Microsoft

Microsoft® Visual C#® 2010 Krok po kroku

John Sharp

przekład: Maria Chaniewska, Krzysztof Szkudlarek

Microsoft® Visual C#® 2010 Krok po kroku

Authorized translation of English edition of
Microsoft® Visual C#® 2010 Step by Step

APN PROMISE Sp. z o.o., biuro: ul. Kryniczna 2, 03-934 Warszawa
tel. +48 22 35 51 642, fax +48 22 35 51 699
e-mail: mspress@promise.pl

ISBN: 978-83-7541-066-2

Przekład: Maria Chaniewska, Krzysztof Szkudlarek
Redakcja: Marek Włodarz
Korekta: Ewa Swędrowska, Anna Wojdanowicz
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 tworzących aplikacje oparte na platformie Microsoft .NET Framework. Język ten odziedziczył wiele z najlepszych cech języka C++ oraz Microsoft Visual Basic, a tylko kilka spośród 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). Ostatnie wcielenie języka C#, którym jest wersja 4.0, przyniosło kolejne udoskonalenia w zakresie polepszenia możliwości współdziałania z innymi technologiami i językami programowania. Funkcje te obejmują obsługę nazwanych i opcjonalnych argumentów, typ *dynamic*, którego użycie wskazuje, że środowisko uruchomieniowe powinno stosować dla danego obiektu tzw. późne wiązanie (ang. late binding) oraz wariancje, których wprowadzenie rozwiązuje pewne problemy związane ze sposobem definiowania ogólnych interfejsów. Język C# 4.0 wykorzystuje w pełni możliwości najnowszej wersji platformy .NET Framework, która również ma numer wersji 4.0. W wersji tej wprowadzono do platformy .NET Framework wiele dodatkowych funkcji, ale zapewne najważniejsze z nich to klasy i typy danych składające się na bibliotekę równoległego realizowania zadań – TPL (Task Parallel Library). Korzystając z biblioteki TPL można szybko i łatwo tworzyć wysoko skalowalne aplikacje, które będą w pełni wykorzystywać możliwości wielordzeniowych procesorów. Rozszerzona została także obsługa usług webowych oraz platformy WCF (Windows Communication Foundation). Obecnie możliwe jest już tworzenie usług zgodnych z modelem REST, a także takich, które wykorzystują bardziej tradycyjny schemat SOAP.

Dzięki środowisku programowania oferowanemu przez program Microsoft Visual Studio 2010 wszystkie te rozbudowane i zaawansowane funkcje są bardzo łatwe w użyciu, a wiele nowych kreatorów i udoskonaleń programu Visual Studio 2010 może znacząco podnieść wydajność pracy każdego programisty.

Dla kogo przeznaczona jest ta książka

Niniejsza książka zakłada, że jej Czytelnik ma już pewne doświadczenie w programowaniu i pragnie poznać podstawy programowania w języku C# przy użyciu programu Visual Studio 2010 i platformy .NET Framework w wersji 4.0. Dzięki tej książce Czytelnicy będą mieli okazję poznać wszystkie cechy języka C# i nauczyć się wykorzystywać je do tworzenia aplikacji działających w systemie operacyjnym Microsoft Windows. Po przeczytaniu tej książki Czytelnicy powinni dysponować dobrą znajomością języka C# i umieć używać tego języka do tworzenia aplikacji typu WPF (Windows Presentation Foundation), odczytywania danych z baz danych Microsoft SQL Server przy użyciu technologii ADO.NET oraz zapytań LINQ, tworzenia skalowalnych aplikacji opartych na bibliotece TPL oraz do tworzenia usług webowych typu REST i SOAP opartych na platformie WCF.

Określenie najlepszego miejsca, od którego należy rozpocząć lekturę tej książki

Niniejsza książka ma za zadanie ułatwić Czytelnikowi podniesienie 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 poniżej 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, z godnie z opisem podanym w dalszej części, zatytułowanej „Instalowanie przykładowych kodów źródłowych”. **2.** Przeczytaj po kolei wszystkie rozdziały z części I, II i III. **3.** Przeczytaj odpowiednie rozdziały z części IV, V i VI, stosownie do poziomu swojego doświadczenia oraz poziomu swoich potrzeb.
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, z godnie z opisem podanym w dalszej części, zatytułowanej „Instalowanie przykładowych kodów źródłowych”. Zapoznaj się pobieżnie z treścią pierwszych pięciu rozdziałów, aby uzyskać ogólny obraz języka C# oraz możliwości programu Visual Studio 2010, a następnie skoncentruj się na rozdziałach od 6 do 21. **2.** Przeczytaj odpowiednie rozdziały z części IV, V i VI, stosownie do poziomu swojego doświadczenia oraz poziomu swoich potrzeb.

Jeżeli jesteś	Wykonaj następujące kroki
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, z godnie z opisem podanym w dalszej części, zatytułowanej „Instalowanie przykładowych kodów źródłowych”. **2.** Zapoznaj się pobieżnie z treścią pierwszych siedmiu rozdziałów, aby uzyskać ogólny obraz języka C# oraz możliwości programu Visual Studio 2010, a następnie skoncentruj się na rozdziałach od 8 do 21. **3.** Przeczytaj rozdziały z części IV i V, aby dowiedzieć się, w jaki sposób tworzyć aplikacje dla systemu Windows oraz w jaki sposób korzystać z baz danych. **4.** Przeczytaj rozdziały z części VI, aby dowiedzieć się, w jaki sposób tworzyć skalowalne aplikacje oraz usługi webowe.
Programistą znającym język Visual Basic 6 i pragnącym nauczyć się języka C#	**1.** Zainstaluj pliki używane w opisywanych w tej książce ćwiczeniach, z godnie z opisem podanym w dalszej części, zatytułowanej „Instalowanie przykładowych kodów źródłowych”. **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 dla systemu Windows. **4.** Przeczytaj rozdziały z części V, aby dowiedzieć się, w jaki sposób korzystać z danych znajdujących się w bazach danych. **5.** Przeczytaj rozdziały z części VI, aby dowiedzieć się, w jaki sposób tworzyć skalowalne aplikacje oraz usługi webowe. **6.** Przeczytaj krótkie podsumowania, znajdujące się na końcu każdego rozdziału, aby szybko uzyskać potrzebne informacje na temat konkretnych konstrukcji języka C# oraz funkcji programu Visual Studio 2010.
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 podsumowania, znajdujące się na końcu każdego rozdziału, aby szybko zapoznać się z omawianymi w danym rozdziale technikami oraz elementami składni.

Konwencje i cechy charakterystyczne stosowane w tej książce

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

Konwencje

- Każde ćwiczenie składa się z serii zadań. Każde zadanie prezentowane jest jako seria ponumerowanych kroków (1, 2, itd).
- Uwagi oznaczone etykietą „wskazówka" zawierają dodatkowe informacje lub opis alternatywnego sposobu wykonania danego kroku.
- Uwagi oznaczone etykietą „ważne" mają za zadanie zwrócenie uwagi na informacje, które należy sprawdzić przed kontynuowaniem danego ćwiczenia.
- Tekst, który powinien zostać wpisany przez Czytelnika, wyróżniany jest za pomocą pogrubionej 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*.

Inne cechy charakterystyczne

- Występujące w tej książce notatki uzupełniające zawierają bardziej wyczerpujące informacje na temat danego ćwiczenia. Notatki uzupełniające mogą zawierać informacje wyjaśniające podstawy, na których opiera się dane ćwiczenie, wskazówki projektowe lub opis funkcji związanych z omawianymi właśnie informacjami.
- Każdy rozdział kończy się krótkim podsumowaniem. Sekcja „Krótkie podsumowanie rozdziału ..." zawiera krótkie streszczenie przypominające sposób wykonywania zadań omawianych w danym rozdziale.

Wstępna wersja oprogramowania

Przy pisaniu tej książki posługiwałem się programem Visual Studio 2010 w wersji Beta 2. Zamieszczone w tej książce przykłady zostały również przejrzane i przetestowane przy użyciu finalnej wersji tego oprogramowania. Może się jednak zdarzyć, że pomiędzy wersją produkcyjną programu Visual Studio a prezentowanymi w tej książce przykładami, tekstami i zrzutami ekranowymi występować będą niewielkie różnice.

Wymagania sprzętowe i programowe

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

- System operacyjny Microsoft Windows 7 Home Premium, Windows 7 Professional, Windows 7 Enterprise lub Windows 7 Ultimate. Prezentowane ćwiczenia będą również działać w systemie Microsoft Windows Vista z dodatkiem Service Pack 2 lub nowszym.
- Oprogramowanie Microsoft Visual Studio 2010 Standard, Visual Studio 2010 Professional lub Microsoft Visual C# 2010 Express oraz Microsoft Visual Web Developer 2010 Express.
- Oprogramowanie Microsoft SQL Server 2008 Express (oprogramowanie to jest dostarczane razem ze wszystkimi edycjami programów Visual Studio 2010, Visual C# 2010 Express oraz Visual Web Developer 2010 Express).
- Procesor taktowany zegarem o częstotliwości 1,6 GHz lub wyższej. Do przeprowadzenia ćwiczeń z rozdziału 27 i 28 wymagany jest procesor dwurdzeniowy lub z większą liczbą rdzeni.
- 1 GB dostępnej pamięci RAM w przypadku procesora 32 bitowego lub 2 GB w przypadku procesora 64 bitowego.
- Monitor (o rozdzielczości 1024 ×768 lub wyższej) wyświetlający co najmniej 256 kolorów.
- Napęd CD-ROM lub DVD-ROM.
- Mysz lub inne kompatybilne urządzenie wskazujące.

Skonfigurowanie oprogramowania SQL Server 2008 Express Edition wymagać będzie także dostępu do konta z uprawnieniami administracyjnymi.

Korzystanie z obrazu dysku CD

Obraz dysku CD towarzyszącego książce jest dostępny na stronie wydawcy przy opisie książki w zakładce Dodatkowe informacje, pod adresem:

http://www.ksiazki.promise.pl/aspx/produkt.aspx?pid=55291

Na podstawie tego obrazu można wykonać fizyczny dysk CD lub zainstalować go jako napęd wirtualny. Dysk ten zawiera przykładowe kody źródłowe, które będą wykorzystywane w trakcie wykonywania opisywanych ćwiczeń. Korzystanie z tych kodów źródłowych pozwala uniknąć marnowania czasu na tworzenie plików, które są nieistotne dla przebiegu danego ćwiczenia. Znajdujące się na płycie pliki oraz szczegółowe instrukcje zawarte w każdym ćwiczeniu pozwalają na praktyczne poznawanie tajników języka C#, co jest łatwym i efektywnym sposobem nabywania i utrwalania nowych umiejętności.

Instalowanie przykładowych kodów źródłowych

Aby zainstalować na komputerze przykładowe kody źródłowe oraz wymagane oprogramowanie, które umożliwi praktyczne wykonywanie opisywanych ćwiczeń, należy wykonać następujące kroki.

1. Płytę CD umieść w napędzie CD-ROM swojego komputera lub zamontuj obraz jako napęd wirtualny.

Uwaga Po włożeniu płyty do napędu powinno nastąpić automatyczne wyświetlenie treści umowy licencyjnej. Jeśli umowa ta nie została wyświetlona, użyj znajdującej się na pulpicie ikony lub skorzystaj z menu *Start* i otwórz okno *Mój komputer*. Następnie kliknij dwukrotnie ikonę swojego napędu CD-ROM, a potem kliknij dwukrotnie plik *StartCD.exe*.

2. Zapoznaj się z treścią umowy licencyjnej. Jeśli zgadzasz się na jej warunki, zaznacz opcję potwierdzającą Twoją zgodę, następnie kliknij przycisk **Next**.

   Spowoduje to wyświetlenie menu opcji.

3. Kliknij przycisk **Install Code Samples** (Instaluj przykładowe kody źródłowe).

4. Postępuj zgodnie z wyświetlanymi na ekranie instrukcjami.

   Przykładowe kody źródłowe zostaną zainstalowane na komputerze w następującym katalogu:

   Documents\Microsoft Press\Visual CSharp Step By Step

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.

Dla tych Czytelników, którzy chcieliby poznać więcej szczegółów, poniżej zamieszczona została lista wszystkich przykładowych projektów i rozwiązań programu Visual Studio 2010, pogrupowanych według katalogów, w których się one znajdują. W wielu przypadkach dostępne są dwie wersje przykładów: jedna z plikami w stanie początkowym, umożliwiającym samodzielne przeprowadzenie danego ćwiczenia zgodnie z podanym opisem i druga, gotowa, której można używać jako punktu odniesienia. Gotowe wersje projektów znajdują się w folderach o nazwie uzupełnionej słowem „– Complete” (Gotowe).

Projekt	Opis
Chapter 1	
TextHello	Pierwszy projekt w języku C#. Projekt ten demonstruje kolejne kroki procesu tworzenia prostego programu wyświetlającego tekst pozdrowienia.
WPFHello	Projekt wykorzystujący platformę WCF (Windows Presentation Foundation) do wyświetlenia w oknie tekstu pozdrowienia.
Chapter 2	
PrimitiveDataTypes	Projekt demonstrujący sposób deklarowania zmiennych każdego z podstawowych typów danych, sposób przypisywania tym zmiennym wartości oraz sposób wyświetlania wartości zmiennych w oknie.
MathsOperators	Projekt wprowadzający operatory arytmetyczne (+ – * / %).
Chapter 3	
Methods	Projekt polegający na ponownym przeanalizowaniu kodu poprzednich projektów 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 2010.
DailyRate Using Optional Parameters	Projekt demonstrujący sposób definiowana 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 znajdującym się na formularzu polu tekstowym.
DoStatement	Projekt wykorzystujący instrukcję *do* do przekształcenia liczby dziesiętnej na jej reprezentację ósemkową.

Projekt	Opis
Chapter 6	
MathsOperators	Projekt pokazujący na przykładzie projektu *MathsOperators* z rozdziału 2, zatytułowanego „Zmienne, operatory i wyrażenia", jak różne nieobsłużone wyjątki mogą doprowadzić do przerwania działania programu. Stabilność działania aplikacji zostaje poprawiona poprzez użycie słów kluczowych *try* i *catch*.
Chapter 7	
Classes	Projekt demonstrujący podstawy definiowana 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 Using Arrays	Projekt demonstrujący zastosowanie tablic do zamodelowania puli kart w grze karcianej.
Cards Using Collections	Projekt demonstrujący sposób przekonstruowania programu do gry w karty tak, by zamiast tablic używane były kolekcje.
Chapter 11	
ParamsArrays	Projekt demonstrujący użycie słowa kluczowego *params* do tworzenia pojedynczej metody mogącej akceptować dowolną liczbę argumentów typu *int*.
Chapter 12	
Vehicles	Projekt wykorzystujący interfejsy do utworzenia prostej hierarchii klas pojazdów. Projekt ten demonstruje także sposób definiowania metod wirtualnych.
ExtensionMethod	Projekt demonstrujący sposób tworzenia metody rozszerzającej dla typu *int*, której zadaniem będzie przekształcanie dziesiętnej liczby całkowitej w jej reprezentację w innym systemie liczenia.
Chapter 13	
Drawing Using Interfaces	Projekt implementujący część pakietu graficznego. Projekt ten wykorzystuje interfejsy do zdefiniowania metod udostępnianych i implementowanych przez obiekty reprezentujące różne figury geometryczne.
Drawing	Projekt rozszerzający projekt *Drawing Using Interfaces* poprzez utworzenie klas abstrakcyjnych, oferujących funkcjonalność wspólną dla różnych figur geometrycznych.

Projekt	Opis
Chapter 14	
UsingStatement	Projekt pokazujący na przykładzie niewielkiego fragmentu kodu z rozdziału 5, zatytułowanego „ Złożone instrukcje przypisania oraz instrukcje iteracji", że kod ten nie jest przystosowany do wykonywania go w środowisku wielowątkowym. Projekt ten pokazuje również, jak za pomocą instrukcji *using* przystosować ten kod do pracy w środowisku wielowątkowym.
Chapter 15	
WindowProperties	Projekt demonstrujący prostą aplikację dla systemu Windows, wykorzystującą kilka właściwości do wyświetlania rozmiarów swojego okna. Zmiana wielkości okna przez użytkownika, powoduje automatyczne aktualizowanie wyświetlanych wartości.
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	
Clock Using Delegates	Projekt wyświetlający zegar, który oprócz czasu lokalnego pokazuje również aktualną godzinę w Londynie, Nowym Jorku i Tokio. Do rozpoczęcia i zakończenia wyświetlania przez zegar aktualnej godziny wykorzystywane są tzw. delegacje.
Clock Using Events	Jest to wersja aplikacji wyświetlającej zegar dla różnych stref czasowych, która do rozpoczęcia i zakończenia wyświetlania przez ten zegar aktualnej godziny wykorzystuje zdarzenia.
Chapter 18	
BinaryTree	Rozwiązanie demonstrujące zastosowanie tzw. typów ogólnych do utworzenia struktury mogącej przechowywać elementy dowolnego typu.
BuildTree	Projekt demonstrujący zastosowanie typów ogólnych do zaimplementowania metod akceptujących parametry dowolnego typu.
BinaryTreeTest	Projekt będący testem współdziałania, polegającym na utworzeniu instancji obiektów typu *Tree*, zdefiniowanego w projekcie *BinaryTree*.
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*.

