Microsoft

John Sharp

Microsoft Visual C# 2015
Krok po kroku

Wydanie ósme

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

APN Promise, Warszawa 2016

Microsoft Visual C# 2015 Krok po kroku

Authorized Polish translation of the English language edition entitled: Microsoft Visual C# Step by Step, 8th Edition, ISBN 978-1-5093-0104-1, by John Sharp, published by Pearson Education, Inc, publishing as Microsoft Press, A Division Of Microsoft Corporation.

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

ISBN: 978-83-7541-179-9

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

Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Spis treści

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. Język C# w wersji 1.0 miał swój publiczny debiut w roku 2001. Pojawienie się wersji C# 2.0 wraz z programem Visual Studio 2005 oznaczało dodanie do tego języka kilku ważnych i nowych funkcji, takich jak ogólne typy wyliczeniowe i metody anonimowe. W wersji C# 3.0, która pojawiła się wraz z opublikowaniem programu Visual Studio 2008, dodana została obsługa 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 najnowszej wersji języka C# 5.0 dodano natywną obsługę dla przetwarzania zadań w sposób asynchroniczny poprzez użycie modyfikatora metody *async* oraz operatora *await*. Wersja C# 6.0 wprowadza kolejne ulepszenia zaprojektowane 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. Funkcje te zostaną opisane w niniejszej książce.

Innym ważnym wydarzeniem dla firmy Microsoft jest premiera systemu Windows 10. Ta nowa wersja systemu Windows łączy w sobie najlepsze (i najbardziej lubiane) aspekty poprzednich wersji systemu operacyjnego z obsługą aplikacji o wysokim stopniu interaktywności, które mogą współdzielić pomiędzy sobą dane i współpracować ze sobą nawzajem lub łączyć się z usługami działającymi w chmurze. Kluczowy element wersji Windows 10 stanowią aplikacje Universal Windows Platform (UWP) – zaprojektowane tak, aby mogły być uruchamiane na dowolnym urządzeniu Windows 10, począwszy od w 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 postanowiliśmy poświęcić temu aspektowi rozwoju oprogramowania ostatni rozdział książki.

Środowisko programowania oferowane przez pakiet Visual Studio 2015 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 użyciu programu Visual Studio 2015 oraz platformy .NET Framework w wersji 4.6. 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.

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.

---

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

Organizacja książki

Niniejsza książka została podzielona na następujące cztery części:

- Część I, zatytułowana „Wprowadzenie do języka Microsoft Visual C# oraz programu Microsoft Visual Studio 2015" 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
Początkującym programistą 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 poziomu swoich potrzeb.

Jeżeli jesteś	Wykonaj następujące kroki
Programistą dobrze obeznanym z proceduralnymi językami programowania, takimi jak np. język C, ale nie znasz jeszcze 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 pięciu rozdziałów, aby uzyskać ogólny obraz języka C# oraz możliwości programu Visual Studio 2015, 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 2015, 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 6 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 2015.
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 uraz 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 (poza blokami kodu), wyróżniany jest za pomocą wytłuszczonej czcionki.
- Znak plus (+) występujący pomiędzy nazwami dwóch klawiszy oznacza, że klawisze te należy wcisnąć równocześnie. Np. zdanie „Wciśnij klawisze Alt+Tab" oznacza, że najpierw należy wcisnąć klawisz Alt, a następnie trzymając ten klawisz wciśnięty wcisnąć klawisz Tab.

Wymagania systemowe

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

- System operacyjny Windows 10 edycja Professional (lub wyższa).
- Edycja Visual Studio Community 2015, Visual Studio Professional 2015 lub Visual Studio Enterprise 2015.

> **Ważne** Trzeba zainstalować narzędzie programistyczne Windows 10 w Visual Studio 2015.

- Komputer z procesorem 1.6 GHz lub szybszym (zaleca się 2 GHz).
- 1 GB (w systemach 32-bitowych) lub 2 GB (w systemach 64-bitowych) pamięci RAM (w przypadku pracy na maszynie wirtualnej, wielkości te należy powiększyć o dodatkowe 512 MB).

- 10 GB wolnego miejsca na dysku twardym.
- Dysk twardy o prędkości obrotowej 5400 RPM lub szybszy.
- Karta graficzna kompatybilna ze standardem DirectX 9, o rozdzielczości 1024 × 768 lub wyższej.
- Napęd optyczny DVD-ROM (w przypadku instalowania oprogramowania Visual Studio z płyty DVD).
- 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 2015 może wymagać posiadania uprawnień lokalnego administratora.

Ponadto trzeba włączyć na komputerze tryb dewelopera, aby móc tworzyć i uruchamiać aplikacje UWP. Dodatkowe informacje dotyczące osiągania tego celu znaleźć można w artykule „Enable Your Device for Development" (w jęz. angielskim) o adresie *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 webowej:

http://aka.ms/sharp8e/companioncontent

Uwaga Oprócz pobrania przykładowych kodów źródłowych, należy pamiętać także o konieczności zainstalowania w systemie pakietu oprogramowania Visual Studio 2015. 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 CSharpSBS.zip pobrany ze strony web powiązanej z książką do swojego katalogu Dokumenty.
2. Zaakceptuj postanowienia licencyjne w wyświetlonym monicie.

Uwaga Jeśli podczas wykonywania tych operacji nie zostanie wyświetlony tekst umowy licencyjnej, to 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 2015, 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 2015.
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.

Projekt	Opis
DoStatement	Projekt wykorzystujący instrukcję do do przekształcenia liczby dziesiętnej na jej reprezentację ósemkową.
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.

Projekt	Opis
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.
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 opracowanym 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.

Projekt	Opis
Chapter 19	
BinaryTree	Projekt demonstrujący sposób zaimplementowania ogólnego interfejsu typu IEnumerator<T>, który umożliwia utworzenie obiektu wyliczeniowego dla ogólnej klasy Tree.
IteratorBinaryTree	Rozwiązanie wykorzystujące iterator do wygenerowania typu wyliczeniowego 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.
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 klika 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.

Projekt	Opis
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.
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 *INotifyPropertyChanged*, 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.

Projekt	Opis
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.

Podziękowania

Pomimo faktu, że na okładce tej książki 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.

W pierwszej kolejności swoje podziękowania kieruję do Devona Musgrave w wydawnictwie Microsoft Press, który przebudził mnie z pisarskiego letargu (w rzeczywistości byłem dość zajęty przygotowywaniem materiałów dla Microsoft Patterns & Practices, ale udało mi się wziąć przedłużony urlop na czas pracy nad tą edycją książki). Zaczepiał mnie, przekabacał i uświadamiał rychłe nadejście wersji Windows 10 i Visual Studio 2015, przygotował kontrakt z uzgodnionymi terminami i nie spoczął, dopóki nie podpisałem go własną krwią!

Chciałbym podziękować również Jasonowi Lee, mojemu byłemu podwładnemu, a teraz przełożonemu w firmie Content Master (to dość skomplikowana historia, ale wygląda na to, że znalazł on pewne interesujące negatywy, które nieopatrznie zostawiłem na wierzchu). Jason zajął się mozolnym zadaniem generowania nowych zrzutów ekranu i sprawdzaniem, czy kod około dwudziestu pierwszych rozdziałów został zaktualizowany (i poprawiony). Jeśli w tej edycji pojawią się jakieś błędy, chętnie zrzuciłbym winę na niego, ale oczywiście jako osoba dokonująca końcowej redakcji ponoszę pełną odpowiedzialność.

Podziękowania należą się także Marcowi Young, któremu w udziale przyszło dość żmudne zadanie analizowania mojego kodu w celu upewnienia się, że może on zostać skompilowany i uruchomiony. Jego porady okazały się bardzo pomocne.

Oczywiście, podobnie jak u wielu programistów, pomimo znajomości opisywanych technologii mój język zapewne nie zawsze jest tak płynny i przejrzysty, jak można by oczekiwać. Dlatego chciałem podziękować Johnowi Pierce za korektę błędów gramatycznych i ortograficznych oraz za ogólną poprawę czytelności prezentowanego materiału, dzięki której stał się on bardziej zrozumiały.

Na zakończenie, muszę podziękować mojej biednej żonie Dianie, która obawiała się, że powoli tracę rozum (co mogło być prawdą), gdy zacząłem mamrotać pod nosem różne dziwne frazy do laptopa, aby przekonać Cortanę do współpracy z moją aplikacją. Diana myślała, że non stop dzwonię do „Orlanda Gee", ponieważ dość głośno wykrzykiwałem to imię i nazwisko (przykładowe dane klienta wykorzystywane przeze mnie w ćwiczeniach tej książki). Niestety ze względu na mój akcent, Cortana myślała, że chodzi mi o Orlanda T, Orlanda Key lub nawet Orlanda Quay, co zmusiło mnie w końcu do zmiany imienia i nazwiska stosowanego w ćwiczeniach na Brian Johnson. Zasłyszałem nawet fragment rozmowy Diany z Jasonem, dekoratorem zajmującym się malowaniem naszego holu, w której rozważała ona możliwość przekształcenia jednego z pokoi w izolatkę obitą poduszkami, tak bardzo niepokoił ją mój stan psychiczny! Ale to już „water under the bridge" (upłynęło, jak woda pod mostem) lub „water under the breach" (woda pod wyłomem), według Cortany interpretującej mój specyficzny akcent, który wywodzi się częściowo z Gloucester i częściowo z Kent.

