

Jacek Matulewski

Visual Studio 2013

Podręcznik programowania w C# z zadaniami



Ucz się, projektuj, zarabiaj!

- Poznaj język C# 5.0 i platformę .NET 4.5.1
 - podstawy nowoczesnego projektowania aplikacji Windows.
- Dowiedz się, jak projektować aplikacje Windows Forms i efektywnie używać kontrolek.
- Wybierz optymalny sposób przechowywania informacji w bazie danych w aplikacjach dla platformy .NET.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiekolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich.

Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Ewelina Burska

Projekt okładki: Studio Gravite/Olsztyn

Obarek, Pokoński, Pazdrijowski, Zaprucki

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie/vs12pa_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-0027-9

Copyright © Helion 2014

Printed in Poland.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Książkę dedykuję moim ukochanym dzieciom — Karolinie i Bartkowi, oraz żonie — Kasi

Spis treści

Wstęp	11
Część I Język C# i platforma .NET	13
Rozdział 1. Pierwsze spotkanie ze środowiskiem Visual Studio i językiem C#	15
Klasa Console	15
Projekt aplikacji konsolowej	15
Drukowanie napisów w konsoli i podpowiadanie kodu	18
Czekanie na akceptację użytkownika	19
Odczytywanie danych z klawiatury	20
Korzystanie z bibliotek DLL. Komunikat „okienkowy” w aplikacji konsolowej	21
Informacje o środowisku aplikacji	23
Podstawowe informacje o systemie i profilu użytkownika	23
Katalogi specjalne zdefiniowane w bieżącym profilu użytkownika	24
Odczytywanie zmiennych środowiskowych	25
Lista dysków logicznych	26
Rozdział 2. O błędach i ich tropieniu	27
Program z błędem logicznym — pole do popisu dla debugera	27
Kontrolowane uruchamianie aplikacji w Visual C#	28
Śledzenie wykonywania programu krok po kroku (F10 i F11)	28
Run to Cursor (Ctrl+F10)	30
Breakpoint (F9)	30
Okna Locals i Watch	31
Stan wyjątkowy	33
Zgłaszanie wyjątków	33
Przechwytywanie wyjątków w konstrukcji try..catch	35
Wymuszenie kontroli zakresu zmiennych	37
Rozdział 3. Język C# 5.0	39
Platforma .NET	40
Środowisko uruchomieniowe	40
Kod pośredni i podwójna komplikacja	40
Wersje	41
Skróty, które warto poznać	43
Podstawowe typy danych	44
Deklaracja i zmiana wartości zmiennej	44
Typy liczbowe oraz typ znakowy	45

O określaniu typu zmiennej przy inicjacji (pseudotyp var)	47
Operatory	47
Konwersje typów podstawowych	49
Operatory is i as	50
Łańcuchy	51
Typ wyliczeniowy	54
Leniwe inicjowanie zmiennych	55
Metody	56
Przeciążanie metod	57
Domyślne wartości argumentów metod — argumenty opcjonalne	58
Argumenty nazwane	59
Wartości zwracane przez metody	59
Zwracanie wartości przez argument metody	59
Delegacje i zdarzenia	61
Wyrażenia lambda	64
Typy wartościowe i referencyjne	66
Nullable	67
Pudełkowanie	68
Typy dynamiczne	69
Sterowanie przepływem	72
Instrukcja warunkowa if..else	72
Instrukcja wyboru switch	73
Pętle	74
Wyjątki	75
Dyrektywy preprocesora	77
Kompilacja warunkowa — ostrzeżenia	78
Definiowanie stałych preprocesora	78
Bloki	79
Atrybuty	80
Kolekcje	81
„Zwykłe” tablice	81
Pętla foreach	83
Sortowanie	84
Kolekcja List	85
Kolekcja SortedList i inne słowniki	87
Kolejka i stos	88
Tablice jako argumenty metod oraz metody z nieokreślona liczbą argumentów	89
Słowo kluczowe yield	90
Nowa forma inicjacji obiektów i tablic	92
Caller Information	93
Rozdział 4. Programowanie obiektowe w C#	95
Przykład struktury (Ulamek)	96
Przygotowywanie projektu	96
Konstruktor i statyczne obiekty składowe	98
Modyfikatory const i readonly	99
Pierwsze testy	99
Konwersje na łańcuch (metoda ToString) i na typ double	100
Metoda upraszczająca ułamek	101
Właściwości	102
Domyślnie implementowane właściwości (ang. auto-implemented properties)	104
Operatory arytmetyczne	105
Operatory porównania oraz metody Equals i GetHashCode	106
Operatory konwersji	108

Implementacja interfejsu (na przykładzie IComparable)	109
Definiowanie typów parametrycznych	110
Definiowanie typów ogólnych	111
Określanie warunków, jakie mają spełniać parametry	113
Implementacja interfejsów przez typ ogólny	114
Definiowanie aliasów	115
Typy ogólne z wieloma parametrami	116
Rozszerzenia	117
Typy anonimowe	119
Dziedziczenie	119
Klasy bazowe i klasy potomne	120
Nadpisywanie a przesłanianie metod	123
Klasy abstrakcyjne	124
Metody wirtualne	126
Polimorfizm	127
Konstruktory a dziedziczenie	129
Singleton	131
Interfejsy	134
Rozdział 5. Biblioteki DLL	137
Tworzenie zarządzanej biblioteki DLL	138
Dodawanie do aplikacji referencji do biblioteki DLL	139
Przenośne biblioteki DLL	140
Rozdział 6. Testy jednostkowe	143
Projekt testów jednostkowych	144
Przygotowania do tworzenia testów	144
Pierwszy test jednostkowy	145
Uruchamianie testów	146
Dostęp do prywatnych pól testowanej klasy	147
Testowanie wyjątków	148
Kolejne testy weryfikujące otrzymane wartości	148
Test ze złożoną weryfikacją	149
Powtarzane wielokrotnie testy losowe	151
Niepowodzenie testu	152
Nieuniknione błędy	154
Rozdział 7. Elementy programowania współbieżnego	159
Równoległa pętla for	159
Przerywanie pętli	161
Programowanie asynchroniczne.	
Modyfikator async i operator await (nowość języka C# 5.0)	162
Część II Projektowanie aplikacji Windows Forms	169
Rozdział 8. Pierwszy projekt aplikacji Windows Forms	171
Projektowanie interfejsu aplikacji	172
Tworzenie projektu	172
Dokowanie palety komponentów Toolbox	174
Tworzenie interfejsu za pomocą komponentów Windows Forms	175
Przełączanie między podglądem formy a kodem jej klasy	176
Analiza kodu pierwszej aplikacji Windows Forms	176
Metody zdarzeniowe	182
Metoda uruchamiana w przypadku wystąpienia zdarzenia kontrolki	182
Testowanie metody zdarzeniowej	183

Przypisywanie istniejącej metody do zdarzeń komponentów	185
Edycja metody zdarzeniowej	185
Modyfikowanie właściwości komponentów	186
Wywoływanie metody zdarzeniowej z poziomu kodu	186
Reakcja aplikacji na naciskanie klawiszy	187
Rozdział 9. Przegląd komponentów biblioteki Windows Forms	189
Notatnik.NET	189
Projektowanie interfejsu aplikacji i menu główne	189
Okna dialogowe i pliki tekstowe	196
Edycja i korzystanie ze schowka	204
Drukowanie	205
Elektroniczna kukulka	214
Ekran powitalny (splash screen)	214
Przygotowanie ikony w obszarze powiadamiania	217
Odtwarzanie pliku dźwiękowego	220
Ustawienia aplikacji	222
Dywany graficzny	225
Rozdział 10. Przeciągnij i upuść	231
Podstawy	231
Interfejs przykładowej aplikacji	232
Inicjacja procesu przeciągania	233
Akceptacja upuszczenia elementu	235
Reakcja na upuszczenie elementu	236
Czynności wykonywane po zakończeniu procesu przenoszenia i upuszczania	237
Przenoszenie elementów między różnymi aplikacjami	238
Zagadnienia zaawansowane	238
Opóźnione inicjowanie procesu przenoszenia	238
Przenoszenie wielu elementów	241
Przenoszenie plików	242
Część III Dane w aplikacjach dla platformy .NET	245
Rozdział 11. LINQ	247
Operatory LINQ	247
Pobieranie danych (filtrowanie i sortowanie)	249
Analiza pobranych danych	250
Wybór elementu	250
Weryfikowanie danych	251
Prezentacja w grupach	251
Łączenie zbiorów danych	252
Łączenie danych z różnych źródeł w zapytaniu LINQ — operator join	252
Możliwość modyfikacji danych źródła	253
Rozdział 12. Przechowywanie danych w plikach XML. LINQ to XML	255
Podstawy języka XML	255
Deklaracja	255
Elementy	256
Atrybuty	256
Komentarze	256
LINQ to XML	257
Tworzenie pliku XML za pomocą klas XDocument i XElement	257
Pobieranie wartości z elementów o znanej pozycji w drzewie	259
Odwzorowanie struktury pliku XML w kontrolce TreeView	261

Przenoszenie danych z kolekcji do pliku XML	262
Zapytania LINQ, czyli tworzenie kolekcji na bazie danych z pliku XML	263
Modyfikacja pliku XML	264
Rozdział 13. Baza danych SQL Server w projekcie Visual Studio	267
Odwarzorowanie obiektowo-relacyjne	267
Szalenie krótki wstęp do SQL	269
Select	269
Insert	270
Delete	270
Projekt aplikacji z bazą danych	270
Dodawanie bazy danych do projektu aplikacji	270
Łańcuch połączenia (ang. connection string)	271
Dodawanie tabeli do bazy danych	272
Edycja danych w tabeli	274
Druga tabela	274
Procedura składowana — pobieranie danych	276
Procedura składowana — modyfikowanie danych	276
Procedura składowana — dowolne polecenia SQL	277
Widok	277
Rozdział 14. LINQ to SQL	279
Klasa encji	280
Pobieranie danych	281
Prezentacja danych w siatce DataGridView	283
Aktualizacja danych w bazie	283
Modyfikacje istniejących rekordów	284
Dodawanie i usuwanie rekordów	285
Inne operacje	286
Wizualne projektowanie klasy encji	286
O/R Designer	287
Współpraca z kontrolkami tworzącymi interfejs aplikacji	290
Korzystanie z widoków	291
Łączanie danych z dwóch tabel — operator join	292
Relacje (Associations)	292
Korzystanie z procedur składowanych	294
Pobieranie danych za pomocą procedur składowanych	294
Modyfikowanie danych za pomocą procedur składowanych	295
Rozdział 15. Kreator źródeł danych	297
Kreator źródła danych	297
Zautomatyzowane tworzenie interfejsu użytkownika	300
Prezentacja tabeli w siatce	300
Klasa BindingSource	301
Sortowanie	301
Filtrowanie	302
Prezentacja rekordów tabeli w formularzu	302
Dwie siatki w układzie master-details	304
Rozdział 16. Tradycyjne ADO.NET (DataSet)	307
Konfiguracja źródła danych DataSet	307
Edycja klasy DataSet i tworzenie relacji między tabelami	309
Podgląd danych udostępnianych przez komponent DataSet	310
Prezentacja danych w siatce	311
Zapisywanie zmodyfikowanych danych	314

Prezentacja danych w formularzu	316
Sortowanie i filtrowanie	319
Odczytywanie z poziomu kodu wartości przechowywanych w komórkach tabeli	319
Zapytania LINQ do danych z DataSet	322
Rozdział 17. Entity Framework	323
Tworzenie modelu danych EDM dla istniejącej bazy danych	324
Użycie klasy kontekstu z modelu danych EF	327
LINQ to Entities	329
Prezentacja i edycja danych w siatce	330
Asynchroniczne wczytywanie danych	332
Użycie widoku i procedur składowanych	333
Połączenie między tabelami	334
Tworzenie źródła danych	336
Automatyczne tworzenie interfejsu	338
Edycja i zapis zmian	341
Część IV Dodatki	343
Zadania	345
Skorowidz	355

Wstęp

Niniejsza książka przeznaczona jest dla osób, które może mają już doświadczenie w programowaniu, ale dopiero zaczynają swoją przygodę z platformą .NET, chcą się nauczyć programować w języku C# lub chcą poszerzyć swoją wiedzę na ten temat. Przygotowując książkę, korzystałem z Visual Studio 2012 i 2013 w wersji Ultimate (taka wersja dostępna jest w przeznaczonym dla studentów programie DreamSpark Premium, w którym Microsoft udostępnia za darmo oprogramowanie potrzebne do nauki). Znacząca część kodu, szczególnie z pierwszej i drugiej części książki, może być jednak odtworzona w darmowej wersji Express.

Informacje umieszczone w tej książce są aktualne, dotyczą najnowszej wersji platformy .NET i bieżących wersji Visual Studio. Zastanawiać może jednak Czytelnika, dlaczego w drugiej części korzystam z biblioteki Windows Forms zamiast z nowszej biblioteki WPF. Zapewne to kwestia przyzwyczajenia; poznalem i polubiłem Windows Forms i trudno mi z niej zrezygnować na rzecz mniej intuicyjnej, przynajmniej moim zdaniem, choć zdecydowanie potężniejszej i bardziej elastycznej biblioteki WPF. Ale nie tylko ja mam taką opinię. Proszę zwrócić uwagę, że spora, jeżeli nie przeważająca część nowych projektów nadal korzysta z tej tradycyjnej biblioteki. Warto też odnotować, że w platformie Mono w ogóle zrezygnowano z przenoszenia nowej biblioteki WPF. Uznano to po prostu za nieopłacalne. Do tego Microsoft ogłosił niedawno, że więcej wersji WPF już nie wyda (po prawdzie rzeczywisty rozwój Windows Forms zakończył się już dawno temu).

Książka ta jest w rzeczywistości kolejnym wydaniem książki, która w różnych „konfiguracjach” ukazuje się już od 2004 roku (zobacz tabela 2.1 w rozdziale 2.). Tym razem bezpośredniem impulsem do przygotowania jej nowej edycji było uruchomienie na UMK pod patronatem Microsoft nowej sekcji *Podyplomowego Studium Programowania i Zastosowania Komputerów* o nazwie *Projektowanie i tworzenie aplikacji dla platformy .NET*. To pierwsze studium w Polsce, któremu patronuje Microsoft. Materiały zawarte w tej książce były wykładowane na przedmiotach: język C# 5.0 i aplikacje Windows na pulpit.

Więcej niż wskazywałby na to wprowadzający charakter książki, jest w niej informacji o danych w aplikacjach. A to dlatego, że każdy programista przedżej czy później napotka problem przechowywania danych. Należy jednak zastrzec, że omawiane przeze

mnie zagadnienia nie należą do bardzo zaawansowanych. Zamiast tego staram się przedstawić jak najszerszą panoramę technologii służących do przechowywania danych, zarówno tych najnowszych, jak Entity Framework, jak i nieco starszych. Chodziło mi przede wszystkim o to, aby po zapoznaniu się z trzecią częścią książki Czytelnik był w stanie dobrać taki sposób przechowywania danych, który będzie optymalny dla jego projektu.

Ważną częścią książki są umieszczone na końcu zadania. Powtarzanie przedstawionych w książce zadań pozwala oswoić się z językiem C# i platformą .NET, ale dopiero samodzielne rozwiązywanie zadań da Czytelnikowi okazję do prawdziwej nauki. Przy rozwiązywaniu zadań można korzystać z pomocy w postaci wyszukiwarki internetowej, ale naprawdę warto najpierw spróbować zmierzyć się z nimi samodzielnie.

Jacek Matulewski,
Toruń, styczeń 2014

Część I

Język C# i platforma .NET

Rozdział 1.

Pierwsze spotkanie ze środowiskiem Visual Studio i językiem C#

Klasa Console

W pierwszej części książki poznasz język C# i najczęściej używane klasy platformy .NET. W tym celu będziemy tworzyć proste aplikacje bez bogatego interfejsu, korzystające z konsoli do wyświetlania komunikatów.

Projekt aplikacji konsolowej

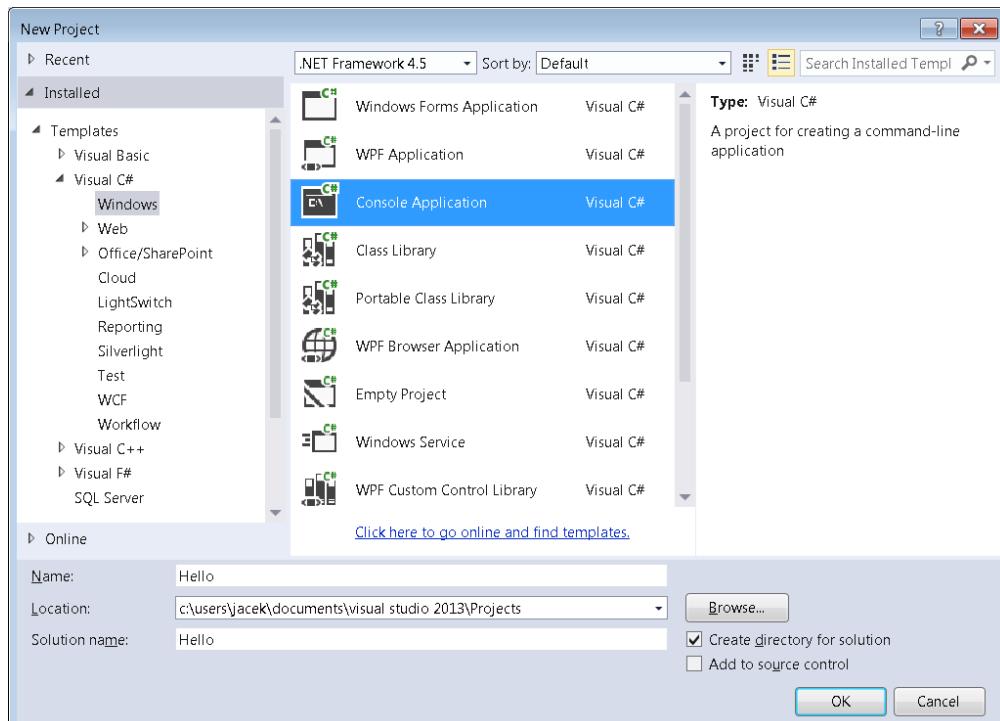
Stwórzmy projekt aplikacji konsolowej.

1. W menu *File* środowiska Visual Studio wybierz pozycję *New*, a następnie *Project...* lub naciśnij kombinację klawiszy *Ctrl+Shift+N*.
2. W oknie *New Project* (rysunek 1.1) zaznacz pozycję *Console Application*.
3. W dolnej części okna znajduje się pole edycyjne *Name* pozwalające na nadanie nowemu projektowi nazwy. Nazwij go *Hello*.



Zwróć uwagę na pozycję *Location* — wskazuje ona katalog, w którym zapisany zostanie projekt. Pliki projektu zostaną zapisane na dysku natychmiast po jego utworzeniu.

4. Kliknij *OK*.



Rysunek 1.1. Tworzenie aplikacji na podstawie szablonu *Console Application*

Po utworzeniu projektu powstał katalog *c:\Users\[Użytkownik]\Documents\Visual Studio 2013\Projects>Hello*, do którego zostały zapisane pliki rozwiązań *Hello.sln* i *Hello.suo*. Pozostałe pliki projektu, w tym pliki z kodem źródłowym, zostały umieszczone w podkatalogu *Hello*. Są to: plik projektu *Hello.csproj*, plik z informacjami o aplikacji *Properties\AssemblyInfo.cs* oraz plik źródłowy *Program.cs*, który powinien pojawić się w edytorze kodu Visual Studio. W aplikacji konsolowej nie ma możliwości przełączania na widok projektowania z tej prostej przyczyny, że klasa znajdująca się w edytowanym pliku nie implementuje okna, które moglibyśmy wizualnie projektować. Zasadniczy kod aplikacji znajduje się w pliku *Program.cs* (listing 1.1). W kilku pierwszych liniach deklarowane jest użycie przestrzeni nazw *System*, *System.Collections.Generic*, *System.Linq*, *System.Text* oraz *System.Threading.Tasks*. Następnie jest otwierana przestrzeń nazw *Hello*, w której znajduje się definicja klasy *Program* zawierającej tylko jedną metodę statyczną *Main*.

Listing 1.1. Kod utworzony przez Visual C#

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

namespace Hello
{
    class Program
```

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



Wskazówka

W Visual Studio 2013 w edytorze kodu nad każdą metodą i klasą widoczna jest liczba jej użyć w pozostałej części kodu. W tej chwili nad metodą Main widoczne powinno być „0 references”, ale gdy odwołań będzie więcej, można kliknąć tę informację, aby zobaczyć numery linii, w których się znajdują.

Kod w pliku *Program.cs* rozpoczyna się od zadeklarowania wykorzystywanych przestrzeni nazw (ang. *namespace*) — służy do tego słowo kluczowe `using` i następująca po nim nazwa przestrzeni nazw. Visual C# zadeklarował użycie przestrzeni `System` i kilku jej podprzestrzeni. Dzięki temu zabiegowi użycie klas znajdujących się w obrębie zadeklarowanych przestrzeni nie będzie wymagało jawnego wymieniania przestrzeni nazw przed nazwą klasy, a więc np. zamiast odwoływać się do pełnej nazwy klasy `System.Console`, możemy poprzestać jedynie na `Console`. Przestrzenie nazw mogą tworzyć hierarchiczną strukturę. Zadeklarowanie korzystania z przestrzeni nazw nie pociąga za sobą automatycznego dostępu do jej „podprzestrzeni” lub „nadprzestrzeni”. Każdą trzeba deklarować osobno. Blok poleceń `using` można „zwinąć” do jednej linii, korzystając z pola ze znakiem minus znajdującego się z lewej strony jego pierwszej linii. Podobnie można postąpić z liniami w obrębie przestrzeni nazw, klasy czy metody. Ułatwia to edycję kodu szczególnie wtedy, gdy już się bardzo rozrośnie.

Za deklaracją używanych w pliku *Program.cs* przestrzeni nazw słowo kluczowe `namespace` otwiera naszą własną przestrzeń nazw o nazwie `Hello`. W niej znajduje się klasa `Program` zawierająca jedną metodę o nazwie `Main`.



Wskazówka

C# to język, którego kod jest podobny do C++ i Javy. Większość słów kluczowych pochodzi wprost z języka C++, tak więc żadna osoba znajdująca C++ lub Javę nie powinna mieć z poznaniem C# większych problemów. Warto zwrócić uwagę na to, że w C# nie ma plików nagłówkowych. Podobnie jak w Javie metody klasy definiowane są — posłużę się terminologią C++ — *inline*.

Platforma .NET stara się wywołać statyczną metodę `Main` którejś z klas zdefiniowanych w naszej aplikacji. Stanowi ona *punkt wejściowy* aplikacji (ang. *entry point*). W tej chwili metoda `Main` jedynej klasy `Program` nie zawiera żadnych instrukcji, gdybyśmy więc naciśnieli `F5` lub `Ctrl+F5`, uruchamiając aplikację, zostanie ona zakończona tuż po uruchomieniu (w tym drugim przypadku okno konsoli pozostanie jednak otwarte aż do naciśnięcia dowolnego klawisza). Aby zmienić ten stan, umieścmy w tej metodzie polecenie przesyłające napis *Hello World!* do standardowego strumienia wyjścia.

Drukowanie napisów w konsoli i podpowiadanie kodu

Umieścimy w metodzie Main polecenie wyróżnione w listingu 1.2.

Listing 1.2. Wyświetlanie napisu Hello World!

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



Podobnie jak w C++ i Java, także i w C# wielkość liter ma znaczenie. `Console` i `console` to dwa różne słowa. I podobnie jak w tych językach nie ma znaczenia sposób ułożenia kodu. Oznacza to, że między słowa kluczowe, nazwy metod, klas itp. możemy w dowolny sposób wstawiać spacje i znaki końca linii. Nie możemy jedynie łamać łańcuchów, które powinny mieć zamykający cudzysłów w tej samej linii co rozpoczętajacy.

Podczas pisania kodu po napisaniu nazwy klasy `Console` i postawieniu kropki, tj. operatora umożliwiającego dostęp do jej statycznych właściwości i metod, pozostawmy przez chwilę kod bez zmian. Po krótkiej chwili pojawi się okno prezentujące wszystkie dostępne statyczne właściwości i metody tej klasy. Jest to element mechanizmu oglądu w kodzie *IntelliSense*. Dzięki niemu ręczne pisanie kodu jest znacznie przyjemniejsze. Wystarczy bowiem z listy wybrać pozycję `WriteLine`, czyli metodę drukującą napis w konsoli, zamiast wpisywać całą jej nazwę ręcznie. Jest to łatwiejsze, gdy wpiszemy początek nazwy, a więc „wr” (nie dbając o wielkość liter) — automatycznie zaznaczona zostanie pierwsza pozycja, która odpowiada tej sekwencji. Wówczas wystarczy nacisnąć klawisz `Enter`, aby potwierdzić wybór, lub przesunąć wybraną pozycję na liście, korzystając z klawiszy ze strzałkami. Jeżeli niechcący zamknijemy okno *IntelliSense*, możemy je przywołać kombinacją klawiszy `Ctrl+J` lub `Ctrl+Spacja` (tabela 1.1). *IntelliSense* pomoże nam również w przypadku argumentów metody `Console.WriteLine`. Warto z tej pomocy skorzystać.

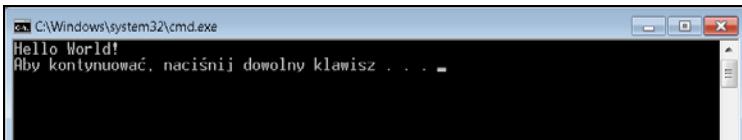


Mechanizm *IntelliSense* podpowiadający składowe obiektów może pracować w dwóch trybach. W pierwszym ogranicza się do pokazania listy pól, metod i właściwości. W drugim zaznacza jedną z nich. W tym drugim przypadku wystarczy nacisnąć klawisz `Enter`, aby kod został uzupełniony, podczas gdy w pierwszym spowoduje to tylko wstawienie znaku nowej linii do kodu. Do przełączania między tymi trybami służy kombinacja klawiszy `Ctrl+Alt+Spacja`.

Po naciśnięciu klawisza `Ctrl+F5` (uruchamianie bez debugowania) projekt zostanie skompilowany i uruchomiony, a na ekranie pojawi się czarny ekran konsoli, na którym zobaczymy napis *Hello World!* oraz napis wyprodukowany przez polecenie pause systemu Windows (rysunek 1.2).

Tabela 1.1. Podstawowe klawisze skrótów edytora Visual C#

Kombinacja klawiszy	Funkcja
<i>Ctrl+F</i>	Przeszukiwanie kodu
<i>Ctrl+H</i>	Przeszukiwanie z zastąpieniem
<i>F3</i>	Poszukiwanie następnego wystąpienia szukanego ciągu
<i>Ctrl+J</i>	Menu uzupełniania kodu
<i>Ctrl+Spacja</i>	Menu uzupełniania kodu lub uzupełnienie, jeżeli jednoznaczne
<i>Ctrl+Shift+Spacja</i>	Informacja o argumentach metody
<i>Ctrl+L</i>	Usunięcie bieżącej linii
<i>Ctrl+S</i>	Zapisanie bieżącego pliku
<i>Ctrl+Z</i>	Cofnięcie ostatnich zmian w kodzie
<i>Ctrl+A</i>	Zaznaczenie kodu w całym pliku
<i>Ctrl+X, Ctrl+C, Ctrl+V</i>	Obsługa schowka
<i>F7, Shift+F7</i>	Przeloczenie między edytorem a widokiem projektowania (w aplikacjach z interfejsem)
<i>Ctrl+Shift+B lub F6</i>	Budowanie całego projektu (klawisz <i>F6</i> może nie działać ¹)
<i>F5</i>	Kompilacja i uruchomienie aplikacji w trybie debugowania
<i>Ctrl+F5</i>	Kompilacja i uruchomienie aplikacji bez debugowania

**Rysunek 1.2.** Zawartość strumienia wyjścia w oknie konsoli w przypadku uruchomienia bez debugowania

Czekanie na akceptację użytkownika

W przypadku uruchomienia aplikacji *Hello.exe* z debugowaniem (klawisz *F5*) lub bezpośredniego uruchomienia pliku *.exe* poza środowiskiem Visual Studio przez chwilę pojawi się okno konsoli i prawie natychmiast zostanie zamknięte, nie dając nam szansy na obejrzenie wydrukowanego napisu. Możemy jednak wymusić na aplikacji, aby ta po wyświetleniu napisu wstrzymała działanie aż do naciśnięcia przez użytkownika klawisza *Enter*. W tym celu jeszcze raz zmodyfikujmy metodę *Main* (listing 1.3).

¹ Wówczas polecenie budowania projektów w rozwiązaniu można przypisać do klawisza *F6*. W tym celu należy z menu *Tools* wybrać polecenie *Customize...*. Pojawi się okno dialogowe, w którym klikamy przycisk *Keyboard...*. Pojawi się okno opcji z wybraną pozycją *Environment, Keyboard*. W polu tekstem *Show Command Containing* należy wpisać *BuildSolution*. Poniżej zaznaczone zostanie polecenie *Build.BuildSolution*. Jeszcze niżej zobaczymy, że przypisana jest do niego kombinacja *Ctrl+Shift+B*. W polu *Press shortcut keys* naciśnijmy klawisz *F6* i kliknijmy *Assign*. Po powrocie do edytora ten klawisz również będzie uruchamiał komplikację projektu.

Listing 1.3. Metoda Main z poleceniem sczytującym z klawiatury linię, czyli ciąg znaków zatwierdzony klawiszem Enter

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

Metoda Main zawiera w tej chwili dwa wywołania metod statycznych klasy Console. Metoda Console.Out.WriteLine umieszcza w standardowym strumieniu wyjścia (Console.→Out) napis podany w argumencie, w naszym przypadku *Hello World!*, dodając na końcu znak końca linii. Istnieje też podobnie działająca metoda Write, która takiego znaku nie dodaje. Obie metody zdefiniowane są również w samej klasie Console, a więc działałaby również instrukcja Console.WriteLine (bez Out). Podobnie jest w przypadku metody Console.In.ReadLine ze standardowego strumienia wejścia (Console.In) — możemy ją uruchomić też za pomocą metody wywołanej bezpośrednio na rzecz klasy Console, tj. Console.ReadLine. Metoda ta czeka na wpisanie przez użytkownika ciągu potwierzonego naciśnięciem klawisza *Enter*. Oprócz metody Console.ReadLine odczytującej cały łańcuch mamy do dyspozycji także metodę Read odczytującą kolejny znak ze strumienia wejściowego (wprowadzonego i potwierzonego klawiszem *Enter*) oraz metodę Console.ReadKey zwracającą naciśkany w danej chwili klawisz (ta ostatnia nadaje się do tworzenia interaktywnych menu sterowanych klawiaturą).

Odczytywanie danych z klawiatury

Wstrzymanie pracy aplikacji nie jest głównym zadaniem metody ReadLine. Przed wszystkim odbiera ona od użytkownika informacje w postaci łańcuchów. Aby zaprezentować jej prawdziwe zastosowanie, zmuśmy ją do pobrania od użytkownika imienia. W tym celu umieszczamy w metodzie Main polecenia widoczne na listingu 1.4.

Listing 1.4. Rozbudowana wersja metody Main

```
static void Main(string[] args)
{
    Console.WriteLine("Jak Ci na imię?");
    Console.Write("Napisz tutaj swoje imię: ");
    string imie = Console.ReadLine();
    if (imie.Length == 0)
    {
        Console.Error.WriteLine("\n\n\t*** Błąd: nie podano imienia!\n\n");
        return;
    }
    bool niewiasta = (imie.ToLower()[imie.Length - 1] == 'a');
    if (imie == "Kuba" || imie == "Barnaba") niewiasta = false;
    Console.WriteLine("Niech zgadnę, jesteś " + (niewiasta ? "dziewczyną" :
    →"chłopakiem") + "!");
    Console.WriteLine("Naciśnij Enter...");
    Console.Read();
}
```

Na rysunku 1.3 widoczny jest przykład interakcji aplikacji z użytkownikiem. Zwróćmy uwagę, że w przypadku zignorowania pytania o imię wygenerowany zostanie komunikat o błędzie, który będzie przesłany do strumienia System.Error. To strumień błędów. Domyślnie jest nim również konsola, ale z łatwością strumień ten, podobnie zresztą jak System.Out, mógłby być „przekierowany” np. do strumienia skojarzonego z plikiem rejestrującym błędy. Do zmian strumienia służą metody SetIn, SetOut i SetError klasy Console.



Rysunek 1.3. Interakcja z użytkownikiem w tradycyjnym stylu lat 80. ubiegłego wieku

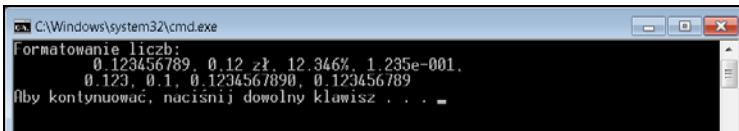
Argumenty metody Console.WriteLine mogą mieć również formę znaną z funkcji PRINT w FORTRAN-ie lub printf w C/C++, a mianowicie:

```
Console.WriteLine("Niech zgadnę, jesteś {0}!",  
    (niewiasta?"dziewczyną":"chłopakiem"));
```

W łańcuchu umieszczamy w nawiasach klamrowych numery wyrażeń (począwszy od zera), które mają być wyświetcone w danym miejscu, a które muszą być podane w kolejnych argumentach metody WriteLine. W przypadku liczb mamy możliwość ich formatowania, np.:

```
Console.WriteLine("Formatowanie liczb:\n\t{0}, {0:c}, {0:p3}, {0:e3}, \n\t{0:f3},  
    {0:g1}, {0:n10}, {0:r}", 0.123456789);
```

Ta ostatnia instrukcja prowadzi do wydruku widocznego na rysunku 1.4.



Rysunek 1.4. Przykłady formatowania liczb

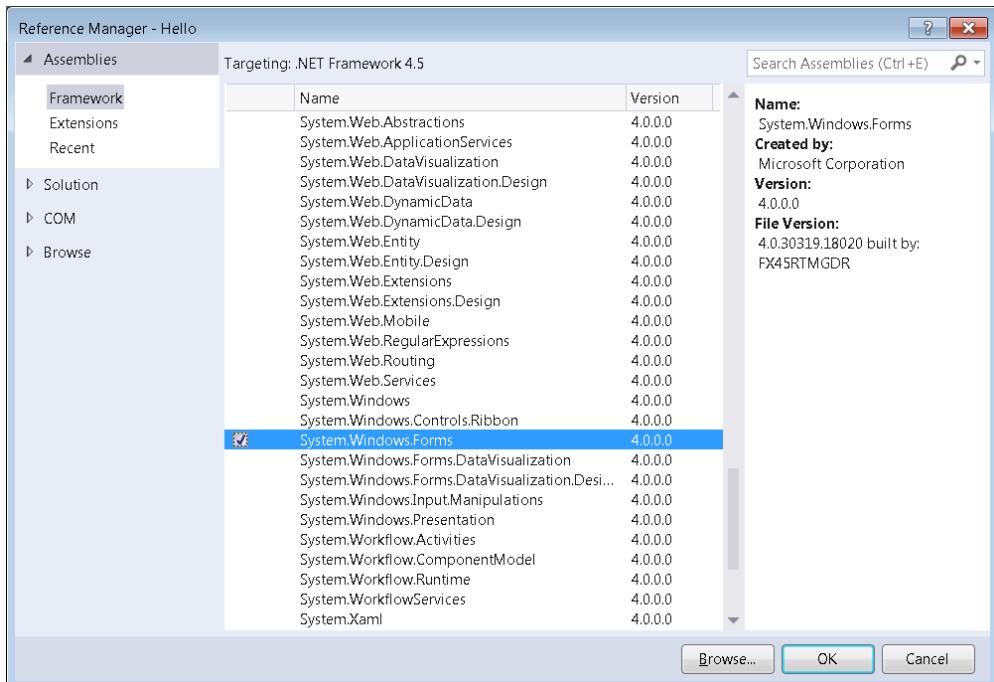
Korzystanie z bibliotek DLL. Komunikat „okienkowy” w aplikacji konsolowej

Choć nie jest to typowe, w zasadzie nic nie stoi na przeszkodzie, żeby w aplikacji konsolowej wykorzystać okna (po prawdziwej przestaje wówczas do niej pasować określenie „konsolowa”). Możemy np. wyświetlić małe okno uzyskane za pomocą metody System.Windows.Forms.MessageBox.Show². Wymaga to jednak pewnego wysiłku, ponieważ konieczne jest dodanie do projektu referencji do biblioteki System.Windows.Forms.dll, której w projekcie aplikacji konsolowej domyślnie nie ma.

² Aby wszystko było jasne: Show to metoda klasy MessageBox, która zdefiniowana jest w przestrzeni nazw System.Windows.Forms.

1. Aby dodać do projektu odwołanie do biblioteki *System.Windows.Forms.dll*:

- w podoknie projektu *Solution Explorer* z menu kontekstowego pozycji *Hello* wybierz *Add*, a następnie *Reference...*;
- w oknie *Reference Manager - Hello* (rysunek 1.5), w zakładce *Framework*, odnajdź i zaznacz pozycję *System.Windows.Forms* (należy zaznaczyć pole opcji z lewej strony tej pozycji);



Rysunek 1.5. W aplikacjach konsolowych biblioteka zawierająca klasy okien Windows nie jest domyślnie podłączana

c) kliknij przycisk *OK*.

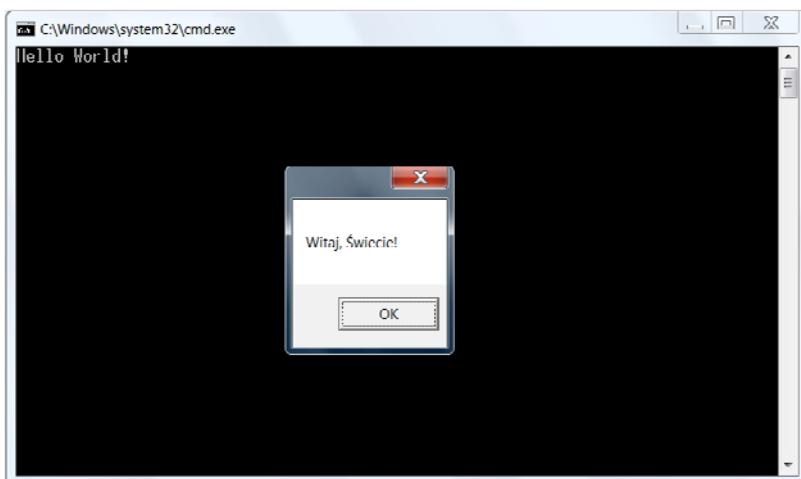
2. Na początku pliku *Program.cs* umieść instrukcję deklarującą użycie przestrzeni nazw *System.Windows.Forms*:

```
using System.Windows.Forms;
```

3. Dzięki temu możesz do metody *Main* dodać wywołanie metody *MessageBox.Show* bez konieczności podawania całej ścieżki dostępu obejmującej przestrzeń nazw (listing 1.5, rysunek 1.6).

Listing 1.5. Proste okno dialogowe w aplikacji konsolowej

```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
    System.Windows.Forms.MessageBox.Show("Witaj, Świecie!");
}
```



Rysunek 1.6. Okno Windows Forms w aplikacji konsolowej

Po wykonaniu kroku 1. powyższego ćwiczenia w gałęzi *References* podokna *Solution Explorer* została dodana pozycja *System.Windows.Forms.dll*. Nazwa biblioteki platformy .NET jest zazwyczaj zgodna z jej przestrzenią nazw.

Informacje o środowisku aplikacji

Żeby pokazać także bardziej praktyczny przykład aplikacji konsolowej, przygotujemy niewielką aplikację, która wyświetli informacje o środowisku, w jakim jest uruchamiana. Informacje te są dostępne dzięki klasie *System.Environment*.

Podstawowe informacje o systemie i profilu użytkownika

Utworzymy projekt aplikacji konsolowej o nazwie *SystemInfo*. Następnie zdefiniujemy statyczne pole typu *string* zawierające informacje o systemie (listing 1.6). Do metody *Program.Main* dodamy natomiast dwa polecenia wyświetlające informacje zebrane w łańcuchu *system* (także listing 1.6). Efekt widoczny jest na rysunku 1.7.

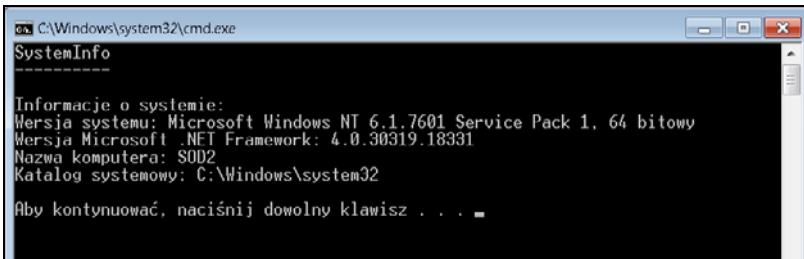
Listing 1.6. Pole pomocnicze, w którym umieszczono informacje o systemie

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

namespace SystemInfo
{
    class Program
    {
```

```
// Informacje o systemie
static private string system = "Informacje o systemie:"
+ "\nWersja systemu: " + Environment.OSVersion
+ (Environment.Is64BitOperatingSystem?", 64 bitowy": "")
+ "\nWersja Microsoft .NET Framework: " + Environment.Version
+ "\nNazwa komputera: " + Environment.MachineName
+ "\nKatalog systemowy: " + Environment.SystemDirectory;

static void Main(string[] args)
{
    Console.WriteLine("SystemInfo\n-----\n");
    Console.WriteLine(system + "\n");
}
}
```



Rysunek 1.7. Podstawowe informacje o systemie

Przygotowując łańcuch system, wykorzystaliśmy niektóre ze statycznych właściwości zdefiniowanych w klasie Environment. Warto zapoznać się z jej dokumentacją w plikach pomocy MSDN, aby poznać je wszystkie. Wśród niewykorzystanych tutaj właściwości jest np. TickCount, która zwraca liczbę milisekund od momentu uruchomienia systemu, lub HasShutdownStarted informująca o tym, czy system jest właśnie zamkany. Są również właściwości przechowujące parametry bieżącej aplikacji, takie jak CommandLine dostarczająca informacji o parametrach podanych w linii komend lub CurrentDirectory będąca katalogiem roboczym aplikacji.

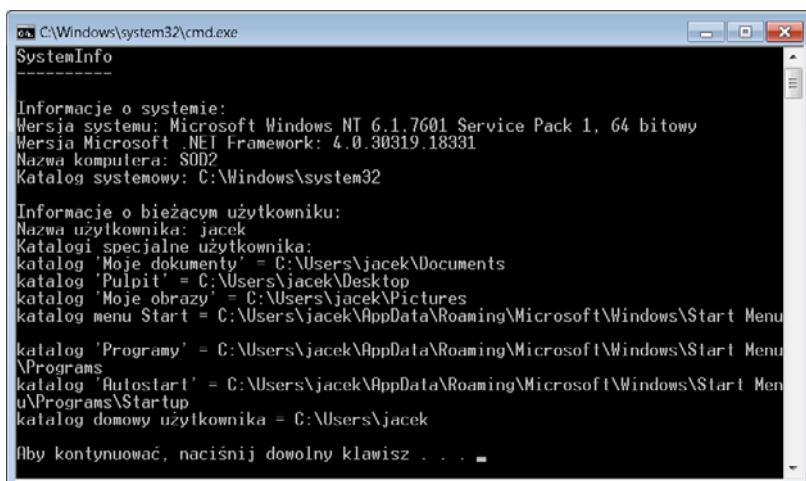
Katalogi specjalne zdefiniowane w bieżącym profilu użytkownika

Klasa Environment posiada również statyczne metody pozwalające na pobranie ścieżek do katalogów specjalnych zdefiniowanych w profilu użytkownika oraz zmiennych środowiskowych (rysunek 1.8).

1. Do klasy Program dodaj kolejne statyczne pole:

```
// Informacje o bieżącym użytkowniku
static private string użytkownik = "Informacje o bieżącym użytkowniku:"
+ "\nNazwa użytkownika: " + Environment.UserName
+ "\nKatalogi specjalne użytkownika:"
+ "\nkatalog 'Moje dokumenty' = " + Environment.GetFolderPath
    (Environment.SpecialFolder.Personal)
```

Rysunek 1.8.
Katalogi specjalne



```
+ "\nkatalog 'Pulpit' = " + Environment.GetFolderPath
    ↵(Environment.SpecialFolder.Desktop)
+ "\nkatalog 'Moje obrazy' = " + Environment.GetFolderPath
    ↵(Environment.SpecialFolder.MyPictures)
+ "\nkatalog menu Start = " + Environment.GetFolderPath
    ↵(Environment.SpecialFolder.StartMenu)
+ "\nkatalog 'Programy' = " + Environment.GetFolderPath
    ↵(Environment.SpecialFolder.ProgramFiles)
+ "\nkatalog 'Autostart' = " + Environment.GetFolderPath
    ↵(Environment.SpecialFolder.Startup)
+ "\nkatalog domowy użytkownika = " + Environment.GetFolderPath
    ↵(Environment.SpecialFolder.UserProfile);
```

2. Do metody Main dodaj polecenie:

```
Console.WriteLine(użytkownik + "\n");
```

Do pobrania katalogów specjalnych służy metoda `Environment.GetFolderPath`, której argumentem jest element typu wyliczeniowego `Environment.SpecialFolder`. Dzięki niej możemy pobrać pełne ścieżki nie tylko do katalogów użytkownika (*Moje dokumenty*, *Pulpit* itp.), ale również do używanego przez wszystkich użytkowników katalogu *Program Files* i innych katalogów związanych z instalacją oprogramowania.

Odczytywanie zmiennych środowiskowych

Pośród powyższych katalogów jest m.in. katalog domowy użytkownika. Jego ścieżkę możemy również odczytać ze zmiennej środowiskowej `USERPROFILE`³. Skorzystamy z metody `GetEnvironmentVariable`, której argumentem jest nazwa zmiennej środowiskowej. Ponadto wyświetlimy wszystkie zmienne środowiskowe. Można je odczytać przy użyciu metody `GetEnvironmentVariables`, która zwraca kolekcję zgodną z interfejsem `IDirectory`, czyli słownikiem (opis w rozdziale 3.), w którym każdy z elementów jest typu `DirectoryEntry` i posiada dwie właściwości: `Key` (klucz, hasło) oraz `Value`

³ Można się o tym przekonać, wpisując w konsoli Windows polecenie SET.

(wartość). W naszym przypadku hasło zawiera nazwę zmiennej środowiskowej, a wartość to przechowywany przez nią łańcuch. Na listingu 1.7 prezentuję zmodyfikowaną metodę Main, która realizuje opisane wyżej czynności.

Listing 1.7. Odczytywanie zmiennych środowiskowych

```
static void Main(string[] args)
{
    Console.WriteLine("SystemInfo\n-----\n");
    Console.WriteLine(system + "\n");
    Console.WriteLine(użytkownik + "\n");

    Console.WriteLine("Katalog domowy użytkownika = " +
        Environment.GetEnvironmentVariable("USERPROFILE"));

    //Zmienne środowiskowe
    string zmienne = "";
    System.Collections.IDictionary zmienneSrodowiskowe =
        Environment.GetEnvironmentVariables();
    foreach (System.Collections.DictionaryEntry zmienna in zmienneSrodowiskowe)
        zmienne += zmienna.Key + " = " + zmienna.Value + "\n";
    Console.WriteLine("\nZmienne środowiskowe:\n" + zmienne);
}
```

Lista dysków logicznych

Kolejną informacją udostępnianą przez klasę Environment, którą zaprezentujemy w naszym prostym programie, jest lista dysków logicznych. Aby ją pokazać, do metody Main dodajemy jeszcze cztery linie widoczne na listingu 1.8.

Listing 1.8. Instrukcje wyświetlające listę dysków logicznych (partyjci) w komputerze

```
//Dyski logiczne
string[] dyski = Environment.GetLogicalDrives();
string driveinfo = "\nDyski: ";
foreach (string dysk in dyski) driveinfo += dysk + " ";
Console.WriteLine(driveinfo + "\n");
```

* * *

Jedyna różnica między aplikacjami konsolowymi a aplikacjami WPF lub Windows Forms, które poznasz w drugiej części książki, to brak okna i związanej z nim pętli głównej odbierającej komunikaty, i w konsekwencji ograniczenie operacji wejścia i wyjścia do konsoli. Aplikacje te mogą jednak korzystać ze wszystkich dobrodziejstw platformy .NET, a więc zapisywać pliki, używać baz danych, wątków itd.

Rozdział 2.

O błędach i ich tropieniu

Nie ma aplikacji bez błędów, ale liczbę błędów można starać się redukować. Pomaga w tym środowisko programistyczne Visual Studio wraz z wbudowanym w nie debu-gerem, wskazującym linie kodu, które nie podobają się kompilatorowi, i pozwalają- cym na kontrolę uruchamianego kodu, jak również śledzenie jego wykonywania linia po linii. Przyjrzymy się bliżej kilku jego najczęściej wykorzystywanym możliwościom, które pomagają w szukaniu błędów.

Program z błędem logicznym — pole do popisu dla debugera

Zacznijmy od przygotowania aplikacji, która kompliluje się, a więc nie posiada błędów składniowych, ale której metody zawierają błąd powodujący jej błędne działanie.

1. Utwórz nowy projekt typu *Console Application* o nazwie *Debugowanie*.
2. W klasie Program, obok metody Main (na przykład nad nią), zdefiniuj metodę Kwadrat zgodnie ze wzorem z listingu 2.1.

Listing 2.1. W wyróżnionej metodzie ukryty jest błąd

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

namespace Debugowanie
{
    class Program
    {
        static private int Kwadrat(int argument)
        {
            int wartosc;
            wartosc = argument * argument;
            return wartosc;
        }
    }
}
```

```
        }

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

-
3. Teraz przejdź do metody `Main` i wpisz do niej polecenia wyróżnione w listingu 2.2.

Listing 2.2. Z pozoru wszystko jest w porządku...

```
static void Main(string[] args)
{
    int x = 1234;
    int y = Kwadrat(x);
    y = Kwadrat(y);
    string sy = y.ToString();
    Console.WriteLine(sy);
}
```

Oczywiście obie metody można „spakować” do jednej lub dwóch linii, ale właśnie taka forma ułatwi nam naukę debugowania.

Zacznijmy od uruchomienia aplikacji (klawisz *Ctrl+F5*), żeby przekonać się, że nie działa prawidłowo. Po kliknięciu przycisku zobaczymy komunikat wyświetlający wynik: $-496\ 504\ 304$. Wynik jest ujemny, co musi budzić podejrzliwość, bo podnoszenie do kwadratu liczby całkowitej nie powinno, oczywiście, zwracać wartości mniejszej od zera. Za pomocą kalkulatora możemy przekonać się ponadto, że nawet wartość bezwzględna wyniku nie jest prawdziwa, bo liczba 1234 podniesiona do czwartej potęgi to 2 318 785 835 536. Wobec tego jest już jasne, że w naszym programie musi być błąd. I to właśnie jego tropienie będzie motywem przewodnim większej części tego rozdziału.

Kontrolowane uruchamianie aplikacji w Visual C#

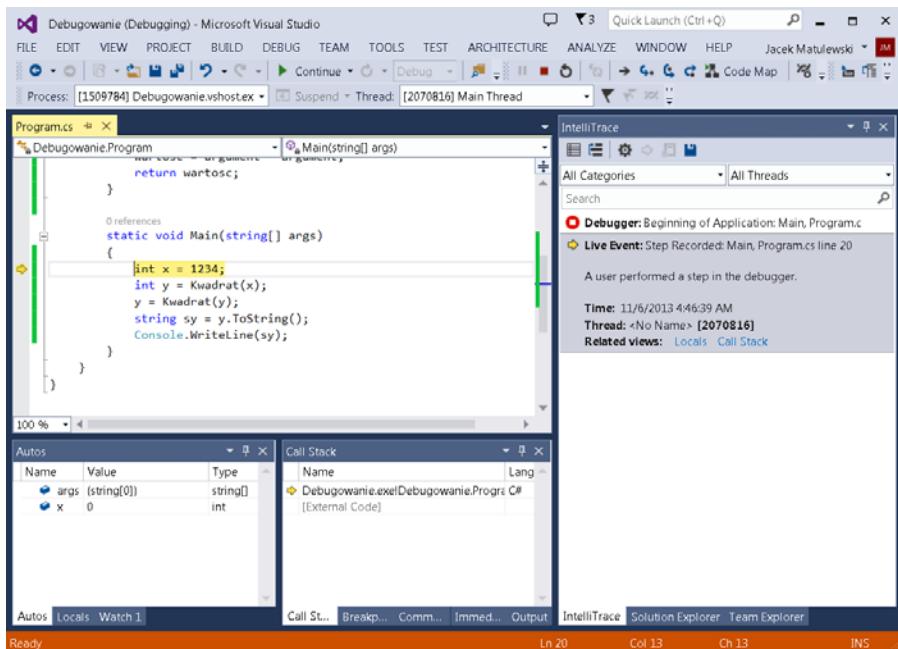
Śledzenie wykonywania programu krok po kroku (F10 i F11)

Po zakończeniu działania aplikacji uruchommy ją ponownie, ale w odmienny sposób. Tym razem naciśnijmy klawisz *F10* lub *F11* (wszystkie klawisze skrótów wykorzystywane podczas debugowania zebrane zostały w tabeli 2.1). W ten sposób aplikacja zostanie uruchomiona, ale jej działanie zostanie natychmiast wstrzymane na pierwszej instrukcji z metody `Main`. Każde naciśnięcie klawisza *F10* powoduje wykonanie jednej instrukcji, bez względu na to, czy jest to inicjacja zmiennej, czy wywołanie metody.

Tabela 2.1. Klawisze skrótów środowiska Visual C# związane z debugowaniem

Funkcja	Klawisz skrótu
Uruchomienie z debugowaniem	<i>F5</i>
Uruchomienie bez debugowania	<i>Ctrl+F5</i>
Uruchomienie i zatrzymanie w linii, w której jest kursor	<i>Ctrl+F10</i>
Krok do następnej linii kodu (<i>Step over</i>)	<i>F10</i>
Krok z wejściem w głęb metody (<i>Step into</i>)	<i>F11</i>
Krok z wyjściem z metody (<i>Step out</i>)	<i>Shift+F11</i>
Ustawienie <i>breakpointu</i> (funkcja edytora)	<i>F9</i>
Zakończenie debugowania (zakończenie działania aplikacji)	<i>Shift+F5</i>

Linia ta jest zaznaczona w edytorze Visual Studio żółtym kolorem (rysunek 2.1). Czynność taka nazywa się *Step Over* (zobacz menu *Debug*), czyli *krok nad*. Nazwa jest dobrze dobrana, bo jeżeli w wykonywanej linii znajduje się wywołanie metody, jest ona wykonywana w całości — nie wchodzimy do jej wnętrza. Natomiast *F11* powoduje wykonanie *kroku w głęb* (ang. *step into*), co w przypadku wywołania metody oznacza, że zostaniemy przeniesieni do pierwszej linii jej definicji i tam będziemy kontynuować śledzenie działania aplikacji. W przypadku gdy ta metoda zawiera wywołanie kolejnej metody, klawisze *F10* i *F11* pozwolą zdecydować, czy chcemy ją wykonać w całości, czy przeanalizować linia po linii. Jeżeli zorientujemy się, że zeszliśmy zbyt głęboko — możemy nacisnąć *Shift+F11*, aby wykonać pozostałą część metody i ją opuścić (ang. *step out — wyjście z*).

**Rysunek 2.1.** Żółtym tłem zaznaczone jest bieżące polecenie

Naciskając kilka razy klawisz *F10*, przechodzimy do tej linii metody *Main*, w której znajduje się drugie wywołanie metody *Kwadrat*. Wtedy naciśnijmy *F11*. Wówczas przemieszczamy się do metody *Kwadrat* (żółta linia będzie znajdować się na otwierającym jej ciało nawiasie klamrowym). Naciśkajmy *F10*, aby zobaczyć krok po kroku wykonywanie kolejnych poleceń. Zauważmy, że pominięta została linia zawierająca deklarację zmiennej *wartosc*, natomiast jednym z kroków była jej inicjacja. Naciśkając *F10*, prześledźmy wykonanie metody *Kwadrat*. Następnie powróćmy do metody *Main* i tak dalej aż do zakończenia programu.

Run to Cursor (**Ctrl+F10**)

Prześledziliśmy krok po kroku działanie całego programu. Tak można szukać błędów w takich prostych projektach jak ten, ale w bardziej złożonych debugowanie programu instrukcją po instrukcji byłoby zbyt czasochłonne. Szczególnie że zwykle szukamy błędów w jakimś określonym fragmencie kodu. Jak w takim razie od razu dostać się do interesującej nas metody? Służy do tego kombinacja klawiszy *Ctrl+F10*. Uručhamia aplikację tak samo jak klawisz *F5*, ale zatrzymuje ją w momencie, w którym wykonana ma być instrukcja z linii, w której znajduje się kursor edytora. Ta kombinacja klawiszy działa także wtedy, gdy aplikacja jest już uruchomiona w trybie śledzenia — wykonywany jest wówczas kod od bieżącego miejsca aż do zaznaczonej kursorem linii. Po zatrzymaniu działania aplikacji znowu możemy korzystać z klawiszy *F10* i *F11* do śledzenia działania metod krok po kroku.

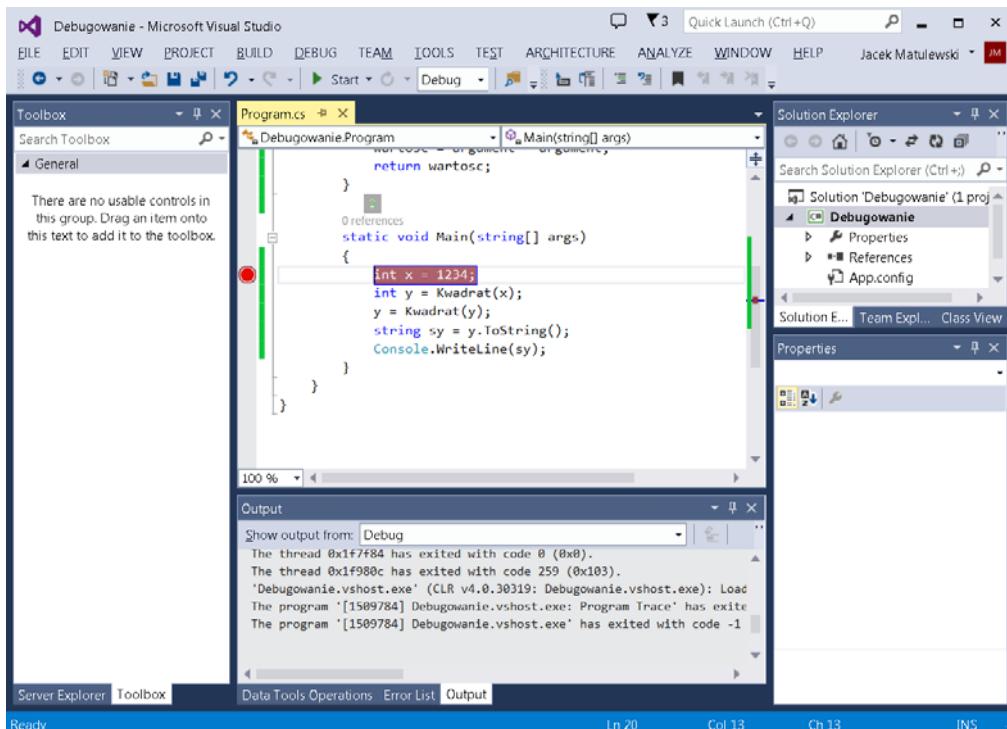


Aby natychmiast przerwać debugowanie programu i powrócić do normalnego trybu edycji Visual Studio, należy naciągnąć klawisze *Shift+F5*. Pasek stanu środowiska powinien zmienić kolor z pomarańczowego na niebieski.

Breakpoint (**F9**)

Gdy przewidujemy, że będziemy wielokrotnie kontrolować wykonywanie pewnych poleceń, np. metody *Kwadrat*, możemy ustawić tzw. *breakpoint*. W tym celu przechodzimy do wybranej linii w edytorze kodu i naciskamy klawisz *F9*. Linia zostanie zaznaczona brązowym kolorem oraz czerwoną kropką na lewym marginesie (rysunek 2.2). Po uruchomieniu programu w trybie debugowania jego działanie zostanie zatrzymane, gdy wątek dotrze do linii, w której ustawiliśmy *breakpoint*. Możemy wówczas przejść do śledzenia kodu (*F10* i *F11*) lub naciągnąć klawisz *F5*, aby wznowić jego działanie w normalnym trybie debugowania. Gdy jednak wątek znowu dotrze do *breakpointu*, działanie programu jeszcze raz zostanie wstrzymane. Aby anulować *breakpoint*, należy ustawić kursor w odpowiedniej linii i jeszcze raz naciągnąć *F9* lub kliknąć widoczną na lewym marginesie czerwoną kropkę.

Breakpoint bywa bardzo użyteczny przy śledzeniu wykonywania pętli. Jeżeli ustawimy go w interesującej nas linii wewnętrz pętli, każda jej iteracja zostanie przerwana i będziemy mieli możliwość np. przyjrzenia się wartościom zmiennych.

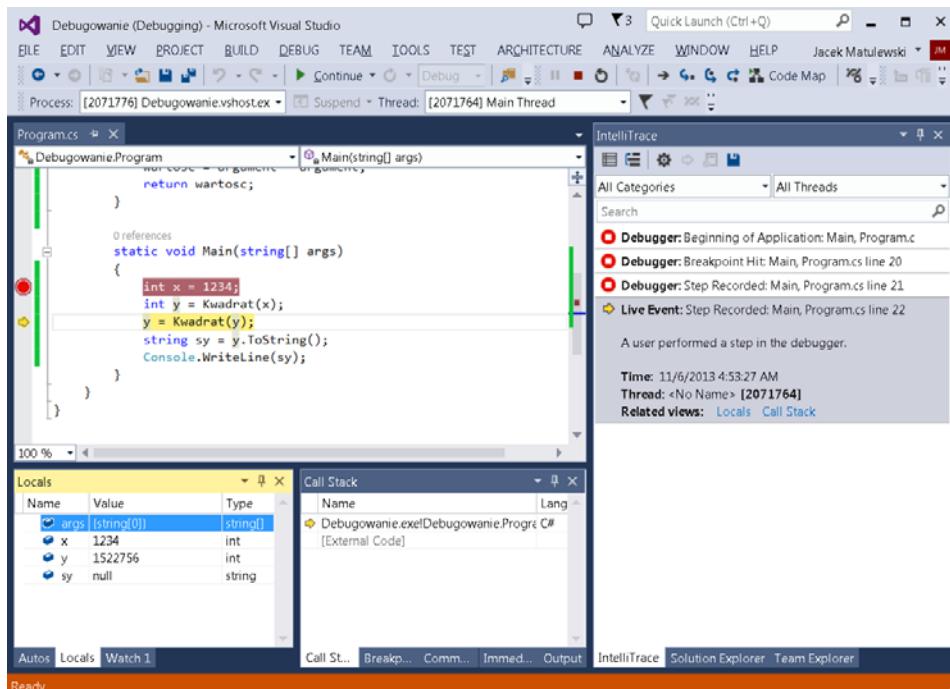


Rysunek 2.2. Wyróżnienie linii kodu, w której ustawiony jest breakpoint

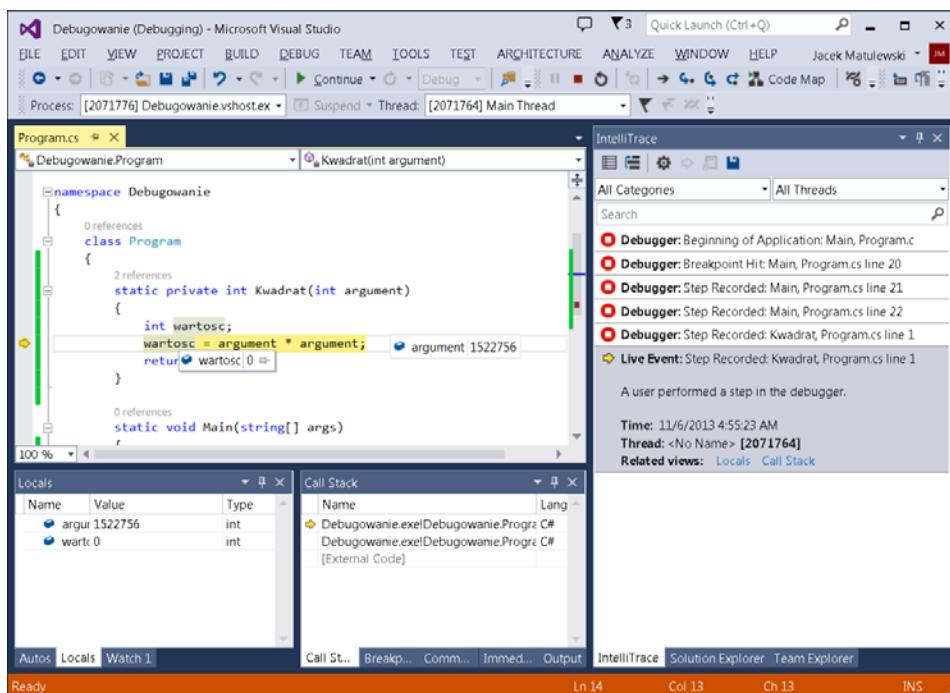
Okna Locals i Watch

Możliwość obserwacji wartości zmiennych używanych w programie to bardzo ważne narzędzie debuggerów. Dzięki niemu możemy w każdej chwili sprawdzić, czy wartości te są zgodne z naszymi oczekiwaniami, a to pozwala ocenić, czy program działa prawidłowo. Wykonywanie aplikacji linia po linii z równoczesnym wpatrywaniem się w wartości zmiennych jest w praktyce najczęściej wykorzystywanym sposobem na odszukanie owego mistycznego „ostatniego błędu”. W Visual Studio służy do tego okno *Locals* (rysunek 2.3), które zawiera listę wszystkich pól i zmiennych lokalnych dostępnych w aktualnie wykonywanej metodzie. Jeżeli zmienne są obiektami, możemy zobaczyć także ich pola składowe. Poza tym mamy do dyspozycji okno *Watch*, do którego możemy dodać nie tylko poszczególne pola i zmienne, ale również poprawne wyrażenia języka C#, np. `x*x`.

Wartości zmiennych można również zobaczyć, jeżeli w trybie debugowania w edytorze przytrzymamy przez chwilę kurSOR myszy nad zmienną użytą w kodzie. Po chwili pojawi się okienko zawierające wartość wskazanej w ten sposób zmiennej (rysunek 2.4). Należy jednak pamiętać, że pokazywana wartość dotyczy aktualnie wykonywanej linii, a nie linii wskazanej kursorem. Okienko, w którym pokazywana jest wartość zmiennej, można przypiąć do okna edytora (ikona pinezki), przez co podczas pracy w trybie debugowania pozostaje stale widoczna na ekranie. Możliwość podglądania wartości zmiennych w oknie edytora dotyczy także obiektów — zobaczymy wówczas wartości wszystkich dostępnych pól i własności.



Rysunek 2.3. Podglądarkanie wartości zmiennych w edytorze



Rysunek 2.4. U dołu okna VC# widoczne są dwa podokna: z lewej okno Watch, z prawej — Call Stack

Przejdzmy w edytorze do linii metody `Main`, w której zdefiniowana jest zmienna `y`, zaznaczmy tę zmienną i prawym przyciskiem myszy rozwińmy menu kontekstowe. Z niego wybierzmy polecenie *Add Watch*. Zmienna zostanie dodana do listy w oknie *Watch*, w której zobaczymy jej nazwę, wartość i typ. Wartość jest aktualizowana przy każdym naciśnięciu klawisza *F10* lub *F11*, co pozwala na śledzenie jej zmian w trakcie wykonywania kodu. Zmienione wartości oznaczane są kolorem czerwonym. Możemy się przekonać, wykonując poprzez ponowne naciskanie klawisza *F10* kolejne polecenia metody `Main`, że po pierwszym uruchomieniu metody `Kwadrat` zmienna `y` uzyskuje wartość 1 522 756, czyli jej wartość jest jeszcze poprawna. Niestety, przy drugim uruchomieniu metody `Kwadrat` jej wartość staje się ujemna. Z tego wynika, że to wywołanie metody `Kwadrat`, a nie np. konwersja wyniku do łańcucha jest przyczyną niepoprawnego wyniku, który widzimy w konsoli. Aby sprawdzić działanie metody `Kwadrat`, wchodzimy do jej „środkę”, używając klawisza *F11* (najlepiej podczas jej drugiego wywołania). W tej metodzie nie ma zmiennej `y`, a jej wartość przejmuje zmienna argument. Dodajmy ją zatem do listy obserwowanych zmiennych, wpisując jej nazwę do pierwszej kolumny w podoknie *Watch*. Dodajmy także wyrażenie `argument*argument`. Jego wartość okaże się nieprawidłowa.

Przyczyna błędu jest już chyba jasna — podczas mnożenia został przekroczyony, i to znacznie, zakres możliwych wartości zmiennej `int` ($1\ 522\ 756 * 1\ 522\ 756 = 2\ 318\ 785\ 835\ 536 > 2\ 147\ 483\ 647$). Innymi słowy, 32 bity, jakie oferuje typ `int`, nie wystarczają do zapisania wyniku, stąd jego zła interpretacja.



Wskazówka

Czytelnik zastanawia się zapewne, czemu maszyna wirtualna nie wykryła przekroczenia zakresu w mnożeniu i nie zgłosiła błędu. Nie robi tego, bo to bardzo obniżyłoby wydajność wykonywanych przez nią operacji arytmetycznych. Ale można ją do tego zmusić, używając słowa kluczowego `checked` (o tym poniżej).