Projekt	Opis
Chapter 19 (cd.) EnumeratorTest	Projekt będący testem współdziałania, polegającym na przetestowaniu działania iteratora oraz modułu wyliczającego (enumeratora) dla ogólnej klasy *Tree*.
Chapter 20 QueryBinaryTree	Projekt demonstrujący sposób pobierania danych z obiektu typu drzewo binarne przy użyciu zapytań w języku LINQ.
Chapter 21 ComplexNumbers	Projekt definiujący nowy typ danych, modelujący liczby złożone i implementujący kilka typowych operatorów używanych dla tego typu danych.
Chapter 22 BellRingers	Projekt będący aplikacją typu WPF (Windows Presentation Foundation), demonstrującą sposób definiowania stylów oraz korzystania z podstawowych kontrolek WPF.
Chapter 23 BellRingers	Projekt będący rozszerzoną wersją aplikacji utworzonej w rozdziale 22, zatytułowanym „Omówienie platformy Windows Presentation Foundation", w której interfejs użytkownika został wzbogacony o elementy menu rozwijanego oraz menu kontekstowego.
Chapter 24 OrderTickets	Projekt demonstrujący na przykładzie zamówienia klienta sposób implementowania reguł biznesowych, służących do sprawdzania poprawności danych wprowadzanych przez użytkownika do aplikacji WCF.
Chapter 25 ReportOrders	Projekt pokazujący sposób korzystania z baz danych przy użyciu kodu ADO.NET. Tworzona w tym projekcie aplikacja pobierać będzie dane z tabeli *Orders* (Zamówienia) przykładowej bazy danych *Northwind*.
LINQOrders	Projekt demonstrujący sposób korzystania z baz danych przy użyciu zapytań LINQ tłumaczonych na zapytania w języku SQL (ang. LINQ to SQL). Tworzona w tym projekcie aplikacja pobierać będzie dane z tabeli *Orders* (Zamówienia) przykładowej bazy danych *Northwind*.
Chapter 26 Suppliers	Projekt demonstrujący zastosowanie w aplikacji WPF tzw. wiązania danych do formatowania i wyświetlania za pomocą kontrolek umieszczonych na formularzu WPF danych pobieranych z bazy danych. Ta aplikacja umożliwiać będzie także modyfikowanie danych z tabeli *Products* (Produkty) przykładowej bazy danych *Northwind*.
Chapter 27 GraphDemo	Projekt generujący skomplikowany wykres i wyświetlający go na formularzu WPF. W tym projekcie niezbędne obliczenia wykonywane są przez jeden wątek.

Projekt	Opis
GraphDemo Using Tasks	Wersja projektu *GraphDemo*, która tworzy kilka zadań do równoległego wykonywania obliczeń potrzebnych do narysowania wykresu.
GraphDemo Using Tasks that Return Results	Jest to rozszerzona wersja projektu *GraphDemo Using Tasks*, demonstrująca sposób zwracania wartości z poziomu zadania.
GraphDemo Using the Parallel Class	Jest to wersja projektu *GraphDemo*, w którym do wyodrębnienia procesu tworzenia i zarządzania zadaniami użyta została klasa *Parallel*.
GraphDemo Canceling Tasks	Projekt pokazujący, jak zaimplementować możliwość zatrzymywania zadań w kontrolowany sposób, zanim same zakończą swoją działanie.
ParallelLoop	Przykładowa aplikacja pokazująca, kiedy nie należy używać klasy *Parallel* do tworzenia i uruchamia kilku zadań równocześnie.
Chapter 28 CalculatePI	Projekt wykorzystujący algorytm statycznego próbkowania do obliczenia przybliżonej wartości liczby PI. Projekt ten wykorzystuje zadania równoległe.
PLINQ	Projekt demonstrujący kilka przykładów wykorzystywania technologii PLINQ do pobierania danych przy użyciu zadań równoległych.
Chapter 29 ProductInformationService	Projekt implementujący usługę webową typu SOAP utworzoną przy użyciu platformy WCF. Utworzona w tym projekcie usługa webowa udostępnia usługę zwracającą informacje o cenach produktów zarejestrowanych w przykładowej bazie danych *Northwind*.
ProductDetailsService	Projekt implementujący usługę webową typu REST, utworzoną przy użyciu platformy WCF. Utworzona w tym projekcie usługa webowa zwraca szczegółowe informacje na temat wskazanego produktu z przykładowej bazy danych *Northwind*.
ProductDetailsContracts	Projekt zawierający kontrakty danych oraz kontrakty usługi implementowane przez usługę webową *ProductDetailsService*.
ProductClient	Projekt ilustrujący sposób tworzenia aplikacji WPF korzystającej z usług webowych. Projekt ten pokazuje, w jaki sposób wywoływać metody webowe usług webowych *ProductInformationService* i *ProductDetailsService*.

Odinstalowanie przykładowych kodów źródłowych

W celu usunięcia z komputera przykładowych kodów źródłowych należy wykonać następujące kroki.

1. Kliknij łącze *Odinstaluj program*, znajdujące się w *Panelu sterowania* w grupie *Programy*.

2. Zaznacz na liście aktualnie zainstalowanych programów pozycję *Microsoft Visual C# 2010 Step By Step*.
3. Kliknij przycisk *Odinstaluj*.
4. Postępuj zgodnie z wyświetlanymi instrukcjami, aby usunąć z komputera przykładowe kody źródłowe.

Wyszukiwanie dodatkowych treści w trybie Online

W miarę pojawiania się nowych lub uaktualnionych materiałów uzupełniających treść tej książki, będą one zamieszczane w sieci, na internetowej witrynie wydawnictwa Microsoft Press, Online Developer Tools. Wśród materiałów, jakie będzie można znaleźć na tej stronie, znajdować się będą aktualizacje treści tej książki, artykuły, łącza do treści uzupełniających, errata, przykładowe rozdziały oraz wiele innych. Wymieniona witryna webowa jest stale aktualizowana i jest dostępna pod adresem *www.microsoft.com/learning/books/online/developer*.

Wsparcie techniczne dla tej książki

Wydawca zapewnia, że dołożył wszelkich starań, aby zapewnić jak największą dokładność podawanych tej książce informacji oraz towarzyszącej jej płyty CD. W miarę gromadzenia się ewentualnych korekt lub zmian będą one publikowane w artykułach z bazy wiedzy firmy Microsoft (Microsoft Knowledge Base).

Wydawnictwo Microsoft Press oferuje pomoc techniczną związaną z tą książką oraz z towarzyszącą jej płytą CD, na następujące stronie internetowej:

http://www.microsoft.com/learning/support/books/.

Pytania i komentarze

Wszelkie komentarze, pytania lub pomysły dotyczące tej książki lub towarzyszącej jej płyty CD, a także pytania, na które nie udało się znaleźć odpowiedzi na wymienionej wcześniej witrynie webowej, prosimy kierować za pomocą poczty elektronicznej do wydawnictwa Microsoft Press na adres:

mspinput@microsoft.com.

Prosimy pamiętać, że pod powyższym adresem nie jest oferowana pomoc techniczna dotycząca oprogramowania firmy Microsoft.

CZĘŚĆ I

Wprowadzenie do języka Microsoft Visual C# oraz Microsoft Visual Studio 2010

ROZDZIAŁ 1

Wprowadzenie do języka C#

Po ukończeniu tego rozdziału Czytelnik będzie potrafił:

- korzystać ze środowiska programowania Microsoft Visual Studio 2010,
- utworzyć aplikację konsolową w języku C#,
- wyjaśnić cel stosowania przestrzeni nazw,
- utworzyć prostą aplikację graficzną w języku C#.

Microsoft Visual C# to opracowany przez firmę Microsoft zorientowany obiektowo język programowania o bardzo dużych możliwościach. Język C# odgrywa istotną rolę w architekturze platformy Microsoft .NET Framework i niektóre osoby przyrównują jego rolę do tej, jaką w tworzeniu systemu UNIX odegrał język C. Osobom znającym już takie języki programowania jak C, C++ lub Java składnia języka C# powinna wydać się znajoma. Osoby programujące w innych językach programowania również powinny szybko przyswoić sobie składnię i specyfikę języka C#; wymaga to jedynie opanowania zasad umieszczania we właściwych miejscach znaków nawiasów oraz średników. Mam nadzieję, że ta książka będzie właściwą pozycją, która pomoże tym osobom w poznaniu języka C#!

W części I przedstawione zostaną podstawy języka C#. Pokażemy w niej, w jaki sposób deklarować zmienne oraz w jaki sposób manipulować wartościami tych zmiennych przy użyciu operatorów arytmetycznych, takich jak np. plus (+) lub minus (–). W części tej pokażemy również, jak tworzyć metody i przekazywać do nich argumenty. Pokażemy także sposób korzystania z instrukcji wyboru, takich jak np. *if* oraz z instrukcji iteracji, takich jak np. *while*. Na zakończenie, omówiony zostanie sposób wykorzystywania wyjątków przez język C# do eleganckiej i łatwej w użyciu obsługi błędów. Wymienione zagadnienia składają się na zasadniczą część języka C#, stanowiąc solidną podstawę umożliwiającą przejście do omawiania bardziej zaawansowanych funkcji, które zostaną przedstawione w częściach od II do VI.

Rozpoczynamy programowanie przy użyciu środowiska Visual Studio 2010

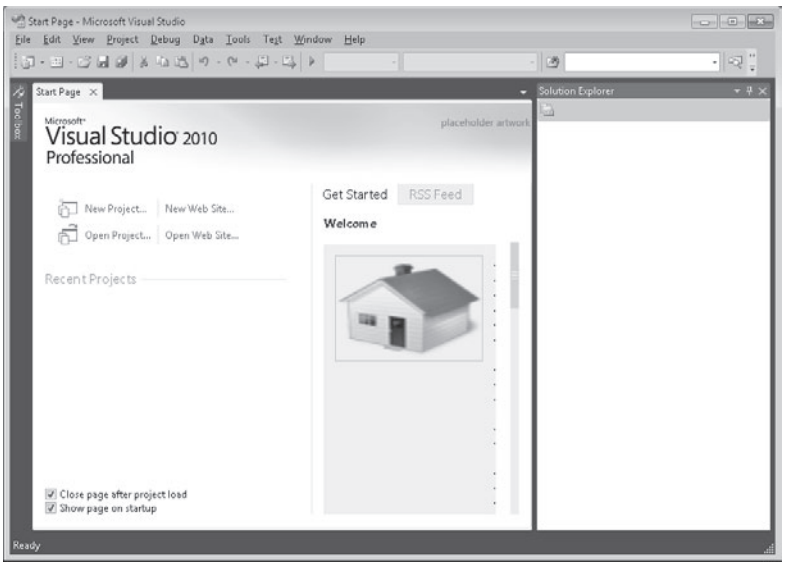
Visual Studio 2010 to rozbudowane środowiska programowania, oferujące funkcjonalność potrzebną przy tworzeniu zarówno małych jak i dużych projektów w języku C#. Możliwe jest nawet konstruowanie projektów, które w gładki i bezproblemowy sposób łączą ze sobą moduły napisane przy użyciu różnych języków programowania, takich jak np. C++, Visual Basic lub F#. Nasze pierwsze ćwiczenie polegać będzie na uruchomieniu środowiska programowania Visual Studio 2010 i utworzeniu aplikacji konsolowej.

UWAGA Aplikacja konsolowa to aplikacja, która nie oferuje graficznego interfejsu użytkownika, lecz jest uruchamiana w oknie wiersza polecenia.

Tworzenie aplikacji konsolowej przy użyciu Visual Studio 2010

- Jeśli używasz programu Visual Studio 2010 Standard lub Visual Studio 2010 Professional, w celu uruchomienia środowiska Visual Studio 2010 wykonaj następujące czynności:
  1. Kliknij przycisk *Start*, znajdujący się na pasku zadań Microsoft Windows, wskaż menu *Wszystkie programy*, a następnie grupę programu *Microsoft Visual Studio 2010*.
  2. Kliknij element *Microsoft Visual Studio 2010* znajdujący się w grupie programu *Microsoft Visual Studio 2010*.

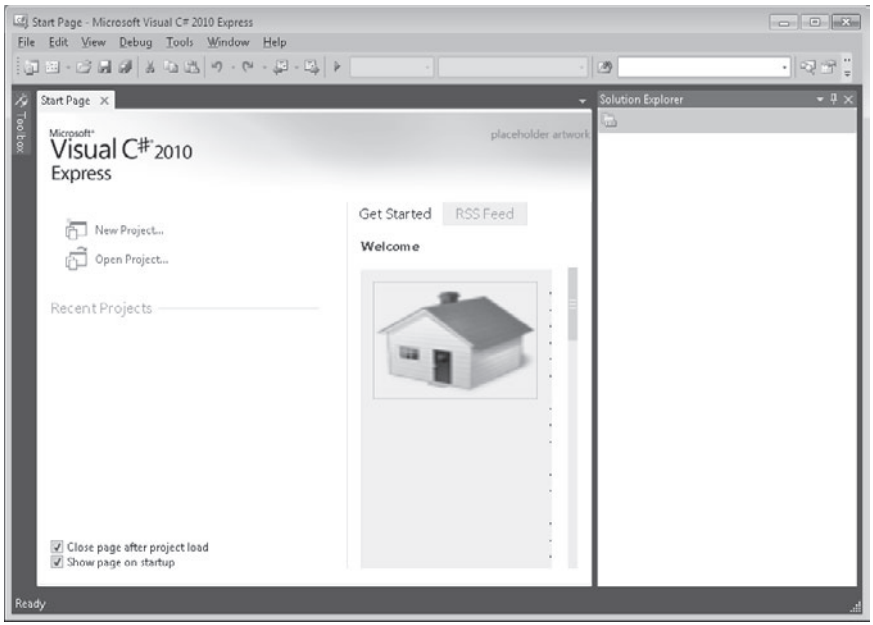
  Po uruchomieniu okno programu Visual Studio 2010 wygląda następująco:



Uwaga W przypadku uruchamiania programu Visual Studio 2010 po raz pierwszy może zostać wyświetlone okno dialogowe umożliwiające wybór domyślnych ustawień środowiska programowania. Program Visual Studio 2010 może dostosować 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# Development Settings* (Ustawienia dla programowania w języku Visual C#), a następnie kliknąć przycisk *Start Visual Studio* (Uruchom Visual Studio). Spowoduje to wyświetlenie po chwili okna środowiska Visual Studio 2010 IDE.

- Jeśli używasz programu Visual C# 2010 Express, kliknij przycisk *Start* znajdujący się na pasku zadań Microsoft Windows, wskaż grupę *Wszystkie programy*, a następnie kliknij element *Microsoft Visual C# 2010 Express*.

  Po uruchomieniu okno programu Visual C# 2010 Express wygląda następująco:

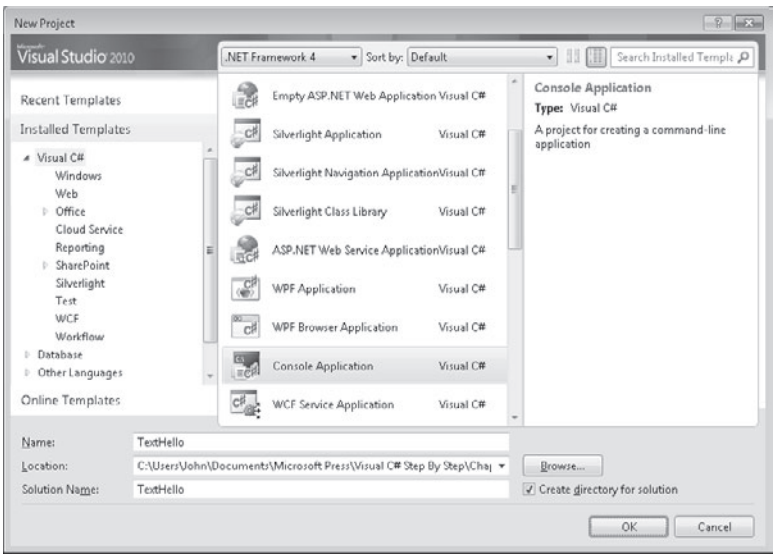


Uwaga W przypadku uruchamiania programu Visual C# 2010 Express po raz pierwszy może zostać wyświetlone okno dialogowe umożliwiające wybór domyślnych ustawień środowiska programowania. Należy wybrać z listy pozycję *Expert Settings* (Ustawienia eksperta), a następnie kliknąć przycisk *Start Visual Studio* (Uruchom Visual Studio). Spowoduje to wyświetlenie po chwili okna środowiska Visual C# 2010 IDE.

