania z funkcji Quick Find, alternatywnym sposobem wyszukiwania metod klasy jest użycie rozwijanej listy członków klasy, znajdującej się z prawej strony, nad oknem edytora kodu i tekstów.

Ciąg dalszy na stronie następnej



Na rozwijanej liście elementów członkowskich klasy wyświetlane są wszystkie metody tej klasy, wraz ze zmiennymi oraz innymi należącymi do niej elementami. (W dalszej części tej książki dowiemy się więcej na temat tych elementów). Np. wybranie z rozwijanej listy pozycji *typeSelectionChanged* spowoduje przeniesienie kursora bezpośrednio do metody *typeSelectionChanged* z bieżącej klasy.

Osoby posiadające pewne doświadczenie w programowaniu przy użyciu innych języków programowania zapewne domyślają się już, jak działa metoda *typeSelectionChanged*; dla pozostałych osób powinno to stać się jasne po lekturze rozdziału 4, zatytułowanego „Instrukcje wyboru”. Na razie wystarczy, jeśli będziemy wiedzieć, że kliknięcie przez użytkownika elementu listy wyświetlanej przez kontrolkę *ListBox* spowoduje przekazanie do tej metody szczegółowych danych wybranego elementu, a informacje te zostaną użyte przez metodę do określenia dalszych działań. Przykładowo, jeśli użytkownik kliknie wartość typu *float*, to nasza metoda wywoła inną metodę o nazwie *showFloatValue*.

5. Przewiń w dół kod źródłowy i odszukaj w nim metodę *showFloatValue*, która powinna wyglądać następująco:

```

private void showFloatValue()
{
    float floatVar;
    floatVar = 0.42F;

```

```
value.Text = floatVar.ToString();  
}
```

Ciało tej metody składa się z trzech instrukcji. Pierwsza z tych instrukcji deklaruje zmienną typu *float*, o nazwie *floatVar*.

Druga instrukcja powoduje przypisanie tej zmiennej wartości 0.42F.

WAŻNE Litera *F* jest w tym przypadku sufiksem typu danych mówiącym, że wartość 0.42 należy traktować jako wartość typu *float*. Gdybyśmy zapomnieli o wpisaniu litery *F*, wartość 0.42 zostałaby zinterpretowana jako wartość typu *double* i nastąpiłby błąd kompilacji, ponieważ bez wpisania dodatkowego kodu nie można przypisywać wartości jednego typu zmiennej innego typu. Język C# jest bardzo surowy pod tym względem.



Trzecia instrukcja powoduje wyświetlenie wartości tej zmiennej w znajdującym się na formularzu polu tekstowym o nazwie *value* (wartość). Instrukcja ta wymaga poświęcenia jej nieco więcej uwagi. Jak pamiętamy z rozdziału 1, wyświetlenie elementu w polu tekstowym polega na przypisaniu odpowiedniej wartości do właściwości *Text* tego pola (w rozdziale 1 zrobiliśmy to przy użyciu języka XAML). Zadanie to może być również wykonane w sposób programowy tak, jak ma to miejsce w tym przypadku. Należy zwrócić uwagę na fakt, że dostęp do właściwości obiektu realizowany jest przy użyciu tej samej notacji „kropkowej”, której używaliśmy już wcześniej do uruchamiania metody (w programie *Console.WriteLine* z rozdziału 1). Ponadto, dane przypisywane do właściwości *Text* muszą być łańcuchem znakowym, a nie liczbą. Jeśli spróbujemy przypisać liczbę do właściwości *Text*, to program się nie skompiluje. Na szczęście platforma .NET Framework ułatwia nam realizację tego zadania, oferując metodę *ToString*.

Każdy typ danych w środowisku .NET Framework ma swoją metodę *ToString*. Metoda ta służy do konwersji danego obiektu na jego reprezentację znakową. Metoda *showFloatValue* używa metody *ToString* wobec zmiennej *floatVar* klasy *float* do wygenerowania znakowej wersji wartości tej zmiennej. Otrzymany w ten sposób łańcuch znakowych można już bez przeszkód przypisać do właściwości *Text* pola tekstowego *value*. Tworząc własne typy danych i klasy można również zdefiniować swoją własną implementację metody *ToString* określającą sposób, w jaki obiekty tej klasy powinny być reprezentowane za pomocą łańcuchów znakowych. Więcej informacji na temat tworzenia własnych klas znajduje się w rozdziale 7, „Tworzenie i zarządzanie klasami oraz obiektami”.