Zwróciły także uwagę na podokno *Call Stack* (z ang. *stos wywołań*), które widoczne jest u spodu okna Visual Studio (rysunek 2.4). Wymienione są w nim wszystkie metody, począwszy od metody `Main`, a skończywszy na metodzie, której polecenie jest obecnie wykonywane. W naszym programie stos wywołań składa się tylko z metody `Main` i metody `Kwadrat`, ale w bardziej rozbudowanych programach tych metod może być sporo. Zawartość tego okna pomaga wówczas w zorientowaniu się, co się aktualnie dzieje w aplikacji, szczególnie po jej zatrzymaniu w wyniku działania *breakpointu*.

Stan wyjątkowy

Zgłaszanie wyjątków

Ostatnią rzeczą, o której chciałbym wspomnieć w kontekście tropienia błędów, jest obsługa wyjątków zgłaszanych przez aplikację i reakcja debugera w tej sytuacji. Dajmy do metody `Kwadrat` polecenie zgłaszające wyjątek w przypadku wykrycia błędnej wartości zmiennej `wartosc`.

1. Do metody Kwadrat dodaj polecenie wyróżnione w listingu 2.3.

Listing 2.3. Metoda Kwadrat z instrukcją zgłaszającą wyjątek w przypadku ujemnego wyniku

```
static private int Kwadrat(int argument)
{
    int wartosc;
    wartosc = argument * argument;
    if (wartosc < 0)
        throw new Exception("Funkcja kwadrat nie powinna zwracać wartości ujemnej!");
    return wartosc;
}
```

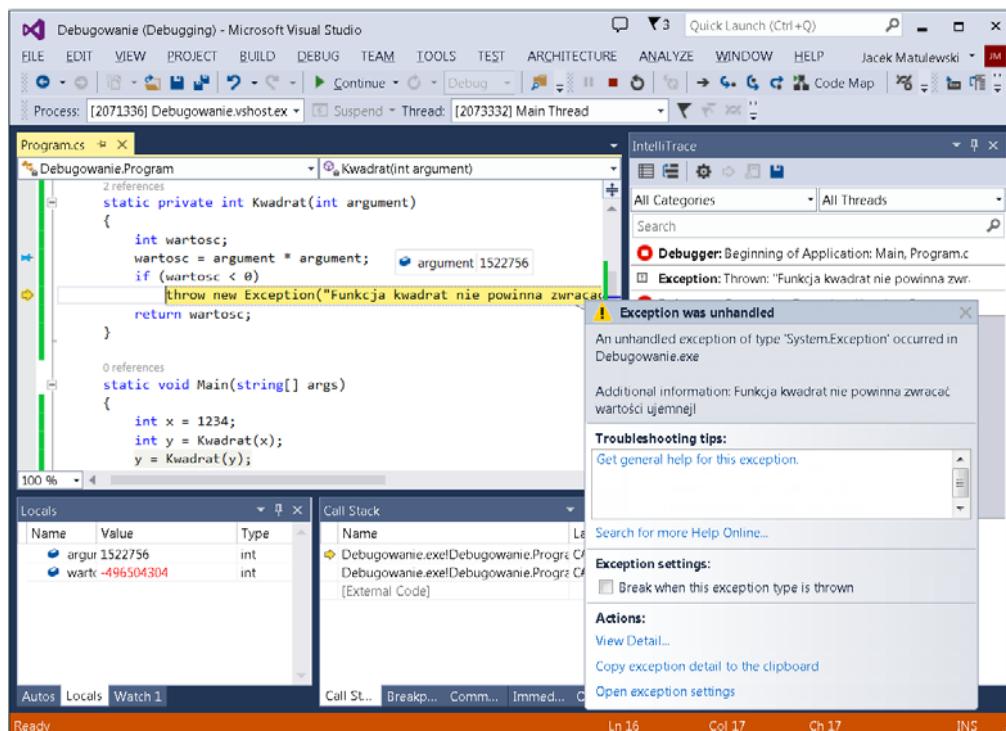
2. Usuń z kodu wszystkie *breakpointy*.

3. Uruchom aplikację klawiszem *F5* (uruchamianie z debugowaniem).

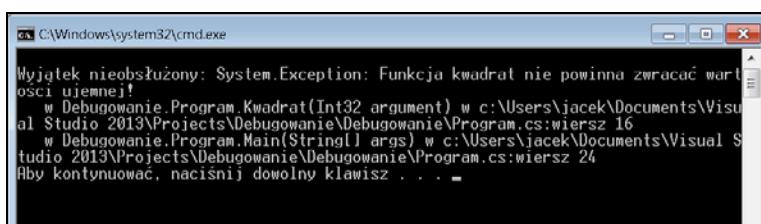
Po uruchomieniu w oknie Visual Studio pojawi się komunikat debugera informujący o tym, że aplikacja Debugowanie zgłosiła wyjątek typu `System.Exception` (rysunek 2.5). Przyjrzymy się temu komunikatowi. Przede wszystkim na pasku tytułu widoczne jest ostrzeżenie *Exception was unhandled*, co oznacza, że słowo kluczowe `throw` znajdowało się we fragmencie kodu, który nie był otoczony obsługą wyjątków. Nie znajdowało się zatem w sekcji try konstrukcji `try..catch` ani w metodzie, która byłaby z tej sekcji wywołana. To ważna informacja, bo w przypadku uruchomienia aplikacji poza środowiskiem Visual Studio wyjątek ten nie byłby w żaden sposób obsłużony przez aplikację i musiałaby się nim zająć sama platforma .NET, co raczej nie powinno mieć miejsca. W oknie komunikatu debugera widoczna jest także treść przekazywanego przez wyjątek komunikatu oraz łącze do dokumentacji klasy wyjątku (w naszym przypadku klasy `Exception`). Po kliknięciu *View Detail...* można obejrzeć stan przesyłanego obiektu w widocznym na dole okna odpowiedniku okna *Locals*.

Jak widzimy, po zgłoszeniu przez debugowany program wyjątku środowisko Visual Studio wstrzymuje działanie aplikacji na tej samej zasadzie jak *breakpoint*, tzn. zwykle możemy przywrócić jej działanie, np. za pomocą klawiszy *F5* lub *F10*. Możemy także zupełnie przerwać działanie aplikacji i przejść do poprawiania kodu, naciskając kombinację klawiszy *Shift+F5*.

Jeżeli aplikacja zostanie uruchomiona poza debuggerem przez naciśnięcie kombinacji klawiszy *Ctrl+F5* lub przez samodzielne uruchomienie pliku `.exe` poza środowiskiem Visual Studio i aplikacja nie przechwytuje zgłoszonych w niej wyjątków, wyjątek taki jest przechwytywany przez platformę .NET. O takiej sytuacji użytkownik powiadamiany jest komunikatem widocznym na rysunku 2.6. W aplikacji konsolowej sytuacja taka oznacza zakończenie działania programu. Warto jednak wiedzieć, że w aplikacji Windows Forms, której poświęcona jest druga część tej książki, po zgłoszeniu nieobsługiwanej wyjątku istnieje możliwość kontynuowania jej działania. Oczywiście wyjątek pojawi się znowu, jeżeli ponownie wejdziemy do metody zawierającej błąd, ale aplikacja się nie zawiesi. Warto docenić tę własność platformy .NET — jeżeli nasza aplikacja byłaby edytorem, a błąd pojawiłby się np. podczas drukowania, platforma .NET, nie zamkując całej aplikacji w przypadku wystąpienia błędu, dałaby nam możliwość m.in. zapisania na dysku edytowanego dokumentu.



Rysunek 2.5. Komunikat o wyjątku zgłoszony przez Visual C#



Rysunek 2.6. Wyjątek nieobsłużony w aplikacji jest przechwytywany przez środowisko .NET (jeżeli aplikacja uruchomiona jest bez debugera)

Przechwytywanie wyjątków w konstrukcji try..catch

Teraz spróbujmy wykryć wystąpienie wyjątku z poziomu programu. W tym celu w metodzie Main otoczymy wywołanie metody Kwadrat konstrukcją przechwytywania wyjątków try..catch.

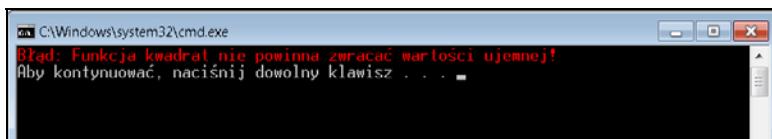
1. Naciśnij Shift+F5, aby zakończyć działanie debugera.
2. Do metody Main dodaj obsługę wyjątku zgodnie ze wzorem na listingu 2.4.

Listing 2.4. Wszystkie dotychczasowe polecenia metody Main umieściliśmy w jednej sekcji try

```
static void Main(string[] args)
{
    try
    {
        int x = 1234;
        int y = Kwadrat(x);
        y = Kwadrat(y);
        string sy = y.ToString();
        Console.WriteLine(sy);
    }
    catch (Exception ex)
    {
        ConsoleColor bieżącyKolor = Console.ForegroundColor;
        Console.ForegroundColor = ConsoleColor.Red;
        Console.Error.WriteLine("Błąd: " + ex.Message);
        Console.ForegroundColor = bieżącyKolor;
    }
}
```

3. Skompiluj i uruchom aplikację.

Teraz po uruchomieniu aplikacji zamiast komunikatu debugera lub komunikatu platformy .NET zobaczymy własny komunikat o błędzie (rysunek 2.7). Debugger i Visual Studio nie zainterwenują w przypadku obsłużonego wyjątku. Wystąpienie błędu w sekcji try powoduje jej przerwanie i przejście do sekcji catch. Po wykonaniu poleceń z tej sekcji wątek nie wróci już do sekcji try; przejdzie do wykonywania instrukcji znajdujących się za konstrukcją try..catch. W przypadku z listingu 2.4 działanie metody Main w takiej sytuacji po prostu się zakończy, a tym samym zakończy się działanie całego programu.

**Rysunek 2.7.** Komunikat wyświetlany użytkownikowi po przechwyceniu wyjątku

Wszystkie polecenia metody Main umieściliśmy w jednej sekcji try. To ma sens, skoro każde wystąpienie błędu powoduje, że wykonywanie dalszej części metody pozbawione jest celu. Jeśli natomiast po pojawienniu się błędu chcielibyśmy podjąć jakąś akcję ratunkową i próbować kontynuować działanie metody, to każda budząca wątpliwości instrukcja powinna być otoczona oddzielną konstrukcją obsługi wyjątków.



Więcej informacji na temat wyjątków można znaleźć w następnym rozdziale.

Wymuszenie kontroli zakresu zmiennych

Weryfikowanie, czy wartość zwracana przez metodę `Kwadrat` jest nieujemna, nie gwarantuje wykrycia wszystkich błędnych wyników. Przy wielokrotnym przekroczeniu zakresu znak wyniku jest w zasadzie przypadkowy. Wspomniałem jednak, że możemy zmusić wirtualną maszynę platformy .NET do kontrolowania przekraczania zakresu w operacjach arytmetycznych (a także podczas konwersji typów). Można to uzyskać, korzystając ze słowa kluczowego `checked`. Wówczas w razie wykrycia przekroczenia zakresu zgłoszany jest wyjątek `OverflowException`. Należy być jednak świadomym, że kontrola taka znacznie obniża wydajność programu. I z tego powodu domyślnie nie jest włączona.

Zmodyfikujmy ostatni raz metodę `Kwadrat` zgodnie ze wzorem widocznym na listingu 2.5. Jeżeli teraz uruchomimy projekt, wyjątek zostanie zgłoszony już w linii, w której zmienna argument podnoszona jest do kwadratu (przy drugim wywoaniu metody `Kwadrat`). Mechanizm przechwytywania wyjątków z metody `Main` zadziała jednak podobnie i w oknie konsoli zobaczymy komunikat *Nastąpiło przepelenie w czasie wykonywania operacji arytmetycznej*.

Listing 2.5. Kontrola przekroczenia zakresu zmiennej int przy mnożeniu

```
static private int Kwadrat(int argument)
{
    int wartosc;
    wartosc = checked(argument * argument);
    if (wartosc < 0)
        throw new Exception("Funkcja kwadrat nie powinna zwracać wartości ujemnej!");
    return wartosc;
}
```



Dodatkowe informacje na temat kontroli zakresu przy operacjach arytmetycznych znajdują się w rozdziale 6.

Rozdział 3.

Język C# 5.0

Język C# został zaprojektowany i jest rozwijany przez grupę programistów Microsoftu pod kierunkiem Andersa Hejlsberga. Jest to osoba, która wcześniej była odpowiedzialna za kompilator Turbo Pascal, środowisko Delphi i bibliotekę VCL w firmie Borland, a od 1996 roku, już w Microsoftie, za projekt J++, a potem C# i platformę .NET.

Język C#, jeżeli brać pod uwagę jego słowa kluczowe i podstawową składnię, bez wątpienia należy do tej samej rodziny co języki C++ i Java. Dynamiczny rozwój tego języka spowodował jednak, że pojawiło się w nim ostatnio wiele nowych konstrukcji wykraczających poza ramy ustalone przez standard języka C++. Najlepszym przykładem są zapytania LINQ i używane w nich operatory. Rozpoczynając naukę języka C#, można więc bazować na znajomości innych języków, ale warto przygotować się również na nowości.

Drugim składnikiem technologii .NET jest środowisko uruchomieniowe CLR (ang. *Common Language Runtime*), którego podstawowe założenia są podobne jak w przypadku wirtualnej maszyny Javy. Separacja aplikacji od warstwy systemu operacyjnego i wstawienie między nie platformy .NET (ang. *.NET framework*) ma kilka korzyści. Przede wszystkim pozwala chronić stabilność systemu. Ponadto pozwala na wprowadzenie nowych mechanizmów, których nie oferuje sam system operacyjny. Najlepszym przykładem jest zarządzanie pamięcią. Korzyścią, którą chciałbym jednak podkreślić w szczególny sposób, jest uniwersalność. Microsoft od kilku lat rozszerza liczbę urządzeń, na których działają jego systemy. Poza komputerami typu PC są to wszelkiego typu urządzenia kontrolowane przez systemy Windows Embedded, czyli smartfony z Windows Phone i tablety z Windows RT, czy wreszcie konsole Xbox i Zune. Na większości z tych urządzeń działa jakaś wersja platformy .NET lub WinRT. Aplikacje dla wszystkich tych platform, włączając w to już nierożwiane technologie Silverlight i XNA, można programować, używając jednego języka — C#.

W pewnym zakresie platforma .NET realizuje też ideę przenośności kodu programów. Na przykład prostych aplikacji napisanych dla platformy .NET dla komputerów PC można było używać na urządzeniach z Windows Mobile i odwrotnie. Gry przygotowane w XNA powinny działać na Xbosie, PC i Windows Phone'ie bez zasadniczych przeróbek. Bardzo podobny jest kod aplikacji WPF, Windows Phone i Windows Store. Co ciekawe, przenośność kodu programów .NET realizowana jest także wysiłkiem społeczności *open source*. Dzięki temu, że Microsoft udostępnił pełną specyfikację

platformy .NET, stale rozwijane są jej wersje otwarte, w szczególności platforma *Mono*, która działa m.in. w systemach Linux. Należy jednak zastrzec, że pewien zakres przenośności kodu nie oznacza wcale przenośności skompilowanych programów napisanych w C#.

Platforma .NET i język C# rozwijane są dopiero od końca lat dziewięćdziesiątych ubiegłego wieku (w 2001 roku udostępniona została ich pierwsza wersja publiczna), a już można je uważać za w pełni dojrzałe. Więcej — wyznaczają nowe kierunki rozwoju informatyki i penetrują mało do tej pory badane komercyjnie jej obszary. Mam tu na myśli technologię LINQ (wersja 3.5 platformy .NET, C# 3.0) czy programowanie asynchroniczne (C# 5.0) i współbieżne (wersja 4.0 platformy .NET) oraz ideę użycia kompilatora jako usługi, która ma pojawić się w wersji 5.0 tej platformy.

Platforma .NET

Środowisko uruchomieniowe

Biblioteki platformy .NET tworzą środowisko, w którym uruchamiane są aplikacje .NET. Wspomniany już „wirtualny komputer”, który uruchamia skompilowane programy .NET, nazywa się *Common Language Runtime* (w skrócie CLR). Tworzy on warstwę pośrednią między aplikacją a systemem Windows, która z jednej strony chroni system przed niebezpiecznymi zachowaniami aplikacji, przejmując kontrolę nad zarządzaniem pamięcią i wątkami, a z drugiej — udostępnia aplikacjom biblioteki klas .NET wraz z ujednoliconym mechanizmem dostępu do zasobów systemu, w tym szczególnie do baz danych (ADO.NET). Ciekawa z punktu widzenia programisty jest dostępność wielu kompilatorów dla platformy .NET i CLR. Obecnie aplikacje .NET możemy tworzyć w C++, Visual Basicu, C#, F#, Delphi, a nawet Perlu lub JavaScriptie. To C# pozostaje jednak z pewnością sztandarowym językiem tej technologii.

W wersji 4.0 platformy .NET do środowiska uruchomieniowego CLR dołączona została warstwa DLR (ang. *Dynamic Language Runtime*) odpowiedzialna za obsługę języków dynamicznych, takich jak Python i Ruby, ale również nowych dynamicznych elementów C# 4.0 (podrozdział „Typy dynamiczne” w tym rozdziale).



Wskazówka

Warto wspomnieć o siostrzanych platformach WinRT i XNA. Pierwsza wykorzystywana jest w aplikacjach Windows Store uruchamianych w Windows 8 w nowym trybie ekranu Start. Druga jest modyfikacją platformy .NET przystosowaną do wyświetlania grafiki 2D i 3D. Niestety Microsoft zarzucił rozwój projektu XNA.

Kod pośredni i podwójna komplikacja

Podobnie jak w przypadku Javy, także w przypadku technologii .NET kod źródłowy programu nie jest komplikowany bezpośrednio do kodu maszynowego. Komplikacja jest dwustopniowa. W pierwszej fazie zostaje on skompilowany do kodu pośredniego

(ang. *Common Intermediate Language*, w skrócie CIL, dawniej MSIL), wspólnego dla wszystkich języków programowania platformy .NET i dla wszystkich środowisk uruchomieniowych .NET, bez względu na system operacyjny i procesor. Kod pośredni CIL jest odpowiednikiem *bytecode* z Javy. Ten kod jest czytelny dla środowiska uruchomieniowego CLR. Druga faza to komplikacja kodu pośredniego przez CLR za pomocą kompilatorów *Just-In-Time* (skrót JIT). Nie musi się ona odbywać bezpośrednio po pierwszym etapie ani nawet na tym samym komputerze. Platforma .NET prowadza ją dopiero w momencie uruchamiania programu.

Nieco zamieszania może powodować fakt, że wynikiem pierwszej komplikacji kodu źródłowego C# — oraz innych języków .NET — są pliki *.exe* lub *.dll*. Mimo tego samego rozszerzenia i nagłówka rozpoczynającego się od „MZ” struktura tych plików jest zupełnie inna niż tradycyjnych plików wykonywalnych i bibliotek zawierających kod uruchamiany w platformie Win32 lub Win64. Zawierają one bowiem kod pośredni CIL¹. Z punktu widzenia użytkownika, jeżeli tylko platforma .NET jest zainstalowana w systemie, nie ma to większego znaczenia — po prostu uruchamia on program, klikając dwukrotnie plik *.exe*. Jednak z punktu widzenia systemu operacyjnego różnica jest ogromna.

Wersje

Najnowsza wersja platformy .NET, wydana równocześnie z Visual Studio 2013, ma numer 4.5.1. Jest drobną modyfikacją platformy 4.5 wydanej razem z Visual Studio 2012. Historię rozwoju platformy .NET można prześledzić w tabeli 3.1. Pierwsze dwie wersje, tj. 1.0 i 1.1, można uznać za prapoczątki platformy .NET. Najstarszą nadal używaną wersją jest 2.0. Co ciekawe, kolejne wydania platformy, a więc wersje 3.0 i 3.5, były w pewnym sensie dodatkami do wersji 2.0, bo korzystały z jej CLR i podstawowych klas, wzmacniając je o nowe technologie. W wersji 3.0 było to *Windows Presentation Foundation* (WPF), czyli nowa biblioteka kontrolek pozwalających na budowanie interfejsu aplikacji „okienkowych”, *Windows Communication Foundation* (WCF), która uporządkowała komunikację sieciową w aplikacjach .NET, technologia kontroli przepływu *Windows Workflow Foundation* (WF) oraz zarządzanie tożsamosciami (*CardSpace*). Z kolei platforma .NET 3.5 dodała LINQ — język zapytań wbudowany w platformę .NET (więcej o nim w trzeciej części książki). To wymagało rozbudowy samego języka C# (wersja 3.0). Platforma .NET 4.0, już w pełni samodzielna, to przede wszystkim nowości związane z programowaniem współbieżnym — nowa klasa Task z biblioteki TPL (ang. *Task Parallel Library*), wspólny LINQ i klasy, które potrafią współpracować z równoczesnym dostępem z wielu wątków².

¹ Można się o tym przekonać, stosując prosty program *ildasm.exe* dołączony do Microsoft .NET Framework SDK.

² Zobacz naszą książkę pt. *Programowanie równolegle i asynchroniczne w C# 5.0*, Helion, 2013.

Tabela 3.1. Oficjalne wersje platformy .NET

Wersja	Rok wydania	Wersja języka C#	Najważniejsze nowości	Wersja Visual Studio	Odpowiednie wydania tej książki
1.0	luty 2002	1.0		Visual Studio .NET, wersja 7.0	J. Matulewski, <i>Język C#</i> . <i>Programowanie dla platformy .NET w środowisku Borland C# Builder</i> , HELP 2004
1.1	kwiecień 2003	1.2	piersza wersja dodawana do systemu Windows; wsparcie dla ASP.NET, ODBC, edycja Compact	Visual Studio 2003, wersja 7.1	J. Matulewski, <i>Visual C# 2005 Express Edition. Od podstaw</i> , Helion 2006
2.0	listopad 2005	2.0	najstarsza wersja nadal wspierana przez Visual Studio; wsparcie dla arch. x64, rozwoj ASP.NET, rozwój kontrolek współpracujących z ADO.NET	Visual Studio 2005, wersja 8.0	J. Matulewski, <i>Visual C# 2005 Express Edition. Od podstaw</i> , Helion 2006
3.0	listopad 2006		wersja niesamodzielna, korzystająca z .NET 2.0; WPF, WCF, WF, Card Space	Expression Blend	J. Matulewski, <i>Visual C# 2008. Projektowanie aplikacji. Pierwsze starcie</i> , Helion 2008
3.5	listopad 2007	3.0	wersja niesamodzielna, korzystająca z .NET 2.0 i 3.0; LINQ, Entity Framework, pseudotyp var	Visual Studio 2008, wersja 9.0	J. Matulewski <i>C# 3.0 i .NET 3.5. Technologia LINQ</i> , Helion 2008
4.0	kwiecień 2010	4.0	Parallel Extensions: TPL, Parallel LINQ, „współbieżne” kolekcje; DLR i dynamic, BigInteger i Complex	Visual Studio 2010, wersja 10.0	J. Matulewski, D. Borycki, G. Krause, M. Grabek, M. Pakulski, M. Wareczak, J. Lewandowski, S. Orlowski <i>Visual Studio 2010 dla programistów C#, Helion 2011</i>
4.5	sierpień 2012	5.0	async/await, caller information, PCL, WinRT/Windows 8	Visual Studio 2012, wersja 11.0	
4.5.1	październik 2013			Visual Studio 2013, wersja 12.0	bieżące wydanie

W odróżnieniu od wcześniejszej praktyki wersje 4.5 i 4.5.1 zastępują wersję 4.0. I to dosłownie — pliki nowej wersji fizycznie zastępują pliki wersji 4.0. Dodatkowo mylące może być to, że wersja zapisana wewnątrz plików pozostaje bez zmian (np. *c:\Windows\Microsoft.NET\Framework64\v4.0.30319\System.dll*). Zmienia się tylko nieco rozmiar niektórych z nich (w przypadku pliku *System.dll* rozmiar zmniejsza się z 3 512 072 bajtów do 3 453 000 bajtów).

Skróty, które warto poznać

Jak każda technologia, także .NET wprowadza wiele nowych nazw i skrótów. Aby ułatwić Czytelnikowi posługiwanie się nowym żargonem związanym z .NET, co jest warunkiem swobodnego korzystania ze źródeł informacji na temat .NET dostępnych w internecie, przedstawiam najczęściej spotykane skróty:

- ◆ **CIL** (ang. *Common Intermediate Language*) — dawniej MSIL (ang. *Microsoft Intermediate Language*), język pośredni, wynik komplikacji z języka C# lub innego języka .NET. Kod CIL jest ponownie kompilowany przez CLR i uruchamiany.
- ◆ **CLR** (ang. *Common Language Runtime*) — środowisko uruchomieniowe aplikacji platformy .NET, warstwa pośrednia między systemem Windows a aplikacją.
- ◆ **CLS** (ang. *Common Language Specification*) — wymagania stawiane językom i ich kompilatorom, aby mogły tworzyć aplikacje dla platformy .NET uruchamiane przez CLR.
- ◆ **CTS** (ang. *Common Type System*) — podzbiór specyfikacji CLS określający jednolity system typów istniejących w środowisku .NET (tabela 3.2), które powinny być dostępne w każdym kompilatorze.
- ◆ **DLR** (ang. *Dynamic Language Runtime*) — dynamiczny system typów i inne mechanizmy wprowadzające do platformy .NET i języka C# elementy programowania dynamicznego.
- ◆ **CAS** (ang. *Code Access Security*) — podsystem środowiska CLR odpowiedzialny za dopilnowanie, aby ze względu na bezpieczeństwo systemu aplikacje nie wykrańczały poza wyznaczone im ramy działania, tzn. aby np. aplikacja uruchamiana z sieci nie miała możliwości zapisu na lokalnych dyskach itp.³
- ◆ **JIT** (ang. *Just-In-Time*) — kompilator, który kod pośredni (CIL) kompiluje do kodu maszynowego, co pozwala na wykonanie poleceń znacznie szybciej niż przy bezpośredniej interpretacji CIL. Kompilator JIT jest elementem środowiska uruchomieniowego (CLR), więc jego wykorzystanie nie ogranicza przenośności kodu pośredniego.

³ Analogicznie jak w Java 2 w platformie .NET od wersji 1.1 użytkownik może kontrolować politykę bezpieczeństwa (aplikacja *CasPol.exe*).

Podstawowe typy danych

Poniżej staram się opisać praktyczne aspekty języka C# od samych podstaw. Zaczynam od definiowania zmiennych i metod, aby dojść do posługiwania się kolekcjami. Ten rozdział został przygotowany w taki sposób, aby nie tylko nadawał się on do lektury jako „tutorial” C#, ale również mógł funkcjonować jako leksykon, do którego Czytelnik może wrócić, gdy w dalszych rozdziałach napotka niezrozumiałą fragment kodu.



Aby testować poniższe instrukcje i fragmenty kodu, wystarczy utworzyć projekt typu *Console Application*. Do wyświetlania komunikatów wykorzystuję konsolę. Wszystkie instrukcje można po prostu umieścić w metodzie Main.

Deklaracja i zmiana wartości zmiennej

Zaczniemy od elementarza. Podobnie jak we wszystkich językach z rodziny C/C++/Java, także w C# deklarujemy zmienną w dowolnym miejscu metody, podając jej typ i nazwę:

typ nazwa_zmiennej;

np.:

```
int i;  
long l;  
string s;  
float f;  
double d;
```

Są to kolejno: zmienne typu całkowitego (cztero- i ośmiobajtowa), łańcuch oraz zmienne zmiennoprzecinkowe pojedynczej i podwójnej precyzji (cztero- i ośmiobajtowa). Zmienne zadeklarowane w ten sposób nie mogą jednak być użyte, zanim nie zostanie im przypisana jakaś wartość (w przeciwnym razie pojawi się błąd kompilatora).

Zmiennej może być nadana wartość za pomocą operatora przypisania =:

```
i = 1;  
l = 1L;  
s = "Helion";  
f = 1.0f;  
d = 1.0;
```

Obie instrukcje mogą być połączone i wówczas deklaracji towarzyszy inicjacja zmiennej. Najczęściej używana forma deklaracji zmiennej wygląda wobec tego następująco:

typ nazwa_zmiennej = wartość_inicjująca;

np.:

```
int i = 1;  
long l = 1L;  
string s = "Helion";  
float f = 1.0f;  
double d = 1.0;
```

Typy liczbowe oraz typ znakowy

W C/C++ wymienione powyżej typy danych poza łańcuchem nazywane byłyby typami prostymi — dla odróżnienia od klas. Z kolei w Javie istnieją typy proste, ale dodatkowo zdefiniowane są odpowiadające im tzw. klasy opakowujące, dostarczające metod pomocniczych i dodatkowych informacji o typie. Projektanci C# poszli natomiast o krok dalej — tzw. typy proste są strukturami, a słowa kluczowe znane z C, C++ i Javy, takie jak `int` lub `double`, są tylko aliasami nazw tych struktur.

W efekcie stała typu `int`, np. 1, jest instancją struktury `System.Int32` i pozwala na dostęp do wszystkich metod tej klasy. Nie powinien zatem dziwić następujący kod:

```
int i=10;
string s=i.ToString();
```

lub wręcz:

```
string s=10.ToString();
```

Dostępne w C# typy danych „prostych” zebrane zostały w tabeli 3.2.

Tabela 3.2. „Proste” typy danych dostępne w C# oraz ich zakresy

Nazwa typu (słowo kluczowe, alias klasy)	Klasa z obszaru nazw System	Liczba zajmowanych bitów	Możliwe wartości (zakres)	Domyślana wartość
<code>bool</code>	<code>Boolean</code>	1 bajt = 8 bitów	<code>true, false</code>	<code>false</code>
<code>sbyte</code>	<code>SByte</code>	8 bitów ze znakiem	od -128 do 127	0
<code>byte</code>	<code>Byte</code>	8 bitów bez znaku	od 0 do 255	0
<code>short</code>	<code>Int16</code>	2 bajty = 16 bitów ze znakiem	od -32 768 do 32 767	0
<code>int</code>	<code>Int32</code>	4 bajty = 32 bity ze znakiem	od -2 147 483 648 do 2 147 483 647	0
<code>long</code>	<code>Int64</code>	8 bajtów = 64 bity ze znakiem	od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807	<code>0L</code>
<code>ushort</code>	<code>UInt16</code>	2 bajty = 16 bitów bez znaku	maks. 65 535	0
<code>uint</code>	<code>UInt32</code>	4 bajty = 32 bity bez znaku	maks. 4 294 967 295	0
<code>ulong</code>	<code>UInt64</code>	8 bajtów = 64 bity bez znaku	maks. 18 446 744 073 709 551 615	<code>0L</code>
<code>char</code>	<code>Char</code>	2 bajty = 16 bitów (zob. komentarz)	Znaki Unicode od U+0000 do U+FFFF (od 0 do 65 535)	<code>'\0'</code>
<code>float</code>	<code>Single</code>	4 bajty (zapis zmiennoprzecinkowy)	Wartość może być dodatnia lub ujemna; najmniejsza bezwzględna wartość to $1,4 \cdot 10^{-45}$, największa to ok. $3,403 \cdot 10^{38}$	<code>0.0F</code>

Tabela 3.2. „Proste” typy danych dostępne w C# oraz ich zakresy — ciąg dalszy

Nazwa typu (słowo kluczowe, alias klasy)	Klasa z obszaru nazw System	Liczba zajmowanych bitów	Możliwe wartości (zakres)	Domyślna wartość
double	Double	8 bajtów (zapis zmiennoprzecinkowy)	Najmniejsza wartość to $4,9 \cdot 10^{-324}$, największa to ok. $1,798 \cdot 10^{308}$	0.0D
decimal	Decimal	16 bajtów (większa precyzja, ale mniejszy zakres niż double)	Najmniejsza wartość to 10^{-28} , największa do $7,9 \cdot 10^{28}$	0.0M

**Wskazówka**

Zakres większości typów „prostych” można sprawdzić za pomocą statycznych pól `MinValue` i `MaxValue`, np. `double.MaxValue`, natomiast ich rozmiar liczony w bajtach zwraca operator `sizeof`.

Wartość domyślną typu można odczytać za pomocą słowa kluczowego `default`, np. `default(int)`. W przypadku typów liczbowych jest nią zero (tabela 3.2).

Warto zwrócić uwagę na to, że C#, podobnie jak Java, posiada typ znakowy `char`, który jest dwubajtowy, i dzięki temu liczba znaków, które mogą być adresowane zmiennymi tego typu, obejmuje całą tablicę Unicode, tj. $2^{16} = 65\,536$ znaków. W ten sposób rozwiązany został problem liter specyficznych dla różnych języków, choćby polskich ą, ć, ę itd.

Do inicjacji liczb wykorzystywane są stałe liczbowe (literły liczbowe), czyli wszystkie te wyrażenia, których wartość znana jest już w momencie komilacji. Poza stałymi o typowej postaci, jak 1, 1.0 lub 1E0, można wykorzystywać dodatkowe stałe wzbogacone o litery określające typ stałej (tabela 3.3).

Tabela 3.3. Stałe liczbowe w C#

Zapis w kodzie C#	Typ	Opis
1	int	Liczba całkowita ze znakiem
1U	uint	Liczba całkowita bez znaku
1L	long	Liczba całkowita 64-bitowa
1F	float	Liczba
1M, 1.0M	decimal	Liczba całkowita ze znakiem o zwiększonej precyzji (dzięki zapisowi podobnemu do liczb zmiennoprzecinkowych)
1E0, 1.0E0, 1.0	double	Liczba zmiennoprzecinkowa
01		Liczba ósemkowa
0x01		Liczba szesnastkowa

Określanie typu zmiennej przy inicjacji (pseudotyp var)

Aby ułatwić odczytywanie danych z nieznanych źródeł, do C# 3.0 wprowadzono słowo kluczowe `var`. Pozwala ono definiować zmienne o typie ustalonym przez komplator na podstawie wartości użytej do ich inicjacji (ang. *implicitly-typed variables*). Inicjacja musi jednak nastąpić w tej samej linii kodu co deklaracja. Tak zainicjowane zmienne nie powinny być mylone ze zmiennymi o dynamicznym typie (typ `dynamic` z C# 4.0); typ `var` po inicjacji respektuje kontrolę typu i np. do zmiennej typu `var` zainicjowanej stałą typu `int` nie można już przypisać łańcucha. Zmienne tego typu mogą być jedynie zmiennymi lokalnymi, a więc mogą być deklarowane i inicjowane tylko wewnątrz metod, a nie mogą być polami klas.

Możliwe jest zatem deklarowanie np. następujących zmiennych:

```
var i = 5;
var l = 5L;
var s = "Helion";
var f = 1.0f;
var d = 1.0;
```

W tym przypadku zmienna `i` będzie typu `int` (`System.Int32`), `l` typu `long` (`System.Int64`), `s` typu `string`, `f` typu `float`, a `d` — `double`. Można się o tym łatwo przekonać, odczytując typ zmiennych poleceniem `i.GetType().FullName`. Typu `var` można również używać do kontrolek bibliotek Windows Forms lub WPF, kolekcji, typów parametrycznych czy klas i struktur definiowanych przez programistę. W pełni odkryjemy jego zalety, gdy będziemy chcieli przechować dane odczytane za pomocą zapytania LINQ — typ zwracany przez zapytania często jest trudny do określenia, a słowo kluczowe `var` pozwala po prostu o tym nie myśleć.

Operatory



Wskazówka

W C# są dwa rodzaje zmiennych: wartościowe i referencyjne. Różnica między nimi jest bardzo istotna i zostanie omówiona w dalszej części tego rozdziału.

Operacje na typach liczbowych i znakowym umożliwiają m.in. operatory. W C# można definiować operatory w projektowanych przez siebie klasach (tym zajmiemy się w kolejnym rozdziale). Dostępne w C# operatory wymienione zostały w tabeli 3.4 w kolejności malejącego priorytetu.



Wskazówka

Operatory arytmetyczne (+, -, *, /) i bitowe (~, &, ^ i |) nie są przeciążone dla typu `byte`. Wobec tego aby uzyskać wyniki z przykładów umieszczonych w tabeli 3.3, należy wykonać odpowiednie rzutowania na ten typ, np. `byte x1 = 74, z1 = (byte)(~x1); Console.WriteLine(" " + z1);` lub `byte x2 = 74, y2 = 15, z2 = (byte)(x2 & y2); Console.WriteLine(" " + z2);`.

Tabela 3.4. Predefiniowane operatory C# (kolejność odpowiada priorytetowi)

Grupa operatorów	Operator	Opis
Podstawowe (ang. <i>primary</i>)	x.y	Dostęp do pola, metody, właściwości i zdarzenia
	f(), f(x)	Wywołanie metody
	a[n]	Odwołanie do elementu tablicy lub indeksatora
	x++, x--	Zwiększenie lub zmniejszenie wartości o 1 po wykonaniu innej instrukcji (np. po wykonaniu int x=2; int y=x++; otrzymamy y=2, x=3)
	new	Tworzenie obiektu, składnia: <i>Klasa referencja=new Klasa(arg_konstr);</i>
	typeof	Zwraca zmienną typu System.Type opisującą typ argumentu: typeof(int).ToString() równy jest System.Int32
	checked, unchecked	Operatory kontrolujące zgłaszanie wyjątku przekroczenia zakresu dla operacji na liczbach całkowitych int i = unchecked(int.MaxValue + 1);
Jednoargumentowe (ang. <i>unary</i>)	+x, -x	Operatory zmiany znaku liczby (np. -3)
	!	Negacja logiczna (np. !true to false)
	~	Negacja bitowa (zdefiniowana dla typów int, uint, long i ulong): ~01001010 = 10110101
	++x, --x	Zwiększenie lub zmniejszenie wartości o 1 przed wykonaniem innej instrukcji (np. w wyniku wykonania operacji int x=2; int y=++x; otrzymamy y=3, x=3)
	(typ)x	Jawne rzutowanie (konwersja) zmiennej x na typ wskazany w nawiasie. Uwaga! (int)0.99 = 0
Mnożenia (ang. <i>multiplicative</i>)	*	Operator dzielenia zwraca wynik zgodny z typem argumentów: 5/2=2, 5/2.0=2.5, 5f/2=2.5
	/, %	Operator reszty z dzielenia: 5%2=1
Dodawania (ang. <i>additive</i>)	+, -	Dodawanie i odejmowanie. Typ wyniku zależy od typu argumentów, więc: uint i=1; uint j=3; uint k=i-j; spowoduje, że k=4294967294
Przesunięcia bitów (ang. <i>shift</i>)	<<, >>	1<<1 = 2, bo jedynka została przesunięta z ostatniej pozycji na przedostatnią i zostało dostawione zero
Porównania wartości i typów (ang. <i>relational and type testing</i>)	<, >, <=, >=	Porównanie wartości: 1>2 ma wartość false
	is	Porównanie typów: 1 is int ma wartość true
	as	Rzutowanie równoznaczne z: <i>wyrażenie is typ ? (typ)wyrażenie : (typ)null</i>
Równości (ang. <i>equality</i>)	==, !=	Wyrażenie 0.5==1/2f jest prawdziwe (wartość logiczna), 0.5!=1/2, czyli 0.5!=0 jest również prawdziwe

Tabela 3.4. Predefiniowane operatory C# (kolejność odpowiada priorytetowi) — ciąg dalszy

Grupa operatorów	Operator	Opis
	&	Bitowe AND (i), tzn. bit jest zapalany tylko wtedy, jeżeli oba odpowiednie bity argumentów są zapalone $74 \& 15 = 10 (01001010 \& 00001111 = 00001010)$
Operacje na bitach	^	Bitowe XOR (rozłączne lub) — bit jest równy 1 tylko wtedy, gdy odpowiednie bity argumentów są różne $74 ^ 15 = 69 (01001010 ^ 00001111 = 01000101)$
		Bitowe OR (lub) — bit jest zapalony, jeżeli przynajmniej jeden odpowiedni bit argumentów jest zapalony $74 15 = 79 (01001010 00001111 = 01001111)$
Operacje na wartościach logicznych (bool)	&&	Logiczne AND (true jedynie wtedy, gdy oba argumenty są true)
		Logiczne OR (true, gdy przynajmniej jeden argument jest true)
Warunkowy (ang. conditional)	?:	warunek?wartość_jeżeli_true:wartość_jeżeli_false, gdzie warunek musi mieć wartość typu bool, np. $x>y?x:y$ zwraca większą wartość spośród x i y
Przypisywanie (ang. assignment)	= oraz *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =	Inicjacja i zmiana wartości zmiennych, np. polecenia: int x=1; x+=2; nadadzą zmiennej x wartość 3

Konwersje typów podstawowych

Poza jawną konwersją, którą programista może uzyskać, korzystając z odpowiedniego operatora (tabela 3.4), w kodzie C# można stosować także konwersję niejawną. Ze względów bezpieczeństwa jest ona jednak bardziej ograniczona niż w C++. Niejawną konwersja dla liczb jest zawsze możliwa z typu całkowitego na typ zmiennoprzecinający (generalnie do typu o większej precyzyji) oraz z typu o mniejszym zakresie na typ o większym zakresie (kodowany przez większą liczbę bitów). Konwersja w drugą stronę, a więc wiążąca się z utratą informacji (precyzji), wymaga jawnego użycia operatora konwersji, a zatem świadomej decyzji programisty. Dlatego przy próbie przypisania wartości typu double do zmiennej typu int pojawi się błąd kompilatora ostrzegający przed możliwą utratą precyzji (jego treść to: *Cannot implicitly convert 'double' to 'int'*):

```
double x=1; // to jest OK
int i=1.0; // tu pojawi się błąd
```

W przypadku operatorów dwuargumentowych, które są użyte dla argumentów różnych typów (np. 1.0+1), niejawną konwersję jest również wykonywana tylko zgodnie z powyższymi zasadami. W takiej sytuacji typ zwracanej przez operator wartości zależy oczywiście od typów użytych argumentów. Wybierany jest typ o większej precyjności lub większym zakresie.

```
1.0+1 // double
1.0/2 // double, wartość 0.5
1/2 // int, wartość 0
```

Spośród operatorów arytmetycznych tylko operacja dzielenia może powodować problemy związane z typami i zasadami ich konwersji. W szczególności w przypadku dzielenia liczb całkowitych wynikiem może być liczba wymierna, ale ze względu na konwencję, w której wynik działania operatora ma typ odpowiadający typowi argumentu o największej precyzji i zakresie, wynikiem dzielenia dwóch liczb o typach całkowitych jest zaokrąglona w kierunku zera liczba całkowita. Oznacza to, że wartość operatora / jest w takim przypadku przybliżona. Jest to jedno z najczęstszych źródeł błędów logicznych w programach.

Należy także zwrócić uwagę, że jawną konwersja liczb zmienoprzecinkowych na całkowite wiąże się z obcięciem części wartości znajdującej się po przecinku, a nie z poprawnym matematycznie zaokrągleniem. Wyrażenia (int)0.99 lub (int)-0.99 mają zatem wartość 0, a nie spodziewane 1 lub -1. Dokładnie tak samo wykonywane jest zaokrąglenie w przypadku dzielenia liczb całkowitych.

Operatory is i as

W C# możliwe jest przypisanie referencji, np. typu `DivideByZeroException` (klasa wyjątku informującego o dzieleniu przez zero) do zmiennej typu `Object`. Klasa `Object` jest bowiem klasą bazową (nie bezpośrednio) klasy `DivideByZeroException`. Możliwe jest bezpieczne rzutowanie na uboższą klasę. W każdej chwili można jednak sprawdzić, z jakiego typu obiektem mamy tak naprawdę do czynienia. Pozwala na to operator `is`, który zwraca wartość logiczną będącą odpowiedzią na pytanie typu: „Czy obiekt jest wyjątkiem?”. Oto przykład:

```
Object o = new DivideByZeroException();
if (o is DivideByZeroException) Console.WriteLine("Obiekt jest wyjątkiem dzielenia
    przez zero");
else Console.WriteLine("Obiekt nie jest wyjątkiem dzielenia przez zero");
```

Operator `is` zwróci wartość `true` także dla klasy bazowej `Exception`:

```
if (o is Exception) Console.WriteLine("Obiekt jest wyjątkiem ");
else Console.WriteLine("Obiekt nie jest wyjątkiem");
```

ale nie dla klas potomnych.

Poza tym na rzecz każdego obiektu w platformie .NET możemy wywołać metodę `GetType`, która zwraca typ tego obiektu.

```
Console.WriteLine(o.GetType().FullName);
```

Po zweryfikowaniu typu obiektu można jego referencję bez obaw zrzutować na właściwy typ:

```
DivideByZeroException e = null;
if (o is DivideByZeroException) exc = (DivideByZeroException)o;
```

Zamiast z takiej konstrukcji wygodniej jednak skorzystać z operatora `as`:

```
DivideByZeroException exc = o as DivideByZeroException;
```

Jest on równoznaczny z konstrukcją: `o is DivideByZeroException ? (DivideByZeroException)o : (DivideByZeroException)null`. Operator `as`, podobnie jak jego pierwowzór z Object Pascal'a, może być używany tylko do zmiennych referencyjnych. To ograniczenie nie dotyczy operatora `is`.

Łańcuchy

Problem łańcuchów został w C# rozwiązany w sposób konsekwentny. Po pierwsze, łańcuchy są implementowane w klasie `System.String`⁴ (alias `string`), w której zdefiniowane zostały wszystkie przydatne metody i właściwości. Pozwalają one m.in. na porównywanie łańcuchów, analizę ich zawartości oraz modyfikacje poszczególnych znaków lub fragmentów. Najważniejsze metody zostały zebrane w tabeli 3.5. Dostęp do metod klasy `String` możliwy jest zarówno wtedy, gdy dysponujemy zmienną typu `string`, jak i na rzecz stałych łańcuchowych.

```
string s="Wydawnictwo Helion";
int dlugosc=s.Length;
```

lub po prostu:

```
int dlugosc="Wydawnictwo Helion".Length;
```

Po drugie, łańcuchy w C# bazują na zestawie znaków Unicode, co oznacza, że każdy znak kodowany jest dwoma bajtami (por. typ `char` w tabeli 3.1). Nie istnieje zatem problem dostępności znaków narodowych. I wreszcie po trzecie, do dyspozycji mamy przeciążony operator `+` służący do łatwego łączenia łańcuchów.

Ciągi definiujące łańcuchy mogą zawierać sekwencje specjalne rozpoczynające się od lewego ukośnika (znaku *backslash*) \ (identycznie jak w C/C++). Często wykorzystywany jest znak końca linii \n oraz znak cudzysłowu \". Ten ostatni nie kończy łańcucha i jest traktowany dokładnie tak samo jak inne znaki. Sekwencja kasująca poprzedni znak to \b. Wreszcie sekwencje pozwalające definiować znaki Unicode (także spoza dostępnego na klawiaturze zestawu ASCII) zaczynają się od \u i obejmują kolejne znaki alfanumeryczne⁵, np. \u0048. Oto przykład łańcucha zdefiniowanego w ten sposób:

```
string helion = "Wydawnictwo \" \u0048 \u0065 \u0066 \u0069 \u006f \u006e \"";
```

Wartość łańcucha `helion` to `Wydawnictwo " H e l i o n "` (spacje w jego definicji zostały umieszczone tylko po to, żeby wyraźniej pokazać sekwencje dla każdego znaku Unicode).

⁴ Jest to typ referencyjny, ale jego operatory przypisania = i porównania == działają tak jak dla typu wartościowego. Kopiowanie oznacza zatem w istocie klonowanie, a porównanie dotyczy wartości, a nie adresu.

⁵ Przypominam, że w .NET typ `char` jest dwubajtowy (tabela typów prostych 3.1), trzeba go zatem kodować aż czterocyfrowymi liczbami szesnastkowymi. Właściwe kody znaków Unicode można znaleźć w aplikacji *Tablica znaków* w menu *Start/Wszystkie programy/Akcesoria/Narzędzia systemowe* systemu Windows.

Tabela 3.5. Najczęściej używane metody klasy String

Funkcja wartość nazwa(argumenty)	Opis	Przykład użycia
<i>indeksator []</i>	Zwraca znak na wskazanej pozycji; pozycja pierwszego znaku to 0	"Helion"[0]
<i>bool Equals(string)</i>	Porównuje łańcuch z podanym w argumencie	"Helion".Equals("HELP")
<i>int IndexOf(char), int IndexOf(String)</i>	Pierwsze położenie litery lub łańcucha; -1, gdy nie zostanie znaleziony	"Lalalalala".IndexOf('a')
<i>int LastIndexOf(char), int LastIndexOf(String)</i>	Jw., ale ostatnie wystąpienie litery lub łańcucha	"Lalalalala".LastIndexOf('a')
<i>int Length</i>	Zwraca długość łańcucha, właściwość	"Helion".Length
<i>string Replace(string,string), string Replace(char,char)</i>	Zamiana wskazanego fragmentu na inny	"HELP".Replace("P", "ION")
<i>string Substring(int,int)</i>	Fragment łańcucha od pozycji w pierwszym argumencie o długości podanej w drugim	"Wydawnictwo".Substring(5,3)
<i>string Remove(int,int)</i>	Usuwa wskazany fragment	"Helion".Remove(1,2)
<i>string Insert(int,string)</i>	Wstawia łańcuch przed literą o podanej pozycji	"Helion".Insert(2, "123")
<i>string ToLower() string ToUpper()</i>	Zmienia wielkość wszystkich liter	"Helion".ToLower()
<i>string Trim() oraz TrimStart, TrimEnd</i>	Usuwa spacje z przodu i z tyłu łańcucha	" Helion ".Trim()
<i>string PadLeft(int,char) string PadRight(int,char)</i>	Uzupełnia łańcuch znakiem podanym w drugim argumencie, aż do osiągnięcia długości zadanej w pierwszym argumencie	"Helion".PadLeft(10, ' ')
<i>bool EndsWith(string), bool StartsWith(string)</i>	Sprawdza, czy łańcuch rozpoczyna się lub kończy podanym fragmentem	"csc.exe".EndsWith("exe")

Ze względu na wykorzystanie znaku \ do rozpoczętania sekwencji kodujących znaki specjalne dla wprowadzenia jego samego konieczna jest specjalna sekwencja \\\. Stąd biorą się podwójne lewe ukośniki w C/C++, Java i C# w przypadku zapisanych w łańcuchach ścieżek dostępu do plików w systemie Windows:

```
string nazwapliku= " c:\\WINDOWS\\Microsoft.NET\\Framework\\v4.0.30319\\csc.exe";
```

Wartość tego łańcucha to w rzeczywistości `c:\WINDOWS\Microsoft.NET\Framework\v4.0.30319\csc.exe`. W C# tą samą wartość można uzyskać, stawiając przed łańcuchem znak @ i pomijając podwójne lewe ukośniki, np.:

```
string nazwapliku=@"c:\WINDOWS\Microsoft.NET\Framework\v4.0.30319\csc.exe";
```

W łańcuchu poprzedzonym znakiem @ kompilator interpretuje wszystkie znaki dosłownie, ignorując ewentualne sekwencje specjalne. Uwzględniane są nawet znaki końca linii w kodzie, jeżeli występują między cudzysłowami. Same cudzysłowy uzyskuje się powtórzonym znakiem "" (podczas gdy bez znaku @ należy użyć sekwencji \").

String kontra StringBuilder

Wspomniałem wyżej, że choć łańcuch (`string`) jest typem referencyjnym, to jego operator przypisania = zdefiniowany jest w taki sposób, że powoduje tak naprawdę klonowanie obiektu. To dotyczy również jego metod służących do manipulacji wartością łańcucha. Wszystkie te, które zwracają wartość typu `string`, a więc m.in. `Insert`, `Remove` czy `Replace`, nie operują na bieżącej instancji łańcucha, a tworzą i zwracają nowy łańcuch. Jeżeli wykonujemy zatem dużo operacji na łańcuchach, np. wewnętrz pętli, koszt wszystkich kopiovań może być spory. Z tego powodu w momencie, w którym chcemy wielokrotnie zmieniać raz utworzony łańcuch, zamiast klasy `string` powinniśmy użyć klasy `StringBuilder`.

Rozważmy dwa łańcuchy: jeden typu `string`, a drugi — `StringBuilder`. Wykonajmy na nich dwie operacje: dołączmy do nich inny łańcuch i zastąpmy ich fragment innym. Realizuje to poniższy kod:

```
//string
string s = "abc---";
s += "xyz";
s = s.Replace("---", " ijk ");
Console.WriteLine(s);

//StringBuilder
System.Text.StringBuilder sb = new System.Text.StringBuilder("abc---");
sb.Append("xyz");
sb.Replace("---", " ijk ");
Console.WriteLine(sb.ToString());
```

Efekt wykonania obu zbiorów instrukcji jest taki sam — pojawi się komunikat o treści *abc ijk xyz*. Przyjrzymy się im jednak uważniej. W przypadku typu `string` działanie operatora += oznacza utworzenie nowej zmiennej typu `string`, której adres⁶ jest zapisywany w tej samej referencji co pierwotny łańcuch. To oznacza kopianie. Zastępowana wartość łańcucha, a więc obiekt typu referencyjnego, pozostaje natomiast bez referencji, co oznacza, że zajmie się nim *garbage collector*. To jednak nie dzieje się natychmiast. Analogicznie działają metody `Replace`, `Insert` czy `Remove`. Również

⁶ Używam tu i poniżej sformułowania „adres”, choć powinienem używać słowa „referencja”. Tego ostatniego zwykle się jednak używać na określenie zmiennej typu referencyjnego. Gdybym był purystą terminologicznym, zamiast sformułowania „adres obiektu jest zapisywany do referencji” użylibym sformułowania „referencja do obiektu jest zapisywana do zmiennej referencyjnej”. Wydaje się to jednak mniej zrozumiałe.

one nie modyfikują obiektu typu `string`, na rzecz którego zostały wykonane, a tworzą nowe łańcuchy. To oznacza, że ich wynik musi być skopiowany (użycie operatora `=`), bo bez tego nie zostanie zachowany.

Odpowiednikiem operatora `+=` w klasie `StringBuilder` jest metoda `Append`. Dodaje ona do bieżącej zawartości łańcucha łańcuch wskazany w argumencie. I robi to bez kopирования — modyfikuje bieżącą instancję obiektu. Analogicznie działają metody `Replace`, `Insert` i `Remove`. Korzystanie z klasy `StringBuilder` jest zatem wydajniejsze.

Z drugiej strony nie ma co popadać w przesadę. Jeżeli manipulacji łańcuchem nie jest więcej niż kilka czy kilkanaście, to różnica czasu wykonywania operacji w przypadku łańcucha typu `string` i `StringBuilder` jest niezauważalna, nawet na mnóstwo wydajnych urządzeniach. I dlatego w tej książce właściwie zawsze korzystam z typu `string`.

Typ wyliczeniowy

Typ wyliczeniowy jest implementowany przez klasę `System.Enum`, do jego definiowania będziemy jednak używać słowa kluczowego `enum`, którego składnia jest podobna jak w języku C/C++.

Typ wyliczeniowy jest stosowany do definiowania grupy powiązanych ze sobą stałych o całkowitych wartościach. Może być zdefiniowany jedynie w przestrzeni nazw lub jako pole klasy — jego definicja nie może znaleźć się wewnątrz metody:

```
public enum DniTygodnia:  
    byte{niedziela=1,poniedzialek,wtorek,sroda,czwartek,piatek,sobota};
```

Domyślnie wartość pierwszego elementu ustalana jest na 0, a każdy kolejny otrzymuje wartość o jeden większą. W powyższym przykładzie zamiast domyślnej wartości pierwszego elementu użyta została wartość 1. Każdy element typu wyliczeniowego może mieć w taki sposób przypisaną wartość.

Konkretny typ wyliczeniowy związany jest zawsze z jakimś typem liczbowym całkowitym. W naszym przykładzie jest nim `byte` (`System.Byte`). Zaskakujące może być jednak to, że wyrażenie `DniTygodnia.poniedzialek` nie jest wcale typu `byte`, a typu `DniTygodnia` (zdefiniowanego typu wyliczeniowego). Gdy chcemy go wykorzystać jako stałą, należy dodać jawne rzutowanie na typ `byte`, tj. `(byte)DniTygodnia.poniedzialek`, np.:

```
byte nrDniaTygodnia=(byte)DniTygodnia.poniedzialek;
```

Zadaniem typu wyliczeniowego nie jest jednak tworzenie zbioru stałych. Jest nim ograniczanie możliwych wartości zmiennych. W odróżnieniu od zmiennej typu `byte` zmienna typu `DniTygodnia` może przyjmować tylko i wyłącznie wartości określone w typie wyliczeniowym. Jeżeli zatem np. definiujemy metodę, której argumentem ma być dzień tygodnia, i chcemy uniknąć przypadkowych wartości prowadzących do błędu, powinniśmy jako argumentu użyć właśnie powyższego typu wyliczeniowego.

Definiowanie typu wyliczeniowego jest w istocie rozszerzaniem klasy `System.Enum`. W tworzonej w ten sposób klasie potomnej definiowane są publiczne pola statyczne typu zadeklarowanego w definicji. Takiego dziedziczenia nie można jednak wykonać jawnie, bo `System.Enum` jest tzw. klasą specjalną⁷.

Kolejna cecha szczególna typu wyliczeniowego mogąca wywołać konsternację u Czytelnika to fakt, że pomimo iż typ ten jest implementowany przez klasę, jest typu wartościowego (podrozdział „Typy wartościowe i referencyjne”). W istocie zdefiniowany przez nas typ `DniTygodnia` zachowuje się bardziej jak struktura niż klasa. Elementy określone w definicji typu wyliczeniowego (poniedziałek, wtorek itd.) są obiektami stałymi typu `DniTygodnia`, a wykonanie polecenia `DniTygodnia pn=DniTygodnia.poniedziałek;` lub `byte nrDniaTygodnia=(byte)DniTygodnia.poniedziałek;` oznacza utworzenie nowego obiektu i skopiowanie do niego wartości z oryginału, a nie utworzenie nowej referencji. Można więc w skrócie powiedzieć, że typ wyliczeniowy należy do grupy typów wartościowych.

Leniwe inicjowanie zmiennych

W .NET 4 w przestrzeni `System` pojawiła się nowa klasa o nazwie `Lazy`. Implementuje ona wzorzec leniwej inicjacji. We wzorcu tym zmienna opakowywana typem `Lazy` nie jest rzeczywiście inicjowana aż do momentu jej pierwszego użycia. Można to wykorzystać do uniknięcia tworzenia „na zapas” obiektu typu referencyjnego w przypadku, gdy o jego użyciu decyduje warunek sprawdzany dopiero podczas działania programu. Oszczędzimy dzięki temu pamięć i czas procesora. Użycie nowego „wrappera” jest bardzo proste:

```
Lazy<int> li = new Lazy<int>(()=>1); // deklaracja zmiennej i wskazanie funkcji
Console.WriteLine(li.Value.ToString()); // jeszcze niezainicjowana
Console.WriteLine("Odwołanie do zmiennej, li=" + li.Value); // leniwa inicjacja
Console.WriteLine(li.Value.ToString()); // już zainicjowana
```

W pierwszej linii jako argumentu konstruktora klasy `Lazy` używam funkcji zapisanej za pomocą wyrażenia lambda (podrozdział „Wyrażenia lambda” poniżej), zwracającej wartość 1. Jest to prosta funkcja, która zostanie użyta do zainicjowania zmiennej. Powyższy przykład jest oczywiście zbyt prosty, ale jego zadaniem jest tylko prezentacja idei leniwej inicjacji. W szczególności lepiej byłoby, gdyby leniwym typem była klasa, a nie struktura. Wówczas argumentem konstruktora powinna być funkcja-fabryka tworząca instancje owej klasy. Na poniższym listingu prezentuję to na przykładzie przycisku w aplikacji Windows Forms:

```
Lazy<Button> lb = new Lazy<Button>(() =>
{
    Button b = new Button();
    b.Text = "Leniwy przycisk";
    b.Left = 100;
```

⁷ Inne tego typu klasy to: `System.Value` będąca prototypem struktur, `System.Delegate` — klasa bazowa delegacji, które zostaną omówione, i `System.Array` — prototyp tablic. Po klasach specjalnych nie można dziedziczyć jawnie, a jedynie przez wykorzystanie odpowiednich konstrukcji (słów kluczowych `struct` i `delegate` dla struktur i delegacji oraz operatora `[]` dla tablic).

```
b.Top = 100;
b.Parent = this;
return b;
});
MessageBox.Show(lb.Value.ToString());
MessageBox.Show("Odwołanie do zmiennej, etykieta przycisku: \""
+ lb.Value.Text + "\"");
MessageBox.Show(lb.Value.ToString());
```

Metody

W C# nie jest możliwe definiowanie funkcji niebędących metodami jakiejś klasy. Funkcja może być wprawdzie statyczną składową klasy, ale tak czy inaczej zawsze jest metodą (tj. właśnie funkcją składową klasy zdefiniowaną w obrębie tej klasy). Na szczęście nie utrudni to nam nauki, zmuszając od razu do skoku na głębokie wody programowania obiektowego — pierwsze metody zdefiniujemy po prostu w obrębie istniejącej klasy (w naszym przypadku klasy `Program`). Będą to metody statyczne, bo statyczna jest metoda `Main`, z której będziemy je wywoływać. Statyczne, czyli takie, które można wywołać bez tworzenia instancji klasy, w której są zdefiniowane. Oznacza to, że do ich definicji dodamy modyfikator `static`.

Obok używanej przez nas do tej pory metody `Main` zdefiniujmy teraz samodzielnie własną metodę. Zaczniemy od prostej, nic nierobiącej metody wyróżnionej na listingu 3.1. Jej kod umieścimy gdziekolwiek wewnątrz klasy `Program`, ale nie wewnątrz innej metody lub konstruktora. Przykładowa metoda `Metoda` nie przyjmuje żadnych argumentów i nie zwraca wartości. Możemy ją wywołać, tak jak pokazano na listingu 3.1, wewnątrz metody `Main`.

Listing 3.1. Przykład metody i jej wywołanie

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

namespace JazykCS
{
    class Program
    {
        static void Metoda() // głowa metody, sygnatura
        {
            Console.WriteLine("Hello World!"); // ciało metody
        }

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

Słowo kluczowe `void` użyte do zaznaczenia, że funkcja nie zwraca żadnej wartości, zostało „pożyczone” z języka C++. W przeciwieństwie do Pascala i Delphi pusta lista argumentów nie zwalnia z używania nawiasów przy wywołaniu metody.

Komentarze w listingu 3.1 wskazują dwie części metody. Część pierwsza to głowa metody zawierająca zwracany przez nią typ wartości, nazwę metody i listę jej parametrów (w tym przykładzie lista ta jest pusta). Elementy te, a szczególnie nazwa metody i jej parametry, składają się na sygnaturę metody, po której jest ona rozpoznawana wśród innych metod klasy.

Przeciążanie metod

Skomplikujmy nieco nasz przykład, definiując drugą metodę o tej samej nazwie. Różnić się ona jednak będzie obecnością parametru. Będzie nim łańcuch określający treść wyświetlanego komunikatu. Zdefiniujmy zatem obok pierwszej metody `Metoda` drugą, tyle że z parametrem (listing 3.2).

Listing 3.2. Metody przeciążone i ich wywołanie

```
static void Metoda()
{
    Console.WriteLine("Hello World!");
}

static void Metoda(string tekst)
{
    Console.WriteLine(tekst);
}

static void Main(string[] args)
{
    Metoda();
    Metoda("Witaj, świecie!");
}
```

Język C# umożliwia definiowanie wielu metod o tych samych nazwach, pod warunkiem że różnią się parametrami (dzięki temu mają również inne sygnatury). Nazywa się to przeciążaniem metody (ang. *overload*). Niemożliwe jest natomiast definiowanie dwóch metod różniących się jedynie zwracanymi wartościami.

W formalnym języku informatyki rozróżnia się dwa pojęcia, które w tym opisie w zasadzie utożsamiam. Mam na myśli parametry i argumenty metod (funkcji). Parametry (lub inaczej parametry formalne) można rozumieć jako zmienne zdefiniowane w sygnaturze metody, a dokładniej w jej liście argumentów. Tam wskazywane są typ i nazwa poszczególnych parametrów. Podczas definiowania ciała metody nie wiemy, jaka będzie wartość poszczególnych parametrów. Ta znana jest dopiero w trakcie działania programu, w momencie wywołania metody. Wówczas z parametrem kojarzony jest argument (inaczej parametr aktualny), tj. wartość zmiennej wartościowej lub referencja do obiektu zmiennej referencyjnej, które wstawiamy w instrukcji wywołania metody, na odpowiedniej pozycji wewnętrz nawiasów znajdujących się za nazwą metody. Tym samym w czasie działania metody parametr przyjmuje wartość argumentu. Jeżeli

parametry są typu wartościowego, zmiana ich wartości podczas działania metody nie zostaje uwzględniona w wartości argumentu po jej zakończeniu — zmieniana jest bowiem tylko lokalna kopia argumentu⁸. W przypadku parametrów typu referencyjnego z argumentu kopowana jest tylko referencja do obiektu (bez jego klonowania), a więc wszelkie modyfikacje obiektu wewnątrz metody są widoczne także po jej zakończeniu. Do zagadnienia można podejść bardziej abstrakcyjnie. Bo czymże jest parametr, jeżeli nie miejscem w pamięci zarezerwowanym przez metodę, a argument — wartością zapisywaną w tym miejscu w momencie wywołania metody. Ja jednak, jak wcześniej zastrzegłem, nie będę zbytnio dbał o te rozróżnienia i w konsekwencji pojęcia „parametr” i „argument” będą przeze mnie utożsamiane.

Domyślne wartości argumentów metod — argumenty opcjonalne

Od wersji 4.0 języka C# możliwe jest wreszcie ustalanie domyślnych wartości parametrów metod. Pokazuję to w listingu 3.3, w którym widoczna jest zmodyfikowana metoda Metoda. Dodałem do niej drugi parametr, który ustala kolor, w jakim wyświetlany jest napis.

Listing 3.3. Metoda z domyślną wartością argumentu

```
static void Metoda(string tekst, ConsoleColor kolor = ConsoleColor.White)
{
    ConsoleColor bieżącyKolor = Console.ForegroundColor;
    Console.ForegroundColor = kolor;
    Console.WriteLine(tekst);
    Console.ForegroundColor = bieżącyKolor;
}
```

Dzięki temu przy wywołaniu tej metody drugi argument jest opcjonalny — jeżeli nie wystąpi w liście argumentów w instrukcji wywołania metody, jej drugi parametr przyjmie wartość domyślną i napis będzie wyświetlany na biało. Rozwiążanie to, znane od zawsze w C++, pozwala ograniczyć liczbę przeciążonych wersji metod. Parametry z domyślną wartością (inaczej argumenty opcjonalne) muszą być jednak zawsze definiowane jako ostatnie.

W naszym przykładzie możliwe jest teraz wywołanie metody Metoda zarówno z jednym, jak i z dwoma argumentami:

```
Metoda("Witaj, świecie!");
Metoda("Witaj, świecie!", ConsoleColor.Green);
```

Gdybyśmy zamiast modyfikować istniejącą wersję metody, zdefiniowali jej trzecią wersję, powstałyby niejednoznaczność. Wywołanie metody z jednym argumentem powałoby zarówno do jej jednoargumentowej wersji, jak i do wersji dwuargumentowej z pominiętym drugim argumentem. W takim przypadku wywoływana jest wersja, której sygnatura dokładnie pasuje do podanych argumentów, a więc jednoargumentowa.

⁸ Wyjątkiem jest sytuacja, w której do parametru dodajemy modyfikator `ref` lub `out` (podrozdział „Zwracanie wartości przez argument metody” w tym rozdziale).

Zwróćmy uwagę, że nawet nasze dwie wersje przeciążonej metody Metoda: bezargumentowa i dwuargumentowa, też są w zasadzie niepotrzebne. Bezargumentowej można by się pozbyć, a w zamian dodać wartość domyślną do pierwszego argumentu tekstu wersji dwuargumentowej.

Argumenty nazwane

Kolejną nowością języka C# w wersji 4.0 jest możliwość identyfikacji parametrów nie za pomocą ich kolejności, a przy użyciu ich nazw, np.:

```
Metoda(kolor: ConsoleColor.Green, tekst: "Witaj, świecie!");  
Metoda(tekst: "Witaj, świecie!", kolor: ConsoleColor.Green);
```

Najważniejszą moim zdaniem zaletą tego nowego rozwiązania jest czytelność kodu. Bez sięgania do definicji metody możemy ustalić, którym parametrom przypisywane są które argumenty. Tym samym trudniej też popełnić błąd w przypadku metod o dużej liczbie parametrów tego samego typu.

Wartości zwracane przez metody

Metoda może zwracać wartość. Na listingu 3.4 pokazuję prosty, używany już w poprzednim rozdziale przykład metody obliczającej kwadrat liczby podanej w argumentem. Pojawiło się w niej słowo kluczowe return. To właśnie następująca po nim wartość zostanie zwrócona przez metodę. Ponadto słowo return kończy działanie metody, a więc wszelkie polecenia występujące po instrukcji return nie są wykonywane⁹.

Listing 3.4. Metoda zwracająca wartość i przykład jej użycia

```
static private int Kwadrat(int arg)  
{  
    return arg * arg;  
}  
  
static void Main(string[] args)  
{  
    int wynik = Kwadrat(2);  
    Console.WriteLine(wynik.ToString());  
}
```

Zwracanie wartości przez argument metody

Problem pojawia się wtedy, gdy metoda oblicza nie jedną, ale kilka wartości. Dobrym przykładem jest metoda obliczająca dwa pierwiastki równania kwadratowego. Jak wówczas zwrócić wyniki do miejsca jej wywołania? Możliwości są co najmniej dwie. Po pierwsze, możemy zdefiniować strukturę, której polami będą wyniki działania metody

⁹ Słowo kluczowe return może się również pojawić w metodzie niezwracającej wartości. Wówczas oznacza wyłącznie miejsce, w którym działanie metody ma się zakończyć. W takiej sytuacji po tym słowie nie umieszcza się zwracanej przez metodę wartości.

i którą metoda będzie zwracać. Jeżeli wyniki są tego samego typu, może to być nawet zwykła tablica lub kolekcja. Drugim sposobem jest zwracanie wartości przez argumenty. Jak pisalem wcześniej, jeżeli argumenty są typu referencyjnego, zmiany stanu obiektu przeprowadzone wewnętrz metod widoczne są po jej zakończeniu. Inaczej jest w przypadku zmiennych typu wartościowego — do parametru kopiowana jest wartość argumentu; po tym ich wartości są od siebie niezależne. Język C# umożliwia jednak potraktowanie parametru typu wartościowego (np. zmiennej liczbowej) jak referencji. Wówczas wszystkie zmiany argumentów dokonane wewnętrz metody będą nadal aktualne po jej zakończeniu. W listingu 3.5 zawarłem przykład, w którym argumenty metody `zakresDouble` pobierane są przez wartości, natomiast w listingu 3.6 metoda ta pobiera już referencje. W obu przykładach metoda `Button1_Click` jest domyślną metodą zdarzeniową przycisku, który należy umieścić na formularzu. Z niej wywoływana jest metoda `zakresDouble`.