Kończąc zakończenie, muszę wspomnieć moją córkę Francescę – w przeciwnym przypadku poczułaby się ona ogromnie urażona. Chociaż nadal mieszka z nami, jest już całkiem dorosła i pracuje dla firmy programistycznej w Cam, Gloucestershire (nie wspominam ich nazwy, ponieważ jeszcze nie otrzymałem od nich żadnych gratisów).

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:

http://aka.ms/sharp8e/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.

Oczekujemy na uwagi Czytelników

Satysfakcja Czytelników jest najwyższym priorytetem wydawnictwa Microsoft Press i dlatego komentarze Czytelników są dla nas niezwykle cenne. Prosimy o dzielenie się swoimi opiniami na temat tej książki pod adresem:

http://aka.ms/tellpress

Pod podanym powyżej adresem dostępna jest krótka ankieta i zapewniamy, że czytamy wszystkie komentarze i pomysły naszych Czytelników. Z góry dziękujemy za czas poświęcony na jej wypełnienie!

Pozostańmy w kontakcie

Pozostań w kontakcie z nami! Wydawnictwo Microsoft Press obecne jest na Twitterze pod następującym adresem: *http://twitter.com/MicrosoftPress*

CZĘŚĆ I

Wprowadzenie do języka Microsoft Visual C# oraz programu Microsoft Visual Studio 2015

W tej początkowej część książki przedstawione zostaną kluczowe aspekty języka C# i zademonstrowane podstawowe techniki budowania aplikacji w Visual Studio 2015.

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 debuger 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 2015.
- 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 2015, środowiskiem programowania oraz zestawem narzędzi stworzonych z myślą o ułatwieniu programiście tworzenia aplikacji przeznaczonych dla systemu Microsoft Windows. Program Visual Studio 2015 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 2015 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 2015

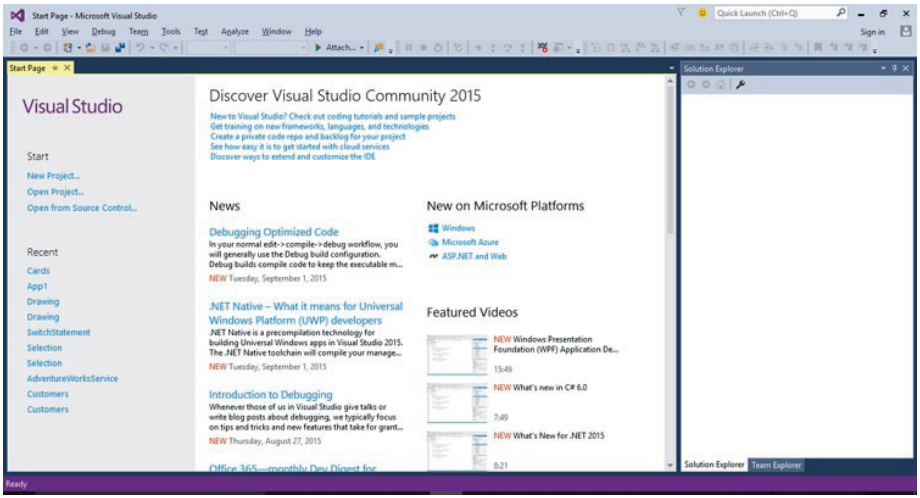
Visual Studio 2015 to rozbudowane środowisko programowania, oferujące funkcjonalność potrzebną przy tworzeniu zarówno małych, jak i dużych projektów w języku C#, przeznaczonych dla systemu Windows. Możliwe jest nawet konstruowanie projektów, które w gładki i bezproblemowy sposób łączą ze sobą moduły 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 2015 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 2015

1. W pasku zadań systemu Windows kliknij Start, wpisz **Visual Studio 2015**, a następnie naciśnij klawisz Enter.

Spowoduje to uruchomienie programu Visual Studio 2015 i wyświetlenie przez ten program strony Start, takiej jak ta pokazana poniżej (w zależności od używanej edycji programu Visual Studio 2015, strona Start na komputerze użytkownika może wyglądać nieco inaczej):



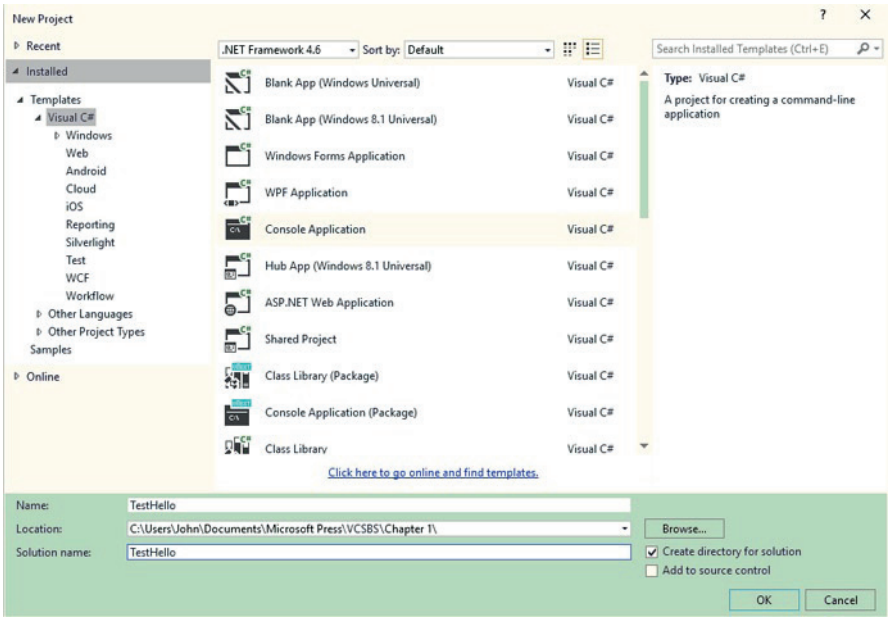
Uwaga Przy pierwszym uruchomieniu programu Visual Studio 2015 może zostać wyświetlone okno dialogowe umożliwiające wybór domyślnych ustawień środowiska programowania. Program Visual Studio 2015 dopasowuje się do preferowanego przez użytkownika języka programowania. Wybór języka programowania spowoduje zmianę domyślnych ustawień w różnych oknach dialogowych i narzędziach środowiska IDE (Integrated Development Environment – zintegrowane środowisko programowania). Należy wybrać z listy pozycję Visual C#, a następnie kliknąć przycisk Start Visual Studio (Uruchom Visual Studio). Spowoduje to wyświetlenie po chwili okna środowiska IDE Visual Studio 2015.

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, które mogą zostać użyte 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, a następnie kliknij znajdującą się w tym panelu ikonę Console Application (Aplikacja konsolowa).



4. W polu Location (Lokalizacja) wpisz **C:\Users*TwojaNazwa*\Dokumenty\Microsoft Press\VCSBS\Chapter 1**. Występującą w tej ścieżce frazę *TwojaNazwa* zastąp własną nazwą użytkownika w systemie Windows.

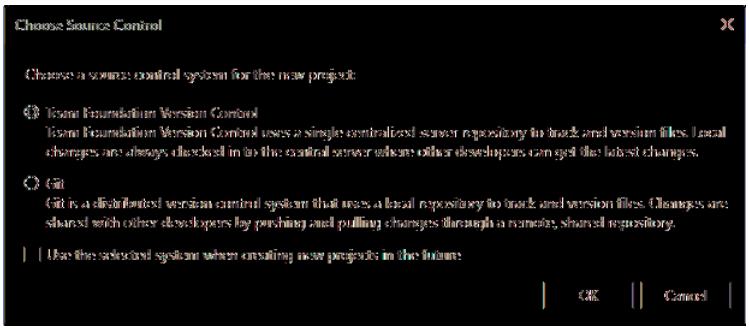
Uwaga Dla skrócenia zapisu, w dalszej części tej książki, zamiast odwoływać się do ścieżki C:\Users*TwojaNazwa*\Dokumenty, będziemy pisać po prostu o folderze Dokumenty użytkownika.

Wskazówka Jeśli określony folder nie istnieje, zostanie stworzony przez Visual Studio 2015.