6. Odszukaj w oknie edytora kodu i tekstu metodę *showIntValue*:

```
private void showIntValue()  
{  
    value.Text = "to do";  
}
```

Metoda *showIntValue* jest wywoływana, gdy z listy typów danych zostanie wybrany typ *int*.

7. Na początku metody *showIntValue* w nowej linii po klamrowym nawiasie otwierającym wpisz następujące dwie instrukcje pokazane pogrubioną czcionką:

```
private void showIntValue()
{
    int intVar;
    intVar = 42;
    value.Text = "to do";
}
```

Pierwsza z tych instrukcji tworzy zmienną o nazwie *intVar*, w której mogą być przechowywane wartości typu *int*. Druga instrukcja przypisuje tej zmiennej wartość 42.

8. W oryginalnej instrukcji tej metody, zmień tekst „to do” na *intVar.ToString()*; Zmieniona metoda powinna teraz wyglądać dokładnie tak jak pokazano poniżej:

```
private void showIntValue()
{
    int intVar;
    intVar = 42;
    value.Text = intVar.ToString();
}
```

9. Wybierz z menu Debug polecenie Start Debugging.
Spowoduje to ponowne wyświetlenie formularza.
10. Wybierz z listy Choose A Data Type typ danych *int*. Sprawdź, czy w polu tekstowym Sample Value została wyświetlona wartość 42.
11. Powróć do programu Visual Studio i wybierz z menu Debug polecenie Stop Debugging.
12. Odszukaj w oknie edytora kodu i tekstu metodę *showDoubleValue*.
13. Zmień treść metody *showDoubleValue* dokładnie tak, jak na pokazanym poniżej fragmencie kodu zostało to pokazane pogrubioną czcionką:

```
private void showDoubleValue()
{
    double doubleVar;
    doubleVar = 0.42;
    value.Text = doubleVar.ToString();
}
```

Kod tej metody jest podobny do kodu metody *showIntValue* z tą tylko różnicą, że tworzy on zmienną o nazwie *doubleVar*, która może przechowywać wartości typu *double* i przypisuje tej zmiennej wartość 0.42.

14. Odszukaj w oknie edytora kodu i tekstu metodę *showBoolValue*.

15. Zmień treść metody `showBoolValue` dokładnie tak, jak to zostało pokazane poniżej:

```
private void showBoolValue()
{
    bool boolVar;
    boolVar = false;
    value.Text = boolVar.ToString();
}
```

Ta metoda również jest podobna do poprzednich dwóch przykładów z tą tylko różnicą, że zmienna `boolVar` może przechowywać wyłącznie wartości logiczne (Boolean) `true` (prawda) lub `false` (fałsz). W tym przypadku została przypisana wartość `false`.

16. Wybierz z menu Debug polecenie Start Debugging.
17. Wybierz z listy Choose a Data Type po kolei następujące typy danych: `float`, `double` i `bool`. Za każdym razem sprawdź, czy w polu tekstowym Sample Value wyświetlana jest właściwa wartość.
18. Powrót do programu Visual Studio i wybierz z menu Debug polecenie Stop Debugging.

Posługiwanie się operatorami arytmetycznymi

Język C# obsługuje zwykle operatory matematyczne, o których jako dzieci nauczyliśmy się w szkole: znak plus `+` dla dodawania, znak minus `-` dla odejmowania, gwiazdkę `*` dla mnożenia i znak ukośnika `/` dla dzielenia. Symbole `+`, `-`, `*` i `/` nazywane są *operatorami*, ponieważ „operują” na wartościach, tworząc nową wartość. W poniższym przykładzie końcową wartością zmiennej `wysokoscWynagrodzeniaKonsultanta` będzie iloczyn liczby 750 (stawki dziennej) i liczby 20 (liczby dni, przez które konsultant był zatrudniony):

```
long wysokoscWynagrodzeniaKonsultanta;
wysokoscWynagrodzeniaKonsultanta = 750 * 20;
```

UWAGA Wartości, na których operuje operator, nazywane są *operandami* (lub *argumentami*). W wyrażeniu `750 * 20` znak `*` jest operatorem, a liczby 750 i 20 są jego operandami.