Listing 3.5. Przekazywanie argumentów do metody przez wartości

```
static private void zakresDouble(double min, double max)
{
    min = double.MinValue;
    max = double.MaxValue;
    Console.WriteLine("Liczby double mogą należeć do przedziału
("+min+","+max+"));
}

static void Main(string[] args)
{
    double min = 0, max = 0;
    zakresDouble(min, max);
    Console.WriteLine("Liczby double mogą należeć do przedziału
("+min+","+max+"));
}
```

Listing 3.6. Przekazywanie argumentów do metody przez referencje

```
static private void zakresDouble(ref double min, ref double max)
{
    min = double.MinValue;
    max = double.MaxValue;
    Console.WriteLine("Liczby double mogą należeć do przedziału
("+min+","+max+"));
}

static void Main(string[] args)
{
    double min = 0, max = 0;
    zakresDouble(ref min, ref max);
    Console.WriteLine("Liczby double mogą należeć do przedziału
("+min+","+max+"));
}
```

W obu przykładach komunikaty wyświetlane z wnętrza metody `zakresDouble` będą takie same. Te, które wyświetlane są po powrocie do metody `Main`, będą się jednak różniły. W pierwszym przypadku zobaczymy zera (zgodnie z inicjacją zmiennych `min`

i `max` w metodzie `Main`). W drugim po dodaniu słów kluczowych `ref` w definicji metody i miejscu jej wywołania zobaczymy w komunikacie wartości ustalone w metodzie `zakresDouble`.

Oprócz słowa kluczowego `ref` można użyć w tym samym kontekście słowa kluczowego `out`. Różnica polega na tym, że `out` wymusza zmianę wartości, a `ref` tylko ją dopuszcza. Zmienna oznaczona przez `ref` musi zostać przed przesaniem do metody zainicjowana, a oznaczona przez `out` — nie.

Delegacje i zdarzenia

Kolejne słowo kluczowe, na które chciałbym zwrócić uwagę w kontekście definiowania metod, to `delegate`. Dzięki niemu możliwe jest w C# uzyskanie zmiennych przechowujących odniesienia do metod bez jawnego używania wskaźników do metod.

Kod z listingu 3.7 należy umieścić wewnątrz jakiejś klasy, choćby `Program`. Zadeklarowana jest w nim delegacja `Delegacja` i metoda `Kwadrat`. Delegacja `Delegacja` jest typem, którego zmienne mogą przechowywać referencje do metod o określonej w nim sygnaturze (konkretnie do metod pobierających argument typu `int` i zwracających wartość typu `int`). Metoda `Kwadrat`, której definicja jest również widoczna na listingu 3.7, pasuje do delegacji `Delegacja`, co oznacza, że odwołanie do niej mogłoby być przechowywane w zmiennej typu `Delegacja`.

Listing 3.7. Definicja delegacji i zgodnej z nią metody

```
delegate int Delegacja(int arg);

static private int Kwadrat(int arg)
{
    return arg*arg;
}
```

Warto to powtórzyć: Delegacja nie jest zmienną, która może przechowywać adres metody, jest typem takiej zmiennej. Zanim zatem przypiszemy referencję do metody, musimy utworzyć zmienną typu `Delegacja`. Poniższy kod — należy umieścić go w jakiejś metodzie lub konstruktorze klasy — tworzy taką zmienną i zapisuje w niej odwołanie do metody `Kwadrat`, a następnie pokazuje, że wywołanie metody w tradycyjny sposób i poprzez delegację wygląda podobnie i daje identyczne efekty:

```
Delegacja ReferencjaDoMetodyKwadrat = Kwadrat;

int i = Kwadrat(2);
int j = ReferencjaDoMetodyKwadrat(2);
Console.WriteLine("i=" + i + ", j=" + j);
```

Ciekawą własnością delegacji jest to, że mogą przechowywać więcej niż jedną referencję do metod o zgodnej sygnaturze. Można sumować zmienne tego typu, co powoduje, że referencje do obu metod argumentów tej operacji przechowywane są w delegacji, która jest wynikiem tej operacji:

```

Delegacja ReferencjaDoMetodyKwadrat = Kwadrat;
Delegacja ReferencjaDoInnejMetody = InnaMetoda;
Delegacja ReferencjaDoWieluMetod = ReferencjaDoMetodyKwadrat +
ReferencjaDoInnejMetody;
ReferencjaDoWieluMetod += ReferencjaDoTrzeciejMetody;
int wynikOstatniejMetody = ReferencjaDoWieluMetod(2);

```

Metody wywoływanie *via* delegacja ReferencjaDoWieluMetod wykonywane są po kolejno. Jeżeli metody te zwracają wartość, wywołanie takie zwróci wartość ostatniej z dodanych metod. Nie powinno także dziwić użycie operatora `+=`. Oznacza on de facto dołożenie nowej referencji do zbioru przechowywanego w instancji delegacji. Działa również operator `-=`, który usuwa wskazaną referencję metody z tego zbioru.

Deklarację zmiennej będącej instancją delegacji może poprzedzać słowo kluczowe `event`:

```
event Delegacja ReferencjaKwadrat;
```

W naszym przypadku pole takie powinno być oznaczone jako statyczne (modyfikator `static`). Są to zdarzenia, które można deklarować jedynie jako pola klasy. Ponieważ ich typem jest delegacja, są zmiennymi składowymi, które mogą przechowywać referencje do metod i za pomocą których można te metody wywołać:

```

ReferencjaKwadrat += Kwadrat;
int k = ReferencjaKwadrat(2);
Console.WriteLine("k=" + k);

```

Na czym zatem polega różnica? Jak wspomniałem, zdarzeniami nie mogą być zmienne lokalne, a jedynie pola klasy lub struktury. Zasadnicza różnica między zdarzeniem a zwykłą delegacją pojawia się jednak dopiero wtedy, gdy do publicznych zdarzeń i delegacji próbujemy odwoływać się z innej klasy. Pokazuje to przykład z listingu 3.8. Zdefiniowana jest w nim klasa `Klasa`, a w niej typ delegacji `Callback` i metoda `Metoda`. W klasie tej zdefiniowane jest pole `DelegacjaMetodaZakonczona` będące delegacją typu `Callback` oraz zdarzenie tego samego typu o nazwie `ZdarzenieMetodaZakonczona`. Oba wywoływanie są tuż przed końcem działania metody. Oprócz klasy `Klasa` na listingu widoczna jest klasa strony o nazwie `Default`. W jej metodzie `Button1_Click` tworzona jest instancja klasy `Klasa`. Następnie do udostępnionych przez tę klasę delegacji i zdarzenia przypisywana jest ta sama metoda `obiekt_MetodaZakonczona`. W ten sposób niejako zapisujemy się, aby zostać powiadomieni o zakończeniu działania metody `obiekt.Metoda`. W konsekwencji po każdorazowym wywołaniu tej ostatniej metody tuż przed jej zakończeniem wywoływaną będzie metoda `obiekt_MetodaZakonczona`. W poniższym przykładzie będzie ona nawet wywoływana dwa razy: raz ze względu na delegację `DelegacjaMetodaZakonczona`, a drugi raz ze względu na zdarzenie `ZdarzenieMetodaZakonczona`¹⁰. Metoda `obiekt_MetodaZakonczona` nazywana jest metodą zdarzeniową, a zdarzeniem jest zakończenie metody.

¹⁰ Mechanizm zdarzeń wykorzystywany jest w aplikacjach desktopowych (zarówno Windows Forms, jak i WPF) oraz w aplikacjach internetowych *ASP.NET Web Forms* (rozdział 1.). Pozwala on na podejście do projektowania aplikacji, w którym programista zamiast implementować to, co aplikacja ma robić, określa raczej sposób reakcji aplikacji na działania użytkownika. Służą do tego metody zdarzeniowe, które wiążane są ze zdarzeniami reprezentującymi możliwe interakcje człowiek – aplikacja i aplikacja – aplikacja.

Listing 3.8. Ograniczenia w przypadku subskrypcji powiadomień zdarzeń

```
class Klasa
{
    public delegate void Callback(object sender, DateTime czasZakonczeniaMetody);

    public Callback DelegacjaMetodaZakonczona;
    public event Callback ZdarzenieMetodaZakonczona;

    public void Metoda()
    {
        Console.WriteLine("Metoda - początek");

        // tu długie działanie metody

        Console.WriteLine("Metoda - tuż przed końcem");

        if (DelegacjaMetodaZakonczona != null)
            DelegacjaMetodaZakonczona(this, DateTime.Now);
        if (ZdarzenieMetodaZakonczona != null)
            ZdarzenieMetodaZakonczona(this, DateTime.Now);

        Console.WriteLine("Metoda - koniec");
    }
}

class Program
{
    static private void obiekt_MetodaZakonczona(object sender, DateTime
        ↪czasZakonczeniaMetody)
    {
        Console.WriteLine("Zakonczona metoda obiektu typu " + sender.GetType().
            ↪Name + " (czas: " + czasZakonczeniaMetody.ToString() + ")");
    }

    static void Main(string[] args)
    {
        Klasa obiekt = new Klasa();

        // subskrypcja
        obiekt.DelegacjaMetodaZakonczona = obiekt_MetodaZakonczona;
        obiekt.ZdarzenieMetodaZakonczona += obiekt_MetodaZakonczona;

        // uruchomienie metody, która wywoła metodę zdarzeniową
        obiekt.Metoda();
    }
}
```

Listing 3.8 miał pokazać różnice między zwykłymi delegacjami a tymi zadeklarowanymi z dodatkowym słowem kluczowym `event` (zdarzeniami), a wydaje się, że takich różnic nie ma. Zwróćmy jednak uwagę na drobną różnicę w metodzie `Program.Main` w instrukcjach wiążących metodę zdarzeniową `obiekt_MetodaZakonczona` z delegacją i zdarzeniem zdefiniowanymi w klasie `Klasa`. W przypadku zwykłej delegacji stosujemy operator przypisania `=`, podczas gdy w przypadku zdarzenia — operator `+=`. Użycie modyfikatora `event` wprowadza bowiem ograniczenia w używaniu delegacji. Jedynymi

operacjami wykonywanymi na zdarzeniach spoza klas, w których są zdefiniowane, mogą być dodanie referencji operatorem `+=` lub jej usunięcie operatorem `-=`. Nic innego nie jest dozwolone (pojawia się błąd kompilatora). W szczególności nie jest możliwe użycie operatora `=`, który wymazałby wszystkie wcześniej przypisane do delegacji referencje metod (mogłyby one być używane wewnętrznie przez klasę Klasa). Ponadto o ile w przypadku zwykłej delegacji w metodzie `Program.Main` możemy wykonać operacje typu:

```
obiekt.DelegacjaMetodaZakonczona(obiekt, DateTime.Now); //symulacja zdarzenia  
if (obiekt.DelegacjaMetodaZakonczona == null)  
    Console.WriteLine("Delegacja nieprzypisana");
```

to w przypadku zdarzeń są one niemożliwe poza klasą, w której zdarzenie jest zdefiniowane:

```
//te instrukcje spowodują błąd kompilacji  
obiekt.ZdarzenieMetodaZakonczona(obiekt, DateTime.Now);  
if (obiekt.ZdarzenieMetodaZakonczona == null)  
    Console.WriteLine("Zdarzenie nieprzypisane");
```

Dzięki tym ograniczeniom zdarzenia mogą w bezpieczny sposób odgrywać rolę funkcji zwrotnych (ang. *callback*).

Wyrażenia lambda

Od wersji 3.0 języka C# można stosować tzw. wyrażenia lambda korzystające z nowego operatora `=>`, które mają zastąpić definiowanie anonimowych metod wprowadzonych do C# 2.0. Ogólna składnia wyrażeń lambda to:

```
(parametry) => wartość  
(parametry) => {instrukcja;}
```

A oto przykłady:

```
(int n) => n + 1  
(x, y) => x == y  
n => { Console.WriteLine(n.ToString()); }
```

W pierwszym przypadku mamy do czynienia z odpowiednikiem definicji metody przyjmującej jako argument wielkość typu `int` o nazwie `n` i zwracającej jej wartość zwiększoną o 1. Powyższe wyrażenie może być przypisane do delegacji opisującej typ metody z jednym argumentem typu `int` i zwracającej wartość typu `int`:

```
delegate int DIInc(int n);
```

i dalej używane jak normalnie zadeklarowana metoda. Drugie wyrażenie porównuje dwie wartości o niezadeklarowanych typach i zwraca wartość `true`, jeżeli są równe, a `false` w przeciwnym wypadku. Może być zatem przypisane do delegacji zadeklarowanej np. jako:

```
delegate bool DIsEqual(double x, double y);
```

Wreszcie trzecie wyrażenie przyjmuje argument dowolnego typu i pokazuje go w oknie komunikatu. Na listingu 3.9 pokazuję, w jaki sposób można definiować wyrażenia lambda, przypisywać je do delegacji i wykorzystywać.

Listing 3.9. Definicje wyrażeń lambda

```
delegate int DIInc(int n);
delegate bool DIsEqual(double x, double y);
delegate void DShow(int n);

static void Main(string[] args)
{
    Console.WriteLine("Wyrażenia Lambda:\n");

    DIInc Inc = (int n) => n + 1;
    Console.WriteLine("Inc(1)=" + Inc(1));

    DIsEqual IsEqual = (x, y) => x == y;
    int a = 10;
    int b = 20;
    Console.WriteLine("Czy równe a='"+a+"' i b='"+b+"'? "+(IsEqual(a, b) ? "Tak" : "Nie"));
    Console.WriteLine("Czy równe a='"+a+"' i a='"+a+"'? "+(IsEqual(a, a) ? "Tak" : "Nie"));

    DShow Show = n => { Console.WriteLine(n.ToString()); };
    Show(10);
}
```

Prawdziwe korzyści i cel wprowadzenia wyrażeń lambda do języka C# ujawnią się jednak niżej, gdy użyjemy ich w zapytaniach LINQ. Przedstawię wówczas zbiór rozszerzeń (to specjalny typ metody, który także zaraz opiszę) wywoływanych na rzecz kolekcji, których argumentami są delegacje. Dla przykładu klasa `Array`, tak jak i wszystkie pozostałe kolekcje, wyposażona została w metodę rozszerzającą `Min` zwracającą najmniejszy element z tablicy, na rzecz której została wywołana. Musimy jej tylko podpowiedzieć, jak ma mierzyć „małoś” elementów. I właśnie delegacja metody, która pomiar „małości” wykona, powinna być podana jako argument metody `Min` (listing 3.10).

Listing 3.10. Praktyczne wykorzystanie wyrażenia lambda

```
string[] słowa = { "czereśnia", "jabłko", "borówka", "wiśnia", "jagoda" };
int długośćNajkrótszego = słowa.Min(słowo => słowo.Length);
```

W powyższym kodzie argumentem metody `Min` jest wyrażenie lambda `słowo=>słowo.Length` (z kontekstu wynika, że przyjmuje ono zmienną typu `string`, a zwraca jej długość). Metoda `Min` wywołuje metodę podaną w argumencie dla każdego elementu tablicy i zwraca jej wartość dla tego elementu, dla którego jest ona najmniejsza. Mówiąc prościej, zwraca długość najkrótszego łańcucha w tablicy.

Dzięki wyrażeniu lambda uniknęliśmy mniej czytelnej konstrukcji zawierającej metodę anonimową:

```
int długośćNajkrótszego = słowa.Min(delegate(string słowo) { return słowo.Length; });
```

Typy wartościowe i referencyjne

C# jest językiem, w którym obiektowość wprowadzona jest w pełni konsekwentnie. Nie ma w nim typów prostych, znanych z C++ czy Javy, a wszystkie zadeklarowane przez nas zmienne są w rzeczywistości obiektami. Należy jednak zwrócić uwagę na to, że nie wszystkie obiekty są instancjami klas — w C# mamy bowiem do dyspozycji również struktury. Poza kilkoma ważnymi ograniczeniami¹¹ definiuje się je identycznie z klasami — z poziomu kodu różnią się one słowem kluczowym `struct` zastępującym słowo `class`. Jest jednak między nimi bardzo istotna różnica w sposobie użycia. Klassy związane są bowiem ze zmiennymi referencyjnymi, a struktury — ze zmiennymi typów wartościowych. Co to oznacza? W przypadku typów referencyjnych obiekt będący instancją klasy może powstać wyłącznie na stercie w wyniku użycia operatora `new`, np.:

```
DivideByZeroException d = new DivideByZeroException();
```

W przypadku typów wartościowych obiekty będące instancjami struktur mogą powstać wyłącznie na stosie. Programista nie ma tu żadnej dowolności. Zmienne wartościowe mogą być wprawdzie inicjowane za pomocą operatora `new`, jednak w praktyce częściej inicjuje się je za pomocą stałych, np.:

```
Int32 i1 = new Int32();
int i2 = new int();
int i = 1;
```

To nie miejsce ulokowania w pamięci jest jednak tą różnicą między strukturami i klasami, której programiści C# powinni być najbardziej świadomi. Ważna różnica ujawnia się bowiem przy próbie kopирования obiektów. Przyjrzyjmy się dwóm poleceniom:

```
DivideByZeroException e = d;
int j = i;
```

Pierwsze z nich prowadzi do utworzenia nowej referencji do wcześniej zdefiniowanego obiektu typu `DivideByZeroException`. W drugim tworzona jest nowa zmienna `j` typu `int`. W pierwszym przypadku nie powstaje jednak nowy obiekt — obie referencje, `d` i `e`, wskazują na tę samą instancję klasy `DivideByZeroException`. Innymi słowy, zmiana wartości `e` oznaczać będzie zmianę wartości `d`. W drugim przypadku zmienna `j` i zmieniona `i` są natomiast związane z zupełnie niezależnymi instancjami struktury `int`. Oznacza to, że zmiana wartości obiektu `i` nie pociągnie za sobą zmiany wartości `j`.

Zmienne wartościowe można w pełni utożsamiać z odpowiadającymi im instancjami struktur. Struktury (typy wartościowe) imitują w ten sposób zachowanie zmiennych prostych.

¹¹ Struktury nie mogą być rozszerzane ani same nie mogą dziedziczyć po innych klasach lub strukturach. Mogą natomiast implementować interfejsy. Pola struktury, poza polami statycznymi, nie mogą być inicjowane w miejscu definicji (pola inicjowane są przez kompilator wartości domyślnymi). Programista nie może też samodzielnie zdefiniować dla struktury bezargumentowego konstruktora — jest on przygotowywany automatycznie przez kompilator.

Kolejna różnica między obiektami będącymi instancjami klas i struktur dotyczy sposobu ich zwalniania. W przypadku struktur sprawa jest w miarę prosta. Obiekt tego typu usuwany jest w momencie wyjścia poza zakres nawiasu {}, w którym został zdefiniowany. Jeżeli zatem zdefiniujemy zmienną i typu int w metodzie, to w momencie zakończenia tej metody pamięć zajmowana przez tę zmienną zostanie zwolniona. Wartość tej zmiennej może być oczywiście skopiowana i przekazana np. przez wartość zwracaną przez metodę, ale wówczas tworzone są kopie obiektu, które nie przedłużają życia oryginału. Inaczej sprawa wygląda w przypadku obiektów — instancji klas. Za ich zwalnianie z pamięci odpowiedzialny jest odśmieacz (ang. *garbage collector*). Zasada jego działania jest prosta. Cyklicznie przeszukuje pamięć sterty w poszukiwaniu obiektów, które nie są wskazywane przez żadną referencję. Jeżeli taki obiekt zostanie znaleziony, jest usuwany. Użytkownik nie ma na to w zasadzie żadnego wpływu (nie ma operatora delete), ale też nie musi się tym martwić. Dzięki temu nie ma niebezpieczeństwstwa wycieku pamięci.

Nullable

Jedną z różnic między typem wartościowym a referencyjnym jest to, że tylko ten drugi może mieć wartość null. Czasem możemy jednak chcieć zdefiniować np. liczbę całkowitą typu int, mogącą przyjmować dowolne wartości dodatnie, ujemne i zero, ale oprócz nich także wartość null, której możemy użyć np. do zasygnalizowania sytuacji, gdy jakieś obliczenia nie zostały jeszcze wykonane lub się nie udało. Możemy wówczas skorzystać z „opakowania” w postaci Nullable<int>. Oto przykład:

```
Nullable<int> ni = 1;
int i1;
if (ni.HasValue)
{
    i1 = ni.Value;
}
else
{
    i1 = default(int);
}
int i2 = ni.GetValueOrDefault();
Console.WriteLine("i1=" + i1 + ", i2=" + i2);
```

Zdefiniowaliśmy zmienną typu Nullable<int> i zainicjowaliśmy ją wartością 1. Widać więc od razu, że istnieje niejawną konwersja z typu int na typ Nullable<int> (lub ogólniej z typu T na Nullable<T>). Równie dobrze do inicjacji moglibyśmy użyć słowa kluczowego null. O tym, czy zmienna ma wartość, czy jest „pusta” (tj. właśnie równa null), możemy się przekonać, sprawdzając jej właściwość HasValue. Jeżeli zmienna nie jest pusta, jej wartość możemy odczytać, korzystając z właściwości tylko do odczytu Value. Udostępnia ona „opakowaną” przez typ Nullable<int> zasadniczą zmienną typu int.

Obie czynności, a więc sprawdzenie, czy zmienna ma wartość, i ewentualne jej odczytanie, realizuje metoda GetValueOrDefault. W wersji bez argumentu, jeżeli zmieniona równa jest null, metoda zwraca domyślną wartość typu (uzyskiwaną słowem kluczowym default). Wersja z argumentem pozwala na ustalenie własnej zwracanej

przez metodę wartości — oczywiście w przypadku braku wartości odpytywanej zmiennej.

W praktyce zamiast korzystać z właściwości `HasValue`, zazwyczaj stosuje się porównanie do wartości `null`, np.:

```
int i3 = (ni == null) ? -1 : ni.Value;
```

Tę konstrukcję można zresztą skrócić, jeżeli użyjemy operatora `??`, np.:

```
int i4 = ni ?? -1;
```

Powyższa instrukcja jest w pełni równoważna tej poprzedniej, a więc zwraca wartość odczytaną z `Value` lub `-1`, jeżeli ta pierwsza nie jest określona.

Warto również wiedzieć, że zamiast jawnego używania typu `Nullable<int>` mamy opcję skorzystania z wygodnego aliasu w postaci znaku zapytania za typem wartościowym:

```
int? ni=1;
```

Pudełkowanie

Zmienne typów wartościowych można zamknąć w pudełku także w inny sposób, korzystając przy tym z jego klasy bazowej `object`, będącej zresztą typem referencyjnym. Proces ten nazywa się *boxing*, co należy przetłumaczyć jako pudełkowanie. Nie potrzeba do tego żadnych specjalnych operatorów. Wystarczy zwykłe przypisanie:

```
int i = 1;
object o = i;
Console.WriteLine(o.GetType().ToString()); //zwraca System.Int32
```

Wartość zmiennej całkowitej, a więc zmiennej typu wartościowego, zamkniętej w zmiennej typu `object`, która jest typu referencyjnego, kopowana jest ze stosu na stertę. Nowa zmienna zachowuje informacje o typie oryginalnej zmiennej, o czym można się przekonać, korzystając z metody `GetType`.

Oryginalną wartość można wydobyć z pudełka, stosując *unboxing* (nie odważ się tłumaczyć tego terminu). W praktyce oznacza to rzutowanie na typ `int`, ale musi to być rzutowanie jasne. W istocie jest to jednak kopowanie wartości ze sterty z powrotem na stos.

```
int j = (int)o;
Console.WriteLine(j.ToString());
```

Do czego używać pudełkowania? W .NET 1.0 i 1.1 było ono fundamentem kolekcji, które potrafiły przechowywać tylko referencje typu `object`. Bez pudełkowania przechowywanie instancji struktur w nieparametrycznych kolekcjach wiążałoby się z utratą informacji o obiekcie, co z kolei niweczyłoby cały pomysł. Obecnie, gdy można używać typów parametrycznych, znaczenie pudełkowania zmalało, szczególnie że jego użycie wiąże się z brakiem ścisłej kontroli typów oraz spadkiem wydajności.

Typy dynamiczne

C# jest językiem bezpiecznym. Dotychczas wiązało się to m.in. ze ścisłą kontrolą typów. W C# 4.0 ta zasada została nieco nagięta. Możliwe jest bowiem definiowanie zmiennych, których typ nie jest ustalony w trakcie komplikacji. Co więcej, nie jest on kontrolowany — może zostać zmieniony przy każdym przypisaniu do tej zmiennej nowej wartości. Zmienne te nazywane są zmiennymi dynamicznymi, a do ich deklaracji używane jest nowe słowo kluczowe `dynamic`. Od razu należy jednak zastrzec, że używanie zmiennych dynamicznych nie powinno stać się regułą. Ich nadużywanie jest zdecydowanie szkodliwe. Powinny być używane tylko w wyjątkowych sytuacjach.

Słowo kluczowe `dynamic` pozwala na definiowanie zmiennych lokalnych, pól, właściwości, indeksatorów czy wartości pobieranych i zwracanych przez metody. We wszystkich tych przypadkach w instrukcjach przypisania zawieszana jest statyczna kontrola typów. Skompiluje się zatem fragment kodu widoczny na listingu 3.11. Co więcej, jego wykonanie nie spowoduje wyjątku.

Listing 3.11. Dynamiczna zmiana typu zmiennej

```
dynamic o; // nie działa IntelliSense
o = 5; Console.WriteLine(o.ToString() + ", " + o.GetType().FullName);
o = 5L; Console.WriteLine(o.ToString() + ", " + o.GetType().FullName);
o = "Helion"; Console.WriteLine(o.ToString() + ", " + o.GetType().FullName);
o = 1.0f; Console.WriteLine(o.ToString() + ", " + o.GetType().FullName);
o = 1.0; Console.WriteLine(o.ToString() + ", " + o.GetType().FullName);
```

Zwróćmy uwagę, że typ, jaki posiada zmienna `o`, ustalany jest w momencie realizacji instrukcji przypisania, tzn. dopiero w momencie działania programu. To je różni od zmiennych zadeklarowanych z użyciem słowa kluczowego `var`, których typ znany jest już w momencie komplikacji. Kolejna różnica względem `var` to fakt, że typ zmiennej ustalony w momencie pierwszego przypisania (inicjacji) nie musi być zachowany. Następne przypisanie może go zmienić, czego dowodzą komunikaty wyświetlane podczas wykonywania kodu z listingu 3.11.

Jak wspomniałem, możliwe jest definiowanie nie tylko lokalnych zmiennych dynamicznych, ale również takich pól i właściwości. Także metoda może zwracać obiekt typu `dynamic`. Oznacza to np., że w zależności od decyzji użytkownika podjętej już w momencie działania programu metoda zwraca liczbę 5 typu `int`, napis `Helion` lub referencję do przycisku (listing 3.12).

Listing 3.12. Typ `dynamic` może być użyty we wszystkich kontekstach

```
class Klasa
{
    public delegate void Callback(object sender, DateTime czasZakonczeniaMetody);

    public Callback DelegacjaMetodaZakonczona;
    public event Callback ZdarzenieMetodaZakonczona;

    public void Metoda()
    {
```

```
Console.WriteLine("Metoda - początek");
// tu długie działanie metody
Console.WriteLine("Metoda - tuż przed końcem");
if (DelegacjaMetodaZakonczona != null)
    DelegacjaMetodaZakonczona(this, DateTime.Now);
if (ZdarzenieMetodaZakonczona != null)
    ZdarzenieMetodaZakonczona(this, DateTime.Now);

        Console.WriteLine("Metoda - koniec");
    }
}

class Program
{
    ...

    static dynamic obiekt = 1; // pole zainicjowane obiektem typu int

    enum Typ { Int, Long, String, Float, Double, DivideByZeroException, Pole,
    ↪InstancjaKlasy };

    dynamic Obiekt // właściwość
    {
        get
        {
            return zwrocObiekt();
        }
        set
        {
            obiekt = value;
        }
    }

    static dynamic zwrocObiekt(Typ ktoryTyp = Typ.Int) // wartość zwracana przez metodę
    {
        dynamic wartosc;
        switch (ktoryTyp)
        {
            case Typ.Int: wartosc = 5; break;
            case Typ.Long: wartosc = 5L; break;
            case Typ.String: wartosc = "Helion"; break;
            case Typ.Float: wartosc = 1.0f; break;
            case Typ.Double: wartosc = 1.0; break;
            case Typ.DivideByZeroException: wartosc=new DivideByZeroException(); break;
            case Typ.Pole: wartosc = obiekt; break;
            case Typ.InstancjaKlasy: wartosc = new Klasa(); break;
            default: wartosc = null; break;
        }
        return wartosc;
    }

    static void Main(string[] args)
    {
        for (Typ typ = Typ.Int; typ <= Typ.InstancjaKlasy; typ++)
        {
            try
            {
                dynamic o = zwrocObiekt(typ);
            }
        }
    }
}
```

```
Console.WriteLine("Obiekt: " + o.ToString() + ", typ: " +
    o.GetType().FullName);
o.Metoda(); // tu pojawi się wyjątek
}
catch (Exception exc)
{
    ConsoleColor bieżącyKolor = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("Błąd: " + exc.Message);
    Console.ForegroundColor = bieżącyKolor;
}
}
```

W poniższym listingu znajduje się odwołanie do egzemplarza zmiennej wyliczeniowej, która w pętli przebiega wszystkie możliwe wartości (a nawet wychodzi poza zakres), kopując różnego typu obiekty.

Warto sobie uświadomić, że podobny efekt można było uzyskać już dawniej. Umożliwia to typ `object` — klasa bazowa wszystkich typów — który może być traktowany jak odpowiednik wskaźnika na `void` w C++. Korzystając z typu `object`, na który można rzutować dowolny typ platformy .NET, możemy napisać kod widoczny na listingu 3.13, który jest przecież podobny do tego z listingu 3.11. Ale choć rzeczywiście efekt wykonania obu fragmentów kodu jest bardzo podobny, ich działanie jest zupełnie inne. Obiekt typu `dynamic` rzeczywiście zmienia swój typ. Do zmiennej typu `object`, która jest typu referencyjnego, zapisywane są referencje kolejnych obiektów (stare obiekty usuwane są przez *garbage collector*). Przy tym w przypadku obiektów typu wartościowego dochodzi proces pudełkowania, opisany wyżej.

Listing 3.13. Użycie typu `object` może dać podobne efekty jak zastosowanie typu `dynamic`

```
object o;
o = 5; Console.WriteLine(o.ToString() + ", " + o.GetType().FullName);
o = 5L; Console.WriteLine(o.ToString() + ", " + o.GetType().FullName);
o = "Helion"; Console.WriteLine(o.ToString() + ", " + o.GetType().FullName);
o = 1.0f; Console.WriteLine(o.ToString() + ", " + o.GetType().FullName);
o = 1.0; Console.WriteLine(o.ToString() + ", " + o.GetType().FullName);
```

Możliwość swobodnej i wielokrotnej zmiany typu zmiennej dynamicznej nie jest jednak zaletą. To raczej cena, jaką trzeba zapłacić za korzyści płynące z braku kontroli typów. Jedną z nich jest łatwość, z jaką na rzecz dynamicznych obiektów można wywoływać metody, odczytywać lub przypisywać ich właściwości. Wszystko to oczywiście pod warunkiem, że programista rzeczywiście wie, z jakimi obiektami ma do czynienia. Tak naprawdę to na niego przerzucona jest teraz kontrola typów. Założmy dla przykładu, że dysponujemy biblioteką DLL, która nie jest ładowana statycznie, i w ten sposób kompilator nie może rozpoznać automatycznie jej zawartości. Przed pojawieniem się platformy .NET 4.0 i C# 4.0 zmuszeni bylibyśmy do korzystania z dynamicznego rozpoznawania typów (mechanizm *reflection* platformy .NET). Jeżeli jednak jesteśmy pewni, że biblioteka zawiera klasę o nazwie `Klasa`, a klasa ta zawiera metodę o nazwie `Metoda` o konkretnej sygnaturze, to korzystając z obiektu typu `dynamic`, bez weryfikowania tych faktów możemy utworzyć instancję tej klasy i wywołać jej metodę

(z podobnym scenariuszem mamy do czynienia np. wtedy, gdy korzystamy z obiektów COM). A nawet jeżeli klasy lub metody jednak nie będzie tam, gdzie się ich spodziewamy, nic bardzo strasznego się nie stanie — po prostu zostanie zgłoszony wyjątek. W końcu implementacja dynamicznego typu również oparta jest na mechanizmie *reflection*.



Wskazówka

Późne wiązanie typu zmiennej wiąże się z brakiem wsparcia IntelliSense podczas edycji kodu. Możliwe jest jednak instalowanie dodatkowych narzędzi (np. ReSharper), dzięki którym pewne informacje staną się dostępne.

Należy wyraźnie rozróżnić sytuacje, w których typ jest trudny do ustalenia, jak np. typ kolekcji zwracanej przez zapytanie LINQ, od sytuacji, w których ustalenie typu w ogóle nie jest możliwe przed uruchomieniem aplikacji. W sytuacjach pierwszego rodzaju należy bezwzględnie korzystać ze słowa kluczowego `var`. Warto powtórzyć jeszcze raz, że różni się ono od `dynamic` tym, że typ zmiennej zadeklarowanej z użyciem słowa kluczowego `var` jest ustalany już w trakcie komplikacji na podstawie wartości przypisanej do zmiennej podczas inicjacji, a następnie nie ulega zmianie. W przypadku słowa kluczowego `dynamic` typ nie jest znany. Jest ustalany dopiero w momencie działania programu. Co więcej, nie musi być także zachowany.

W praktyce nadal mam wątpliwości co do zasadności wprowadzania typu dynamicznego (i idących za tym zmian w platformie .NET¹²). Z pewnością można dzięki niemu uniknąć długiego kodu opartego na refleksji¹³ i w prostszy sposób poradzić sobie z obiekttami, których typ ustalany jest dopiero w trakcie działania programu. Taka sytuacja często ma miejsce w przypadku prób korzystania z kodu niezarządzanego, w tym z technologii COM. Nie mam jednak wątpliwości, że należy unikać typu dynamicznego we wszystkich pozostałych sytuacjach.

Sterowanie przepływem

Umiejętność programowania była dotychczas kojarzona głównie ze znajomością instrukcji sterujących języka. Obecnie środek ciężkości przesuwa się w kierunku poznawania bogatych bibliotek klas związanych z Java, Delphi lub C#. Mimo to nadal nie da się uniknąć nauki instrukcji warunkowych czy pętli.

Instrukcja warunkowa `if..else`

Składnia tej instrukcji jest następująca: `if(warunek) instrukcja;`. Oznacza ona tyle, że `instrukcja` jest wykonywana jedynie wtedy, gdy `warunek` ma wartość równą `true`. Składnia rozszerzona to `if(warunek) instrukcja; else alternatywna_instrukcja;`. Tu `alternatywna_instrukcja` jest realizowana, gdy `warunek` nie jest spełniony. W listingu 3.14 zawarłem przykłady użycia tej instrukcji.

¹² Warto w tym kontekście przeszukać MSDN z hasłem DLR (ang. *Dynamic Language Runtime*).

¹³ Refleksja to mechanizm platformy .NET dający możliwość odczytywania typów obiektów i ich składowych (metod, właściwości i pól) w trakcie działania programu.

Listing 3.14. Przykłady wykorzystania instrukcji if i if..else

```
Random r=new Random();
int n=r.Next(8);
Console.WriteLine(n.ToString());

// Składnia podstawowa if
if (n<6) Console.WriteLine("Wylosowana liczba jest mniejsza od 6.");

// Składnia rozszerzona if..else
if (n<=4) Console.WriteLine("Wylosowana liczba jest mniejsza lub równa 4.");
else Console.WriteLine("Wylosowana liczba jest większa od 4.");
```

Instrukcja wyboru switch

Listing 3.15 zawiera przykładowy kod korzystający z instrukcji wyboru switch.

Listing 3.15. Przykład wykorzystania instrukcji switch

```
Random r=new Random();
int n=r.Next(8);
string opis;
switch(n)
{
    case 1: opis="niedziela"; break;
    case 2: opis="poniedziałek"; break;
    case 3: opis="wtorek"; break;
    case 4: opis="środa"; break;
    case 5: opis="czwartek"; break;
    case 6: opis="piątek"; break;
    case 7: opis="sobota"; break;
    default: opis="błąd!"; break;
}
Console.WriteLine("Dzień tygodnia: " + n + ", " + opis);
```

W pierwszej linii tworzymy obiekt — generator liczb pseudolosowych (konstruktor domyślny klasy Random inicjuje generator za pomocą zależnego od czasu ziarna). Następnie losowo wybraną liczbę z zakresu [0, 8) przypisujemy do zmiennej całkowitej n. Deklarujemy także łańcuch opis. Założymy, że n jest numerem dnia tygodnia. Dzięki instrukcji switch możemy w łatwy sposób przypisać do łańcucha opis nazwę dnia tygodnia w zależności od wartości liczby n. Każdy z opisanych w konstrukcji switch przypadków rozpoczyna się od słowa kluczowego case i obowiązkowo kończy się polecienniem break. Między nimi mogą znaleźć się dowolne instrukcje C#.

Za pomocą case określiliśmy czynności dla wartości od 1 do 7. Co pewien czas wylosowana będzie jednak również liczba 0. Wówczas wykorzystana zostanie linia rozpoznająca się od słowa kluczowego default. Jej obecność nie jest w C# obowiązkowa. Jeżeli jednak jej zabraknie, w powyższym kodzie kompilator zgłosi błąd informujący, że w ostatniej linii zmienna opis jest wykorzystywana, choć może nie być zainicjowana. W naszym przykładzie taka sytuacja może rzeczywiście mieć miejsce, gdy n będzie miało wartość 0.

Pętle

Pętle są przykładem zagadnienia, o którym można powiedzieć krótko (tabela 3.6) lub mówić bez końca, omawiając poszczególne rodzaje pętli oraz ich typowe i nietypowe zastosowania. Wybieram pierwszy sposób, ponieważ w kodach, które znajdują się w następnych rozdziałach, pętle będą wykorzystywane w sposób naturalny i typowy.

Tabela 3.6. Typy pętli dostępne w C#

Typ pętli	Przykład
for (<i>inicjacja indeksu; warunek; inkrementacja</i>) <i>instrukcja</i> ; Typowa pętla: for (int i=0;i<ilosc;i++) <i>instrukcja</i> ; Pętla nieskończona: for (;true;) <i>instrukcja</i> ; <i>warunek</i> <i>instrukcja</i> ;	<pre>long silnia(byte arg) { if (arg==0) return 0; long wartosc=1; for(byte i=1;i<=arg;i++) wartosc*=i; return wartosc; }</pre> <pre>long najwiekszyDzielnik(long arg) { long dzielnik=arg-1; while(arg%dzielnik!=0) dzielnik--; return dzielnik; }</pre> <pre>Random r=new Random(); int n=r.Next(8); // Zadanie komputera to odgadnąć // liczbę z zakresu od 0 do 7 int z; int licznik=0; do { licznik++; z=r.Next(8); } while (n!=z); Console.WriteLine("Komputer zgadł liczbę " + z + " po " + licznik + " próbach!");</pre>

Wszystkie powyższe pętle są wykonywane, dopóki *warunek* ma wartość true. *Instrukcja* może być również blokiem instrukcji otoczonych operatorem zakresu { }.

Różnica między pętlami while i do..while polega na tym, że w pierwszej *warunek* sprawdzany jest już przed wykonaniem *instrukcji*, a w do..while dopiero po jej wykonaniu. Z tego wynika, że stosując pętlę do..while, mamy gwarancję, iż *instrukcja* zostanie wykonana przynajmniej raz nawet wtedy, jeżeli *warunek* nigdy nie zostanie spełniony.

Instrukcje wykonywane w każdej iteracji pętli mogą być przerwane za pomocą poleceń break i continue. Słowo kluczowe break przerywa w ogóle działanie pętli i przechodzi do instrukcji znajdującej się bezpośrednio za nią. Natomiast continue przerywa działanie bieżącej iteracji i rozpoczyna następną. W przypadku pętli for oznacza to odpowiednie zwiększenie indeksu, zgodnie z poleceniem inkrementacji (listing 3.16).

Listing 3.16. Przykład pętli for z instrukcją continue

```
for (int i=-1; i<=1; i++)
    for (int j=-1; j<=1; j++)
    {
        if (i==0 && j==0) continue; //pomijamy przypadki, gdy obie zmienne równe są 0
        Console.WriteLine("i=" + i + ", j=" + j);
    }
```

W powyższym przykładzie pokazanych zostanie tylko osiem komunikatów. Przypadek, gdy oba indeksy są równe 0, zostanie bowiem pominięty. Gdybyśmy zamiast continue wstawili break, pominięte zostałyby iteracje pętli wewnętrznej, następujące po iteracji z zerowymi indeksami, ale pętla zewnętrzna byłaby kontynuowana.



Istnieje jeszcze jeden rodzaj pętli — foreach — który związany jest ściśle z kolekcjami i dlatego zostanie omówiony w dalszej części rozdziału, która ich dotyczy.

Wyjątki

Ważnym elementem nowoczesnego programowania jest korzystanie z wyjątków (ang. *exceptions*) do sygnalizowania błędów. W C# przez wyjątek można rozumieć obiekt, który przenosi wiadomość o błędzie z miejsca jego wystąpienia, tzn. z miejsca, w którym wyjątek jest zgłoszany, do miejsca jego obsługi.

W poniższym przykładzie wyjątek jest generowany podczas wykonywania operacji dzielenia, ponieważ dzielnik (zmienna x) jest równy zeru¹⁴. Obszar, w którym śledzone jest ewentualne wystąpienie wyjątków, jest otoczony blokiem następującym po słowie kluczowym try. Ewentualny wyjątek, który może tam powstać, zostanie przekazany do sekcji catch wykonywanej jedynie w takim przypadku. Należy mieć świadomość tego, że po przejściu do sekcji catch nie ma już możliwości powrotu do sekcji try; pozostała część znajdujących się tam poleceń nie zostanie wykonana, a po wykonaniu poleceń w sekcji catch wątek przejdzie do poleceń znajdujących się po konstrukcji try..catch (listing 3.17).

Listing 3.17. Przykład konstrukcji try..catch przechwytyjącej wyjątek dzielenia przez 0

```
try
{
    int x=0;
    int y=1/x;
}
catch(Exception exc)
{
```

¹⁴ Konieczne było dzielenie przez zmienną równą 0 zamiast jawnie przez 0, żeby oszukać kompilator, który próbuje wykrywać tego typu błędy. Warto również wiedzieć, że dzielenie przez 0 jest niedopuszczalne w liczbach całkowitych, ale np. w przypadku liczb double wyjątek nie zostałby zgłoszony, a zmienna przyjęłaby wartość równą Infinity lub -Infinity.

```
        Console.WriteLine("Wyjątek: " + exc.Message);
    }
```

Po sekcji try może znajdować się kilka sekcji catch wychwytyjących poszczególne typy wyjątków. Nazywa się to filtrowaniem wyjątków. Na listingu 3.18 pokazany jest prosty przykład działania takiego mechanizmu.

Listing 3.18. Przykład wielokrotnej sekcji catch przechwytyjącej coraz bardziej ogólne klasy wyjątków

```
try
{
    int x=0;
    int y=1/x;
}
catch(DivideByZeroException exc)
{
    Console.WriteLine("Dzielenie przez zero (" + exc.Message + ")");
}
catch(ArithmeticException exc)
{
    Console.WriteLine("Błąd arytmetyki (" + exc.Message + ")");
}
catch(Exception exc)
{
    Console.WriteLine("Wyjątek: " + exc.Message);
}
```

Sekcje catch powinny być ułożone od wychwytyjącej najbardziej szczegółową klasę wyjątku do najbardziej ogólnej, tj. od klasy wielokrotnie dziedziczącej po `Exception` do samego `Exception`. W przeciwnym razie gdybyśmy na pierwszym miejscu ustawiли sekcję catch przechwytyującą wyjątki typu `Exception`, to ze względu na to, że mogą być na ten typ niejawnie rzutowane wszystkie inne klasy wyjątków (jako na ich klasę bazową), automatycznie wychwytywałaby ona wszystkie wyjątki, nie pozostawiając nic dla kolejnych sekcji catch.



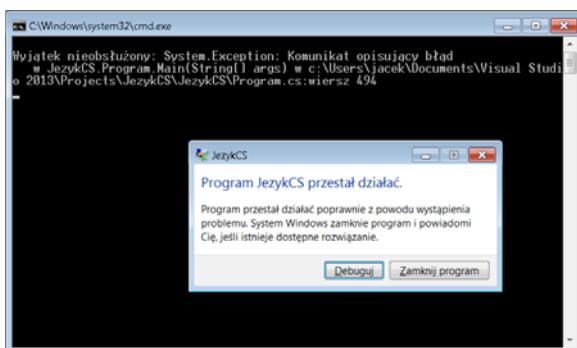
Wskazówka

W aplikacjach „okienkowych” Windows Forms wystąpienie nieobsłużonego wyjątku nie oznacza zakończenia działania aplikacji (vide przycisk *Kontynuuj* w oknie komunikatu informującego o wyjątku). Zakończy się jednak działanie metody, w której on wystąpił. W aplikacjach ASP.NET nieobsłużony wyjątek spowoduje pojawienie się (w sesji użytkownika, w której wyjątek wystąpił) strony informującej o błędzie, ale aplikacja nadal będzie działać. Natomiast w aplikacjach konsolowych wystąpienie nieobsłużonego wyjątku (przy uruchomieniu aplikacji bez debugowania, *Ctrl+F5*) oznacza nieuniknione zakończenie działania aplikacji (rysunek 3.1).

Składnia konstrukcji obsługującej wyjątek może być rozszerzona o sekcję `finally`. Polecenia z tej sekcji wykonywane są w każdym przypadku, bez względu na to, czy wyjątek wystąpił, czy nie. Wykonywane są także wtedy, gdy w sekcji `catch` znajdzie się instrukcja `return` lub inne polecenie kończące działanie metody. W razie wystąpienia wyjątku polecenia z tej sekcji wykonywane są po poleceniach z sekcji `catch`. Typowym wykorzystaniem sekcji `finally` jest zwalnianie zasobów, np. zamknięcie pliku otwartego w sekcji `try`. Na listingu 3.19 pokazuję dość banalny przykład wykorzystania sekcji `finally`.

Rysunek 3.1.

Nieobsłużony wyjątek
przechwycony przez
platformę .NET



Listing 3.19. Przykład wykorzystania sekcji *finally*

```
try
{
    int x=0;
    int y=1/x;
}
catch(DivideByZeroException exc)
{
    Console.WriteLine("Dzielenie przez zero (" + exc.Message + ")");
    return;
}
catch(ArithmetичException exc)
{
    Console.WriteLine("Błąd arytmetyki (" + exc.Message + ")");
    return;
}
catch(Exception exc)
{
    Console.WriteLine("Wyjątek: " + exc.Message);
    return;
}
finally
{
    Console.WriteLine("Kod wykonywany w każdym przypadku");
}
```

Do zgłaszania wyjątku służy słowo kluczowe `throw` z następującą po nim referencją do obiektu wyjątku. W momencie, w którym wykryjemy krytyczny błąd programu i będziemy chcieli go zasygnalizować zgłoszeniem wyjątku, należy zatem w kodzie umieścić instrukcję analogiczną do poniższej:

```
throw new Exception("Komunikat opisujący błąd");
```

Dyrektywy preprocesora

Termin „dyrektywy preprocesora” może być mylący dla osób znających dobrze C++, bo w odróżnieniu od C++ w C# nie ma osobnego etapu preprocesora podczas komplikacji kodu. Tzw. dyrektywy preprocesora są zatem analizowane równocześnie z analizą

kodu języka C#. Wobec tego w C# nie można używać dyrektyw do tworzenia makr; ich głównym zadaniem jest warunkowa kompilacja kodu.

Kompilacja warunkowa — ostrzeżenia

Poniżej znajduje się przykład wykorzystania dyrektywy do generowania ostrzeżenia, które przypomina, aby przed rozpowszechnieniem programu skompilować go z opcjami usuwającymi informacje przeznaczone dla debugera i włączoną optymalizacją. Poniższe dyrektywy można umieścić na samym początku pliku z kodem źródłowym:

```
#if DEBUG
    #warning Pamiętaj, aby przed publikacją skompilować w wersji "Release"
#endif
```

Symbol preprocessora DEBUG, którego obecność sprawdzamy w powyższym kodzie, może być zdefiniowany przez środowisko Visual Studio, jeżeli zaznaczymy opcję *Define DEBUG constant* na zakładce *Build* własności projektu (ostatnia pozycja w menu *Project*). Dyrektywa `#if` pełni podobną funkcję jak instrukcja `if` z języka C# — podczas komplikacji uwzględniany jest kod znajdujący się między `#if` i `#endif`, jeżeli spełniony jest wskazany warunek (np. obecność symbolu preprocessora). Instrukcja ta może mieć również rozbudowaną postać:

```
#if DEBUG
    #warning Kompilacja "debug"
#else
    #warning Kompilacja "release"
#endif
```

Należy podkreślić różnice między dyrektywą `#if` a instrukcją warunkową `if`. Sprawdzenie warunku znajdującego się za dyrektywą `#if` odbywa się podczas komplikacji, a nie w trakcie działania programu. Jeżeli zatem w metodzie `Main` umieścimy taką instrukcję:

```
#if DEBUG
    Console.Title = "Kompilacja \"debug\"";
#endif
```

to przy każdym uruchomieniu programu tytuł okna konsoli zostanie zmieniony tak, żeby informować o tym, że działająca aplikacja została skompilowana w trybie debogowania. Tak się będzie działało nie dlatego, że wartość `DEBUG` jest sprawdzana za każdym razem. Została ona sprawdzona już podczas komplikacji i jeżeli warunek dyrektywy `#if` został spełniony, powyższa instrukcja została „na stałe” umieszczona w kodzie pośrednim.

Definiowanie stałych preprocessora

Przy użyciu dyrektyw preprocessora możliwe jest przygotowanie jednego kodu w taki sposób, żeby łatwe było jego kompilowanie w różnych warunkach. Jak wspomniałem, typowym celem dyrektyw preprocessora jest komplikacja warunkowa, która umożliwia np. przygotowanie kodu do komplikacji przez różne kompilatory i na odmiennych platformach.

Dla przykładu można wyobrazić sobie klasę, która korzysta z przycisków z bibliotek kontrolek System.Windows.Forms lub System.Web.UI.WebControls (Windows Forms lub Web Forms) w zależności od obecności symbolu preprocesora o nazwie Web. Od jego obecności zależy to, która przestrzeń nazw zostanie zadeklarowana:

```
#define Web
#if Web
    using System.Web.UI.WebControls;
#else
    using System.Windows.Forms;
#endif
```

W pierwszej linii za pomocą dyrektywy `#define` zdefiniowany jest symbol preprocesora Web. Jego obecność powoduje, że zadeklarowana jest przestrzeń nazw `System.Web.UI.WebControls`. Usunięcie lub oznaczenie tej linii komentarzem spowoduje zmianę deklarowanej przestrzeni nazw na `System.Windows.Forms`. W obu tych przestrzeniach zdefiniowana jest klasa `Button` z podobnymi właściwościami i metodami. O tym, który z nich będzie używał program, decyduje zatem obecność symbolu Web.

W analogiczny sposób od obecności symbolu preprocesora można uzależnić wybór całych grup instrukcji języka C#. I to jest właśnie cel komplikacji warunkowej: aby zawartością jednej linii kontrolować przebieg procesu komplikacji. Bez tej możliwości często konieczne byłoby utrzymywanie dwóch równoległych wersji kodu lub ciągłe komentowanie i usuwanie komentarzy całych bloków instrukcji.

Definiowanie symboli preprocesora może mieć miejsce jedynie przed pierwszym polecienniem języka C#, co w praktyce oznacza, że dyrektywy `#define` powinny znaleźć się przed blokiem poleceń `using`. Ponadto stałe preprocesora są widoczne jedynie w obrębie pliku, w którym zostały zdefiniowane. Jeżeli potrzebujemy stałych globalnych, należy je dopisać do pola *Conditional compilation symbols* w opcjach projektu (ostatnia pozycja w menu *Project*) na zakładce *Build*.

Bloki

Do wykorzystania symboli preprocesora i warunkowej komplikacji dochodzi raczej w większych projektach, w których unika się utrzymywania wielu wersji kodu nawet przy tworzeniu kilku wersji programu. W mniejszych i prostszych projektach, które nie przekraczają kilkuset linii kodu źródłowego, najważniejszym zastosowaniem dyrektywy jest zazwyczaj budowanie bloków kodu ułatwiających uporządkowanie plików źródłowych. Służą do tego dyrektywy `#region` i `#endregion` (listing 3.20).

Listing 3.20. Przykład bloku zawierającego metody zdarzeniowe

```
#region Metody związane z obsługą przycisków
private void Button1_Click(object sender, EventArgs e)
{
    // pierwsza metoda
}

private void Button2_Click(object sender, EventArgs e)
{
    // druga metoda
```

```
}

private void Button3_Click(object sender, EventArgs e)
{
    // trzecia metoda
}
#endregion
```

Nie są to dyrektywy, które wpływają na działanie kompilatora — on je całkowicie ignoruje. Mają natomiast wpływ na edytor. Definiują blok kodu źródłowego, który może być zwinięty do jednej linii w podobny sposób, jak można zwinać każdą metodę czy klasę. Jeżeli edytor znajdzie parę dyrektyw `#region` i `#endregion`, to natychmiast w linii, w której znajduje się pierwsza z nich, pojawi się kwadracik z minusem. Kliknięcie go zwinie zaznaczony w ten sposób blok. Zwijanie nieedytowanych partii kodu bardzo poprawia jego czytelność, dlatego zachęcam do grupowania partii kodu związanych z jedną funkcjonalnością w oddzielnych blokach.

Atrybuty

Atrybuty są kolejnym sposobem wpływania na proces komplikacji. Stanowią źródło dodatkowych informacji dla kompilatora. Przykładowo atrybuty umieszczone w pliku *Properties\AssemblyInfo.cs* dostarczają kompilatorowi informacje o opisie i wersji pliku wykonywalnego lub biblioteki, którą umieszcza w skompilowanym kodzie pośrednim. Z kolei atrybuty znajdujące się bezpośrednio przed definicją metod pozwalają na poinformowanie kompilatora o własnościach lub ograniczeniach tej metody. Dobrym przykładem jest atrybut `STAThread` znajdujący się przed metodą `Main` aplikacji „okienkowych”. Z kolei atrybut `DllImport` pozwala deklarować metodę, której definicja znajduje się w niezarządzanej bibliotece DLL. Za pomocą atrybutów można także zaznaczyć metodę jako wychodzącą z użycia (atrybut `Obsolete`). Jeżeli umieścimy ten atrybut przed metodą:

```
[Obsolete("Metoda przestarzała. Lepiej użyj metody NowaMetoda", false)]
```

to w momencie gdy kompilator napotka jej wywołanie, pokaże komunikat informujący, że używamy metody zaznaczonej jako przestarzała. Jeżeli drugi argument jest równy `false` lub nie ma go w ogóle, komunikat będzie ostrzeżeniem. Gdy drugi argument to `true`, pojawi się błąd.

Atrybuty wykorzystywane są również do określania grup, do których trafiają właściwości i zdarzenia projektowanych w oknie właściwości komponentów oraz ich opisy widoczne na dole tego okna. Warto również zwrócić uwagę na atrybut `Conditional` wykorzystywany w komplikacji warunkowej i wiele innych.

Kolekcje

Tablice to istotny element niemal każdego języka programowania, ale oprócz tablic istnieją także inne struktury danych: listy, kolejki, drzewa i różne ich podtypy. Wszystkie one, łącznie z tablicami, zostały nazwane w C# kolekcjami. Co więcej, większość typowych kolekcji (a także kilka dodatkowych) została zaimplementowana w platformie .NET i jest gotowa do użycia. Tradycyjne kolekcje z platformy .NET 1.0 i 1.1, w których obiekty przechowywane są w referencjach typu `object`, dostępne są w przestrzeni nazw `System.Collections`. Nie należy ich używać, ponieważ w wersji 2.0 platformy .NET dodana została przestrzeń nazw `System.Collections.Generic` zawierająca parametryzowaną wersję kolekcji. Dostępna jest również przestrzeń nazw `System.Collections.Specialized` zawierająca ich wyspecjalizowane wersje. Warto zwrócić szczególną uwagę na często używaną listę zaimplementowaną w klasie `List<>` oraz na `SortedList<>` — słownik, w którym elementy są automatycznie sortowane.

„Zwykłe” tablice

Definicja tablicy w dowolnym języku programowania jest zawsze taka sama: tablica to struktura, która przechowuje elementy tego samego typu. Inną sprawą jest jej implementacja. W C# zrobiono to w taki sposób, aby niemożliwe było popełnienie często występującego w C/C++ błędu, w którym dochodzi do odczytu lub zapisu danych „za” zadeklarowanym obszarem tablicy. W C# indeksy są kontrolowane przez klasę implementującą tablicę. O ile w C/C++ tablica to po prostu fragment pamięci z adresem zapamiętanym we wskaźniku do pierwszego elementu, to w C# jest ona instancją klasy `System.Array`. Oznacza to, że tablica jest obiektem. Dzięki temu możliwe jest łatwe kontrolowanie tego, czy indeksy nie są mniejsze od zera lub czy nie są większe od rozmiaru tablicy (minus jeden).

Pomimo że implementacja jest całkiem inna, praktyczne zasady dotyczące tworzenia tablic w C# są niemal identyczne z zasadami rzadzącymi dynamicznie tworzonymi tablicami w C++¹⁵. Możemy zadeklarować tablicę, co oznacza utworzenie referencji do tablicy, oraz ją zdefiniować, czyli zarezerwować pamięć, a następnie zainicjować, czyli zapełnić wartościami.

Oto poprawna składnia deklaracji referencji do tablicy z elementami typu `int`:

```
int[] ti;
```

A oto przykład deklaracji referencji wraz z utworzeniem obiektu tablicy (rezerwowana jest pamięć na stercie):

```
int[] ti=new int[3];
```

Po utworzeniu obiektu tablicy (tj. w istocie instancji klasy `System.Array`) jest ona automatycznie inicjowana wartościami domyślnymi dla danego typu (tabela 3.2), czyli w przypadku typu `int` — zerami. Możemy także samodzielnie zainicjować elementy tablicy:

```
int[] ti=new int[3] {1,2,4};
```

¹⁵ Dynamicznie, czyli za pomocą operatora `new`. Deklarowanie tablic w stylu `int i[3];`, czyli tak jak tablic na stosie w C++, w C# nie jest możliwe.

W tablicy zawierającej typy referencyjne domyślne wartości elementów tablicy to null. Konieczna jest wobec tego jawna inicjacja elementów przez wywołanie operatora new dla każdego elementu tablicy i utworzenie odpowiedniej grupy obiektów. Nie należy mylić w tym przypadku wykorzystania operatora new do utworzenia tablicy z użyciem operatora new do utworzenia jej elementów. Rozważmy następujące polecenia:

(1)

```
Exception[] te  
Console.WriteLine("Komunikat: " + te[0].Message);
```

(2)

```
Exception[] te = new Exception[3];  
Console.WriteLine("Komunikat: " + te[0].Message);
```

(3)

```
Exception[] te = new Exception[3] { new Exception(), new Exception(), new  
Exception() };  
Console.WriteLine("Komunikat: " + te[0].Message);
```

Polecenia z grupy (1) w ogóle się nie skompilują, ponieważ w drugiej linii próbujemy odwołać się do zmiennej te (referencji do tablicy), która nie została zainicjowana. Polecenia (2) skompilują się i program można uruchomić, ale podczas działania zgłoszony zostanie wyjątek System.NullReferenceException. Dlaczego? Tablica te jest co prawda utworzona (obiekt tablicy powstał) i zainicjowana, ale domyślne wartości referencji, a to one są tu elementami przechowywanymi w tablicy, to null. Nie powstał żaden obiekt klasy Exception, więc elementy tablicy te nie mogą przechowywać żadnych adresów, a co za tym idzie próba odczytania właściwości Message musi skończyć się błędem. Dopiero w polecenях (3) tablica zostaje zapelniona referencjami zawierającymi adresy istniejących obiektów i instrukcja Console.WriteLine("Komunikat: " + te[0].Message); pokaże domyślny komunikat pierwszego wyjątku z tablicy.

Obiekty, do których odnoszą się referencje przechowywane w tablicy, nie muszą być tworzone w tej samej linii co deklaracja tablicy. Można do tego równie dobrze wykorzystać pętlę. Pokazuję to na listingu 3.21.

Listing 3.21. Przykład tworzenia grupy obiektów i gromadzenia ich referencji w tablicy

```
Exception[] te = new Exception[3];  
for (int i = 0; i < te.Length; i++)  
{  
    te[i] = new Exception("Komunikat " + i);  
    te[i].HelpLink = "http://www.fizyka.umk.pl/~jacek/";  
}
```

Elementy tablicy są indeksowane od 0. Ostatni element ma zatem indeks rozmiar-1. W każdej chwili rozmiar tablicy można odczytać z właściwości Length. Rozmiar tablicy nie można zmieniać — jeżeli potrzebujemy takiej możliwości, musimy wybrać inną kolekcję, np. opisaną niżej listę List<>. Jak pokazuje powyższy przykład (listing 3.21), dostęp do elementów tablicy jest realizowany za pomocą operatora [] z indeksem elementu wewnętrz, czyli w sposób identyczny jak w C/C++ czy Javie.

Nieco inaczej niż w C++ definiuje się natomiast tablice wielowymiarowe. Nie jest to tablica tablic, a nadal jeden obiekt klasy System.Array, którego elementy są odpowiednio poukładane. Oto przykład:

```
int[,] ti2 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };
```

Wielkość tej tablicy, odczytana za pomocą właściwości i2.Length, równa jest 6, ale dostęp do elementów możliwy jest jedynie przez umieszczenie dwóch indeksów wewnątrz nawiasów kwadratowych. Dla przykładu ostatni element w tablicy to i2[1,2].

Do inicjacji tablicy dwuwymiarowej można wykorzystać podwójną pętlę:

```
int[,] ti2 = new int[2, 3];
for (int i = 0; i < 2; i++)
    for (int j = 0; j < 3; j++)
        ti2[i, j] = 3 * i + j;
```

Pętla foreach

Po zdefiniowaniu i zainicjowaniu tablic (np. poleceniami z listingu 3.21) możemy wykonywać na ich elementach dowolne operacje. Często wykorzystywane są do tego pętle for o budowie identycznej z budową choćby tej na listingu 3.21. W niektórych przypadkach, gdy pętla for przebiega po wszystkich elementach tablicy, wygodniej skorzystać z nowego typu pętli, a mianowicie foreach (listing 3.22).

Listing 3.22. Przykład wykorzystania pętli foreach

```
foreach (Exception ei in te)
{
    ei.Source = AppDomain.CurrentDomain.FriendlyName;
    Console.WriteLine(ei.Message);
}
```

Pętla taka nie nadaje się jednak do inicjowania lub zmieniania wartości obiektów. Na listingu 3.22 referencja ei w tego typu pętli jest zadeklarowana *implicite* jako tylko do odczytu. A to oznacza, że niemożliwe jest również modyfikowanie elementów tablic o typach wartościowych. Następujące polecenie nie będzie mogło zatem być skompilowane:

```
foreach(int i in i2) i=1; //bląd
```

Nic nie stoi natomiast na przeszkodzie, aby w pętli foreach modyfikować obiekty typów referencyjnych. Przykładowo na listingu 3.22 obiekt, którego adres przechowujemy w referencji ei (w każdej iteracji inny), jest modyfikowany — jak widać, zmieniana jest jego etykieta. Tu nie ma jednak sprzeczności, ponieważ to nie obiekt przycisku jest tylko do odczytu, a referencja do niego (wartość zmiennej ei). Gdybyśmy natomiast próbowali wykonać pętlę widoczną na listingu 3.23, zobaczylibyśmy podczas komplikacji błąd z komunikatem informującym, że próbujemy zmienić wartość samej referencji, przypisując jej adres nowego obiektu, czego w pętli foreach robić nie wolno. Bez problemu możemy za to zmieniać stan obiektu, na który referencja bi wskazuje.

Listing 3.23. Poniższa pętla jest niepoprawna — zmienna bi nie może być modyfikowana

```
foreach (Exception ei in te)
{
    ei = new Exception("Nowy obiekt");
    ei.Source = AppDomain.CurrentDomain.FriendlyName;
    Console.WriteLine(ei.Message);
}
```

Pętla foreach działa także w odniesieniu do tablic wielowymiarowych, np. po wykonaniu polecień:

```
int[,] i2=new int[2,3] {{0,1,2},{3,4,5}};
foreach (int i in i2) Console.Write("") + i + ";" );
```

zobaczmy sześć cyfr o wartościach od 0 do 5.

Od C# 3.0 w pętli foreach do deklaracji elementu tablicy możemy użyć typu var. Kompilator wywnioskuje typ elementu na podstawie typu elementów kolekcji:

```
foreach (var i in i2) Console.Write("") + i + ";" );
```

Sortowanie

Typy liczbowe i znakowy implementują interfejs `IComparable` (z ang. *comparable to* porównywalny, ale nie w sensie „podobnej wielkości”, a raczej wspólnierny, oznaczający obiekty, w przypadku których jest sens, żeby je ze sobą porównywać). Interfejs ten zostanie dokładniej omówiony niżej. Tu tylko wspomnę, że wymaga on od implementującego go typu zdefiniowania metody `CompareTo` umożliwiającej porównywanie wartości bieżącego egzemplarza obiektu z obiektem wskazanym w argumentie. Jeżeli w tablicy znajdują się obiekty typu implementującego ten interfejs, mogą być sortowane. Bez problemu można zatem posortować np. tablicę liczb całkowitych, rzeczywistych lubłańcuchów. Służy do tego metoda statyczna `Array.Sort` (listing 3.24).

Listing 3.24. Sortowanie tabeli liczb pseudolosowych

```
int[] losy = new int[30];
Random r = new Random();
for (int indeks = 0; indeks < losy.Length; indeks++)
    losy[indeks] = r.Next(100);

string s = "Przed sortowaniem: ";
foreach (int los in losy) s += los.ToString() + " ";
Console.WriteLine(s);

Array.Sort(losy);

s = "Po sortowaniu: ";
foreach (int los in losy) s += los.ToString() + " ";
Console.WriteLine(s);
```

Próba napisania tego kodu dla zmiennych, które nie implementują tego interfejsu, skończy się zgłoszeniem błędu podczas działania programu System.InvalidOperationException. Przykład takiej skazanej na niepowodzenie próby widoczny jest na listingu 3.25.

Listing 3.25. Sortowanie np. obiektów typu *Exception* nie jest możliwe

```
// uruchomienie tego kodu skończy się błędem
Exception[] wyjątki = new Exception[100];
Random r = new Random();
for (int indeks = 0; indeks < wyjątki.Length; indeks++)
{
    wyjątki[indeks] = new Exception(r.Next(100).ToString());
}
Array.Sort(wyjątki);
```

Jeżeli chcielibyśmy sortować przyciski, np. według etykiet, musielibyśmy utworzyć klasę potomną do Button, która implementuje interfejs IComparable i w której zdefiniowana byłaby metoda CompareTo wskazująca, w jaki sposób porównywać takie obiekty. O tym w kolejnym rozdziale. Możliwe jest również posortowanie obiektów, które nie implementują interfejsu IComparable, jeżeli zdefiniujemy specjalną klasę porównującą obiekty tego typu implementującą interfejs IComparer. Dla obiektów klasy Exception może to być np. klasa:

```
class PorownywaczWyjatkow : IComparer<Exception>
{
    int IComparer<Exception>.Compare(Exception e1, Exception e2)
    {
        return e1.Message.CompareTo(e2.Message);
    }
}
```

Wówczas możemy tablicę wyjątków posortować instrukcją:

```
Array.Sort(wyjątki,new PorownywaczWyjatkow());
```



Na rzecz kolekcji, w tym tablic, można wywołać wiele metod, pozwalających m.in. przeprowadzić uśrednienie, sumowanie czy zliczanie elementów spełniających warunek określony wyrażeniem lambda. Nieco informacji na ten temat znajdzie się niżej, przy okazji omawiania technologii LINQ.

Kolekcja List

Przyjrzyjmy się teraz najczęściej wykorzystywanej kolekcji System.Collections.
↳Generics.List, czyli implementacji listy. Należy ona do grupy typów ogólnych (ang. *generic types*), nazywanych też typami parametrycznymi. Ma tę zaletę w porównaniu ze zwykłą tablicą (System.Array), że liczba elementów może być zmieniana już po utworzeniu listy — można dodawać elementy na koniec, wstawiać do środka i usuwać dowolnie wybrany element. Jednocześnie możliwy jest dostęp do dowolnego elementu, tak samo jak w przypadku tablicy. W poniższym przykładzie (listing 3.26) utworzymy kolekcję tego typu składającą się ze stu losowo wybranych liczb całkowitych. Następnie do jej końca dodamy dziesięć liczb o wartości 0, a do środka wstawimy kolej-

nych pięć o wartości -1. Wreszcie na pierwszej pozycji ustawimy liczbę 1. Na koniec usuniemy wszystkie te liczby, które są większe od 20, i posortujemy resztę.