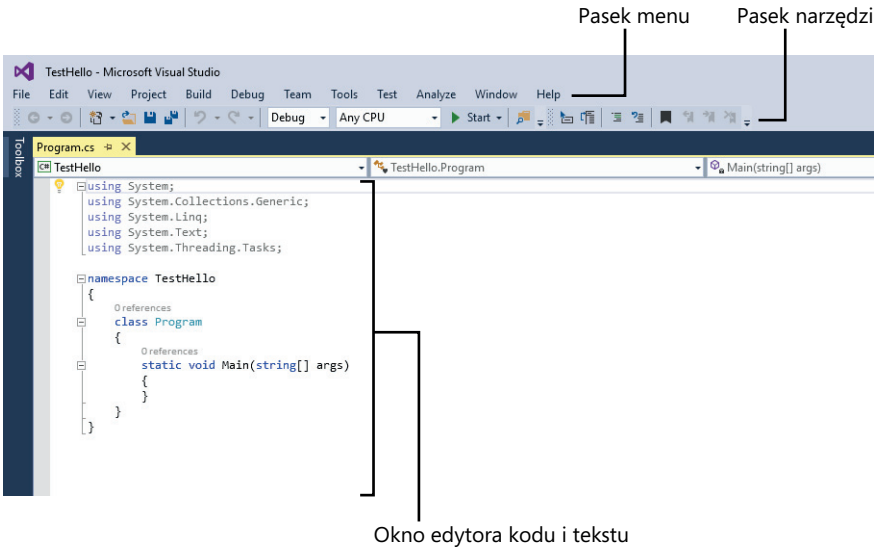
---

5. W polu Name (Nazwa) wpisz tekst **TestHello**, nadpisując znajdującą się w tym polu nazwę ConsoleApplication1.
6. 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 (Aplikacja konsolowa). Wyświetlenie następującego okna dialogowego monitującego o wybranie mechanizmu kontroli źródła do użycia oznacza, że pomyłkowo zaznaczone zostało pole wyboru Add To Source Control (Dodaj do kontroli źródła). W takiej sytuacji wystarczy kliknąć przycisk Cancel (Anuluj) i projekt zostanie utworzony bez kontroli źródła.



Visual Studio wyświetli początkowy kod aplikacji tak, jak to zostało pokazane na poniższym rysunku:

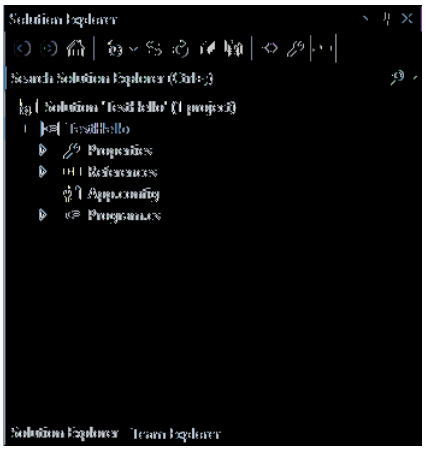


Pasek menu, znajdujący się w górnej części ekranu, zapewnia dostęp do różnych funkcjonalności używanych w środowisku programowania. Wszelkie polecenia oraz różne pozycje menu można uruchamiać przy pomocy klawiatury lub myszy, dokładnie w taki sam sposób, w jaki odbywa się to we wszystkich programach

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 cześć 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 (Eksplorator rozwiązań) 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 2015 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 2015 tworzy plik rozwiązania, aby ułatwić organizowanie projektów. Jeśli sprawdzimy za pomocą Eksploratora zawartość swojego 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 użytkownika \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 unikalna nazwa. W środowisku Microsoft .NET Framework biblioteki te nazywane są *zestawami wykonywalnymi* (ang. assembly). Programiści mogą używać zestawów wykonywalnych do umieszczania w nich napisanych przez siebie użytecznych fragmentów kodu w sposób umożliwiający ich dystrybuowanie i używanie przez innych programistów we własnych aplikacjach. Jeśli rozwiniemy folder References, to przekonamy się, że zawiera on zestaw domyślnych odwołań, dodanych do projektu przez program Visual Studio 2015. Te zestawy wykonywalne zapewniają dostęp do wielu powszechnie wykorzystywanych funkcji platformy .NET Framework i zostały dostarczone przez firmę Microsoft wraz z oprogramowaniem Visual Studio 2015. Podczas wykonywania zamieszczonych w tej książce ćwiczeń będziemy mieli okazję zapoznać się dokładniej z wieloma z tych zestawów wykonywalnych.
- **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 2015, a 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, zatytułowanym „Tworzenie i zarządzanie klasami oraz obiektami", natomiast więcej informacji na temat struktur znaleźć można w rozdziale 9, zatytułowanym „Tworzenie typów wartości 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, zatytułowanym „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.

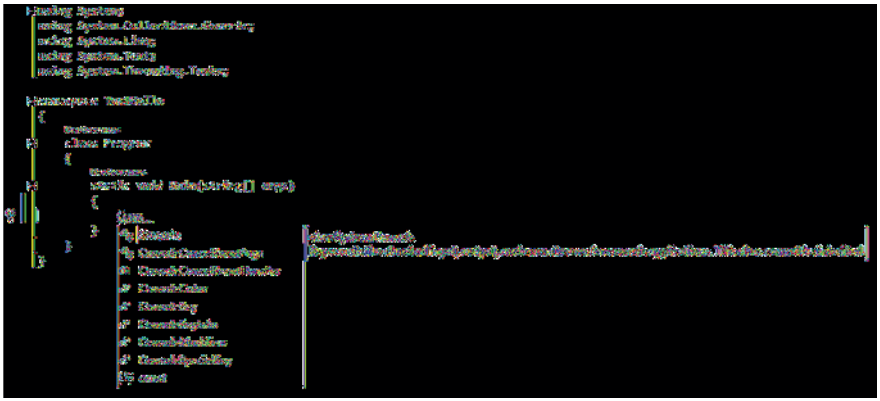
W kolejnych ćwiczeniach napiszemy kod wyświetlający w oknie konsoli komunikat „Hello World!" (Witaj świecie!); zbudujemy i uruchomimy naszą aplikację konsolową Hello World oraz poznamy sposób wykorzystywania przestrzeni nazw do dzielenia kodu na różne elementy.

➜ 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 zestawów wykonywalnych dołączonych automatycznie do naszej aplikacji. Klasa ta oferuje metody pozwalające na wyświetlanie komunikatów 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 wcisnąć klawisz Enter.

   Okienko z listą funkcji IntelliSense zostanie wówczas zamknięte, 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 znaki takie jak nawiasy zwykłe (i) oraz klamrowe { i } wpisywać parami, jeszcze przed wypełnieniem ich treścią. Jeśli odłożymy wpisanie znaku nawiasu zamykającego do chwili wprowadzenia treści, która powinna być ujęta w te nawiasy, można łatwo zapomnieć o wpisaniu znaku zamykającego.

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:

Ciąg dalszy na stronie następnej

Ciąg dalszy ze strony poprzedniej

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 znaku ukośnika i gwiazdki – /*. Po napotkaniu tych znaków kompilator pomija wszystko, aż do napotkania sekwencji znaków 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

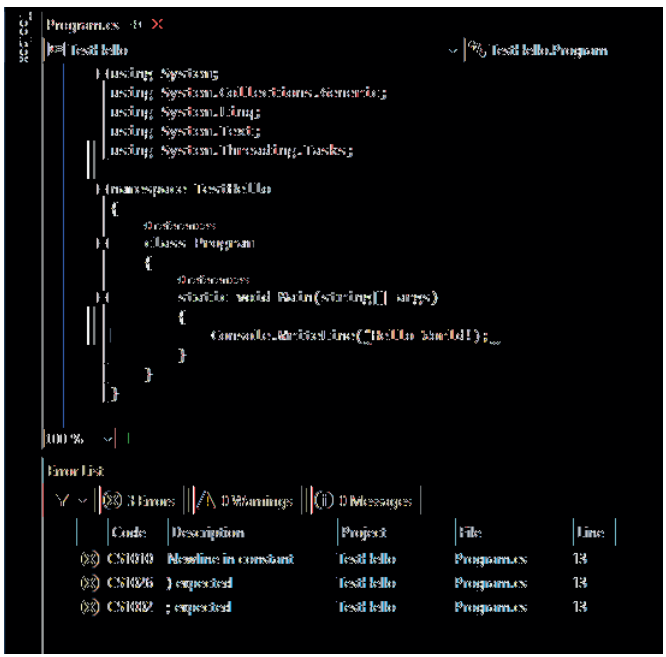
1. Wybierz z menu Build (Kompilowanie) polecenie Build Solution (Kompiluj rozwiązanie).

   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 (Dane wyjściowe).

W oknie Output (Dane wyjściowe) powinien wówczas zostać wyświetlony komunikat podobny do tego, który został pokazany 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 (Lista błędów) 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 (Kompiluj rozwiązanie) 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.

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

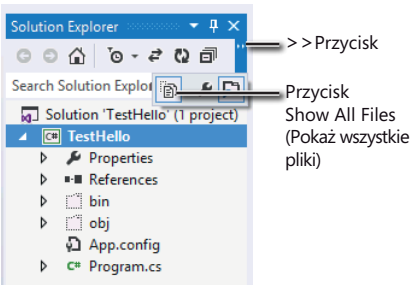
   Spowoduje to otwarcie okna wiersza poleceń i uruchomienie programu. Uruchomiony program wyświetli komunikat Hello World! i będzie oczekiwać na wciśnięcie przez użytkownika dowolnego klawisza, tak jak to zostało pokazane na poniższym rysunku:

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 2015.

4. W oknie Solution Explorer (Eksplorator rozwiązań) 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. Należy pamiętać, że wyświetlenie tego przycisku może wymagać kliknięcia przycisku >>, znajdującego się po prawej stronie paska narzędziowego eksploratora rozwiązań.



   Kliknięcie tego przycisku spowoduje wyświetlenie ponad plikiem *Program.cs* elementów o nazwach *bin* i *obj*. Elementy te odpowiadają folderom *bin* i *obj*, znajdującym się 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 (Eksplorator rozwiązań) rozwiń gałąź *bin*.