Operatory i typy danych

Nie wszystkie operatory można stosować dla wszystkich typów danych. To, jakich operatorów można użyć wobec określonej wartości, zależy od typu danych tej wartości. Przykładowo, operatory arytmetyczne można stosować wobec wartości typu `char`, `int`, `long`, `float`, `double` lub `decimal`. Z wyjątkiem operatora plus `+`, operatorów tych nie można jednak stosować np. wobec wartości typu `string`, a wobec wartości typu `bool` nie można użyć żadnego z nich. Tak więc pokazana poniżej instrukcja jest nieprawidłowa,

ponieważ typ *string* nie obsługuje operatora minus (wynik odejmowania jednego łańcucha znakowego od drugiego byłby nieokreślony):

```
// błąd kompilacji
Console.WriteLine("Gillingham" - "Forest Green Rovers");
```

Możliwe jest natomiast używanie operatora + do konkatenacji (scalania) wartości typu *string*. Stosując ten operator w taki sposób należy jednak zachować ostrożność, ponieważ rezultat jego działania może być różny od naszych oczekiwań. Przykładowo, pokazana poniżej instrukcja spowoduje wypisanie w oknie konsoli wartości „431” (a nie „44”):

```
Console.WriteLine("43" + "1");
```



Wskazówka Platforma .NET Framework oferuje metodę o nazwie *Int32.Parse*, za pomocą której można przekształcać łańcuchy znakowe na liczby całkowite, gdy zachodzi potrzeba wykonania operacji arytmetycznych na wartościach przechowywanych jako łańcuchy znakowe.

Interpolacja łańcuchów

W ostatniej wersji języka C# wprowadzona została nowa funkcja nazywana interpolacją łańcuchów, która sprawia, że niektóre techniki konkatenacji łańcuchów przy użyciu operatora + stają się przestarzałe.

Konkatenacja łańcuchów jest często wykorzystywana do generowania łańcuchów, które zawierają wartości zmiennych. Przykład tego zastosowania został przedstawiony w rozdziale 1 w ćwiczeniu, w którym tworzyliśmy aplikację graficzną. Do metody *onClick* dodaliśmy następujący kod:

```
MessageBox msg = new MessageBox("Hello " + userName.Text);
```

Interpolacja łańcuchów umożliwia użycie poniższej alternatywnej techniki:

```
MessageBox msg = new MessageBox($"Hello {userName.Text}");
```

Symbol \$ znajdujący się na początku łańcucha sygnalizuje, że jest to łańcuch interpolowany oraz że wszystkie wyrażenia znajdujące się między nawiasami klamrowymi { } muszą zostać przetworzone, a następnie zastąpione wynikiem. Bez początkowego symbolu \$ łańcuch {userName.Text} zostałby potraktowany dosłownie.

Interpolacja łańcuchów jest efektywniejsza niż użycie operatora + (konkatenacja łańcuchów przy pomocy operatora + może wymagać dużo pamięci ze względu na sposób przetwarzania łańcuchów w .NET Framework). Ponadto interpolacja łańcuchów jest czytelniejsza i mniej podatna na błędy.

Należy również mieć świadomość, że typ danych rezultatu działania operatora arytmetycznego zależy od typu danych użytych argumentów tego operatora. Przykładowo, wartością wyrażenia $5.0/2.0$ jest 2.5 ; obydwa argumenty są w tym przypadku typu *double*, a więc rezultat również jest typu *double*. (W języku C# literały liczbowe, w których występuje kropka* zawsze są typu *double*, a nie *float*, co pozwala zachować maksymalną możliwą dokładność). Jednakże wartością wyrażenia $5/2$ jest 2 . W tym przypadku obydwa argumenty są typu *int*, a więc rezultat również jest typu *int*. W takich sytuacjach jak ta, język C# zawsze zaokrągla wartości w dół (a ściślej mówiąc zaokrągla część w dół wartość bezwzględną rezultatu). Sytuacja staje się nieco bardziej złożona, jeśli używamy argumentów różnych typów. Przykładowo, wyrażenie $5/2.0$ zawiera kombinację typów *int* i *double*. Kompilator wykryje takie niedopasowanie typów i przed wykonaniem operacji niejawnie przekształci wartość typu *int* do typu *double*. Wynikiem tej operacji będzie więc wartość typu *double* (2.5). Wprawdzie mieszanie typów danych w jednym wyrażeniu nie jest błędem, ale jest uznawane za złą praktykę w programowaniu.