Listing 3.26. Przykład wykorzystania list, który zawiera kilka charakterystycznych dla nich błędów

```

int rozmiar = 30;
Random r = new Random();
List<int> l = new List<int>(new int[rozmiar]);
for (int i = 0; i < rozmiar; i++) l[i] = r.Next(100);
l.AddRange(new int[10]);
int[] i5 = { -1, -1, -1, -1, -1 };
l.InsertRange(rozmiar / 2, i5);
l.Insert(0, 1);

// tu kryją się błędy
for (int i = 0; i < rozmiar; i++)
{
    if (l[i] > 20)
        l.RemoveAt(i);
}

l.Sort();

string s = "Elementy listy: ";
foreach (object ai in l) s += ai.ToString() + " ";
Console.WriteLine(s);

```

Z góry uprzedzam, że powyższy kod zawiera błędy, które trudno w pierwszej chwili zauważyc, a które są bardzo typowe dla kolekcji ze zmienną długością.

Błędy, i to aż trzy, kryją się w pętli usuwającej liczby większe niż 20. Pierwszy polega na tym, że po wcześniejszych manipulacjach lista nie ma już wielkości określonej przez zmienną rozmiar. W tej chwili jest ona równa rozmiar+10+5. Możemy więc albo kontrolować na bieżąco wartość zmiennej rozmiar, zmniejszając ją w iteracji, gdy usuwany jest element z listy, albo — co jest zdecydowanie lepszym rozwiązaniem — wymuścić sprawdzanie rzeczywistego rozmiaru listy, odczytując właściwość `List<>.Count`, w tym przypadku `l.Count` (listing 3.27).

Listing 3.27. Pierwsza poprawka

```

// lepiej, ale nadal z błędem
for(int i=0;i<l.Count;i++)
{
    if (l[i]>20)
        l.RemoveAt(i);
}

```

Drugi, poważniejszy błąd kryje się w samej pętli. Otóż usuwając w niektórych jej iteracjach elementy listy, zmniejszamy jej rozmiar. Jeżeli zatem na liście jest więcej niż piętnaście liczb większych od 20, a zapewne tak jest, rozmiar listy zmniejszy się i znajdzie się poniżej wartości określonej przez wielkość zapamiętaną w zmiennej rozmiar, a w efekcie kolejne iteracje będą musiały się skończyć zgłoszeniem wyjątku `System.ArgumentOutOfRangeException` (z ang. *argument poza zakresem*). Nie pomoże

używanie pętli `foreach`, bo ona również nie sprawdza, czy w wyniku wykonania iteracji zmienił się rozmiar kolekcji ustalony podczas inicjacji pętli. To jednak nie wszystko. W pętli kryje się również trzeci błąd logiczny. W razie spełnienia warunku `l[i]>20` usuwany jest element listy na pozycji `i`, co doprowadza do zmniejszenia listy o 1. Pętla przechodzi wówczas do następnej pozycji listy, czyli `i+1`. Ale po usunięciu elementu `i` obecny element `i+1` spada na pozycję `i`. W efekcie element ten nie jest w ogóle testowany. Należałoby więc wymusić ponowne sprawdzenie elementu o tym numerze, obniżając indeks pętli (listing 3.28).

Listing 3.28. Druga poprawka

```
for(int i=0;i<l.Count;i++)
{
    if (l[i]>20)
    {
        l.RemoveAt(i);
        i--;
    }
}
```

Jeżeli ktoś nie lubi ingerencji w indeksy pętli, może wykorzystać pętlę `do..while` wiadoczną na listingu 3.29. Innym rozwiązaniem jest sprawdzanie kolekcji w odwrotnym kierunku, tj. od ostatniego do pierwszego elementu. Wówczas usuwanie elementów nie wprowadza takiego zamieszania.

Listing 3.29. Inne podejście do drugiej poprawki

```
int j=0;
do
{
    if (l[j]>20)
        l.RemoveAt(j);
    else
        j++;
}
while(j<l.Count);
```

Kolekcja `SortedList` i inne słowniki

Poza `List` w przestrzeni nazw `System.Collections.Generics` dostępne są także inne typowe struktury danych. Pierwsza z nich to kolejka zaimplementowana w klasie `Queue`. Do kolejki można dodać element tylko na końcu, a zdjąć — tylko na początku. Inna popularna struktura danych to stos (klasa `Stack`), tj. zbiór, z którego można zdjąć jedynie element ostatnio dokołały. Spośród innych kolekcji na uwagę zasługuje `SortedList`, która w odróżnieniu od omówionych wcześniej jest „dwukolumnowa”. Każdy element listy przechowuje bowiem klucz i wartość (właściwości `Key` i `Value`). Pozwala to m.in. na sortowanie obu wartości według klucza, co często bywa przydatne. Podobnie zbudowane są kolekcje `Dictionary` i `SortedDictionary`. Wszystkie implementują wspólny interfejs `IDictionary` i dlatego często nazywane są słownikami.

Typowym zastosowaniem tej kolekcji są różnego typu listy z kluczem, np. słowniki, listy zmiennych itp. W poniższym przykładzie korzystam z kolekcji `SortedList` z oboma parametrami typu `String`, wykorzystując ją do przechowywania imion i nazwisk osób, które posługują się pseudonimami. Pseudonimy będą pełniły funkcję *kluczy* (ang. *key*), a prawdziwe personalia — *wartości* (ang. *value*). Wygodną właściwością kolekcji `SortedList` jest to, na co wskazuje jej nazwa, a więc to, że jest automatycznie sortowana zgodnie z kolejnością alfabetyczną kluczy. Listing 3.30 zawiera deklaracje kolekcji `artysci`, jejinicacji i prezentacji.

Listing 3.30. Przykład użycia listy parametrycznej `SortedList`

```
SortedList<string, string> artysci = new SortedList<string, string>();
artysci.Add("Sting", "Gordon Matthew Sumner");
artysci.Add("Bolesław Prus", "Aleksander Główacki");
artysci.Add("Pola Negri", "Barbara Apolonia Chałupiec");
artysci.Add("John Wayne", "Marion Michael Morrison");
artysci.Add("Chico", "Leonard Marx");
artysci.Add("Harpo", "Arthur Marx");
artysci.Add("Groucho", "Julius Marx");
artysci.Add("Bono", "Paul Hewson");
artysci.Add("Ronaldo", "Luiz Nazario de Lima");
artysci.Add("Madonna", "Madonna Louise Veronica Ciccone");
artysci.Add("Gabriela Zapolska", "Maria G. Śnieżko-Błocka");

string komunikat = "Zawartość listy:\n";
foreach (KeyValuePair<string, string> artysta in artysci)
    komunikat += artysta.Key + " - " + artysta.Value + "\n";
Console.WriteLine(komunikat);
```

Kolejka i stos

Przestrzeń `System.Collections.Generic` zawiera również implementacje dwóch standardowych struktur danych: kolejki (ang. *queue*) i stosu (ang. *stack*). Pierwsza jest strukturą typu FIFO (ang. *first in, first out*) — element włożony jako pierwszy może być pierwszy zdjęty. Kolejkę można sobie wyobrażać jako rurę, do której wkłada się piłki z jej jednego końca i wyjmuję z drugiego. Inaczej jest w przypadku stosu. Jest to struktura typu FILO (ang. *first in, last out*) — element włożony jako pierwszy dostępny będzie jako ostatni. Nazwa tej struktury dobrze oddaje też to, jak można sobie ją wyobrażać — jako stos np. kartek. To, co położymy na stosie jako pierwsze, będzie przykrywane przez następne elementy i tym samym dostępne dopiero, gdy zdejmujemy wszystkie położone później.

Na listingu 3.31 ilustruję tę podstawową różnicę między kolejką i stosem. W obu strukturach umieszczamy liczby od 0 do 9. Następnie kolejno je zdejmujemy. W przypadku kolejki otrzymamy liczby w tej samej kolejności, w jakiej je włożyliśmy. W przypadku stosu ich kolejność będzie odwrócona.

Listing 3.31. Użycie stosu i kolejki

```
int rozmiar = 10;
Queue<int> kolejka = new Queue<int>(rozmiar);
Stack<int> stos = new Stack<int>(rozmiar);
for (int i = 0; i < rozmiar; ++i)
```

```
{  
    kolejka.Enqueue(i);  
    stos.Push(i);  
}  
  
string s = "Elementy zdjęte z kolejki (" + kolejka.Count + " elementów): ";  
for (int i = 0; i < rozmiar; ++i) s += kolejka.Dequeue().ToString() + " ";  
Console.WriteLine(s);  
s = "Elementy zdjęte ze stosu (" + stos.Count + " elementów): ";  
for (int i = 0; i < rozmiar; ++i) s += stos.Pop().ToString() + " ";  
Console.WriteLine(s);
```

Tablice jako argumenty metod oraz metody z nieokreślona liczbą argumentów

C# dopuszcza definiowanie metod, w których liczba argumentów nie jest z góry określona. Możliwość ta nie jest może wykorzystywana zbyt często we własnym kodzie, ale czasem może być bardzo przydatna. Dobrym przykładem jest jej użycie w konstruktorach klas używanych w technologii LINQ to XML (rozdział 4.). Poniżej przedstawię prosty przykład takiej metody. Zacznijmy jednak od metody z listingu 3.32, której argumentem jest tablica.

Listing 3.32. Funkcja, której argumentem jest tablica liczb całkowitych

```
static private int Suma(int[] lista)  
{  
    Console.WriteLine("Liczba argumentów: " + lista.Length.ToString());  
    int suma = 0;  
    foreach (int liczba in lista) suma += liczba;  
    return suma;  
}
```

Do takiej metody przekazywana jest tablica liczb typu `int`, tzn. jej wywołanie powinno wyglądać następująco:

```
int suma = Suma(new int[] { 1, 2, 3 });  
Console.WriteLine("Suma: " + suma.ToString());
```

Gdy jednak zmienimy sygnaturę metody, dodając przed typem tablicowym słowo kluczowe `params` (listing 3.33), możemy ją wywoływać tak, że argumentami są elementy tablicy — dowolna ilość liczb typu `int`:

```
int suma = Suma(1, 2, 3);
```

Listing 3.33. Użycie modyfikatora `params`

```
static private int Suma(params int[] lista)  
{  
    Console.WriteLine("Liczba argumentów: " + lista.Length.ToString());  
    int suma = 0;  
    foreach (int liczba in lista) suma += liczba;  
    return suma;  
}
```

Kompilator sam utworzy z nich tablicę. Takie przekazanie elementów tablicy nie jest jednak obowiązkowe. Nadal możemy jako argumentu użyć tablicy, a nawet kolekcji będącej wynikiem zapytania LINQ, które na tablicę można skonwertować.

Co więcej, ponieważ argumentem metody jest tablica, a więc w istocie referencja do niej (tablice, jak już wiemy, są typem referencyjnym), możliwa jest zmiana elementów tychże i w efekcie zwrot wartości przez argumenty, tak jak w przypadku słów kluczowych `ref` i `out`.

Słowo kluczowe `yield`

Wyobraźmy sobie metodę, której zadaniem jest generowanie pewnego zbioru elementów. Przyjmijmy, że chodzi o zbiór liczb pseudolosowych — to jednak ma naprawdę drugorzędne znaczenie. Najbardziej naturalny przepis na taką metodę to: zadeklarować tablicę liczb całkowitych, utworzyć instancję klasy `Random`, w pętli zapisać tablicę liczbami losowymi i całość zwrócić przez wartość funkcji. Listing 3.34 zawiera metodę przygotowaną zgodnie z tym przepisem.

Listing 3.34. Tradycyjna metoda zwracająca kolekcję i metoda zdarzeniowa wyświetlająca tę kolekcję

```
static int[] metoda(int rozmiar)
{
    int[] wynik = new int[rozmiar];
    Random r = new Random();
    for (int i = 0; i < rozmiar; ++i)
    {
        wynik[i] = r.Next();
    }
    return wynik;
}

static void Main(string[] args)
{
    int[] ti = metoda(10);
    foreach (int element in ti) Console.WriteLine(element.ToString());
}
```

Korzystając z tego schematu, można łatwo przygotować funkcję zwracającą np. kolejne potęgi dwójki lub kolekcję przycisków. Zmieni się tylko typ kolekcji i zapełniająca ją pętla. W C# istnieje jednak stosunkowo mało znana alternatywa dla tego schematu — słowo kluczowe `yield` dodane do języka w wersji 2.0. Przyjrzyjmy się metodzie widocznej na listingu 3.35. Realizuje ona to samo zadanie co metoda z listingu 3.34.

Listing 3.35. Prosty przykład użycia słowa kluczowego `yield`

```
IEnumerable<int> metoda(int rozmiar)
{
    Random r = new Random();
    for (int i = 0; i < rozmiar; ++i)
    {
        yield return r.Next();
```

```
        }
        yield break;
    }

    static void Main(string[] args)
    {
        int[] ti = metoda(10).ToArray();
        foreach (int element in ti) Console.WriteLine(element.ToString());
    }
}
```

Zwróćmy uwagę, że zmienił się typ wartości zwracanej przez metodę. Zamiast tablicy jest nim teraz kolekcja policzalnych elementów typu `IEnumerable<int>`. Słowo kluczowe `yield` pojawia się w metodzie w dwóch kontekstach. W pierwszym występuje przed słowem kluczowym `return`. Zwróćmy uwagę, że za słowem `return` nie ma kolekcji, która przecież jest typem wartości zwracanej przez metodę, a jedynie są jej elementy. Co więcej, konstrukcja `yield return` znajduje się wewnątrz pętli. Ale właśnie na tym polega innowacja — instrukcja `yield return` nie kończy działania metody tak jak samo `return`, a jedynie dodaje występujący po niej element do niejawnie zdefiniowanej kolekcji. Kolekcja ta budowana jest dopóty, dopóki wątek nie natrafi na instrukcję `yield break`. Jest to sygnał do przerwania zbierania elementów. W naszym prostym przykładzie instrukcja ta wykonywana jest po zakończeniu pętli.

Kolekcja zwracana przez metodę nie musi mieć z góry ustalonego rozmiaru. Jest tworzona dynamicznie, wobec tego proces jej budowania można przerwać w dowolnym momencie. Na listingach 3.36 i 3.37 prezentuję wersję powyższych metod (korzystającą ze słowa kluczowego `yield` i obywającą się bez niego), w której tworzenie kolekcji zostanie zakończone, jeżeli wylosowana zostanie liczba większa niż podany w argumencie próg.

Listing 3.36. Wersja tradycyjna...

```
IEnumerable<int> metoda(int próg)
{
    List<int> wynik=new List<int>();
    Random r = new Random();
    int los;
    do
    {
        los = r.Next(15);
        wynik.Add(los);
    }
    while (los < próg);
    return wynik.AsEnumerable<int>();
}
```

Listing 3.37. ... i korzystająca ze słowa kluczowego `yield`

```
IEnumerable<int> metoda(int próg)
{
    Random r = new Random();
    int los;
    do
    {
```

```

        los = r.Next(15);
        yield return los;
    }
    while (los < próg);
    yield break;
}

```

Nowa forma inicjacji obiektów i tablic

Jedną ze zmian w C# 3.0, która skraca wyrażenia pobierające dane, jest nowa forma inicjacji tworzonych obiektów. Pozwala ona zainicjować obiekt bez konieczności stosowania dodatkowych instrukcji ustalających wartości poszczególnych pól i właściwości. Jeżeli zatem używając operatora new, utworzymy obiekt będący instancją struktury Button, to za nazwą typu można będzie w nawiasach klamrowych umieścić listę inicjacji jego dostępnych „z zewnątrz” pól i właściwości zgodnie ze schematem:

```
Button b = new Button { Text = "Przycisk", BackColor=System.Drawing.Color.Green };
```

Zastąpi to konstrukcję:

```

Button u = new Button();
b.Text = "Przycisk";
b.BackColor=System.Drawing.Color.Green;

```

W uproszczeniu można więc powiedzieć, że nowa forma inicjacji zastępuje definowanie konstruktorów inicjujących stan obiektów. Konieczne są jednak udostępnione pola lub właściwości¹⁶.

Nowy sposób inicjacji dotyczy również tablic. Jedynym warunkiem jest, aby tablice były jednorodne, tzn. ich elementy muszą być tego samego typu:

```

var it = new[] {5,4,3,2,1,0};
var lt = new[] {5L,4L,3L,2L,1L,0L};
var st = new[] {"Helion", "Onepress", "Sensus", "Septem", "Editio"};
var ft = new[] {1.0f, 0.75f, 0.5f, 0.25f, 0.0f};
var dt = new[] {1.0, 0.75, 0.5, 0.25, 0.0};

```

Podobna zmiana dotyczy inicjowania kolekcji — analogicznie do powyższego przykładowa można inicjować wartości przechowywanych przez nią elementów, np.:

```
List<string> lista = new List<string> { "Helion", "Onepress", "Sensus", "Septem",
                                         "Editio" };
```

¹⁶ Pola i właściwości muszą być zadeklarowane z modyfikatorami public, internal lub protected internal (tabela 3.1).

Caller Information

W C# 5.0 pojawiła się również możliwość uzyskania w metodzie informacji o miejscu, z którego nastąpiło jej wywołanie. Możemy uzyskać informacje o nazwie metody wywołującej, ścieżce do pliku, w którym owa metoda jest zdefiniowana, i numerze linii w pliku, w której znajduje się instrukcja wywołania. Ilustruje to listing 3.38. Informacje uzyskane w ten sposób mogą być szczególnie przydatne przy śledzeniu błędów aplikacji internetowych. Najwygodniej rejestrować je wówczas za pomocą klasy `System.Diagnostics.Trace`.

Listing 3.38. Metoda prezentująca informacje o miejscu, z którego została wywołana

```
static public void PokazInformacje(
    string zwyklyArgument,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    Console.WriteLine("Informacje o miejscu wywołania metody:");
    Console.WriteLine("nazwa elementu składowego: " + memberName);
    Console.WriteLine("ścieżka pliku z kodem źródłowym: " + sourceFilePath);
    Console.WriteLine("numer linii, w której nastąpiło wywołanie: " +
        sourceLineNumber);
}
```

Atrybuty, którymi w metodzie `PokazInformacje` udekorowano trzy specjalne argumenty, zostały zdefiniowane w przestrzeni nazw `System.Runtime.CompilerServices`. Trzeba ją zatem uwzględnić w sekcji poleceń `using` na początku pliku.

Rozdział 4.

Programowanie obiektowe w C#

Programista C# może realizować swoje zadania, korzystając z gotowych komponentów dostarczonych wraz z platformą .NET. Ponadto cały swój kod może umieszczać w tworzonych automatycznie metodach zdarzeniowych. Jednym słowem, może dowolnie długo unikać samodzielnego definiowania własnych klas. Jakość takiego kodu nie będzie jednak najwyższa. Dążenie do unikania błędów, łatwiejszej konserwacji kodu i możliwości jego modyfikacji przez innych programistów wymaga przejścia do innego paradymatu — projektowania zorientowanego obiektowo. Mówiąc wprost, dobry programista nie obejdzie się bez tworzenia własnych klas. Dzięki nim możliwe jest „zamknięcie” implementowanych funkcjonalności w eleganckim pudełku — klasie. Klasa ukrywa szczegóły implementacji, udostępniając jedynie interfejs pozwalający na kontrolę jej działania¹. Takie wyraźne „zamknięcie” kodu bardzo pomaga również w jego testowaniu, wyznaczając jasno jednostki podlegające sprawdzaniu.

Zacznijmy jednak od elementarza. Celem **klasy** jest zaimplementowanie jakiegoś nowego typu obiektów. Egzemplarze tego typu, a więc **obiekty**, nazywane są **instancjami klasy**. W klasie można zdefiniować **pola** przechowujące dane, **metody**, które mogą manipulować danymi, oraz **właściwości**, które są wygodnym mariażem jednych i drugich. Poza polami, metodami i właściwościami można w jej obrębie zdefiniować także indeksatory, delegacje i zdarzenia.

¹ Słowo „interfejs” używane jest w dwóch różnych, choć związanych ze sobą znaczeniach. Po pierwsze: używa się go do określenia części publicznej klasy, pozwalającej na interakcję z innymi obiektami. W tym sensie słowo to ma podobne znaczenie jak w zwrocie „interfejs aplikacji” (np. zbudowany z kontrolek), który służy do interakcji z użytkownikiem. W platformie .NET interfejs ma jednak także bardziej konkretne znaczenie. Korzystając ze słowa kluczowego `interface`, można zdefiniować interfejs, który deklaruje zbiór metod i właściwości bez ich definiowania. Muszą być one zdefiniowane natomiast w klasach, które taki interfejs chcą zaimplementować. Interfejs w tym znaczeniu określa zatem zakres czynności dostępnych w implementujących go klasach. Wróćmy do tego tematu w tym rozdziale, omawiając implementację interfejsu w definiowanym typie, a potem omawiając polimorfizm.

W C# mamy do dyspozycji dwa „metatypy”: referencyjny i wartościowy (rozdział 2.), zatem pierwszą decyzją, jaką musimy podjąć w procesie projektowania nowego typu, jest wybór między strukturą a klasą. W przypadku nowych typów liczb wybór jest oczywisty — zalety typu wartościowego przemawiają za wyborem struktury. Podobnie oczywista jest sytuacja, gdy tworzymy nową kontrolkę — wówczas musimy definiować klasę. W pozostałych przypadkach, aby ocenić, którego „metotypu” powinniśmy użyć, musimy zastanowić się, jak wiele pamięci będzie on zajmował i jak często będziemy go kopować lub przekazywać do metod. W przypadku dużej klasy możemy do metody przekazać referencję, co uchroni przed tworzeniem kopii i niepotrzebnym zajmowaniem pamięci. Z kolei używanie struktur i związanego z nimi prostego modelu zarządzania pamięcią zmniejsza obciążenie odśmiecaca, co wpływa na wydajność całej platformy .NET. Należy także wziąć pod uwagę to, że tworzenie i usuwanie obiektów na stosie jest szybsze niż na stercie. A z tego wynika, że struktur należy użyć, jeżeli obiekty projektowanego typu będą niewielkie i wykorzystywane raczej w małych zakresach, gdzie mogą być często tworzone i zaraz usuwane. Natomiast klasy powinny być używane raczej do implementacji typów związanych z wykorzystaniem większych obszarów pamięci (np. przez tablice) czy takich, których inicjacja wiąże się z czasochronnymi czynnościami lub użyciem zewnętrznych zasobów, np. plików lub baz danych. Wówczas przekazywanie adresu obiektu zamiast jego klonowania jest lepszym rozwiązaniem.

Przykład struktury (Ulamek)

Jako przykład definiowania typów przedstawię strukturę `Ulamek`, która będzie implementacją liczb wymiernych. Jak wspomniałem, wybór struktury w przypadku implementacji liczb jest na szczęście naturalny i nie musimy się zastanawiać, czy nie lepiej byłoby użyć w tym celu klasy.

Przygotowywanie projektu

Zaczynamy od utworzenia projektu aplikacji i w jego obrębie pliku dla struktury, którą będziemy projektować.

1. W środowisku Visual Studio naciśnij kombinację klawiszy `Ctrl+Shift+N`.
2. W oknie *New Project* wybierz typ projektu *Console Application*, ustal jego nazwę na *UlamekDemo* i kliknij *OK*.
3. Do projektu dodaj plik zawierający klasę. W tym celu z menu *Project* wybierz *Add Class....* Pojawi się okno *Add New Item* z zaznaczoną pozycją *Class*. W polu *Name* podaj nazwę pliku *Ulamek.cs* i kliknij *Add*.

Dodany plik pojawi się w podoknie *Solution Explorer*, a umieszczony w nim bardzo prosty kod klasy `Ulamek` (listing 4.1) zobaczymy w edytorze kodu.

Listing 4.1. Utworzona przez Visual C# klasa pozbawiona jest nawet konstruktora

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

namespace UlamekDemo
{
    public class Ulamek
    {
    }
}
```

My jednak planowaliśmy zdefiniować strukturę, a nie klasę. Wobec tego pierwszą czynnością będzie zmiana słowa kluczowego `class` na `struct`. W tym celu modyfikujemy kod zgodnie ze wzorem z listingu 4.2. Następnie uzupełniamy powyższy szkielet definicją dwóch pól odpowiadających licznikowi i mianownikowi liczby wymiernej.

Listing 4.2. Transformacja klasy w strukturę wymaga zmiany jednego słowa kluczowego

```
public struct Ulamek
{
    private int licznik, mianownik;
}
```

Pola `licznik` i `mianownik` są prywatne (modyfikator `private`). Oznacza to, że tylko metody zdefiniowane w strukturze `Ulamek`, tj. w tej samej strukturze, w której zdefiniowane są te pola, będą mogły odczytać lub zmienić ich wartość. Natomiast będą niedostępne spoza struktury `Ulamek`. Elementy składowe, które będziemy chcieli udostępniać, powinny być publiczne (modyfikator `public`). Publiczny powinien być w szczególności konstruktor. Wszystkie możliwe zakresy dostępu pól, właściwości i metod w języku C# prezentuje tabela 4.1.

Tabela 4.1. Modyfikatory dostępu do składowych klas i struktur w C#²

Nazwa	Modyfikator	Opis
publiczny	<code>public</code>	Element dostępny bez ograniczeń
chroniony	<code>protected</code>	Element dostępny tylko z klasy, w której został zdefiniowany, lub z jej klas potomnych
wewnętrzny	<code>internal</code>	Element dostępny, ale tylko z klas zespołu (ang. <i>assembly</i>), np. biblioteki DLL, wewnątrz którego zdefiniowana jest jego klasa macierzysta
wewnętrzny chroniony	<code>protected internal</code>	Element dostępny jak w przypadku wewnętrznego, ale również z klas dziedziczących z klasy, w której został zdefiniowany
prywatny	<code>private</code>	Element dostępny tylko z klasy, w której został zdefiniowany

² Źródło: [http://msdn.microsoft.com/en-us/library/ba0alyw2\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/ba0alyw2(v=vs.80).aspx).

Konstruktor i statyczne obiekty składowe

W strukturach nie można umieścić polecenia inicjującego pole w miejscu jego deklaracji. Inicjowane jest ono automatycznie wartością domyślną (zerem lub wartością null). Do samodzielnego określenia wartości pola można wykorzystać konstruktor, ale w przypadku struktur i tu napotkamy ograniczenie — programista nie może bowiem zdefiniować samodzielnie konstruktora domyślnego (bezargumentowego). Jest on tworzony automatycznie przez kompilator. Można natomiast zdefiniować konstruktor zawierający argumenty. Aby to zrobić, do definicji struktury dodajemy dwuargumentowy konstruktor i korzystające z niego statyczne obiekty składowe. W tym celu uzupełniamy strukturę zgodnie z listingiem 4.3.

Listing 4.3. Konstruktor i obiekty stałe

```
struct Ulamek
{
    private int licznik, mianownik;

    public Ulamek(int licznik, int mianownik = 1)
    {
        if (mianownik == 0)
            throw new ArgumentException("Mianownik musi być różny od zera");
        this.licznik = licznik;
        this.mianownik = mianownik;
    }

    public static readonly Ulamek Zero = new Ulamek(0);
    public static readonly Ulamek Jeden = new Ulamek(1);
    public static readonly Ulamek Polowa = new Ulamek(1, 2);
    public static readonly Ulamek Cwierc = new Ulamek(1, 4);
}
```

 **Wskazówka**

Słowo kluczowe `this` użyte w konstruktorze z listingu 4.3 to referencja do bieżącego obiektu. W strukturze `Ulamek` jest to referencja do bieżącego obiektu tego typu. Nie może być używany w metodach statycznych, które wywoływane są na rzecz typu (klasy lub struktury) i nie mają dostępu do żadnej jego instancji. Istnieje również słowo kluczowe `base`, które umożliwia odwoływanie się do metod klasy bazowej z klasy potomnej (o tym niżej).

Użytkownik naszej struktury może korzystać z konstruktora, aby nadać polom odpowiednie wartości. Nasz konstruktor dba przy tym, aby nie powstał obiekt `Ulamek`, którego mianownik miałby wartość równą zero. Należy jednak pamiętać, że — pomimo naszych wysiłków — taki obiekt powstanie, jeżeli użyjemy konstruktora domyślnego! Zwrócić uwagę, że drugi argument konstruktora ma domyślną wartość równą 1. Dzięki temu obiekt możemy tworzyć, podając jedynie wartość licznika. Wówczas ułamek przyjmuje wartość równą liczbie całkowitej.

Obecność konstruktora pozwala na zdefiniowanie wewnętrz struktury statycznych pól tylko do odczytu (modyfikatory `static` i `readonly`), które będą ułatwiać jej inicjację³.

³ W identyczny sposób zdefiniowane są np. stałe identyfikujące kolory w strukturze `Color`.

Ponieważ są one statyczne, nawet w strukturze można je zainicjować w miejscu deklaracji.

Oprócz statycznych pól struktury można również definiować statyczne metody. Są one wywoływanego na rzecz struktury, a nie jej instancji i wobec tego nie mogą w żaden sposób zależeć od stanu obiektu, tj. od wartości pól niestatycznych (w naszym przypadku od pól licznik i mianownik). Możemy np. zdefiniować metodę statyczną wyświetlającą informacje o strukturze (listing 4.4).

Listing 4.4. Statyczna metoda struktury *Ulamek*

```
public static string Info()
{
    return "Struktura Ulamek, (c) Jacek Matulewski 2014";
}
```

Modyfikatory **const** i **readonly**

Deklarując zmienne oraz pola i metody klas, można użyć szeregu modyfikatorów, z których dwa już znamy — modyfikatory **static** i **readonly**. Pierwszy powoduje, że pole lub metoda klasy nie wymaga tworzenia obiektu, aby móc się do niej odwołać. W przypadku pól oznacza to również, że w programie pole to będzie miało tylko jedną wartość.

W przypadku pól można użyć także modyfikatorów **const** i **readonly**. Pierwszy z nich powoduje, że pole ma wartość ustaloną już w trakcie komplikacji i wartość ta nie może być zmieniona. Dodatkowym skutkiem jest to, że pole opatrzone modyfikatorem **const** zachowuje się tak jak pole statyczne, tzn. odczytujemy je, korzystając z klasy, a nie instancji obiektu. Stałe mogą być tylko zmienne liczbowe, łańcuch oraz zmienna logiczna.

W odróżnieniu od **const** modyfikator **readonly** powoduje tylko, że zmienna po inicjacji nie może być już modyfikowana. Możliwość inicjacji ograniczona jest do miejsca deklaracji pola lub konstruktora. Można sobie wobec tego wyobrazić sytuację, w której wartość takiej zmiennej zależy od użytego konstruktora. W przypadku zmiennej typu **const** taka sytuacja nie może się zdarzyć.

Możliwe jest łączenie modyfikatorów, np. **static readonly**. Oznacza to, że pola są statyczne, odczytywane na rzecz klasy, a jednocześnie tylko do odczytu — nie można zmienić ich wartości.

Pierwsze testy⁴

Przetestujmy działanie struktury, tworząc jej instancję w metodzie **Program.Main**.

1. Przejdz na zakładkę *Program.cs*.
2. W metodzie **Main** umieść polecenia z listingu 4.5.

⁴ W rozdziale 6. tę samą strukturę będziemy testować, korzystając z testów jednostkowych.

Listing 4.5. Pierwsze testy naszej implementacji ułamka

```
static void Main(string[] args)
{
    Ulamek u2 = new Ulamek(2, 1);
    Ulamek u0 = Ulamek.Zero;
    Ulamek uP = Ulamek.Polowa;
    Console.WriteLine(uP.ToString());

    Console.WriteLine(Ulamek.Info());
}
```

Projekt kompliuje się, co oznacza, że struktura `Ulamek` nie zawiera błędów składniowych. Co więcej, podczas uruchamiania także nie pojawiają się żadne wyjątki. Niestety pierwszy wyświetlany komunikat zamiast wartości ułamka pokazuje jedynie nazwę typu, a więc `UlamekDemo.Ulamek`. To oznacza jednak tylko tyle, że wywołana na rzecz obiektu ułamka metoda `ToString` wymaga nadpisania. Przy tej okazji przygotujemy także dodatkową metodę służącą do konwersji ułamka na liczbę rzeczywistą.

Konwersje na łańcuch (metoda `ToString`) i na typ `double`

Zdefiniujmy metody `ToString` i `ToDouble` służące do konwersji struktury `Ulamek` na łańcuch (`string`) oraz liczbę typu `double`. W tym celu definicję struktury `Ulamek` (z pliku `Ulamek.cs`) uzupełniamy o definicje dwóch metod publicznych widocznych na listingu 4.6.

Listing 4.6. Nadpisana metoda `ToString` oraz nowa metoda `ToDouble`

```
public override string ToString()
{
    return licznik.ToString() + "/" + mianownik.ToString();
}

public double ToDouble()
{
    return licznik / (double)mianownik;
}
```

Teraz po uruchomieniu aplikacji zobaczymy wartość przechowywaną przez obiekt ułamka.

Zauważmy, że przy definiowaniu metody `ToString`, a dokładnie po wpisaniu słowa kluczowego `override` i spacji, automatycznie pojawiła się lista szablonów. Jeśli wybraliśmy z niej `ToString`, edytor uzupełnił kod o typ `string` zwracany przez tę metodę i wewnętrz niej umieścił wywołanie metody `ToString` z klasy bazowej. Tę linię należy zastąpić linią widoczną na listingu 4.6.

W przypadku konwersji na typ rzeczywisty moglibyśmy pójść dalej, implementując interfejs `IConvertible` (o tym, co znaczy „implementacja interfejsu” — niżej), który zmusiłby nas do zdefiniowania metod konwersji na wszystkie typy liczbowe platformy .NET (zobacz zadania z części IV).

Metoda upraszczająca ułamek

Kolejną metodą, w jaką warto wyposażyć strukturę `Ułamek`, jest metoda upraszczająca licznik i mianownik poprzez znalezienie ich największego wspólnego dzielnika. Do definicji struktury dodajemy wobec tego publiczną metodę widoczną na listingu 4.7, która implementuje „naiwny” algorytm szukania największego wspólnego dzielnika obu liczb. Zaczynając od mniejszej z nich, „schodzi” w dół, testując kolejne wartości aż do liczby 2. Jeżeli znajdzie taką, przez którą można podzielić licznik i mianownik bez reszty, robi to. A ponieważ szuka od „góry”, pierwszy dzielnik, jaki znajduje, musi być największym wspólnym dzielnikiem i wobec tego poszukiwania mogą być zakończone (instrukcja `break`). Następnie ustalane są znaki obu pól. Oba powinny być dodatnie lub ujemny powinien być tylko licznik. Wykorzystujemy przy tym prostą sztuczkę — dwie liczby mają różny znak, jeżeli ich iloczyn jest mniejszy od zera⁵.

Listing 4.7. Definicja metody upraszczającej licznik i mianownik

```
public void Uprosc()
{
    //NWD
    int mniejsza=Math.Min(Math.Abs(licznik),Math.Abs(mianownik));
    for (int i = mniejsza; i > 1; i--)
        if ((licznik%i==0) && (mianownik%i==0))
    {
        licznik/=i;
        mianownik/=i;
        break;
    }

    //znaki
    if (licznik*mianownik<0)
    {
        licznik=-Math.Abs(licznik);
        mianownik=Math.Abs(mianownik);
    }
    else
    {
        licznik=Math.Abs(licznik);
        mianownik=Math.Abs(mianownik);
    }
}
```

Sprawdźmy jej działanie, umieszczając w metodzie `Main` następujące instrukcje:

```
Ułamek u = new Ułamek(4,-2);
u.Uprosc();
Console.WriteLine(u.ToString());
```

To jest jednak bardzo naiwna implementacja, która przy dużych wartościach licznika i mianownika jest niezwykle kosztowna. Już od ponad 2300 lat znany jest znacznie lepszy algorytm, nazywany imieniem jego twórcy Euklidesa⁶. Jego implementacja

⁵ Ten prosty warunek okaże się problematyczny. Wróćmy do niego w rozdziale 6.

⁶ Zobacz http://pl.wikipedia.org/wiki/Algorytm_Euklidesa
lub <http://mathworld.wolfram.com/EuclideanAlgorithm.html>.

(listing 4.8) z użyciem dzielenia bez reszty jest bardzo wydajna — ilość iteracji nie jest większa niż ilość cyfr liczb, których wspólnego dzielnika poszukujemy. O skali przyspieszenia niech świadczy to, że na moim komputerze biurkowym próba uproszczenia stu losowo zainicjowanych ułamków pierwszą wersją metody trwa aż 430 000 milisekund (ponad 7 minut), podczas gdy nowa metoda uporała się z milionem takich ułamków w nieco ponad 500 milisekund (pół sekundy).

Listing 4.8. Warto dbać o optymalizację algorytmów

```
public void Uprosc()
{
    int a = licznik;
    int b = mianownik;

    // NWD
    int c;
    while (b != 0)
    {
        c = a % b;
        a = b;
        b = c;
    }

    licznik /= a;
    mianownik /= a;

    // znaki
    if (licznik * mianownik < 0)
    {
        licznik = -Math.Abs(licznik);
        mianownik = Math.Abs(mianownik);
    }
    else
    {
        licznik = Math.Abs(licznik);
        mianownik = Math.Abs(mianownik);
    }
}
```

Właściwości

Warto zauważyc, że dotychczasowa struktura `Ułamek`, nie licząc konstruktora, nie posiada żadnych metod pozwalających na modyfikacje pól `licznik` i `mianownik`. Móglby to być stan zamierzony, ponieważ w strukturach często ogranicza się możliwość inicjacji jedynie do momentu, w którym obiekt powstaje. My jednak zdefiniujemy właściwości `Licznik` i `Mianownik`, które pozwolą na manipulację tymi składnikami. Ich definicję pokazuję na listingu 4.9. Kod znajdujący się w sekcji `get` wykonywany jest w trakcie odczytu, a w sekcji `set` — w czasie przypisania wartości do właściwości. W tym względzie właściwości przypominają metody dostępowe (tzw. akcesory). A przecież sposób ich użycia jest identyczny ze sposobem korzystania z pól. Właściwości łączą więc wygodę pól z elastycznością akcesorów — warto się nimi posługiwać.

Listing 4.9. Właściwości zajmują zwykle sporo linii, więc warto umieścić je w bloku

```
#region Wlasnosci
public int Licznik
{
    get //odczyt
    {
        return licznik;
    }
    set //zapis
    {
        licznik = value;
    }
}

public int Mianownik
{
    get //odczyt
    {
        return mianownik;
    }
    set //zapis
    {
        if (value == 0)
            throw new ArgumentException("Mianownik musi być różny od zera");
        mianownik = value;
    }
}
#endregion
```

Jak widać, właściwość `Licznik` w obecnej postaci nie przynosi wiele pożytku. Równie dobrze można by zmienić zakres pola `licznik` na publiczny albo zdefiniować tzw. właściwości *auto-implemented* (o tym już za chwile). Inaczej jest w przypadku właściwości `Mianownik`. Wykorzystaliśmy w niej to, że podczas przypisania wartości do właściwości można wykonać dowolny kod. Umożliwia to sprawdzenie, czy przypisywana wartość jest różna od zera. Jeżeli jest — możemy zareagować, zgłaszając wyjątek.

Oczywiście właściwości nie są nowym pomysłem. W różnych postaciach obecne są w większości języków, które związane były z narzędziami projektowania wizualnego, a więc współpracującymi ze środowiskami wyposażonymi w okno właściwości lub jego odpowiednik. To właśnie publiczne właściwości komponentów możemy modyfikować za pomocą podokna *Properties* w Visual Studio.

Aby przetestować działanie właściwości, wykorzystajmy je do inicjacji obiektu (podrozdział „Nowa forma inicjacji obiektów i tablic” z poprzedniego rozdziału):

```
Ulamek u = new Ulamek { Licznik = 1, Mianownik = 2 };
Console.WriteLine(u.ToString());
```

Jest to równoważne konstrukcji:

```
Ulamek u = new Ulamek();
u.Licznik = 1;
u.Mianownik = 2;
```

lub użyciu konstruktora:

```
Ulamek k = new Ulamek(1, 2);
```

Jak wspomniałem w poprzednim rozdziale, ta nowa forma inicjacji w pewnym sensie zwalnia z obowiązku definiowania konstruktorów inicjujących stan tworzonych obiektów. Zamiast konstruktorów wystarczy zdefiniować publiczne pola lub właściwości.

Domyślnie implementowane właściwości (ang. auto-implemented properties)

Właściwość `Mianownik`, umożliwiając dostęp do pola `mianownik`, sprawdza dodatkowo, czy przypisywane do niego wartości są różne od zera. W przypadku właściwości `Licznik` jej rola ogranicza się jedynie do odczytania i przekazania wartości pola `licznik` w sekcji `get` i jej zmiany w sekcji `set`. W takim przypadku można użyć konstrukcji *domyślnie implementowanych właściwości* (ang. *auto-implemented properties*), która pojawiła się w C# 3.0. Korzystając z niej, definiujemy jedynie właściwość, natomiast pole jest niejako dodawane do klasy przez kompilator. Zatem jeżeli w strukturze `Ulamek` zmienimy właściwość `Licznik` w następujący sposób:

```
public int Licznik { get; set; }
```

i jednocześnie usuniemy z niej pole `licznik`, funkcjonalność tej właściwości w żaden sposób się nie zmieni. Taka zmiana (usunięcie pola) pociąga jednak za sobą kilka kolejnych modyfikacji struktury `Ulamek`. Przede wszystkim zmiany wymaga konstruktor, w którym odwołania do pola `licznik` należy zastąpić użyciem właściwości `Licznik`:

```
public Ulamek(int licznik, int mianownik = 1)
    : this()
{
    Licznik = licznik;
    Mianownik = mianownik;
}
```

Jak widać, przed przypisaniem właściwości w ciele konstruktora wywoływany jest domyślny konstruktor, który w strukturach jest definiowany przez kompilator bez względu na obecność konstruktora zdefiniowanego przez programistę. Bez tego wywołania w przypadku struktury pojawiłby się błąd komplikacji informujący, że automatycznie tworzone pole dla domyślnie zaimplementowanej właściwości nie jest zainicjowane. W przypadku klas takiego problemu nie ma. Natomiast w ciele konstruktora do inicjacji obiektu używamy właściwości — dzięki temu nie ma już potrzeby sprawdzania, czy `mianownik` jest różny od zera. Zrobi to kod sekcji `set` właściwości `Mianownik`. Unikamy w ten sposób powtarzania kodu.

Poza tym w całej klasie `Ulamek` należy zastąpić wszystkie odwołania do pola `licznik` przez odwołania do właściwości `Licznik` (dotyczy to metod `ToString`, `ToDouble` i `Uprosc`); przypomni nam o tym zresztą kompilator. Można do tego użyć narzędzia do automatycznego zastępowania (`Ctrl+H`), ale należy zachować ostrożność.

Operatory arytmetyczne

Język C# pozwala na przeciążanie operatorów dla definiowanych typów. Szczególnie wygodne jest to w przypadkach takich jak nasz, tzn. gdy implementujemy typ, w którym mają sens operacje arytmetyczne i dla którego operatory zachowują swoje naturalne znaczenie. Unikamy wtedy największej groźby związanej z przeciążaniem operatorów — nadawania im znaczenia niezgodnego z intuicją, co łatwo może prowadzić do ich niewłaściwego używania. Użycie operatorów skraca kod i sprawia, że jest on znacznie łatwiejszy do czytania, a dzięki temu bardziej odporny na błędy. Należy jednak zachować zdrowy rozsądek i nie definiować operatorów tylko dlatego, że jest taka możliwość.

W strukturze `Ulamek` zdefiniujemy operatory wykonujące operacje arytmetyczne na ułamkach. Posłużymy się do tego wzorem widocznym na listingu 4.10.

Listing 4.10. Definicje operatorów `+, -, * i /`

```
#region Operatory
//operatorы arytmetyczne
public static Ulamek operator -(Ulamek u)
{
    return new Ulamek(-u.Licznik,u.Mianownik);
}

public static Ulamek operator +(Ulamek u1, Ulamek u2)
{
    Ulamek wynik=new Ulamek(u1.Licznik*u2.Mianownik+u2.Licznik*u1.
    ↪Mianownik,u1.Mianownik*u2.Mianownik);
    wynik.Uprosc();
    return wynik;
}

public static Ulamek operator -(Ulamek u1, Ulamek u2)
{
    Ulamek wynik=new Ulamek(u1.Licznik*u2.Mianownik-u2.Licznik*u1.Mianownik,
    ↪u1.Mianownik*u2.Mianownik);
    wynik.Uprosc();
    return wynik;
}

public static Ulamek operator *(Ulamek u1, Ulamek u2)
{
    Ulamek wynik=new Ulamek(u1.Licznik*u2.Licznik,u1.Mianownik*u2.Mianownik);
    wynik.Uprosc();
    return wynik;
}

public static Ulamek operator /(Ulamek u1, Ulamek u2)
{
    Ulamek wynik=new Ulamek(u1.Licznik*u2.Mianownik,u1.Mianownik*u2.Licznik);
    wynik.Uprosc();
    return wynik;
}
#endregion
```

Operatory zostały zdefiniowane jako statyczne elementy składowe struktury. Wobec tego do wykonywania operacji nie jest angażowany bieżący stan obiektu; obiekty, na podstawie których operator oblicza nową wartość, przekazywane są przez argumenty operatora. Nie musimy, a nawet nie możemy, definiować operatorów `+=`, `-=`, `*=` i `/=`. Kompilator buduje je sam na podstawie operatorów arytmetycznych i operatora przy-pisania.

Aby przetestować zdefiniowane przed chwilą operatory, przechodzimy na zakładkę *Program.cs* i w metodzie *Main* umieszczaemy polecenia widoczne na listingu 4.11.

Listing 4.11. Testowanie operatorów arytmetycznych

```
static void Main(string[] args)
{
    Ulamek a = Ulamek.Polowa;
    Ulamek b = Ulamek.Cwierc;

    Console.WriteLine((a + b).ToString());
    Console.WriteLine((a - b).ToString());
    Console.WriteLine((a * b).ToString());
    Console.WriteLine((a / b).ToString());
}
```

W konsoli powinniśmy zobaczyć kolejno $3/4$, $1/4$, $1/8$ i $2/1$.

Operatory porównania oraz metody Equals i GetHashCode

Możemy również zdefiniować operatory pozwalające na porównywanie dwóch instancji struktury *Ulamek*. Zdefiniujemy operatory `==`, `!=`, `<`, `>`, `<=` i `>=` oraz metody *Equals* i *GetHashCode*. W tym celu wracamy do edycji pliku *Ulamek.cs* i do bloku kodu zawierającego zdefiniowane w poprzednim ćwiczeniu operatory dodajemy definicję kolejnych operatorów oraz metod widocznych na listingu 4.12.

Listing 4.12. Definicja operatorów `==`, `!=`, `<`, `<=`, `>`, `i >=` oraz metod *Equals* i *GetHashCode*

```
// operatory logiczne
public static bool operator ==(Ulamek u1, Ulamek u2)
{
    return (u1.ToDouble() == u2.ToDouble());
}

public static bool operator !=(Ulamek u1, Ulamek u2)
{
    return !(u1 == u2);
}

public override bool Equals(object obj)
{
    if (!(obj is Ulamek)) return false;
    Ulamek u = (Ulamek) obj;
    return (this == u);
```

```
}

public override int GetHashCode()
{
    return Licznik ^ Mianownik;
}

public static bool operator >(Ulamek u1, Ulamek u2)
{
    return (u1.ToDouble()>u2.ToDouble());
}

public static bool operator >=(Ulamek u1, Ulamek u2)
{
    return (u1.ToDouble()>=u2.ToDouble());
}

public static bool operator <(Ulamek u1, Ulamek u2)
{
    return (u1.ToDouble()<u2.ToDouble());
}

public static bool operator <=(Ulamek u1, Ulamek u2)
{
    return (u1.ToDouble()<=u2.ToDouble());
}
```

Niektóre operatory muszą być definiowane jednocześnie. I tak operator == nie może być zdefiniowany bez operatora != oraz metod Equals i GetHashCode, operator < bez operatora > itd. W przypadku struktur sens operatora == sprowadza się w zasadzie do działania metody Equals. Natomiast w klasie operator == powinien sprawdzać, czy porównywane referencje są sobie równe, a więc czy przechowują ten sam adres i w konsekwencji czy wskazują na ten sam obiekt. Do porównywania stanu obiektów powinna służyć metoda Equals. Natomiast w przypadku struktur metodę Equals zazwyczaj implementuje się poprzez operator ==.

W przypadku definiowania operatora == i metody Equals w strukturze Ulamek konieczne jest podjęcie decyzji, w jaki sposób porównywać dwa ułamki. Czy przez równość ułamków rozumiemy dosłowną równość obiektów? W takim przypadku operator porównania powinien porównywać właściwości Licznik i Mianownik i jeżeli są takie same, to wtedy (i tylko wtedy) uznamy, że obiekty są identyczne, a operator == zwróci wartość true. A może należy brać pod uwagę interpretację tej struktury jako liczby wymiernej i sprawdzać, czy wartości ułamków utworzonych z liczników i mianowników porównywanych obiektów są sobie równe? W pierwszym przypadku porównanie jednej drugiej (1/2) i dwóch czwartych (2/4) da wynik negatywny, a w drugim — pozytywny. Powyżej zaimplementowaliśmy tę drugą możliwość, porównując ułamki uprzednio skonwertowane do liczb typu double.

Metoda GetHashCode służy do tworzenia liczby na podstawie stanu obiektu (tj. na podstawie wartości jego wszystkich pól). Liczba taka będzie równa dla obiektów, które mają ten sam stan. Nie ma jednak gwarancji, że dla różnych obiektów wyniki funkcji

haszującej będą różne. Tak jest zresztą w naszym przypadku: ułamek i jego odwrotność (np. $1/2$ i $2/1$) będą miały te same wartości. Niemniej jeżeli wyniki funkcji `GetHashCode` dla dwóch różnych obiektów nie są sobie równe, wywołanie metody `Equals` nie jest już potrzebne⁷.

Operatory konwersji

Kolejna rodzina operatorów to operatory konwersji. Przygotujemy operator konwersji na typ `double`, który będzie mógł być użyty jedynie jawnie, oraz operator konwersji z typu `int` na typ `Ulamek`, który będzie mógł być używany niejawnie.

1. Wcześniej zdefiniowaliśmy metodę `ToDouble`, której teraz użyjemy do zdefiniowania operatora konwersji na typ `double` (listing 4.13).

Listing 4.13. Definicja operatora jawnnej konwersji na typ `double`

```
public static explicit operator double(Ulamek u)
{
    return u.ToDouble();
}
```

2. Z kolei do zdefiniowania operatora konwersji z typu `int` na `Ulamek` użyjemy konstruktora z mianownikiem równym domyślnie jedności (listing 4.14).

Listing 4.14. Definicja operatora konwersji z typu `int` na `Ulamek`

```
public static implicit operator Ulamek(int n)
{
    return new Ulamek(n);
}
```

3. I na koniec przetestujmy nowe operatory w metodzie `Main` (zakładka *Program.cs*, listing 4.15).

Listing 4.15. Test operatorów konwersji

```
static void Main(string[] args)
{
    double r=(double)Ulamek.Polowa;
    Console.WriteLine(r.ToString());
    Ulamek c=2;
    Console.WriteLine(c.ToString());
}
```

⁷ Jeżeli metody `GetHashCode` chcemy użyć do rozróżnienia różnych obiektów (nawet różnych instancji tej samej klasy o takich samych stanach), to możemy uciec się do prostej sztuczki: w klasie zdefiniujmy prywatne statyczne pole typu `int`, którego wartość będziemy inkrementować (najlepiej użyć `Interlocked.Increment`, aby zapewnić bezpieczeństwo ze względu na wątki) i której użyjemy jako wartości zwracanej przez metodę `GetHashCode`. To zapewni jej unikalność dla każdej instancji.

Zdefiniowaliśmy dwa operatory konwersji: z typu `Ulamek` na typ `double` oraz z typu `int` na `Ulamek`. Dlaczego jeden jest jawnym, a drugi nie? W przypadku liczb wymiernych konwersja na liczbę rzeczywistą jest wprawdzie zawsze możliwa, ale wiąże się z utratą informacji (jedna trzecia (1/3) to nie do końca to samo co 0,3333333333333). I dlatego nie zezwalamy na niejawne wykonywanie konwersji na typ `double` (co gwarantuje modyfikator `explicit`). Dzięki temu programista powinien być świadomy stosowanego przybliżenia. Inaczej wygląda sprawa z konwersją liczby całkowitej na wymierną. Ta nie pociąga za sobą żadnego ryzyka, więc konwersja może być niejawna (modyfikator `implicit`).

Konwersja z liczb całkowitych do `Ulamek` bardzo ułatwia korzystanie z operatorów i funkcji matematycznych. Za jej pomocą możemy zainicjować ułamek liczbą całkowitą (np. `Ulamek u = 2;`) lub dodać ułamek do liczby całkowitej (możliwe są zarówno operacje `u+2`, jak i `2+u`) bez definiowania odpowiedniego operatora.

Implementacja interfejsu (na przykładzie `IComparable`)

Nic nie stoi na przeszkodzie, aby instancje struktury `Ulamek` były umieszczane w tablicach. Możemy np. zadeklarować dziesięcioelementową tablicę, co zostało pokazane na listingu 4.16.

Listing 4.16. Przykład definiowania tablicy zawierającej ułamki

```
Ulamek[] tablica = new Ulamek[10];
for (int i = 0; i < tablica.Length; i++) tablica[i] = new Ulamek(1, i+1);

foreach (Ulamek u in tablica)
    Console.WriteLine(u.ToString() + " = " + u.ToDouble().ToString());
```

Próba ich posortowania poleceniem `Array.Sort(tablica)`; skończy się jednak niepowodzeniem. Zgłoszony zostanie wyjątek `InvalidOperationException`. Metoda ta wymaga bowiem od sortowanych elementów, aby implementowały wspomniany w poprzednim rozdziale interfejs `IComparable<T>`, który wymusza na implementującej go klasie lub strukturze obecność metody `CompareTo` (z ang. *porównaj do*). Argumentem tej metody jest obiekt, do którego porównywana jest bieżąca instancja struktury.

Metoda `CompareTo` powinna zwracać wartość całkowitą, która jest mniejsza od zera, jeżeli obiekt porównywany jest większy od bieżącego (od obiektu wskazywanego przez `this`); równa zero, jeżeli oba obiekty są równe; i większa od zera, jeżeli to bieżący obiekt jest większy. My użyjemy liczb -1, 0 i 1.

1. Do sygnatury struktury `Ulamek` dodaj implementowany interfejs. Nowa sygnatura powinna wyglądać następująco:

```
struct Ulamek : IComparable<Ulamek>
```

2. W obrębie struktury `Ulamek` zdefiniuj metodę `CompareTo` widoczną na listingu 4.17.

Listing 4.17. Metoda `CompareTo` wymagana przez interfejs `IComparable<Ulamek>`

```
public int CompareTo(Ulamek u)
{
    double roznica = this.ToDouble() - u.ToDouble();
    if (roznica != 0) roznica /= Math.Abs(roznica);
    return (int)(roznica);
}
```

3. Teraz możesz posortować tablicę z listingu 4.16. W tym celu uzupełnij metodę `Main` o polecenia widoczne na listingu 4.18.

Listing 4.18. Sortowanie tablicy ułamków

```
private void Button1_Click(object sender, EventArgs e)
{
    Ulamek[] tablica = new Ulamek[10];
    for (int i = 0; i < tablica.Length; i++) tablica[i] = new Ulamek(1, i+1);

    Console.WriteLine("Przed sortowaniem:");
    foreach (Ulamek u in tablica)
        Console.WriteLine(u.ToString() + "=" + u.ToDouble().ToString());

    try
    {
        Array.Sort(tablica);
    }
    catch (Exception exc)
    {
        ConsoleColor bieżącyKolor = Console.ForegroundColor;
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(exc.Message);
        Console.ForegroundColor = bieżącyKolor;
    }

    Console.WriteLine("Po sortowaniu:");
    foreach (Ulamek u in tablica)
        Console.WriteLine(u.ToString() + "=" + u.ToDouble().ToString());
}
```

Na tym zakończymy implementację ułamka. Warto zwrócić uwagę, że zbudowaliśmy w miarę pełną strukturę, która pozwala na swobodne korzystanie z ułamków do wykonywania obliczeń.

Definiowanie typów parametrycznych

Typy parametryczne są w oficjalnej nomenklaturze języka C# nazywane **typami ogólnymi** (ang. *generic types*)⁸. Pozwalają na definiowanie klas i struktur bazujących na typie, który wskazywany jest przez programistę dopiero w momencie tworzenia

⁸ Warto zauważyć, że typy ogólne są powszechnym trendem w językach zarządzanych. Pojawiły się również w Javie 5.0.

ich instancji. W ten sposób można przygotować struktury danych, które mogą być używane do przechowywania obiektów różnego typu. W pewnym zakresie typy parametryczne są więc zarządzaną wersją szablonów z biblioteki STL języka C/C++.

Typy parametrów mogą być zawężone do pewnych grup, np. do klas, struktur czy typów posiadających konstruktor domyślny lub implementujących wskazany interfejs. Niestety nie ma sposobu, aby parametry ograniczyć do liczb (lub jeszcze lepiej do typów, w których zdefiniowane są operatory arytmetyczne). Typy „podstawowe” C# nie implementują bowiem wspólnego interfejsu. W konsekwencji nie ma prostego sposobu, aby zdefiniować typ parametryczny, w którym używane są operatory arytmetyczne. To oznacza też, że nie ma prostego sposobu, aby sensownie sparametryzować zdefiniowaną wyżej strukturę `Ulamek`. I choć można ten problem obejść⁹, w tej książce powinien znaleźć się raczej jakiś charakterystyczny przykład użycia typów ogólnych. A ich typowym zastosowaniem są kolekcje.

Definiowanie typów ogólnych

Przygotujmy zatem typ o nazwie `Para` zawierający dwa pola typu określonego przez parametr `T`: pierwszy i drugi.

1. Utwórz nowy projekt aplikacji konsolowej i dodaj do niego plik klasy (menu `Project/Add Class...`) o nazwie `Para.cs` (klasa zostanie automatycznie nazwana `Para`).
2. W pliku `Para.cs` rozbuduj definicję klasy `Para` według wzoru zamieszczonego na listingu 4.19.

Listing 4.19. Typ ogólny `Para`

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

namespace ParaDemo
{
    class Para<T>
    {
        private T pierwszy = default(T), drugi = default(T);

        public Para(T pierwszy, T drugi)
        {
            this.pierwszy = pierwszy;
            this.dragi = drugi;
        }

        public override string ToString()
        {
```

⁹ Różne podejścia do rozwiązywania problemu obliczeń w typach parametrycznych przedstawione są w artykule *Using Generics for Calculations* Rüdiger Klaehna opublikowanym w serwisie www.codeproject.com. Warto również zaznajomić się z artykułem Keitha Farmera pt. *Operator Overloading with Generics* z tego samego serwisu oraz ze stroną <http://www.yoda.arachsys.com/csharp/miscutil/usage/genericoperators.html>.

```
        return pierwszy.ToString() + "\t" + drugi.ToString();
    }
}
```

- 3.** W pliku *Program.cs*, w metodzie *Main*, umieść polecenia testujące nową klasę widoczne na listingu 4.20.

Listing 4.20. Tworzymy tablice Para<int>, Para<double> i Para<string>

```

static void Main(string[] args)
{
    Random r=new Random();

    // int
    Para<int>[] pi=new Para<int>[10];
    for (int i = 0; i < pi.Length; i++)
        pi[i]=new Para<int>(r.Next(10),r.Next(10));
    Console.WriteLine("Para<int>:");
    foreach (Para<int> para in pi) Console.WriteLine(para.ToString());

    // double
    Para<double>[] pd=new Para<double>[10];
    for (int i = 0; i < pd.Length; i++)
        pd[i] = new Para<double>(r.NextDouble(), r.NextDouble());
    Console.WriteLine("\nPara<double>:");
    foreach (Para<double> para in pd) Console.WriteLine(para.ToString());

    // string
    Para<string>[] ps = new Para<string>[12];
    ps[0] = new Para<string>("Bester", "Alfred");
    ps[1] = new Para<string>("Dick", "Philip");
    ps[2] = new Para<string>("Tolkien", "John");
    ps[3] = new Para<string>("Lem", "Stanisław");
    ps[4] = new Para<string>("Anderson", "Poul");
    ps[5] = new Para<string>("Pohl", "Frederik");
    ps[6] = new Para<string>("Le Guin", "Ursula");
    ps[7] = new Para<string>("Card", "Orson");
    ps[8] = new Para<string>("Gibson", "William");
    ps[9] = new Para<string>("Asimov", "Isaac");
    ps[10] = new Para<string>("Niven", "Larry");
    ps[11] = new Para<string>("Herbert", "Frank");
    Console.WriteLine("\nPara<string>:");
    foreach (Para<string> para in ps) Console.WriteLine(para.ToString());
}

```

Nowy parametryzowany typ `Para<T>` zawiera dwa prywatne pola typu określonego przez parametr `T`, o nazwach pierwszy i drugi. Klasa `Para<T>` zawiera również konstruktor, który jest jedynym sposobem inicjacji pól¹⁰, oraz metodę `ToString`, nadpisującą metodę odziedziczoną z klasy `System.Object`¹¹ niejawnej klasy bazowej.

¹⁰ W przypadku klas, a Para jest klasą, zdefiniowanie własnego konstruktora powoduje, że kompilator nie tworzy konstruktora domyślnego (bezargumentowego). Jak pamiętamy, inaczej jest w przypadku struktur, w których taki konstruktor powstaje zawsze.

¹¹ Jeżeli definiując klasę, nie wskażemy jawnie klasy bazowej, będzie nią klasa System.Object (alias object). Definiując struktury, dziedziczymy tak naprawdę z klasy System.ValueType, która dziedziczy

Słowo kluczowe `default` użyte do inicjacji pól klasy `Para` (tych, które są typu określonego przez parametr) zwraca wartość domyślną dla typu `T` podanego w jego argumentie, czyli 0 dla `int`, 0.0 dla `double` i łańcuch pusty dla `string`.

Określanie warunków, jakie mają spełniać parametry

W tej chwili parametrem `T` klasy `Para<T>` może być dowolny typ platformy .NET. W przykładzie z listingu 4.20 używamy typów `int`, `double` i `string`, ale mógłby to być również dobrze typ `Button` lub zdefiniowana przez nas wcześniej struktura `Ulamek`. Możemy więc jedynie założyć, że parametr posiada metody z klasy `object` — ostatecznej klasy bazowej wszystkich typów. To wystarczy do ich gromadzenia, kopiowania, przenoszenia i prezentacji użytkownikowi (klasa `object` posiada metodę `ToString`), czyli wszystkiego, co jest potrzebne, aby instancje klasy `Para<T>` mogły być elementami kolekcji. Kolekcja taka nie mógłaby być jednak sortowana. Nie wiemy, jak porównywać obiekty typu `Para<T>`. Do tego konieczne jest zaimplementowanie interfejsu `IComparable<Para<T>>` i zdefiniowanie metody `CompareTo` porównującej dwie pary. Jak mają być porównywane? Przyjmijmy następujący przepis: najpierw porównujemy pierwsze pola obu obiektów i dopiero jeżeli są one równe, o kolejności decyduje drugi element. Dzięki sparametryzowaniu klasy pola będą mogły być liczbami, łańcuchami lub innymi obiektami, które same mogą być porównywane. Od typu mającego pełnić funkcję parametru `T` wymagać musimy jedynie tego, aby sam implementował interfejs `IComparable<T>`, a więc aby można było porównywać nawzajem elementy tego typu.

Zacznijmy od nałożenia na parametr `T` żądania, aby implementował interfejs `IComparable<T>`. W tym celu zmieniamy sygnaturę klasy `Para<T>`, uzupełniając ją o kod wyraźniejący w listingu 4.21.

Listing 4.21. Żądamy, aby typ `T` zawierał metodę `CompareTo`

```
private class Para<T> where T : IComparable<T>
{
    private T pierwszy=default(T), drugi=default(T);

    public Para(T pierwszy,T drugi)
    {
        this.pierwszy=pierwszy;
        this.dragui=dragui;
    }

    public override string ToString()
    {
        return pierwszy.ToString() + "\t" + drugi.ToString();
    }
}
```

z klasy `Object`. Co ciekawe, klasy `ValueType` nie możemy sami wskazać jako klasy bazowej. Można jej natomiast użyć jako argumentu metody, jeżeli chcemy je ograniczyć do typów wartościowych.

Implementacja interfejsów przez typ ogólny

Żądanie, aby obiekty typu T można było ze sobą porównywać, to dopiero pierwszy krok do możliwości porównywania całych par. Jeżeli chcemy sortować elementy tablic pi, pd i ps zdefiniowanych w listingu 4.20, czyli elementy typu Para<int>, Para<double> i Para<string>, to także typ Para<T> musi implementować interfejs IComparable<Para<T>>. Wskażmy zatem ten interfejs w klasie Para<T>:

1. Jeszcze raz zmień sygnaturę klasy Para<T>, dodając do niej implementowany interfejs IComparable<Para<T>>. W konsekwencji do klasy należy dodać także metodę CompareTo widoczną na listingu 4.22.

Listing 4.22. Także klasa Para implementuje interfejs IComparable<>

```
private class Para<T> : IComparable<Para<T>> where T : IComparable<T>
{
    private T pierwszy=default(T), drugi=default(T);

    public Para(T pierwszy, T drugi)
    {
        this.pierwszy=pierwszy;
        this.dragi=dragi;
    }

    public override string ToString()
    {
        return pierwszy.ToString() + "\t" + drugi.ToString();
    }

    public int CompareTo(Para<T> innaPara)
    {
        int wartosc = this.pierwszy.CompareTo(innaPara.pierwszy);
        if (wartosc != 0) return wartosc;
        else return this.dragi.CompareTo(innaPara.dragi);
    }
}
```

2. Następnie przejdźmy do edycji pliku *Program.cs* i uzupełnijmy metodę Main o możliwe teraz polecenia sortowania i prezentację posortowanych tablic (listing 4.23).

Listing 4.23. W celu sortowania korzystamy ze statycznej metody Array.Sort

```
static void Main(string[] args)
{
    Random r = new Random();

    // int
    Para<int>[] pi = new Para<int>[10];
    for (int i = 0; i < pi.Length; i++) pi[i] = new Para<int>(r.Next(10), r.Next(10));
    Console.WriteLine("Para<int>:");
    foreach (Para<int> para in pi) Console.WriteLine(para.ToString());
    Array.Sort(pi);
    Console.WriteLine("\nPara<int> po sortowaniu:");
    foreach (Para<int> para in pi) Console.WriteLine(para.ToString());
```

```
//double
Para<double>[] pd = new Para<double>[10];
for (int i = 0; i < pi.Length; i++)
    pd[i] = new Para<double>(r.NextDouble(), r.NextDouble());
Console.WriteLine("\nPara<double>:");
foreach (Para<double> para in pd) Console.WriteLine(para.ToString());
Array.Sort(pd);
Console.WriteLine("\nPara<double> po sortowaniu:");
foreach (Para<double> para in pd) Console.WriteLine(para.ToString());

//string
Para<string>[] ps = new Para<string>[12];
ps[0] = new Para<string>("Bester", "Alfred");
ps[1] = new Para<string>("Dick", "Philip");
ps[2] = new Para<string>("Tolkien", "John");
ps[3] = new Para<string>("Lem", "Stanisław");
ps[4] = new Para<string>("Anderson", "Poul");
ps[5] = new Para<string>("Pohl", "Frederik");
ps[6] = new Para<string>("Le Guin", "Ursula");
ps[7] = new Para<string>("Card", "Orson");
ps[8] = new Para<string>("Gibson", "William");
ps[9] = new Para<string>("Asimov", "Isaac");
ps[10] = new Para<string>("Niven", "Larry");
ps[11] = new Para<string>("Herbert", "Frank");
Console.WriteLine("\nPara<string>:");
foreach (Para<string> para in ps) Console.WriteLine(para.ToString());
Array.Sort(ps);
Console.WriteLine("\nPara<string> po sortowaniu:");
foreach (Para<string> para in ps) Console.WriteLine(para.ToString());
}
```



Warto zauważyć, że o ile klasa może mieć tylko jedną klasę bazową, to może implementować wiele interfejsów jednocześnie. My tej możliwości nie wykorzystaliśmy.

Definiowanie aliasów

Zdefiniujmy aliasy dla `Para<int>`, `Para<double>` i `Para<string>`. Zmieniamy nazwę przestrzeni nazw w pliku `Para.cs` na `Para`. Dzięki temu klasa `Para` będzie znajdowała się w innej przestrzeni nazw niż klasa `Program`, a przykład zyska na ogólności. Następnie w sekcji `using` pliku `Program.cs` dodajemy linie widoczne na listingu 4.24.

Listing 4.24. Zmiany w sekcji `using`

```
using Para;
using ParaInt = Para.Para<int>;
using ParaDouble = Para.Para<double>;
using ParaString = Para.Para<string>;
```

Tworzenie aliasu może mieć miejsce tylko w sekcji poleceń `using`, czyli przed deklaracją bieżącej przestrzeni nazw. Po zdefiniowaniu aliasu możemy korzystać z typów `ParaInt`, `ParaDouble` i `ParaString` w taki sam sposób jak z `Para<int>`, `Para<double>` i `Para<string>`.

Typy ogólne z wieloma parametrami

Na koniec zauważmy, że nie jest konieczne, aby oba pola klasy `Para` (tj. pierwszy i drugi) były tego samego typu. Rozdzielimy je zatem, definiując szablon, w którym skorzystamy z dwóch parametrów typów `T` i `C` określających niezależnie typ obu przechowywanych wartości.

1. Możesz albo zdefiniować drugą klasę `Para` o innych parametrach (jest to w C# dopuszczalne), albo zmodyfikować istniejącą już klasę `Para` z pliku `Para.cs` (listing 4.25).