   Ukaże 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 (Eksplorator rozwiązań) 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 (Uruchom bez debugowania). Pozostałe trzy pliki zawierają informacje, które są używane przez program Visual Studio 2015 podczas uruchamiania programu w trybie debugowania – po wybraniu z menu Debug (Debugowanie) polecenia Start Debugging (Rozpocznij debugowanie).

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 procesu budowy projektu, spowodowane konfliktem dwóch lub więcej nazw; np. na skutek próby utworzenia dwóch klas o takiej samej nazwie. Sytuacja komplikuje się jeszcze bardziej, gdy tworzony program odwołuje się do zestawów wykonywalnych 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 pomylone, 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 *NowaPrzestrzeńNazw* i zestaw wykonywalny zawierający tę klasę zostanie zainstalowany 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 *NowaPrzestrzeńNazw.Pozdrowienia*.

Dobre praktyki programowania wymagają, aby wszystkie tworzone klasy były definiowane przy użyciu przestrzeni nazw, do której należą i środowisko Visual Studio 2015 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 unikalnych nazw, takich jak np. *SystemConsole*. Na szczęście problem ten można rozwiązać stosując w swoich programach dyrektywę *using*. Jeśli cofniemy się do otwartego w środowisku Visual Studio 2015 projektu *TestHello* 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 2015 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*.

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 (Kompilowanie) polecenie Build Solution (Kompiluj rozwiązanie).

   Proces kompilacji zakończy się niepowodzeniem, a w oknie Error List (Lista błędów) 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 (Kompilowanie) polecenie Build Solution (Kompiluj rozwiązanie).

   Tym razem proces budowania 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 (Debugowanie) polecenie Start Without Debugging (Uruchom bez debugowania), 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 programu Visual Studio 2015.

Przestrzenie nazw a wykonywalne zestawy 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 tzw. *binarne zestawy wykonywalne* (ang. assemblies). Binarny zestaw wykonywalny to plik, który zwykle ma rozszerzenie .dll, ale ściśle rzecz biorąc, są nimi również programy wykonywalne z rozszerzeniem .exe.

Binarny zestaw wykonywalny może zawierać wiele klas. Klasy składające się na bibliotekę klas platformy .NET Framework, takie jak np. *System.Console*, są dostarczane w zestawach binarnych instalowanych na komputerze razem z programem Visual Studio. Jak wkrótce się przekonamy, biblioteka klas platformy .NET Framework zawiera tysiące różnych klas. Gdyby wszystkie te klasy znajdowały się w jednym zestawie binarnym, miałby on bardzo duży rozmiar i był trudny do utrzymania (gdyby firma Microsoft uaktualniła tylko jedną metodę pojedynczej klasy, musiałaby na nowo rozdystrybuować całą bibliotekę klas do wszystkich programistów!).

Z tego względu biblioteka klas platformy .NET Framework została podzielona na kilka mniejszych zestawów wykonywalnych, odpowiadających różnym obszarom funkcjonalnym, z którymi związane są zawarte w nich klasy. Istnieje np. „zasadniczy" zestaw wykonywalny (zestaw ten nosi nazwę *mscorlib.dll*), który zawiera wszystkie typowe klasy, takie jak np. *System.Console*, a także inne zestawy wykonywalne, zawierające klasy służące do manipulowania bazami danych, korzystania z usług webowych, tworzenia graficznego interfejsu użytkownika itd. Jeśli zamierzamy skorzystać z klasy zawartej w binarnym zestawie wykonywalnym, konieczne jest dodanie we własnym projekcie odwołania do takiego zestawu. Następnie można dodać w kodzie źródłowym dyrektywę *using*, która spowoduje włączenie do zasięgu (ang. scope) elementów z przestrzeni nazw zawartych w danym zestawie binarnym.

Należy w tym miejscu podkreślić, że relacja pomiędzy binarnym zestawem wykonywalnym a przestrzenią nazw niekoniecznie musi być relacją typu 1:1. Pojedynczy zestaw wykonywalny może zawierać klasy zdefiniowane w wielu różnych przestrzeniach nazw, a pojedyncza przestrzeń nazw może rozciągać się na kilka zestawów wykonywalnych. Przykładowo, klasy oraz inne elementy z przestrzeni nazw *System* zostały faktycznie zaimplementowane w formie kilku różnych zestawów wykonywalnych, między innymi *mscorlib.dll*, *System.dll*, oraz

Ciąg dalszy na stronie następnej

System.Core.dll. Wszystko to może początkowo wydawać się bardzo zagmatwane, ale szybko można się do tego przyzwyczaić.

Szablon wybrany podczas tworzenia nowej aplikacji za pomocą programu Visual Studio powoduje automatyczne dołączenie odwołań do właściwych zestawów binarnych. Rozwińmy np. folder *References* (Odwołania), widoczny w oknie Solution Explorer (Eksplorator rozwiązań) dla otwartego projektu *TestHello*. Zobaczymy wówczas, że użycie szablonu *Console application* (Aplikacja konsolowa) spowodowało dołączenie odwołań do zestawów binarnych 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 pewne podstawowe funkcje i musi być używany przez wszystkie aplikacje korzystające z platformy .NET Framework. Folder *References* zawiera bowiem tylko opcjonalne zestawy wykonywalne i jeśli zachodzi taka potrzeba, to możliwe jest dodawanie nowych i usuwanie istniejących zestawów wykonywalnych z tego folderu.

Dodatkowe odwołania do zestawów wykonywalnych 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 zestawu wykonywalnego 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 2015 do utworzenia i uruchomienia prostej aplikacji konsolowej. Środowisko programowania Visual Studio 2015 zawiera także wszystko, czego będziemy potrzebować do tworzenia graficznych aplikacji dla systemu Windows 10. Udostępnia szablony nazywane uniwersalnymi aplikacjami systemu Windows (Universal Windows Platform – UWP), ponieważ umożliwiają tworzenie aplikacji, które mogą być uruchamiane na dowolnym urządzeniu z systemem Windows, takim jak komputer, tablet czy telefon. Graficzny interfejs użytkownika dla systemu Windows można projektować w sposób interaktywny. Oprogramowanie Visual Studio 2015 wygeneruje następnie odpowiednie instrukcje programu, implementujące zaprojektowany interfejs użytkownika.

Środowisko Visual Studio 2015 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.

Zamieszczony poniżej zestaw ćwiczeń demonstruje sposób tworzenia aplikacji graficznej przy użyciu Visual Studio 2015. 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.

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

➜ Tworzenie aplikacji graficznej w środowisku Visual Studio 2015

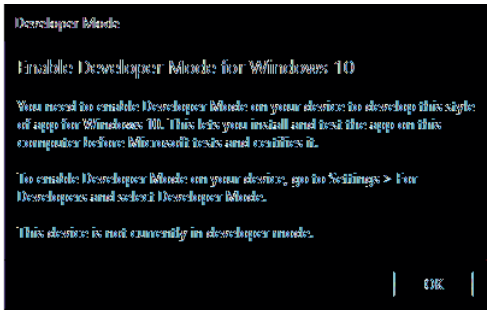
1. Uruchom program Visual Studio 2015, jeśli nie jest on jeszcze uruchomiony.
2. W menu File (Plik) wskaż menu podrzędne New (Nowy), a następnie wybierz z niego polecenie Project (Projekt).

   Spowoduje to otwarcie okna dialogowego New Project (Nowy projekt).
3. Rozwiń węzeł Installed (jeśli nie jest on już rozwinięty), rozwiń węzeł Templates (Szablony), rozwiń folder Visual C#, rozwiń element Windows, a następnie kliknij Universal.
4. Kliknij znajdującą się w środkowym panelu ikonę Blank App (Windows Universal) (Pusta aplikacja (aplikacja uniwersalna)).

> **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.

5. Upewnij się, że ścieżka podana w polu Location (Lokalizacja) wskazuje katalog \Microsoft Press\VCSBS\Chapter 1 w folderze Dokumenty.
6. Wpisz w polu Name (Nazwa) tekst **Hello**.
7. Upewnij się, że zaznaczone zostało pole wyboru opcji Create Directory For Solution (Utwórz katalog dla rozwiązania), a następnie kliknij przycisk OK.

   Jeśli aplikacja UWP jest tworzona po raz pierwszy, to w tym miejscu może pojawić się monit o włączenie trybu dewelopera dla systemu Windows 10. Możliwość włączenia trybu dewelopera za pośrednictwem interfejsu użytkownika zależy od urządzenia oraz wersji systemu Windows 10. Wskazówki dotyczące włączania trybu dewelopera znaleźć można w artykule „Enable your device for development" dostępnym (w jęz. angielskim) pod adresem: *https://msdn.microsoft.com/library/windows/apps/xaml/dn706236.aspx*.



Uwaga To okno dialogowe może pojawić się podczas tworzenia nowej aplikacji UWP lub pierwszej próby uruchomienia aplikacji UWP w Visual Studio.

8. Po utworzeniu aplikacji przejrzyj zawartość okna Solution Explorer (Eksplorator rozwiązań).

   Nazwa użytego szablonu aplikacji jest nieco mylącą – wprawdzie Blank App oznacza „pustą aplikację", ale faktycznie szablon ten dostarcza kilku 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.