Uwaga Aby uniknąć powtarzania się, ilekroć w tej książce mowa będzie o uruchamianiu programu Visual Studio 2010 Standard, Visual Studio 2010 Professional lub Visual C# 2010 Express będziemy pisać po prostu „uruchom program Visual Studio". Jeśli nie zostanie zaznaczone, że jest inaczej, wszelkie odwołania do terminu Visual Studio 2010 dotyczyć będą w równej mierze programów Visual Studio 2010 Standard, Visual Studio 2010 Professional oraz Visual C# 2010 Express.

- Jeśli używasz programu Visual Studio 2010 Standard lub Visual Studio 2010 Professional, w celu utworzenia nowej aplikacji konsolowej wykonaj następujące czynności:

  1. 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.
  2. Kliknij element *Visual C#*, znajdujący się w panelu po lewej stronie, w gałęzi *Installed Templates* (Zainstalowane szablony). Sprawdź, czy w polu rozwijanej listy, w górnej części środkowego panelu istnieje wartość *.NET Framework 4.0*, a następnie kliknij znajdującą się w tym panelu ikonę *Console Application* (Aplikacja konsolowa). Wyświetlenie ikony *Console Application* może wymagać przewinięcia zawartości środkowego panelu.



  3. Jeśli używasz systemu Windows Vista, wpisz w polu *Location* (Położenie) wartość **C:\Użytkownicy***TwojaNazwa***\Dokumenty\Microsoft Press\Visual CSharp Step By Step\Chapter 1**. Jeśli używasz systemu Windows 7, wpisz

C:\Użytkownicy*TwojaNazwa***\My Dokumenty\Microsoft Press\Visual CSharp Step By Step\Chapter 1**. Występującą w tych ścieżkach frazę *TwojaNazwa* zastąp własną nazwą użytkownika w systemie Windows.

Uwaga Dla skrócenia zapisu w dalszej części tej książki, odwołując się do ścieżki "*C:\Użytkownicy\TwojaNazwa\Dokumenty*" lub „*C:\Użytkownicy\TwojaNazwa\Moje Dokumenty*" będziemy pisać po prostu o folderze użytkownika **Dokumenty**.

Wskazówka Jeśli wskazany folder nie będzie istnieć, zostanie on utworzony przez program Visual Studio 2010.

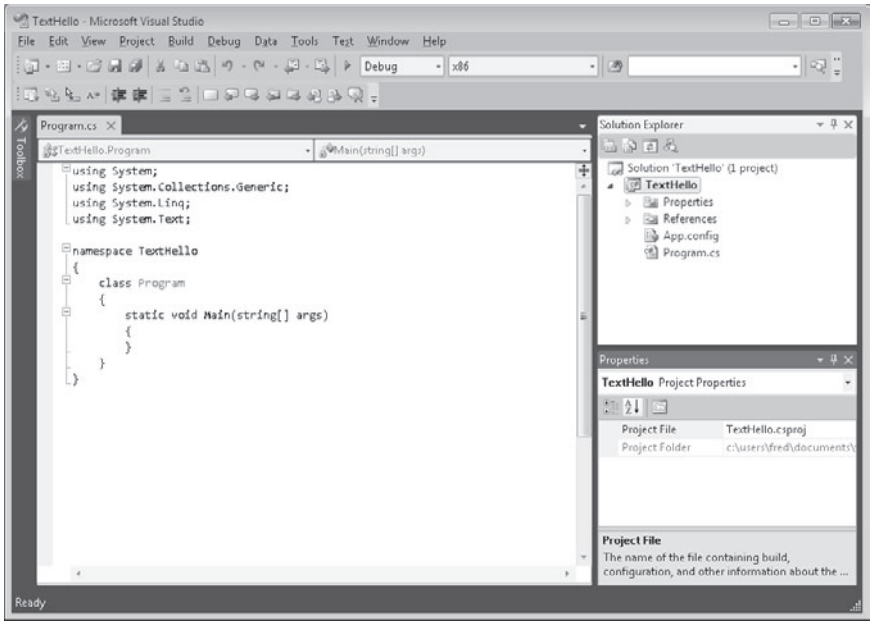
4. Wpisz w polu *Name* (Nazwa) tekst **TextHello**.
5. Upewnij się, że pole wyboru opcji *Create directory for solution* (Utwórz katalog dla rozwiązania) jest zaznaczone, a następnie kliknij przycisk *OK*.

- Jeśli używasz programu Visual C# 2010 Express, w celu utworzenia nowej aplikacji konsolowej wykonaj następujące czynności:
  1. W menu *File* (Plik) kliknij polecenie *New Project* (Nowy projekt).
  2. Kliknij ikonę *Console Application* (Aplikacja konsolowa) znajdującą się w środkowym panelu okna dialogowego *New Project* (Nowy projekt).
  3. Wpisz w polu *Name* (Nazwa) tekst **TextHello**.
  4. Kliknij przycisk *OK*.
     Domyślnie program Visual C# 2010 Express zapisuje nowe rozwiązania w katalogu *C:\Użytkownicy\TwojaNazwa\AppData\Local\Temporary Projects*. Podczas zapisywania rozwiązania możliwe jest jednak wskazanie dla projektu innej lokalizacji.
  5. Wybierz z menu *File* (Plik) polecenie *Save TextHello As* (Zapisz TextHello jako).
  6. W oknie dialogowym *Save Project* (Zapisz projekt) wskaż w polu *Location* (Położenie) folder *Dokumenty\Microsoft Press\Visual CSharp Step By Step\Chapter 1*.
  7. Kliknij przycisk *Save* (Zapisz).

Program Visual Studio utworzy nowy projekt, korzystając z szablonu *Console Application* (Aplikacja konsolowa) i wyświetli początkowy kod aplikacji, tak jak to zostało pokazane na rysunku na następnej stronie:

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 narzędzi*, który oferuje w formie przycisków skróty umożliwiające uruchamianie najczęściej używanych komend. W panelu *edytora kodu i tekstów* zajmującym główną cześć środowiska IDE wyświetlana

jest zawartość plików źródłowych. Jeśli podczas pracy z projektami złożonymi z kilku plików edytowany jest więcej niż jeden plik źródłowy, każdy z tych plików będzie mieć swoją własną zakładkę oznaczoną nazwą danego pliku. W celu przywołania wybranego pliku źródłowego na pierwszy plan okna *edytora kodu i tekstów* wystarczy kliknąć zakładkę z nazwą tego pliku. Wśród elementów wyświetlanych w panelu *Solution Explorer* (Eksplorator rozwiązania) (znajdującym się po prawej stronie okna dialogowego) wyświetlane są między innymi nazwy związanych z projektem plików. Przywołanie wybranego pliku źródłowego na pierwszy plan okna *edytora kodu i tekstów* jest również możliwe po dwukrotnym kliknięciu tego pliku w panelu eksploratora rozwiązań.



Zanim przystąpimy do pisania kodu zapoznajmy się bliżej z plikami wyświetlanymi w panelu eksploratora rozwiązań, które zostały utworzone przez program Visual Studio 2010 jako część nowego projektu:

- **Solution 'TextHello' (Rozwiązanie 'TextHello')** Jest to plik leżący na najwyższym poziomie hierarchii rozwiązania. Dla każdej aplikacji istnieje tylko jeden taki plik. Jeśli sprawdzimy za pomocą Eksploratora Windows zawartość swojego katalogu *Dokumenty\Microsoft Press\Visual CSharp Step By Step\Chapter 1\TextHello*, przekonamy się, że faktycznie plik ten nosi nazwę *TextHello.sln*. Każdy plik rozwiązania zawiera odwołanie do jednego lub kilku plików projektu.
- **TextHello** Jest to plik projektu w języku C#. Każdy plik projektu ma odwołania do jednego lub kilku plików zawierających kod źródłowy lub inne elementy projektu. 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 *TextHello.csproj* i znajduje się

w katalogu użytkownika *Dokumenty\Microsoft Press\Visual CSharp Step By Step\Chapter 1\TextHello\TextHello*.

- **Properties (Właściwości)** Jest to folder będący częścią projektu *TextHello*. Po rozwinięciu tego folderu 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 skompilowanego kodu, który może być używany przez tworzoną aplikację. Podczas kompilacji kodu następuje jego konwersja na tzw. zestaw wykonywalny (ang. *assembly*), który otrzymuje unikalną nazwę. 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. Większość funkcji, które będziemy stosować podczas tworzenia opisywanych w tej książce przykładowych aplikacji, korzysta z zestawów wykonywalnych dostarczonych przez firmę Microsoft razem z oprogramowaniem Visual Studio 2010.
- **App.config** Jest to plik konfiguracyjny aplikacji. Plik ten umożliwia określanie ustawień, które będą mogły być wykorzystywane przez uruchomioną aplikację do modyfikowania sposobu jej działania, takie 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#, ten sam, który został wyświetlany w oknie *edytora kodu i tekstów* 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 2010, kod ten wkrótce zostanie przez nas dokładniej przeanalizowany.

Piszemy pierwszy program

Plik *Program.cs* definiuje klasę o nazwie *Program*, która zawiera metodę o nazwie *Main*. Wszystkie metody muszą być definiowane wewnątrz klasy. Więcej na temat klas dowiemy się z rozdziału 7 zatytułowanego „Tworzenie i zarządzanie klasami oraz obiektami”. Metoda *Main* ma specjalne znaczenie – określa tzw. punkt wejścia do programu. Metoda ta musi być statyczna (szczegóły budowy metod zostaną omówione w rozdziale 3 zatytułowanym „Tworzenie metod i stosowanie zasięgów zmiennych”, a metody statyczne zostaną opisane we wspomnianym już rozdziale 7).

Ważne W języku C# są rozróżniane małe i wielkie litery. Metoda *Main* musi mieć nazwę pisaną z 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 tekstów* wyświetlającym zawartość pliku *Program.cs* umieść kursor wewnątrz metody *Main*, bezpośrednio za otwierającym nawiasem klamrowym ({) a następnie wciśnij klawisz Enter, aby utworzyć nową linię. Wpisz w nowej linii słowo *Console*, które jest nazwą wbudowanej klasy. Po wpisaniu litery *C* będącej pierwszą literą słowa *Console* funkcja IntelliSense wyświetli pewną listę. Lista ta zawierać będzie wszystkie słowa kluczowe języka C# oraz typy danych, które są poprawne w danym kontekście. Możesz albo kontynuować wpisywania całego słowa *Console*, albo odszukać je na liście i dwukrotnie kliknąć myszą. Po wpisaniu liter *Con* 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 wbudowaną klasą zawierającą metody służące do wyświetlania komunikatów na ekranie oraz do pobierania danych wprowadzanych przy użyciu klawiatury.

2. 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*.
3. 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
}
```

4. 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 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*.

5. Wpisz znak nawiasu zamykającego [)] a po nim znak średnika (;).

   Metoda *Main* powinna teraz wyglądać następująco:

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

6. 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 zwyczaj, by znaki takie jak **(** i **)** oraz **{** i **}** wpisywać parami jeszcze przed wypełnieniem ich treścią. Zwlekając z wpisaniem znaku nawiasu zamykającego do chwili wprowadzenia treści, która powinna być ujęta w te nawiasy, można łatwo zapomnieć o wpisaniu znaku zamykającego.

Budowanie i uruchamianie aplikacji konsolowej

1. Wybierz z menu *Build* (Budowa) polecenie *Build Solution* (Zbuduj rozwiązanie).

   Wykonanie tej akcji spowoduje skompilowanie kodu w języku C#, prowadząc do utworzenia wykonywalnego programu. Poniżej okna *edytora kodu i tekstów* wyświetlone zostanie okno *Output* (Wyjście).

Ikony funkcji IntelliSense

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

Ikona	Znaczenie
	metoda (zostanie omówiona w rozdziale 3 „Tworzenie metod i stosowanie zasięgów zmiennych”)
	właściwość (zostanie omówiona w rozdziale 15 „Implementacja właściwości pól dostępu”)
	klasa (zostanie omówiona w rozdziale 7 „Tworzenie i zarządzanie klasami oraz obiektami”)
	struktura (zostanie omówiona w rozdziale 9 „Tworzenie typów wartości przy użyciu wyliczeń oraz struktur”)
	zmienna wyliczeniowa (zostanie omówiona w rozdziale 9 „Tworzenie typów wartości przy użyciu wyliczeń oraz struktur”)
	interfejs (zostanie omówiony w rozdziale 13 „Tworzenie interfejsów oraz definiowanie klas abstrakcyjnych”)
	delegacja (zostanie omówiona w rozdziale 17 „Przerywanie działania programu oraz obsługa zdarzeń”)
	metoda rozszerzająca (zostanie omówiona w rozdziale 12 „Dziedziczenie”)

Podczas wpisywania kodu w innych kontekstach można spotkać się także z innymi ikonami funkcji IntelliSense.

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

```
Console.ReadLine(); // Czekaj na wciśniecie przez użytkownika klawisza Enter
```

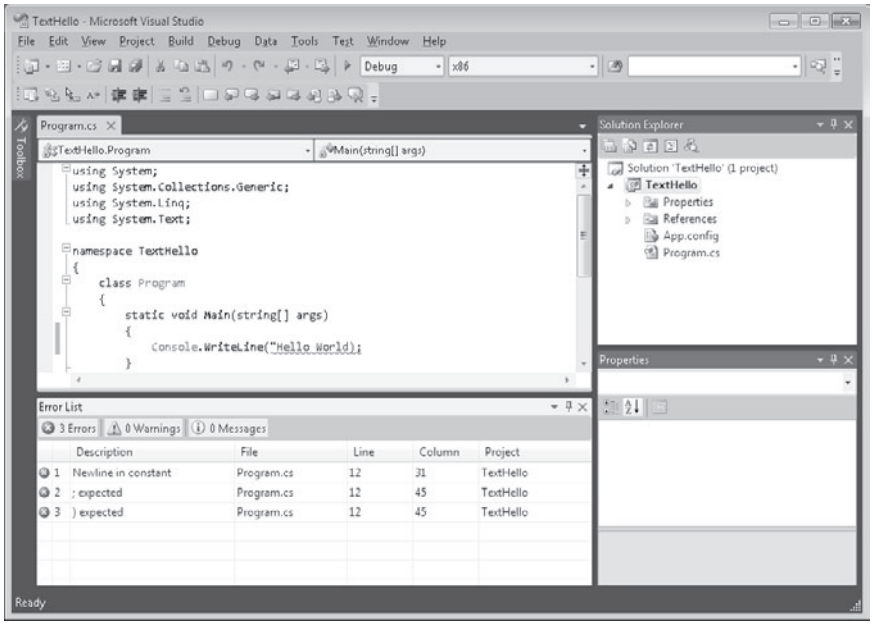
Kompilator pomija cały tekst począwszy od dwóch znaków ukośnika aż do końca danej linii. Możliwe jest także dodawanie komentarzy składających się z wielu linii, które rozpoczynają się od znaku ukośnika i gwiazdki (**/***). Po napotkaniu tych znaków kompilator pomija wszystko aż do napotkania sekwencji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

Wskazówka Jeśli okno *Output* (Wyjście) nie zostanie wyświetlone, można je wyświetlić, wybierając z menu *View* (Widok) polecenie *Output*.

W oknie *Output* (Wyjście) powinien wówczas zostać wyświetlony komunikat podobny do tego, który został pokazany poniżej, informujący o przebiegu procesu kompilacji programu:

```
------ Build started: Project: TextHello, Configuration: Debug x86 ----
CopyFilesToOutputDirectory:
  TextHello -> C:\Users\John\My Documents\Microsoft Press\Visual CSharp Step By
Step\Chapter 1\TextHello\TextHello\bin\Debug\TextHello.exe
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ========
```

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



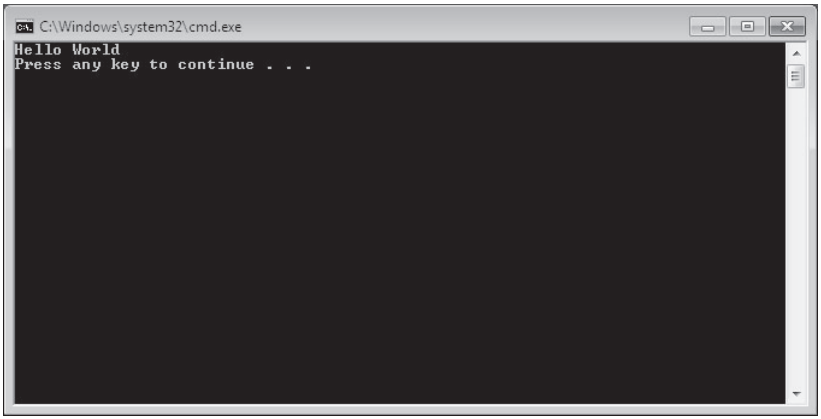
Wskazówka Dwukrotne kliknięcie wybranego elementu w oknie *Error List* (Lista błędów) spowoduje umieszczenie kursora w linii, która wywołała dany błąd. Należy także zauważyć, że podczas wpisywania kodu źródłowego program Visual Studio podkreśla czerwoną falistą linią wszystkie linie kodu, które nie będą mogły zostać poprawnie skompilowane.

Jeśli wszystkie poprzednie instrukcje zostały wykonane dokładnie i z należytą starannością, 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* (Zbuduj rozwiązanie) powoduje automatyczne zapisanie pliku źródłowego.
Gwiazdka widniejąca obok nazwy pliku na zakładce okna *edytora kodu i tekstów* 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 polecenia i uruchomienie programu. Uruchomiony program wyświetli komunikat „Hello World" i będzie oczekiwać na wciśnięcie przez użytkownika dowolnego klawisza, tak jak to zostało pokazane na rysunku:

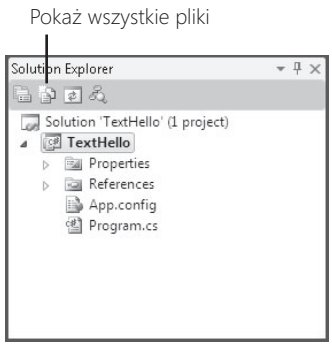


Uwaga Tekst monitu „Press any key to continue . . ." (Wciśnij dowolny klawisz, aby kontynuować ...) został wygenerowany przez program Visual Studio; w naszym projekcie nie ma żadnego kodu powodującego wypisanie tego komunikatu. Jeśli program zostałby uruchomiony przy użyciu polecenia *Start Debugging* (Rozpocznij debugowanie) z menu *Debug*, 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 polecenia, w którym wyświetlane są rezultaty działania programu, jest aktywne (ma tzw. fokus), a następnie wciśnij klawisz Enter.

Spowoduje to zamknięcie okna wiersza polecenia i powrót do środowiska programowania Visual Studio 2010.

4. W oknie *Solution Explorer* (Eksplorator rozwiązania) kliknij projekt *TextHello* (projekt, a nie rozwiązanie), a następnie przycisk *Show All Files* (Pokaż wszystkie pliki) znajdujący się na pasku narzędziowym eksploratora rozwiązań – drugi od lewej strony.



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\TextHello\TextHello*). 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 rozwiązania) rozwiń gałąź *bin*.

Ukaże się wówczas kolejny folder o nazwie *Debug*.

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

6. W oknie *Solution Explorer* (Eksplorator rozwiązania) rozwiń folder *Debug*.

Po rozwinięciu tego folderu pojawią się w nim cztery kolejne elementy o nazwach *TextHello.exe, TextHello.pdb, TextHello.vshost.exe* oraz *TextHello.vshost.exe.manifest*.

Plik *TextHello.exe* jest skompilowanym programem 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 2010 podczas uruchamiania programu w trybie debugowania (po wybraniu z menu *Debug* polecenia *Start Debugging*).

Przestrzenie nazw

Prezentowany dotychczas przykład to bardzo mały program. Małe programy mogą jednak bardzo szybko rozrosnąć się do dużo większych rozmiarów. Wraz z powiększaniem się rozmiarów programu, pojawiają się dwa główne problemy. Po pierwsze, w przypadku dużych programów utrzymywanie ich kodu oraz zrozumienie sposobu działania staje się trudniejsze niż w przypadku małych programów. Po drugie, większa ilość

kodu zwykle oznacza większą liczbę nazw, metod oraz klas. Wraz ze wzrostem liczby nazw rośnie również prawdopodobieństwo niepowodzenia procesu budowy projektu spowodowane konfliktem dwóch lub więcej nazw (zwłaszcza gdy program korzysta także z bibliotek dostarczonych przez innych producentów oprogramowania, które przecież również zostały napisane przez programistów używających różnych nazw).

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 nazwanego 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 *TextHello* może wyglądać następująco:

```
namespace TextHello
{
    class Pozdrowienia
    {
        ...
    }
}
```

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

Zaleca się, aby wszystkie tworzone klasy były definiowane przy użyciu przestrzeni nazw, do której należą, i środowisko Visual Studio 2010 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 faktyczną, pełna nazwą tej klasy jest *System.Console*.

Oczywiście, jeśli korzystając z klasy, musielibyśmy za każdym razem wpisywać jej pełną nazwę, sytuacja nie byłaby wcale lepsza niż wówczas, gdybyśmy w ogóle nie korzystali z przestrzeni nazw tylko stosowali 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 2010 projektu *TextHello* i przyjrzymy

się widocznej w oknie edytora kodu i tekstów zawartości pliku *Program.cs*, zauważymy na początku tego pliku następujące instrukcje:

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

Instrukcja *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. Cztery przestrzenie nazw pokazane w tym przykładzie zawierają klasy, które są używane na tyle często, że program Visual Studio 2010 dodaje je automatycznie przy użyciu instrukcji *using*, do każdego nowo tworzonego projektu. Możliwe jest oczywiście dodanie na początku pliku źródłowego kolejnych instrukcji *using*.

Koncepcja przestrzeni nazw zostanie zaprezentowana dokładniej w poniższym ćwiczeniu.

Ręczne wpisywanie długich nazw

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

   ```
   //using System;
   ```

2. Wybierz z menu *Build* polecenie *Build Solution*.

   Proces budowy 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 wygenerował wystąpienie tego błędu.

4. Popraw w oknie *edytora kodu i tekstów* treść metody *Main* tak, by używała 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 słowa *System*, funkcja IntelliSense wyświetli nazwy wszystkich elementów istniejących w przestrzeni nazw *System*.

5. Wybierz z menu *Build* polecenie *Build Solution*.

   Tym razem proces budowy zakończy się powodzeniem. Jeśli tak się nie stanie, należy sprawdzić, czy metoda *Main* wygląda dokładnie tak, jak w pokazanym wcześniej fragmencie kodu, a następnie ponowić próbę kompilacji kodu.

6. Uruchom aplikację, wybierając z menu *Debug* polecenie *Start Without Debugging*, aby przekonać się, że nadal działa ona poprawnie.

Przestrzenie nazw a wykonywalne zestawy binarne

Instrukcja *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 kilka tysięcy 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, który zawiera wszystkie typowe klasy, takie jak np. *System.Console*, a także osobne zestawy zawierające klasy służące do manipulowania bazami danych, korzystania z usług sieci web, 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 instrukcje *using*, które spowodują 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 pochodzące z różnych przestrzeni nazw, a pojedyncza przestrzeń nazw może rozciągać się na kilka zestawów wykonywalnych. 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 rozwiązania) dla otwartego projektu *TextHello*. 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.DataExtensions*, *System.Xml* oraz *System.Xml.Linq*. Dodatkowe odwołania można dodać do projektu, klikając folder *References* prawym klawiszem myszy i wybierając z menu kontekstowego polecenie *Add Reference* (Dodaj odwołanie) – zadanie to zostanie wykonane podczas ćwiczeń zamieszczonych w dalszej części tego rozdziału.

Tworzenie aplikacji graficznej

Dotychczas użyliśmy programu Visual Studio 2010 do utworzenia i uruchomienia prostej aplikacji konsolowej. Środowisko programowania Visual Studio 2010 zawiera także wszystko, czego będziemy potrzebować do utworzenia graficznych aplikacji dla systemu Windows. Oparty na formularzach graficzny interfejs aplikacji dla systemu Windows można projektować w sposób interaktywny. Oprogramowanie Visual Studio 2010 wygeneruje następnie instrukcje programu implementujące zaprojektowany interfejs użytkownika.

Środowisko Visual Studio 2010 oferuje dwa rodzaje widoków dla aplikacji graficznych: *widok projektowy* (Design view) oraz *widok kodu* (Code view). Okno *edytora kodu i tekstów* pozwala na modyfikowanie i utrzymywanie kodu oraz logiki tworzonej aplikacji graficznej, a widok projektowy pozwala na graficzne rozmieszczanie elementów interfejsu użytkownika. Możliwe jest przełączenie się w dowolnej chwili pomiędzy tymi dwoma widokami.

Zamieszczony poniżej zestaw ćwiczeń demonstruje sposób tworzenia aplikacji graficznej przy użyciu Visual Studio 2010. Program ten wyświetlać będzie prosty formularz zawierający pole tekstowe, w którym można będzie wpisać swoje imię oraz przycisk, którego naciśnięcie spowoduje wyświetlenie okna ze spersonalizowanym tekstem pozdrowienia.

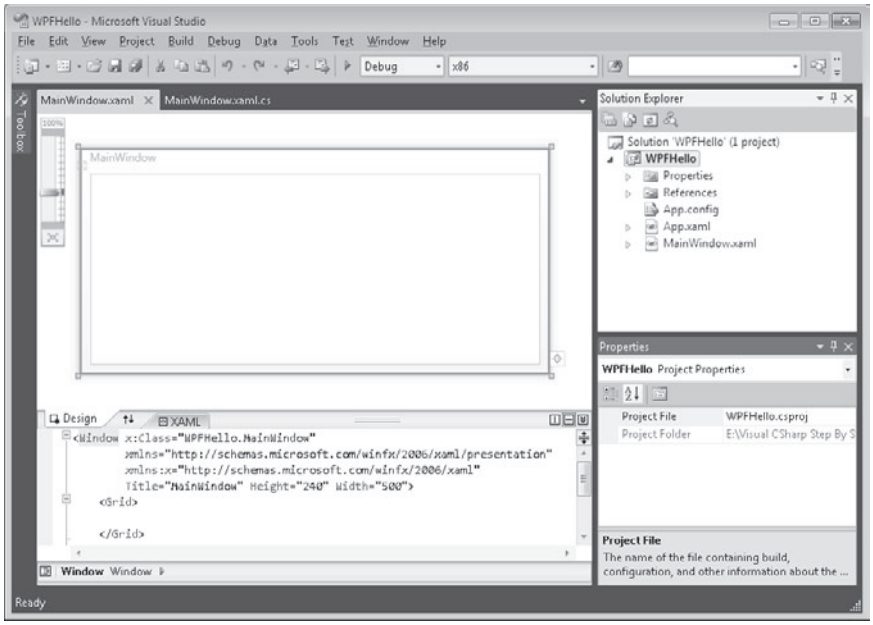
Uwaga Visual Studio 2010 oferuje dwa szablony służące do tworzenia aplikacji graficznych – szablon o nazwie *Windows Forms Application* oraz szablon *WPF Application*. Windows Forms to technologia, która po raz pierwszy pojawiła się środowisku .NET Framework w wersji 1.0. WPF, czyli Windows Presentation Foundation, rozszerzona i unowocześniona technologia, wprowadzona po raz pierwszy w środowisku .NET Framework w wersji 3.0. Technologia ta oferuje wiele dodatkowych funkcji i możliwości w stosunku do technologii Windows Forms i dlatego powinna być preferowana we wszystkich nowych projektach.

Tworzenie aplikacji graficznej w środowisku Visual Studio 2010

- Korzystając z oprogramowania Visual Studio 2010 Standard lub Visual Studio 2010 Professional do utworzenia nowej aplikacji graficznej, należy wykonać następujące kroki:
  1. 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).
  2. Kliknij element *Visual C#* znajdujący się w lewym panelu w sekcji *Installed Templates* (Zainstalowane szablony).
  3. Kliknij znajdującą się w środkowym panelu ikonę *WPF Application*.
  4. Upewnij się, że ścieżka podana w polu *Location* (Położenie) wskazuje katalog *Dokumenty\Microsoft Press\Visual CSharp Step By Step\Chapter 1*.
  5. Wpisz w polu *Name* (Nazwa) tekst **WPFHello**.
  6. Upewnij się, że w sekcji *Solution* (Rozwiązanie) zaznaczone zostało pole wyboru opcji *Create new solution* (Utwórz nowe rozwiązanie).
     Wykonanie opisanych działań spowoduje utworzenie nowego rozwiązania, które będzie mogło zawierać tworzony projekt. Możliwe jest również skorzystanie z polecenia *Add to Solution* (Dodaj do rozwiązania), które spowoduje dodanie nowego projektu do istniejącego już rozwiązania *TextHello*.
  7. Kliknij przycisk *OK*.
- W przypadku korzystania z oprogramowania Visual C# 2010 Express w celu utworzenia nowej aplikacji graficznej należy wykonać następujące kroki:
  1. Wybierz z menu *File* (Plik) polecenie *New Project* (Nowy projekt).
  2. Jeśli zostanie wyświetlone okno dialogowe *New Project* (Nowy projekt), kliknij przycisk *Save* (Zapisz), aby zapisać dotychczasowe zmiany wprowadzone w projekcie *TextHello*. W oknie dialogowym *Save Project* (Zapisz projekt) upewnij się, że wartość w polu *Location* (Położenie) wskazuje folder *Dokumenty\Microsoft Press\Visual CSharp Step By Step\Chapter 1*, a następnie kliknij przycisk *Save* (Zapisz).
  3. W oknie dialogowym *New Project* (Nowy projekt), kliknij ikonę *WPF Application*.
  4. Wpisz w polu *Name* (Nazwa) tekst **WPFHello**.
  5. Kliknij przycisk *OK*.

  Program Visual Studio 2010 zamknie wówczas aktualnie otwartą aplikację i utworzy nową, opartą na technologii WPF. Początkowo wyświetlane będzie okno działające w trybie widoku projektowego i zawierające pusty formularz WPF oraz inne okno zawierające opis tego formularza w języku XAML, tak jak to zostało pokazane na następnej stronie.

Wskazówka Aby zyskać na ekranie więcej miejsca potrzebnego do wyświetlania okna *widoku projektowego*, można zamknąć okna *Output* (Wyjście) oraz *Error List* (Lista błędów).



Skrót XAML pochodzi od słów Extensible Application Markup Language, co oznacza rozszerzalny język znaczników aplikacji. Jest to język podobny do języka XML i używany przez aplikacje WPF do definiowania rozkładu graficznych elementów formularza oraz jego zawartości. Osobom znającym już język XML język XAML powinien wydać się znajomy. Jeśli ktoś nie lubi korzystać z okna *widoku projektowego* programu Visual Studio lub nie ma dostępu do tego programu, może zdefiniować formularz WPF wyłącznie poprzez napisanie jego opisu w języku XAML. Firma Microsoft oferuje edytor XAML o nazwie *XAMLPad*, który jest instalowany w systemie razem z pakietem Windows SDK (Software Development Kit).

W kolejnym ćwiczeniu posłużymy się oknem *widoku projektowego*, za pomocą którego dodamy do formularza systemu Windows trzy nowe kontrolki oraz zapoznamy się z kodem w języku *C#* implementującym funkcjonalność tych kontrolek, który zostanie automatycznie wygenerowany przez program Visual Studio 2010.

Tworzenie interfejsu użytkownika

1. Kliknij zakładkę *Toolbox* (Skrzynka narzędziowa) znajdującą się w oknie widoku projektowego po lewej stronie formularza.

   Spowoduje to wyświetlenie panelu narzędziowego *Toolbox* częściowo przesłaniającego projektowany formularz systemu Windows i zawierającego różne komponenty

oraz składniki, które można dodawać do formularza. Rozwiń sekcję o nazwie *Common WPF Controls* (Typowe kontrolki WPF). Znajduje się tu lista kontrolek używanych przez większość aplikacji WPF.

2. Kliknij ikonę *Label* (Etykieta) znajdującą się w sekcji *Common WPF Controls* (Typowe kontrolki WPF), a następnie przeciągnij tę etykietę do widocznej części formularza.

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

Wskazówka Jeśli chcesz, by panel *Toolbox* (Skrzynka narzędziowa) 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 trwałe wyświetlanie panelu *Toolbox* po lewej stronie okna programu Visual Studio 2010 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*.

3. Umieszczona na formularzu kontrolka 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 etykiety w okolicę górnego, prawego narożnika formularza (w przypadku tej aplikacji dokładne umiejscowienie etykiety nie jest ważne).

Uwaga Opis formularza w języku XAML, który jest wyświetlany w dolnym panelu, obejmuje teraz kontrolkę etykiety wraz z jej właściwościami, takimi jak np. położenie w obrębie formularza, ustalane przez wartość właściwości kontrolki o nazwie *Margin* (Margines). Właściwość *Margin* składa się z czterech liczb oznaczających odległość każdej krawędzi etykiety od krawędzi formularza. Przesuwanie etykiety w obrębie formularza powodować będzie odpowiednią zmianę wartości tych liczb. W przypadku zmiany rozmiarów formularza nastąpi także zmiana rozmiarów tych kontrolek, które są zakotwiczone do jednej z krawędzi. Można temu zapobiec, ustawiając dla właściwości *Margin* wartości *0*. Więcej informacji na temat właściwości *Margin* (Margines), *Height* (Wysokość) oraz *Width* (Szerokość) kontrolek WPF znajduje się w rozdziale 22 zatytułowanym „Omówienie platformy Windows Presentation Foundation".

4. 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 rozwiązania), 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 tego okna zależna jest od kontekstu, co oznacza, że wyświetlane są w nim właściwości aktualnie zaznaczonego elementu. Kliknięcie paska tytułowego formularza wyświetlanego w oknie *widoku projektowego* spowoduje wyświetlenie w oknie *Properties* właściwości samego formularza. Po kliknięciu kontrolki etykiety w oknie tym wyświetlone zostaną właściwości tej etykiety. Natomiast kliknięcie w dowolnym miejscu formularza spowoduje wyświetlenie właściwości tajemniczego elementu nazywanego *siatką* (ang. *grid*). Siatka pełni rolę kontenera dla elementów znajdujących się na formularzu WPF i pozwala między innymi na określanie sposobu wzajemnego rozmieszczenia względem siebie, wyrównywania i grupowania tych elementów.

5. Kliknij znajdującą się na formularzu kontrolkę etykiety, a następnie odszukaj w oknie *Properties* właściwość *FontSize* (Rozmiar czcionki). Zmień wartość właściwości *FontSize* na 20, a następnie kliknij w oknie *widoku projektowego* pasek tytułowy formularza.

   Spowoduje to zmianę rozmiaru tekstu wyświetlanego na etykiecie.

6. Korzystając z panelu XAML wyświetlanego poniżej okna *widoku projektowego*, przeanalizuj tekst definiujący kontrolkę etykiety. Na końcu linii powinien znajdować się tekst *FontSize=„20”*. Wszelkie zmiany wykonywane za pomocą okna *Properties* są automatycznie odzwierciedlane w definicji zapisanej w języku XAML i na odwrót.

   Przywróć poprzednią wartość właściwości *FontSize*, nadpisując jej bieżącą wartość w panelu XAML wartością 12. Spowoduje to ponowną zmianę wielkości tekstu wyświetlanego na etykiecie w oknie widoku projektowego.

7. Korzystając z panelu XAML, zapoznaj się z innymi właściwościami kontrolki etykiety.

   W panelu XAML widoczne są tylko te właściwości, które mają wartość inną niż domyślna. Jeśli zmodyfikujemy wartość dowolnej właściwości za pomocą okna *Properties* (Właściwości), pojawi się ona w panelu XAML jako cześć definicji etykiety.

8. Zmień wartość właściwości *Content* (Zawartość) z **Label** (Etykieta) na **Please enter your name** (Proszę podać swoje imię).

   Zwróć uwagę na fakt, że spowoduje to zmianę tekstu wyświetlanego na umieszczonej na formularzu etykiecie, choć sama etykieta jest zbyt mała, aby wyświetlić ten tekst w całości.

9. Kliknij kontrolkę etykiety, znajdującą się w oknie *widoku projektowego*. Umieść wskaźnik myszy nad prawą krawędzią tej kontrolki. Wskaźnik myszy powinien wówczas zmienić się na skierowaną w dwie strony strzałkę, oznaczającą możliwość zmiany rozmiarów kontrolki przy pomocy myszy. Naciśnij przycisk myszy

i przeciągaj prawą krawędź kontrolki na prawą stronę, tak długo, aż cały tekst etykiety stanie się widoczny.

10. Kliknij formularz znajdujący się w oknie *widoku projektowego*, a następnie wyświetl ponownie panel *Toolbox* (Skrzynka narzędziowa).

11. 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 etykiety.

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.

12. Gdy zaznaczoną kontrolką jest kontrolka pola tekstowego, zmień w oknie *Properties* wartość wyświetlanej na początku listy właściwości *Name* na *userName* (Nazwa użytkownika).

Uwaga Konwencja tworzenia nazw dla kontrolek i zmiennych zostanie omówiona w rozdziale 2 zatytułowanym „Zmienne, operatory i wyrażenia".

13. Wyświetl ponownie panel *Toolbox*, a następnie kliknij znajdującą się w tym panelu kontrolkę *Button* (Przycisk) i przeciągnij ją na formularz. Umieść kontrolkę przycisku po prawej stronie kontrolki pola tekstowego w taki sposób, by dolna krawędź przycisku była wyrównana w poziomie z dolną krawędzią pola tekstowego.

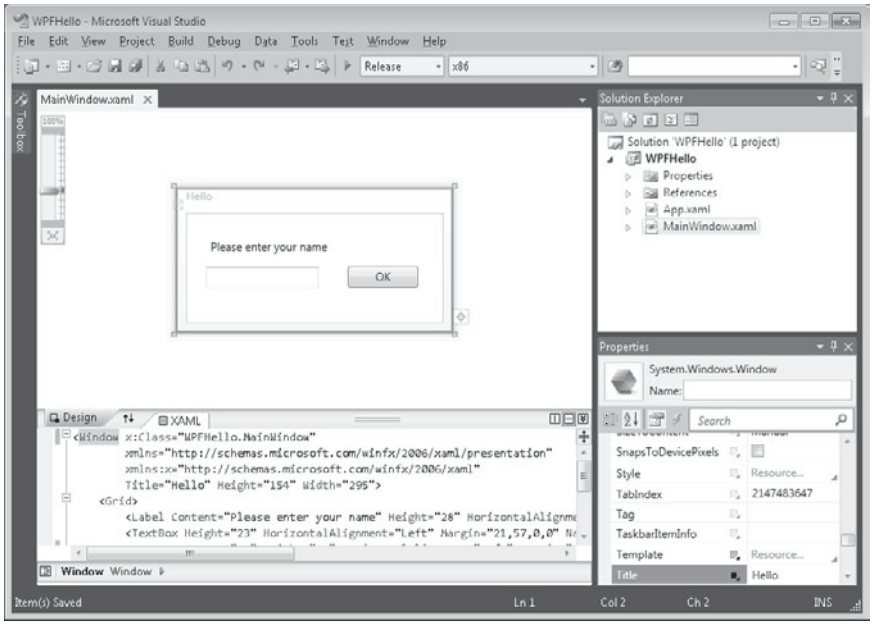
14. Korzystając z okna *Properties,* zmień dla kontrolki przycisku wartość jej właściwości o nazwie *Name* (Nazwa) na **Ok**. Zmień również wartość właściwości *Content* property (Zawartość) z *Button* (Przycisk) na **OK**. Upewnij się, że działanie to spowodowało również zmianę tekstu wyświetlanego na znajdującej się na formularzu kontrolce przycisku.

15. Korzystając z okna widoku projektowego, kliknij pasek tytułowy formularza *MainWindow.xaml*. Zmień w oknie *Properties* wartość właściwości *Title* na **Hello**.

16. Zwróć uwagę na fakt, że po zaznaczeniu formularza w oknie *widoku projektowego* w dolnym lewym narożniku tego formularza wyświetlany jest uchwyt służący do zmiany rozmiaru (niewielki kwadrat). Umieść wskaźnik myszy nad tym uchwytem. Gdy kursor zmieni się w ukośną, skierowaną w dwie strony strzałkę, wciśnij przycisk myszy i przeciągnij wskaźnik, zmieniając rozmiar formularza. Zaprzestań przeciągania kursora i zwolnij przycisk myszy, gdy ilość wolnego miejsca otaczającego umieszczone na formularzu kontrolki będzie mniej więcej jednakowa.

Ważne Przed zmianą rozmiaru formularza należy kliknąć pasek tytułowy tego formularza, a nie kontur znajdującej się wewnątrz formularza siatki. Zaznaczenie siatki spowoduje zmodyfikowanie rozmieszczenia kontrolek na formularzu zamiast zmiany rozmiarów samego formularza.

Gotowy formularz powinien wyglądać podobnie do pokazanego na poniższym rysunku.



17. Wybierz z menu *Build* polecenie *Build Solution* (Buduj rozwiązanie) i upewnij się, czy proces budowy przebiegł bez błędów.

18. Wybierz z menu *Debug* polecenie *Start Without Debugging*.

    Powinno to spowodować uruchomienie aplikacji i wyświetlenie przez nią naszego formularza. Możesz wpisać swoje imię w polu tekstowym i kliknąć przycisk *OK*, ale nic się wówczas nie stanie. Konieczne jest jeszcze dodanie kodu, który przetworzy zdarzenie *Click* dla przycisku *OK*, co uczynimy w następnym kroku.

19. Kliknij przycisk *Zamknij* (symbol *X* znajdujący się w prawym, górnym narożniku formularza), aby zamknąć uruchomiony formularz 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 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

własny kod, dobrze będzie zaznajomić się z kodem wygenerowanym dla nas przez program Visual Studio.

W tym celu należy w oknie *Solution Explorer* (Eksplorator rozwiązania) rozwinąć węzeł o nazwie *MainWindow.xaml*. Węzeł ten zawiera plik *MainWindow.xaml.cs*. Kliknij dwukrotnie plik *MainWindow.xaml.cs*. W oknie *edytora kodu i tekstów* zostanie wówczas wyświetlony kod formularza. Kod ten wygląda następująco:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace WPFHello
{
    /// <summary>
    /// Logika interakcji dla MainWindow.xaml
    /// </summary>

    public partial class MainWindow : Window
    {

        public MainWindow()
        {
            InitializeComponent();
        }

    }
}
```