Listing 4.25. Wytnięte zostały zmiany względem oryginalnej klasy `Para`

```
public class Para<T,C> : IComparable<Para<T,C>>
    where T : IComparable<T>
    where C : IComparable<C>
{
    private T pierwszy = default(T);
    private C drugi = default(C);

    public Para(T pierwszy, C drugi)
    {
        this.pierwszy = pierwszy;
        this.dragi = drugi;
    }

    public override string ToString()
    {
        return pierwszy.ToString() + "\t" + drugi.ToString();
    }

    public int CompareTo(Para<T,C> innaPara)
    {
        int wartosc = this.pierwszy.CompareTo(innaPara.pierwszy);
        if (wartosc != 0) return wartosc;
        else return this.dragi.CompareTo(innaPara.dragi);
    }
}
```

2. W pliku `Program.cs` przedefiniuj aliasy dla par zawierających wyłącznie elementy typu `int`, `double` i `string` oraz dodaj przykład aliasu dla typu mieszanego (listing 4.26).

Listing 4.26. Nowe aliasy

```
using Para;
using ParaInt = Para.Para<int,int>;
using ParaDouble = Para.Para<double,double>;
using ParaString = Para.Para<string,string>;
using ParaIntDouble = Para.Para<int,double>;
```

3. Zdefiniuj nową zawartość metody testującej `Main` (listing 4.27).

Listing 4.27. Testujemy pary mieszane

```
private void Button1_Click(object sender, EventArgs e)
{
    Random r = new Random();

    // int-double
    ParaIntDouble[] p = new ParaIntDouble[10];
    for (int i = 0; i < p.Length; i++)
        p[i] = new ParaIntDouble(r.Next(10), r.NextDouble());
    Console.WriteLine("Para<int,double>:");
    foreach (ParaIntDouble para in p) Console.WriteLine(para.ToString());
    Array.Sort(p);
    Console.WriteLine("Para<int,double> po sortowaniu:");
    foreach (ParaIntDouble para in p) Console.WriteLine(para.ToString());
}
```

Rozszerzenia

Czasem zdarza się sytuacja, w której chcielibyśmy jakiś typ (klasę bądź strukturę) rozszerzyć o nową metodę, jednak nie jest to możliwe, ponieważ etap edycji tej klasy został już zamknięty lub po prostu nie mamy jej kodu (np. jest to jedna z klas platformy .NET). W takiej sytuacji możemy oczywiście wykorzystać dziedziczenie (zbacz poniżej), aby zdefiniować klasę potomną zawierającą potrzebną nam metodę. Konieczność zamiany interesującej nas klasy na przygotowaną przez nas klasę potomną w przygotowanym wcześniej kodzie może być jednak bardzo kłopotliwa, a czasem wręcz może to być niemożliwe. Istnieje na szczęście inne rozwiązanie. Jeżeli problem dotyczy platformy .NET 3.5 lub nowszej, możemy zdefiniować metodę rozszerzającą. *Metody rozszerzające* (ang. *extension methods*) lub po prostu rozszerzenia są definiowane jako metody statyczne w statycznych klasach, ale mogą być wywoływanne tak jak metody składowe innych klas. Dla przykładu „dodajmy” do klasy *String* metodę zmieniającą apostrof w znak *hash*. W tym celu musimy zdefiniować statyczną klasę niezagnieżdżoną w innej klasie (zdefiniowaną bezpośrednio w przestrzeni nazw) i umieścić w niej definicję statycznej metody, której argument jest typu, jaki rozszerzamy, i poprzedzony jest słowem kluczowym *this* (listing 4.28). Trochę to zawiłe, ale wystarczy spojrzeć na przykład, aby wszystko się wyjaśniło.

Listing 4.28. Rozszerzanie klasy *String* o metodę *UsunApostrof*

```
static class Rozszerzenia //poniższa klasa nie może być zagnieżdżona w innej
{
    public static string UsunApostrof(this String argument)
    {
        return argument.Replace('\'', '#');
    }
}
```

Od tej chwili możemy metodę *UsunApostrof* wywołać na rzecz dowolnego łańcucha (instancji klasy *String*), np. `Console.WriteLine("O\'Brien\'s Tower".UsunApostrof());`.

Rozszerzenie to widoczne jest również w listach podpowiedzi generowanych przez mechanizm *IntelliSense*.

Metoda rozszerzona może mieć więcej argumentów, ale tylko jeden (i to tylko pierwszy) ze słowem kluczowym `this`. Kolejne argumenty przy jej wywoływaniu będą używane jako normalne argumenty metody. Zdefiniujmy dla przykładu rozszerzenie, w którym jako argumentu możemy użyć znaku mającego w łańcuchu zastąpić apostrofy (listing 4.29).

Listing 4.29. Poniższa metoda powinna znaleźć się w klasie *Rozszerzenia*

```
public static string UsunApostrof(this String argument, char zamiennik)
{
    return argument.Replace('\'', zamiennik);
}
```

Możemy jej użyć następująco: `Console.WriteLine("O\'Brien\'s Tower".UsunApostrof('\''));`.

Często rozszerzenia pobierają jako argument delegacje do metod (robi tak większość rozszerzeń zdefiniowanych na potrzeby LINQ). Wówczas jako ich argumentów możemy używać wyrażeń lambda opisanych w poprzednim rozdziale. W naszym przypadku rozszerzenie `UsunApostrof` mogłoby potrzebować metody precyzującej, w jaki sposób ma zastępować znaki w łańcuchu. Aby to zrobić, w klasie *Rozszerzenia* zdefiniujmy typ delegacji:

```
public delegate char DZmieniacz(char znak);
```

lub użyjmy predefiniowanej delegacji `Func<char,char>`. Teraz możemy do klasy *Rozszerzenia* dodać kolejną wersję metody `UsunApostrof` (listing 4.30).

Listing 4.30. Rozszerzenie z delegacją jako argumentem

```
public static string UsunApostrof(this String argument, Func<char,char> zmieniacz)
{
    string wynik = "";
    foreach(char znak in argument.ToCharArray())
        wynik += zmieniacz(znak);
    return wynik;
}
```

Wywołanie tak zdefiniowanej metody aż prosi się o wykorzystanie wyrażenia lambda: `Console.WriteLine("O\'Brien\'s Tower".UsunApostrof(c => (c == '\'') ? '#' : c));`.

O rozszerzeniach należy powiedzieć jeszcze trzy rzeczy. Pierwsza: metodę rozszerzającą można zdefiniować także dla typu `object` (listing 4.31). A ponieważ rozszerzenia są przejmowane przez klasy potomne, dodalibyśmy w ten sposób metodę do wszystkich typów .NET. Druga: metoda nie musi zwracać wartości — może tylko wykonywać jakąś czynność, np. pokazywać komunikat (druga metoda w listingu 4.31). I trzecia: warto powtórzyć, że operatory LINQ zostały zdefiniowane właśnie jako metody rozszerzające.

Listing 4.31. Metody dodane do wszystkich typów

```
public static string PobierzNazweTypu(this object argument)
{
    return argument.GetType().FullName;
}

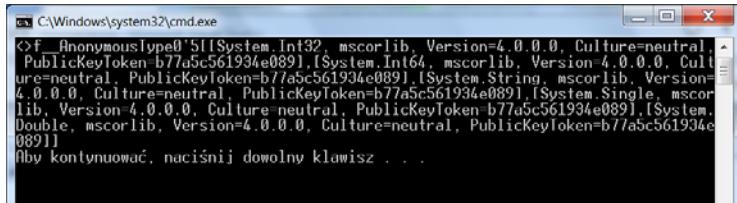
public static void UstawNull(this object argument)
{
    argument = null;
}
```

Typy anonimowe

Warto również wspomnieć o jeszcze jednej „innowacji” wprowadzonej w wersji 3.0 języka C#. Umożliwia ona tworzenie obiektów bez definiowania ich typu (klasy lub struktury). Możliwa jest konstrukcja typu:

```
var anonim=new {i=1, l=1L, s="Helion", f=1.0f, d=1.0};
```

W efekcie powstaje typ o pięciu polach publicznych tylko do odczytu, o nazwach i, l, s, f i d. Ich typy ustalane są na podstawie inicjującej je wartości (jak w typie var). Typ tworzonego obiektu jest dość złożony (na rysunku 4.1 pokazuję typ zwracany przez polecenie `anonim.GetType().FullName`), ale na szczęście wyposażeni jesteśmy w słowo kluczowe var, które pozwala go nie deklarować. Nowe składowe są poprawnie rozpoznawane przez mechanizm *IntelliSense* (rysunek 4.2) — nie ma więc kłopotu z korzystaniem z tak utworzonego obiektu. Nowy sposób tworzenia obiektu jest szczególnie wygodny przy zwracaniu danych w zapytaniach LINQ.



The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. The command entered is 'anonim.GetType().FullName'. The output is a long string representing the full name of the anonymous type, which includes 'System.Nullable`1[System.Int32]', 'System.Nullable`1[System.Int64]', 'System.String', 'System.Double', and 'System.Single'.

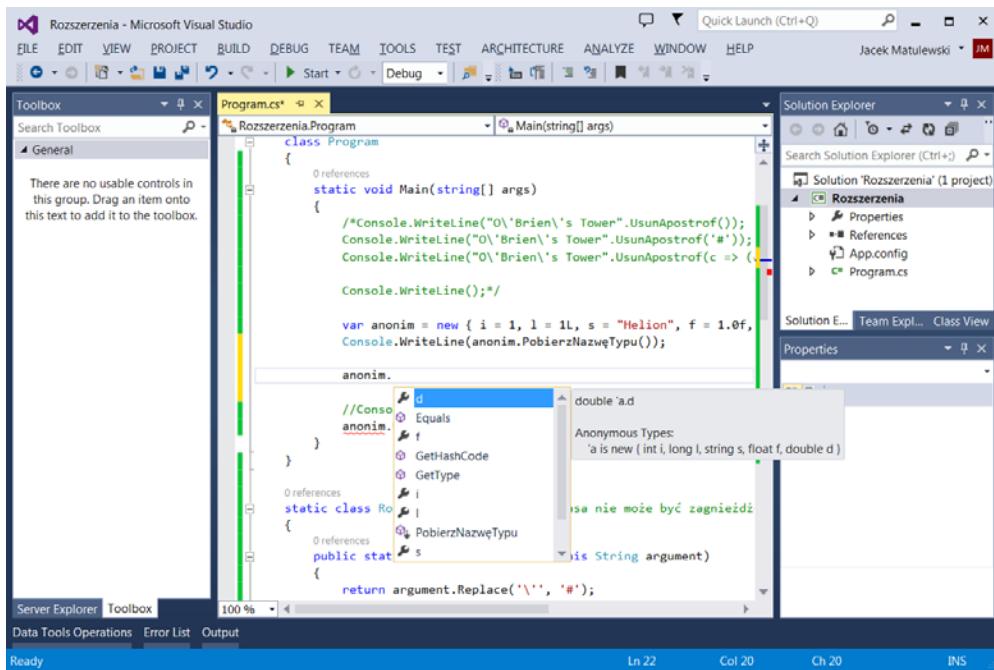
Rysunek 4.1. Po komplikacji struktura typu anonimowego jest już znana

Dziedziczenie



Wskazówka

W tym miejscu zaczynają się zagadnienia, które można nazwać zaawansowanym programowaniem obiektowym. Z pewnością należą one do kanonu wiedzy o programowaniu w C#, ale nie będą używane w dalszej części książki, dlatego omawiam je bardziej hasłowo.



Rysunek 4.2. Typy anonimowe są poprawnie rozpoznawane przez narzędzia uzupełniania kodu

Definiując klasę, możemy wskazać klasę, z której definiowany typ ma przejąć gotowe pola, metody, własności i zdarzenia. Nazywa się to **dziedziczeniem** (ang. *inheritance*). Klasę, która przejmuje te składowe, nazywa się **potomną**, a klasę, z której są one przejmowane — **bazową**. A zatem klasa potomna dziedziczy po klasie bazowej. Jak to wygląda w praktyce? Mogliśmy się już o tym przekonać, definiując klasę `Para<>`. Nie wskazaliśmy wprawdzie jawnie jej klasy bazowej, ale w takiej sytuacji jest nią klasa `System.Object`. Z niej klasa `Para<>` odziedziczyła m.in. metodę `ToString`, którą my nadpisaliśmy (listing 4.19).

Wskazówka

Definiując struktury, nie możemy (inaczej niż w przypadku klas) wskazać klas bazowej. Swobodne dziedziczenie dotyczy tylko typów referencyjnych. Typy wartościowe zawsze dziedziczą z klasy `ValueType`, która dziedziczy bezpośrednio z klasy `Object`.

Klasy bazowe i klasy potomne

Aby lepiej zrozumieć mechanizm dziedziczenia, przygotujmy prosty przykład klasy bazowej `Osoba` i dziedziczącej po niej klasy potomnej `OsobaZameldowana`. Obie klasy widoczne są na listingu 4.32. Towarzyszy im klasa `Adres`, której używam w klasie potomnej. Wszystkie te klasy można wstawić zarówno do klasy `Program` z projektu aplikacji konsolowej, jak i bezpośrednio do przestrzeni nazw. Na tym etapie nie ma to większego znaczenia.

Listing 4.32. Ilustracja relacji „jest” i „ma”. Wyjaśnienie poniżej

```
class Osoba
{
    public string Imię;
    public string Nazwisko;
    public int Wiek;

    public override string ToString()
    {
        return Imię + " " + Nazwisko + " (" + Wiek + ")";
    }
}

class Adres
{
    public string Miasto;
    public string Ulica;
    public int NumerDomu;
    public int? NumerMieszkania;

    public override string ToString()
    {
        return Miasto + ", ul. " + Ulica + " " + NumerDomu +
            (NumerMieszkania.HasValue ? ("/" + NumerMieszkania) : "");
    }
}

class OsobaZameldowana : Osoba
{
    public Adres AdresZameldowania;

    public override string ToString()
    {
        return base.ToString() + "; " + AdresZameldowania.ToString();
    }
}
```

We wszystkich trzech klasach zdefiniowaliśmy metody `ToString`. Nadpisują one metodę `ToString` z klasy `Object`, co oznaczyliśmy, używając modyfikatora `override`. Ponadto metoda `ToString` z klasy `OsobaZameldowana` nadpisuje metodę z klasy bazowej `Osoba`. Nie definiuje jednak całej swojej funkcjonalności od nowa, a używa implementacji z klasy bazowej do utworzenia tej części łańcucha, która związana jest z polem zdefiniowanym w klasie `Osoba`. W tym celu użyte zostało słowo kluczowe `base`. W konstrukcji `base.ToString()` wskazuje ono, że chodzi o wywołanie metody `Osoba.ToString`, a nie metody bieżącego obiektu (bez niego uzyskalibyśmy nieskończoną rekurencję wywołań).

Moglibyśmy sprawdzić działanie powyższych klas, tworząc przykładową instancję klasy `OsobaZameldowana`. Pokazuje to listing 4.33.

Listing 4.33. Tworzenie instancji klasy potomnej

```
static void Main(string[] args)
{
    OsobaZameldowana jk = new OsobaZameldowana()
    {
        Imię = "Jan",
        Nazwisko = "Kowalski",
        Wiek = 42,
        AdresZameldowania = new Adres
        {
            Miasto = "Toruń",
            Ulica = "Grudziądzka",
            NumerDomu = 5,
            NumerMieszkania = null
        }
    };
    Console.WriteLine(jk.ToString());
}
```

W listingu 4.32 zdefiniowane są trzy klasy. Zachodzą między nimi różne relacje. W założeniu mają one odpowiadać relacjom między pojęciami z „prawdziwego” świata, który chcemy w klasach odwzorować. Po pierwsze: relacja dziedziczenia między klasą `OsobaZameldowana` i jej klasą bazową `Osoba` odpowiada relacji „jest”. `Osoba zameldowana` **jest** osobą. Jeżeli taką relację możemy wskazać między dwoma pojęciami użytymi do opisu problemu, to znaczy, że definiując klasy, powinniśmy użyć dziedziczenia. `Pies` jest zwierzęciem, a więc klasa `Pies` powinna dziedziczyć z klasy `Zwierzę`. Analogicznie samochód osobowy jest pojazdem, ułamek jest liczbą, trójkąt, prostokąt i okrąg są figurami itd. Dziedziczenie może być wielokrotne: np. pies jest ssakiem, który jest zwierzęciem, który z kolei jest organizmem żywym, itd.

Inna relacja zachodzi między klasą `OsobaZameldowana` a klasą `Adres`. `Osoba zameldowana` **ma** adres. W takiej sytuacji adres powinien być elementem składowym klasy opisującej osobę zameldowaną. Adres jest tym, co odróżnia osobę zameldowaną od osoby jako takiej. Rozszerzamy zatem klasę `Osoba`, dodając adres. Tym samym zawężamy zakres obiektów, które klasa potomna `OsobaZameldowana` opisuje. Mówimy, że klasa `Osoba ↞ Zameldowana` jest bardziej **wyspecjalizowana** od klasy `Osoba`.

Projektując architekturę systemu i zastanawiając się, jakie klasy musimy zdefiniować i jakie powinny być między nimi relacje, zawsze powinniśmy zadawać sobie dwa pytania. Pierwszym jest pytanie o to, jakich pojęć (rzeczowników) używam do opisu problemu. Powinny im odpowiadać typy, jakie należy zdefiniować. Natomiast drugie pytanie brzmi: jakie relacje wiążą te pojęcia? Czy do opisu relacji między tymi pojęciami używam słów „jest” czy „ma”? Determinuje to wybór między dziedziczeniem a definiowaniem jako elementu składowego. Oczywiście nie zawsze relacja „ma” musi oznaczać posiadanie lub zawieranie przestrzenne. Rodzic może mieć dzieci — wówczas polem składowym klasy `Rodzic` jest kolekcja `Dzieci`, samochód może mieć silnik i skrzynię biegów, a figura pewną ilość wierzchołków. Relacja „ma” może również opisywać posiadanie jakiejś własności, np. samochód może mieć jakąś prędkość, ma kolor, ma bagażnik, który ma jakąś wielkość, itd.



Wskazówka

Rzadko używanym w praktyce modyfikatorem jest sealed (z ang. zapieczętowana). Tak oznaczone powinny być klasy, które nie zostały zaprojektowane do dziedziczenia, i w efekcie chcemy zablokować taką możliwość. Tego modyfikatora używa się przy definiowaniu singletonów (podrozdział „Singleton”). Modyfikatorem sealed można również oznaczyć metody. Nie mogą one być nadpisywane, a tym samym ich funkcjonalność w klasach potomnych nie może być zmieniona.

Wszystkie pola składowe w klasie bazowej Osoba zadeklarowaliśmy jako publiczne (modyfikator public). Jeżeli byłyby prywatne (modyfikator private), to nie byłyby widoczne w klasie potomnej. Z kolei pola chronione (modyfikator protected) byłyby w klasie potomnej widoczne, ale już nie poza nią. Pola zdefiniowane w klasie bazowej jako chronione w klasie potomnej stają się niejako prywatne. To samo dotyczy własności, metod i zdarzeń. Aby się o tym przekonać, zdefiniujmy w klasie bazowej prywatną własność tylko do odczytu Personalia, łączącą imię i nazwisko w jeden łańcuch (listing 4.34).

Listing 4.34. Elementy prywatne są dziedziczone przez klasy potomne, ale nie są z nich dostępne

```
class Osoba
{
    public string Imię;
    public string Nazwisko;
    public int Wiek;

    public override string ToString()
    {
        return Imię + " " + Nazwisko + " (" + Wiek + ")";
    }

    private string Personalia
    {
        get
        {
            return Imię + " " + Nazwisko;
        }
    }
}
```

Korzystając z *IntelliSense*, możemy się łatwo przekonać, że nowa własność nie będzie dostępna ani z metody Main, w której możemy próbować odwoływać się do niej na rzecz obiektu jk (z listingu 4.33), ani z klasy potomnej (możemy próbować użyć jej chociażby w nadpisanej metodzie ToString). To się zmieni, jeżeli modyfikator private zmienimy na protected. Własność Personalia nadal nie będzie dostępna z metody Main, ale będzie widoczna w metodach i właściwościach klasy potomnej.

Nadpisywanie a przesłanianie metod

W klasie bazowej Osoba zdefiniowana jest wirtualna metoda ToString. W klasie potomnej OsobaZameldowana metoda ta jest nadpisana (ang. override), aby uwzględnić obecność adresu. W efekcie bez względu na typ referencji, na rzecz której wywoływana

jest ta metoda (`Osoba` lub `OsobaZameldowana`), jeżeli tylko utworzymy obiekt klasy potomnej `OsobaZameldowana`, to wykonywana będzie metoda z tej klasy.

Aby się o tym przekonać, do metody `Main` dopisz dwie instrukcje:

```
static void Main(string[] args)
{
    ...
    Console.WriteLine(jk.ToString());
    Osoba jkb = jk;
    Console.WriteLine(jkb.ToString());
}
```

W efekcie zobaczymy w konsoli dwałańcuchy zawierające poza imieniem, nazwiskiem i wiekiem także adres. A teraz w definicji metody `ToString` w klasie potomnej `OsobaZameldowana` zmień modyfikator `override` na `new`:

```
public new string ToString()
{
    ...
}
```

i jeszcze raz uruchommy aplikację. Okaże się, że tym razemłańcuchy widoczne w konsoli są różne. W przypadku referencji `jkb` typu `Osoba` wykonana została metoda zdefiniowana w tej klasie (bez adresu), natomiast w przypadku referencji `jk` typu `OsobaZameldowana` — metoda z klasy potomnej, pokazująca adres.

W przypadku użycia modyfikatora `new` w metodzie z klasy potomnej mamy do czynienia z tzw. *przesłananiem* (ang. *hiding*) metody z klasy bazowej przez metodę z klasy potomnej. Należy od razu zastrzec, że modyfikator `new` stosowany jest bardzo rzadko, w zasadzie tylko w sytuacjach, w których obie metody pomimo jednakowej nazwy i listy argumentów semantycznie nie mają ze sobą nic wspólnego. Dlaczego wobec tego o tym piszę? Z jednego powodu: jeżeli w sygnaturze metody potomnej zabraknie modyfikatora `override`, to domyślnie metoda będzie przesłaniać metodę z klasy bazowej, a nie ją nadpisywać. Wprawdzie pojawi się ostrzeżenie, ale jeżeli nie będzie jedyne, łatwo je przeoczyć.



Zmień z powrotem modyfikator metody `OsobaZameldowana.ToString` z `new` na `virtual`!

Klasy abstrakcyjne

Podany wyżej przepis na projektowanie systemu klas jako odwzorowania pojęć używanych do opisania problemu sprawdza się całkiem nieźle w praktyce. Zauważmy jednak, że nie wszystkim pojęciom odpowiadają rzeczywiste obiekty. O ile z łatwością możemy sobie wyobrazić przykłady prostokątów, okręgów czy trójkątów, to wyobrażenie sobie figury samej w sobie, która nie byłaby jakąś konkretną figurą z określoną liczbą wierzchołków, jest niemożliwe. Co wcale nie oznacza, że nie możemy sprawnie operować abstrakcyjnym pojęciem figury. Podobnie łatwo wyobrazić sobie samochód osobowy, ale trudno pojazd, który nie byłby ani samochodem osobowym, ani ciężarowym, ani rowerem czy furmanką. Projektując zbiór klas, możemy zablokować możliwość tworzenia instancji takich klas opisujących najbardziej ogólne pojęcia.

Używa się do tego słowa kluczowego `abstract`, a klasy takie nazywa się abstrakcyjnymi. Listing 4.35 pokazuje przykład abstrakcyjnej klasy `Figura` i trzech dziedziczących z niej klas `Okrąg`, `Trójkąt` i `Koło`.

Listing 4.35. Przykład z figurami to klasyczny przykład jeszcze z pierwszych książek o C++

```
abstract class Figura {};
class Okrąg : Figura {}
class Trójkąt : Figura {}
class Prostokąt : Figura {}
class Kwadrat : Prostokąt {}
```

Próba utworzenia instancji abstrakcyjnego obiektu zakończy się błędem kompilatora:

```
Figura f = new Figura(); // błąd kompilatora
```

Można natomiast użyć klasy abstrakcyjnej do utworzenia zmiennej typu referencyjnego:

```
Figura f;
```

a w takiej zmiennej można zapisać referencję do instancji jej klasy potomnej:

```
Figura f = new Kwadrat();
```

Po co? To wyjaśnię za chwilę.

Klasa abstrakcyjna może (choć nie musi) zawierać metody abstrakcyjne, a więc takie, które są zadeklarowane, ale nie posiadają ciała (listing 4.36). Metody takie muszą być natomiast zdefiniowane w klasach potomnych, które nie są klasami abstrakcyjnymi. W przeciwnym razie pojawi się błąd kompilatora.

Listing 4.36. Metoda abstrakcyjna i jej implementacja w klasach potomnych

```
abstract class Figura
{
    public abstract int IleWierzchołków();
}

class Okrąg : Figura
{
    public override int IleWierzchołków()
    {
        return 0;
    }
}

class Trójkąt : Figura
{
    public override int IleWierzchołków()
    {
        return 3;
    }
}

class Prostokąt : Figura
{
```

```

    public override int IleWierzchołków()
    {
        return 4;
    }
}

class Kwadrat : Prostokąt {}

```

Abstrakcyjne mogą także być własności (ale nie pola). Oto przykład takiej własności z klasy bazowej:

```

abstract class Figura
{
    public abstract int IleWierzchołków();
    public abstract int LiczbaWierzchołków { get; }
}

```

Jej obecność w klasie bazowej Figura wymusza konieczność zdefiniowania tej własności we wszystkich klasach bazowych. Oto przykład jej nadpisania z klasy Okrąg:

```

class Okrąg : Figura
{
    public override int LiczbaWierzchołków
    {
        get
        {
            return IleWierzchołków();
        }
    }
    ...
}

```

Jak już wiemy, do oznaczenia, że własność lub metoda nadpisuje element składowy z klasą bazową, używamy modyfikatora `override`. Nadpisywane własności `LiczbaWierzchołków` nie mogą mieć sekcji `set`, ponieważ sekcja ta nie została uwzględniona we właściwości abstrakcyjnej z klasy `Figura`.

Metody wirtualne

Takie definiowanie własności `LiczbaWierzchołków` nie ma jednak większego sensu. Naszą uwagę powinno zwrócić to, że jej definicja wyglądałaby tak samo we wszystkich klasach potomnych. W takiej sytuacji bardziej naturalne byłoby zdefiniowanie jej w klasie bazowej. Nie szkodzi, że to jest klasa abstrakcyjna — nie wszystkie metody takiej klasy muszą być abstrakcyjne. Aby nie blokować możliwości nadpisywania, oznaczmy ją jednak jako wirtualną:

```

abstract class Figura
{
    public abstract int IleWierzchołków();

    public virtual int LiczbaWierzchołków
    {
        get
        {
            return IleWierzchołków();
        }
    }
}

```

```
        }  
    }  
}
```

Aby można było zdefiniować metody abstrakcyjne, konieczne jest oznaczenie całej klasy jako abstrakcyjnej. Metody takie nie mają bowiem ciała. Muszą być wobec tego zdefiniowane w klasach potomnych, które już abstrakcyjne nie są. Są więc de facto komunikatem, że klasy potomne mają mieć pewną funkcjonalność, która nie może być zdefiniowana w klasie bazowej albo z powodu braku odpowiednich informacji (np. liczby wierzchołków), albo dlatego, że jest ona zbyt abstrakcyjna, aby można było ją zaimplementować w kodzie.

Czasem możliwa jest jednak sytuacja, w której można pokusić się o zdefiniowanie metody w klasie bazowej, dopuszczając jej przeddefiniowanie w klasie potomnej. Z taką sytuacją mamy do czynienia w przypadku własności `LiczbaWierzchołków`. Takie metody nazywamy wirtualnymi i oznaczamy modyfikatorem `virtual`. Dobrym przykładem takiej metody jest `ToString` z klasy `Object`. Wersja tej metody zdefiniowana w klasie `Object` zwraca po prostu nazwę typu. Wiele klas nadpisuje ją jednak, określając bardziej pozytyczny sposób konwersji na łańcuch. My zrobiliśmy to w strukturze `Ulamek` oraz klasach `Para<>` czy `Osoba`.

Różnica między metodą wirtualną i abstrakcyjną jest więc taka, że metodę abstrakcyjną trzeba nadpisać, a metodę wirtualną jedynie można. Co więcej, nadpisując metodę wirtualną, można odwołać się do jej implementacji w klasie bazowej (*vide casus* metody `OsobaZameldowana.ToString` z listingu 4.32). W przypadku metody abstrakcyjnej nie ma się do czego odwoływać, bo w klasie bazowej nie ma ona przecież ciała.

Polimorfizm

Załóżmy, że mamy program, który przechowuje zbiór figur. Chcemy mieć możliwość narysowania ich wszystkich w podobny sposób¹². Problem w tym, że inaczej rysowany jest okrąg, trójkąt i prostokąt. Jak ten problem rozwiązać? Kanonicznym rozwiązaniem jest zdefiniowanie klasy bazowej `Figura` z abstrakcyjną metodą `Rysuj`. Następnie definiujemy zbiór klas potomnych `Okrąg`, `Trójkąt`, `Prostokąt` i `Kwadrat` odpowiadających poszczególnym figurom, w których metodę `Rysuj` nadpisujemy. Wówczas możemy utworzyć listę typu `List<Figura>`, która może przechowywać instancje wszystkich klas potomnych klasy `Figura`, a więc obiekty typu `Okrąg`, `Trójkąt`, `Prostokąt` i `Kwadrat`. I wreszcie tworzymy pętlę, która wywołuje metodę `Rysuj` po kolej na rzecz wszystkich elementów tej kolekcji. Czy wywołanie metody `Rysuj` na rzecz referencji typu `Figura` nie skończy się jednak błędem? Przecież w klasie `Figura` ta metoda nie ma nawet ciała. Otóż właśnie nie i to jest kluczowa własność mechanizmu metod wirtualnych i abstrakcyjnych. Bez względu na to, na rzecz jakiego typu referencji wywoływane będą metody obiektu, użyte będą definicje właściwe dla jego prawdziwego typu. Mechanizm ten nazywa się **polimorfizmem** i w skrócie oznacza, że obiekty różnych, choć spokrewnionych typów możemy traktować w jednorodny sposób.

¹² Zobacz <http://msdn.microsoft.com/en-us/library/ms173152.aspx>.

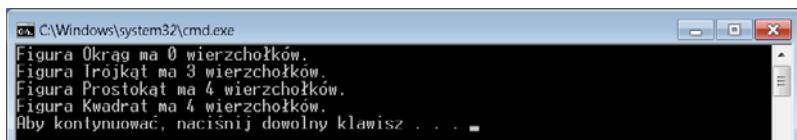
Pokazuje to listing 4.37, który wprawdzie nie używa metody Rysuj, bo jej implementacja byłaby trudna w oknie konsoli, ale ilustruje działanie mechanizmu metod wirtualnych na przykładzie metody IleWierzchołków. Efekt, jaki zobaczymy, widoczny jest na rysunku 4.3.

Listing 4.37. Polimorfizm w najprostszym przykładzie

```
static void Main(string[] args)
{
    List<Figura> figury = new List<Figura>();
    figury.Add(new Okrąg());
    figury.Add(new Trójkąt());
    figury.Add(new Prostokąt());
    figury.Add(new Kwadrat());
    foreach (Figura figura in figury)
        Console.WriteLine("Figura " + figura.GetType().Name + " ma " +
            figura.IleWierzchołków());
}
```

Rysunek 4.3.

Przykład użycia
metod wirtualnych



Przykład ten obrazuje podstawową ideę polimorfizmu uzyskiwanego dzięki dziedziczeniu. A mianowicie obiekty wielu różnych typów (Okrąg, Trójkąt, Prostokąt i Kwadrat) traktujemy w podobny sposób. Umożliwia to ich wspólny mianownik, czyli klasa bazowa określająca zakres funkcjonalności, która jest na pewno dostępna w każdej z nich. Funkcjonalności te mogą się jednak różnić — choć sygnatura metod IleWierzchołków musi być taka sama we wszystkich klasach potomnych, to jej definicja może być inna w każdej z tych klas. Dzięki temu możemy obiekty poszczególnych figur gromadzić we wspólnej kolekcji zdefiniowanej dla parametru Figura. Scenariuszy użycia polimorfizmu jest zresztą wiele. Możemy na przykład przygotować metodę, która przyjmuje jako argument klasę bazową Figura. Wówczas funkcjonalność obiektów przesyłanych do tej metody ograniczona jest do tej określonej w klasie Figura¹³, ale wywoływane metody są odpowiednie dla typu przesyłanego argumentu (listing 4.38).

Listing 4.38. Użycie polimorfizmu w metodzie obsługującej wiele typów

```
static void WyświetlIlośćWierzchołków(Figura figura)
{
    Console.WriteLine("Liczba wierzchołków figury: " + figura.IleWierzchołków());
```

¹³ Cała prawda jest taka, że obiekt zawsze zna swój typ dzięki mechanizmowi *Reflection*. Zatem zawsze możemy upewnić się, że przesłana do metody figura jest np. prostokątem (pozwala na to operator `is`), i jeżeli tak, zrzutować na typ `Prostokąt`. W ten sposób można odzyskać pełną funkcjonalność typu `Prostokąt`. Co więcej, jeżeli nie znamy typu obiektu, jaki przesyłamy, mechanizm *Reflection* pozwala na jego analizę i wywołanie dowolnej metody.

Konstruktory a dziedziczenie

Warto w kontekście dziedziczenia zwrócić uwagę na konstruktory. Na początek zdefiniujmy domyślny konstruktor w klasie bazowej Figura oraz konstruktor w jednej z jego klas potomnych, np. Trójkąt (listing 4.39).

Listing 4.39. Konstruktory domyślne dodane do klasy bazowej i potomnej

```
abstract class Figura
{
    ...
    public Figura()
    {
        Console.WriteLine("Konstruktor klasy Figura");
    }
}

...
class Trójkąt : Figura
{
    ...
    public Trójkąt()
    {
        Console.WriteLine("Konstruktor klasy Trójkąt");
    }
}
```

Co się stanie, jeżeli utworzymy obiekt reprezentujący trójkąt polecienniem new Trójkąt();? Wydruk z konsoli przekona nas, że najpierw wywołany zostanie konstruktor klasy bazowej Figura, a dopiero po nim konstruktor klasy potomnej Trójkąt. Nie ma tu wobec tego mowy o żadnej wirtualności czy nadpisywaniu konstruktorów — gdy powstaje obiekt, który jest figurą, wywoływany jest konstruktor klasy Figura. Dzięki temu w konstruktorze klasy potomnej nie trzeba, a wręcz nie należy powtarzać żadnych instrukcji inicjujących elementy klasy bazowej (chyba że chcemy zmienić ich wartość). Za ich inicjację powinien być odpowiedzialny wyłącznie konstruktor klasy bazowej.

Teraz zdefiniujmy w klasie bazowej i potomnej konstruktory z jakimś argumentem, np. typu string (listing 4.40).

Listing 4.40. Konstruktory z argumentem typu string

```
abstract class Figura
{
    ...
    public Figura()
    {
        Console.WriteLine("Konstruktor klasy Figura");
    }
}
```

```
public Figura(string argument)
{
    Console.WriteLine("Konstruktor klasy Figura, komunikat: " + argument);
}
}

class Trójkąt : Figura
{
    ...

    public Trójkąt()
    {
        Console.WriteLine("Konstruktor klasy Trójkąt");
    }

    public Trójkąt(string argument)
    {
        Console.WriteLine("Konstruktor klasy Trójkąt, komunikat: " + argument);
    }
}
```

Jakie konstruktory zostaną wywołane, gdy utworzymy obiekt trójkąta poleciением new Trójkąt("jestem trójkątem")? Łatwo się domyślić, że zostanie wywołany nowy konstruktor klasy potomnej Trójkąt, ale zaskoczeniem może być, że wcześniej zostanie wywołany domyślny, czyli bezargumentowy konstruktor klasy bazowej Figura. Jeżeli chcemy, aby było inaczej, tj. aby wywołany był konstruktor z argumentem string, należy to wyraźnie zaznaczyć w konstruktorze klasy potomnej w sposób widoczny na listingu 4.41.

Listing 4.41. Wskazywanie konstruktora klasy bazowej

```
public Trójkąt(string argument)
    : base(argument)
{
    Console.WriteLine("Konstruktor klasy Trójkąt, komunikat: " + argument);
}
```

Pamiętamy, że jeżeli klasa nie posiada żadnego konstruktora, kompilator tworzy automatycznie konstruktor domyślny inicjujący wszystkie pola wartościami domyślnymi dla ich typów. Jeżeli w takiej klasie sami zdefiniujemy konstruktor, ale nie będzie to konstruktor domyślny, to kompilator nie utworzy już konstruktora domyślnego. Zatem jeżeli w klasie bazowej zdefiniujemy konstruktor z jakimś argumentem, nie definiując jednocześnie konstruktora domyślnego, to wszystkie klasy potomne będą musiały posiadać konstruktory, w których wskażą jawnie ów konstruktor. Inaczej pojawi się błąd komplikacji. Możemy się o tym przekonać, usuwając konstruktor domyślny z klasy Figura.

Singleton

Zdarzają się sytuacje, w których chcemy ograniczyć liczbę instancji klasy czy wręcz umożliwić utworzenie tylko jednej. Powodów może być kilka. Taki unikatowy egzemplarz może służyć do przechowywania ustawień aplikacji, które dostępne są w całej aplikacji, i nie jest potrzebne, a nawet szkodliwe byłoby ich powielanie. Taki obiekt używany jest *de facto* zamiast zbioru zmiennych globalnych. Innym przykładem jest klasa zaprojektowana jako reprezentacja jakiegoś zewnętrznego unikatowego zasobu (np. okna konsoli, bazy danych itp.). Wzorzec projektowy, który opisuje taką klasę, nazywany jest singletonem.

Załóżmy, że chcemy utworzyć prosty singleton, którego jedynym zadaniem będzie przechowanie i udostępnienie wartości typu `string`. Ponadto będzie zawierał metodę wyświetlającą tę zmienną w konsoli. Singleton można zaprojektować na wiele sposobów, ale tu przedstawię tylko trzy najbardziej popularne: statyczną klasę, implementację opartą na użyciu statycznego pola tylko do odczytu oraz implementację używającą statycznej metody inicjującej kontrolującej liczbę utworzonych instancji¹⁴.

Pierwsze rozwiązanie, czyli statyczna klasa, to proste i eleganckie rozwiązanie, ale mające pewne ograniczenia. Podstawową własnością statycznej klasy jest to, że nie można utworzyć jej instancji. Nie pozwala na to kompilator. W efekcie możemy odnosić się jedynie do statycznych pól, właściwości i metod, które siłą rzeczy funkcjonują w jednym egzemplarzu dostępnym na rzecz klasy. Pola mogą być dowolnie zmieniane, co powoduje, że statyczna klasa może posiadać pewien stan, który odpowiada stanowi instancji zwykłej klasy. Brak rzeczywistej instancji uniemożliwia jednak przesłanie jej jako argumentu metody; nie można nawet utworzyć referencji tego typu. Ponadto klasa statyczna nie może dziedziczyć z klasy innej niż `Object` i nie może implementować interfejsów. To w zasadniczy sposób ogranicza jej użyteczność w niektórych scenariuszach. Nie może być też klasą bazową w dziedziczeniu, co zapobiega próbom utworzenia instancji klasy potomnej takiego singletonu. Wiele klas w platformie .NET jest klasami statycznymi. My dobrze poznaliśmy klasę statyczną `System.Console` umożliwiającą korzystanie z konsoli. Implementację naszego przykładu singletonu za pomocą klas statycznej prezentuje listing 4.42.

Listing 4.42. Singleton jako klasa statyczna

```
public static class KlasaStatyczna
{
    private static string pole = "Domyślna wartość pola";

    public static string Właściwość
    {
        get
        {
            return pole;
        }
        set
    }
}
```

¹⁴ Dodatkowym, nie omawianym tu zagadnieniem jest zapewnienie bezpieczeństwa singletonu w kontekście wielu wątków, zobacz <http://msdn.microsoft.com/en-us/library/ff650316.aspx>.

```

        {
            pole = value;
        }
    }

    public static void Metoda()
    {
        Console.WriteLine(pole);
    }
}

```

Sposób użycia klasy statycznej jest prosty i polega na odwoływaniu się do jej statycznych pól i metod. Możliwe to jest jedynie na rzecz klasy:

```
KlasaStatyczna.Własność = "Wartość pola ustalona w metodzie Program.Main";
KlasaStatyczna.Metoda();
```

Nie ma mowy o tworzeniu instancji tej klasy ani nawet o tworzeniu zmiennej referencyjnej tego typu.

Dwa pozostałe sposoby opierają się na pewnym triku — należy ukryć konstruktor domyślny, definiując go jako prywatny, i umożliwić pobranie instancji klasy za pomocą publicznej metody lub własności. Zaczniemy od rozwiązania, w którym klasa przechowuje swoją jedyną instancję w statycznym prywatnym polu i udostępnia ją tylko za pomocą własności tylko do odczytu (listing 4.43). Ta jedyna instancja powstaje natychmiast po uruchomieniu programu i jest utrzymywana aż do jego zakończenia. W odróżnieniu od klasy statycznej klasa ta może dziedziczyć z dowolnej klasy i implementować dowolną liczbę interfejsów. Możliwe jest także tworzenie wielu referencji, choć wszystkie wskazywać będą na jeden obiekt, i przekazywanie ich do metod. Dzięki użyciu modyfikatora sealed niemożliwe jest dziedziczenie¹⁵.

Listing 4.43. Singleton jako pole statyczne z kontrolą dostępu

```

public sealed class InstancjaWStatycznymPolu
{
    private static readonly InstancjaWStatycznymPolu instancja
        = new InstancjaWStatycznymPolu();

    private InstancjaWStatycznymPolu() { } //ukryty konstruktor

    public static InstancjaWStatycznymPolu Instancja
    {
        get
        {
            return instancja;
        }
    }
}

```

¹⁵ Zwróćmy uwagę, że ukrycie jedynego konstruktora (oznaczenie go jako prywatnego), niezależnie od modyfikatora sealed, uniemożliwia dziedziczenie po klasie. Konstruktor klasy potomnej, bez względu na to, czy zdefiniujemy go sami, czy stworzy go kompilator, musi wywołać konstruktor klasy bazowej. Jeżeli ten ostatni jest prywatny, a klasa potomna nie jest zagnieżdzona lub zaprzyjaźniona, to takie odwołanie jest niemożliwe.

```
private string pole = "Domyślna wartość pola";  
  
public string Własność  
{  
    get  
    {  
        return pole;  
    }  
    set  
    {  
        pole = value;  
    }  
}  
  
public void Metoda()  
{  
    Console.WriteLine(pole);  
}  
}
```

W tym przypadku możemy używać właściwości Instancja jako referencji do jedynego egzemplarza klasy, co umożliwia użycie tej klasy do stylu koniecznego w przypadku klasy statycznej:

```
InstancjaWStatycznymPolu.Instancja.Własność  
    = "Wartość pola ustalona w metodzie Program.Main";  
InstancjaWStatycznymPolu.Instancja.Metoda();
```

Można także utworzyć zmienną referencyjną i korzystając z niej, wywoływać metody i właściwości:

```
InstancjaWStatycznymPolu referencja = InstancjaWStatycznymPolu.Instancja;  
referencja.Własność = "Wartość pola ustalona w metodzie Program.Main";
```

Trzecie podejście (listing 4.44) różni się od poprzedniego tym, że instancja nie jest tworzona w momencie uruchomienia programu, a tworzona jest dopiero w chwili pierwszego odwołania do właściwości Instancja (typ *Lazy<>* omówiony w poprzednim rozdziale). Zaletami takiego podejścia, wprost wynikającymi z leniwej inicjalizacji, jest przede wszystkim unikanie zajmowania zasobów komputera w sytuacji, w której obiekt nie zostałby nigdy użyty. Tylko w tym podejściu do implementacji singletonu kod, który tworzy instancję, może być uzależniony od dodatkowych warunków. Dlatego to podejście można łatwo uogólnić do sytuacji, w której powstanie instancji jest uzależnione od dodatkowych warunków lub zależy nam na ograniczeniu instancji nie do pojedynczej, ale do określonej liczby instancji. Mimo tych różnic sposób użycia tej wersji klasy jest taki sam jak w drugim podejściu.

Listing 4.44. Singleton — kontrola liczby tworzonych instancji

```
public sealed class KontrolaLiczbyInstancji  
{  
    private static KontrolaLiczbyInstancji instancja;  
  
    private KontrolaLiczbyInstancji() { }  
  
    public static KontrolaLiczbyInstancji Instancja
```

```
{  
    get  
    {  
        if (instancja == null) instancja = new KontrolaLiczbyInstancji();  
        return instancja;  
    }  
}  
  
private string pole = "Domyślna wartość pola";  
  
public string Własność  
{  
    get  
    {  
        return pole;  
    }  
    set  
    {  
        pole = value;  
    }  
}  
  
public void Metoda()  
{  
    Console.WriteLine(pole);  
}
```

Interfejsy

W C# nie ma dziedziczenia wielokrotnego — klasy mogą dziedziczyć tylko po jednej klasie bazowej. Definiując struktury, w ogóle nie możemy wskazać klasę bazowej. W zamian C# oferuje możliwość implementacji interfejsów. Poznaliśmy je już, definiując strukturę Ułamek i klasę Para<> — w obu przypadkach definiowany przez nas typ implementował interfejs IComparable, który wymuszał na nas zdefiniowanie metody CompareTo. Teraz przyszedł czas, aby wiedzieć o interfejsach uporządkować i nieco poszerzyć.

Wróćmy do klasy OsobaZameldowana. Założymy, że bierzemy pod uwagę, że osoba zameldowana posiada telefon stacjonarny. Moglibyśmy wobec tego zdefiniować klasę potomną OsobaZameldowanaPosiadającąTelefon, która dziedziczyłaby z klasy OsobaZameldowana. Ale przecież nie tylko osoby mają telefony. Możemy zadzwonić także do urzędu lub restauracji. Czy należy stworzyć klasę bazową dla osób, urzędów, restauracji i innych miejsc, do których można zadzwonić? Nie, bo to niemożliwe. Klasa OsobaZameldowana już dziedziczy z klasy Osoba, w której nie można zakładać istnienia telefonu domowego, skoro nawet nie zakładamy zameldowania. Klasa OsobaZameldowana musiałaby wobec tego dziedziczyć z dwóch klas, co w C# nie jest możliwe. A może ten problem rozwiązać inaczej? W końcu telefon raczej się ma, niż jest się osobą „utelefonicznioną”. Może po prostu telefon powinien być elementem składowym. Być może tak, ale wówczas nie mielibyśmy żadnego wspólnego mianownika, który

pozwoliłby nam np. tworzyć kolekcje obiektów z dostępnym numerem telefonu. Rozwiążaniem tego problemu jest możliwość zdefiniowania interfejsu. Jeżeli zdefiniujemy interfejs `IPosiadaTelefonStacjonarny`¹⁶ (listing 4.45), to implementować go będzie mogła nie tylko klasa `OsobaZameldowana`, ale również klasy Urząd czy Restauracja. Co więcej, można w analogiczny sposób zdefiniować interfejs `IPosiadaAdresEmail` i również go implementować. Nie ma ograniczenia co do ilości implementowanych interfejsów.

Listing 4.45. Definicja interfejsu i jego implementacja

```
interface IPosiadaTelefonStacjonarny
{
    int? NumerTelefonu { get; set; }
}

class OsobaZameldowana : Osoba, IPosiadaTelefonStacjonarny
{
    public Adres AdresZameldowania;

    public override string ToString()
    {
        return base.ToString() + "; " + AdresZameldowania.ToString();
    }

    private int? numerTelefonu;

    public int? NumerTelefonu
    {
        get
        {
            return numerTelefonu;
        }
        set
        {
            numerTelefonu = value;
        }
    }

    public bool CzyPosiadaTelefonStacjonarny
    {
        get
        {
            return numerTelefonu.HasValue;
        }
    }
}
```

Czym zatem jest interfejs i czym różni się od klasy abstrakcyjnej? O abstrakcyjnej klasie bazowej należy myśleć jak o fundamencie, na którym budowana jest hierarchia

¹⁶ Zgodnie z konwencją przyjętą w platformie .NET nazwy interfejsów zaczynają się od dużej litery „I”, po której następuje nazwa własności, np. `IComparable`, co można by przetłumaczyć jako `IWspółmierny`. W nazwie interfejsu określającego posiadanie telefonu zastanawiałem się nad nazwą `IposiadającyTelefonStacjonarny`, co bardziej pasowałoby do konwencji, ale brzmi to trochę sztucznie i postanowiłem „posiadający” skrócić do „posiada”.

jej klas potomnych. Natomiast interfejs jest raczej czymś w rodzaju deklaracji, że implementujące ją klasy będą posiadały pewną funkcjonalność (metody lub własności). Interfejs nie realizuje zatem relacji „jest”, a raczej „potrafi”. Natomiast sposób użycia może być bardzo podobny jak w przypadku klas abstrakcyjnych i często wybór między jednym i drugim mechanizmem może być dość trudny. W praktyce warto zastanowić się, czy w opisie problemu jakaś grupa pojęć może być określona bardziej abstrakcyjnym pojęciem. Jeżeli tak, to temu pojęciu powinna odpowiadać klasa bazowa, może abstrakcyjna. Jeżeli natomiast jakąś grupę pojęć wiąże nie pojęcie (rzecznik), a raczej wspólna czynność lub umiejętność (czasowniki), to powinna to być dla nas wskazówka, że warto rozważyć zdefiniowanie interfejsu, który tę czynność opisuje.

Jak wspomniałem, sposób użycia interfejsów jest podobny do korzystania z klas bazowych. Możemy stworzyć kolekcję obiektów, które mogłyby posiadać telefon stacjonarny, i np. odczytać ich numery. Interfejsy mogą też być używane jako argumenty metod. Pokazuję to listing 4.46.

Listing 4.46. Użycie interfejsu jako argumentu metody i parametru kolekcji

```
static void WyświetlNumerTelefonu(IPosiadaTelefonStacjonarny telefon)
{
    Console.WriteLine("Numer telefonu: " + telefon.NumerTelefonu);
}

static void Main(string[] args)
{
    List<IPosiadaTelefonStacjonarny> listaTelefonów =
        new List<IPosiadaTelefonStacjonarny>();
    listaTelefonów.Add(new OsobaZameldowana() { NumerTelefonu = 123456789 });
    listaTelefonów.Add(new OsobaZameldowana() { NumerTelefonu = 987654321 });
    foreach (IPosiadaTelefonStacjonarny telefon in listaTelefonów)
        WyświetlNumerTelefonu(telefon);
}

static void WyświetlIlośćWierzchołków(Figura figura)
{
    Console.WriteLine("Liczba wierzchołków figury: " + figura.IleWierzchołków());
}
```

Rozdział 5.

Biblioteki DLL

Biblioteki DLL (ang. *dynamic link library*) są głównym sposobem dystrybucji klas i komponentów zarówno w aplikacjach dla platformy Win32, jak i dla platformy .NET. Oczywiście biblioteki obu platform różnią się w takim samym stopniu jak ich pliki wykonywalne .exe¹. Niemniej jednak idea stojąca za umieszczaniem modułów aplikacji w osobnych bibliotekach jest podobna.

Poniżej pokazuję, jak utworzyć bibliotekę DLL zawierającą klasy .NET. Użyjemy do tego przygotowanej w poprzednim rozdziale klasy `Ulamek`. Następnie wykorzystamy ją w aplikacji konsolowej. Ograniczę się przy tym wyłącznie do tzw. statycznego łączenia bibliotek². W efekcie klasa `Ulamek` nie zostanie wkompilowana do pliku .exe tej aplikacji — biblioteka będzie ładowana do pamięci w momencie uruchamiania aplikacji. Taki sposób postępowania ma kilka zalet. Jeżeli napiszemy lepszą wersję klasy `Ulamek`, ale taką, której interfejs nie ulegnie zmianie, to łatwo można zaktualizować cały program bez potrzeby ponownej komplikacji i dystrybucji pliku .exe. Wystarczy podmienić plik .dll. W większych aplikacjach może to bardzo ułatwić rozpowszechnianie uaktualnień. Ważne jest również to, że klasy umieszczone w bibliotece DLL mogą być wielokrotnie wykorzystywane przez różne aplikacje bez konieczności ich wkompilowywania w każdą z nich osobno. Dodatkową zaletą jest przenośność bibliotek PCL (ang. *Portable Class Library*). Biblioteki tego typu mogą być używane bez rekompilacji w różnych platformach zarządzanych Microsoftu. Jest to szczególnie interesujące w kontekście niedawno ogłoszonej współpracy Microsoftu i Xamarin, który rozszerzy listę platform obsługiwanych przez te biblioteki o Android i iOS (na bazie platformy Mono).

¹ Zarówno pliki .dll, jak i pliki .exe są podzespołami (ang. *assembly*), czyli skompilowanymi do kodu pośredniego MSIL samodzielnymi kawałkami kodu. Na poziomie podzespołów realizowana jest główna polityka bezpieczeństwa platformy .NET.

² Zagadnienia dynamicznego ładowania biblioteki już w trakcie działania aplikacji oraz związane z tym zagadnienia dynamicznego typowania (słowo kluczowe `dynamyc`) oraz korzystanie z platformy MEF do tworzenia bibliotek DLL funkcjonujących jako wtyczki omówione zostały w książce *Visual Studio 2010 dla programistów C#*, Helion, 2011 (obecnie dostępna w postaci ebooka). Umieszczone w tej książce informacje są nadal aktualne dla Visual Studio 2013.

Tworzenie zarządzanej biblioteki DLL

Jeżeli dysponujemy gotowymi klasami, które chcemy umieścić w zarządzanej bibliotece DLL, stworzenie takiej biblioteki zajmie krótką chwilę. Przećwiczmy to, umieszczając w bibliotece DLL klasę Ułamek z poprzedniego rozdziału.

1. Otwórz opisane w poprzednim rozdziale rozwiązanie *UlamekDemo* zawierające projekt *UlamekDemo* z klasą Ułamek.
2. Do projektu dodajmy projekt biblioteki. W tym celu:
 - a) Z menu *File* wybierz polecenie *New Project* lub wcisnij kombinację klawiszy *Ctrl+Shift+N*.
 - b) Zaznacz ikonę *Class Library*.
 - c) W polu *Name* wpisz nazwę *UlamekBiblioteka*.
 - d) Ponieważ chcemy nowy projekt dodać do istniejącego rozwiązania, należy w oknie *New Project* przełączyć rozwijaną listę *Solution* z *Create new solution* na *Add to solution*.
 - e) Kliknij *OK*.
3. Po utworzeniu projektu zobaczysz okno edytora kodu z definicją pustej klasy. Jednocześnie w podoknie *Solution Explorer* widoczny będzie nowy projekt w bieżącym rozwiązaniu, a w nim wyświetlany w edytorze plik *Class1.cs*. Usuńmy ten plik, zaznaczając go w podoknie *Solution Explorer* i naciskając klawisz *Delete*. Zostaniemy zapytani o potwierdzenie.
4. W jego miejsce skopiujmy plik *Ulamek.cs* z projektu *UlamekDemo*. Wystarczy go przeciągnąć w podoknie *Solution Explorer* i upuścić na pozycję *UlamekBiblioteka*.
5. W pliku *Ulamek.cs* musimy wprowadzić trzy zmiany (listing 5.1). Tylko pierwsza jest naprawdę ważna, dwie następne to zmiany czysto kosmetyczne.
 - a) Pierwszą jest dodanie do deklaracji klasy Ułamek modyfikatora *public*. Bez tego nie będzie ona widoczna poza biblioteką.
 - b) Drugą zmianą jest zmiana nazwy przestrzeni nazw. Wystarczy w kodzie zmienić nazwę za słowem kluczowym *namespace* z *UlamekDemo* na *Helion*.
 - c) Trzecia to usunięcie niepotrzebnych poleceń *using* w nagłówku pliku. Z zadeklarowanych tam przestrzeni nazw używamy tak naprawdę tylko przestrzeń *System*. Co ciekawe, nie musimy sami sprawdzać, które polecenia *using* są potrzebne, a które zbędne. Wystarczy, że z menu kontekstowego w edytorze wybierzemy pozycję *Organize Usings*, a z rozwiniętego w ten sposób podmenu — polecenie *Remove Unused Usings*.

Listing 5.1. Zmiany w pliku *Ulamek.cs*

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

```
using System.Text;
using System.Threading.Tasks;

namespace Helion
{
    public struct Ulamek : IComparable<Ulamek>
    {
        private int licznik, mianownik;
        ...
    }
}
```

6. Aby sprawdzić, czy wszystko jest w porządku, skompilujmy projekt biblioteki.

W tym celu z menu kontekstowego pozycji *UlamekBiblioteka* wybierzmy polecenie *Build*. Po chwili na pasku stanu Visual Studio powinniśmy zobaczyć komunikat *Build succeeded*.



Efektem komplikacji jest biblioteka *UlamekBiblioteka.dll*, którą znajdziemy w podkatalogu *UlamekDemo\UlamekBiblioteka\bin\Debug* (jeżeli komplujemy projekt w trybie *Debug*).

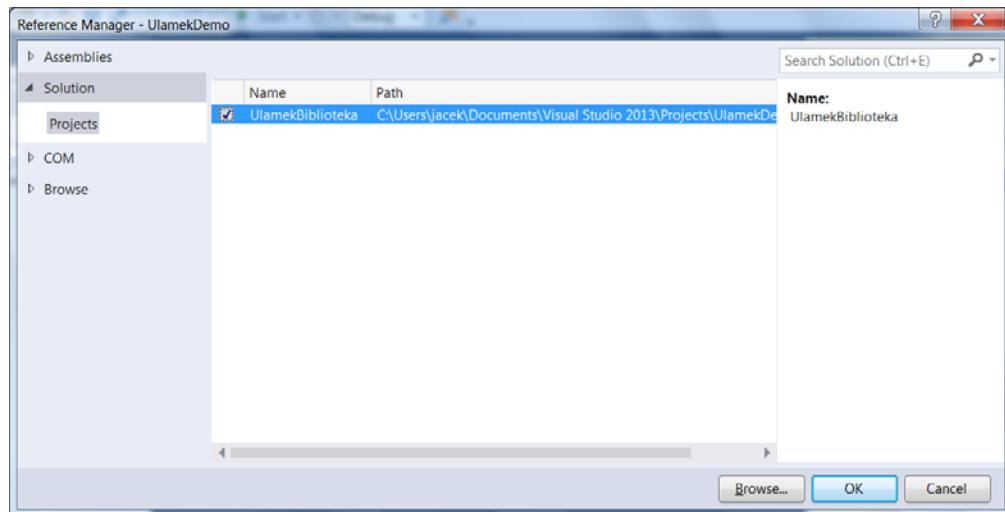
7. Jeżeli komplikacja się powiodła, możemy śmiało skasować plik *Ulamek.cs* z projektu *UlamekDemo*. Projekt nie będzie chciał się kompliować, ale niedługo to naprawimy.

Jeśli chcielibyśmy zachować konwencję nazewnictwa proponowaną w platformie .NET, nasza biblioteka powinna przejąć nazwę po przestrzeni nazw, tj. powinna nosić się *Helion.dll*. Skoro jednak nasza biblioteka zawiera tylko jedną klasę, uważam, że większy sens ma wyeksponowanie nazwy tej klasy.

Dodawanie do aplikacji referencji do biblioteki DLL

W tej chwili projekt *UlamekDemo* nie chce się kompliować, ponieważ nie widzi klasy *Ulamek*, do której odwohuje się w statycznej metodzie *Program.Main*. Musimy podpowiedzieć komplikatorowi, żeby klasy *Ulamek* szukały w bibliotece *UlamekBiblioteka.dll*.

1. W podoknie *Solution Explorer* z menu kontekstowego projektu *UlamekDemo* wybierz pozycję *Add, Reference...*.
2. Pojawi się okno *Reference Manager* — *UlamekDemo* (rysunek 5.1). W lewym panelu zaznaczmy pozycję *Solution*. Ograniczy to wyświetlane biblioteki do tych, które są zdefiniowane w bieżącym rozwiążaniu, a więc tylko do jednej.
3. Z lewej strony tej biblioteki na liście znajduje się słabo widoczne pole opcji, które należy zaznaczyć.
4. Następnie klikamy *OK*.



Rysunek 5.1. Dodawanie do projektu referencji do biblioteki DLL

5. Dodanie biblioteki jednak nie wystarczy. Znajduje się ona bowiem w innej przestrzeni nazw niż przestrzeń nazw projektu aplikacji (jeżeli zmieniliśmy nazwę przestrzeni nazw na `Helion`). Musimy wobec tego zadeklarować użycie tej przestrzeni nazw, dodając na początku pliku `Program.cs` instrukcję `using Helion;`.
6. Po tym oba projekty powinny się kompilować bez problemów. Klasa `Ulamek` jest dostępna i jej metody mogą być swobodnie wywoływane z metody `Program.Main`.

To najprostszy, a jednocześnie najczęstszy sposób tworzenia i użycia zarządzanych bibliotek DLL. Jest to tzw. statyczne łączenie bibliotek. Oprócz tego możliwe jest ich ładowanie dynamiczne już w trakcie działania programu. Wówczas konieczna jest czasem analiza zawartości biblioteki i rozpoznawanie umieszczonej w niej typów, weryfikowanie, czy biblioteka zawiera konkretną klasę, której chcemy użyć, i czy ta z kolei zawiera potrzebną nam metodę. Konieczne jest wówczas korzystanie z technologii odzwierciedlania typów (ang. *reflection*). Jest to odpowiednik RTTI z C++. Kolejnym zagadnieniem jest uruchomienie znalezionych metod. Wysiłek włożony w przygotowanie odpowiedniego kodu może się opłacić, np. w sytuacji, w której bibliotek DLL chcemy używać jako znajdywanych już w trakcie działania programu pluginów. Wówczas jednak lepiej użyć technologii MEF (zobacz książkę polecaną w przypisie nr 2).

Przenośne biblioteki DLL

W Visual Studio 2012 pojawiła się możliwość tworzenia bibliotek przenośnych PCL (ang. *Portable Class Library*), których można używać na kilku zarządzanych platformach oferowanych przez Microsoft. Pośród nich są: platforma .NET, WinRT, XNA,

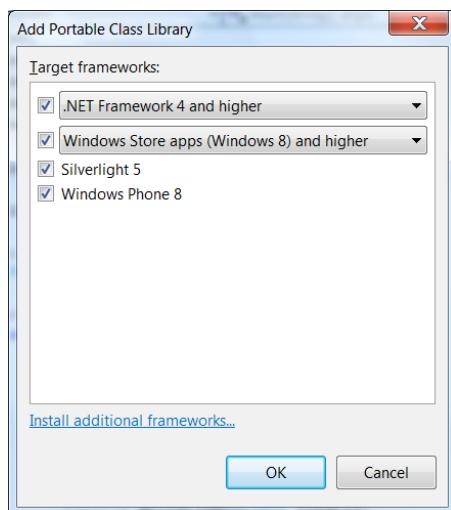
platforma instalowana w Xbox czy urządzeniach z Windows Phone. Ma to ogromną zaletę — raz przygotowany kod nie musi być modyfikowany w siostrzanych projektach dla różnego typu urządzeń. A tym samym ułatwia zarządzanie zmianami, które są zmorą projektów wieloplatformowych.

Tworzenie tego typu bibliotek nie różni się zbytnio od tworzenia zwykłej biblioteki klas:

1. W podoknie *Solution Explorer* zaznaczmy pozycję *UlamekDemo* odpowiadającą całemu rozwiązaniu (nie projektowi aplikacji).
2. Z jej menu kontekstowego wybierzmy *Add*, a następnie *New Project....*
3. Pojawi się znajome okno *Add New Project*. Zaznaczmy w nim pozycję *Portable Class Library*.
4. W polu *Name* wpisujemy *UlamekBibliotekaPrzenosna* i klikamy *OK*.
5. Zanim utworzony zostanie nowy projekt, zostaniemy spytani o to, z jakimi platformami nowa biblioteka ma być zgodna (rysunek 5.2). Ja obniżyłem tylko wymóg co do wersji platformy .NET do wersji 4. Dzięki temu uzyskaliśmy zgodność wsteczną z Visual Studio 2010.

Rysunek 5.2.

Wybór platform, z którymi nowa biblioteka może być zgodna



6. Następnie klikamy *OK*. Powstanie biblioteka PCL.

Dalsze postępowanie jest identyczne jak w przypadku zwykłej biblioteki, tj. kasujemy plik *Class1.cs*, a w zamian kopujemy do nowego projektu plik *Ulamek.cs*. Dołączanie referencji do bibliotek przenośnych PCL również odbywa się identycznie jak zwykłych bibliotek DLL. Jedyna różnica polega na tym, że tak utworzonej bibliotece PCL możemy użyć nie tylko w aplikacjach konsolowych, Windows Forms, WPF lub aplikacjach internetowych ASP.NET, ale również w aplikacjach dla Windows 8 na ekran Start (tzw. aplikacje Windows Store) lub w aplikacjach dla smartfonów z systemem Windows Phone.

**Wskazówka**

Oprócz platform, które widoczne są na rysunku 5.2, możliwe jest także zainstalowanie dodatkowych. Zobaczmy je, klikając łącze *Install additional frameworks...* widoczne w oknie *Add Portable Class Library*. Są to m.in. Windows Azure, Xbox czy Kinect for Windows.

Rozdział 6.

Testy jednostkowe

Testowanie oprogramowania, niezależnie od tego, czy traktowane jest jako osobny etap projektu, czy jako integralna część procesu wytwarzania kodu, jest tematem bardzo obszernym. Wystarczy wspomnieć rodzaje testów, jakie mogą i powinny być przeprowadzone: testy funkcjonalne aplikacji; testy integracyjne, stosowane gdy cały produkt składany jest z modułów często tworzonych przez różne osoby; testy systemowe, które mają ułatwić późniejsze bezproblemowe wdrożenia; wreszcie testy wydajnościowe; testy wykonywane wspólnie z klientem i testy interakcji z innymi systemami działającymi u klienta. Na pierwszej linii powinny jednak zawsze stać testy jednostkowe, które warto tworzyć zawsze, bez względu na charakter i rozmiar projektu, a które mają za zadanie pilnować, że nasz kod robi to, czego od nie oczekujemy. To ten rodzaj testów, z którym powinien być „zaprzyjaźniony” nie tylko wyspecjalizowany tester oprogramowania, ale również „zwykły” koder, programista i projektant. Są one po prostu, przynajmniej w części, gwarancją poprawności kodu. W tym rozdziale opiszę zagadnienia związane z tworzeniem testów jednostkowych i zarządzaniem nimi w najnowszej wersji Microsoft Visual Studio.



Wskazówka

Opisane w tym rozdziale narzędzie nie jest dostępne w wersji Express środowiska Visual Studio. Warto więc zainteresować się NUnit — narzędziem z rodziny xUnit przeznaczonym dla platformy .NET i języków zarządzanych.

Jeszcze przed napisaniem kodu, a najpóźniej w trakcie jego pisania powinniśmy przygotowywać sprawdzające go testy jednostkowe. W rozdziale 4., rozwijając strukturę Ułamek, ewidentnie nie zastosowałem się do tej zasady. W zamian testowałem przygotowywany kod w prostych testach umieszczonych wprost w metodzie Main, wyświetlających wyniki w konsoli. To wymagało śledzenia wydrukowanych wartości i samodzielnego ich weryfikowania. Teraz nadrobimy tę zaległość i przygotujemy wygodniejsze do obsługi i zarządzania testy jednostkowe.

Projekt testów jednostkowych

Wczytajmy do Visual Studio 2013 rozwiązańe *UlamekDemo* z poprzedniego rozdziału — to, w którym oprócz projektu aplikacji konsolowej *UlamekDemo* są również dwa projekty bibliotek DLL i PCL. Do tego rozwiązania dodamy kolejny projekt przeznaczony dla klas, w których zdefiniujemy metody testów. W tym celu:

1. W podoknie *Solution Explorer* rozwini menu kontekstowe i wybierz *Add, New Project...*.
2. Pojawi się okno *Add New Project*, w którego lewym panelu należy wybrać kategorię projektów: *Visual C#, Test*.
3. W środkowym panelu zaznacz pozycję *Unit Test Project*.
4. Ustal nazwę projektu na *UlamekTestyJednostkowe* i kliknij *OK*.

Powstanie nowy projekt, a do edytora zostanie wczytany plik *UnitTest1.cs*, automatycznie dodany do tego projektu. W pliku tym zdefiniowana jest przykładowa klasa *UnitTest1* z pustą metodą *TestMethod1*. Klasa ozdobiona jest atrybutem *TestClass*, a metoda — atrybutem *TestMethod*.

Przygotowania do tworzenia testów

W Visual Studio 2010, w menu kontekstowym edytora, dostępne było bardzo wygodne polecenie *Create Unit Test...* pozwalające na tworzenie testów jednostkowych dla wskazanej kursorem metody. W Visual Studio 2012 i 2013, w których zmodyfikowany został moduł odpowiedzialny za testy jednostkowe, to wygodne polecenie zniknęło. W zamian będziemy ręcznie przygotowywać metody testów. Zmuszeni jesteśmy także do samodzielnego przygotowania projektu testów, udostępniając mu testowaną klasę.

1. Aby przygotować projekt do testowania klasy *Ulamek* z biblioteki DLL *UlamekBiblioteka* lub biblioteki PCL *UlamekBibliotekaPrzenosna*, dodaj referencję jednej z nich do utworzonego przed chwilą projektu. W tym celu z menu kontekstowego projektu *UlamekTestyJednostkowe* wybierz *Add, Reference...* i postępuj analogicznie jak w poprzednim rozdziale.



Testy jednostkowe w Visual Studio wcale nie wymagają, aby testowany kod był zamknięty w bibliotece DLL lub PCL. W podany wyżej sposób można do projektu testów dodać również moduł programu .exe.