9. Korzystając z okna 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 (ilustracja na stronie następnej).

   W górnej części wyświetlany jest widok graficzny domyślnie odzwierciedlający wygląd 5-calowego ekranu telefonu, a w dolnej części znajduje się opis zawartości tego ekranu, zapisany w języku XAML. Język XAML jest podobny do języka XML i jest używany przez aplikacje UWP oraz aplikacje WPF do definiowania graficznego układu formularzy oraz ich zawartości. Dla osób znających język XML 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, to można zamknąć okna Output (Dane wyjściowe) oraz Error List (Lista błędów).

Uwaga Zanim przejdziemy dalej, warto wyjaśnić pewne kwestie związane z używaną terminologią. W typowych 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 będziemy nazywać po prostu ogólnym terminem *formularz*. Słowo *okno* nadal jednak będzie używane, ilekroć mowa będzie o elementach środowiska IDE oprogramowania Visual Studio 2015, 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 kontrolek w języku C#, wygenerowanym automatycznie przez program Visual Studio 2015.

➜ 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.
2. Rozwiń sekcję o nazwie Common XAML Controls (Typowe kontrolki XAML).

   W sekcji tej znajduje się lista kontrolek używanych przez większość aplikacji graficznych.

Wskazówka Bardziej obszerną listę kontrolek 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 Należy koniecznie upewnić się, że zaznaczona została kontrolka *TextBlock* (Blok tekstu), a nie kontrolka *TextBox* (Pole tekstowe). Jeśli na formularzu została niechcący umieszczona niewłaściwa kontrolka, to można ją łatwo usunąć poprzez kliknięcie tej kontrolki i wciśnięcie klawisza 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 2015 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 okolicę górnego, lewego narożnika formularza. (W tym konkretnym przypadku dokładne umiejscowienie tej kontrolki

nie jest ważne). Należy pamiętać, że przesunięcie tej 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, określany 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 w języku 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 w języku 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*.

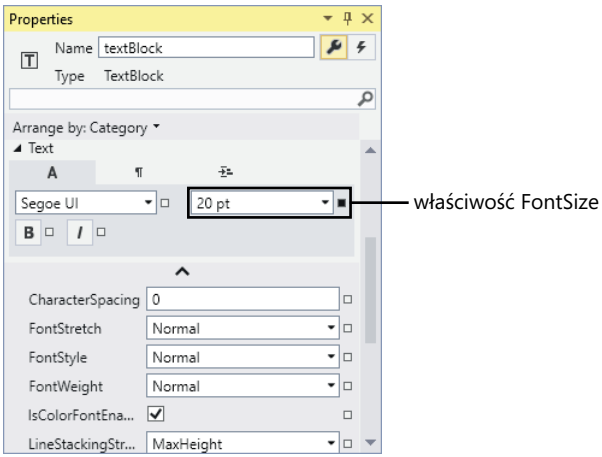
5. Wybierz z menu View (Widok) polecenie Properties Window (Okno właściwości).

Polecenie to spowoduje wyświetlenie okna Properties (Właściwości) w dolnej prawej części ekranu, poniżej panelu Solution Explorer (Eksplorator rozwiązań), o ile okno to nie było wyświetlane już wcześniej. Właściwości kontrolek 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 zależna jest 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.

   Przywróć poprzednią 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.

Należy pamiętać, 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.

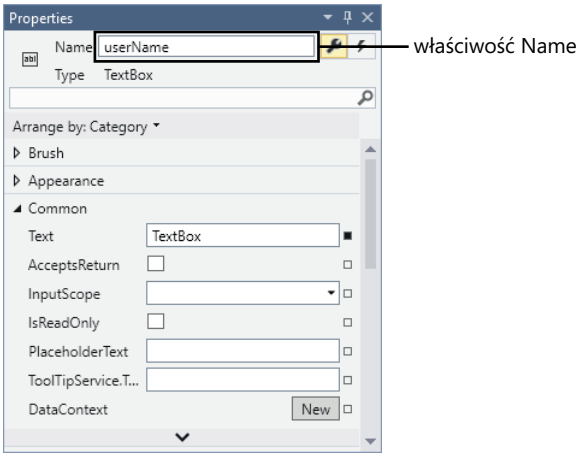
12. Kliknij znajdującą się w panelu Toolbox kontrolkę *TextBox* (Pole tekstowe) i przeciągnij ją na formularz. Przesuń kontrolkę pola tekstowego tak, by znajdowała się bezpośrednio pod kontrolką *TextBlock*.

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 kontrolek.

---

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 znajdującej 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* (Nazwa), wyświetlanej w górnej części okna Properties, z *textBox* na **userName** (nazwa użytkownika):

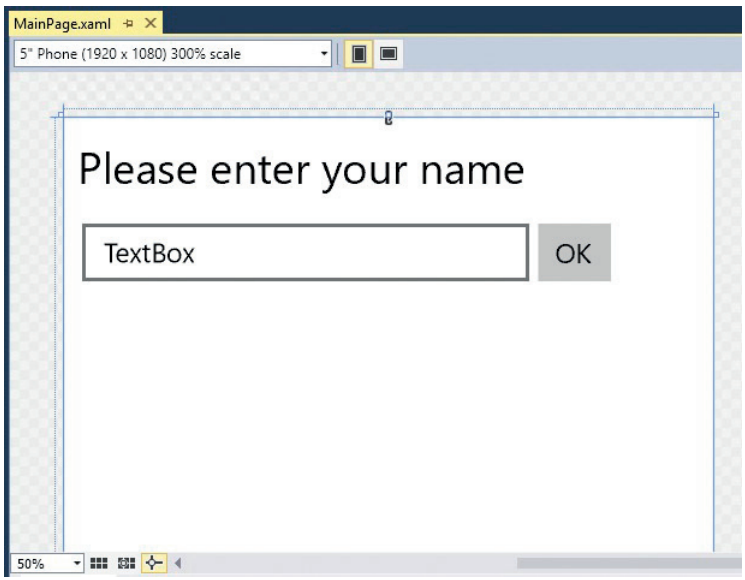


Uwaga Konwencje tworzenia nazw dla kontrolek i zmiennych zostaną omówione w rozdziale 2, zatytułowanym „Zmienne, operatory i wyrażenia".

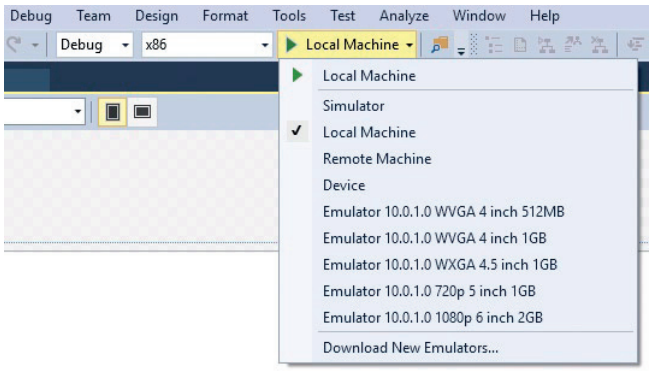
15. Wyświetl ponownie panel Toolbox (Przybornik), a następnie kliknij znajdującą się w tym panelu kontrolkę *Button* (Przycisk) i przeciągnij ją na formularz. Umieść kontrolkę przycisku po prawej stronie kontrolki pola tekstowego (*TextBox*) w taki sposób, by dolna krawędź przycisku była wyrównana w poziomie z dolną krawędzią pola tekstowego.
16. Korzystając z okna Properties zmień dla kontrolki *Button* wartość jej właściwości o nazwie *Name* (Nazwa) 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 poniżej:

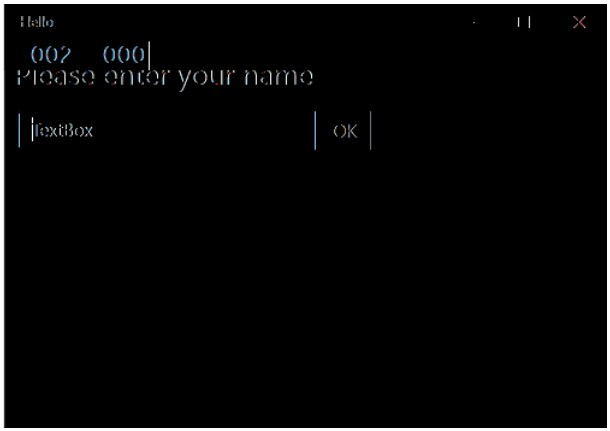
Uwaga Menu rozwijane w lewym górnym rogu okna Design View (Projektuj) umożliwia sprawdzanie, w jaki sposób formularz będzie wyglądał dla różnych rozmiarów i rozdzielczości ekranu. W tym przykładzie wybrany jest domyślny widok 5-calowego ekranu telefonu o rozdzielczości 1920 x 1080. Po prawej stronie 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, ale w tym przykładzie można pozostawić widok 5-calowego ekranu telefonu.



17. Wybierz z menu Build (Kompilowanie) polecenie Build Solution (Kompiluj rozwiązanie) i upewnij się, czy proces budowy przebiegł 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 (Debugowanie) wybierz opcję Start Debugging (Rozpocznij debugowanie).