Oprócz sporej liczby instrukcji *using* powodujących włączenie do zasięgu przestrzeni nazw używanych przez większość aplikacji WPF plik ten zawiera definicję klasy o nazwie *MainWindow* i prawie nic po za tym. W pliku istnieje pewna niewielka ilość kodu dla klasy *MainWindow* będąca konstruktorem tej klasy i zawierająca wywołanie metody o nazwie *InitializeComponent* (Zainicjuj składnik) i to wszystko (konstruktor to specjalna metoda o takiej samej nazwie jak klasa, której dotyczy. Metoda ta jest wykonywana podczas tworzenia nowego egzemplarza danej klasy i może zawierać kod służący do inicjowania nowej instancji klasy). W rzeczywistości utworzona przez nas aplikacja zawiera o wiele więcej kodu, ale większość z niego 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.

Widoczna część kodu tej klasy służy do tego, by można było umieścić w niej własne metody służące do obsługi logiki aplikacji, tak jak np. metoda określająca, co się stanie po kliknięciu przez użytkownika przycisku *OK*.

> **Wskazówka** Kliknięcie prawym przyciskiem myszy w dowolnym miejscu okna *widoku projektowego* i wybranie z menu kontekstowego polecenia *View Code* (Pokaż kod) pozwala także na wyświetlenie napisanego w języku C# kodu samego formularza.

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. Należy bowiem pamiętać, że to właśnie metoda *Main* definiuje punkt, od którego rozpoczyna się wykonywanie aplikacji. W oknie eksploratora rozwiązania znajduje się jeszcze jeden plik źródłowy o nazwie *App.xaml*. Dwukrotne kliknięcie tego pliku spowoduje wyświetlenie opisu w języku XAML. Jedną z właściwości występujących w tym kodzie XAML jest właściwość o nazwie *StartupUri*, a jej wartość wskazuje na plik *MainWindow.xaml*, co w pokazanym poniżej przykładowym fragmencie kodu zostało wyróżnione pogrubioną czcionką:

```
<Application x:Class="WPFHello.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com.winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Kliknięcie zakładki *Design* (Projekt) znajdującej się w dolnej części panelu XAML spowoduje wyświetlenie okna widoku projektowego dla pliku *App.xaml*, w którym wyświetlany będzie tekst „*Intentionally left blank. The document root element is not supported by the visual designer*" (To miejsce zostało celowe pozostawione puste. Główny element dokumentu nie jest obsługiwany przez graficzne narzędzia projektowe). Dzieje się tak, dlatego że pliku *App.xaml* nie można modyfikować za pomocą okna widoku projektowego.

Po rozwinięciu w panelu eksploratora rozwiązania węzła *App.xaml* okaże się, że węzeł ten zawiera także plik *Application.xaml.cs*. Dwukrotne kliknięcie tego pliku spowoduje wyświetlenie jego zawartości, na którą składa się pokazany poniżej kod:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Windows;

namespace WPFHello
{
    /// <summary>
```

```
    /// Logika interakcji App.xaml
    /// </summary>

    public partial class App : Application
    {
    }
}
```

Również i tym razem otwarty plik zawiera kilka instrukcji *using* i prawie nic poza tym, nie ma w nim nawet metody *Main*. W rzeczywistości plik ten zawiera metodę *Main*, ale pozostaje ona ukryta. Kod metody *Main* jest generowany na podstawie ustawień zapisanych w pliku *App.xml*; w szczególności metoda *Main* powoduje wyświetlenie formularza wskazanego za pomocą właściwości *StartupUri*. Jeśli chcemy, by tworzona aplikacja wyświetlała inny formularz, należy dokonać edycji pliku *App.xaml*.

Nadszedł wreszcie czas na samodzielne napisanie części kodu!

Pisanie kodu do obsługi przycisku *OK*

1. Kliknij zakładkę *MainWindow.xaml* widoczną ponad górną krawędzią okna edytora kodu i tekstów, aby wyświetlić w oknie *widoku projektowego* formularz zdefiniowany w pliku *MainWindow*.
2. Kliknij dwukrotnie widniejący na formularzu przycisk *OK*.

   Spowoduje to wyświetlenie w oknie *edytora kodu i tekstów* zawartości pliku *MainWindow.xaml.cs*, do którego została dodana nowa metoda o nazwie *ok_Click*. Program Visual Studio automatycznie wygeneruje kod powodujący wywoływanie tej metody zawsze, gdy użytkownik kliknie przycisk *OK*. Jest to przykład tzw. zdarzenia. W dalszej części tej książki dowiemy się znacznie więcej o sposobie działania zdarzeń.
3. Dodaj do metody *ok_Clik* kod, który poniżej został wyróżniony pogrubioną czcionką:

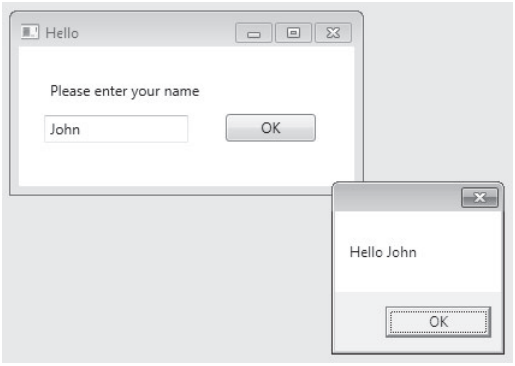
   ```
   void ok_Click(object sender, RoutedEventArgs e)
   {
       MessageBox.Show("Hello " + userName.Text);
   }
   ```

   Jest to kod, który będzie uruchamiany zawsze, gdy użytkownik kliknie przycisk *OK*. Na razie nie należy zbytnio przejmować się składnią pokazanego fragmentu kodu (wystarczy po prostu skopiować go dokładnie w takiej postaci, w jakiej został pokazany powyżej), ponieważ w rozdziale 3 dowiemy się więcej o budowie metod. W tym przypadku interesująca jest instrukcja *MessageBox.Show*. Instrukcja ta powoduje wyświetlenie okna komunikatu zawierającego połączenie słowa „*Hello*” oraz imienia wpisanego przez użytkownika w znajdującym się na formularzu polu tekstowym.

4. Kliknij zakładkę *MainWindow.xaml* widoczną ponad krawędzią okna *edytora kodu i tekstów*, aby ponownie wyświetlić zawartość formularza *MainWindow* w trybie *widoku projektowego*.
5. Korzystając z dolnego panelu, w którym wyświetlanym jest 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 na element podrzędny o nazwie *Click*, który zawiera odwołanie do metody *ok_Click*:

```
<Button Height="23" … Click="ok_Click" />
```

6. Wybierz z menu *Debug* polecenie *Start Without Debugging*.
7. Po wyświetleniu formularza wpisz w polu tekstowym swoje imię i naciśnij przycisk *OK*. Zostanie wówczas wyświetlone okno komunikatu z imiennym pozdrowieniem:



8. Kliknij przycisk *OK* znajdujący się w oknie komunikatu.

   Spowoduje to zamknięcie okna komunikatu.
9. Zamknij formularz.

W tym rozdziale pokazaliśmy, jak za pomocą programu Visual Studio 2010 można tworzyć, budować i uruchamiać aplikacje graficzne. Utworzyliśmy prostą aplikację konsolową, która wyświetlała wyniki w oknie konsoli oraz aplikację WPF z prostym graficznym interfejsem użytkownika.

- Jeśli zamierzasz przejść do kolejnego rozdziału,

  pozostaw uruchomiony program Visual Studio 2010 i przejdź do rozdziału 2.
- Jeśli chcesz na tym zakończyć pracę z programem Visual Studio 2010,

  wybierz z menu *File* (Plik) polecenie *Exit* (Wyjście). Jeśli spowoduje to wyświetlenie okna dialogowego z pytaniem o zapisanie otwartego projektu, należy zapisać ten projekt i kliknąć przycisk *Yes*.