2. Na początku pliku *UnitTest1.cs* dodaj polecenie `using Helion;`, aby klasa `Helion.Ulamek` była łatwo dostępna.
3. Zmień nazwę pliku *UnitTest1.cs* na *UlamekTesty.cs*. Pojawi się pytanie o to, czy zmienić także nazwę klasy. Pozwól na to, klikając *Tak*.



Projekt testów jednostkowych może mieć wiele klas. Dla wygody i przejrzystości warto dbać o to, żeby każda testowana klasa miała osobną klasę z testami.

Pierwszy test jednostkowy

Przygotujemy teraz pierwszy test jednostkowy, który będzie sprawdzał działanie konstruktora klasy `Ulamek` i jej własności `Licznik` i `Mianownik`. Teoretycznie rzecz biorąc, w metodzie, która przeprowadza test, można wyróżnić trzy etapy: *przygotowanie* (ang. *arrange*), *działanie* (ang. *act*) i *weryfikacja* (ang. *assert*). Etapy te zaznaczone zostały w komentarzach widocznych na listingu 6.1. W praktyce granica między tymi etapami czasem się zaciera.

Listing 6.1. Metoda testująca nie może zwracać wartości ani pobierać parametrów, a dodatkowo musi być ozdobiona atrybutem `TestMethod`

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Helion;

namespace UlamekTestyJednostkowe
{
    [TestClass]
    public class UlamekTesty
    {
        [TestMethod]
        public void TestKonstruktoraIWlasnosci()
        {
            // przygotowania (assert)
            int licznik = 1;
            int mianownik = 2;

            // działanie (act)
            Ulamek u = new Ulamek(licznik, mianownik);

            // weryfikacja (assert)
            Assert.AreEqual(licznik, u.Licznik, "Niezgodność w liczniku");
            Assert.AreEqual(mianownik, u.Mianownik, "Niezgodność w mianowniku");
        }
    }
}
```

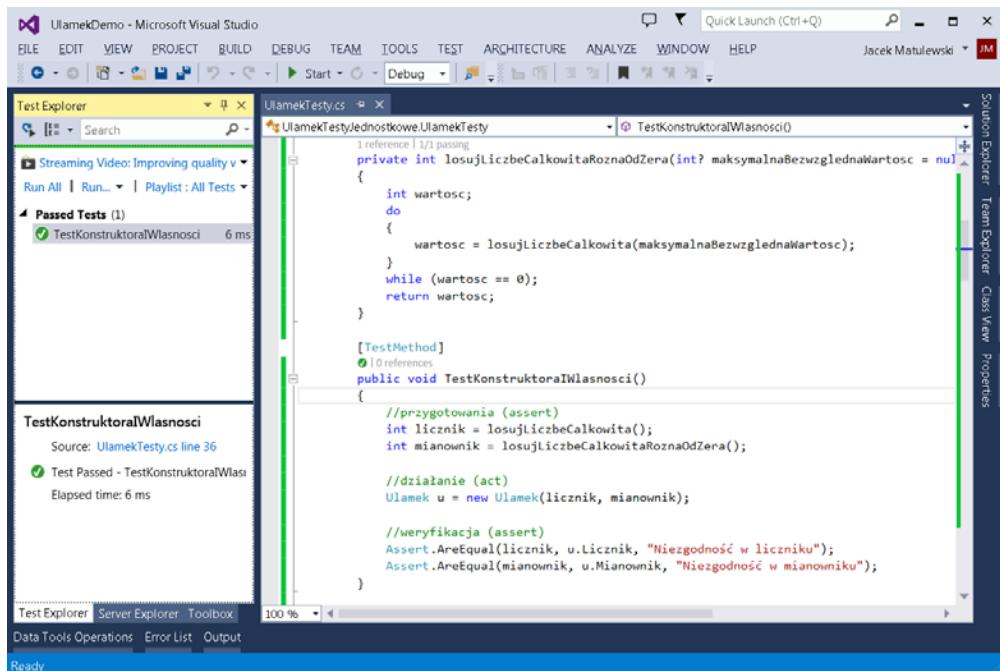
Zmieńmy nazwę metody `TestMethod1` na `TestKonstruktoraIWlasnosci` i umieścmy w niej kod widoczny na listingu 6.1. Pamiętajmy, że metoda testująca nie może zwracać wartości ani pobierać parametrów, a dodatkowo musi być ozdobiona atrybutem `TestMethod`.

Powyższy test należy do najczęściej używanego rodzaju testów, w którym weryfikacja polega na porównaniu jakiejś wartości otrzymanej w wyniku działania (drugi etap testu) z wartością oczekiwana. W tego typu testach należy użyć wywołania statycznej metody `Assert.AreEqual`. Decyduje ona o powodzeniu testu. Może być wywoływana wielokrotnie. Wówczas o zaliczeniu całego testu decyduje zaliczenie wszystkich wywołań tej metody. I odwrotnie — jedna niezgodność powoduje, że cały test uznany zostanie za „niezaliczony”.

W przypadku, gdy w metodzie testującej jest kilka poleceń weryfikujących, warto użyć możliwości podania komunikatu, który wyświetlany jest w oknie *Test Explorer* (o którym za chwilę) w razie niepowodzenia testu. Możliwość ta jest jednak opcjonalna i metoda `Assert.AreEqual` może być wywoływaną tylko z dwoma argumentami, czyli porównywanymi wartościami.

Uruchamianie testów

Sprawdźmy nasz test, komplikując małe rozwiązanie razem z projektem testów (*Ctrl+Shift+B* lub *F6*). Aby uruchomić test, wybierzmy z menu *Test* polecenie *Run, All Tests*. Pojawi się wówczas nowe podokno Visual Studio o nazwie *Test Explorer* (widoczne z lewej strony na rysunku 6.1). W podoknie tym widoczne są wszystkie uruchomione testy oraz efekt przeprowadzonej w nich weryfikacji. W Visual Studio 2013 ikona pokazująca efekt weryfikacji widoczna jest również w edytorze kodu nad sygnaturą metody testującej, obok liczby wywołań.



Rysunek 6.1. Podokno *Test Explorer* — nowość z Visual Studio 2012. W Visual Studio 2013 niewiele się w nim zmieniło

Dostęp do prywatnych pól testowanej klasy

Test konstruktora z listingu 6.1 ma zasadniczą wadę — testuje jednocześnie działanie konstruktora i właściwości `Licznik` i `Mianownik`. W razie niepowodzenia nie wiemy, który z tych elementów jest wadliwy. A co jeżeli zarówno w konstruktorze, jak i we właściwościach są błędy, które wzajemnie się kompensują? Warto byłoby wobec tego oprócz powyższego testu przygotować także test, w którym konstruktor sprawdzany jest przez bezpośrednią weryfikację zainicjowanych w nim wartości pól `licznik` i `mianownik`. Ale jak to zrobić, skoro są one prywatne? Pomocą służy klasa `PrivateObject` z przestrzeni nazw `Microsoft.VisualStudio.TestTools.UnitTesting`, tej samej, w której zdefiniowana jest klasa `Assert`. Przykład jej użycia pokazuje listing 6.2. W nim do odczytu wartości prywatnego pola używam metody `PrivateObject.GetField`.

Listing 6.2. Weryfikowanie wartości prywatnych pól

```
[TestMethod]
public void TestKonstruktora()
{
    // przygotowania (assert)
    int licznik = 1;
    int mianownik = 2;

    // działanie (act)
    Ulamek u = new Ulamek(licznik, mianownik);

    // weryfikacja (assert)
    PrivateObject po = new PrivateObject(u);
    int u_licznik = (int)po.GetField("licznik");
    int u_mianownik = (int)po.GetField("mianownik");
    Assert.AreEqual(licznik, u_licznik, "Niezgodność w liczniku");
    Assert.AreEqual(mianownik, u_mianownik, "Niezgodność w mianowniku");
}
```

Warto również zwrócić uwagę na metodę `PrivateObject.SetField` pozwalającą na zmianę prywatnego pola. Można jej użyć na przykład przy testowaniu właściwości w oderwaniu od konstruktora. Klasa `Private` posiada również metody `GetProperty` i `SetProperty` do testowania prywatnych właściwości oraz metodę `Invoke` do testowania prywatnych metod.

Dostęp do prywatnych elementów testowanej klasy w testach jednostkowych nie zawsze jest pożądany. W niektórych przypadkach testy jednostkowe powinny ograniczyć się do testowania klasy w takim zakresie, w jakim jest ona widoczna dla innych modułów projektu, nie wnikając w szczegóły jej implementacji (testy czarnej skrzynki). Natomiast na etapie tworzenia oprogramowania przydatne są wszystkie testy, które dają możliwość znalezienia błędu, także te, które zależą od szczegółów implementacji (testy białej skrzynki).

Testowanie wyjątków

Ułamek nie może mieć mianownika równego零. Próba użycia takiej wartości w konstruktorze powinna spowodować zgłoszenie wyjątku. To, czy rzeczywiście tak się stanie, możemy sprawdzić, jeżeli metodę testującą poprzedzimy atrybutem `ExpectedException` (listing 6.3). Jego parametrem jest oczekiwany typ wyjątku — w naszym przypadku jest to `ArgumentException` — taki wyjątek jest zgłaszany w konstruktorze klasy `Ułamek`. Dzięki temu atrybutowi test się powiedzie, jeżeli w metodzie testującej zgłoszony zostanie wyjątek typu `ArgumentException` lub potomny.

Listing 6.3. Prowokowanie wyjątku w metodzie testującej

```
[TestMethod]
[ExpectedException(typeof(Exception))]
public void TestKonstruktoraWyjatek()
{
    Ułamek u = new Ułamek(1, 0);
}
```

Przed metodą widoczny jest nowy atrybut `ExpectedException`. Jego parametrem jest oczekiwany typ wyjątku. Dzięki temu atrybutowi test się powiedzie, jeżeli w metodzie testującej zgłoszony zostanie wyjątek typu `Exception`.

Ponownie uruchommy testy, aby sprawdzić, czy nowy test działa prawidłowo. Możemy to zrobić z okna *Test Explorer*, klikając *Run All*. Testy uruchamiane są równolegle, co jest zarówno wygodne, jak i kłopotliwe. Wygodne, bo to w oczywisty sposób przyspiesza ich wykonanie, a dodatkowo pozwala sprawdzić ukryte zależności przy jednoczesnym dostępie do zasobów. Kłopotliwe, bo utrudnia separację testów w takich przypadkach. Weźmy chociażby często używaną w testach klasę `Random` — wbrew temu, co mogłoby się wydawać, nie jest ona wcale bezpieczna ze względu na wątki. Przy dużej liczbie równoległych wywołań metoda `Random.Next` może zwracać niepoprawne wyniki. W takiej sytuacji lepiej korzystać z generatorów liczb pseudolosowych tworzonych lokalnie w metodach testujących, a inicjowanych ziarnem pobranym z generatora zdefiniowanego jako pole klasy testującej.

Kolejne testy weryfikujące otrzymane wartości

Idąc za ciosem, przygotujmy kolejne testy (listing 6.4), w których sprawdzane są metody klasy `Ułamek` dla z góry ustalonych, przykładowych wartości licznika i mianownika.

Listing 6.4. Testowanie metod dla przykładowych wartości parametrów

```
[TestMethod]
public void TestPolaStatycznegoPolowa()
{
    Ułamek uP = Ułamek.Polowa;
    Assert.AreEqual(1, uP.Licznik);
```

```
        Assert.AreEqual(2, uP.Mianownik);
    }

    [TestMethod]
    public void TestMetodyUprosc()
    {
        Ulamek u = new Ulamek(4, -2);

        u.Uprosc();

        Assert.AreEqual(-2, u.Licznik);
        Assert.AreEqual(1, u.Mianownik);
    }

    [TestMethod]
    public void TestOperatorow()
    {
        Ulamek a = Ulamek.Polowa;
        Ulamek b = Ulamek.Cwierc;

        Assert.AreEqual(new Ulamek(3, 4), a + b, "Niepowodzenie przy dodawaniu");
        Assert.AreEqual(Ulamek.Cwierc, a - b, "Niepowodzenie przy odejmowaniu");
        Assert.AreEqual(new Ulamek(1, 8), a * b, "Niepowodzenie przy mnożeniu");
        Assert.AreEqual(new Ulamek(2), a / b, "Niepowodzenie przy dzieleniu");
    }
}
```

Test ze złożoną weryfikacją

Jak sprawdzić poprawność sortowania kolekcji z elementami typu `Ulamek`? Można oczywiście przygotować niewielką tabelę i jej posortowaną wersję, która będzie wartością oczekiwanaą używaną do weryfikacji. Warto jednak przetestować działanie sortowania dla różnych, losowo wybranych wartości. Wówczas na etapie weryfikacji zamiast prostego porównania wartości oczekiwanej z uzyskaną podczas testu należy wykonać bardziej złożone operacje sprawdzające, czy uzyskana w wyniku sortowania tabela jest rzeczywiście prawidłowo posortowana. Pokazuje to przykład widoczny na listingu 6.5, w którym sprawdzam, czy w tabeli każda kolejna wartość jest nie mniejsza niż poprzednia. Używam do tego zmiennej logicznej `tablicaJestPosortowanaRosnaco`, która inicjowana jest wartością `true`. Wartość ta zmieni się jedynie w przypadku wykrycia niemonotonicznej zmiany wartości w tabeli. O powodzeniu testu decyduje wywołanie metody `Assert.IsTrue`, która sprawdza, czy wartość zmiennej `tablicaJestPosortowanaRosnaco` pozostała równa `true`. Dodatkowo przygotujmy dwie metody pomocnicze `losujLiczbeCalkowita` i `losujLiczbeCalkowitaRoznaOdZera` (także widoczne na listingu 6.5), które będziemy wykorzystywali w tym i kolejnych testach. Korzystają one z generatora liczb losowych, który deklarujemy jako pole klasy `UlamekTesty`. Ani to pole, ani metody nie wymagają specjalnych atrybutów (pamiętajmy jednak, że klasa `Random` nie jest w pełni bezpieczna przy wspólniejącym uruchamianiu testów).

Listing 6.5. W tym przypadku etap weryfikacji nie jest prostym porównaniem wartości dwóch zmiennych

```

Random r = new Random();

private int losujLiczbeCalkowita(int? maksymalnaBezwzglednaWartosc = null)
{
    if (!maksymalnaBezwzglednaWartosc.HasValue)
        return r.Next(int.MinValue, int.MaxValue);
    else
    {
        maksymalnaBezwzglednaWartosc =
            Math.Abs(maksymalnaBezwzglednaWartosc.Value);
        return r.Next(-maksymalnaBezwzglednaWartosc.Value,
                      maksymalnaBezwzglednaWartosc.Value);
    }
}

private int losujLiczbeCalkowitaRoznaOdZera(int? maksymalnaBezwzglednaWartosc = null)
{
    int wartosc;
    do
    {
        wartosc = losujLiczbeCalkowita(maksymalnaBezwzglednaWartosc);
    }
    while (wartosc == 0);
    return wartosc;
}

[TestMethod]
public void TestSortowania()
{
    Ulamek[] tablica = new Ulamek[100];
    for (int i = 0; i < tablica.Length; i++)
        tablica[i] = new Ulamek(losujLiczbeCalkowita(),
                               losujLiczbeCalkowitaRoznaOdZera());

    Array.Sort(tablica);

    bool tablicaJestPosortowanaRosnaco = true;
    for (int i = 0; i < tablica.Length - 1; i++)
        if (tablica[i] >= tablica[i + 1]) tablicaJestPosortowanaRosnaco = false;
    Assert.IsTrue(tablicaJestPosortowanaRosnaco);
}

```

**Wskazówka**

Dodatkowe pola wspomagające testy mogą wymagać specjalnej inicjacji lub innego typu przygotowania. Można do tego użyć dodatkowych metod opatrzonych specjalnymi atrybutami. I tak metoda z atrybutem `ClassInitialize` będzie uruchamiana na początku wszystkich testów, gdy tworzona jest instancja klasy testującej. W tej metodzie można np. zainicjować generator liczb pseudolosowych. Natomiast metoda z atrybutem `TestInitialize` będzie uruchamiana przed pojedynczym testem. Tym atrybutom odpowiadają atrybuty `ClassCleanup` i `TestCleanup`. Opatrzone nimi metody uruchamiane są odpowiednio po wykonaniu wszystkich testów i po pojedynczym teście; służą do usunięcia zbędnych już obiektów lub przywrócenia pierwotnego stanu obiektów pomocniczych.

Powtarzane wielokrotnie testy losowe

Choć testowanie działania metod lub operatorów dla wybranych wartości jest potrzebne i użyteczne, to konieczne jest również przeprowadzenie testów dla większego zakresu wartości parametrów, szczególnie w końcowym etapie prac nad klasą. Trudno jednak spodziewać się, że przygotujemy pętlę iterującą po wszystkich możliwych wartościach licznika i mianownika. Zajęłoby to wieki. Możemy się ratować, testując metody klasy `Ulamek` dla wartości losowych. Żeby to miało jednak sens, należy takich testów wykonać wiele. Na tyle dużo, żeby losowe wartości pokryły cały zakres obu pól klasy `Ulamek`. Przykłady takich testów dla operatorów konwersji widoczne są na listingu 6.6.

Listing 6.6. Testy zawierające elementy losowe mogą być powtarzane w jednej metodzie

```
const int liczbaPowtorzen = 100;

[TestMethod]
public void TestKonwersjiDoDouble()
{
    for (int i = 0; i < liczbaPowtorzen; ++i)
    {
        int licznik = losujLiczbeCalkowita();
        int mianownik = losujLiczbeCalkowitaRoznaOdZera();
        Ulamek u = new Ulamek(licznik, mianownik);

        double d = (double)u;

        Assert.AreEqual(licznik / (double)mianownik, d);
    }
}

[TestMethod]
public void TestKonwersjiZInt()
{
    for (int i = 0; i < liczbaPowtorzen; ++i)
    {
        int licznik = losujLiczbeCalkowita();

        Ulamek u = licznik;

        Assert.AreEqual(licznik, u.Licznik);
        Assert.AreEqual(1, u.Mianownik);
    }
}
```

Wielokrotne powtarzanie testów, i co za tym idzie wielokrotne wywoływanie metod `Assert.AreEqual` lub `Assert.IsTrue`, nie naraża nas na zafałszowanie wyniku całego testu. Jak pamiętamy, do „zaliczenia” testu niezbędne jest, żeby wszystkie wywołania tych metod potwierdziły poprawność kodu. Z kolei do uzyskania negatywnego wyniku wystarczy, że niezgodność pojawi się choćby w jednym z nich.

Niepowodzenie testu

Według optymistów celem testów jednostkowych jest potwierdzenie poprawności kodu. Według realistów jest nim znalezienie ukrytych w nim błędów logicznych. W praktyce oba cele wymagają przeprowadzania jak największej liczby różnorodnych testów, nawet jeżeli wydaje się nam, że wszystkie błędy już znaleźliśmy¹. Jednych i drugich zachęcam do wykonania testu z listingu 6.7. W teście tym tworzone są dwa równe sobie obiekty `Ulamek` o losowych wartościach licznika i mianownika. Następnie jeden z nich jest upraszczany metodą `Uprosc`, a następnie porównywane są wartości obu ułamków po ich konwersji do liczby rzeczywistej typu `double`. Sprawdzamy zatem, czy uproszczenie ułamka nie spowodowało zmiany jego rzeczywistej wartości. Pamiętajmy jednak, że w przypadku losowo wybieranych wartości uproszczenie będzie możliwe dość rzadko — wymaga to licznika i mianownika, które mogą być podzielone przez tę samą liczbę większą od jedności. Zatem stosunkowo niewielka część iteracji będzie miała rzeczywisty wkład do testu.

Listing 6.7. Hasło programisty: „Test zakończony niepowodzeniem jest szansą na poprawienie testowanego kodu”

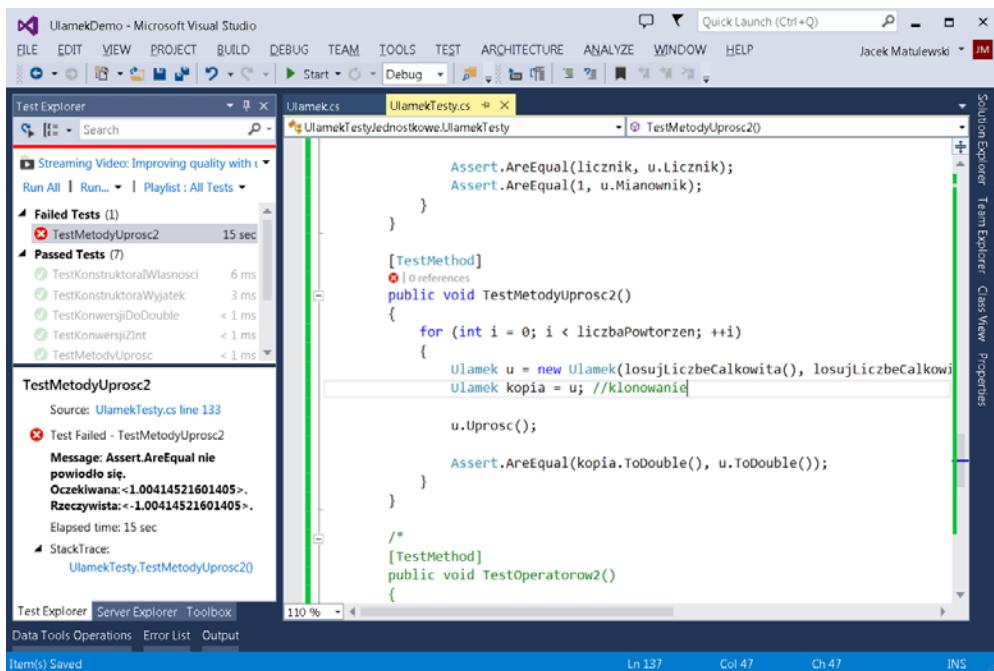
```
[TestMethod]
public void TestMetodyUprosc_Losowy()
{
    for (int i = 0; i < liczbaPowtorzen; ++i)
    {
        Ulamek u = new Ulamek(losujLiczbeCalkowita(),
                               losujLiczbeCalkowitaRoznaOdZera());
        Ulamek kopia = u; // klonowanie

        u.Uprosc();

        Assert.IsTrue(u.Mianownik > 0);
        Assert.AreEqual(kopia.ToDouble(), u.ToDouble());
    }
}
```

Uruchamiając ów test, przekonamy się, że klasa `Ulamek` go nie „zaliczy” (rysunek 6.2). Obok testu pojawi się czerwona ikona, a zaznaczając ów test na liście testów w podoknie *Test Explorer*, możemy obejrzeć szczegóły informacji o niepowodzeniu. Okazuje się, że po konwersji ułamków do liczb typu `double` mają one różne znaki, choć są równe co do bezwzględnej wartości. Możemy powtórzyć ten test wielokrotnie (co ma sens przy losowo wybieranych wartościach), jednak wynik jest zawsze taki sam.

¹ Takie przekonanie jest naturalne u programisty, który tworzy kod. Stąd częsty brak wystarczającego zaangażowania w proces testowania i pokusa, żeby ten etap projektu „odbębnić” jak najszybciej. I dlatego dobrze jest, jeżeli testowaniem nie zajmuje się programista, który przygotowuje kod, a ktoś inny, najlepiej osoba wyspecjalizowana w przeprowadzaniu testów. Z drugiej strony często można usłyszeć opinię zupełnie odwrotną, której też trudno odmówić słuszności: według niej każdy programista jest odpowiedzialny za własny kod, dlatego powinien sam pisać dla siebie testy jednostkowe, które pozwolą mu pilnować jego poprawności. Oba poglądy mogą być jednocześnie słuszne, bo dotyczą innych płaszczyzn. Pierwszy mówi o tym, jak jest (psychologia pracy), a drugi o tym, jak być powinno (etyka programisty). W praktyce trzeba oba w jakimś zakresie połączyć, w czym pomagają nowoczesne metodyki wytwarzania oprogramowania i pracy w zespołach.



Rysunek 6.2. Szczegóły testu można obejrzeć po jego kliknięciu w podoknie Test Explorer

W takiej sytuacji nie od razu wiadomo, czy niepoprawny jest sprawdzany kod, czy sprawdzający go test. To drugie zdarza się zresztą nader często. Oszczędzę Czytelniowi wspólnego szukania błędów i od razu wyjaśnię, na czym polega problem. Błąd tkwi w testowanej metodzie `Ulamek.Uprosc`, a dokładniej w warunku sprawdzającym znaki licznika i mianownika (fragment wyróżniony na listingu 6.8). Znaki tych pól są różne, jeżeli ich iloczyn jest mniejszy od zera. Wówczas znak licznika ustalamy na minus, a mianownika na plus. Problem w tym, że dla dużych wartości licznika i mianownika iloczyn łatwo może przekroczyć zakres liczb całkowitych typu `int`, co prowadzi do zafałszowania wyników.

Listing 6.8. Metoda `Uprosc` z wyróżnionym problematycznym fragmentem kodu

```

public void Uprosc()
{
    //NWD
    int mniejsza = Math.Min(Math.Abs(licznik), Math.Abs(mianownik));
    for (int i = mniejsza; i > 0; i--)
        if ((licznik % i == 0) && (mianownik % i == 0))
    {
        licznik /= i;
        mianownik /= i;
    }

    //znaki
    if (licznik * mianownik < 0)
    {
        licznik = -Math.Abs(licznik);
        mianownik = Math.Abs(mianownik);
    }
}

```

```

    }
    else
    {
        licznik = Math.Abs(licznik);
        mianownik = Math.Abs(mianownik);
    }
}

```

Jak można poprawić ten warunek, aby działał prawidłowo dla dowolnych wartości licznika i mianownika? Najprostsze co możemy zrobić, to rozłożyć ów warunek na warunki elementarne:

```

if ((licznik < 0 && mianownik > 0) || (licznik >= 0 && mianownik < 0))
{
    ...
}
```

Można także pozostać przy iloczynie, ale zmniejszyć wartości czynników:

```

if (Math.Sign(licznik) * Math.Sign(mianownik) < 0)
{
```

Oba rozwiązania są poprawne i dzięki nim poprawiona metoda `Ułamek.Uprosc`, „zda test”. Test ten będzie teraz trwał dłużej. Wcześniej test przerywała pierwsza znaleziona niezgodność, teraz wykonywany jest pełny zestaw powtórzeń.

Nieuniknione błędy

O ile w przypadku metody upraszczającej ułamek przekroczenie zakresu należy traktować jak błęd, to w przypadku operatorów arytmetycznych jest to nie do uniknięcia. Wówczas to nie działanie tych operatorów, ale ich użycie dla zbyt dużych wartości argumentów może być uważane za błędne. Można się oczywiście zastanawiać, czy operatory nie powinny zwracać typu o większym zakresie lub czy nie powinny zgłaszać wyjątków, gdy zakres zostanie przekroczyony. Pierwszego rozwiązania się nie praktykuje, bo prowadzi do kolejnych trudności. W końcu iloczyn dwóch liczb całkowitych typu `int` jest wartością typu `int` nawet, gdyby miało to prowadzić do przekroczenia zakresu tego typu². Podobnie jest w przypadku typu `Ułamek`. Kłopotliwy jest również drugi sposób, tj. zgłaszanie wyjątku `OverflowException` w przypadku przekroczenia zakresu. Jest to nawet proste do realizacji — wystarczy umieścić mnożenia liczb typu całkowitego, jakie znajdują się w definicjach operatorów arytmetycznych, w kontekście słowa kluczowego `checked`. Wówczas wyjątki takie będą zgłaszcane. Dramatycznie wpłynie to jednak na wydajność tych operatorów. Idealnie byłoby, gdyby użytkownik klasy `Ułamek` mógł sam decydować o tym, czy przekroczenie zakresu ma być sprawdzane, otaczając operacje na ułamkach konstrukcją `checked()`. To nie jest jednak

² Należy pamiętać, że przy domyślnych ustawieniach przekroczenie zakresu nie jest nawet sygnalizowane za pomocą wyjątku. Aby to uzyskać, należy użyć słowa kluczowego `checked` (rozdział 2.) lub zmienić ustawienia całego projektu.

możliwe do zrealizowania³. Jedyne, co może zrobić użytkownik, to włączyć kontrolę zakresu globalnie dla całego projektu (przynajmniej na czas projektowania). Najbardziej eleganckim sposobem wydaje mi się wobec tego zdefiniowanie dodatkowych metod klasy `Ulamek`, które będą realizowały operacje arytmetyczne z kontrolą zakresu (listing 6.9), lub przełącznika, który decydowałby o tym, czy taką kontrolę stosować w przypadku operatorów.

Listing 6.9. Dodawanie z kontrolą zakresu

```
public static Ulamek Dodaj0vf(Ulamek u1, Ulamek u2)
{
    Ulamek wynik =
        new Ulamek(checked(u1.Licznik * u2.Mianownik + u2.Licznik * u1.Mianownik),
                   checked(u1.Mianownik * u2.Mianownik));
    wynik.Uprosc();
    return wynik;
}
```

Jak widać, testowanie operatorów arytmetycznych w takiej postaci, w jakiej są one obecnie zdefiniowane, dla losowo wybieranych wartości licznika i mianownika z całego zakresu typu `int` nie ma sensu — z pewnością skończyłoby się to przekroczeniem zakresu, a tym samym negatywnym wynikiem testu. Zakres testowanych wartości należy wobec tego ograniczyć. Ograniczenie na możliwe wartości liczników i mianowników sumowanych, odejmowanych, mnożonych i dzielonych ułamków wynika wprost z definicji ich operatorów. We wszystkich pojawiają się iloczyny liczników i mianowników (listing 4.10). Zatem pierwszym ograniczeniem jest to, że nie mogą one być jednocześnie większe od pierwiastka kwadratowej maksymalnej wartości typu `int` (czyli `Math.Sqrt(int.MaxValue)`). Bardziej restrykcyjne ograniczenie dotyczy tylko operatorów dodawania i odejmowania i związane jest z obliczanymi w nich licznikami. Dla przykładu w operatorze dodawania licznik jest równy sumie dwóch iloczynów liczników i mianowników. Wynika z tego, że liczniki i mianowniki dodawanych ułamków nie mogą być jednocześnie większe od pierwiastka z połowy maksymalnej wartości typu `int` (czyli `Math.Sqrt(int.MaxValue/2)`). I właśnie do takich liczb ograniczyłem się w teście widocznym na listingu 6.10. Można oczywiście pójść nieco dalej. W końcu jeżeli wylosowany licznik pierwszego składnika nie jest duży, to większe mogą być pozostałe trzy wartości. Można zatem maksymalną wartość mianownika pierwszego ułamka uzależnić od wylosowanej wcześniej wartości licznika. Z kolei wartości te wpływają na dopuszczalne wartości licznika i mianownika drugiego ułamka. Metoda testująca stałaby się jednak dość zawiła, co zmniejszałoby zaufanie do jej poprawności (a nie chcemy przecież testować testów).

Listing 6.10. Uwzględnienie bezpiecznego zakresu

```
[TestMethod]
public void TestOperatorow_Losowy()
{
```

³ Powodem jest sposób implementacji słowa kluczowego `checked`. W języku pośrednim CIL dostępne są osobne instrukcje dla operacji arytmetycznych dla liczb całkowitych i dla analogicznych operacji, ale z kontrolą zakresu. Użycie słowa kluczowego `checked` jest sygnałem dla kompilatora, że ma użyć tych drugich. W języku pośrednim nie ma oczywiście takich instrukcji dla typu `Ulamek`.

```
// ograniczenie maksymalnej wartości
int limit = (int)(Math.Sqrt(int.MaxValue / 2) - 1);

for (int i = 0; i < liczbaPowtorzen; ++i)
{
    Ulamek a = new Ulamek(losujLiczbeCalkowita(limit),
                           losujLiczbeCalkowitaRoznaOdZera(limit));
    Ulamek b = new Ulamek(losujLiczbeCalkowita(limit),
                           losujLiczbeCalkowitaRoznaOdZera(limit));

    double suma = (a + b).ToDouble();
    double roznica = (a - b).ToDouble();
    double iloczyn = (a * b).ToDouble();
    double iloraz = (a / b).ToDouble();

    Assert.AreEqual(a.ToDouble() + b.ToDouble(), suma,
                    "Niepowodzenie przy dodawaniu");
    Assert.AreEqual(a.ToDouble() - b.ToDouble(), roznica,
                    "Niepowodzenie przy odejmowaniu");
    Assert.AreEqual(a.ToDouble() * b.ToDouble(), iloczyn,
                    "Niepowodzenie przy mnożeniu");
    Assert.AreEqual(a.ToDouble() / b.ToDouble(), iloraz,
                    "Niepowodzenie przy dzieleniu");
}
}
```

Przekraczanie zakresu to jednak nie jedyny problem, jaki pojawi się przy okazji tego testu. Przekonajmy się o tym, uruchamiając go. Okaże się, że uzyskamy wynik negatywny. Tym razem problemem nie jest jednak testowana klasa, a sam test. Porównujemy wynik działania operatorów skonwertowany do liczb rzeczywistych typu `double` i wyniki odpowiadających im operatorów przeciążonych dla typu `double`. Należy wziąć pod uwagę, że oba typy, `Ulamek` i `double`, mają ograniczoną dokładność. Ograniczona jest bowiem najmniejsza możliwa do zapisania w nich wartość absolutna. W przypadku typu `Ulamek` jest to `1/int.MaxValue`, czyli `4.656612875245797e-10`. W przypadku typu `double` wartość tę można odczytać z właściwości `double.Epsilon`, która jest równa `4.94066E-324`. Należy się zatem spodziewać, że wyniki działania operatorów dla typów `Ulamek` i `double` będą równe z dokładnością nie większą niż 10^{-10} (lub jak kto woli `0.0000000001` lub `1E-10`). Na szczęście metoda `Assert.AreEqual` pozwala na uwzględnienie dokładności (listing 6.11). To wreszcie pozwoli na prawidłowe przeprowadzenie tego testu.

Listing 6.11. Skorygowana metoda testująca

```
[TestMethod]
public void TestOperatorow_Losowy()
{
    // ograniczenie maksymalnej wartości
    int limit = (int)(Math.Sqrt(int.MaxValue / 2) - 1);
    // dopuszczalna różnica w wyniku
    const double dokładosc = 1E-10;

    for (int i = 0; i < liczbaPowtorzen; ++i)
    {
        Ulamek a = new Ulamek(losujLiczbeCalkowita(limit),
                               losujLiczbeCalkowitaRoznaOdZera(limit));
```

```
Ulamek b = new Ulamek(losujLiczbeCalkowita(limit),
                      losujLiczbeCalkowitaRoznaOdZera(limit));

double suma = (a + b).ToDouble();
double roznica = (a - b).ToDouble();
double iloczyn = (a * b).ToDouble();
double iloraz = (a / b).ToDouble();

Assert.AreEqual(a.ToDouble() + b.ToDouble(), suma, dokładosc,
                "Niepowodzenie przy dodawaniu");
Assert.AreEqual(a.ToDouble() - b.ToDouble(), roznica, dokładosc,
                "Niepowodzenie przy odejmowaniu");
Assert.AreEqual(a.ToDouble() * b.ToDouble(), iloczyn, dokładosc,
                "Niepowodzenie przy mnożeniu");
Assert.AreEqual(a.ToDouble() / b.ToDouble(), iloraz, dokładosc,
                "Niepowodzenie przy dzieleniu");
}
```

* * *

Pisanie testów jest trudnym, a jednocześnie często niedocenianym elementem projektów informatycznych. Wymaga wyobraźni, ogromnej skrupulatności i odpowiedzialności. I z reguły jest bardziej pracochłonne niż samo pisanie kodu. Zauważmy, że przedstawiony w tym rozdziale kod testów jest większy niż kod testowanej klasy, a to przecież tylko reprezentacja testów, jakie należy przygotować. Nie wszystkie scenariusze użycia klasy `Ulamek` zostały już sprawdzone.

Na koniec warto wspomnieć o częstej sytuacji, w której testowana klasa zależy od jakichś zewnętrznych obiektów lub danych, np. odczytywanych z baz danych lub urządzeń fizycznych. Obiekty te lub dane mogą być trudne do kontroli, choćby dlatego że odczytywane z nich wartości zależą od jeszcze innych czynników, nie są deterministyczne, zależą od decyzji użytkownika programu lub po prostu nie są jeszcze przetwarzane i tym samym godne zaufania. Wówczas wygodne może być zastąpienie tych zewnętrznych kłopotliwych obiektów, od których zależy testowana klasa, przez tzw. obiekty zastępcze, bardziej obrazowo nazywane *zaślepkami* (ang. *mock objects*), których stan i dynamikę możemy w pełni kontrolować. Szczególnie wygodne jest to w sytuacji, w której chcemy odtworzyć błąd występujący dla pewnego trudnego do odtworzenia w rzeczywistym obiekcie zewnętrznym stanu. Wówczas obiekty zastępcze są wręcz nieocenione. Pozwalają one również uniknąć sytuacji, w których nie jest jasne, co jest testowane i co jest źródłem ewentualnego błędu — testowana klasa czy obiekt zewnętrzny.

Rozdział 7.

Elementy programowania współbieżnego

Do platformy .NET w wersji 4.0 dodana została biblioteka TPL (ang. *Task Parallel Library*) oraz kolekcje umożliwiające pracę w różnych scenariuszach pojawiających się przy programowaniu współbieżnym. Całość popularnie nazywana jest *Parallel Extensions*. TPL nadbudowuje klasyczne wątki i pulę wątków, korzystając z nowej klasy *Task* (z ang. *zadanie*).

W szczegółach biblioteka TPL, mechanizmy synchronizacji „kolekcje równoległe” oraz narzędzia pozwalające na debugowanie i analizowanie programów równoległych opisane zostały w naszej książce *Programowanie równoległe i asynchroniczne w C# 5.0* wydanej przez wydawnictwo Helion w grudniu 2013 roku. Tu chciałbym skupić się jedynie na najczęściej używanym jej elemencie — współbieżnej pętli *for* — oraz na nowości C# 5.0, ogniskujących się na nowych słowach kluczowych *async* i *await*.

Równoległa pętla *for*

Załóżmy, że mamy zbiór stu liczb rzeczywistych, dla których musimy wykonać jakieś stosunkowo czasochłonne obliczenia. W naszym przykładzie będzie to obliczenie wartości funkcji $y = f(x) = \arcsin(\sin(x))$. Funkcja ta powinna z dokładnością numeryczną zwrócić wartość argumentu. I zrobi to, jednak nieźle się przy tym namęczy — funkcje trygonometryczne są bowiem wymagające numerycznie. Powtórzmy te obliczenia kilkukrotnie, aby dodatkowo przedłużyć czas obliczeń. Listing 7.1 prezentuje kod pliku *Program.cs* z aplikacji konsolowej, w której zaimplementowany został powyższy pomysł.

Listing 7.1. Metoda zajmująca procesor

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

```
namespace ProgramowanieRównolegle
{
    class Program
    {
        static private double obliczenia(double argument)
        {
            for (int i = 0; i < 10; ++i)
            {
                argument = Math.Asin(Math.Sin(argument));
            }
            return argument;
        }

        static void Main(string[] args)
        {
            // przygotowania
            int rozmiar = 10000;
            Random r = new Random();
            double[] tablica = new double[rozmiar];
            for (int i = 0; i < tablica.Length; ++i) tablica[i] = r.NextDouble();

            // obliczenia sekwencyjne
            int liczbaPowtorzen = 100;
            double[] wyniki = new double[tablica.Length];
            int start = System.Environment.TickCount;
            for (int powtorzenia = 0; powtorzenia < liczbaPowtorzen; ++powtorzenia)
                for (int i = 0; i < tablica.Length; ++i)
                    wyniki[i] = obliczenia(tablica[i]);
            int stop = System.Environment.TickCount;
            Console.WriteLine("Obliczenia sekwencyjne trwały "
                +(stop - start).ToString() + " ms.");

            /*
            // prezentacja wyników
            Console.WriteLine("Wyniki:");
            for (long i = 0; i < tablica.Length; ++i)
                Console.WriteLine(i + ". " + tablica[i] + " ?= " + wyniki[i]);
            */
        }
    }
}
```

Na tym samym listingu, w metodzie `Main`, widoczna jest pętla wykonująca obliczenia wraz z przygotowaniem tablicy. Wyniki nie są drukowane — tablica jest zbyt duża, żeby to miało sens (poza tym drukowanie w konsoli jest bardzo wolne). Poniższy kod zawiera dwie zagnieżdżone pętle `for`. Interesuje nas tylko ta wewnętrzna. Zewnętrzna jedynie powtarza obliczenia, co po prostu zwiększa wiarygodność pomiaru czasu, który realizujemy na bazie zliczania taktów procesora. Pętlę wewnętrzną można bez większego wysiłku zrównoleglić dzięki klasie `Parallel` z przestrzeni nazw `System.Threading.Tasks`. W Visual Studio 2013 przestrzeń ta jest uwzględniona w domyślnie zadeklarowanych przestrzeniach w sekcji `using` na początku pliku. Zrównolegiona wersja tej pętli widoczna jest na listingu 7.2. Dodajmy ją do metody `Main`.

Listing 7.2. Przykład zrównoległonej pętli for

```
// obliczenia równolegle
start = System.Environment.TickCount;
for (int powtorzenia = 0; powtorzenia < liczbaPowtorzen; ++powtorzenia)
{
    Parallel.For(0, tablica.Length, (int i) => wyniki[i] = obliczenia(tablica[i]));
}
stop = System.Environment.TickCount;
Console.WriteLine("Obliczenia równoległe trwały " + (stop - start).ToString() + " ms.");
```

Jak wspomniałem, zrównoleglamy tylko wewnętrzną pętlę. Użyta do tego metoda `Parallel.For` jest dość intuicyjna w użyciu. Jej dwa pierwsze argumenty określają zakres zmiany indeksu pętli. W naszym przypadku jest on równy [0,10000]. Wobec tego do metody podanej w trzecim argumencie przekazywane są liczby od 0 do 9999. Trzeci argument jest natomiast delegacją, do której można przypisać metodę (lub jak w naszym przypadku wyrażenie lambda) realizującą polecenia w każdej iteracji pętli. Powinna się tam zatem znaleźć zawartość oryginalnej pętli. Argumentem wyrażenia lambda jest bieżący indeks pętli.

Proces zrównoleglania kodu może być tak prosty, jak pokazałem powyżej, ale są to rzadkie przypadki. Często w ogóle nie jest on możliwy (tak jest w przypadku pętli `for` w tabeli 3.5 obliczającej silnię podanej liczby, w której kolejne iteracje zależą od poprzednich — muszą być wobec tego wykonywane sekwencyjnie) albo wymaga synchronizacji. Metoda `Parallel.For` automatycznie synchronizuje zadania po zakończeniu wszystkich iteracji, dlatego nie ma zagrożenia zamazania danych w ramach kolejnych jej powtórzeń. Jeżeli jednak zajdzie taka potrzeba, synchronizację można zrealizować za pomocą operatora `lock`, tego samego, którego używa się w przypadku wątków.

Warto przestrzec, że nie należy się spodziewać, że dzięki użyciu równoległej pętli `for` nasze obliczenia przyspieszą tyle razy, ile rdzeni procesora mamy do dyspozycji. Tworzenie i usuwanie zadań również zajmuje nieco czasu. Eksperymentując z rozmiarem tablicy i liczbą obliczanych sinusów, można sprawdzić, że zrównoleglanie opłaca się tym bardziej, im dłuższe są obliczenia wykonywane w ramach jednego zadania. Dla krótkich zadań użycie równoległej pętli może wręcz wydłużyć całkowity czas obliczeń. Na komputerze z jednym procesorem dwurdzeniowym uzyskane przez mnie uśrednione przyspieszenie dla powyższych parametrów było równe 66,4%. Z kolei w przypadku ośmiu rdzeni czas obliczeń równoległych spadł do zaledwie 34,8%.

Przerywanie pętli

Podobnie jak w klasycznej pętli `for`, również w przypadku wersji równoległej możemy w każdej chwili przerwać jej działanie. Służy do tego klasa `ParallelLoopState`. Aby to umożliwić, należy dodać drugi argument do metody lub wyrażenia lambda wykonywanego w każdej iteracji — właśnie obiekt typu `ParallelLoopState`. Udostępnia on dwie metody: `Break` i `Stop`. Różnią się one tym, że `Break` pozwala na wcześniejsze zakończenie bieżącej iteracji bez uruchamiania następnych, a `Stop` nie tylko natychmiast kończy bieżący wątek, ale również podnosi flagę `IsStopped`, która powinna być sprawdzona we wszystkich uruchomionych wcześniej iteracjach i która jest sygnałem do ich

zakończenia (to leży jednak w gestii programisty przygotowującego kod wykonywany w każdej iteracji). Listing 7.3 pokazuje przykład, w którym pętla przerywana jest, jeżeli wylosowana zostanie liczba 0.

Listing 7.3. Przerywanie pętli równoległej

```
static void Main(string[] args)
{
    Random r = new Random();
    long suma = 0;
    long licznik = 0;
    string s = "";

    // iteracje zostaną wykonane tylko dla liczb parzystych
    // pętla zostanie przerwana wcześniej, jeżeli wylosowana liczba jest równa 0
    Parallel.For(0, 10000, (int i, ParallelLoopState stanPetli) =>
    {
        int liczba = r.Next(7); // losowanie liczby oczek na kostce
        if (liczba == 0)
        {
            s += "Stop:";  
            stanPetli.Stop();
        }
        if (stanPetli.IsStopped) return;
        if (liczba % 2 == 0)
        {
            s += liczba.ToString() + "; ";
            obliczenia(liczba);
            suma += liczba;
            licznik++;
        }
        else s += "[" + liczba.ToString() + "]"; ";
    });
}

Console.WriteLine(
    "Wylosowane liczby: " + s +
    "\nLiczba pasujących liczb: " + licznik +
    "\nSuma: " + suma +
    "\nŚrednia: " + (suma / (double)licznik).ToString());
}
```

Programowanie asynchroniczne. Modyfikator `async` i operator `await` (nowość języka C# 5.0)

Programowanie asynchroniczne to niezwykle istotny element aplikacji Windows Store uruchamianych na nowym ekranie Start Windows 8. Jednak również w „zwykłych” aplikacjach mechanizm ten może być z powodzeniem używany. Pojawia się chociażby w Entity Framework 6.0 (rozdział 17.), choć w tym przypadku jest wyraźnie mniej gotowych metod korzystających z tej techniki. Tak czy inaczej można go bez problemu

użyć także we własnych rozwiązańach. Dlatego w dalszej części rozdziału postaram się opisać mechanizm asynchroniczności właśnie w taki sposób, w jaki jego poznanie jest potrzebne, aby używać go we własnych projektach. Muszę jednak uprzedzić, że w kolejnych rozdziałach nie znalazłem żadnego pretekstu, żeby tego mechanizmu użyć.

Jak wspomniałem, język C# 5.0 wyposażony został w nowy operator `await`, ułatwiający synchronizację dodatkowych uruchomionych przez użytkownika zadań. Poniżej zaprezentuję prosty przykład jego użycia, który chyba najlepiej wyjaśni jego działanie. Nie omówię wprawdzie zadań (mam na myśli bibliotekę TPL i jej sztandarową klasę `Task`), jednak podobnie jak w przypadku opisanej wyżej pętli równoległej `Parallel`. ↵
For, tak i w przypadku operatora `await` dogłębia znajomość biblioteki TPL nie jest do tego konieczna. Spójrzmy na przykład widoczny na listingu 7.4. Przedstawia on metodę `Main`, która definiuje przykładową czynność, zapisuje referencję do niej w zmiennej akcja i wykonuje ją synchronicznie. Czynność ta wprowadza półsekundowe opóźnienie metodą `Thread.Sleep`, które oczywiście opóźnia jej zakończenie i wydrukowanie informującego o tym komunikatu.

Listing 7.4. Synchroniczne wykonywanie kodu zawartego w akcji

```
static void Main(string[] args)
{
    // czynność
    Func<object, long> akcja =
        (object argument) =>
    {
        Console.WriteLine("Początek działania akcji - " + argument.ToString());
        System.Threading.Thread.Sleep(500); // opóźnienie 0.5s
        Console.WriteLine("Koniec działania akcji - " + argument.ToString());
        return DateTime.Now.Ticks;
    };

    long wynik = akcja("synchronicznie");
    Console.WriteLine("Synchronicznie: " + wynik.ToString());
}
```

W listingu 7.5 ta sama akcja wykonywana jest asynchronicznie w osobnym wątku utworzonym na potrzeby zdefiniowanego przez nas zadania. Synchronizacja następuje w momencie odczytania wartości zwracanej przez czynność, tj. w momencie odczytania właściwości `Result`. Jej sekcja `get` czeka ze zwroceniem wartości aż do zakończenia zadania i tym samym wstrzymuje wątek, w którym wykonywana jest metoda `Main`. Jest to zatem punkt synchronizacji. Zwróćmy uwagę, że po instrukcji `zadanie.Start`, a przed odczytaniem właściwości `Result` mogą być wykonywane dowolne czynności, o ile są niezależne od wartości zwróconej przez zadanie.

Listing 7.5. Użycie zadania do asynchronicznego wykonania kodu

```
static void Main(string[] args)
{
    // czynność
    Func<object, long> akcja =
        (object argument) =>
```

```

Console.WriteLine("Początek działania akcji - " + argument.ToString());
System.Threading.Thread.Sleep(500); //opóźnienie 0.5s
Console.WriteLine("Koniec działania akcji - " + argument.ToString());
return DateTime.Now.Ticks;
};

long wynik = akcja("synchronicznie");
Console.WriteLine("Synchronicznie: " + wynik.ToString());

//w osobnym zadaniu
Task<long> zadanie = new Task<long>(akcja, "zadanie");
zadanie.Start();
Console.WriteLine("Akcja została uruchomiona");
//właściwość Result czeka ze zwroceniem wartości, aż zadanie zostanie zakończone
//(synchronizacja)
long wynik = zadanie.Result;
Console.WriteLine("Zadanie: " + wynik.ToString());
}

```

Ponadto nie jest konieczne, aby instrukcja odczytania właściwości `Result` znajdowała się w tej samej metodzie co uruchomienie zadania — należy tylko do miejsca jej odczytania przekazać referencję do zadania (w naszym przypadku zmienną typu `Task<long>`). Zwykle referencję tę przekazuje się jako wartość zwracaną przez metodę uruchamiającą zadanie. Zgodne z konwencją metody tworzące i uruchamiające zadania powinny zawierać w nazwie przyrostek `.Async` (listing 7.6).

Listing 7.6. Wzór metody wykonującej jakąś czynność asynchronous

```

Task<long> ZróbCośAsync(object argument)
{
    //czynność, która będzie wykonywana asynchronous
    Func<object, long> akcja =
        (object _argument) =>
    {
        Console.WriteLine("Początek działania akcji - " + _argument.ToString());
        System.Threading.Thread.Sleep(500); //opóźnienie 0.5s
        Console.WriteLine("Koniec działania akcji - " + _argument.ToString());
        return DateTime.Now.Ticks;
    };

    Task<long> zadanie = new Task<long>(akcja, argument);
    zadanie.Start();
    return zadanie;
}

static void Main(string[] args)
{
    Task<long> zadanie2 = ZróbCośAsync("zadanie-metoda");
    Console.WriteLine("Akcja została uruchomiona (metoda)");
    long wynik = zadanie2.Result;
    Console.WriteLine("Zadanie-metoda: " + wynik.ToString());
}

```

Wraz z wersjami 4.0 i 4.5 w platformie .NET (oraz w platformie Windows Runtime) pojawiło się wiele metod, które wykonują długotrwałe czynności asynchronicznie. Znajdziemy je w klasie `HttpClient`, w klasach odpowiedzialnych za obsługę plików (`StorageFile`, `StreamWriter`, `StreamReader`, `XmlReader`), w klasach odpowiedzialnych za kodowanie i dekodowanie obrazów czy też w klasach WCF. Asynchroniczność jest wręcz standardem w aplikacjach Windows 8 z interfejsem Modern UI (aplikacje Windows Store). I właśnie aby ich użycie było (prawie) tak proste jak metod synchronicznych, wprowadzony został w C# 5.0 (co odpowiada platformie .NET 4.5) operator `await`. Ułatwia on synchronizację dodatkowego zadania tworzonego przez te metody. Należy jednak pamiętać, że metodę, w której chcemy użyć operatora `await`, musimy oznaczyć modyfikatorem `async`. A ponieważ modyfikatora takiego nie można dodać do metody wejściowej `Main`, stworzyłem dodatkową, wywoływaną z niej metodę `ProgramowanieAsynchroniczne`. Prezentuje to listing 7.7.

Listing 7.7. Przykład użycia modyfikatora `async` i modyfikatora `await`

```
static async void ProgramowanieAsynchroniczne()
{
    Task<long> zadanie2 = ZróbCośAsync("zadanie-metoda");
    Console.WriteLine("Akcja została uruchomiona (metoda)");
    long wynik = await zadanie2;
    Console.WriteLine("Zadanie-metoda: " + wynik.ToString());
}

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

    Console.Write("Naciśnij Enter..."); Console.ReadLine();
}
```

Operator `await` zwraca parametr typu `Task<>` (czyli `long` w przypadku `Task<long>`) lub `void`, jeżeli użyta została wersja nieparametryczna klasy `Task`.

Metody oznaczone modyfikatorem `async` nazywane są w angielskiej dokumentacji MSDN *async methods*. Może to jednak wprowadzać pewne zamieszanie. Metody z modyfikatorem `async` (w naszym przypadku metoda `ProgramowanieAsynchroniczne`) mylone są bowiem z metodami wykonującymi asynchronicznie jakieś czynności (w naszym przypadku `ZróbCośAsync`). Osobom poznającym dopiero temat często wydaje się, że aby metoda wykonywana była asynchronicznie, wystarczy dodać do jej sygnatury modyfikator `async`. To nie jest prawda.

Mogliśmy oczywiście wywołać metodę `ZróbCośAsync` w taki sposób, że umieścimy ją bezpośrednio za operatorem `await`, np. `long wynik = await ZróbCośAsync("async/await");`. Ale czy to ma sens? Wykonywanie metody `ProgramowanieAsynchroniczne`, w której znajduje się to wywołanie, zostanie wstrzymane aż do momentu zakończenia metody `ZróbCośAsync`, więc efekt, jaki zobaczymy na ekranie, będzie identyczny jak w przypadku synchronicznym (listing 7.5). Różnica jest jednak zasadnicza, ponieważ instrukcja zawierająca operator `await` nie blokuje wątku, w którym wywołana została metoda

ProgramowanieAsynchroniczne. Kompilator zawiesza jej wywołanie, przechodząc do kolejnych czynności aż do momentu zakończenia uruchomionego zadania. W momencie gdy to nastąpi, wątek wraca do metody ProgramowanieAsynchroniczne i kontynuuje jej działanie.

„Zawieszenie działania metody ProgramowaniaAsynchroniczne” to obrazowe, ale mało precyzyjne sformułowanie. Tak naprawdę kompilator tnie tę metodę w miejscu wystąpienia operatora await i tę część, która ma być wykonana po zakończeniu zadania i odebraniu wyniku, umieszcza w metodzie zwrotnej (ang. *callback*). Co więcej, ta metoda zwrotna wcale nie będzie wykonywana w tym samym wątku co pierwsza część metody ProgramowanieAsynchroniczne i metoda Main; wykorzystany zostanie wątek, w którym pracowało zadanie tworzone przez metodę ZróbCośAsync.

Z tego wynika, że efekt działania operatora await zobaczymy dopiero, gdy metodę ProgramowanieAsynchroniczne wywołamy z innej metody, w której będą dodatkowe instrukcje wykonywane w czasie wstrzymania metody ProgramowanieAsynchroniczne. W naszym przykładzie wywołujemy ją z metody Main, która po wywołaniu metody ProgramowanieAsynchroniczne wstrzymuje działanie aż do naciśnięcia klawisza *Enter*. W serii instrukcji wywołanie metody oznaczonej modyfikatorem *async* nie musi się zakończyć przed wykonaniem następnej instrukcji — w tym sensie jest ona asynchroniczna. Aby tak się stało, musi w niej jednak zadziałać operator await, w naszym przykładzie czekający na wykonanie metody ZróbCośAsync. W efekcie, jeżeli w metodzie Main usuniemy ostatnie polecenia wymuszające oczekiwanie na naciśnięcie klawisza *Enter*, metoda ta zakończy się przed zakończeniem metody ProgramowanieAsynchroniczne, kończąc tym samym działanie całego programu i nie pozwalając metodzie ZróbCośAsync wykonać całego zadania. Jeżeli polecenia te są obecne, instrukcje z metody ZróbCośAsync zostaną wykonane już po wyświetleniu komunikatu o konieczności naciśnięcia klawisza *Enter* wyświetlonego przez metodę Main. Dowodzi tego rysunek 7.1 (zobacz diagram przy listingu 7.8).

The screenshot shows two separate instances of the Windows Command Prompt (cmd.exe) running side-by-side. Both windows have the title bar 'C:\Windows\system32\cmd.exe'.

Top Window (Task 1):

```
Akcja została uruchomiona (metoda)
Początek działania akcji - zadanie-metoda
Koniec.
Naciśnij Enter... Koniec działania akcji - zadanie-metoda
Zadanie-metoda: 635198493763740000
Aby kontynuować, naciśnij dowolny klawisz . . .
```

Bottom Window (Task 2):

```
Akcja została uruchomiona (metoda)
Początek działania akcji - zadanie-metoda
Koniec.
Aby kontynuować, naciśnij dowolny klawisz . . . -
```

Rysunek 7.1. Zadania utworzone w ten sposób nie blokują wątku głównego — działają jak wątki tła

Diagram na listingu 7.8 obrazuje przebieg programu. Liczby w okręgach oznaczają kolejne ważne punkty programu, m.in. te, w których następuje przepływ wątku z metody do metody (dwa okręgi z tą samą liczbą) lub uruchamiane jest zadanie. Lewa kolumna odpowiada wątkowi głównemu (w nim wykonywana jest metoda Main), a prawa — wątkowi tworzonemu na potrzeby zadania, a potem wykorzystywanemu do dokończenia metody zawierającej operator await. Warto nad tym diagramem spędzić chwilę czasu z ołówkiem w ręku i postarać się zrozumieć, jak działa operator await i które fragmenty kodu czekają na zakończenie zadania, a które nie.

Listing 7.8. Diagram przebiegu programu



* * *

Zgodnie z zapowiedzią to tylko najbardziej podstawowe wiadomości o podstawowych elementach TPL i programowaniu asynchronicznym w C# 5.0. Więcej informacji na ten temat, w szczególności o bardzo ważnym zagadnieniu synchronizacji wątków, znajdziesz Czytelnik we wspomnianej już książce *Programowanie równolegle i asynchroniczne w C# 5.0* (Helion, 2013).

Część II

Projektowanie aplikacji Windows Forms

Rozdział 8.

Pierwszy projekt aplikacji Windows Forms

Obecnie aplikacje z graficznym interfejsem użytkownika (ang. *graphical user interface* — GUI) przeznaczone na pulpit systemu Windows¹ można tworzyć w Visual Studio, korzystając z dwóch bibliotek kontrolek: tradycyjnej Windows Forms oraz znacznie nowszej Windows Presentation Foundation (WPF). Postanowiłem przedstawić tylko tę pierwszą. Dlaczego? Głównym powodem jest jej prostota i elegancja, choć przy mniejszych możliwościach i elastyczności. Dzięki temu Czytelnik, który jak przypuszczam, nie jest zaawansowanym programistą, a wręcz przeciwnie — dopiero rozpoczyna przygodę z platformą .NET i językiem C#, nie zniechęci się do projektowania aplikacji Windows. Poza tym oba projekty są już zakończone i nie będą znacząco rozwijane. To stawia je w jednym rzędzie. Przy tym projektów korzystających z WPF jest znacznie mniej, choć nowych powstaje zapewne więcej.

Interfejs aplikacji Windows Forms można w Visual Studio zaprojektować w pełni „wizualnie”, składając go przy użyciu myszy z kontrolek umieszczanych na podglądzie okna i modyfikując ich własności udostępnione w podoknie *Properties*. Po utworzeniu w ten sposób interfejsu, co jest nawet przyjemnym zadaniem, porównywalnym z budowaniem z klocków Lego, programista powinien przejść do kolejnego etapu, w którym określone są reakcje aplikacji na różnego rodzaju akcje jej użytkownika, a więc powinien np. zdefiniować funkcję wykonywaną po kliknięciu myszą przycisku umieszczonego na oknie, po przesunięciu suwaka lub wybraniu pozycji w menu. Definiując owe funkcje, a raczej metody, bo jak już wiemy, w C# wszystkie funkcje muszą być składowymi klas, musimy wykorzystać umiejętności przedstawione w pierwszej części tej książki. Ta część książki będzie jednak poświęcona przede wszystkim narzędziom programowania wizualnego i wsparciu, jakie przy budowaniu interfejsu daje nam środowisko Visual Studio.

¹ Osobnym, nieomawianym w tej książce zagadnieniem jest projektowanie aplikacji na ekran *Start*, czyli tzw. aplikacji *Windows Store* lub aplikacji z interfejsem *Modern*. O projektowaniu tego typu aplikacji więcej znajdziesz się w powstającej dopiero książce *Projektowanie aplikacji Windows Store w Visual Studio 2013*.

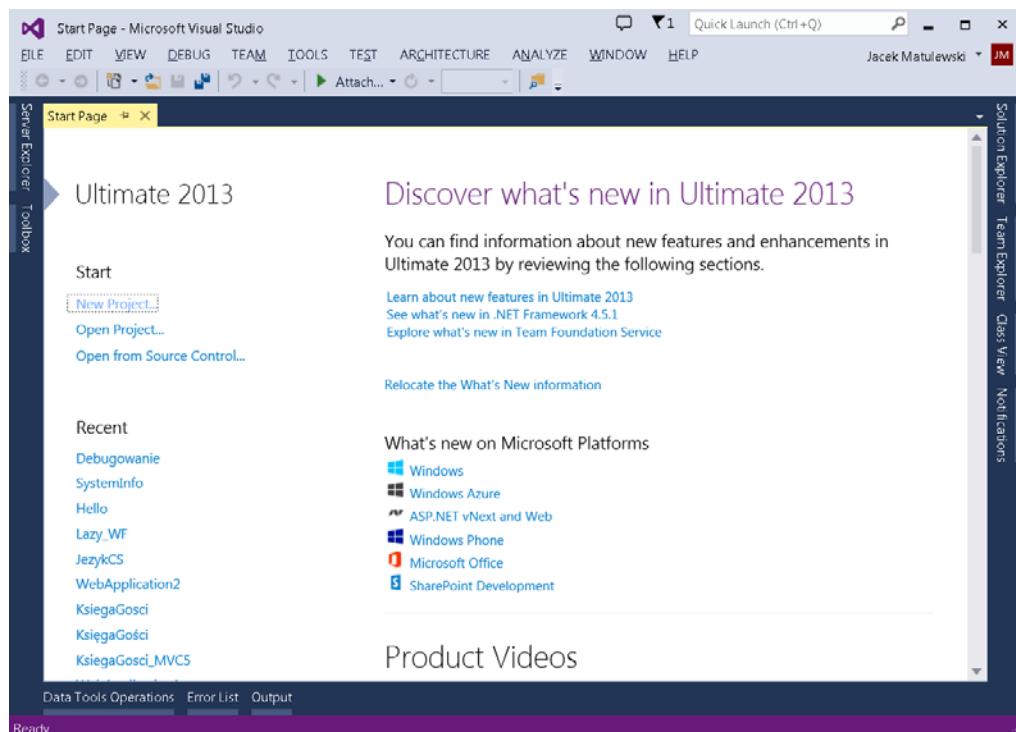
Projektowanie interfejsu aplikacji

Pierwsza aplikacja Windows Forms, którą przygotujemy, będzie umożliwiać użytkownikowi zmianę koloru panelu za pomocą trzech suwaków. Jej działanie ma z założenia być dość proste — kod wprowadzony ręcznie będzie ograniczał się do jednego polecenia. Ważniejsze jest to, że podczas tworzenia aplikacji poznasz charakterystyczne elementy środowiska Visual Studio 2013 i programowania wizualnego w tym środowisku.

Tworzenie projektu

Utwórzmy projekt aplikacji okienkowej (typu Windows Forms Application):

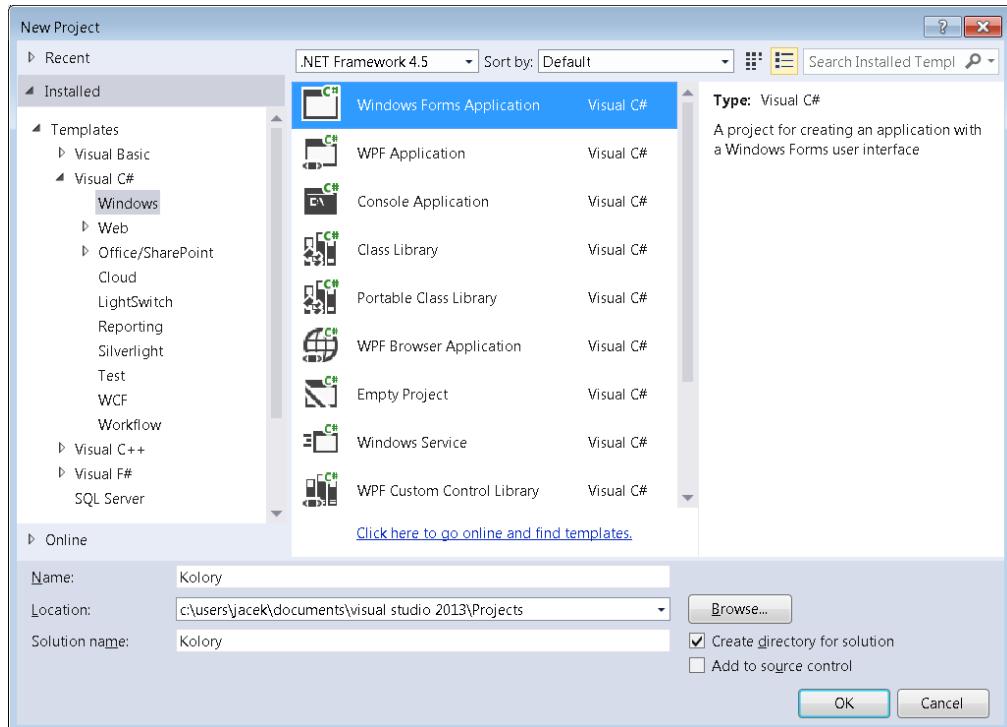
1. W menu *File* środowiska Visual Studio wybierz podmenu *New*, a następnie *Project....*
2. Alternatywnie możesz ze strony startowej (strona z etykietą *Start Page*, rysunek 8.1) wybrać link *New Project....*.



Rysunek 8.1. Strona startowa Visual Studio z widoczną pozycją *New Project...*

3. W oknie *New Project* (rysunek 8.2):

a) w panelu *Installed Templates* zaznacz pozycję *Visual C#*;



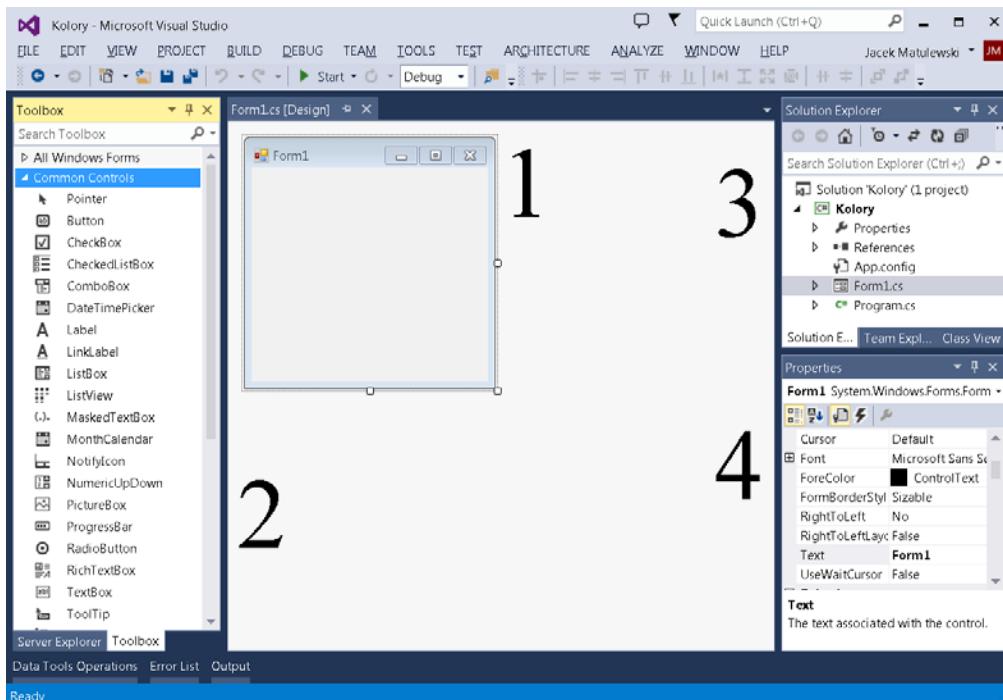
Rysunek 8.2. Zbiór projektów C# dostępnych w Visual Studio 2010

- b) następnie w głównej części okna zaznacz pozycję *Windows Forms Application*;
- c) w polu edycyjnym *Name* podaj nazwę aplikacji *Kolory*;
- d) kliknij *OK*.

Utworzyliśmy projekt o nazwie *Kolory*, którego pliki zostały umieszczone w katalogu *Moje dokumenty\Visual Studio 2013\Projects\Kolory\Kolory*. Projekt otoczony jest ponadto rozwiązaniem (ang. *solution*), którego plik konfiguracyjny znajduje się w katalogu nadrzednym, tj. *Moje dokumenty\Visual Studio 2013\Projects\Kolory\Kolory.sln*. Plik projektu nazywa się *Kolory.csproj*. Twarzyszy mu sporo dodatkowych plików zawierających kod C#, w tym *Program.cs* z metodą *Main*, będącą głównym punktem wejściowym (ang. *entry point*) programu — identycznie jak w przypadku aplikacji konsolowych. To ta metoda tworzy obiekt formy (okna)² i uruchamia pętle główną. Inne pliki to m.in. *Form1.cs* i *Form1.Designer.cs* zawierające definicję klasy opisującej okno aplikacji. Ponadto w katalogu *Properties* można znaleźć kilka plików pomocniczych z kodem źródłowym, których przeznaczenie zostanie opisane niżej.

Po wykonaniu powyższych poleceń wygląd środowiska Visual C# zmieni się (rysunek 8.3). W głównym oknie dodana zostanie nowa zakładka o nazwie *Form1.cs [Design]*, a na niej zobaczymy obraz pustej jeszcze formy (wskazuje ją numer 1 na rysunku 8.3).

² W kontekście programowania dla Windows z historycznych powodów, których nie warto tu już nawet wyjaśniać, okna aplikacji nazywa się zazwyczaj formami (ang. *form*).



Rysunek 8.3. Visual C# jest gotowy do projektowania aplikacji

Z lewej strony znajduje się zminimalizowane podokno zawierające zbiór kontrolek (*Toolbox*), oznaczone na rysunku numerem 2. Z kolei z prawej strony widoczne jest podokno zawierające listę wszystkich plików rozwiązań i projektu (numer 3). Bardzo pomocne jest także okno *Properties* (numer 4), które najłatwiej otworzyć, naciskając klawisz *F4*. Do tego okna będę się w tej książce odwoływał wiele razy. Jeżeli klikniemy podgląd formy, okno własności będzie pokazywać jej własności (klasa *Form1*).



Wskazówka Możemy wybrać dwa sposoby ułożenia własności i zdarzeń w oknie własności: alfabetyczny (ikona *Alphabetic* w pasku narzędzi tego okna) oraz według kategorii (ikona *Categorized*).

Dokowanie palety komponentów Toolbox

Zadokujmy podokno *Toolbox* tak, aby było stale widoczne w widoku projektowania. W tym celu:

1. Kliknij przycisk *Toolbox* znajdujący się na pasku przy lewej krawędzi okna (oznaczony numerem 2 na rysunku 8.3).
2. Następnie kliknij ikonę w kształcie pinezki, aby „przypiąć” rozwinięte okno na stałe i zapobiec jego ukrywaniu się.
3. Dopasuj szerokość podokna tak, żeby widoczne były całe opisy wszystkich kontrolek.

Podokno *Toolbox* jest bardzo ważne. W nim zgromadzone są kontrolki, czyli „klocki”, z których złożymy interfejs aplikacji. Znajdziemy tam m.in. tak typowe kontrolki jak napisy, przyciski, pola edycyjne, pola opcji, różnego typu listy, a także komponenty bazodanowe i typowe okna dialogowe. Wszystkie te komponenty pogrupowane są w kilku rozwijanych zakładkach.

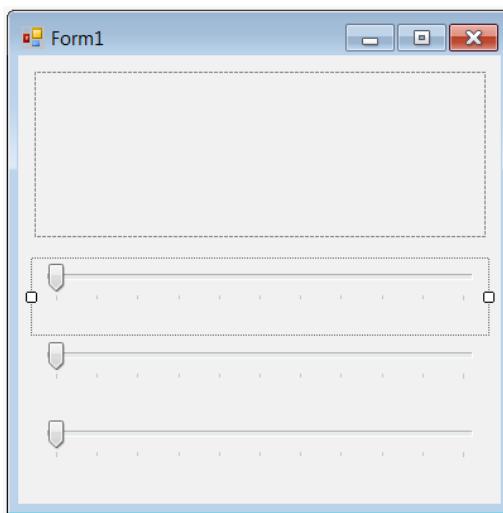
W taki sam sposób warto zadokować podokno *Properties*.

Tworzenie interfejsu za pomocą komponentów Windows Forms

W tym rozdziale przedstawię dwie kontrolki: panel (klasa *Panel*) i suwak (klasa *TrackBar*). Zbudujemy z nich interfejs aplikacji:

1. Powiększ podgląd formy w widoku projektowania (zakładka *Form1.cs [Design]*) do rozmiarów 400×400 (mniej więcej). Po zaznaczeniu kontrolki jej rozmiar jest widoczny na pasku stanu okna środowiska programistycznego.
2. Rozwiń grupę *All Windows Forms* w podoknie *Toolbox*.
3. Odnajdź w niej komponent *Panel* i zaznacz go.
4. Następnie umieść go na podglądzie formy, przeciągając myszą (z wcisniętym lewym przyciskiem) nad obszarem, na którym ma być umieszczony.
5. Postępując tak samo, umieść na formie trzy suwaki (kontrolka *TrackBar*) i rozmieśc je według wzoru z rysunku 8.4.

Rysunek 8.4.
Projekt formy aplikacji Kolorы



6. Następnie zaznacz wszystkie trzy suwaki (przytrzymując klawisz *Shift*) i za pomocą okna własności zmodyfikuj ich własność *Anchor* (zakotwiczenie) tak, aby kontrolki zaczepione zostały do lewej, dolnej i prawej krawędzi okna. Natomiast panel zakotwicz do wszystkich czterech krawędzi formy.

Po zaznaczeniu komponentu w podoknie *Toolbox* (punkt 1.) kurSOR myszy zmienia się i pokazuje ikonę wybranego komponentu. Po umieszczeniu go na formie kurSOR wraca do normalnego wyglądu, co oznacza, że żaden komponent nie jest aktualnie wybrany. Należy zwrócić uwagę na to, że przy umieszczaniu na podglądzie formy drugiego i trzeciego paska przewijania Visual C# pomaga dopasować ich położenie do wcześniej umieszczonego komponentu.

Własność *Anchor* kontrolek jest bardzo wygodnym narzędziem zwiększającym elastyczność interfejsu. Ustawienie zakotwiczenia między kontrolką a krawędzią okna wymusza stałą odległość tychże, co będzie ważne podczas zmiany rozmiaru okna. W odróżnieniu od własności *Dock*, którą poznasz w następnym rozdziale, własność *Anchor* nie przysuwa kontrolki do całej krawędzi, a jedynie „pilnuje” odległości.

Umieszczone na formie komponenty można — oczywiście — dopasowywać. Można zmieniać ich rozmiar i położenie, modyfikując je bezpośrednio na podglądzie formy. Poza tym wszystkie ich własności dostępne są w oknie własności (podokno *Properties*).

Przełączanie między podglądem formy a kodem jej klasy

Podgląd naszej formy widoczny jest w zakładce *Form1.cs [Design]*. Możemy nacisnąć klawisz *F7*, co spowoduje przeniesienie do edytora kodu klasy opisującej projektowaną formę (nowa zakładka *Form1.cs* w oknie głównym Visual Studio). Ponowne naciśnięcie klawisza *F7* lub kombinacji *Shift+F7* (to zależy od ustawień klawiszy) przeniesie nas z powrotem do widoku projektowania formy. To będą często używane klawisze.



Jeżeli klawisz *F7* nie przełącza między podglądem projektowania formy i edytorem kodu, można to ustawić (zobacz przypis nr 1 w rozdziale 1.). Należy z menu *Tools* wybrać polecenie *Customize...*, a w oknie dialogowym, które się wówczas pojawi, kliknąć przycisk *Keyboard....* Pojawi się okno opcji z wybraną pozycją *Environment, Keyboard*. W polu tekstowym *Show Command Containing* należy wpisać *View.Toggle →Designer*. W polu *Press shortcut keys* należy nacisnąć klawisz *F7* i kliknąć przycisk *Assign*.

Analiza kodu pierwszej aplikacji Windows Forms

Kod klasy opisującej projektowaną przez nas formę znajduje się w dwóch plikach. Tego nie było w projektach aplikacji konsolowej. Rozdzielenie przez Visual C# klasy na dwa pliki lub więcej możliwe jest dzięki słowu kluczowemu *partial*. Zasadnicza część klasy znajduje się w pliku *Form1.cs*. Obejmuje definicję klasy *Form1* wraz z definicją

jej konstruktora — funkcją składową o nazwie identycznej z nazwą klasy, uruchamianą po utworzeniu obiektu tej klasy. Klasa ta należy do przestrzeni nazw `Kolory` (listing 8.1). W przyszłości umieszczane tu będą także wszystkie metody zdarzeniowe.

Listing 8.1. Klasa `formy` — część przeznaczona do edycji przez programistę

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Kolory
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Jak widać na listingu 8.1, w przestrzeni nazw `Kolory` zadeklarowana jest jedna klasa (słowo kluczowe `class`) o nazwie `Form1`. Jest to klasa publiczna (słowo kluczowe `public`), czyli dostępna także spoza przestrzeni nazw `Kolory`. Klasa `Form1` dziedziczy po klasie `Form`, która jest prototypem typowego pustego okna z paskiem tytułu, z możliwością zmiany rozmiaru i kilkoma funkcjonalnościami typowymi dla okien systemu Windows.

Obecnie w tej części klasy `Form1`, która umieszczona jest w pliku `Form1.cs`, znajduje się jedynie definicja konstruktora, a więc metody o takiej samej nazwie jak nazwa klasy. Konstruktor jest specjalnym typem metody, która jest wykonywana tylko i wyłącznie w momencie tworzenia obiektu danej klasy. W tej chwili jedynym zadaniem konstruktora klasy `Form1` jest wywołanie metody `InitializeComponents`, której definicja umieszczona jest w pliku `Form1.Designer.cs` razem z kodem tworzonym automatycznie przez Visual C# (listing 8.2). Plik ten można wczytać do edytora, korzystając z przeglądarki rozwiązania (ang. *Solution Explorer*). Należy rozwinąć gałąź `Form1.cs` i dwukrotnie kliknąć w niej odpowiedni plik. Dzięki podzieleniu klasy kod, który modyfikuje programista, jest oddzielony od kodu tworzonego automatycznie w momencie wprowadzania zmian w widoku projektowania. To znacznie zwiększa przejrzystość tej części, która jest przeznaczona do edycji, ułatwia panowanie nad całością, a przede wszystkim zapobiega niezamierzonej modyfikacji kodu odpowiedzialnego za wygląd interfejsu aplikacji.

Listing 8.2. Część klasy formy znajdująca się w pliku *Form1.Designer.cs* (po dodaniu trzech pasków narzędzi i panelu)