Numeryczne typy danych a wartości nieskończone

W języku C# istnieją także inne cechy liczb, o których należy wiedzieć. Przykładowo, rezultatem dzielenia dowolnej liczby przez zero jest nieskończoność, czyli wartość wykraczająca poza zakres typów danych *int*, *long* lub *decimal*; w konsekwencji próba wyznaczenia wartości wyrażenia takiego jak np. $5/0$ zakończy się błędem. Typy *double* i *float* oferują jednak specjalną wartość, która może reprezentować nieskończoność i wartością wyrażenia $5.0/0.0$ jest *Infinity* (nieskończoność). Wyjątkiem od tej reguły jest wartość wyrażenia $0.0/0.0$. Zwykle wynikiem dzielenia zera przez dowolną liczbę jest zero, a wynikiem dzielenia dowolnej liczby przez zero jest nieskończoność. Wyrażenie $0.0/0.0$ prowadzi więc do paradoksu – jego wartością musi być jednocześnie zero i nieskończoność. W języku C# istnieje jeszcze jedna specjalna wartość przewidziana właśnie na tę sytuację – jest to tak zwana wartość *NaN*, co oznacza skrót od słów „not a number” (to nie jest liczba). Tak więc wartością wyrażenia $0.0/0.0$ jest wartość *NaN*.

Wartości *NaN* i *Infinity* użyte jako argumenty wyrażenia ulegają propagacji na wynik tego wyrażenia. Wartością wyrażenia $10 + NaN$ jest wartość *NaN*, a wartością wyrażenia $10 + Infinity$ jest wartość *Infinity*. Wartością wyrażenia $Infinity * 0$ jest wartość *NaN*.

* We wszystkich językach programowania, podobnie jak w języku angielskim, część dziesiętną liczb od części całkowitej oddziela się znakiem kropki, a nie przecinka.

Język C# oferuje również jeden mniej znany operator arytmetyczny: jest to reprezentowany przez znak procentu (%) operator reszty z dzielenia lub tzw. dzielenia modulo. Wynikiem operacji $x \% y$ jest reszta pozostała z dzielenia wartości x przez wartość y . Przykładowo $9 \% 2$ jest równe 1, ponieważ 9 podzielone na 2 równa się 4 z resztą 1.



UWAGA Osoby znające język C lub C++ wiedzą, że w tych językach nie można stosować operatora dzielenia modulo na wartościach typu *float* lub *double*. Jednak język C# znosi to ograniczenie. Operator reszty z dzielenia można stosować ze wszystkimi numerycznymi typami danych, a rezultatem jego działania wcale nie musi być liczba całkowita. Przykładowo wartość wyrażenia $7.0 \% 2.4$ jest 2.2.

Poznajemy operatory arytmetyczne

Przedstawione poniżej ćwiczenia demonstrują sposób używania operatorów arytmetycznych do wykonywania obliczeń na wartościach typu *int*.

→ Uruchamianie projektu MathsOperators

1. Jeśli program Visual Studio 2017 nie został uruchomiony już wcześniej, to uruchom go teraz.
2. Otwórz projekt MathsOperators znajdujący się w katalogu \Microsoft Press\VC SBS\Chapter 2\MathsOperators.
3. Wybierz z menu Debug polecenie Start Debugging.
Zostanie wyświetlony następujący formularz:

MathsOperators

Left Operand

Right Operand

☒ + Addition
☐ - Subtraction
☐ * Multiplication
☐ / Division
☐ % Remainder

Calculate

Expression:

Result:

4. Wpisz liczbę 54 w polu tekstowym Left Operand (Lewy argument).
5. Wpisz liczbę 13 w polu tekstowym Right Operand (Prawy argument).