Powinno to spowodować uruchomienie aplikacji i wyświetlenie przez nią przygotowanego formularza. Formularz będzie wyglądać 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 określać, co ma się stać po kliknięciu przez użytkownika przycisku OK, co uczynimy w następnym kroku.

19. Powróć do programu Visual Studio 2015 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 2015 wygenerował dla nas całkiem sporo kodu zajmującego się obsługą rutynowych działań, które muszą być wykonywane przez każdą aplikację

graficzną, takich jak np. 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 Sklepu Windows

W panelu Solution Explorer (Eksplorator rozwiązań) 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;

// The Blank Page item template is documented at
// http://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409
// Szablon elementu Blank Page jest udokumentowany pod adresem
// http://go.microsoft.com/fwlink/?LinkId=234238 */

namespace Hello
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// 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 (*constructor*) tej klasy i zawierająca wywołanie metody o nazwie *InitializeComponent* (Zainicjuj składnik). Konstruktor to specjalna metoda, która ma taką samą nazwę jak klasa, której dotyczy. Metoda ta jest wykonywana podczas tworzenia nowego egzemplarza danej klasy i może zawierać kod służący do inicjowania nowej instancji klasy. Więcej informacji na temat konstruktorów podanych zostanie w rozdziale 7.

Faktycznie klasa *MainPage* zawiera znacznie więcej kodu niż te kilka linijek widocznych w pliku MainPage.xaml.cs, ale większość tego kodu jest generowana automatycznie w oparciu o opis formularza w języku XAML i pozostaje dla nas 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 zapewne pamiętamy, w aplikacjach konsolowych to właśnie metoda *Main* definiuje punkt, od którego rozpoczyna się wykonywanie aplikacji. Aplikacje graficzne różnią się nieco pod tym względem.

W panelu Solution Explorer (Eksplorator rozwiązań) 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;
```

```
// The Blank Application template is documented at
// http://go.microsoft.com/fwlink/?LinkId=402347&clcid=0x409

// Szablon elementu Pusta strona jest udokumentowany na stronie
// http://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409

namespace Hello
{
    /// <summary>
    /// Provides application-specific behavior to supplement
    /// the default Application class.

    /// Zapewnia zachowanie specyficzne dla aplikacji,
    /// aby uzupełnić domyślną klasę aplikacji.
    /// </summary>
    sealed partial class App : Application
    {
        /// <summary>
        /// Allows tracking page views, exceptions and other telemetry
        /// through the Microsoft Application Insights service.
        /// </summary>
        public static Microsoft.ApplicationInsights.TelemetryClient TelemetryClient;

        /// <summary>
        /// Initializes the singleton application object.
        /// This is the first line of authored code
        /// executed, and as such is the logical equivalent of main() or WinMain().

        /// Inicjuje pojedynczy obiekt aplikacji.
        /// Jest to pierwszy wiersz napisanego kodu
        /// wykonywanego i jest logicznym odpowiednikiem metod main() lub WinMain().
        /// </summary>
        public App()
        {
                 Microsoft.ApplicationInsights.WindowsAppInitializer.InitialzeAsync(
                 Microsoft.ApplicationInsights.WindowsCollectors.Metadata |
                 Microsoft.ApplicationInsights.WindowsCollectors.Session);
            this.InitializeComponent();
            this.Suspending += OnSuspending;
        }

        /// <summary>
        /// Invoked when the application is launched normally by the end user.
        /// Other entry points
        /// will be used such as when the application is launched
        /// to open a specific file.

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

        /// <param name="e">Details about the launch request and process.
        ///(Szczegóły dotyczące żądania uruchomienia i procesu)</param>
```

```
        protected override void OnLaunched(LaunchActivatedEventArgs e)
        {
#if DEBUG
            if (System.Diagnostics.Debugger.IsAttached)
            {
                this.DebugSettings.EnableFrameRateCounter = true;
            }
#endif

            Frame rootFrame = Window.Current.Content as Frame;

            // Do not repeat app initialization when the Window already has content,
            // just ensure that the window is active

            // Nie powtarzaj inicjowania aplikacji,
            // gdy w oknie znajduje się już zawartość,
            // upewnij się tylko, że okno jest aktywne.
            if (rootFrame == null)
            {
                // Create a Frame to act as the navigation context
                // and navigate to the first page

                // 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: Load state from previously suspended application
                    //(Załaduj stan z wstrzymanej wcześniej aplikacji)
                }

                // Place the frame in the current Window
                // Umieść ramkę w bieżącym oknie
                Window.Current.Content = rootFrame;
            }
            if (rootFrame.Content == null)
            {

                // When the navigation stack isn't restored navigate to the
                // first page, configuring the new page by passing required
                // information as a navigation parameter

                // 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);
            }
            // Ensure the current window is active
            // Upewnij się, że bieżące okno jest aktywne
            Window.Current.Activate();
        }
```

```
        /// <summary>
        /// Invoked when Navigation to a certain page fails
        /// Wywoływane, gdy nawigacja do konkretnej strony nie powiedzie się
        /// </summary>

        /// <param name="sender">The Frame which failed navigation
        /// (Ramka, do której nawigacja nie powiodła się)</param>
        /// <param name="e">Details about the navigation failure
        /// (Szczegóły dotyczące niepowodzenia nawigacji)</param>
        void OnNavigationFailed(object sender, NavigationFailedEventArgs e)
        {
            throw new Exception("Failed to load Page " + e.SourcePageType.FullName);
        }

        /// <summary>
        /// Invoked when application execution is being suspended.
        /// Application state is saved without knowing whether the application will
        /// be terminated or resumed with the contents of memory still intact.

        /// 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">The source of the suspend request
        /// (Źródło żądania wstrzymania).</param>
        /// <param name="e">Details about the suspend request
        /// (Szczegóły żądania wstrzymania).</param>
        private void OnSuspending(object sender, SuspendingEventArgs e)
        {
            var deferral = e.SuspendingOperation.GetDeferral();
            //TODO: Save application state and stop any background activity
            //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 oraz składnię użytych w nim instrukcji. Wystarczy nam po prostu wiedzieć, że to właśnie w ten sposób następuje wyświetlenie formularza aplikacji po jej uruchomieniu.