```
namespace Kolory
{
    partial class Form1
    {
        /// <summary>
/// Required designer variable.
/// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
/// Clean up any resources being used.
/// </summary>
/// <param name="disposing">true if managed resources should be disposed; otherwise,
/// false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

#region Windows Form Designer generated code

/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
        private void InitializeComponent()
        {
            this.panell = new System.Windows.Forms.Panel();
            this.trackBar1 = new System.Windows.Forms.TrackBar();
            this.trackBar2 = new System.Windows.Forms.TrackBar();
            this.trackBar3 = new System.Windows.Forms.TrackBar();
            ((System.ComponentModel.ISupportInitialize)(this.trackBar1)).BeginInit();
            ((System.ComponentModel.ISupportInitialize)(this.trackBar2)).BeginInit();
            ((System.ComponentModel.ISupportInitialize)(this.trackBar3)).BeginInit();
            this.SuspendLayout();
            //
            // panell
            //
            this.panell.Anchor = ((System.Windows.Forms.AnchorStyles)((System.Windows.Forms.AnchorStyles.Top | System.Windows.Forms.AnchorStyles.Bottom | System.Windows.Forms.AnchorStyles.Left | System.Windows.Forms.AnchorStyles.Right)));
            this.panell.Location = new System.Drawing.Point(13, 13);
            this.panell.Name = "panell";
            this.panell.Size = new System.Drawing.Size(357, 131);
            this.panell.TabIndex = 0;
            //
            // trackBar1
            //
            this.trackBar1.Anchor = ((System.Windows.Forms.AnchorStyles)((System.Windows.Forms.AnchorStyles.Bottom | System.Windows.Forms.AnchorStyles.Left)))

```

```
| System.Windows.Forms.AnchorStyles.Right)));
this.trackBar1.Location = new System.Drawing.Point(13, 163);
this.trackBar1.Name = "trackBar1";
this.trackBar1.Size = new System.Drawing.Size(357, 56);
this.trackBar1.TabIndex = 1;
//
//trackBar2
//
this.trackBar2.Anchor = ((System.Windows.Forms.AnchorStyles)
➥((System.Windows.Forms.AnchorStyles.Bottom |
➥System.Windows.Forms.AnchorStyles.Left) |
| System.Windows.Forms.AnchorStyles.Right)));
this.trackBar2.Location = new System.Drawing.Point(13, 225);
this.trackBar2.Name = "trackBar2";
this.trackBar2.Size = new System.Drawing.Size(357, 56);
this.trackBar2.TabIndex = 2;
//
//trackBar3
//
this.trackBar3.Anchor = ((System.Windows.Forms.AnchorStyles)
➥((System.Windows.Forms.AnchorStyles.Bottom |
➥System.Windows.Forms.AnchorStyles.Left) |
| System.Windows.Forms.AnchorStyles.Right)));
this.trackBar3.Location = new System.Drawing.Point(13, 287);
this.trackBar3.Name = "trackBar3";
this.trackBar3.Size = new System.Drawing.Size(357, 56);
this.trackBar3.TabIndex = 3;
//
//Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 16F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(382, 355);
this.Controls.Add(this.trackBar3);
this.Controls.Add(this.trackBar2);
this.Controls.Add(this.trackBar1);
this.Controls.Add(this.panel1);
this.Name = "Form1";
this.Text = "Form1";
((System.ComponentModel.ISupportInitialize)(this.trackBar1)).EndInit();
((System.ComponentModel.ISupportInitialize)(this.trackBar2)).EndInit();
((System.ComponentModel.ISupportInitialize)(this.trackBar3)).EndInit();
this.ResumeLayout(false);
this.PerformLayout();
this.PerformLayout();

}

#endregion

private System.Windows.Forms.Panel panel1;
private System.Windows.Forms.TrackBar trackBar1;
private System.Windows.Forms.TrackBar trackBar2;
private System.Windows.Forms.TrackBar trackBar3;
}
}
```

Zwrócić uwagę na ostatnie linie tej części klasy (wyróżnione na listingu 8.2). Zdefiniowane są tam prywatne pola klasy odpowiadające komponentom, które umieściliśmy na formie, projektując jej interfejs. Jak pamiętamy z rozdziału 4., pola mogą być inicjowane także w tym miejscu, tj. poza konstruktorem lub któryś z metod, ale Visual Studio nie skorzystało z tej możliwości.

**Wskazówka**

Przypominam, że polecenie `Panel panel1;` widoczne na listingu 8.2 oznacza jedynie zadeklarowanie referencji (zmiennej obiektowej) typu `Panel`. Nie powstaje w tym momencie obiekt, do którego utworzenia konieczne byłoby wykorzystanie operatora `new`. Jest tak, ponieważ `Panel` jest klasą, czyli typem referencyjnym. Inaczej wygląda sprawa tworzenia obiektów będących instancjami struktur, ale o tym już była mowa w rozdziale 3.

Zwrócić uwagę na definicję metody `Form1.Dispose`. Nadpisuje ona metodę z klasy bazowej (tj. metodę `Form.Dispose`). Metoda ta jest wykorzystywana przy usuwaniu obiektu z pamięci i — podobnie jak pozostałych części kodu z pliku `Form1.Designer.cs` — nie będziemy jej samodzielnie modyfikować. Ważne, aby zdawać sobie sprawę, że wywołanie tej metody nie spowoduje usunięcia obiektu — to nie jest odpowiednik destruktora w C++. Zresztą programista ma w platformie .NET ograniczoną możliwość usuwania obiektów z pamięci. Tym zajmuje się *odśmieczacz* (ang. *garbage collector*), który cyklicznie przegląda wszystkie obiekty typów referencyjnych utworzone w trakcie działania aplikacji i sprawdza, czy nadal wskazują na nie jakieś referencje. Jeżeli nie, obiekt na pewno nie jest używany i może zostać usunięty³.

Kolejną metodą klasy `Form1` umieszczoną w tym pliku jest wywoływana z konstruktora metoda `InitializeComponents`. W niej znajdują się polecenia tworzące obiekty komponentów i konfigurujące ich własności. Pierwsze linie tej metody tworzą paski przewijania za pomocą operatora `new` i następujących po nim wywołań konstruktorów domyślnych (tj. bezargumentowych) klas `Panel` i `TrackBar`. Kolejność tych poleceń zależy od kolejności umieszczania kontrolek na podglądzie formy. Następne linie tej metody określają kolejno: położenie (własność `Location`) i rozmiar (`Size`) komponentu, jego nazwę (co może być zaskakujące) oraz własność `TabIndex`, odpowiadającą za kolejność przejmowania przez komponenty „focusu” przy korzystaniu z klawisza `Tab` do nawigacji na formie. Wreszcie na końcu zebrana jest seria wywołań metody `this.Controls.Add`, która gotowe komponenty przypisuje do formy.

**Wskazówka**

Jak wiemy z rozdziału 4., słowo kluczowe `this` jest predefiniowaną zmienną referencyjną pozwalającą na odwołanie do bieżącego obiektu. W tym przypadku wskazuje obiekt klasy `Form1`, który zostanie utworzony po uruchomieniu aplikacji.

Istotny z punktu widzenia uruchamiania programu jest jeszcze jeden plik zawierający kod źródłowy, a mianowicie `Program.cs`. Możemy go otworzyć, korzystając z podokna `Solution Explorer`. Zobaczmy wówczas, że podobnie jak to miało miejsce

³ Można wymusić cykl *garbage collectora*, a tym samym odzyskać nieużywaną pamięć, wywołując statyczną metodę `GC.Collect`. Metoda ta nie działa asynchronicznie, a więc zakończy się dopiero po wykonaniu pełnego cyklu. Ilość zrealizowanych cykli można sprawdzić metodą `GC.CollectionCount`.

w aplikacjach konsolowych, zawiera on prostą klasę statyczną `Program` (listing 8.3), w której znajduje się tylko jedna metoda statyczna o nazwie `Main` — główne wejście do aplikacji.

Listing 8.3. Kod znajdujący się w pliku `Program.cs`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Kolory
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

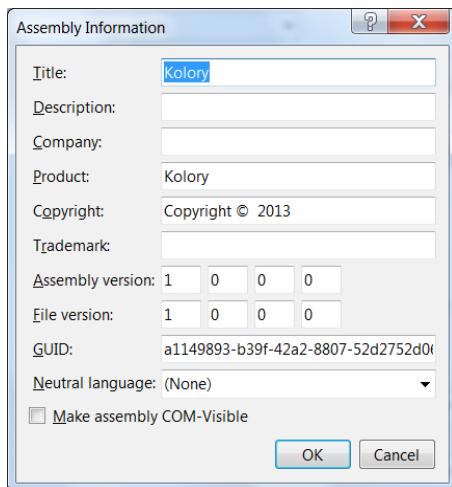
To właśnie metoda `Main` (należy pamiętać o tym, że w C# piszemy ją wielką literą) jest wywoływana przez system i platformę .NET po próbie uruchomienia aplikacji przez użytkownika. W metodzie `Main` powinny znaleźć się polecenia, które zbudują okno aplikacji, a więc w naszym przykładzie polecenie utworzenia obiektu klasy `Form1`⁴.

Wyrażenie `[STAThread]` znajdujące się bezpośrednio przed sygnaturą metody `Main` jest atrybutem, czyli pewną wiadomością dla kompilatora, a po skompilowaniu również dla platformy .NET, na której będzie uruchomiona aplikacja *Kolory*. W tym przypadku atrybut informuje, że aplikacja powinna komunikować się z systemem w trybie pojedynczego wątku.

Zajrzyjmy jeszcze na chwilę do podkatalogu *Properties*. Znajduje się w nim kilka plików. Część z nich ma rozszerzenia `.cs`, co oznacza, że zawierają kod C#. Plik *AssemblyInfo.cs* przechowuje informacje o numerze wersji, zastrzeżonych znakach towarzyszących itp. Jest to plik C#, ale zamiast edytować go samodzielnie, wygodniej użyć edytora. Można go uruchomić w oknie opcji: menu *Project, Kolory Properties...*, zakładka *Application*, przycisk *Assembly Information...* (rysunek 8.5). Pliki z *Settings* w nazwie przechowują ustawienia aplikacji. One również powinny być raczej edytowane

⁴ W zasadzie metoda `Main` mogłaby być z powodzeniem zdefiniowana w obrębie klasy `Form1` jako jej metoda statyczna. Z takiego rozwiązania korzystał swego czasu konkurencyjny Borland C#Builder. Wystarczy przenieść w całości metodę `Main` do klasy `Form1` (nie można zostawić pierwotnego!). Plik `Program.cs` można wówczas w ogóle usunąć z projektu.

Rysunek 8.5.
Okno pozwalające na edycję wizytówki aplikacji



za pomocą edytora w oknie opcji (więcej na ten temat w następnym rozdziale). Z kolei pliki zawierające w nazwie słowo *Resources* tworzone są na potrzeby menedżera zasobów — ułatwia on m.in. tworzenie międzynarodowych wersji interfejsu.

Metody zdarzeniowe

Esencja „projektowania wizualnego” jest intuicyjne tworzenie interfejsu aplikacji, co zrobiliśmy w poprzednich zadaniach, oraz możliwość łatwego określania reakcji aplikacji na zdarzenia, czym zajmiemy się właśnie w tej chwili. Przede wszystkim chodzi o reagowanie na czynności wykonane przez użytkownika, np. zmianę pozycji suwaka.

Metoda uruchamiana w przypadku wystąpienia zdarzenia kontrolki

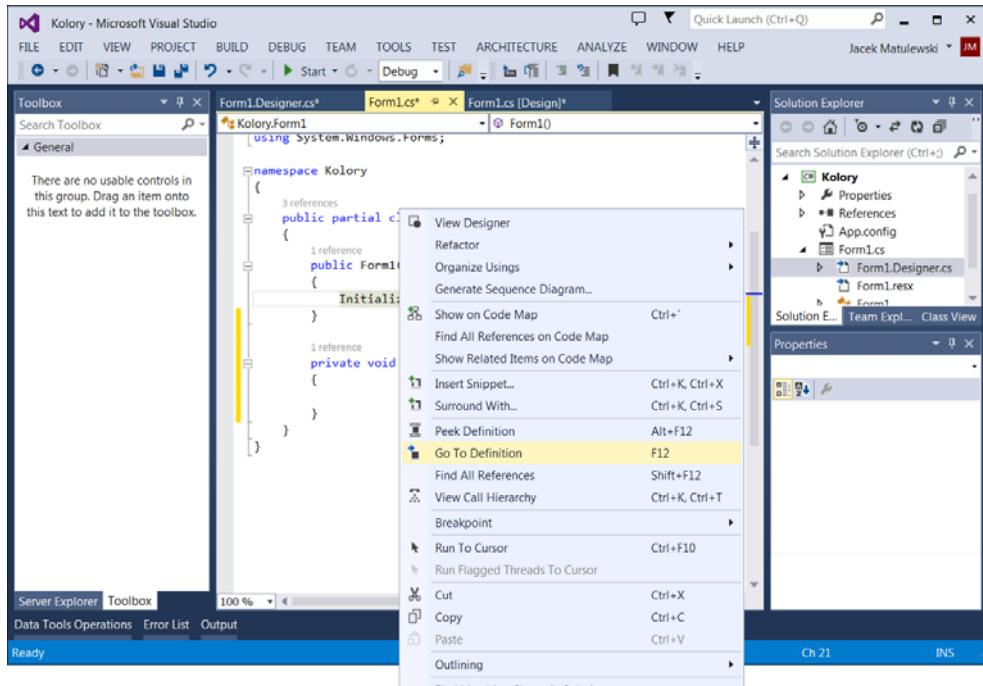
Utwórzmy metodę związaną ze zdarzeniem `ValueChanged` kontrolki `trackBar1`:

1. Zmień zakładkę na *Form1.cs [Design]*, aby wrócić do trybu projektowania formy (np. naciskając klawisz *F7*).
2. Jeżeli nie zrobiłeś tego już wcześniej, naciśnij klawisz *F4*, aby zobaczyć okno właściwości. Przypnij je tak, aby jego zawartość była stale widoczna.
3. Następnie na podglądzie formy zaznacz pierwszy suwak o nazwie `trackBar1`. Nazwa zazначенego komponentu powinna stać się widoczna w oknie właściwości.
4. Na pasku narzędzi samego okna właściwości kliknij ikonę *Events* (zdarzenia). Ikona ta ma kształt błyskawicy.
5. Odnajdź zdarzenie `ValueChanged` i dwukrotnie kliknij związane z nim pole edycyjne.

W efekcie z powrotem przenieśliśmy się do edytora kodu (zakładka *Form1.cs*). Kursor został ustawiony w nowo utworzonej metodzie *trackBar1_ValueChanged*, dokładnie w miejscu, w którym będziemy musieli napisać kod określający reakcję aplikacji na zmianę pozycji suwaka. Umieścimy tu polecenie zmieniające kolor panelu.

Po utworzeniu metody zdarzeniowej warto zająć się do metody *InitializeComponent*. Metoda ta zdefiniowana jest w pliku *Form1.Designer.cs*, ale najprościej można przejść do niej, wybierając pozycję *Go To Definition* w menu kontekstowym edytora rozwiniętym w miejscu wywołania tej metody w konstruktorze (rysunek 8.6). W metodzie tej znajdziemy nową linię.

```
this.trackBar1.ValueChanged += new  
System.EventHandler(this.trackBar1_ValueChanged);
```



Rysunek 8.6. Menu kontekstowe kryje wiele wygodnych funkcji

Została ona dodana w momencie tworzenia metody zdarzeniowej. Jest odpowiedzialna za przypisanie metody zdarzeniowej do zdarzenia *ValueChanged* paska przewijania.

Testowanie metody zdarzeniowej

A teraz zmodyfikujmy metodę tak, aby po poruszeniu pierwszym suwakiem panel zmieniał kolor na szkarłatny.

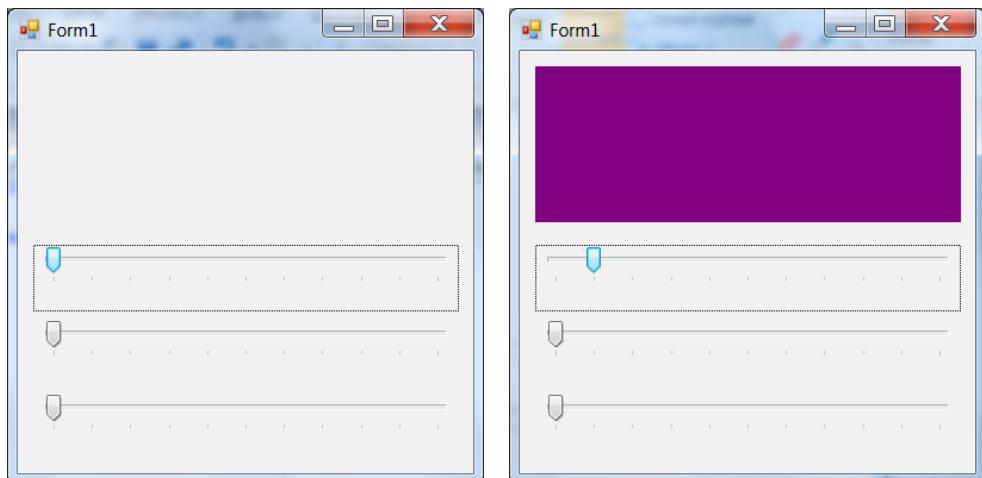
1. Do definicji metody zdarzeniowej dopisz polecenie zmiany koloru panelu, zgodnie z listkiem 8.4.

Listing 8.4. Do metody należy dopisać jedynie wyróżnioną linię

```
private void trackBar1_ValueChanged(object sender, EventArgs e)
{
    panel1.BackColor = Color.Purple;
}
```

2. Skompiluj i uruchom projekt, naciskając klawisz *F5*.

Naciskając klawisz *F5* (tabela 8.1), po raz pierwszy komplujemy naszą aplikację. Po komplikacji i uruchomieniu aplikacji w trybie debugowania w oknie Visual Studio ukryte zostały podokna własności i palety narzędzi, a w zamian udostępnione zostały nowe podokna służące właśnie do debugowania. I co najważniejsze, na ekranie pojawiło się okno uruchomionej aplikacji *Kolor* (rysunek 8.7, lewy). Jej skompilowany plik *Kolor.exe* został umieszczony w podkatalogu *bin\Debug* katalogu projektu.



Rysunek 8.7. Aplikacja *Kolory* po skompilowaniu i uruchomieniu. Z lewej — zaraz po uruchomieniu, z prawej — po poruszeniu pierwszym paskiem przewijania

Jeżeli poruszmy drugim lub trzecim suwakiem, nic się nie stanie, ale jeżeli zmienimy pozycję pierwszego suwaka, kolor panelu zmieni się (rysunek 8.7, prawy).

Do określenia koloru panelu w metodzie zdarzeniowej wykorzystaliśmy strukturę⁵ *Color*, a dokładniej jej element *Purple*. Wybór predefiniowanych kolorów jest ogromny. Poza typowymi kolorami są również takie jak cytrynowy szyfon, kolor orchidei czy wykorzystany przez nas purpurowy. Ponadto dostępna jest osobna struktura *System.Drawing.Color*, z której programista może wybrać kolory zdefiniowane w bieżącej konfiguracji interfejsu systemowego (np. kolor pulpitu lub kolor paska tytułu).

⁵ *Color* nie jest klasą, choć jego instancje są obiektami. Jest strukturą. O różnicach między tymi dwoma typami danych można przeczytać w rozdziale 3.

Przypisywanie istniejącej metody do zdarzeń komponentów

Przypiszemy teraz metodę `trackBar1_ValueChanged` do zdarzenia `ValueChanged` dwóch pozostałych suwaków:

1. Zamknij uruchomioną aplikację *Kolory* i wróć do okna Visual C#.
2. Naciskając klawisz *F7*, przejdź do widoku projektowania na zakładce *Form1.cs [Design]*.
3. Przytrzymując klawisz *Ctrl*, zaznacz dwa suwaki o nazwach `trackBar2` i `trackBar3`.
4. Odszukaj zdarzenie `ValueChanged` w oknie właściwości i z rozwijanej listy znajdującej się przy nim wybierz metodę `trackBar1_ValueChanged`.
5. Ponownie skompiluj i uruchom aplikację (*F5*).

Możemy się przekonać, że po wykonaniu ostatniego zadania poruszenie którymkolwiek suwakiem powoduje zmianę koloru panelu. Oznacza to, że metoda zdarzeniowa związana jest teraz ze zdarzeniem `ValueChanged` wszystkich trzech suwaków.

Edycja metody zdarzeniowej

Zbliżamy się do końca projektowania pierwszej aplikacji. Nadajmy zatem metodzie zdarzeniowej `trackBar1_ValueChanged` jej ostateczną postać, uzależniając kolor panelu od pozycji suwaków. W tym celu w edytorze kodu zmieniamy polecenie metody zdarzeniowej zgodnie z listingiem 8.5 (zalecam korzystanie z mechanizmu *IntelliSense*) i jeszcze raz uruchamiamy aplikację (*F5*).

Listing 8.5. Kolor panelu określamy na podstawie składowych *RGB* wyznaczonych przez pozycje suwaków

```
private void hScrollBar1_ValueChanged(object sender, EventArgs e)
{
    panel1.BackColor = Color.FromArgb(trackBar1.Value,
                                      trackBar2.Value,
                                      trackBar3.Value);
}
```

Do wykonania zadania wykorzystaliśmy metodę statyczną `Color.FromArgb`. Metoda ta jest wielokrotnie przeciążona, tzn. możemy ją wywoływać, podając różne zestawy argumentów. Tu wykorzystaliśmy taką jej wersję, która pobiera trzy liczby typu `int`. Oznaczają one wówczas składowe R, G i B koloru, a więc intensywność komponentów czerwieni, zieleni i niebieskiego. Kolory w C# obejmują również czwartą składową określającą przezroczystość (zwykle nazywaną kanałem *alpha*, stąd jej oznaczenie A), ale nie będziemy jej tutaj wykorzystywać. Argumentami metody `Color.FromArgb` są właściwości `Value` poszczególnych suwaków informujące o ich pozycji.

Modyfikowanie właściwości komponentów

Po uruchomieniu aplikacji przekonamy się jednak, że coś jest nie do końca w porządku. Kolory pokazywane na panelu są bardzo ciemne. Gdy przesuniemy wszystkie suwaki do położenia maksymalnie na prawo, zamiast białego zobaczymy jedynie ciemnoszary. Dlaczego? Przyczyną jest określany przez właściwości `Minimum` i `Maximum` zakres wartości właściwości `Value` suwaków. Domyslnie minimalna i maksymalna wartość to 0 i 100, podczas gdy argumenty metody `Color.FromArgb` mogą przyjmować wartości od 0 do 255 (trzy zera dla czarnego i trzy wartości 255 dla białego). Możemy przeliczyć zakres, korzystając z prostych operacji arytmetycznych, ale naturalniejszym i bardziej eleganckim rozwiązaniem będzie zwiększenie maksymalnej wartości właściwości `Value` suwaka.

Zmieńmy zatem właściwość `Maximum` suwaków tak, aby przyjmowały wartości do 255:

1. Zamknij działającą aplikację *Kolory*, jeżeli jest uruchomiona.
2. Przełącz się do widoku projektowania (*F7*).
3. Zaznacz wszystkie trzy paski przewijania (przytrzymując klawisz *Shift* lub *Ctrl*).
4. W pasku narzędzi okna właściwości kliknij ikonę *Properties*.
5. Znajdź właściwość `Maximum` i zmień jej wartość na 255 (po wpisaniu wartości należy ją potwierdzić, naciskając *Enter* lub zmieniając zaznaczoną właściwość).
6. Zmień również właściwość `TickFrequency` suwaków na 15.

Teraz po uruchomieniu aplikacji możemy na panelu uzyskać całą gamę barw od czerni do bieli.

Wywoływanie metody zdarzeniowej z poziomu kodu

Ostatnią czynnością będzie uzgodnienie koloru panelu z pozycją suwaków. W tej chwili paski przewijania wskazują na kolor czarny, a panel ma kolor identyczny z płaszczystą formą i w efekcie jest niewidoczny. Z tego powodu po poruszeniu któregokolwiek z suwaków następuje nagła zmiana koloru. Rozwiążemy ten problem, wymuszając uzgodnienie koloru panelu tuż po uruchomieniu aplikacji przez wywołanie metody zdarzeniowej suwaków w konstruktorze formy. W tym celu do konstruktora klasy `Form1` należy dodać wywołanie metody `trackBar1_ValueChanged` (listing 8.6).

Listing 8.6. Konstruktor klasy opisującej formę

```
public Form1()
{
    InitializeComponent();
    this.trackBar1_ValueChanged(this, null);
}
```

Wywołując metodę zdarzeniową, należy jako jej pierwszy argument podać obiekt, który wysyła zdarzenie (ang. *sender*, czyli nadawca; zwykle jest nim któryś z pasków

przewijania). Tu wskazałem obiekt formy, korzystając z referencji `this`. Drugi argument pozwala przekazać dodatkowe informacje do metody. Podałem po prostu pustą referencję `null`. Nie ma to w naszym przypadku większego znaczenia, bo w metodzie zdarzeniowej `trackBar1_ValueChanged` w ogóle nie korzystamy z jej argumentów.

Aby rozwiązać problem uzgodnienia koloru panelu z pozycjami suwaków, moglibyśmy także postąpić inaczej. Wystarczyłoby za pomocą okna własności ustawić własność `BackColor` panelu na czarny. Jednak wersja przedstawiona na listingu 8.6 ma tę zaletę, że w trakcie projektowania możemy dowolnie ustawić pozycję suwaków, a kolor panelu po uruchomieniu aplikacji automatycznie się do nich dostosuje.

Reakcja aplikacji na naciśkanie klawiszy

Do obsługi aplikacji okienkowej można oprócz myszy używać klawiatury. Często spotykanym rozwiązaniem jest np. zamykanie okna aplikacji po naciśnięciu przez użytkownika klawisz *Esc*. Aby uzyskać taki efekt w naszej aplikacji, należy wykorzystać zdarzenie `KeyPress`, które sygnalizuje naciśnięcie (wciśnięcie i zwolnienie) któregokolwiek z klawiszy poza klawiszami specjalnymi (tj. *Ctrl*, *Alt* i *Shift*).

1. Przejdź do widoku projektowania.
2. Zaznacz formę, ale nie żaden z umieszczonych na niej komponentów (najłatwiej kliknąć jej pasek tytułu).
3. W oknie własności, na zakładce *Properties*, zmień własność `KeyPreview` formy na `true`.
4. Zmień zakładkę na *Events* i kliknij dwukrotnie pole przy zdarzeniu `KeyPress`.
5. W utworzonej w ten sposób metodzie zdarzeniowej umieść polecenie wyróżnione w listingu 8.7.

Listing 8.7. Jeżeli użytkownik naciśnie klawisz *Esc*, aplikacja zostanie zamknięta

```
private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == '\u001B') Close();
}
```

Dodane polecenie to instrukcja warunkowa `if` (rozdział 3.) sprawdzająca, czy wciśnięty klawisz, który można odczytać z właściwości `e.KeyChar` drugiego z argumentów metody, jest klawiszem *Esc*. Klawiszowi temu nie odpowiada żaden znak alfabetu, ale mimo to posiada on w zestawie znaków ASCII numer 27, czyli w układzie szesnastkowym 1B. Właśnie dlatego naciśnięty klawisz przymywany jest do \u001B, czyli znaku, któremu w tablicy znaków Unicode⁶ odpowiada kod 27 (1B). Jeżeli warunek zostanie spełniony, wywoływana jest metoda `Close` formy, która ją zamyka. A ponieważ jest to jedyna forma, aplikacja kończy działanie.

⁶ Zestaw znaków Unicode obejmuje zestaw ASCII (pozycje od 0 do 127).

Możemy uniknąć bezpośredniego odwoływanego do kodu ASCII, jeżeli zamiast zdarzenia KeyPress użyjemy zdarzenia KeyDown. W jego metodzie zdarzeniowej drugi argument jest bogatszy i zawiera między innymi własność KeyCode typu Keys — typu wyliczeniowego zawierającego wszystkie kody znaków. Wówczas warunek z instrukcji if mógłby być zapisany jako e.KeyCode == Keys.Escape.

Rozdział 9.

Przegląd komponentów biblioteki Windows Forms

Podstawowe zagadnienia związane z tzw. projektowaniem wizualnym, w trakcie którego wykorzystywane są kontrolki z biblioteki Windows Forms, zostały już omówione w poprzednim rozdziale. Teraz chciałbym się skupić na przeglądzie najczęściej używanych kontrolek z tej biblioteki.

Notatnik.NET

Pierwszym projektem przedstawionym w tym rozdziale będzie prosty edytor — zasadzany odpowiednik systemowego notatnika. Na nim przećwiczmy większość typowych zagadnień związanych z przygotowywaniem interfejsu (szczególnie projektowanie menu głównego, paska stanu, okien dialogowych) oraz kwestii dotyczących wczytywania i zapisywania tekstu do plików oraz ich drukowania.

Projektowanie interfejsu aplikacji i menu główne

Przygotowanie interfejsu wymaga staranności — jest to przecież jedyny kanał komunikacji aplikacji z jej użytkownikiem. Musimy zatem wziąć pod uwagę możliwość zmiany rozmiaru okna, zmiany kolorów w środowisku Windows, zmiany domyślnej wielkości czcionki w systemie itp. Część z tych „niespodzianek” uwzględniona została już na poziomie kontrolek, z których budujemy interfejs. Dla przykładu domyślny kolor komponentów jest ustalany na podstawie parametrów odczytanych z systemu i przechowywanych w klasie `SystemColor` (zobacz domyślne wartości właściwości `ForeColor` i `BackColor` komponentów).

Interfejs naszego notatnika nie będzie odbiegał od standardu prostych edytorów. Oznacza to, że jego podstawą będzie pole tekstowe `TextBox` wypełniające niemal całą wolną przestrzeń na formie (właściwość `Dock` komponentu ustawimy na `Fill`). Nad nim znajdzie się tylko menu główne, a pod nim — pasek stanu.

Tworzenie projektu aplikacji i jej interfejsu

Zacznijmy od utworzenia projektu aplikacji z polem tekstowym wypełniającym cały obszar użytkownika formy.

1. W środowisku Visual Studio wcisnij kombinację klawiszy *Ctrl+Shift+N*.

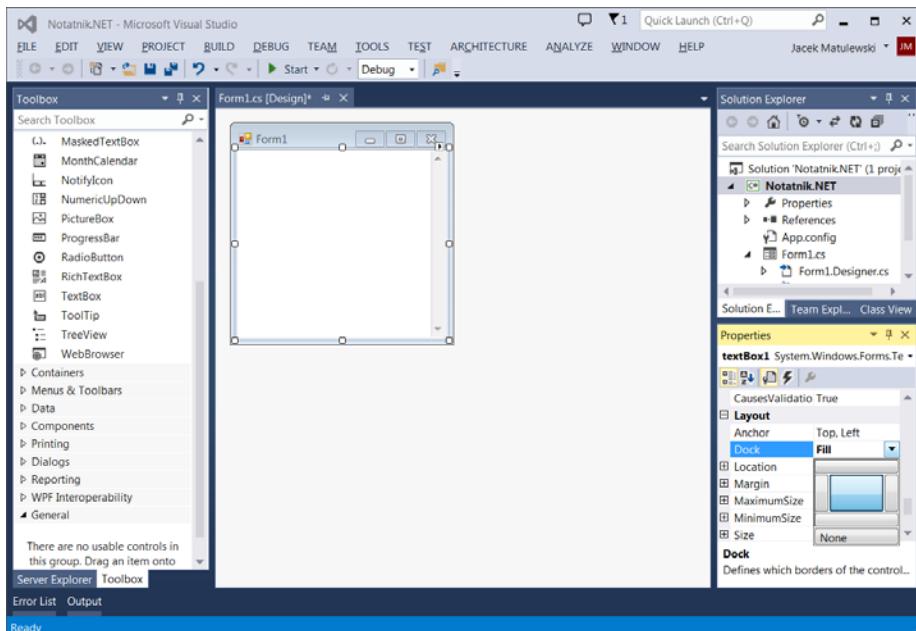
2. W oknie *New Project*:

- a)** zaznacz pozycję *Windows Forms Application*;
- b)** w polu *Name* wpisz nazwę aplikacji **Notatnik.NET**;
- c)** kliknij *OK*.

3. Następnie w widoku projektowania (zakładka *Form1.cs [Design]*), do którego zostałeś automatycznie przeniesiony po utworzeniu projektu, w palecie komponentów (podokno *Toolbox*), w kategorii *Common Controls*, odnajdź komponent *TextBox* i umieść go na powierzchni formy.

4. Zaznacz dodany do formy komponent i w oknie właściwości (*F4*):

- a)** przełącz okno właściwości do widoku w kategoriach (przycisk *Categorized*);
- b)** w grupie *Behavior* zmień właściwość *Multiline* na *True* (to ustawienie dostępne jest też w liście bocznej tej kontrolki);
- c)** w grupie *Layout* odnajdź właściwość *Dock*; z właściwością tą związane jest proste okno dialogowe, które pozwala na wybór położenia komponentu na formie (rysunek 9.1); wybierz *Fill* — pole tekstowe wypełni całą dostępną przestrzeń okna;



Rysunek 9.1. Edytor właściwości *Dock*

- d) zmień także widoczną w grupie *Appearance* własność *ScrollBars* (paski przewijania) na *Both*.

Dzięki ustawieniu właściwości *Dock* nie musimy się martwić o zachowanie pola tekstowego w razie zmiany rozmiaru formy przez użytkownika — komponent powinien zmieniać swój rozmiar tak, żeby dopasować go do nowego rozmiaru okna. Warto uruchomić aplikację (*F5*) i sprawdzić, jak to działa.

Zmiana nazwy okna

Nazwa okna pozwala na identyfikację aplikacji. Warto zatem umieścić na pasku tytułu nazwę naszej aplikacji, tj. *Notatnik.NET*. Będzie ona również widoczna na pasku zadań.

1. W widoku projektowania zaznacz klasę okna *Form1* (np. klikając pasek tytułu formy na podglądzie) i przejdź do listy właściwości na zakładce *Properties*.
2. Odnajdź właściwość *Text* formy i w związanym z nią polu edycyjnym wpisz *Notatnik.NET*. Naciśnij *Enter*, aby potwierdzić zmianę.

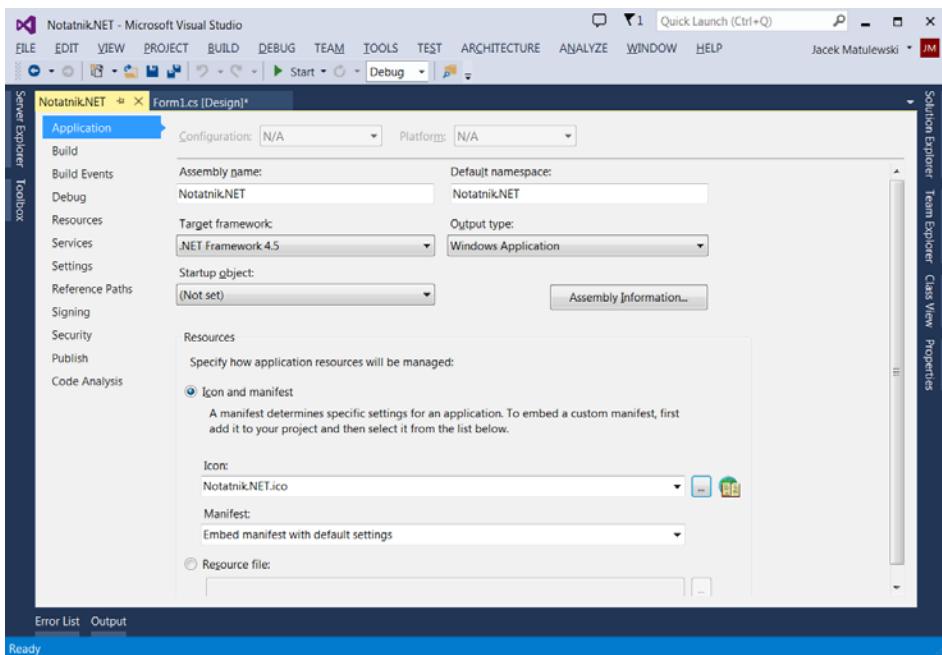
Zmiana ikony okna i aplikacji

Ustawmy ikonę formy i projektu. W tym celu:

1. Ponownie zaznacz obiekt okna *Form1*, klikając jego pasek tytułu w podglądzie.
2. Odnajdź jego właściwość *Icon* w grupie *Window Style* i kliknij przycisk z trzema kropkami.
3. Otworzy się standardowe okno dialogowe, w którym możesz wskazać plik o rozszerzeniu *.ico*. Zostanie z niego pobrana ikona wyświetlna na pasku tytułu formy. Po kliknięciu *Otwórz* wybierz jakiś plik *.ico* i zamknij okno dialogowe. Przykładowy plik można znaleźć w kodach źródłowych dołączonych do książki.
4. Następnie z menu *Project* wybierz pozycję *Notatnik.NET Properties....*
5. Pojawi się zakładka *Notatnik.NET* (rysunek 9.2), na której kliknij przycisk z trzema kropkami przy polu *Icon* w grupie *Resources*.
6. Ponownie zobaczysz okno dialogowe wyboru pliku. Wybierz w nim plik zawierający ikonę aplikacji¹, włączoną do skompilowanego pliku *.exe*. Nic nie stoi na przeszkodzie, a nawet wskazane jest, aby użyć tej samej ikony, którą wcześniej umieściłeś na pasku tytułu. Kliknij *Otwórz*, aby zamknąć okno dialogowe.
7. Teraz zamknij zakładkę z właściwościami projektu i naciśnij klawisz *F6*, aby zbudować cały projekt, a następnie *F5*, aby go uruchomić.

W obu przypadkach ikona zostanie umieszczona w zasobach aplikacji, a więc oryginalny plik *.ico* nie będzie więcej potrzebny. Nie trzeba go więc dołączać do rozpoznanej aplikacji.

¹ Chodzi o ikonę, którą można zobaczyć w Eksploratorze Windows przy przeglądaniu plików.



Rysunek 9.2. Ustawienia projektu

W ikonie może znajdować się kilka obrazów o różnych rozmiarach i głębi kolorów. Windows sam wybierze najwłaściwszy, najlepiej pasujący do miejsca, w którym ma być wyświetlony. Jeżeli nie znajdzie idealnie dopasowanego, wybierze najbardziej zbliżony i go dostosuje. Przykładowo plik *.ico* z ikoną dla aplikacji i panelu sterowania w nowych wersjach systemu Windows (od wersji Vista) powinien zawierać pełny zestaw obrazów 16×16, 32×32, 48×48, 256×256 pikseli, ponieważ użytkownik może dowolnie zmieniać wielkość ikon oglądanych w oknach eksploratora. Ikony paska narzędzi powinny z kolei zawierać obrazy o rozmiarach 16×16, 24×24 i 32×32. Najlepszym rozmiarem dla ikon paska szybkiego uruchamiania jest 40×40. Ten rozmiar powinien też znaleźć się w ikonach wykorzystywanych w *dymkach* (ang. *balloon*)².

Pasek stanu

Biblioteka Windows Forms udostępnia komponent *StatusStrip* implementujący pasek stanu. Umieścmy go na formie i przygotujmy do wyświetlania nazwy pliku wczytanego do notatnika. W tym celu:

1. Dodaj do okna pasek stanu, czyli komponent *StatusStrip* z grupy *Menus & Toolbars* (wystarczy kliknąć go dwukrotnie w podoknie *Toolbox*). Jego własność *Dock* jest automatycznie ustawiona na *Bottom*, co powoduje, że jest przyklejony do dolnej krawędzi formy.

² Przewodnik dla projektantów ikon dla aplikacji systemu Windows można znaleźć w MSDN pod adresem <http://msdn.microsoft.com/en-us/library/windows/desktop/aa511280.aspx> (fragment dokumentacji *Windows Desktop Developement*).

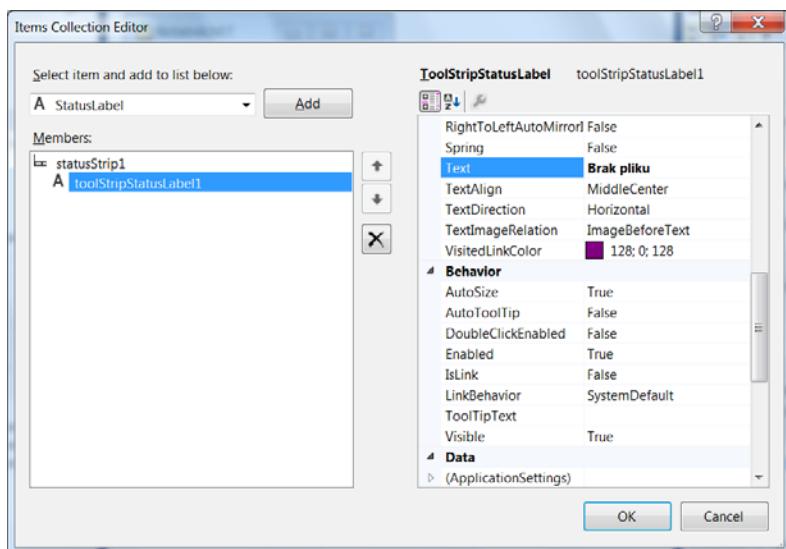


Wskazówka

Przy okazji zmian wprowadzanych w ustawieniach projektu warto zwrócić uwagę na rozwijaną listę *Target Framework*. Jej domyślna wartość w Visual Studio 2013 to .NET Framework 4.5.1. Oprócz tej wersji platformy .NET dostępne są również wersje 4.5, 4.0, 3.5, 3.0 i 2.0. Wersje 4 i 3.5, oprócz pełnej, posiadają również wariant z dopiskiem *Client Profile*. Jest to wersja platformy .NET zoptymalizowana dla zwykłych aplikacji „okienkowych”, tzn. zawierająca m.in. biblioteki Windows Forms, WPF, WCF i biblioteki konieczne do obsługi instalatorów ClickOnce, biblioteki Entity Framework, LINQ to SQL, typy dynamiczne itd. W wersji tej pominięto natomiast biblioteki ASP.NET, obsługę baz danych Oracle i niektóre zaawansowane funkcje WCF. Warto rozważyć możliwość zachowania zgodności ze starszymi wersjami platformy .NET, które teoretycznie mogą być zainstalowane na znacznie większej liczbie komputerów niż najnowsza wersja 4.0. W praktyce należy pamiętać, że za aktualizację platformy .NET odpowiada mechanizm Windows Update. W większości omawianych w tym rozdziale projektów można wybrać nawet wersję 2.0.

2. Powstanie obiekt statusStrip1. Zaznacz go. W oknie właściwości odszukaj właściwość Items i kliknij przycisk z wielokropkiem, aby uruchomić edytor kolekcji (rysunek 9.3).

Rysunek 9.3.
Konfigurowanie
paska stanu



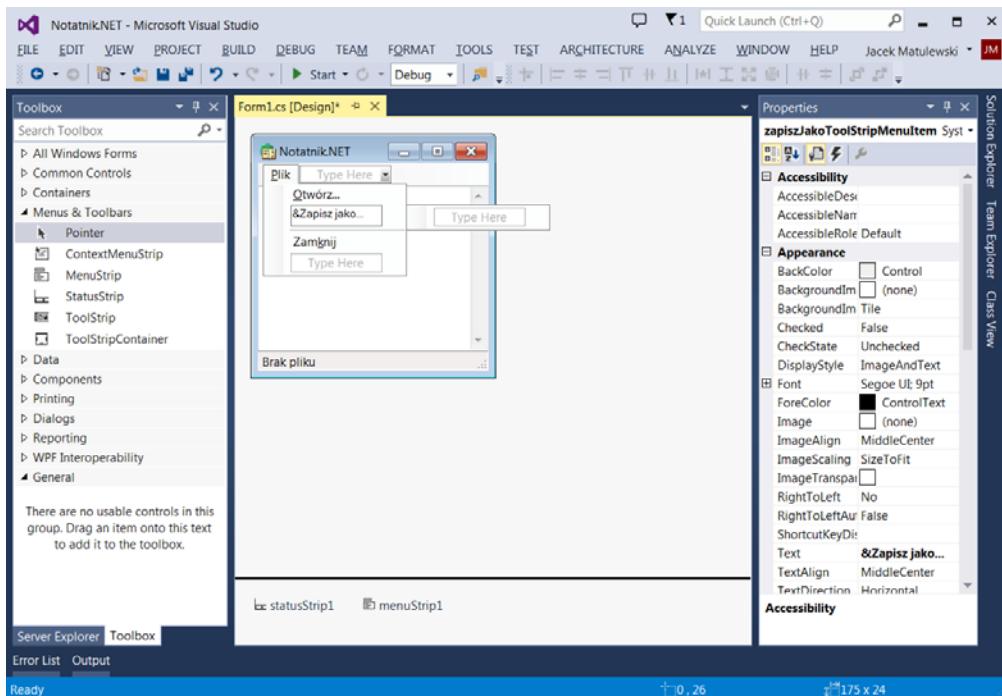
3. W oknie *Items Collection Editor* z rozwijanego menu z lewej strony wybierz *Status Label* (w zasadzie jest on wybrany domyślnie) i kliknij przycisk *Add*. W ten sposób dodasz do paska stanu pole tekstowe.
4. Nie zamkując okna edytora, zaznacz dodany przed chwilą element *toolStripStatusLabel1* na lewej liście i za pomocą lokalnego okna właściwości zmień jego właściwość *Text* na *Brak pliku*.
5. Kliknij przycisk *OK*, aby zamknąć edytor.

Z powyższego ćwiczenia wynika, że komponent paska stanu jest pojemnikiem dla innych elementów, np. użytego przez nas *ToolStripStatusLabel*. Dopiero ten element pozwala na umieszczenie napisów widocznych na pasku stanu.

Menu główne aplikacji

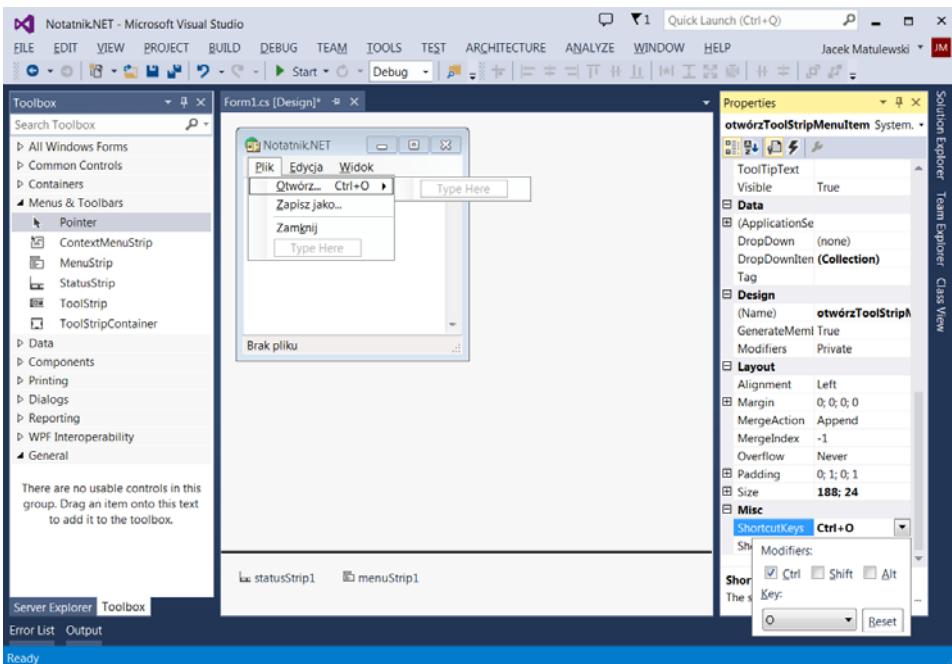
Kolejnym bardzo typowym elementem, który można dodać do aplikacji, jest menu główne. Umieścmy je na formie i wyposażmy w pozycje, jakie znajdują się w standardowym notatniku systemu Windows.

1. W widoku projektowania, w podoknie *Toolbox*, w grupie *Menus & Toolbars*, zaznacz komponent *MenuStrip* i umieść go na formie. Podobnie jak komponent paska stanu, także i ten komponent znajdzie się na dodatkowym pasku pod podglądem formy. Ten komponent ma jednak także reprezentację na podglądzie formy — jest to zarazem edytor menu.
2. Po wstawieniu menu sposób ułożenia pola edycyjnego *textBox1* względem okna (własność *Dock* ustawiona na *Fill*) może nie być prawidłowy, ponieważ menu ułożone jest niejako nad polem edycyjnym i przesłania jego górną część. W takiej sytuacji zaznacz pole edycyjne i z jego menu kontekstowego wybierz polecenie *Bring to Front*.
3. Jeżeli komponent *menuStrip1* jest zaznaczony, w podglądzie okna staje się widoczny całkiem wygodny edytor menu. Miejsce, w którym możesz wpisać nazwę podmenu, jest oznaczone napisem *Type Here* (z ang. *pisz tutaj*).
4. Wprowadź nazwę pierwszego podmenu: *&Plik*. Po rozpoczęciu pisania natychmiast pojawią się dodatkowe miejsca pozwalające na utworzenie kolejnego podmenu oraz pierwszej pozycji w podmenu *Plik*. Oba oznaczone napisami *Type Here* (rysunek 9.4).



Rysunek 9.4. Edytor menu

5. W podmenu *Plik* umieść cztery pozycje: *&Otwórz...*, *&Zapisz jako...*, separator (należy wpisać pojedynczy znak myślnika) i *Zamknij*.
6. Następnie dodaj podmenu *&Edycja* z pozycjami *Cofnij*, *Wytnij*, *Kopiuj*, *Wklej*, *Usuń* oraz *Zaznacz wszystko*. W razie kłopotów z edycją nowego podmenu możesz użyć klawisza *Tab* do przejścia do odpowiedniej pozycji w edytorze menu.
7. Dodaj także podmenu *&Widok*, w którym możesz umieścić pozycje *&Czcionka...*, *&Tło...* i *Pasek &stanu*.
8. Następnie zaznacz w edytorze pozycję *Otwórz...* i w oknie własności odnajdź *ShortcutKeys*. Po uruchomieniu edytora tej własności zaznacz w nim opcję *Ctrl*, a z rozwijanego menu wybierz *O* (rysunek 9.5). Analogicznie powiąż z pozycją *Zapisz jako...* kombinację klawiszy *Ctrl+S*.



Rysunek 9.5. Przypisywanie poleceń menu klawiszom skrótów

Umieszczony w nazwie pozycji menu znak **&** wskazuje na aktywny klawisz (oznaczona w ten sposób litera będzie podkreślona po naciśnięciu lewego klawisza *Alt* i uaktywnieniu menu). Klawisze aktywne pozwalają na nawigację w menu bez użycia myszy — wystarczy nacisnąć klawisz odpowiadający podkreślonej literze (*O* w przypadku *Otwórz...*), żeby uruchomić związaną z nią metodę zdarzeniową. To nie koniec ułatwień dla osób nielubiących odrywać dloni od klawiatury: w punkcie 8. zdefiniowaliśmy dwa klawisze skrótu związane z operacjami na plikach. Naciśnięcie kombinacji *Ctrl+O* spowoduje (oczywiście dopiero po przygotowaniu w następnych ćwiczeniach odpowiednich metod) otwarcie okna dialogowego odpowiedzialnego za wskazanie pliku do wczytania, a *Ctrl+S* — okna dialogowego pozwalającego na wybranie nazwy pliku do zapisu zawartości notatnika.

Okna dialogowe i pliki tekstowe

Zakończyliśmy przygotowywanie interfejsu. Możemy zatem przejść do programowania poleceń, które umieściliśmy w menu. Zajmiemy się najpierw operacjami na plikach, a więc wczytywaniem tekstu z pliku do edytora i zapisywaniem do pliku ewentualnych zmian.

Zgodnie z filozofią programowania zdarzeniowego skupimy się na obsłudze zdarzeń, jakie może wygenerować użytkownik — w naszym przypadku będzie to reakcja na wybór którejś z trzech pozycji menu: *Otwórz...*, która pozwoli na wczytanie obrazu ze wskazanego przez użytkownika pliku tekstowego, *Zapisz jako...*, która zapisze go do wskazanego pliku, oraz *Zamknij*, której wybranie zamknie naszą aplikację. Ponieważ obsługa ostatniej jest najprostsza, zaczniemy właśnie od niej.

Metoda zdarzeniowa zamykająca aplikację

Z pozycją *Zamknij* w menu *Plik* wiążemy metodę zdarzeniową zamykającą okno aplikacji.

1. W widoku projektowania na podglądzie formy kliknij dwukrotnie pozycję menu *Zamknij*.
2. W powstałej w ten sposób metodzie zdarzeniowej (zostaniesz automatycznie przeniesiony do edytora kodu) wpisz polecenie `Close();` (listing 9.1).

Listing 9.1. Trudno o prostszą metodę!

```
private void zamknijToolStripMenuItem_Click(object sender, EventArgs e)
{
    Close();
}
```

Wskazówka

Dwukrotne kliknięcie komponentu powoduje utworzenie metody zdarzeniowej związanej ze zdarzeniem domyślnym tego komponentu. W przypadku komponentów sterujących jest to zazwyczaj zdarzenie wywoływanie kliknięciem (przyciski) lub zmianą stanu (np. zmianą pozycji suwaka).

Polecenie `Close();` umieszczone w metodzie `zamknijToolStripMenuItem_Click` jest wywołaniem metody formy, a więc metody instancji klasy `Form1`. Metoda ta została zdefiniowana w jej klasie bazowej, tj. w `System.Windows.Forms.Form`. Skutkiem jej działania jest zamknięcie okna aplikacji. Zgodnie z filozofią systemu Windows zamknięcie głównego okna (w tym przypadku jedynego) oznacza także zamknięcie samej aplikacji. W ten sposób nasz cel zostaje osiągnięty w najprostszym z możliwych sposobów. Innym rozwiązaniem byłoby wykorzystanie polecenia `Application.Exit();`, które informuje bezpośrednio aplikację, że ma zamknąć wszystkie swoje okna, a następnie zakończyć działanie. Klasa `Application` jest zbiorem metod statycznych, które pozwalają zarządzać działaniem bieżącej aplikacji. Poza `Exit` są tam metody pozwalające kontrolować działanie wątków aplikacji, pętli obsługi komunikatów, a także metody pozwalające na pobranie informacji o aplikacji zapisanych w pliku `.exe`, ścieżki dostępu do tego pliku itp.

Potwierdzenie zamknięcia aplikacji

Zamykanie wszelkiego typu edytorów wiąże się z ryzykiem niezamierzonej utraty niezapisanych zmian w edytowanym dokumencie. Dlatego warto przed zamknięciem aplikacji poprosić użytkownika o potwierdzenie. Pozwala na to zdarzenie `FormClosing`, które wywoływanie jest w przypadku próby zamknięcia formy, ale jeszcze przed zdarzeniem `FormClose`, i które w odróżnieniu od tego drugiego pozwala na anulowanie procesu zamknięcia formy. Zatem przed zakończeniem działania aplikacji wyświetlimy komunikat z pytaniem, czy zapisać tekst lub może anulować zamknięcie.

1. Przejdź do widoku projektowania i utwórz metodę zdarzeniową do zdarzenia `FormClosing`.
2. Umieść w niej polecenie wyświetlające komunikat z odpowiednim pytaniem (listing 9.2). Nie dysponujesz jeszcze metodą zapisującą tekst do pliku, więc w odpowiednim miejscu metody `Form1_FormClosing` wstaw tymczasowe polecenie wyświetlające komunikat, który ma przypominać o uzupełnieniu kodu, ewentualnie zgłoszenie wyjątku `NotImplementedException`.

Listing 9.2. Zapisać czy zapomnieć?

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    DialogResult dr = MessageBox.Show("Czy zapisać zmiany w edytowanym dokumencie?",  

        this.Text,  

        MessageBoxButtons.YesNoCancel,  

        MessageBoxIcon.Question,  

        MessageBoxDefaultButton.Button3);  

    switch (dr)
    {
        case DialogResult.Yes: MessageBox.Show("Wstawić wywołanie metody zapisującej  

        ↴zawartość notatnika do pliku"); break;  

        case DialogResult.No: break;  

        case DialogResult.Cancel: e.Cancel = true; break;  

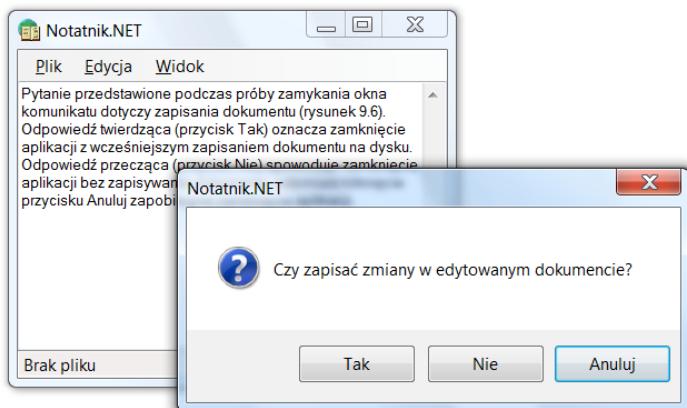
        default: e.Cancel = true; break;
    }
}
```

3. Pytanie o zachowanie dokumentu nie ma sensu, jeżeli nie został on zmieniony po otwarciu aplikacji lub po ostatnim zapisie na dysku. Dlatego warto w klasie `Form1` zdefiniować pole `tekstZmieniony` typu `bool`, którego wartość będzie ustalana na `true` w przypadku modyfikacji zawartości notatnika (pomocne będzie zdarzenie `TextChanged` komponentu `TextBox`). Jeżeli podczas zamknięcia formy jest ono równe `false`, komunikat z pytaniem nie zostanie pokazany.

Pytanie przedstawione podczas próby zamknięcia okna komunikatu dotyczy zapisania dokumentu (rysunek 9.6). Odpowiedź twierdząca (przycisk *Tak*) oznacza zamknięcie aplikacji z uprzednim zapisaniem dokumentu na dysku. Odpowiedź przecząca (przycisk *Nie*) spowoduje zamknięcie aplikacji bez zapisywania dokumentu, natomiast kliknięcie przycisku *Anuluj* zapobiegnie zamknięciu aplikacji.

Rysunek 9.6.

Każdy edytor powinien w taki sposób chronić użytkownika przed nieumyślną utratą danych

**Ukrywanie i pokazywanie paska stanu (polecenie w menu)**

Teraz zajmiemy się pozycją *Pasek stanu* z menu *Widok*. Wykorzystamy jej własność *Checked* w taki sposób, żeby użytkownik mógł zmieniać stan pozycji (oznaczony „ptaszkiem” dodawanym z lewej strony pozycji w menu) i w ten sposób uzyskał możliwość kontrolowania tego, czy pasek stanu na dole formy jest widoczny. Innymi słowy, przygotujemy metodę wiążącą stan właściwości *Visible* paska stanu z pozycją *Pasek stanu* w menu *Widok*.

1. Zamknij działającą aplikację i wróć do podglądu formy (klawisz F7, jeżeli aktualnie jesteś w edytorze kodu).
2. W podględzie formy zaznacz pozycję *Pasek stanu* z menu *Widok*. W oknie właściwości znajdź jej właściwość *CheckOnClick* i ustaw ją na *True*.
3. Następnie kliknij dwukrotnie tę pozycję w edytorze menu. Utworzysz w ten sposób metodę zdarzeniową, w której należy umieścić polecenie z listingu 9.3.

Listing 9.3. Zmiana stanu pozycji menu wymaga zmiany wartości jej właściwości *Checked*

```
private void pasekStanuToolStripMenuItem_Click(object sender, EventArgs e)
{
    statusStrip1.Visible = pasekStanuToolStripMenuItem.Checked;
}
```



Wskazówka Ustawiona na *True* właściwość *CheckOnClick* komponentu *ToolStripMenuItem* powoduje, że stan zaznaczenia pozycji w menu jest automatycznie przełączany po jej wybraniu. Zamiast tego zmianę zaznaczenia można wykonywać ręcznie w metodzie zdarzeniowej, korzystając z instrukcji postaci *pasekStanuToolStripMenuItem.Checked = !pasekStanuToolStripMenuItem.Checked;*.