Po wprowadzeniu wartości w polach tekstowych możliwe jest wykonanie na nich obliczeń z użyciem dowolnego operatora.

6. Kliknij przycisk – Subtraction (Odejmowanie), a następnie kliknij przycisk Calculate (Oblicz).

Spowoduje to zmianę tekstu wyświetlanego w polu Expression (Wyrażenie) na $54 - 13$, ale w polu Result (Wynik) wyświetlona zostanie wartość 0, która oczywiście nie jest poprawną wartością tego wyrażenia.

7. Kliknij przycisk / Division (Dzielenie), a następnie kliknij przycisk Calculate.

Spowoduje to zmianę tekstu wyświetlanego w polu Expression (Wyrażenie) na $54/13$, ale w polu Result (Wynik) nadal wyświetlana będzie wartość 0.

8. Kliknij przycisk % Remainder (Reszta), a następnie kliknij przycisk Calculate.

Spowoduje to zmianę tekstu wyświetlanego w polu Expression (Wyrażenie) na $54 \% 13$, lecz w polu Result (Wynik) po raz kolejny wyświetlona zostanie wartość 0. Przetestuj inne kombinacje liczb i operatorów. Wszystkie te próby powinny na razie dawać wynik 0.

UWAGA Jeśli w polu tekstowym któregośkolwiek z argumentów zostanie wpisana wartość niebędąca liczbą całkowitą, to aplikacja wykryje to jako błąd i wyświetli komunikat „Input string was not in a correct format”. (Znakowy łańcuch wejściowy używa nieprawidłowego formatu). Sposób przechwytywania i obsługiwanie błędów i wyjątków zostanie omówiony w rozdziale 6, „Obsługa błędów i wyjątków”.



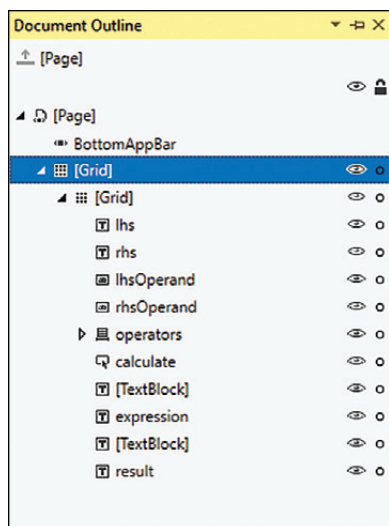
9. Po zakończeniu tych testów wróć do programu Visual Studio 2017 i wybierz z menu Debug polecenie Stop Debugging.

Jak można się domyślić, w aplikacji MathsOperators nie zaimplementowano jeszcze żadnego z tych obliczeń. Implementacja tych operacji będzie bowiem przedmiotem następnego ćwiczenia.

➔ Wykonywanie obliczeń w aplikacji MathsOperators

1. Wyświetl w oknie widoku projektowego formularz MainPage.xaml. (Kliknij dwukrotnie plik MainPage.xaml, znajdujący w panelu Solution Explorer w gałęzi projektu MathsOperators).
2. W menu View wskaż menu podrzędne Other Windows (Inne okna), a następnie wybierz polecenie Document Outline (Konspekt dokumentu).

Spowoduje to wyświetlenie okna Document Outline, w którym wyświetlane będą nazwy i typy danych dla wszystkich znajdujących się na formularzu kontroltek.



Okno Document Outline zapewnia możliwość łatwego wyszukiwania i zaznaczania kontroltek, znajdujących się na złożonym formularzu WPF. Wszystkie kontrolki są uporządkowane w hierarchię rozpoczynającą się od elementu *Page*, reprezentującego sam formularz. Jak już wspomniano w poprzednim rozdziale, strona aplikacji Universal Windows Platform (UWP) zawiera kontrolkę typu *Grid*, w której umieszczane są pozostałe kontrolki. Rozwinięcie w oknie Document Outline węzła *Grid* spowoduje wyświetlenie pozostałych kontroltek, a pierwszą z nich będzie kolejna kontrolka typu *Grid* (zewnętrzna kontrolka typu *Grid* pełni funkcję ramki [frame], a wewnętrzna kontrolka *Grid* zawiera kontrolki umieszczane na formularzu). Po rozwinięciu wewnętrznej kontrolki *Grid* wyświetlone zostaną wszystkie kontrolki umieszczone na formularzu.