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

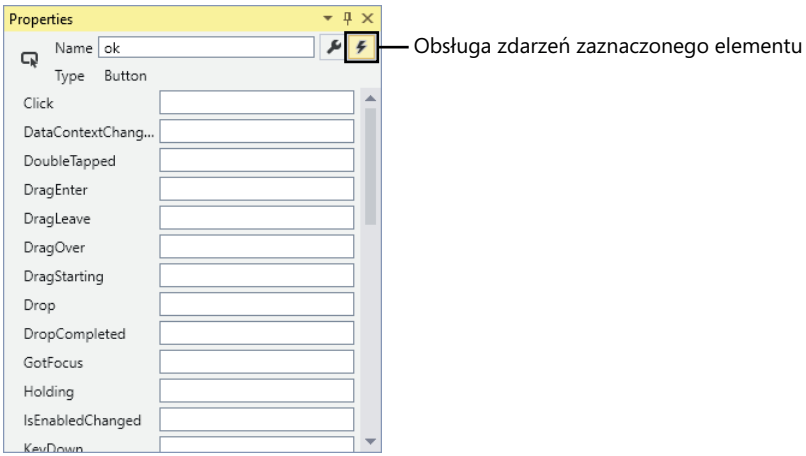
Dodawanie kodu do aplikacji graficznej

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 przykładać 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 za pomocą pogrubionej czcionki:

```
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 2015 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* (Przycisk), 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. Powróć do programu Visual Studio 2015 i wybierz z menu Debug polecenie Stop Debugging.

Podsumowanie

W tym rozdziale pokazaliśmy, jak za pomocą programu Visual Studio 2015 można tworzyć, budować i uruchamiać aplikacje graficzne. 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 2015 i przejdź do rozdziału 2.
- Jeśli chcesz na tym zakończyć pracę z programem Visual Studio 2015, wybierz z menu File (Plik) 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 2015	W menu File (Plik) wskaż menu podrzędne New (Nowy), a następnie kliknij polecenie Project (Projekt), co spowoduje otwarcie okna dialogowego New Project (Nowy projekt). W lewym panelu rozwiń Installed (Zainstalowane), rozwiń Templates (Szablony), a następnie kliknij element Visual C#. W środkowym panelu kliknij ikonę Console Application (Aplikacja konsolowa). Określ w polu Location (Lokalizacja) katalog, w którym należy umieścić pliki projektu. Wpisz nazwę projektu i kliknij przycisk OK.
Utworzenia przy użyciu programu Visual Studio 2015 nowej aplikacji UWP	W menu File (Plik) wskaż menu podrzędne New (Nowy), a następnie kliknij polecenie Project (Projekt), co spowoduje otwarcie okna dialogowego New Project (Nowy projekt). W lewym panelu rozwiń Installed (Zainstalowane), rozwiń Templates (Szablony), 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 (Lokalizacja) katalog, w którym należy umieścić pliki projektu. Wpisz nazwę projektu i kliknij przycisk OK.
Zbudowania aplikacji	Wybierz z menu Build (Kompilowanie) polecenie Build Solution (Kompiluj rozwiązanie).
Uruchomienia aplikacji w trybie debugowania	Wybierz z menu Debug (Debugowanie) polecenie Start Debugging (Rozpocznij debugowanie).
Uruchomienia aplikacji bez debugowania	Wybierz z menu Debug polecenie Start Without Debugging (Uruchom bez debugowania).

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, zatytułowanym „Wprowadzenie do języka C#" pokazaliśmy, w jaki sposób, korzystając ze środowiska programowania Microsoft Visual Studio 2015, 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, zatytułowanym „Tworzenie metod i stosowanie zasięgów 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 określonemu 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 sposobu 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 przestrzenie 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 sami 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 *słowami 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#:

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

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.

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

Wskazówka Podczas wpisywania słów kluczowych w oknie edytora kodu i tekstu są one oznaczane przez program Visual Studio 2015 kolorem niebieskim.

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 nazewniczą, 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 nazewniczej 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 wielbłądzią – ang. *camelCase*).

- Nie należy stosować tzw. notacji węgierskiej. (Notacja ta jest prawdopodobnie dobrze znana programistom używającym języka Microsoft Visual C++. Pozostałe osoby, które nie znają notacji węgierskiej, nie mają 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 mogą być przechowywane i przetwarzane – 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 tzw. 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 demonstruje 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 2015 powoduje wyświetlenie podpowiedzi informującej o typie danej zmiennej.

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

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 2015 nie został uruchomiony już wcześniej, to uruchom go teraz.
2. Wskaż w menu File (Plik) menu podrzędne Open (Otwórz), a następnie wybierz z niego polecenie Project/Solution (Projekt/Rozwiązanie).

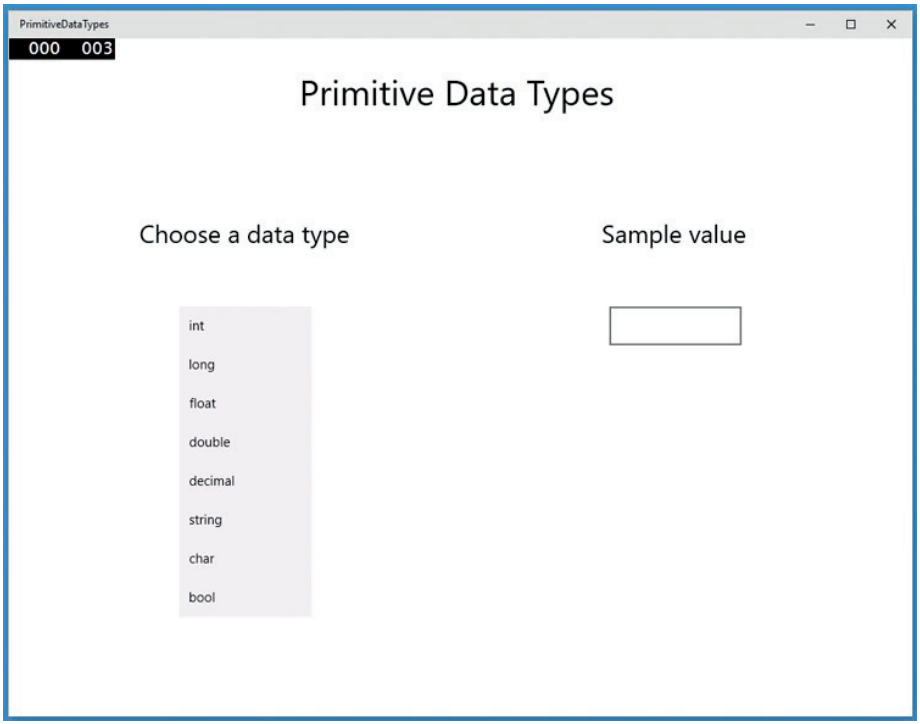
   Spowoduje to otwarcie okna dialogowego Open Project (Otwórz projekt).
3. Przejdź do katalogu \Microsoft Press\VCSBS\Chapter 2\PrimitiveDataTypes w folderze Dokumenty.
4. Zaznacz plik rozwiązania *PrimitiveDataTypes* i naciśnij przycisk Open (Otwórz).

   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 2015 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 (Debugowanie) polecenie Start Without Debugging (Uruchom bez debugowania).

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* (łańcuch znakowy).

   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 (Przykładowa wartość) 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 2015 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 wybranie żądanego rozmiaru z rozwijanej listy skali powiększenia, znajdującej się w dolnym lewym rogu okna widoku projektowego.

2. 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.

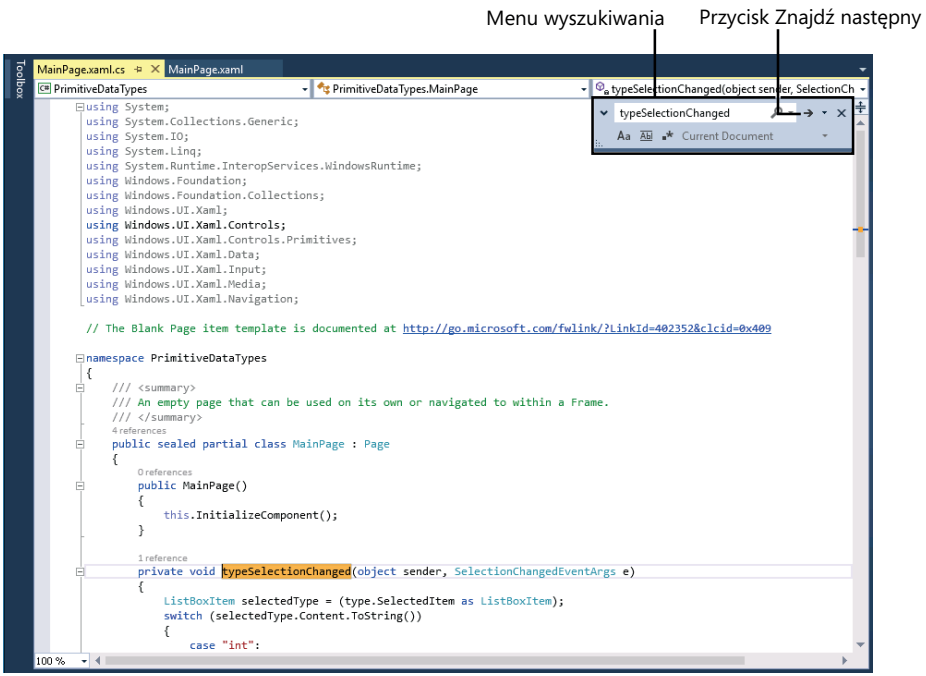
3. Wybierz z menu View (Widok) polecenie Code (Kod).

   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.

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

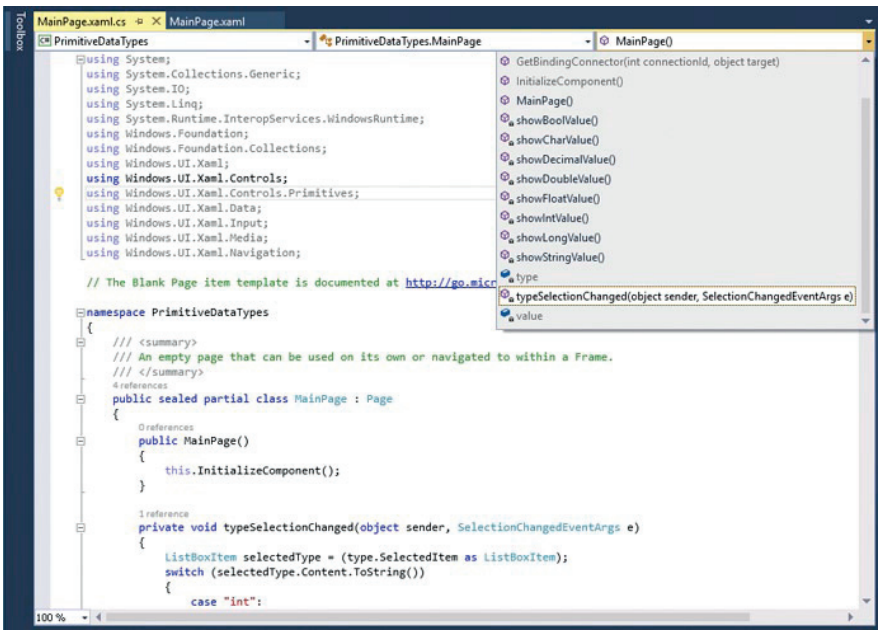
Wskazówka Chcąc zlokalizować określony element projektu, można wskazać w menu Edit (Edycja) menu podrzędne Find and Replace (Znajdź i zamień), a następnie wybrać z niego polecenie Quick Find (Szybkie wyszukiwanie). Spowoduje to otwarcie menu wyszukiwania w górnym prawym narożniku okna edytora kodów i 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 (Szybkie wyszukiwanie). Podobnie, wciśnięcie kombinacji klawiszy Ctrl+H spowoduje otwarcie okna dialogowego Quick Replace (Szybkie zamienianie).

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



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 tzw. określnikiem 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, zatytułowanym „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 (Rozpocznij debugowanie).

Spowoduje to ponowne wyświetlenie formularza.

10. Wybierz z listy Choose A Data Type (Wybierz typ danych) typ danych *int*. Sprawdź, czy w polu tekstowym Sample Value (Wartość przykładowa) 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. Powróć do programu Visual Studio i wybierz z menu Debug polecenie Stop Debugging.

Posługiwanie się operatorami arytmetycznymi

Język C# obsługuje zwykłe operatory matematyczne, o których jako dzieci uczyliśmy 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 *argumentami* (lub *operandami*). W wyrażeniu 750 * 20, znak * jest operatorem, a liczby 750 i 20 są jego argumentami.

Operatory i typy danych

Nie wszystkie operatory można stosować ze wszystkimi typami 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.

Należy również mieć świadomość, że typ danych rezultatu działania operatora arytmetycznego zależny jest 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 przekształci wartość typu *int* do typu *double*. Wynikiem tej operacji będzie więc wartość typu *double* (2.5). Wprawdzie mieszanie

* W językach programowania, a także w języku angielskim część dziesiętną liczb od części całkowitej oddziela się znakiem kropki, a nie znakiem przecinka.

typów danych w jednym wyrażeniu nie jest błędem, ale jest to uznawane za złą praktykę w programowaniu.

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 *okClick* dodaliśmy następujący kod:

```
MessageDialog msg = new MessageDialog(„Hello „ + userName.Text);
```

Interpolacja łańcuchów umożliwia użycie poniższej alternatywnej techniki:

```
MessageDialog msg = new MessageDialog($”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.