Krótkie podsumowanie rozdziału 1

W celu	Wykonaj następujące czynności
Utworzenia nowej aplikacji konsolowej przy użyciu programu Visual Studio 2010 w edycji Standard lub Professional	W menu *File* (Plik) wskaż menu podrzędne *New* (Nowy), a następnie kliknij polecenie *Project* (Projekt), co spowoduje otwarcie okna dialogowego *New Project* (Nowy projekt). Kliknij element *Visual C#* znajdujący się w lewym panelu w sekcji *Installed Templates* (Zainstalowane szablony). Kliknij ikonę *Console Application* (Aplikacja konsolowa) znajdującą się w środkowym panelu. Określ w polu *Location* (Położenie) katalog, w którym należy umieścić nowy projekt. Wpisz nazwę projektu i kliknij przycisk *OK*.
Utworzenia nowej aplikacji konsolowej przy użyciu programu Visual C# 2010 Express	Wybierz z menu *File* (Plik) polecenie *New Project* (Nowy projekt), co spowoduje otwarcie okna dialogowego *New Project* (Nowy projekt). Wybierz szablon *Console Application* (Aplikacja konsolowa). Wpisz nazwę projektu i kliknij przycisk *OK*.
Utworzenia nowej aplikacji graficznej przy użyciu programu Visual Studio 2010 w edycji Standard lub Professional	W menu *File* (Plik) wskaż menu podrzędne *New*, a następnie kliknij polecenie *Project*, co spowoduje otwarcie okna dialogowego *New Project* (Nowy projekt). Kliknij element *Visual C#* znajdujący się w lewym panelu w sekcji *Installed Templates* (Zainstalowane szablony). Kliknij ikonę *WPF Application* znajdującą się w środkowym panelu. Określ w polu *Location* (Położenie) katalog, w którym należy umieścić nowy projekt. Wpisz nazwę projektu i kliknij przycisk *OK*.
Utworzenia nowej aplikacji graficznej przy użyciu programu Visual C# 2010 Express	Wybierz z menu *File* (Plik) polecenie *New Project* (Nowy projekt), co spowoduje otwarcie okna dialogowego *New Project* (Nowy projekt). Wybierz szablon *WPF Application*. Wpisz nazwę projektu i kliknij przycisk *OK*.
Zbudowania aplikacji	Wybierz z menu *Build* polecenie *Build Solution* (Zbuduj rozwiązanie).
Uruchomienia aplikacji	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 2010, można zbudować/skompilować i uruchomić program konsolowy oraz aplikację opartą na platformie WPF (Windows Presentation Foundation). 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 operatory lub więcej.

Instrukcje

Instrukcja to polecenie wykonujące określone działanie. 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 nazwaną sekwencję 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 określająca, *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 wielkich), cyfr oraz znaków podkreślenia.
- Identyfikator musi rozpoczynać się od litery lub od znaku podkreślenia.

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 wielkie 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 własnych celów. Te zarezerwowane identyfikatory nazywane są *słowami kluczowymi* i każde 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*	

Wskazówka Podczas wpisywania słów kluczowych w oknie *edytora kodu i tekstów* są one oznaczane przez program Visual Studio 2010 kolorem niebieskim.

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

dynamic	*join*	*set*
from	*let*	*value*
get	*orderby*	*var*
group	*partial*	*where*
into	*select*	*yield*

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 mieć 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*, 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. Poniższa lista zawiera pewne ogólne rekomendacje w tym zakresie:

- Nie należy rozpoczynać identyfikatorów od znaku podkreślenia.
- 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.

Uwaga Stosowanie identyfikatorów różniących się jedynie wielkością liter może ograniczać możliwości używania we własnych aplikacjach klas utworzonych w innych językach programowania, w których małe i wielkie litery nie są rozróżniane, takich np. jak Microsoft Visual Basic.

- Nazwy powinny rozpoczynać się małą literą.
- W przypadku identyfikatorów będących złożeniem kilku słów, drugie i każde kolejne słowo powinno rozpoczynać się wielką literą (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!).

Ważne Pierwsze dwa z wymienionych powyżej zaleceń należy traktować jako obowiązkowe, ponieważ są one związane ze zgodnością ze specyfikacją CLS (Common Language Specification). Przestrzeganie tych zaleceń jest konieczne, jeśli zamierzamy napisać program, który będzie współdziałał z innymi językami programowania, takimi jak np. Microsoft Visual Basic.

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 (prymitywnych) typów danych języka C# oznaczająca liczby całkowite (w dalszej części tego rozdziału poznamy więcej podstawowych typów danych).

> **Uwaga** Programiści używający języka Microsoft Visual Basic powini 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* (wiek) wartość 42. Na przykładzie tej instrukcji widzimy raz jeszcze konieczność używania znaku 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 wartości zmiennej *wiek*, czyli liczby 42, na konsoli:

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

> **Wskazówka** Zatrzymanie wskaźnika myszy nad nazwą zmiennej w oknie *edytora kodu i tekstów* programu Visual Studio 2010 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	od -2^{31} do $2^{31} - 1$	int count; count = 42;
long	Liczby całkowite (o większym zakresie)	64	od -2^{63} do $2^{63} - 1$	long wait; wait = 42L;
float	Liczby zmiennoprzecinkowe	32	od $\pm 1.5 \times 10^{45}$ do $\pm 3.4 \times 10^{38}$	float away; away = 0.42F;
double	Liczby zmiennoprzecinkowe o podwójnej precyzji (dokładniejsze)	64	od $\pm 5.0 \times 10^{-324}$ do $\pm 1.7 \times 10^{308}$	double trouble; trouble = 0.42;
decimal	Wartości finasowe	128	28 cyfr znaczących	decimal coin; coin = 0.42M;
string	Sekwencja znaków	16 bitów na każdy znak	Nie dotyczy	string vest; vest = "fortytwo";
char	Pojedynczy znak	16	od 0 do $2^{16} - 1$	char grill; grill = 'x';
bool	Wartość logiczna (boolowska)	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 może się nie skompilować. Wymóg ten nosi nazwę *reguły przypisania określonej wartości* (ang. Definite Assignment). Przykładowo pokazane poniżej instrukcje spowodują wygenerowanie błędu w trakcie kompilacji, ponieważ zmiennej *wiek* nie przypisano żadnej wartości:

```
int wiek;
Console.WriteLine(wiek); // błąd podczas 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 2010 nie został uruchomiony już wcześniej, uruchom go teraz.
2. Jeśli używasz programu Visual Studio 2010 Standard lub Visual Studio 2010 Professional, wskaż w menu *File* (Plik) menu podrzędne *Open* (Otwórz), a następnie wybierz z niego polecenie *Project/Solution* (Projekt/Rozwiązanie).

   Jeśli używasz programu Visual C# 2010 Express, wybierz z menu *File* (Plik) polecenie *Open Project* (Otwórz projekt).

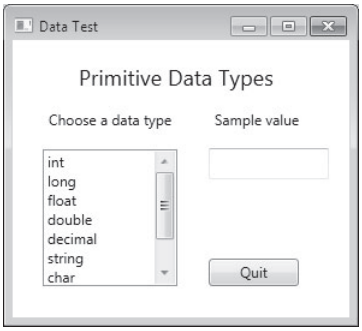
   Spowoduje to otwarcie okna dialogowego *Open Project* (Otwórz projekt).
3. Przejdź do folderu *Dokumenty\Microsoft Press\Visual CSharp Step By Step\Chapter 2\PrimitiveDataTypes*. Zaznacz plik rozwiązania *PrimitiveDataTypes* i naciśnij przycisk *Open* (Otwórz).

   Nastąpi wówczas załadowanie rozwiązania, a w panelu *Solution Explorer* (Eksplorator rozwiązania) wyświetlony zostanie projekt *PrimitiveDataTypes*.

Uwaga Pliki rozwiązań mają rozszerzenie .sln, np. PrimitiveDataTypes.sln. Rozwiązanie może zawierać jeden lub więcej projektów. Pliki projektów mają rozszerzenia .csproj. Jeśli zamiast pliku rozwiązania otwarty zostanie plik projektu, program Visual Studio 2010 automatycznie utworzy dla niego nowy plik rozwiązania. Operacja budowy rozwiązania powoduje automatyczne zapisanie przez program Visual Studio 2010 wszystkich nowych lub zmodyfikowanych plików, a więc użytkownik zostanie poproszony o podanie nazwy i lokalizacji dla nowego pliku rozwiązania.

4. Wybierz z menu *Debug* 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). Zostanie wówczas wyświetlone pokazane na następnej stronie okno aplikacji:



5. 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).

6. 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.

7. Wybierz po kolei każdy, dostępny na liście typ danych. Upewnij się, że kod wyświetlający wartości typu *double* oraz *bool* również nie został jeszcze zaimplementowany.

8. Klik przycisk *Quit* (Wyjdź), aby zamknąć okno i zakończyć działanie programu.

   Spowoduje to powrót do środowiska programowania Visual Studio 2010.

Używanie podstawowych typów danych w kodzie źródłowym

1. Kliknij dwukrotnie plik *MainWindow.xaml* znajdujący się w oknie *Solution Explorer* (Eksplorator rozwiązania).

   Spowoduje to wyświetlenie w oknie *widoku projektowego* formularza aplikacji WPF.

2. Kliknij prawym klawiszem myszy w dowolnym miejscu okna *widoku projektowego*, w którym wyświetlany jest plik *MainWindow.xaml*, a następnie wybierz z menu kontekstowego polecenie *View Code* (Pokaż kod).

   Spowoduje to otwarcie okna *edytora kodu i tekstów* i wyświetlenie w nim zawartości pliku *MainWindow.xaml.cs*.

Uwaga Należy pamiętać, że kod ten można także wyświetlić, korzystając z panelu *Solution Explorer* (Eksplorator rozwiązania). W tym celu należy kliknąć symbol + znajdujący się po lewej stronie pliku MainWindow.xaml, a następnie dwukrotnie kliknąć plik MainWindow.xaml.cs.

3. Odszukaj w oknie *edytora kodów i tekstu* metodę *showFloatValue* (Pokaż wartość typu float).

Wskazówka Chcąc zlokalizować określony element projektu, można wskazać w menu *Edit* (Edycja) menu podrzędne *Find and Replace* (Wyszukaj i zamień), a następnie wybrać z niego polecenie *Quick Find* (Szybkie wyszukiwanie). Spowoduje to otwarcie okna dialogowego pozwalającego określić, co chcemy wyszukać. Należy wpisać w tym oknie nazwę szukanego elementu i kliknąć przycisk *Find Next* (Znajdź następny). Domyślnie funkcja wyszukiwania nie rozróżnia małych i wielkich liter. Jeśli chcemy, by małe i wielkie litery były rozróżniane, należy kliknąć symbol +, znajdujący się obok etykiety *Find Options* (Opcje wyszukiwania), a następnie zaznaczyć pole wyboru opcji *Match Case* (Uwzględniaj wielkość liter). Zachęcamy także do wypróbowania pozostałych opcji wyszukiwania.

Okno dialogowe *Quick Find* (Szybkie wyszukiwanie) można również przywołać bez korzystania z menu *Edit* – poprzez wciśnięcie kombinacji klawiszy Ctrl+F (należy najpierw wcisnąć i przytrzymać klawisz Ctrl, a następnie wcisnąć klawisz F). Podobnie wciśnięcie kombinacji klawiszy Ctrl-H powoduje otwarcie okna dialogowego *Quick Replace* (Szybkie zastępowanie).

Oprócz korzystania z funkcji *Quick Find* (Szybkie wyszukiwanie) alternatywnym sposobem wyszukiwania metod klasy jest użycie rozwijanej listy członków klasy znajdującej się z prawej strony, ponad 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 *showFloatValue()* spowoduje przeniesienie kursora bezpośrednio do metody *showFloatValue* z bieżącej klasy.

Metoda *showFloatValue* jest uruchamiana, gdy z listy typów danych zostanie wybrany typ *float*. Metoda ta zawiera następujące trzy instrukcje:

```
float variable;
variable=0.42F;
value.Text = "0.42F";
```

Pierwsza z tych instrukcji deklaruje zmienną typu *float* o nazwie *variable* (zmienna).

Druga instrukcja powoduje przypisanie tej zmiennej wartości 0.42F. Litera F jest w tym przypadku tzw. określnikiem typu danych informującym, 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 napisania 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. Wyświetlanie elementów w polu tekstowym polega na ustawieniu odpowiedniej wartości dla właściwości *Text* tego pola. Należy zwrócić uwagę na fakt, że dostęp do właściwości obiektu realizowany jest przy użyciu tej samej notacji „kropkowej", z którą spotkaliśmy się już wcześniej przy okazji uruchamiania metody (w programie *Console.WriteLine* z rozdziału 1). Dane przypisywane do właściwości *Text* muszą być łańcuchem znakowym (sekwencją znaków ujętych w znaki podwójnego cudzysłowu), a nie liczbą. Jeśli spróbujemy przypisać liczbę do właściwości *Text*, program się nie skompiluje. W tym programie pokazana instrukcja wyświetla w polu tekstowym po prostu tekst „0.42F". W rzeczywistej aplikacji umieścilibyśmy instrukcję dokonującą konwersji wartości zmiennej *variable* na łańcuch znakowy, a następnie przepisali ten łańcuch do właściwości *Text*, ale aby to zrobić, musielibyśmy wiedzieć nieco więcej o języku C# oraz platformie Microsoft .NET Framework (zagadnienia konwersji typów danych zostaną omówione w rozdziale 11 zatytułowanym „Tablice parametrów" oraz w rozdziale 21 zatytułowanym „ Przeciążanie operatorów").

4. Odszukaj w oknie *edytora kodu i tekstów* metodę *showIntValue*. Metoda ta powinna wyglądać następująco:

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

Metoda to jest uruchamiana, gdy z listy typów danych zostanie wybrany typ *int*.

5. Wpisz na początku metody *showIntValue* w nowej linii rozpoczynającej się bezpośrednio po klamrowym nawiasie otwierającym dwie następujące instrukcje, które poniżej zostały wyróżnione pogrubioną czcionką:

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

6. W oryginalnej instrukcji tej metody, zmień tekst *„to do"* na *„42"*.

Zmieniona metoda powinna teraz wyglądać dokładnie tak, jak pokazano poniżej:

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

Uwaga Osoby mające już pewne doświadczenie w programowaniu mogą odczuwać pokusę by zmienić trzecią instrukcję w następujący sposób

```
value.Text = variable;
```

Mogłoby się wydawać, że taka instrukcja powinna wyświetlić wartość zmiennej *variable* w znajdującym się na formularzu polu tekstowym *value*. Język C# przeprowadza jednak ścisłą kontrolę typów danych; pola tekstowe mogą wyświetlać jedynie wartości typu *string*, a zmienna *variable* jest typu *int*, a więc taka instrukcja nie zostałaby skompilowana. W dalszej części tego rozdziału pokazane zostaną proste techniki konwersji wartości numerycznych i znakowych.

7. Wybierz z menu *Debug* polecenie *Start Without Debugging*.

   Spowoduje to ponowne wyświetlenie formularza.

8. 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.

9. Kliknij przycisk *Quit*, aby zamknąć okno i powrócić do programu Visual Studio.

10. Odszukaj w oknie *edytora kodu i tekstów* metodę *showDoubleValue*.

11. Zmień treść metody *showDoubleValue* dokładnie, tak jak na poniższym fragmencie kodu zostało to pokazane pogrubioną czcionką:

```
private void showDoubleValue()
{
    double variable;
    variable = 0.42;
    value.Text = "0.42";
}
```

12. Odszukaj w oknie *edytora kodu i tekstów* metodę *showBoolValue*.

13. Zmień treść metody *showBoolValue* dokładnie, tak jak to zostało pokazane poniżej:

```
private void showBoolValue()
{
    bool variable;
    variable = false;
    value.Text = "false";
}
```

14. Wybierz z menu *Debug* polecenie *Start Without Debugging*.

15. Wybierz z listy *Choose a data type* (Wybierz typ danych) po kolei następujące typy danych: *int*, *double* i *bool*. Za każdym razem sprawdź, czy w polu tekstowym *Sample value* wyświetlana jest właściwa wartość.

16. Kliknij przycisk *Quit*, aby przerwać działanie programu.

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, gwiazdka (*) 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 wobec wartości typu *string* lub *bool*. Tak więc pokazana poniżej instrukcja jest nieprawidłowa, ponieważ typ *string* nie obsługuje operatora minus (wynik odejmowania jednego łańcucha znakowego od drugiego byłby nieokreślony):

```
// błąd podczas 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 oczekiwanego. 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 w formie łańcuchów znakowych.

Należy również mieć świadomość tego, że typ danych rezultatu działania operatora arytmetycznego zależny jest od typu danych użytych argumentów tego operatora. Przykładowo wartością wyrażania *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 {w językach programowania, a także w krajach zachodnich część dziesiętną liczb od części całkowitej oddziela się znakiem kropki, a nie znakiem przecinka – przyp. tłum.} 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. Sytuacja staje się nieco bardzo 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* w *double*. Wynikiem tej operacji będzie więc wartość typu double (*2.5*). Wprawdzie mieszanie typów danych w jednym wyrażeniu nie jest błędem, ale jest to uznawane za złą praktykę w programowaniu.

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 *x* przez *y*. Przykładowo *9 % 2* jest równe *1*, ponieważ *9* podzielone na *2* równa się *4* z resztą *1*.

Numeryczne typy danych a wartości nieskończone

W języku C3 istnieją także inne cechy liczby, 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 – 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*. Wyjątkiem od tej reguły jest przypadek mnożenia wartości *Infinity* przez 0. Wartością wyrażenia *Infinity * 0* jest *0*, ale wartością wyrażenia *NaN * 0* jest wartość *NaN*.