Kliknięcie którejkolwiek z tych kontroltek spowoduje zaznaczenie odpowiedniego elementu w oknie widoku projektowego. Podobnie, zaznaczenie kontrolki w oknie widoku projektowego spowoduje również zaznaczenie odpowiedniej kontrolki w oknie Document Outline (zaobserwowanie tego działania może wymagać przypięcia okna Document Outline poprzez odznaczenie przycisku Auto Hide [Autoukrywanie], znajdującego się w prawym górnym rogu tego okna).

3. Kliknij po kolei, znajdujące się na formularzu dwie kontrolki typu *TextBox*, służące do wprowadzania wartości liczbowych. Korzystając z okna Document Outline sprawdź, czy nazwy tych kontroltek to *lhsOperand* oraz *rhsOperand*.

Po uruchomieniu formularza wprowadzone przez użytkownika wartości przechowane są we właściwościach *Text* tych kontroltek.

4. Sprawdź, czy znajdujące się w dolnej części formularza dwie kontrolki typu *TextBlock*, służące do wyświetlania obliczanego wyrażenia oraz wyniku obliczeń, mają odpowiednio nazwy *expression* (wyrażenie) oraz *result* (wynik).
5. Zamknij okno Document Outline.
6. Wybierz z menu View polecenie Code (Kod), aby wyświetlić w oknie edytora kodu i tekstu zawartość pliku MainPage.xaml.cs.
7. Odszukaj w oknie edytora kodu i tekstu metodę *addValues* (dodaj wartości). Metoda ta wygląda następująco:

```
private void addValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: dodać rhs do lhs i zapisać wynik w zmiennej outcome
    expression.Text = $"{lhsOperand.Text} + {rhsOperand.Text}";
    result.Text = outcome.ToString();
}
```

Pierwsza instrukcja tej metody deklaruje zmienną typu *int* o nazwie *lhs* i inicjalizuje ją, przypisując jej liczbę całkowitą odpowiadającą liczbie wpisanej przez użytkownika w polu tekstowym *lhsOperand*. Należy pamiętać, że właściwość *Text* kontrolki typu *TextBox* zawiera łańcuch znakowy typu *string*, a zmienna *lhs* jest typu *int*, a więc przed przypisaniem tego łańcucha do zmiennej *lhs* konieczne jest jego przekształcenie na liczbę całkowitą. Typ danych *int* oferuje metodę *int.Parse*, która wykonuje właśnie tę operację.

Druga instrukcja deklaruje zmienną typu *int* o nazwie *rhs* i inicjalizuje ją wartością wpisaną w polu tekstowym *rhsOperand*, po uprzednim przekształceniu tej wartości do typu *int*.

Trzecia instrukcja deklaruje zmienną typu *int* o nazwie *outcome*.

Kolejna linia zawiera komentarz informujący, że należy dodać do siebie wartości zmiennych *rhs* i *lhs*, a wynik zapisać w zmiennej *outcome*. Jest to właśnie ta brakująca część kodu, którą mamy zaimplementować w kolejnym kroku.

Piąta instrukcja wykorzystuje interpolację łańcuchów do skonstruowania łańcucha określającego rodzaj wykonywanej operacji i przypisuje rezultat do właściwości *expression.Text*. Powoduje to wyświetlenie wynikowego łańcucha znakowego w znajdującym się na formularzu polu tekstowym *Expression* (wyrażenie).

Ostatnia instrukcja wyświetla rezultat obliczeń poprzez przypisanie go do właściwości *Text* pola tekstowego *Result* (wynik). Należy pamiętać, że właściwość *Text* jest typu *string*, a rezultat obliczeń jest typu *int*, a więc przed przypisaniem go do właściwości *Text* konieczne jest przeprowadzenie konwersji do typu *string*. Jak pamiętamy, zadanie to realizuje metoda typu danych *int* o nazwie *ToString* (na łańcuch znakowy).