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 wartością wyrażenia 7.0 % 2.4 jest 2.2.

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*.

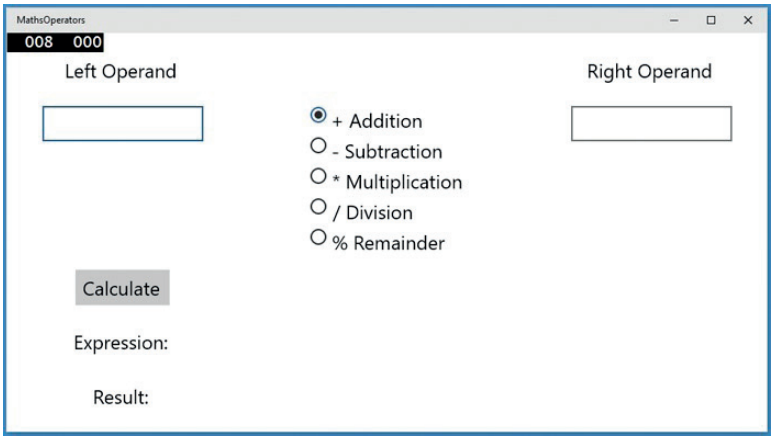
Poznajemy operatory arytmetyczne

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

➜ Uruchamianie projektu MathsOperators

1. Jeśli program Visual Studio 2015 nie został uruchomiony już wcześniej, to uruchom go teraz.
2. Otwórz projekt MathsOperators znajdujący się w katalogu \Microsoft Press\VCSBS\Chapter 2\MathsOperators.
3. Wybierz z menu Debug polecenie Start Debugging.

Zostanie wyświetlony następujący formularz:



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 któregokolwiek 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, zatytułowanym „Obsługa błędów i wyjątków".

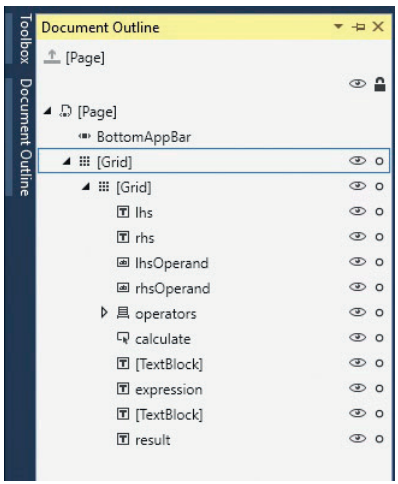
9. Po zakończeniu tych testów wróć do programu Visual Studio 2015 i wybierz z menu Debug polecenie Stop Debugging.

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 (Eksplorator rozwiązań) w gałęzi projektu MathsOperators).
2. W menu View (Widok) 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 kontrolek.



   Okno Document Outline zapewnia możliwość łatwego wyszukiwania i zaznaczania kontrolek, 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 kontrolek, 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 kontrolek 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 kontrolek to *lhsOperand* oraz *rhsOperand*.

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

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: Add rhs to lhs and store the result in outcome
    // 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: Add rhs to lhs and store the result in outcome
    // 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: Subtract rhs from lhs and store the result in outcome
    // 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: Multiply lhs by rhs and store the result in outcome
    // 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: Divide lhs by rhs and store the result in outcome
    // 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: Work out the remainder after dividing lhs by rhs and store
    // the result in outcome
    // 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 (Lewy argument), wpisz liczbę **13** w polu Right Operand (Prawy argument), kliknij opcję + Addition (Dodawanie),

a następnie kliknij przycisk Calculate (Oblicz). W polu Result (Wynik) powinna wówczas zostać wyświetlona wartość 67.

3. Kliknij przycisk Subtraction (Odejmowanie), a następnie kliknij przycisk Calculate. Sprawdź, czy wynik jest teraz równy 41.
4. Kliknij przycisk Multiplication (Mnożenie), a następnie kliknij przycisk Calculate. Sprawdź, czy wynik jest teraz równy 702.
5. Kliknij przycisk Division (Dzielenie), 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 (Reszta), 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.

Kontrolowanie pierwszeństwa

Kolejnością stosowania występujących w wyrażeniu operatorów rządzą reguły *pierwszeństwa* (tzw. priorytet operatorów). Rozważmy następujące wyrażenie, w którym występuje operator + oraz *:

```
2 + 3 * 4
```

Wyrażenie to jest potencjalnie niejednoznaczne; czy najpierw wykonamy dodawanie czy mnożenie? Kolejność wykonywania tych operacji jest ważna, ponieważ ma ona wpływ na wynik:

- Jeśli najpierw wykonamy dodawanie, a potem mnożenie, to rezultat operacji dodawania (2 + 3) stanie się lewym argumentem operatora * i w rezultacie otrzymamy wyrażenie 5 * 4, którego wartością jest liczba 20.
- Jeśli najpierw wykonamy mnożenie, a potem dodawanie to rezultat operacji mnożenia (3 * 4) stanie się prawym argumentem operatora + i w rezultacie otrzymamy wyrażenie 2 + 12, którego wartością jest liczba 14.

W języku C# tzw. operatory multiplikatywne (*, / i %) mają pierwszeństwo przed operatorami addytywnymi (+ i –), a więc w wyrażeniach takich jak 2 + 3 * 4, najpierw

wykonane zostanie mnożenie, a dopiero potem dodawanie. Wartością wyrażenia 2 + 3 * 4 jest więc liczba 14.

Zastosowanie nawiasów pozwala na zmianę domyślnych reguł pierwszeństwa operatorów i wymuszenie innego sposobu wiązania operatorów z argumentami. Przykładowo, w pokazanym poniżej wyrażeniu nawiasy wymuszają powiązanie liczb 2 i 3 z operatorem + (co daje wartość 5), a rezultat tego dodawania tworzy lewy argument operatora *, dając w efekcie wartość 20:

```
(2 + 3) * 4
```

Uwaga W tej książce termin *nawiasy* lub *nawiasy zwykłe* oznacza znaki ( ). Termin *nawiasy klamrowe* oznacza nawiasy { }, a termin *nawiasy kwadratowe* oznacza nawiasy [ ].

Stosowanie zasad łączności przy wyznaczaniu wartości wyrażeń

Zasady pierwszeństwa operatorów to jeszcze nie wszystko. Co się stanie, jeśli w jednym wyrażeniu występować będą różne operatory o takim samym pierwszeństwie? Wówczas do głosu dochodzą zasady łączności. *Zasady łączności* operatorów to kierunek (w lewo lub w prawo), w którym wyznaczane są wartości argumentów operatora. Rozważmy następujące wyrażenie, w którym występuje operator / oraz *:

```
4 / 2 * 6
```

Już na pierwszy rzut oka wyrażenie to jest potencjalnie niejednoznaczne. Czy najpierw należy wykonać dzielenie, czy mnożenie? W tym przypadku obydwa operatory mają takie samo pierwszeństwo (obydwa są operatorami multiplikatywnymi), ale kolejność stosowania tych operatorów do wyznaczenia wartości całego wyrażenia jest ważna, ponieważ możemy otrzymać dwa różne rezultaty:

- Jeśli najpierw wykonamy dzielenie, a potem mnożenie to rezultat operacji dzielenia (4 / 2) stanie się lewym argumentem operatora * i w rezultacie otrzymamy wyrażenie (4 / 2) * 6, którego wartością jest liczba 12.
- Jeśli najpierw wykonamy mnożenie, a potem dzielenie to rezultat operacji mnożenia (2 * 6) stanie się prawym argumentem operatora / i w rezultacie otrzymamy wyrażenie 4 / (2 * 6), którego wartością jest ułamek 4/12.

W tym przypadku kolejność wyznaczania wartości wyrażenia dyktowana jest przez reguły łączności operatorów. Operatory * i / są operatorami łącznymi lewostronnie, co oznacza, że ich argumenty są wyznaczane od lewej do prawej strony. W podanym przykładzie najpierw zostanie wyznaczany rezultat wyrażenia 4/2, który następnie zostanie pomnożony przez 6, dając wynik końcowy 12.