Uwaga Osoby znające język C lub C++ wiedzą, że w tych językach nie można stosować operatora dzielenia modulo dla wartości typu *float* lub *double*. Język C# znosi jednak 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*.

Poznajemy operatory arytmetyczne

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

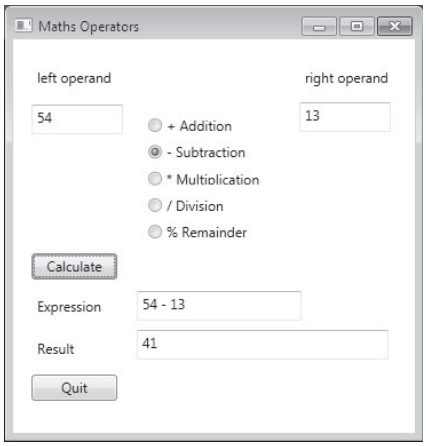
Badanie działania operatorów arytmetycznych

1. Otwórz projekt *MathsOperators* (Operatory matematyczne), znajdujący się w Twoim katalogu *Dokumenty\Microsoft Press\Visual CSharp Step By Step\Chapter 2\MathsOperators*.
2. Wybierz z menu *Debug* polecenie *Start Without Debugging*.

   Spowoduje to wyświetlenie na ekranie formularza WPF.
3. Wpisz w polu tekstowym *left operand* (lewy argument) wartość **54**.
4. Wpisz w polu tekstowym *right operand* (prawy argument) wartość **13**.

   Po wprowadzeniu wartości w polach tekstowych możliwe jest wykonanie na nich obliczeń z użyciem dowolnego operatora.
5. Kliknij przycisk – *Subtraction* (Odejmowanie), a następnie przycisk *Calculate* (Oblicz).

   Spowoduje to zmianę tekstu wyświetlanego w polu *Expression* (Wyrażenie) na 54 – 13 oraz wyświetlenie w polu *Result* (Wynik) wartości 41, tak jak to zostało pokazane na poniższym rysunku:



6. Kliknij przycisk / *Division* (Dzielenie), a następnie przycisk *Calculate*.

   Spowoduje to zmianę tekstu wyświetlanego w polu *Expression* (Wyrażenie) na 54/13 oraz wyświetlenie w polu *Result* (Wynik) wartości 4. W rzeczywistości 54/13 równa się 4,153846 (jest to ułamek dziesiętny okresowy), jednak w tym przypadku język C# wykonuje operację dzielenia liczb całkowitych – jak już wyjaśniono wcześniej, rezultatem dzielenia jednej liczby całkowitej przez drugą liczbę całkowitą jest również liczba całkowita.

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

   Spowoduje to zmianę tekstu wyświetlanego w polu *Expression* (Wyrażenie) na 54 % 13 oraz wyświetlenie w polu *Result* (Wynik) wartości 2, ponieważ resztą pozostałą z dzielenia 54 przez 13 jest właśnie 2. Jeśli wykonując działanie arytmetyczne (54 – ((54/13) * 13)), będziemy zaokrąglać w dół wszystkie wyniki pośrednie, otrzymamy 2 – mój nauczyciel matematyki byłby przerażony, słysząc, że (54/13) * 13 nie równa się 54!

8. Przetestuj inne kombinacje liczb i operatorów. Po zakończeniu tych testów kliknij przycisk *Quit*, aby powrócić do środowiska programowania Visual Studio 2010.

W kolejnym ćwiczeniu przyjrzymy się treści kodu programu *MathsOperators*.

Badanie kodu programu *MathsOperators*

1. Wyświetl w oknie widoku projektowego formularz *MainWindow.xaml* (kliknij dwukrotnie plik *MainWindow.xaml*, widniejący w panelu *Solution Explorer* (Eksplorator rozwiązań)).

2. W menu *View* (Widok), wskaż menu podrzędne *Other Windows* (Inne okna), a następnie wybierz polecenie *Document Outline* (Zarys dokumentacji).

   Spowoduje to wyświetlenie okna *Document Outline*, w którym wyświetlane będą nazwy i typy danych 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 *Window*, który stanowi formularz WPF. Jak już wspomniano w poprzednim rozdziale, formularz WPF w rzeczywistości zawiera kontrolkę typu *Grid*, w której umieszczane są pozostałe kontrolki. Rozwinięcie w oknie *Document Outline* węzła *Grid* spowoduje wyświetlenie pozostałych kontrolek. Kliknięcie wybranej kontrolki na formularzu spowoduje zaznaczenie nazwy tej kontrolki w oknie *Document Outline*. Możliwe jest także zaznaczenie kontrolki w oknie *Document Outline*, co spowoduje zaznaczenie tej kontrolki w oknie widoku projektowego. Zatrzymanie wskaźnika myszy nad wybraną kontrolką w oknie *Document Outline* spowoduje wyświetlenie obrazu tej kontrolki (oraz wszystkich jej kontrolek podrzędnych).

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* (nazwa kontrolki wyświetlana jest w nawiasach, obok samej kontrolki).

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 *TextBox* służące do wyświetlania obliczanego wyrażenia oraz wyniku obliczeń mają odpowiednio nazwy *expression* (wyrażenie) oraz *result* (wynik).
5. Zamknij okno *Document Outline*.
6. Wyświetl w oknie *edytora kodu i tekstów* zawartość pliku *MainWindow.xaml.cs*.
7. Odszukaj w oknie *edytora kodu i tekstów* metodę *subtractValues* (odejmij wartości). Metoda ta wygląda następująco:

```
private void subtractValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome;
    outcome = lhs – rhs;
    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* pola tekstowego zawiera łańcuch znakowy typu *string*, a więc przed przypisaniem tego łańcucha do zmiennej typu *int* 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*.

Czwarta instrukcja odejmuje wartość zmiennej *rhs* od wartości zmiennej *lhs* i przypisuje rezultat zmiennej *outcome*.

Piąta instrukcja dokonuje złączenia trzech łańcuchów znakowych oznaczających rodzaj wykonywanej operacji (przy użyciu operatora plus, +) 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*. Zadanie to realizuje metoda typu danych *int* o nazwie *ToString* (na łańcuch znakowy).

Metoda *ToString*

Każda klasa platformy .NET Framework ma własną metodę *ToString*. Metoda ta służy do przekształcenia obiektu danej klasy w jego reprezentację znakową. W poprzednim przykładzie metoda *ToString* będącego liczbą całkowitą obiektu *outcome* została użyta do przekształcenia całkowitoliczbowej wartości tego obiektu na równoważny mu łańcuch znakowy. Wykonanie takiej konwersji było konieczne, ponieważ wartość obiektu miała zostać wyświetlona za pomocą właściwości *Text* pola tekstowego *result*, a właściwość *Text* może zawierać jedynie łańcuchy znakowe. Tworząc własną klasę, można zdefiniować własną implementację metody *ToString* określającą sposób reprezentowania obiektów tej klasy za pomocą łańcuchów znakowych. Więcej informacji na temat tworzenia własnych klas zostanie podanych w rozdziale 7 zatytułowanym „Tworzenie i zarządzanie klasami oraz obiektami".

Kontrolowanie pierwszeństwa

Kolejnością stosowania występujących w wyrażeniu operatorów rządzą reguły *pierwszeństwa* (tzw. piorytet 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, rezultat operacji dodawania (2 + 3) stanie się lewym argumentem operatora * i w rezultacie otrzymamy wyrażenie 5 * 4, którego wartością jest 20.
- Jeśli najpierw wykonamy mnożenie, a potem dodawanie, rezultat operacji mnożenia (3 * 4) stanie się prawym argumentem operatora + i w rezultacie otrzymamy wyrażenie 2 + 12, którego wartością jest 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
```

To wyrażenie również 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 wyznaczania wartości tego wyrażenia jest ważna, ponieważ możemy otrzymać dwa różne rezultaty:

- Jeśli najpierw wykonamy dzielenie, a potem mnożenie, rezultat operacji dzielenia (4/2) stanie się lewym argumentem operatora * i w rezultacie otrzymamy wyrażenie (4/2) * 6, którego wartością jest 12.
- Jeśli najpierw wykonamy mnożenie, a potem dzielenie, rezultat operacji mnożenia (2 * 6) stanie się lewym argumentem operatora / i w rezultacie otrzymamy wyrażenie 4/(2*6), którego wartością jest 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. W podanym przykładzie najpierw zostanie wyznaczany rezultat wyrażenia 4/2, który następnie zostanie pomnożony przez 6, dając wynik końcowy 12.

Uwaga Zasady łączności pozostałych operatorów będą przedstawiane systematycznie w kolejnych rozdziałach przy okazji omawiania tych operatorów.

Zasady łączności a operator przypisania

W języku C# operatorem jest także znak równości (=) Wszystkie operatory zwracają pewną wartość wyznaczaną na podstawie wartości swoich argumentów. Dotyczy to również operatora =. Operator ten ma dwa argumenty; po wyznaczeniu wartości argumentu znajdującego się z prawej strony zostaje ona zapisana w argumencie znajdującym się z lewej strony. Wartością operatora przypisania jest wartość, która została

przypisana do jego lewego argumentu. Przykładowo w pokazanej poniżej instrukcji przypisania wartością zwróconą przez operator przypisania będzie wartość 10, ponieważ jest to także wartość, która została przypisana do zmiennej *myInt*:

```
int myInt;
myInt = 10; // wartością wyrażenia przypisania jest 10
```

Być może część z Czytelników myśli sobie, „no dobrze, wszystko to jest bardzo ładne i osobliwe, ale co z tego wynika?". No cóż. Ponieważ operator przypisania zwraca wartość, więc wartość ta może zostać użyta w innej instrukcji przypisania, np.:

```
int myInt;
int myInt2;
myInt2 = myInt = 10;
```

Wartością przypisaną zmiennej *myInt2* będzie wartość przypisana zmiennej *myInt*. Pokazana instrukcja przypisania przypisze tę samą wartość obydwu zmiennym. Taka technika jest bardzo użyteczna, jeśli zachodzi potrzeba zainicjowania tą samą wartością kilku zmiennych. Dzięki takiemu zapisowi dla każdej osoby czytającej kod źródłowy, będzie zupełnie jasne, że wszystkie zmienne muszą mieć taką samą wartość:

```
myInt5 = myInt4 = myInt3 = myInt2 = myInt = 10;
```

Z powyższego opisu można wywnioskować, że operator przypisania jest operatorem podlegającym łączeniu od prawej do lewej (jest operatorem łącznym prawostronnie). Jako pierwsze wykonywane jest przypisanie występujące skrajnie z prawej strony, a przypisana wartość ulega propagacji na kolejne zmienne od prawej do lewej. Jeśli którakolwiek z tych zmiennych miała poprzednio jakąś wartość, zostanie ona zastąpiona wartością przypisywaną.

Podobne konstrukcje należy jednak traktować z pewną ostrożnością. Jednym z częstych błędów popełnianych przez początkujących programistów w języku C# jest próba łącznego zastosowania operatora przypisania wraz z deklaracją zmiennej, np. tak jak w pokazanym poniżej przykładzie:

```
int myInt, myInt2, myInt3 = 10;
```

Jest to poprawny kod w języku C# (ponieważ zostanie skompilowany). Pokazany fragment kodu powoduje zadeklarowanie zmiennych *myInt*, *myInt2* oraz *myInt3* i zainicjowanie zmiennej *myInt3* wartością 10. Zmienne *myInt* i *myInt2* pozostaną jednak niezainicjowane. Jeśli spróbujemy użyć zmiennej *myInt* lub *myInt2* np. w wyrażeniu

```
myInt3 = myInt / myInt2;
```

kompilator wygeneruje następujące komunikaty błędu:

```
Use of unassigned local variable 'myInt'
Use of unassigned local variable 'myInt2'
(Użycie niezainicjowanej zmiennej lokalnej ‘myInt’
Użycie niezainicjowanej zmiennej lokalnej ‘myInt2)
```

Inkrementacja i dekrementacja wartości zmiennych

Chcąc zwiększyć wartość zmiennej o 1, możemy posłużyć się operatorem dodawania (+):

```
count = count + 1;
```

Operacja dodawania do zmiennej jedności jest jednak na tyle powszechna, że język C# oferuje specjalny operator służący właśnie do tego celu: jest to operator ++. W celu zwiększenia wartości zmiennej count o 1, możemy napisać następującą instrukcję:

```
count++;
```

Język C# oferuje również drugi, podobny operator (--) który pozwala na odjęcie 1 od wartości zmiennej, np.:

```
count--;
```

Operatory ++ i -- to tzw. operatory *jednoargumentowe*, co oznacza, że mają one tylko jeden argument. Operatory te mają taki sam priorytet (pierwszeństwo) oraz ten sam rodzaj łączności co jednoargumentowy operator !, który zostanie omówiony w rozdziale 4 zatytułowanym „Instrukcje wyboru".

Formy przyrostkowe i przedrostkowe

Operator inkrementacji ++ oraz operator dekrementacji -- są o tyle niezwykłymi operatorami, że mogą być umieszczane zarówno przed, jak i za zmienną. Umieszczenie symbolu operatora przed nazwą zmiennej nazywane jest *formą przedrostkową* tego operatora, a umieszczenie go za nazwą zmiennej nazywane jest *formą przyrostkową*. Poniżej podano kilka przykładów:

```
count++; // inkrementacja przyrostkowa
++count; // inkrementacja przedrostkowa
count--; // dekrementacja przyrostkowa
--count; // dekrementacja przedrostkowa
```

Użycie formy przedrostkowej lub przyrostkowej operatorów ++ i -- nie ma znaczenia dla inkrementowanej lub dekrementowanej zmiennej. Przykładowo zapis *count*++ spowoduje zwiększenie wartości zmiennej count o 1 i taki sam skutek będzie mieć zapis ++*count*. Skoro tak, można postawić pytanie, dlaczego istnieją dwa sposoby zapisywania tego samego zadania. Aby zrozumieć odpowiedź na tak postawione pytanie, musimy pamiętać, że symbole ++ i -- są operatorami, a wszystkie operatory służą do wyznacza wartości jakiegoś wyrażenia. Wartością wyrażenia *count*++ będzie ta wartość zmiennej *count*, jaką miała ona *przed* wykonaniem operacji inkrementacji, podczas gdy wartością wyrażenia ++*count* będzie ta wartość zmiennej *count*, jaką będzie ona miała *po* wykonaniu operacji inkrementacji:

```
int x;
x = 42;
Console.WriteLine(x++); // x ma teraz wartość 43, ale wypisana zostanie wartość 42
x = 42;
Console.WriteLine(++x); // x ma teraz wartość 43, i wypisana zostanie wartość 43
```

Sposób działania obydwu form operatorów można łatwo zapamiętać, patrząc na kolejność elementów (operatora i argumentu) występujących w przyrostkowej i przedrostkowej formie operatora. W wyrażeniu *x++* zmienna *x* występuje jako pierwsza, a więc wartością wyrażenia będzie wartość tej zmiennej przed jej inkrementacją. W wyrażeniu *++x* najpierw występuje operator, a więc najpierw wykonane zostanie działanie określone przez ten operator, a dopiero potem wartość zmiennej *x* (już zwiększona o 1) zostanie użyta jako wartość całego wyrażenia.

Operatory inkrementacji i dekrementacji są najczęściej używane w instrukcjach *while* oraz *do*, które zostaną omówione w rozdziale 5 zatytułowanym „Złożone instrukcje przypisania oraz instrukcje iteracji". Przy stosowaniu operatorów inkrementacji i dekrementacji w izolowany sposób należy przyjąć raczej formę przyrostkową i zachować konsekwencję w jej stosowaniu.