8. Poniżej linii z komentarzem, znajdującej się w połowie metody *addValues*, dodaj instrukcję wyróżnioną poniżej za pomocą pogrubionej czcionki:

```
private void addValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: dodać rhs do lhs i zapisać wynik w zmiennej outcome
    outcome = lhs + rhs;
    expression.Text = $"{lhsOperand.Text} + {rhsOperand.Text}";
    result.Text = outcome.ToString();
}
```

Instrukcja ta wyznacza wartość wyrażenia *lhs + rhs* i zapisuje rezultat w zmiennej *outcome*.

9. Zapoznaj się z treścią metody *subtractValues*. Metoda ta jest zbudowana w podobny sposób jak poprzednia i należy do niej dodać instrukcję obliczającą rezultat odejmowania wartości zmiennej *rhs* od *lhs* i zapisującą rezultat w zmiennej *outcome*. Dodaj do tej metody instrukcję wyróżnioną poniżej za pomocą pogrubionej czcionki:

```
private void subtractValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: odjąć rhs od lhs i zapisać wynik w zmiennej outcome
    outcome = lhs - rhs;
    expression.Text = $"{lhsOperand.Text} - {rhsOperand.Text}";
    result.Text = outcome.ToString();
}
```

10. Zapoznaj się z treścią metod *multiplyValues*, *divideValues* oraz *remainderValues*. We wszystkich tych metodach również brakuje decydującej instrukcji, wykonującej odpowiednie obliczenia. Dodaj odpowiednie instrukcje do każdej z tych metod (wyróżnione poniżej za pomocą pogrubionej czcionki).

```
private void multiplyValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: pomnożyć rhs przez lhs i zapisać wynik w zmiennej outcome
    outcome = lhs * rhs;
    expression.Text = $"{lhsOperand.Text} * {rhsOperand.Text}";
    result.Text = outcome.ToString();
}

private void divideValues()
{
    int lhs = int.Parse(lhsOperand.Text);
```

```

    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: podzielić lhs przez rhs i zapisać wynik w zmiennej outcome
    outcome = lhs / rhs;
    expression.Text = $"{lhsOperand.Text} / {rhsOperand.Text}";
    result.Text = outcome.ToString();
}

private void remainderValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: Wyznaczyć resztę z dzielenia lhs przez rhs i zapisać rezultat
    outcome = lhs % rhs;
    expression.Text = $"{lhsOperand.Text} % {rhsOperand.Text}";
    result.Text = outcome.ToString();
}

```

➔ Testowanie aplikacji MathsOperators

1. Wybierz z menu Debug polecenie Start Debugging, aby zbudować i uruchomić poprawioną aplikację.
2. Wpisz liczbę 54 w polu Left Operand, wpisz liczbę 13 w polu Right Operand, kliknij opcję + Addition, a następnie kliknij przycisk Calculate. W polu Result (Wynik) powinna wówczas zostać wyświetlona wartość 67.
3. Kliknij przycisk Subtraction, a następnie kliknij przycisk Calculate. Sprawdź, czy wynik jest teraz równy 41.
4. Kliknij przycisk Multiplication, a następnie kliknij przycisk Calculate. Sprawdź, czy wynik jest teraz równy 702.
5. Kliknij przycisk Division, a następnie kliknij przycisk Calculate. Sprawdź, czy wynik jest teraz równy 4.

W rzeczywistości wynikiem dzielenia 54/13 jest ułamek okresowy 4.(153846), ale tu mamy do czynienia z językiem C#, wykonującym dzielenie całkowitoliczbowe. Jak już wyjaśnialiśmy wcześniej, wynikiem dzielenia jednej liczby całkowitej przez inną liczbę całkowitą jest również liczba całkowita.

6. Kliknij przycisk Remainder, a następnie kliknij przycisk Calculate. Sprawdź, czy wynik jest teraz równy 2.

W przypadku liczb całkowitych, resztą pozostałą z dzielenia 54 przez 13 jest właśnie 2; $(54 - ((54/13) * 13))$ jest równe 2. Jest to skutek tego, że wszystkie wyniki pośrednie są zaokrąglane w dół (mój nauczyciel matematyki byłby przerażony słysząc, że $(54/13) * 13$ nie równa się 54!).

7. Powróć do programu Visual Studio i zatrzymaj debugowanie.