Deklarowanie zmiennych lokalnych o niejawnie określonym typie danych

We wcześniejszej części tego rozdziału pokazany został sposób deklarowania zmiennych polegający na określeniu typu danych i podaniu identyfikatora zmiennej, np. tak jak to pokazano poniżej:

```
int myInt;
```

Wspomnieliśmy również, że przed użyciem zmiennej należy przypisać jej jakąś wartość. Możliwe jest zadeklarowanie i zainicjowanie zmiennej w tej samej instrukcji, np.:

```
int myInt = 99;
```

Zakładając, że zmienna *myOtherInt* jest zainicjowaną zmienną typu *int*, możliwe jest nawet zastosowanie następującego zapisu:

```
int myInt = myOtherInt * 99;
```

Jak pamiętamy, wartość przypisywana zmiennej, musi być tego samego typu co ta zmienna. Przykładowo wartość typu *int* można przypisać wyłącznie zmiennej typu *int*. Kompilator języka C# może szybko ustalić typ wyrażenia użytego do zainicjowania zmiennej i poinformuje nas w razie stwierdzenia niezgodności typów. Zastępując typ danych słowem kluczowym *var*, możemy również zlecić kompilatorowi języka C#, by sam ustalił typ zmiennej na podstawie wyrażania użytego do jej zainicjowania. Na przykład:

```
var myVariable = 99;
var myOtherVariable = "Hello";
```

O występujących w powyższym przykładzie zmiennych *myVariable* i *myOtherVariable* mówimy, że są zmiennymi o *niejawnie określonym typie danych*. Słowo kluczowe *var* powoduje, że kompilator ustala typ zmiennej na podstawie typu danych wyrażenia użytego do zainicjowania tej zmiennej. W pokazanym powyżej przykładzie zmienna *myVariable* jest typu *int*, a zmienna *myOtherVariable* typu *string*. Należy tu jednak podkreślić, że taki sposób deklarowania zmiennych to tylko wygodna konwencja i po zadeklarowaniu zmiennej można przypisywać jej wyłącznie wartości tego samego typu, z jakim została ona niejawnie zadeklarowana tzn., że nie można np. w dalszej części programu przypisać zmiennej *myVariable* wartości typu *float*, *double* lub *string*. Należy również pamiętać, że słowo kluczowe *var* może być używane tylko wtedy, gdy podane zostanie wyrażenie służące do zainicjowania wartości zmiennej. Pokazana poniżej deklaracja jest nieprawidłowa i spowoduje wystąpienie błędu kompilacji:

```
var yetAnotherVariable; // Błąd – kompilator nie może określić typu danych
```

Ważne Osoby, które w przeszłości zetknęły się już z językiem Visual Basic, mogą mieć w pamięci typ danych *Variant* pozwalający na przechowywanie w zmiennej wartości dowolnego typu danych. Należy w tym miejscu stanowczo podkreślić, że programując w języku C#, należy zapomnieć o znanym z języka Visual Basic typie danych *Variant*. Wprawdzie słowa kluczowe *var* i *Variant* wyglądają podobnie, ale oznaczają zupełnie coś innego. Deklarując zmienną w języku C# przy użyciu słowa kluczowego *var* typ wartości, które będzie można przypisywać tej zmiennej, nie będzie mógł być inny niż typ danych użyty do jej zainicjalizowania.

Być może osoby będące purystami mogą w tym miejscy zazgrzytać zębami i zastanawiać się, jak projektanci tak uporządkowanego języka, jakim jest język C#, mogli w ogóle dopuścić do przeniknięcia do niego takiej funkcjonalności jak deklarowanie zmiennych przy użyciu słowa kluczowego *var*. Ostatecznie ta funkcjonalność wydaje się niczym więcej jak tylko wymówką dla wyjątkowego lenistwa ze strony programisty, utrudniającą zrozumienie działania programu lub zlokalizowanie ewentualnych błędów (a nawet z łatwością sama może stać się źródłem nowych błędów). Proszę mi jednak uwierzyć, że słowo kluczowe *var* odgrywa bardzo ważną rolę w języku C#, o czym przekonamy się wkrótce w kolejnych rozdziałach tej książki. Na razie z wyjątkiem sytuacji, w których niejawne określanie typu danych jest koniecznością, pozostaniemy jednak przy jawnym deklarowaniu typu danych dla zmiennych.

W tym rozdziale pokazaliśmy, w jaki sposób tworzyć zmienne i posługiwać się nimi, a także przedstawiliśmy kilka typów danych, jakich można używać do deklarowania zmiennych w języku C#. Przedstawione zostało pojęcie identyfikatorów, pokazany został sposób tworzenia wyrażeń przy użyciu różnych operatorów, a także omówiono reguły pierwszeństwa (priorytety) i łączności operatorów określające sposób obliczania wartości wyrażeń.

- Jeśli zamierzasz przejść do kolejnego rozdziału,

  pozostaw uruchomiony program Visual Studio 2010 i przejdź do rozdziału 3.
- Jeśli chcesz na tym zakończyć pracę z programem Visual Studio 2010,

  wybierz z menu *File* (Plik) polecenie *Exit* (Wyjście). Jeśli spowoduje to wyświetlenie okna dialogowego z pytaniem o zapisanie otwartego projektu, należy zapisać ten projekt i kliknąć przycisk *Yes*.

Krótkie podsumowanie rozdziału 2

W celu	Wykonaj następujące czynności
Zadeklarowania zmiennej	Wpisz nazwę typu danych, a po niej nazwę zmiennej zakończoną znakiem średnika. Np.: `int outcome;`
Zmiany wartości zmiennej	Wpisz nazwę zmiennej z lewej strony operatora przypisania, a z prawej strony zakończone średnikiem wyrażenie wyznaczające nową wartość zmiennej. Np.: `outcome = 42;`
Przekształcenia łańcucha znakowego (wartości typu *string*) na liczbę całkowitą (wartość typu *int*)	Wywołaj metodę *System.Int32.Parse*. Np.: `System.Int32.Parse("42");`
Wymuszenia zastosowania innej niż domyślna kolejności wykonywania operatorów	Użyj w wyrażeniu nawiasów wymuszających pożądaną kolejność obliczeń. Np.: `(3 + 4) * 5`
Przypisania tej samej wartości kilku zmiennym	Użyj instrukcji przypisania, w której występować będą wszystkie zmienne. Np.: myInt94 = myInt3 = myInt2 = myInt = 10;
Zwiększenia lub zmniejszenia wartości zmiennej o jeden	Użyj operatora ++ lub --. Np.: `count++;`

ROZDZIAŁ 3

Tworzenie metod i stosowanie zasięgów zmiennych

Po ukończeniu tego rozdziału Czytelnik będzie potrafił:

- deklarować i wywoływać metody,
- przekazywać informacje do metody,
- zwracać informację z metody,
- definiować zasięg lokalny oraz zasięg klasy,
- używać zinterowanego debugera do krokowego wykonywania metod.

Z rozdziału 2 zatytułowanego „Zmienne, operatory i wyrażenia" dowiedzieliśmy się, jak deklarować zmienne, jak tworzyć wyrażania przy użyciu operatorów oraz jak reguły pierwszeństwa i łączności operatorów wpływają na kolejność operacji wykonywanych podczas wyznaczania wartości wyrażenia. Z tego rozdziału dowiemy się więcej na temat metod. Pokażemy, w jaki sposób za pomocą argumentów i parametrów można przekazywać informacje do metod oraz w jaki sposób metody mogą zwracać informacje za pomocą instrukcji *return*. Na zakończenie pokażemy, w jaki sposób można śledzić wykonywanie metod przy użyciu zintegrowanego debugera ze środowiska Microsoft Visual Studio 2010. Informacje te mogą być przydatne w razie konieczności prześledzenia sposobu wykonywania utworzonych metod, gdy ich działanie nie jest dokładnie takie, jak oczekiwano.

Tworzenie metod

Metoda to nazwana sekwencja instrukcji. Osoby mające już doświadczenie w programowaniu w takich językach programowania jak C lub Microsoft Visual Basic przekonają się wkrótce, że metody są bardzo podobne do funkcji lub podprocedur. Metoda składa się z nazwy oraz z ciała. Nazwa metody powinna być identyfikatorem wskazującym ogólne przeznaczenie danej metody (np. metoda *obliczPodatekDochodowy*). Ciało metody składa się z instrukcji, które będą wykonywane po wywołaniu danej metody. Ponadto metody mogą otrzymywać pewne dane do przetworzenia i mogą również zwracać pewne informacje, które zwykle są rezultatem tego przetwarzania. Metody to podstawowy i zarazem bardzo potężny mechanizm.

Deklarowanie metody

W języku C# składnia deklaracji metody wygląda następująco:

```
typZwracany nazwaMetody (listaParametrów)
{
    // w tym miejscu należy umieścić instrukcje tworzące ciało metody
}
```

- Identyfikator *typZwracany* jest nazwą typu danych i określa rodzaj informacji, jaki zostanie zwrócony jako rezultat działania metody. Może to być dowolny typ danych, np. *int* lub *string*. W przypadku metody, która nie zwraca żadnej wartości, zamiast typu danych konieczne jest użycie słowa kluczowego *void*.
- Identyfikator *nazwaMetody* to nazwa, która będzie używana do wywoływania metody. Nazwy metod podlegają takim samym regułom tworzenia identyfikatorów jak nazwy zmiennych. Przykładowo nazwa *dodajWartosci* jest poprawną nazwą metody, ale *dodaj$Wartosci* już nie. Tworząc nazwy metod, należy stosować tzw. *konwencję wielbłądzią* (ang. *camelCase*), np. *wyswietlKlienta*.
- Element *listaParametrów* jest opcjonalny i opisuje typy oraz nazwy argumentów, za pomocą których przekazywane są do metody informacje, które mają zostać przetworzone przez tę metodę. Lista parametrów zapisywana jest pomiędzy znakiem nawiasu otwierającego i znakiem nawiasu zamykającego w taki sam sposób, w jaki deklaruje się zmienne, podając najpierw nazwę typu danych, a po niej nazwę parametru. Jeśli tworzona metoda ma mieć dwa parametry lub więcej, należy oddzielić je od siebie znakiem przecinka.
- Instrukcje tworzące *ciało metody* to linie kodu, które zostaną wykonane po wywołaniu danej metody. Instrukcje te umieszczone są pomiędzy znakami nawiasów klamrowych ({ }).

Ważne Programiści znający język C, C++ lub Microsoft Visual Basic powinni zwrócić uwagę na fakt, że język C# nie obsługuje metod globalnych. Wszystkie tworzone metody muszą być zapisane wewnątrz pewnej klasy, gdyż w przeciwnym wypadku zostanie zgłoszony błąd kompilacji.

Poniżej podana została definicja metody o nazwie *dodajWartosci*, która zwraca wynik typu *int* i ma dwa parametry, również typu *int* o nazwach *lewyParamter* i *prawyParametr*:

```
int dodajWartosci(int lewyParamter, int prawyParametr)
{
    // ...
    // w tym miejscu znajdują się instrukcje tworzące ciało metody
    // ...
}
```

Uwaga Typ wartości zwracanej przez metodę oraz typy jej parametrów muszą zostać określone w sposób jawny. Nie można w tym celu posłużyć się słowem kluczowym *var*.

Poniżej pokazana została definicja metody o nazwie *pokazWynik*, która nie zwraca żadnej wartości i ma tylko jeden parametr typu *int* o nazwie *odpowiedz*:

```
void pokazWynik(int odpowiedz)
{
    // ...
}
```

Należy tu zwrócić uwagę na użycie słowa kluczowego *void* oznaczającego, że dana metoda nie zwraca żadnego rezultatu.

Ważne Programiści znający język Visual Basic powinni zauważyć, że język C# nie używa różnych słów kluczowych do rozróżniania metod zwracających pewną wartość (funkcji) i metod nie zwracających żadnej wartości (procedur lub podprocedur). W języku C# konieczne jest zawsze określenie typu zwracanej wartości lub użycie słowa kluczowego *void*.

Zwracanie danych przez metodę

Jeśli chcemy, by metoda zwracała jakąś informację (tzn. gdy typem danych tej metody jest typ różny od *void*), konieczne jest umieszczenie w ciele tej metody instrukcji *return* (instrukcji powrotu), która kończy proces przetwarzania danych przez metodę. Instrukcja powrotu składa się ze słowa kluczowego *return*, po którym następuje wyrażenie określające zwracaną wartość zakończone znakiem średnika. Typ danych tego wyrażenia musi być taki sam jak typ danych określony w deklaracji metody. Przykładowo jeśli metoda zwraca wartość typu *int*, instrukcja *return* również musi zwracać wartość typu *int*, gdyż w przeciwnym razie dojdzie do błędu podczas kompilacji programu. Poniżej pokazany został przykład metody zawierającej instrukcję *return*:

```
int dodajWartosci(int lewyParamter, int prawyParametr)
{
    // ...
    return lewyParamter + prawyParametr;
}
```

Instrukcja *return* zwykle znajduje się na końcu metody, ponieważ powoduje zakończenie jej działania i zwrócenie kontroli do instrukcji, która wywołała daną metodę, co zostanie omówione dokładniej w dalszej części tego rozdziału. Wszelkie instrukcje znajdujące się za instrukcją *return* nie zostaną wykonane (ale umieszczenie instrukcji po instrukcji *return* spowoduje wygenerowanie przez kompilator odpowiedniego ostrzeżenia).

Jeśli nie chcemy, by tworzona metoda zwracała jakiekolwiek informacje (tzw. jeśli jej typem jest typ *void*), możemy posłużyć się odmianą instrukcji *return* powodującą natychmiastowe zakończenie działania metody. W tym celu należy wpisać zakończone znakiem średnika samo słowo kluczowe *return*. Np.:

```
void pokazWynik(int odpowiedz)
{
    // Wyświetlenie odpowiedzi (wartości zmiennej odpowiedz)
    ...
    return;
}
```

Jeśli metoda nie zwraca żadnej wartości, możliwe jest również całkowite pominięcie instrukcji *return*, ponieważ działanie metody zakończy się automatycznie po napotkaniu znaku klamrowego nawiasu zamykającego kończącego ciało metody. Wprawdzie takie postępowanie jest dość powszechne, ale nie zawsze stanowi przejaw dobrej praktyki w programowaniu. W zamieszczonym poniżej ćwiczeniu zapoznamy się ze zmienioną wersją projektu *MathsOperators* (Operatory arytmetyczne), z którym zetknęliśmy się już wcześniej w rozdziale 2. Obecna wersja została ulepszona poprzez staranne dobranie kilku niewielkich metod.

Analizowanie definicji metod

1. Jeśli program Visual Studio 2010 nie został uruchomiony już wcześniej, uruchom go teraz.
2. Otwórz projekt *Methods* znajdujący się w katalogu *Dokumenty\Microsoft Press\Visual CSharp Step By Step\Chapter 3\Methods*.
3. Wybierz z menu *Debug* polecenie *Start Without Debugging*.

   Spowoduje to zbudowanie aplikacji i jej uruchomienie przez program Visual Studio 2010.
4. Zapoznaj się z aplikacją i przypomnij sobie sposób jej działania, a następnie kliknij przycisk *Quit* (Wyjście).
5. Wyświetl w oknie *edytora kodu i tekstów* zawartość pliku *MainWindow.xaml.cs*.
6. Odszukaj w oknie *edytora kodu i tekstów* metodę *addValues* (dodaj wartości).

   Metoda ta wygląda następująco:

   ```
   private int addValues(int leftHandSide, int rightHandSide)
   {
       expression.Text = leftHandSide.ToString() + " + " + rightHandSide.ToString();
       return leftHandSide + rightHandSide;
   }
   ```

   Metoda *addValues* zawiera dwie instrukcje. Pierwsza z nich wyświetla w znajdującym się na formularzu polu tekstowym *expression* (wyrażenie) treść wyrażenia, którego wartość zostanie obliczona w drugiej instrukcji. Wartości parametrów *leftHandSide* i *rightHandSide* zostają przekształcone na łańcuchy znakowe (przy użyciu metody

ToString, z którą zetknęliśmy się już w rozdziale 2) i połączone ze sobą oraz z umieszczoną w środku, znakową reprezentacją operatora dodawania (+).

Druga instrukcja dodaje do siebie za pomocą operatora + wartości typu *int* przechowywane w zmiennych *leftHandSide* i *rightHandSide* i zwraca rezultat tej operacji. Jak pamiętamy, w wyniku dodawania dwóch wartości typu *int* otrzymamy również wartość typu *int*, a więc typem danych metody *addValues* jest typ *int*.

Jeśli przyjrzymy się metom *subtractValues*, *multiplyValues*, *divideValues* oraz *remainderValues* przekonamy się, że są one zbudowane w podobny sposób.

7. Odszukaj w oknie *edytora kodu i teksów* metodę *showResult* (pokaż wynik).

Metoda *showResult* wygląda następująco:

```
private void showResult(int answer)
{
    result.Text = answer.ToString();
}
```

Metoda ta składa się z jednej instrukcji, wyświetlającej w polu tekstowym *result* (wynik) znakową reprezentację wartości parametru *answer* (odpowiedź) Metoda ta nie zwraca żadnej wartości, a więc jej typem jest typ *void*.

Wskazówka Nie ma żadnej minimalnej długości metody. Jeśli utworzenie metody pomoże w uniknięciu powtórzeń i sprawi, że kod źródłowy stanie się bardziej zrozumiały, taka metoda będzie użyteczna, niezależnie od tego jak będzie ona krótka. Nie istnieje także maksymalna długość metody, ale zwykle należy dążyć do tego, by metody były na tyle małe, żeby realizowały jakieś jedno, proste zadanie. Jeśli kod metody nie mieści się w całości na ekranie komputera, warto rozważyć podzielenie jej na kilka mniejszych metod w celu poprawienia czytelności kodu.

Wywoływanie metod

Metody istnieją po to, by je wywoływać! Robi się to za pomocą ich nazw, powierzając im wykonanie zadań. Jeśli metoda wymaga do działania pewnych informacji (określonych za pomocą parametrów), konieczne jest podanie tych informacji. Jeśli metoda zwraca pewną informację (określoną przez jej typ danych), należy zadbać o to, by w jakiś sposób odebrać tę informację.

Określanie składni wywołania metody

W języku C# składnia wywołania metody ma następującą postać:

```
rezultat = nazwaMetody(listaArgumentów)
```

- Identyfikator *nazwaMetody* musi mieć dokładnie taką samą formę jak nazwa wywoływanej metody. Należy przy tym pamiętać, że w języku C# są rozróżniane małe i wielkie litery.