4. Jeszcze raz wróć do podglądu formy i ponownie zaznacz pozycję *Pasek stanu* w menu *Widok*.
5. Za pomocą właściwości ustal wartość początkową właściwości *Checked* na *True* (lub równoważnie zmień wartość właściwości *CheckState* na *Checked*).

Tym razem sposób wykorzystania pozycji menu różni się nieco od poprzedniego przypadku, w którym służyła ona po prostu do uruchomienia metody. Teraz wykorzystaliśmy własność Checked, która odpowiada za umieszczenie z lewej strony menu symbolu zaznaczenia. Dzięki niemu pozycja *Pasek stanu* działa jak pole opcji, które może mieć jeden z dwóch stanów — zaznaczony lub nie. Od tego stanu zależy to, czy w oknie aplikacji widoczny jest pasek stanu.

Odczytywanie tekstu z pliku

Użyty w projekcie notatnika komponent TextBox nie posiada żadnych metod pozwalających na łatwe czytanie lub zapisywanie zawartości do plików tekstowych. Musimy zatem sami przygotować odpowiednie metody. Szczególnie potrzebujemy metody wczytującej plik tekstowy do tablicy łańcuchów (każdy łańcuch będzie osobnym akapitem tekstu widocznego w edytorze).

1. Przejdź do edytora kodu (*F7*).
2. Do zbioru deklaracji przestrzeni nazw na początku pliku dodaj:

```
using System.IO;
```
3. W klasie Form1 umieść kod metody widocznej na listingu 9.4.

Listing 9.4. Metoda kopiąjąca zawartość pliku tekstowego do tablicy łańcuchów

```
public static string[] CzytajPlikTekstowy(string nazwaPliku)
{
    List<string> tekst=new List<string>();
    try
    {
        using (StreamReader sr = new StreamReader(nazwaPliku))
        {
            string wiersz;
            while ((wiersz = sr.ReadLine()) != null)
                tekst.Add(wiersz);
        }
        return tekst.ToArray();
    }
    catch (Exception e)
    {
        MessageBox.Show("Błąd odczytu pliku " + nazwaPliku + " (" + e.Message + ")");
        return null;
    }
}
```

Do gromadzenia i przechowywania tekstu wewnętrz metody korzystamy z typu ogólnego (ang. *generics*) List sparametryzowanego typem string. Każdy łańcuch przechowuje osobny wiersz (akapit) odczytanego z pliku tekstu. W przypadku powodzenia (tj. braku wyjątków) metoda konwertuje ową listę na tablicę łańcuchów. Dzięki temu wartość zwracaną przez metodę będzie można wprost przypisać do właściwości Lines komponentu TextBox, która jest właśnie tablicą łańcuchów. Jeżeli jednak nie interesowałby nas podział na akapity (wiersze) wczytywanego tekstu, moglibyśmy go jednym poleceniem odczytać w całości, korzystając z metody ReadToEnd klasy StreamReader.

Należy zwrócić uwagę na użycie słowa kluczowego `using` w listingu 9.4. Konstrukcja, w której po słowie `using` w nawiasach znajduje się polecenie tworzące instancję klasy, zapewnia, że utworzony obiekt zostanie usunięty z pamięci po opuszczeniu zakresu wyznaczonego przez nawiasy klamrowe (operator zakresu), pomimo że jest typu referencyjnego.

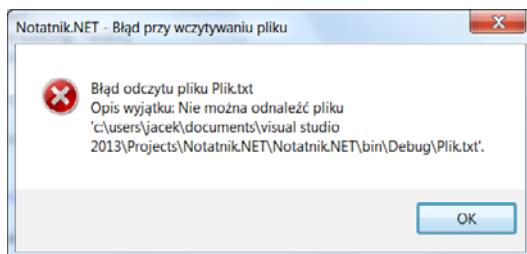
Warto także zwrócić uwagę na konstrukcję `try..catch`. Ograniczyliśmy się w niej tylko do jednej sekcji `catch` wyłapującej wszystkie wyjątki, wszystkie bowiem dziedziczą po klasie `Exception`. Moglibyśmy przechwycone wyjątki filtrować, umieszczając po sekcji `try` kolejne sekcje `catch` z np. wyjątkami `FileNotFoundException`, `UnauthorizedAccessException` itd. Pamiętać tylko należy, aby wyjątki najbardziej szczegółowe (najniższe w hierarchii dziedziczenia) umieszczać przed wyjątkami bardziej ogólnymi. Zauważmy także, że linia wyświetlająca nazwę pliku na pasku stanu została włączona do sekcji `try`. Unikamy w ten sposób sytuacji, w której byłaby ona wykonywana, gdy wcześniej zgłoszony został wyjątek. Należy bowiem pamiętać, że po zgłoszeniu wyjątku i wykonaniu poleceń z sekcji `catch` wątek nie wraca już do sekcji `try`, a wykonuje polecenia znajdujące się za sekcją `catch`, czyli w naszym przypadku metoda `button1_Click` kończy działanie.

Przygotowując tekst informacji wyświetlanej za pomocą metody `MessageBox.Show`, można korzystać ze znanych z C oraz C++ znaków formatujących, np. `\n` wstawia znak końca linii, a `\t` — znak tabulatora. Okno komunikatu może zawierać również tytuł oraz ikonę informującą o charakterze komunikatu (o tym, czy jest to błąd, ostrzeżenie, informacja). Aby tak było, należy podać dodatkowe argumenty metody `MessageBox.Show`, np.:

```
MessageBox.Show("Błąd odczytu pliku " + nazwaPliku + "\nOpis wyjątku: " + e.Message,
    "Notatnik.NET - Błąd przy wczytywaniu pliku",
    MessageBoxButtons.OK,
    MessageBoxIcon.Error);
```

Dodatkowe argumenty to kolejno: tytuł okna, rodzaj umieszczonych na oknie przycisków i wygląd ikony widocznej w oknie wiadomości. Po tych zmianach okno powinno wyglądać tak jak na rysunku 9.7.

Rysunek 9.7.
Pełnia możliwości
okna wiadomości



Wybór pliku za pomocą okna dialogowego

Przygotujemy teraz metodę pozwalającą na wybór pliku za pomocą standardowego okna dialogowego, a następnie przy użyciu przygotowanej w poprzednim ćwiczeniu metody pomocniczej wczytującą zawartość wybranego pliku do pola tekstowego.

1. Wróć do widoku projektowania, przełączając zakładkę edytora na *Form1.cs [Design]* lub naciskając klawisz F7.
2. W podoknie *Toolbox*, w grupie *Dialogs*, zaznacz komponent *OpenFileDialogs* i umieść go na podglądzie formy. Nowy obiekt *openFileDialog1* nie pojawi się jednak na samej formie, ale pod nią, na liście komponentów „niewizualnych”, razem z *menuStrip1* i *statusStrip1*.
3. Dwukrotnie kliknij pozycję menu *Otwórz...* w podglądzie menu, tworząc w ten sposób metodę zdarzeniową uruchamianą w przypadku wybrania tej pozycji menu w działającej aplikacji.
4. Do nowej metody wpisz kod z listingu 9.5.

Listing 9.5. Metoda związana z poleceniem Otwórz... w menu Plik

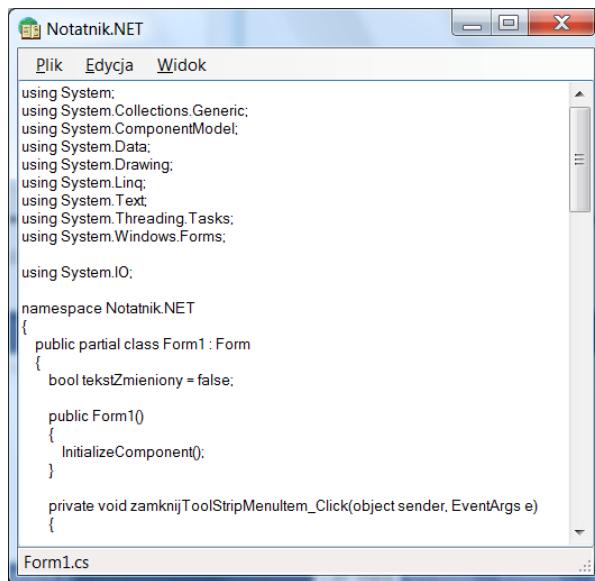
```
private void otwórzToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        string nazwaPliku=openFileDialog1.FileName;
        textBox1.Lines = CzytajPlikTekstowy(nazwaPliku);
        int ostatniSlash=nazwaPliku.LastIndexOf('\\');
        toolStripStatusLabel1.Text = nazwaPliku.Substring(ostatniSlash+1,
            ↵nazwaPliku.Length-ostatniSlash-1);
    }
}
```

5. Ustaw także kilka opcji okna dialogowego. Aby to zrobić, wróć do widoku projektowania, zaznacz komponent okna dialogowego *openFileDialog1* i korzystając z okna własności:
 - a) zmień tytuł okna, ustawiając właściwość **Title** np. na *Wczytaj plik*;
 - b) właściwość **DefaultExt** zmień na *txt* (jest to rozszerzenie dopisywane do nazwy pliku, jeżeli użytkownik żadnego rozszerzenia nie doda sam);
 - c) wyczyść zawartość pola przy właściwości **FileName**;
 - d) edytuj pole przy właściwości **InitialDirectory**, podając katalog domyślny równy katalogowi bieżącemu, tj. po prostu kropkę;
 - e) wpisz łańcuch określający filtry (właściwość **Filter**), np.: **Pliki tekstowe (*.txt)|*.txt|Pliki INI (*.ini)|*.ini|Pliki źródłowe (*.cs)|*.cs| Wszystkie pliki (*.*)|*.***. Uważaj, aby przy znakach | nie wstawić spacji.

Po wprowadzeniu powyższych zmian i komplikacji projektu warto sprawdzić, czy udało nam się wczytać jakiś plik tekstowy, choćby plik *Form1.cs* z kodem źródłowym C# (rysunek 9.8).

W Visual Studio nie ma porządkowych edytorów pozwalających na wygodne tworzenie filtrów okien dialogowych (punkt 5e). Do dyspozycji mamy tylko zwykłe pole edycyjne. Właściwość **Filter** jest bowiem zwykłym łańcuchem, czyli zmienną typu **string**.

Rysunek 9.8.
Notatnik
z wczytanym plikiem



Fragment tego łańcucha definiujący jeden filtr składa się z dwóch segmentów oddzielonych znakiem | . W pierwszym określamy opis wyświetlany w oknie dialogowym, a w drugim — maskę filtru. Trochę zamieszania powoduje to, że kolejne filtry również oddzielane są znakiem | , przez co cały łańcuch traci na czytelności. Tym bardziej dotkliwy staje się brak wygodnego edytora tej własności.

Zapisywanie tekstu do pliku

Przygotujmy teraz metodę pomocniczą *ZapiszDoPlikuTekstowego* zapisującą wskazaną w argumencie tablicę łańcuchów do pliku tekstowego. Następnie w metodzie zdzieleniowej związanego z pozycją *Zapisz jako...* z menu *Plik* wykorzystamy tę metodę do zapisania do pliku zawartości komponentu *TextBox*.

1. W obrębie klasy *Form1* zdefiniuj metodę z listingu 9.6.

Listing 9.6. Metoda zapisująca zawartość tabeli łańcuchów do pliku tekstowego

```

public static void ZapiszDoPlikuTekstowego(string nazwaPliku, string[] tekst)
{
    using (StreamWriter sw = new StreamWriter(nazwaPliku))
    {
        foreach (string wiersz in tekst)
            sw.WriteLine(wiersz);
    }
}

```

2. W widoku projektowania umieść na formie komponent *SaveFileDialog* z zakładki *Dialogs* podokna *Toolbox*.
3. Zaznacz go i w oknie własności na zakładce *Properties*:
 - a) zmień właściwość *Title* na *Zapisz do pliku*;

- b) DefaultExt na *txt*;
- c) odnajdź jego właściwość Filter i w związanym z nią polu wpisz łańcuch definiujący filtr: Pliki tekstowe (*.txt)|*.txt|Pliki INI (*.ini)|*.ini| Pliki źródłowe (*.cs)|*.cs| Wszystkie pliki (*.*)|*.*;
- d) zmień InitialDirectory na kropkę.
4. Utwórz metodę zdarzeniową do pozycji *Zapisz jako...* w menu *Plik* i umieść w niej kod z listingu 9.7.

Listing 9.7. Metoda zdarzeniowa związana z poleceniem *Zapisz jako* z menu *Plik*

```
private void zapiszJakoToolStripMenuItem_Click(object sender, EventArgs e)
{
    string nazwaPliku = openFileDialog1.FileName;
    if (nazwaPliku.Length > 0) saveFileDialog1.FileName = nazwaPliku;
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
    {
        nazwaPliku = saveFileDialog1.FileName;
        ZapiszDoPlikuTekstowego(nazwaPliku, textBox1.Lines);
        int ostatniSlash = nazwaPliku.LastIndexOf('\\');
        toolStripStatusLabel1.Text = nazwaPliku.Substring(ostatniSlash + 1,
            ↴nazwaPliku.Length - ostatniSlash - 1);
    }
}
```

5. Teraz możesz uzupełnić metodę *Form1_FormClosing* wywołaniem powyższej metody z argumentami równymi *null* (należy je wstawić zamiast tymczasowego komunikatu wyróżnionego w listingu 9.2):

```
case DialogResult.Yes: zapiszJakoToolStripMenuItem_Click(null, null); break;
```

Na początku metody *zapiszJakoToolStripMenuItem_Click* podejmowana jest próba pobrania z właściwości *FileName* obiektu *openFileDialog* ścieżki dostępu do pliku, z którego ewentualnie wcześniej wczytany był tekst. Odczytanie łańcucha z tej właściwości może zwrócić łańcuch pusty jedynie wtedy, gdy wcześniej nie został wczytany żaden plik. Odczytana ścieżka pliku przypisywana jest właściwości *saveFileDialog1.FileName*. Ciekawe jest, że komponent sam rozpoznaje w ścieżce pliku właściwy katalog i zastępuje nim katalog domyślny z właściwości *InitialDirectory*. W efekcie okno dialogowe pozwalające na wybór pliku do zapisania pokazuje katalog, z którego wybrany został plik do wczytania.

Okna dialogowe wyboru czcionki i koloru

Przygotujmy teraz metody związane z pozycjami menu *Widok*, które będą pozwalać na wybór koloru tła notatnika i użytej w nim czcionki. Ponownie użyjemy standardowych okien dialogowych. W przypadku czcionki umożliwimy także wybór jej koloru.

1. Na podglądzie formy umieść okno dialogowe *ColorDialog* z zakładki *Dialogs*.
2. Następnie dla pozycji *Tło...* w menu *Widok* utwórz metodę zdarzeniową z poleceniami widocznymi na listingu 9.8.

Listing 9.8. Metoda zdarzeniowa wywoływana po kliknięciu pozycji Czcionka w menu Widok

```
private void tłoToolStripMenuItem_Click(object sender, EventArgs e)
{
    colorDialog1.Color = textBox1.BackColor;
    if (colorDialog1.ShowDialog() == DialogResult.OK)
        textBox1.BackColor = colorDialog1.Color;
}
```

3. W podobny sposób możesz przygotować metodę pozwalającą na wybór czcionki i jej koloru (listing 9.9), korzystającą z komponentu FontDialog.
4. Po umieszczeniu tego komponentu na podglądzie okna ustaw jego własność ShowColor na True.
5. Następnie w metodzie zdarzeniowej związanej ze zdarzeniem Click pozycji Czcionka... z menu Widok umieść polecenia widoczne na listingu 9.9.

Listing 9.9. Metoda zdarzeniowa wywoływana po kliknięciu pozycji Czcionka w menu Widok

```
private void czcionkaToolStripMenuItem_Click(object sender, EventArgs e)
{
    fontDialog1.Font = textBox1.Font;
    fontDialog1.Color = textBox1.ForeColor;
    if (fontDialog1.ShowDialog() == DialogResult.OK)
    {
        textBox1.Font = fontDialog1.Font;
        textBox1.ForeColor = fontDialog1.Color;
    }
}
```

W obu metodach polecenia sprzed instrukcji `if` uzgadniają początkową wartość przedstawianą w oknie dialogowym z bieżącą wartością odpowiednich właściwości komponentu `textBox1`. Dzięki temu okno dialogowe pozwala na modyfikowanie bieżącej wartości, zamiast wymuszać za każdym razem określanie jej od nowa.

Edycja i korzystanie ze schowka

Kolejna grupa poleceń związana jest z menu *Edycja*. Ich obsługa okazuje się łatwa, ponieważ komponent `TextBox` posiada odpowiednie metody, co nasze zadanie sprowadza do ich właściwego wywołania. Tworzymy wobec tego metody zdarzeniowe do wszystkich pozycji z menu *Edycja* i odpowiednio wywołujemy w nich metody widoczne na listingu 9.10.

Listing 9.10. Zbiór metod związanych z pozycjami menu *Edycja*

```
private void cofnijToolStripMenuItem_Click(object sender, EventArgs e)
{
    textBox1.Undo();
}

private void wytnijToolStripMenuItem_Click(object sender, EventArgs e)
{
    textBox1.Cut();
```

```
}

private void kopiujToolStripMenuItem_Click(object sender, EventArgs e)
{
    textBox1.Copy();
}

private void wklejToolStripMenuItem_Click(object sender, EventArgs e)
{
    textBox1.Paste();
}

private void usuńToolStripMenuItem_Click(object sender, EventArgs e)
{
    textBox1.SelectedText = "";
}

private void zaznaczwszystkoToolStripMenuItem_Click(object sender, EventArgs e)
{
    textBox1.SelectAll();
}
```

Zwróćmy uwagę na to, że zawartość utworzonego podmenu *Edycja* pokrywa się przynajmniej częściowo z zawartością menu kontekstowego domyślnie przypisanego do pola tekstopowego. Wywoływane z obu menu metody obiektu textBox1 też są tożsame.

Drukowanie

Ostatnią rzeczą, jaką zaimplementujemy w naszym notatniku, jest drukowanie edytowanego tekstu. Podobnie jak w przypadku operacji na plikach, tak i tu nie możemy liczyć na wygodną w użyciu metodę, która czarną robotę wykonałaby za nas. To oznacza, że i tym razem odpowiedni kod musimy napisać samodzielnie. Za to po przygotowaniu metody organizującej wydruk stron niemal za darmo uzyskamy podgląd wydruku.

Do drukowania w trybie graficznym należy użyć klasy *Graphics* — jej instancja będzie reprezentowała powierzchnię drukowanej kartki³. Zrobimy użytkowi z metody *DrawString*, która pozwala na umieszczenie tekstu w dowolnym miejscu reprezentowanej przez klasę powierzchni (kartki). W przypadku drukowania inne metody (*DrawText* i *TextRender*) — niestety — się do tego nie nadają.

Polecenia związane z drukowaniem w menu Plik

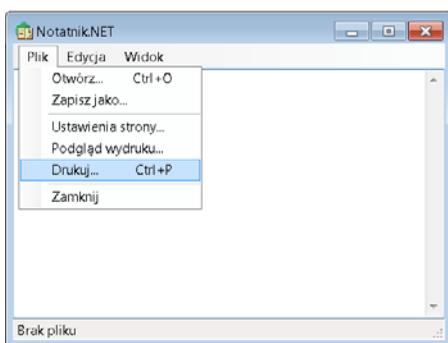
Zacznijmy jednak od uzupełnienia naszego menu *Plik* o pozycje związane z drukowaniem.

1. Przejdź do widoku projektowania.
2. Zaznacz komponent *menuStrip1*.
3. W widocznym wówczas na podglądzie formy edytorze menu zaznacz pozycję *Zamknij* w podmenu *Plik* i z menu kontekstowego wybierz *Insert/Menu Item*.

³ Klasę tę wykorzystuje się także do rysowania na powierzchni formy. Ilustrujący to projekt o nazwie *Dywany graficzny* został opisany na końcu tego rozdziału.

- 4.** Kliknij dodany element i zmień jego etykietę na *&Ustawienia strony...* (rysunek 9.9).

Rysunek 9.9.
Menu Plik uzupełniamy o sekcję zawierającą polecenia związane z drukowaniem



- 5.** Powtórz czynności z punktów 3. i 4., aby dodać polecenia *Podgląd wydruku...* oraz *&Drukuj....*
- 6.** Ustaw klawisz skrótu tego ostatniego na *Ctrl+P*.
- 7.** Za polem *Drukuj* wstaw separator.
- 8.** Tak dodane komponenty mają nazwy *toolStripMenuItem_n*, gdzie *n* to kolejna liczba naturalna. Jeżeli chcesz nadać im bardziej przyjazne nazwy, zaznacz je w widoku projektowania, przejdź do okna własności i zmień ich nazwy, korzystając z właściwości (Name). Użyj nazw *ustawieniaStronyToolStripMenuItem*, *podgladWydrukuToolStripMenuItem* i *drukujToolStripMenuItem*.

Warto zwrócić uwagę, że wywołana w punkcie 3. gałąź *Insert* menu kontekstowego zawiera nie tylko pozycję *Menu Item*, ale również rozwijaną listę i pole tekstowe. Takie komponenty możemy również umieścić w menu głównym aplikacji.

Metoda zarządzająca drukowaniem tekstu

Kolejną czynnością będzie dodanie do aplikacji komponentu *PrintDocument* reprezentującego drukowany dokument i wykorzystanie jego zdarzenia *PrintPage* do wydrukowania zawartości pola tekstopowego. Do inicjacji wydruku wykorzystamy okno dialogowe udostępniane przez kontrolkę *PrintDialog*.

- 1.** Na podglądzie formy umieść komponent *PrintDocument* z zakładki *Printing*.
2. Zdefiniuj prywatne pole:

```
private StringReader sr = null;
```

- 3.** W widoku projektowania interfejsu zaznacz obiekt *printDocument1*, w oknie własności przejdź na zakładkę zdarzeń i utwórz metodę związaną ze zdarzeniem *PrintPage*⁴. Umieść w niej pętlę przenoszącą zawartość pola tekstopowego linia po linii na powierzchnię drukowanej strony (listing 9.11).

⁴ Zdarzenie *PrintPage* pełni podobną rolę do zdarzenia *Paint* w przypadku formy. Zobacz projekt *Dywany graficzny* w dalszej części rozdziału.

Listing 9.11. Metoda odpowiedzialna za przygotowanie wydruku poszczególnych stron tekstu

```
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    Font czcionka = textBox1.Font;
    int wysokoscWiersza=(int)czcionka.GetHeight(e.Graphics);
    int iloscLinii=e.MarginBounds.Height/wysokoscWiersza;

    if (sr == null) sr = new StringReader(textBox1.Text); //czy pierwsza strona?
    e.HasMorePages = true;

    for(int i=0;i<iloscLinii;i++)
    {
        string wiersz = sr.ReadLine();
        if (wiersz == null)
        {
            e.HasMorePages = false;
            sr = null;
            break;
        }
        e.Graphics.DrawString(wiersz,
                             czcionka,
                             Brushes.Black,
                             e.MarginBounds.Left,           //x
                             e.MarginBounds.Top+i*wysokoscWiersza); //y
    }
}
```



Dopuszciliśmy się tu poważnego uproszczenia, zakładając, że żadna linia tekstu nie jest dłuższa niż szerokość kartki. To założenie jest zwykle nieprawdziwe i dlatego za chwilę do tego problemu wróćmy.

4. Teraz pozostaje uruchomić proces drukowania. W tym celu:

- a) na podglądzie formy umieść komponent PrintDialog z zakładki *Printing*;
- b) za pomocą okna *Properties* powiąż jego własność Document z obiektem printDocument1;
- c) utwórz metodę zdarzeniową do pozycji *Drukuj...* w menu *Plik* i umieść w niej polecenia z listingu 9.12.

Listing 9.12. Metoda zdarzeniowa związana z polecienniem Drukuj z menu Plik

```
private void drukujToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (printDialog1.ShowDialog() == DialogResult.OK)
    {
        printDocument1.DocumentName = "Notatnik.NET - " + toolStripStatusLabel1.Text;
        printDocument1.Print();
    }
}
```

Komentarza wymaga metoda zdefiniowana w punkcie 3. Przede wszystkim należy zauważyc, że zdarzenie, z którym jest związana, a więc printDocument1.PrintPage, jest wywoływanie w momencie drukowania pojedynczej strony. Użytkownik sam musi zadbać o to, co będzie na tej stronie umieszczone. Co więcej, ustalając wartość e.HasMorePages, sam musi poinformować system, czy w wydruku będzie więcej stron (czy metoda zdarzeniowa ma być uruchomiona ponownie). Komponent printDocument1 nie udostępnia żadnego pola, które informowałoby o tym, ile stron zostało już wydrukowanych. To pozostawia programistom pełną swobodę korzystania z drukarki, ale jednocześnie zmusza go do wykonania samodzielnie całej pracy związanej z organizacją wydruku strony.



Wskazówka

Jedno wywołanie metody zdarzeniowej printDocument1_PrintPage związane jest z wydrukowaniem jednej strony.

Metoda widoczna na listingu 9.11 korzysta ze strumienia `StringReader`, co pozwala uniknąć kłopotliwego wyliczania zakresu linii, które mają być drukowane na konkretnej stronie. Dzięki inicjacji strumienia przy drukowaniu pierwszej strony i zwalnianiu go dopiero przy ostatniej możemy pobierać z niego kolejne wiersze tekstu linia po linii bez obliczania, na której aktualnie stronie się znajdujemy.

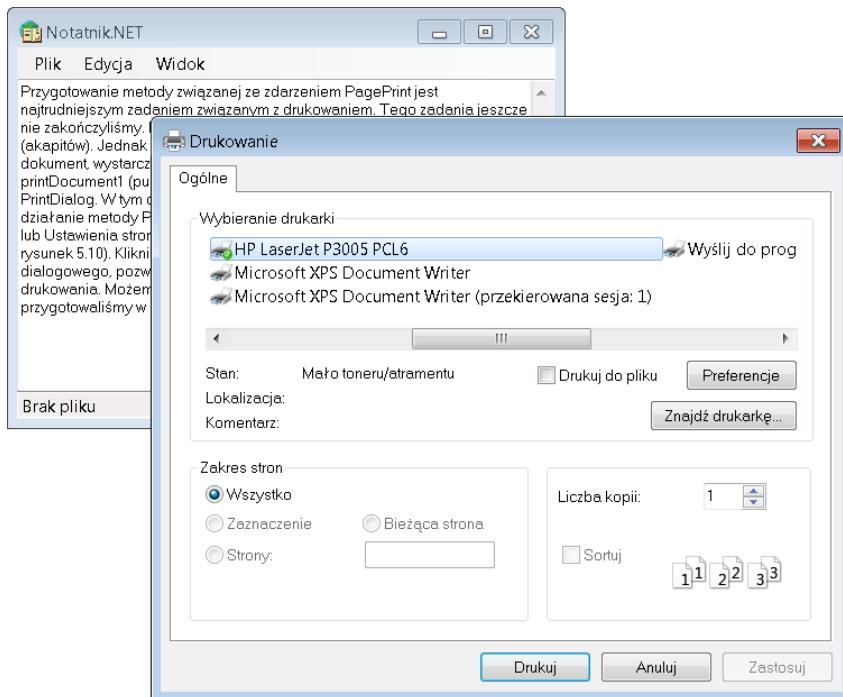
Wszystkie informacje o ustawieniach strony pobierane są z obiektu `e` typu `PrintPageEventArgs`. Jest to odpowiednik argumentu `PaintEventArgs` w zdarzeniu `Paint`. Oba zawierają obiekt `Graphics`, który w tym przypadku umożliwia umieszczanie na powierzchni strony kolejnych linii. Korzystając z metod tego obiektu, bez trudu można tam umieścić także obraz.

Przygotowanie metody związanej ze zdarzeniem `PagePrint` jest najtrudniejszym zadaniem związanym z drukowaniem. Tego zadania jeszcze nie zakończyliśmy. Nadal pozostaje do rozwiązania problem długich linii (akapitów). Jednak prowizoryczna wersja już działa. Aby wydrukować dokument, wystarczy teraz wywołać metodę `Print` komponentu `printDocument1` (punkt 4.). Korzystamy przy tym z okna dialogowego `PrintDialog`. W tym oknie poza przyciskami *Drukuj* i *Anuluj*, które kończą działanie metody `PrintDialog.ShowDialog`, jest także przycisk *Preferencje* lub *Ustawienia strony* (w zależności od wersji systemu operacyjnego, rysunek 9.10). Kliknięcie tego przycisku powoduje uruchomienie okna dialogowego pozwalającego na ustawienie drukarki i własności drukowania. Możemy również wywołać to okno samodzielnie — do tego celu przygotowaliśmy w menu *Plik* pozycję o etykiecie *Ustawienia strony*....

Okno dialogowe ustawień wydruku

Dodaj do formy komponent `PageSetupDialog` pozwalający na ustalenie przez użytkownika aplikacji sposobu, w jaki będą drukowane strony.

1. Na formie umieść komponent `PageSetupDialog` (powstanie obiekt `pageSetupDialog1`).
2. Jego własność `Document` powiąż z `printDocument1`.



Rysunek 9.10. Wygląd okna dialogowego drukowania zależy od używanej wersji systemu Windows

3. Utwórz metodę zdarzeniową do pozycji *Ustawienia strony...* z menu *Plik* i umieść w niej wywołanie metody ShowDialog komponentu pageSetupDialog1 (listing 9.13).

Listing 9.13. Metoda związana z pozycją *Ustawienia strony* menu *Plik*

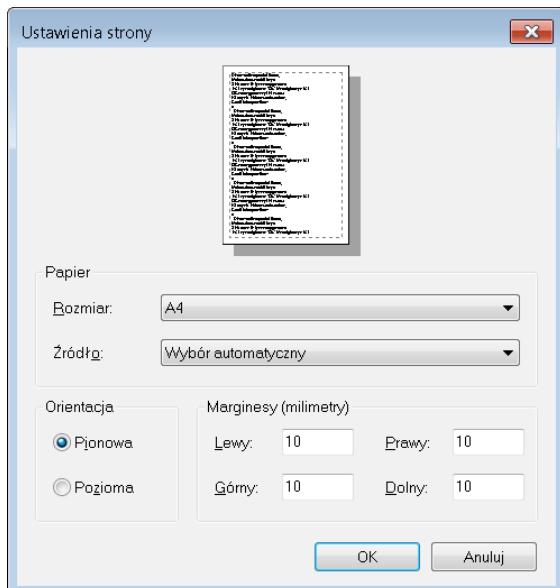
```
private void ustawieniaStronyToolStripMenuItem_Click(object sender, EventArgs e)
{
    pageSetupDialog1.ShowDialog();
}
```

Nowe okno dialogowe umożliwia zmianę ustawień drukowania związanych z komponentem printDocument1. Okno, które zobaczymy (rysunek 9.11), może zależeć nie tylko od wersji systemu, ale również od wybranej drukarki.

Podgląd wydruku

Jeżeli podołaliśmy trudowi przygotowania metody związanej ze zdarzeniem printDocument1.PagePrint, to niewielkim kosztem możemy uruchomić podgląd wydruku. Służy do tego okno dialogowe PrintPreviewDialog. Sam obszar podglądu, jeżeli chcemy go wkomponować w główne okno aplikacji, jest również dostępny dzięki komponentowi PrintPreviewControl. My jednak, głównie aby oszczędzić sobie kłopotu, skorzystamy z wariantu podglądu w osobnym oknie.

Rysunek 9.11.
Okno dialogowe
z ustawieniami
drukowania strony



1. Na formie umieść komponent PrintPreviewDialog (nie należy mylić go z PrintPreviewControl).
2. Jego własność Document ustaw na printDocument1.
3. Utwórz metodę zdarzeniową do pozycji *Podgląd wydruku...* z menu *Plik* i umieść w niej wywołanie metody ShowDialog komponentu printDocument1 (listing 9.14).

Listing 9.14. Metoda związana z pozycją *Podgląd wydruku* menu *Plik*

```
private void toolStripMenuItem3_Click(object sender, EventArgs e)
{
    printPreviewDialog1.ShowDialog();
}
```

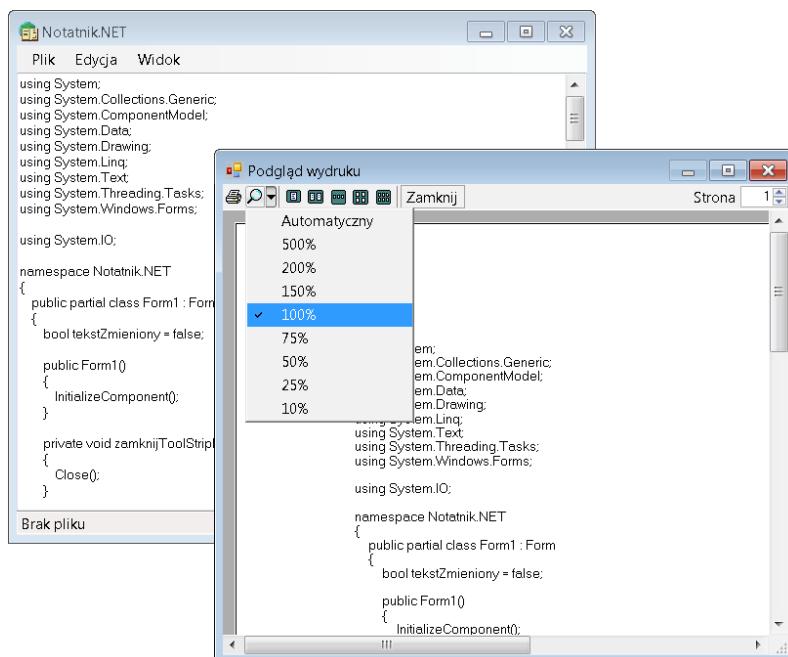
Po uruchomieniu notatnika i wybraniu pozycji *Podgląd wydruku...* powinniśmy zobaczyć okno podobne do widocznego na rysunku 9.12.

Rozwiązywanie problemu długich linii

Nie można już dłużej odkładać problemu długich linii. Co zrobić, aby pokazywane były w całości w wielu liniach? Bardzo kłopotliwe byłoby ich dzielenie w trakcie drukowania, szczególnie jeśli pojawią się na końcu strony. Najbardziej dogodnym momentem na podział linii wydaje się chwila, w której tworzony jest strumień reprezentujący zawartość pola tekstowego, tj. linia if (sr == null) sr = new StringReader ↵(textBox1.Text);. W tym celu musimy zastąpić argument konstruktora klasy String ↵Reader, a więc łańcuch odczytany z właściwości textBox1.Text, łańcuchem, w którym długie linie będą podzielone. Modyfikujemy wobec tego metodę printDocument1_ ↵PrintPage zgodnie z listingiem 9.15.

Rysunek 9.12.

Podgląd wydruku w osobnym oknie



Listing 9.15. Metoda odpowiedzialna za wydruk po zmianach związanych z drukowaniem długich linii

```
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    Font czcionka = textBox1.Font;
    int wysokoscWiersza=(int)czcionka.GetHeight(e.Graphics);
    int iloscLinii=e.MarginBounds.Height/wysokoscWiersza;

    //przy dzieleniu linii nie dbamy o spacje
    if (sr == null)
    {
        string tekst = "";
        foreach (string wiersz in textBox1.Lines)
        {
            float szerokosc=e.Graphics.MeasureString(wiersz, czcionka).Width;
            if (szerokosc<e.MarginBounds.Width)
            {
                tekst+=wiersz+"\n";
            }
            else
            {
                float sredniaSzerokoscLitery = szerokosc / wiersz.Length;
                int ileLITERWWierszu=(int)(e.MarginBounds.Width/
                ↳sredniaSzerokoscLitery);
                int ileRazy = wiersz.Length/ileLITERWWierszu; //ile pełnych wierszy
                for (int i=0;i<ileRazy;i++)
                {
                    tekst += wiersz.Substring(i*ileLITERWWierszu,ileLITERWWierszu)+"\n";
                }
                //kopiowanie reszty wiersza
            }
        }
    }
}
```

```
        tekst += wiersz.Substring(ileRazy * ileLiterWWierszu) + "\n";
    } //if-else
} //foreach
sr = new StringReader(tekst);
} //if(sr==null)

e.HasMorePages = true;

...// dalszy ciąg metody
```

W zmodyfikowanym fragmencie obliczamy szerokość odczytanego z pola tekstopwego wiersza (akapitu). Jeżeli jest mniejsza od szerokości strony, kopujemy ją do łańcucha tekstu i przechodzimy do kolejnego. Natomiast jeśli jest większa, szacujemy liczbę liter w wierszu i dzielimy go na równe fragmenty. Następnie każdy z nich umieszczamy w zmiennej tekst jako osobną linię.

Podział linii uwzględniający wyrazy

Rozwiążanie problemu podziału długich linii nadal nie jest ostateczne ze względu na jego oczywistą wadę — ignoruje podział linii na wyrazy. Ale i temu można zaradzić. W podziale linii na wydruku przygotowanym w poprzednim ćwiczeniu uwzględnijmy integralność wyrazów. W tym celu jeszcze raz modyfikujemy metodę printDocument1_PrintPage (wyróżnienia na listingu 9.16).

Listing 9.16. Kolejne modyfikacje metody printDocument1_PrintPage

```
    tekst += skracanyWiersz.Substring(0, iloscLiter) + "\n";
    skracanyWiersz = skracanyWiersz.Substring(iloscLiter);
    ↳TrimStart(' ');
} while (skracanyWiersz.Length > ileLiterWWierszu);
tekst += skracanyWiersz + "\n"; //ostatnia część
} //if-else
} //foreach
sr = new StringReader(tekst);
} //if(sr==null)

e.HasMorePages = true;

//dalsza część metody
```

Po tych zmianach zamiast dzielić akapit na fragmenty o stałej liczbie znaków, szukamy ostatniej spacji w wydzielonym bloku i łamiemy linię w tym miejscu. Dzięki tej sztuczce podział linii powinien być teraz bardziej zadowalający.

Drukowanie w tle

Ostatnim zadaniem będzie uruchomienie procesu drukowania w taki sposób, aby nie blokować interfejsu całego programu. Oznacza to, że musimy uruchomić metodę `printDocument1.Print` w osobnym wątku. Do zrealizowania tego zadania wykorzystamy kolejny niewizualny komponent o nazwie `BackgroundWorker`. Jego użycie zwalnia programistę z konieczności samodzielnego tworzenia wątku; zadanie programisty ogranicza się do umieszczenia wszystkich poleceń wykonywanych w osobnym wątku w metodzie związanej ze zdarzeniem `BackgroundWorker.DoWork`. W naszym przypadku będzie to jedynie polecenie `printDocument1.Print`. Metodę tę można uruchomić w osobnym wątku poleciением `backgroundWorker1.RunWorkerAsync()`.

1. Na podglądzie formy umieść komponent `BackgroundWorker` z palety *Components*.
2. Klikając dwukrotnie ikonę nowego komponentu na pasku pod podglądem formy w widoku projektowania, utwórz jego domyślną metodę zdarzeniową.
3. Następnie umieść w niej polecenie drukowania (listing 9.17).

Listing 9.17. Metoda zawierająca polecenia wykonywane w osobnym wątku

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    printDocument1.Print();
}
```

4. Następnie w metodzie zdarzeniowej `drukujToolStripMenuItem_Click` związanej z poleceniem *Drukuj...* z menu *Plik* zastąp polecenie drukowania uruchomieniem „`backgroundworker1`” (listing 9.18).

Listing 9.18. Zmodyfikowana metoda zarządzająca drukowaniem

```
private void drukujToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (printDialog1.ShowDialog() == DialogResult.OK)
    {
```

```
    printDocument1.DocumentName = "Notatnik.NET - " + toolStripStatusLabel1.Text;
    printDocument1.Print();
    backgroundWorker1.RunWorkerAsync();
}
}
```

Jak wspomniałem, zamiast komponentu BackgroundWorker można również użyć tradycyjnej klasy Thread implementującej wątki w platformie .NET lub klasy zadania Task z biblioteki TPL. W tym drugim przypadku można zmodyfikować metodę z listingu 9.18 w sposób pokazany poniżej (listing 9.19).

Listing 9.19. Drukowanie w tle wymaga utworzenia dodatkowego wątku

```
private void drukujToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (printDialog1.ShowDialog() == DialogResult.OK)
    {
        printDocument1.DocumentName = "Notatnik.NET - " + toolStripStatusLabel1.Text;
        printDocument1.Print();
        Task.Run((Action)printDocument1.Print);
    }
}
```



Bogatsza wersja projektu Notatnik.NET, w której m.in. dodano pasek narzędzi, ikony w menu, możliwość przeszukiwania i zastępowania tekstu czy szybkie wyszukiwanie, znajduje się w kodach źródłowych dołączonych do książki (zobacz też zadania).

Elektroniczna kukułka

Interfejs następnej aplikacji będzie ograniczony do minimum. Główne okno, widoczne jedynie przez chwilę po uruchomieniu aplikacji, będzie pełniło rolę wizytówka (ang. *splash screen*). W zamian w zasobniku systemowym umieścimy ikonę aplikacji i zwiążemy z nią menu kontekstowe zawierające m.in. pozycję pozwalającą na zamknięcie aplikacji. A coż ma ta aplikacja robić? Będzie powiadamiać o mijaniu pełnych godzin za pomocą odtwarzanego z pliku .wav sygnału dźwiękowego. Do tego wykorzystamy klasy z przestrzeni nazw System.Media, a szczególnie komponent SoundPlayer.

Ekran powitalny (splash screen)

Ekran powitalny (ang. *splash screen*) możemy przygotować na podstawie obrazu w formacie .bmp, .gif, .jpeg, .png, .wmf lub .emf, który przypiszemy do formy, korzystając z właściwości BackgroundImage (obraz tła). Możemy go także zbudować z komponentów typu Label. To drugie rozwiązanie opiszę poniżej.

Projekt okna wizytówki aplikacji

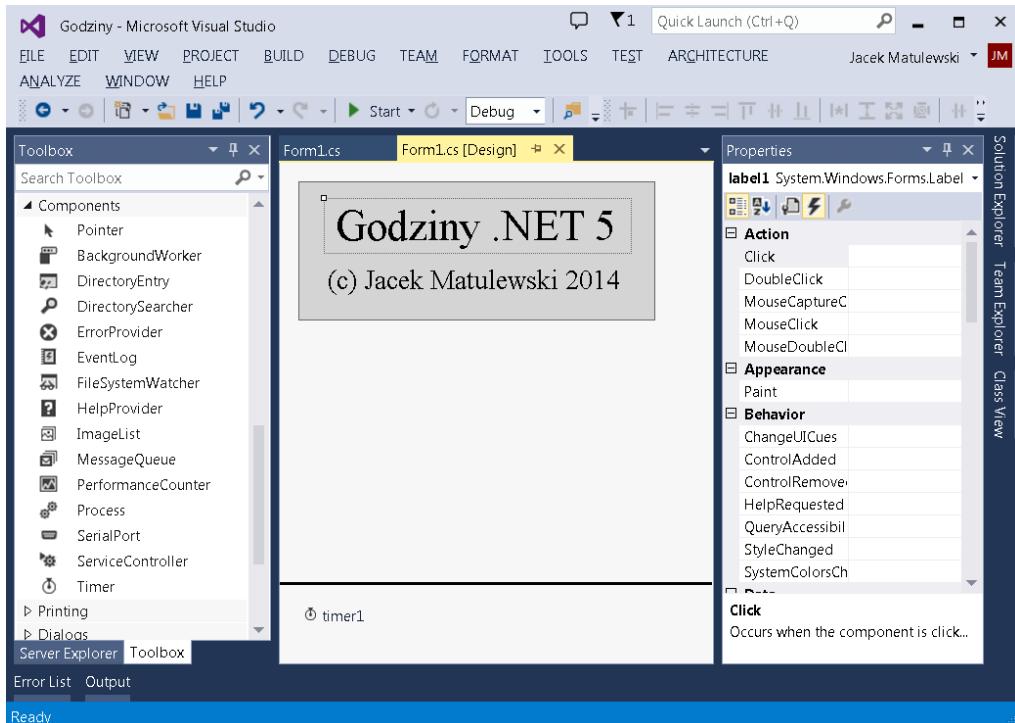
Przygotujmy formę zawierającą tytuł programu i informacje o autorze, czyli tzw. wizytówkę programu.

1. Zapisz (*Ctrl+Shift+S*) i zamknij bieżący projekt (menu *File, Close Solution*). Następnie utwórz nowy projekt typu Windows Forms Application, o nazwie *Godziny*.
2. W widoku projektowania zaznacz podgląd formy i za pomocą okna własności zmień styl i tło formy:
 - a) jej własność *FormBorderStyle* (grupa *Appearance*) zmień na *None*;
 - b) własność *StartPosition* (grupa *Layout*) ustaw na *CenterScreen*;
 - c) aby zapobiec pokazywaniu okna w pasku zadań, zmień własność *ShowInTaskbar* (grupa *Window Style*) na *False*.
3. Na formie umieść komponent *Panel*, a następnie skonfiguruj jego własności:
 - a) *BorderStyle* na *FixedSingle*;
 - b) kolory tła (*BackColor*) na jasnokoralowy (*LightGray*), a ramki (*ForeColor*) na czarny (*Black*);
 - c) własność *Dock* na *Fill* (rysunek 9.1).
4. Na panelu umieść dwa komponenty *Label* z tytułem aplikacji i notą praw autorskich (rysunek 9.13):
 - a) korzystając z edytora własności *Font*, ustal wielkość czcionki pierwszej etykiety na 26, a drugiej na 16;
 - b) krój czcionki (*Font.Name*) zmień na *Times New Roman*;
 - c) kolor napisów obu etykiet (własność *ForeColor*) zmień na czarny (*Black*);
 - d) skorzystaj z własności *AutoSize* (grupa *Layout*) komponentów *Label* ustawionej na *True* (wartość domyślna), aby nie martwić się o to, czy cały tekst etykiety zmieści się w zarezerwowanym dla komponentu obszarze.

Ukrywanie okna po dwóch sekundach od uruchomienia aplikacji

Aby forma pełniła rolę prawdziwego *splash screenu*, powinna zniknąć po kilku sekundach od utworzenia. Możemy ten efekt łatwo uzyskać, korzystając z komponentu *Timer*.

1. Umieść komponent *Timer* z zakładki *Components* na formie (jest to kolejny komponent niewizualny, wobec tego w widoku projektowania pojawi się w specjalnym obszarze pod nią).
2. Jego własność *Interval* ustal na np. 2000 milisekund (2 sekundy).
3. Aktywuj go, ustawiając własność *Enabled* na *True*.
4. Następnie dwukrotnie kliknij komponent w widoku projektowania, aby utworzyć domyślną metodę zdarzeniową do jego jedynego zdarzenia *Tick*. Umieść w niej dwa polecenia: ukrywające formę i dezaktywujące obiekt *timer1* (listing 9.20).



Rysunek 9.13. Przykładowy wygląd wizytówki

Listing 9.20. Ukrycie okna i wyłączenie timera

```
private void timer1_Tick(object sender, EventArgs e)
{
    Hide();
    timer1.Enabled=false;
}
```

5. W rozwijanej liście okna właściwości wybierz obiekt Form1. Przejdź do listy zdarzeń i zwiąż zdarzenie Click z metodą `time1_Tick`. Dzięki temu będziesz mógł przyspieszyć zniknięcie okna powitalnego, klikając je.
6. Następnie zaznacz wszystkie komponenty na formie (tj. obie etykiety) i również powiąż ich zdarzenie Click z tą metodą. To na wypadek, gdyby zamiast formy ktoś kliknął jeden z umieszczonej na niej napisów.
7. Skompiluj projekt bez jego uruchamiania, naciskając klawisz `Ctrl+Shift+B` lub `F6`.

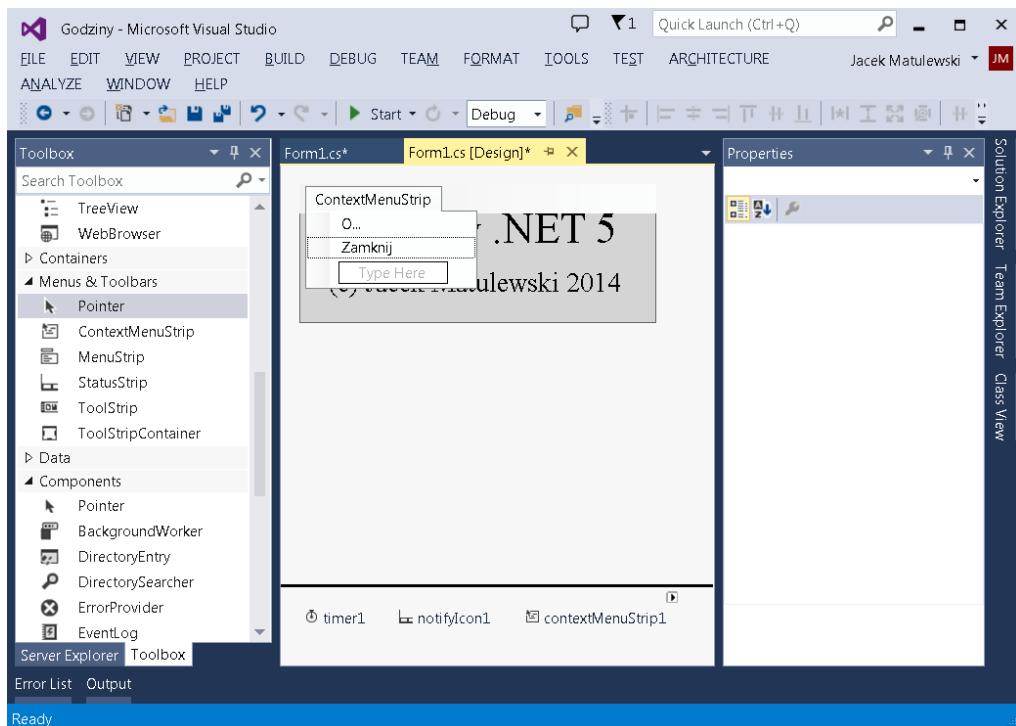
W efekcie powstało typowe okno powitalne. Możemy je przetestować już teraz (`F5`), jednak po zniknięciu okna jedynym sposobem, aby zamknąć aplikację, jest kliknięcie przycisku *Stop Debugging* z paska narzędzi Visual Studio lub użycie kombinacji klawiszy `Shift+F5`. Proponuję zatem, żeby z testowaniem aplikacji wstrzymać się do momentu przygotowania menu ikony, którą umieścimy w zasobniku.

Przygotowanie ikony w obszarze powiadamiania

Ikona w zasobniku (obszarze powiadamiania)⁵ i jej menu

Dodaj obiekt NotifyIcon do aplikacji i wyposaż go w menu kontekstowe rozwijane po kliknięciu prawym przyciskiem myszy.

1. Przejdź do widoku projektowania.
2. Na formie umieść dwa komponenty niemające swojej reprezentacji na formie: NotifyIcon (zakładka *Common Controls*) i ContextMenuStrip (zakładka *Menus & Toolbars*).
3. Zaznacz komponent contextMenuStrip1. Na podglądzie formy pojawi się prosty edytor menu (rysunek 9.14), w którym wpisz dwie pozycje: *O...* i *Zamknij*. Nazwy pozycji można edytować w bardzo podobny sposób, jak wcześniej edytowaliśmy menu główne, tj. wpisując je wprost w edytorze, w miejscu oznaczonym przez *Type here*.



Rysunek 9.14. Edytor menu kontekstowego jest bardzo podobny do edytora menu głównego

⁵ Zasobnik, który bywa również nazywany obszarem powiadamiania, to miejsce w pobliżu zegara, znajdujące się domyślnie w prawym dolnym rogu ekranu. W Windows 7 i 8 nie wszystkie ikony pokazywane są na ekranie. Część z nich jest ukryta i aby je zobaczyć, należy kliknąć ikonę *Pokaż ukryte ikony*.

- 4.** Kliknij dwukrotnie pierwszą pozycję menu, tworząc w ten sposób domyślną metodę zdarzeniową (do zdarzenia Click). Umieść w niej polecenie pokazujące formę, czyli nasz *splash screen*, i ponownie aktywujące timer1 (listing 9.21), który ukryje formę po dwóch sekundach.

Listing 9.21. Przywrócenie okna i włączenie timera, które je ukryje

```
private void oToolStripMenuItem_Click(object sender, EventArgs e)
{
    Show();
    timer1.Enabled = true;
}
```

- 5.** W ten sam sposób utworzysz metodę uruchamianą po wybraniu pozycji *Zamknij*. W niej umieść wywołanie metody Close zamkającej aplikację (listing 9.22).

Listing 9.22. Metoda zamkająca aplikację

```
private void zamknijToolStripMenuItem_Click(object sender, EventArgs e)
{
    notifyIcon1.Visible = false;
    Close();
}
```

- 6.** Następnie zaznacz komponent notifyIcon1; jego własność Visible powinna być ustawiona na True.
- 7.** Aby ikona była widoczna w zasobniku, musisz wczytać jeszcze jej obraz z pliku .ico. Umożliwia to edytor właściwości Icon dostępny z podokna *Properties*. Przykładowa ikona dostępna jest w źródłach dołączonych do książki.
- 8.** Ostatnią czynnością jest związanie (za pomocą okna właściwości) menu kontekstowego, a więc obiektu contextMenuStrip1, z właściwością ContextMenuStrip obiektu notifyIcon1.

Teraz możemy uruchomić aplikację bez obawy, że nie będzie można jej w naturalny sposób zamknąć. Na zamknięcie aplikacji pozwala teraz pozycja *Zamknij* w menu, które możemy rozwinąć, klikając prawym przyciskiem myszy ikonę widoczną w zasobniku (rysunek 9.15).

Rysunek 9.15.

Ikona aplikacji w zasobniku i związanego z nią menu



Tekst podpowiedzi ikony w zasobniku

Zabawnie będzie, jeżeli w oknie podpowiedzi pojawiącym się, gdy przez chwilę przytrzymamy kursor myszy nad ikoną w zasobniku, będzie pokazywany bieżący czas.

Aby to umożliwić, w widoku projektowania zaznaczamy obiekt notifyIcon1 i za pomocą okna własności tworzymy metodę zdarzeniową związaną ze zdarzeniem MouseMove, w której umieszczamy polecenie z listingu 9.23.

Listing 9.23. Metoda aktualizująca zawartość okienka podpowiedzi

```
private void notifyIcon1_MouseMove(object sender, MouseEventArgs e)
{
    notifyIcon1.Text = "Godziny .NET 5 (" + DateTime.Now.ToString("T") + ")";
}
```

W efekcie po najechaniu myszą na ikonę komponentu zaktualizowana zostanie treść pokazywana w okienku podpowiedzi i gdy to okienko zobaczymy, zawierać będzie nazwę programu i aktualną godzinę (rysunek 9.16).

Rysunek 9.16.
Okienko podpowiedzi ikony z zasobnika



Dymek

Po dwukrotnym kliknięciu ikony w zasobniku pokażemy na ekranie *dymek*⁶ (ang. *balloon tip*) prezentujący nie tylko aktualną godzinę, ale również datę, z dniem tygodnia i dniem roku włącznie.

1. W widoku projektowania zaznacz komponent notifyIcon1 i za pomocą okna własności ustal:
 - a) BalloonTipTitle na *Godziny .NET 5*;
 - b) z rozwijanej listy przy BalloonTipIcon wybierz *Info*.
2. Utwórz metodę związaną ze zdarzeniem DoubleClick komponentu notifyIcon1 i umieść w nim polecenia z listingu 9.24.

Listing 9.24. Dwukrotne kliknięcie ikony spowoduje pokazanie dymku

```
private void notifyIcon1_DoubleClick(object sender, EventArgs e)
{
    string s = "Aktualna data: " + DateTime.Today.ToString("T");
    string[] DniTygodnia = {"Niedziela", "Poniedziałek",
                           "Wtorek", "Środa", "Czwartek", "Piątek", "Sobota"};
    byte numerDniaTygodnia = (byte)DateTime.Now.DayOfWeek;
    s += "\nDzień tygodnia: " + DniTygodnia[numerDniaTygodnia];
    s += "\nDzień roku: " + DateTime.Now.DayOfYear;
    s += "\nAktualny czas: " + DateTime.Now.ToString("T");
    s += "\n\n(c) Jacek Matulewski 2014";
    notifyIcon1.BalloonTipText = s;
    notifyIcon1.ShowBalloonTip(20 * 1000);
}
```

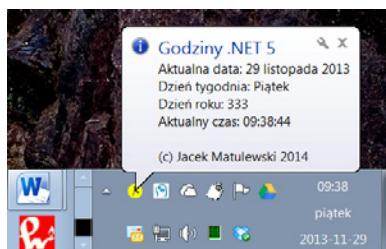
⁶ Ze względu na podobieństwo użyciem nazwy stosowanej w komiksach. Słyszałem jednak także określenia „balon” czy „chmurka”.

Na razie pokazywanie „dymku” związałem się z dwukrotnym kliknięciem ikony w zasobniku. Później będę chciał go pokazać także w trakcie wybijania pełnej godziny. Nie jest to jednak jego typowe zastosowanie. Jest raczej przeznaczony do przekazywania informacji, jakie powinny zwrócić szczególną uwagę użytkownika systemu, ale nie na tyle ważnych, by zmieniać aktywne okno poprzez wyświetlenie komunikatu (tj. odbierać „focus” bieżącemu oknu). Oczywiście pełna godzina nie należy do tak ważnych zdarzeń, ale jeżeli nasza aplikacja pozwalałaby na ustawienie alarmu, dymek byłby idealnym sposobem komunikowania o nadjeściu wyznaczonej w alarmie godziny.

Efekt dwukrotnego kliknięcia ikony aplikacji w zasobniku widoczny jest na rysunku 9.17. Dymek powinien być widoczny przez 20 sekund (argument metody ShowBalloonTip powiedany jest w milisekundach). Zgodnie z regułami systemu Windows okres ten nie może być krótszy niż 10 sekund (włączając wygaszanie dymku), a dłuższy niż 30 sekund. Jeżeli argument wykracza poza ten zakres, jest ignorowany.

Rysunek 9.17.

Jeden ze sposobów
nieinwazyjnego
powiadamiania
użytkownika



Odtwarzanie pliku dźwiękowego

Pozostaje przygotować kod, który będzie odpowiedzialny za wybijanie pełnych godzin. Użyjemy do tego kolejnego komponentu Timer, do którego właściwości Interval zostanie przypisana liczba milisekund do najbliższej pełnej godziny, obliczona w momencie uruchomienia aplikacji. Po tym czasie odtworzy on wskazany plik .wav i wówczas do właściwości Interval przypiszemy liczbę milisekund odpowiadającą pełnej godzinie. W ten sposób unikniemy ciągłego sprawdzania bieżącego czasu i ograniczymy do minimum zasoby komputera (czas procesora) wykorzystywane przez aplikację.

A zatem korzystając z komponentu System.Media.SoundPlayer, o pełnych godzinach odtwarzmy plik dźwiękowy.

1. Do projektu dodaj plik dźwiękowy *WAVE*, którego nazwę zmień na *Godziny.wav* (przykładowy plik dołączony jest do materiałów dołączonych do książki). W tym celu:
 - a) dla pozycji *Godziny* (projekt, nie rozwiązanie) w podoknie *Solution Explorer* wywołaj menu kontekstowe, z którego wybierz polecenie *Add*, a następnie *Existing Item...;*
 - b) w oknie dialogowym wyboru pliku zmień filtr na *Audio Files* i wskaz plik; kliknij przycisk *Add*;
 - c) zaznacz nowy plik widoczny w podoknie *Solution Explorer* i zmień jego właściwość *Copy to Output Directory* na *Copy if newer* lub *Copy always*. W ten sposób po komplikacji plik dźwiękowy znajdzie się w katalogu pliku *.exe*.

2. Na formie umieść kolejny komponent Timer (powinien to być timer2).
3. W rozwijanej liście okna własności wybierz obiekt Form1. Utwórz metodę zdarzeniową do zdarzenia Load i umieść w niej polecenia z listingu 9.25.

Listing 9.25. Konfigurowanie drugiego timera tuż po uruchomieniu aplikacji

```
private void Form1_Load(object sender, EventArgs e)
{
    DateTime nastepnaPelnaGodzina=new DateTime(DateTime.Now.Year,
                                                DateTime.Now.Month,
                                                DateTime.Now.Day,
                                                DateTime.Now.Hour+1,
                                                0,0,0);

    long ileMilisekundDoPelnejGodziny=
        (nastepnaPelnaGodzina.Ticks-DateTime.Now.Ticks)/10000;
    timer2.Interval=(int)ileMilisekundDoPelnejGodziny;
    timer2.Enabled=true;
}
```

4. Wróć do widoku projektowania i klikając dwukrotnie ikonę timer2, utwórz metodę zdarzeniową do zdarzenia Tick tego komponentu. W nim umieść polecenia z listingu 9.26.

Listing 9.26. Ten sposób wyznaczania czasu do następnego zdarzenia Tick nie jest odporny na manipulacje użytkownika, a dokładnie na ręczne zmianianie przez niego czasu systemu, ale ogranicza wykorzystanie procesora

```
private void timer2_Tick(object sender, EventArgs e)
{
    timer2.Interval = 3600000; // 1000ms*60sek*60min
    (new System.Media.SoundPlayer("Godziny.wav")).Play();
    notifyIcon1_DoubleClick(sender, e); // dymek
}
```

Dlaczego w metodzie Form1_Load dzielimy wartość uzyskaną z porównania własności DateTime.Now.Ticks dla pełnej godziny i obecnego czasu przez 10 000? Najmniejszą gwarantowaną w systemie Windows jednostką czasu jest 100 nanosekund⁷. W takich jednostkach podawana jest wartość we właściwości DateTime.Now.Ticks. 100 nanosekund to $100 \cdot 10^{-9}$ sekund, czyli 10^{-7} sekund. Natomiast Timer.Interval odmierzany jest w milisekundach, czyli zaledwie 10^{-3} sekund, co jest $10^{-3}/10^{-7} = 10^4 = 10\ 000$ razy większą jednostką niż ta w DateTime.Now.Ticks.

Klasa SoundPlayer wykorzystana w metodzie timer2_Tick potrafi odtwarzać dźwięk ze wskazanego pliku .wav obecnego na dysku (plik musi być rozpowszechniany razem z aplikacją) lub może skorzystać z dźwięku dołączonego do zasobów aplikacji. Wybrane przez nas parametry oznaczają, że plik .wav będzie odtwarzany asynchronicznie z pliku Godziny.wav. Asynchronicznie, czyli że działanie funkcji skończy się

⁷ Dokładniejszy pomiar czasu umożliwia funkcja WinAPI QueryPerformanceCounter dostępna od Windows 2000. Uzyskiwana w ten sposób rozdzielcość pomiaru czasu zależy od sprzętu, ale jeżeli pomiar jest możliwy, jest znacznie mniejsza od kilkuset nanosekund.

natychmiast po rozpoczęciu odtwarzania, bez czekania na jego koniec. Odtwarzanie pliku jest wówczas uruchamiane w osobnym wątku⁸. Jeżeli plik .wav został podany bez ścieżki dostępu, przeszukany zostanie bieżący katalog aplikacji, katalog *Windows* i jego podkatalog *System32*, katalogi wymienione w zmiennej środowiskowej PATH i zmapowane katalogi sieciowe. Dzięki ustawieniom dołączonego do projektu pliku *Godziny.wav* podczas komplikacji zostanie on automatycznie skompilowany w katalogu, w którym znajduje się skompilowany plik .exe, tj. *Moje dokumenty\Visual Studio 2013\Projects\Godziny\bin\Debug*.

Ustawienia aplikacji

Istnieje kilka sposobów, aby w aplikacjach .NET przechować drobne dane lub ustawienia. Często używane są pliki XML (rozdział 12.) i rejestr systemu Windows (klasa *Microsoft.Win32.Registry*). Ustawienia aplikacji powinny być jednak zapisywane i odczytywane za pomocą mechanizmu, którego zaletą jest przede wszystkim to, że nie korzysta z mechanizmów spoza samej platformy .NET. Są to tzw. *ustawienia aplikacji* (ang. *application settings*).

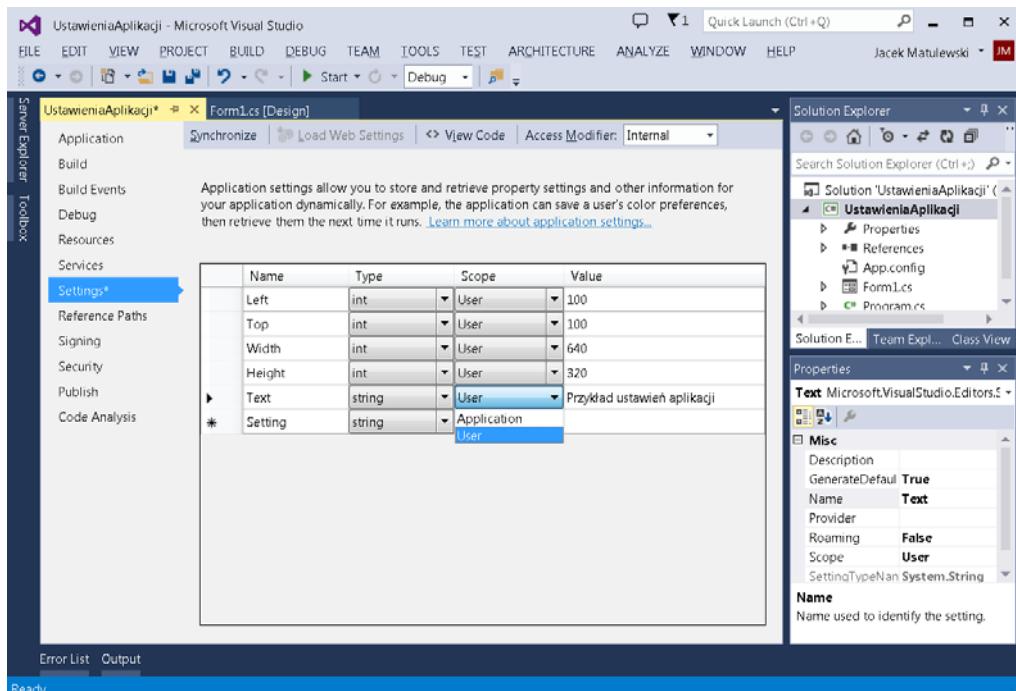
Tworzenie ustawień w trakcie projektowania aplikacji

Zdefiniujmy zatem ustawienia przechowujące informacje o położeniu i wielkości okna.

1. Utwórz nowy projekt aplikacji typu Windows Forms Application o nazwie *UstawieniaAplikacji*.
2. Z menu *Project* wybierz pozycję *UstawieniaAplikacji Properties...*, a po otwarciu okna ustawień zmień zakładkę na *Settings* (chodzi o zakładki widoczne na pionowym panelu z lewej strony okna).
3. Pojawi się tabela, w której można deklarować przechowywane wartości. Zdefiniuj pięć ustawień zgodnie ze wzorem na rysunku 9.18. W naszym przykładzie będą one przechowywać położenie i rozmiar formy.
4. Naciśnij *Ctrl+S*, zachowując w ten sposób ustawienia w pliku *app.config*.

Warto się przyjrzeć plikowi ustawień *app.config*, co można łatwo zrobić, klikając go dwukrotnie w podoknie *Solution Explorer*. Jest to plik zapisany w formacie XML. Jego elementem nadzawanym jest *configuration*. W nim znajduje się *userSettings* i w nim zdefiniowane są elementy dla poszczególnych ustawień. Każdy z takich elementów nazywa się *setting*. Nazwa ustawienia przechowywana jest w atrybucie *name*, a jej wartość w pudełku *value*. Gdybyśmy użyli zakresu aplikacji zamiast zakresu użytkownika (kolumna *Scope* w tabeli widocznej na rysunku 9.18), to w pliku *app.config* oprócz elementu *userSettings* powstalby także element *applicationSettings*, w którym byłyby przechowywane ustawienia o tym zakresie. Z naszego punktu widzenia różnią się one przede wszystkim tym, że nie mogą być zmieniane w trakcie działania programu.

⁸ Dostępna jest również metoda *PlaySync*, która odtwarza dźwięk w bieżącym wątku i kończy się, jak łatwo się domyślić, dopiero po zakończeniu odtwarzania.



Rysunek 9.18. Definiowanie ustawień w trakcie projektowania aplikacji

Jeżeli przyjrzymy się dokładnie plikowi *app.config*, zauważymy, że nie zawiera kompletnych informacji o ustawieniach. Przede wszystkim nie ma w nim zapisanych typów zmiennych. Są one bowiem przechowywane w pliku *Settings.settings* z podkatalogu *Properties* i po komplikacji znajdują się w pliku wykonywalnym *.exe*. Plik *Settings.settings* jest niczym więcej jak wiernie zapisaną tabelą widoczną na rysunku 9.18.

Po komplikacji plik *app.config* kopiowany jest do katalogu *bin/Debug*, gdzie znajdziemy go pod nazwą *UstawieniaAplikacji.exe.config*. Dzieje się to niezależnie od ustawienia własności *Copy to Output Directory*, która powinna być ustaliona na *Do not copy*. Tak naprawdę plik *UstawieniaAplikacji.exe.config* to nie jest miejsce, w którym przechowywane będą ustawienia o zakresie użytkownika. Wyjaśnię to jednak dopiero za chwilę.

Odczytywanie ustawień z poziomu kodu

Aplikacja po uruchomieniu powinna odczytywać przechowane w ustawieniach informacje o położeniu i wielkości formy, a następnie użyć ich.

1. Przejdź do widoku projektowania formy (zakładka *Form1.cs [Design]*).
2. Za pomocą podokna *Properties* utwórz metodę zdarzeniową do zdarzenia *Load* formy.
3. Dodaj do pliku polecenie `using UstawieniaAplikacji.Properties;` uwzględniające przestrzeń nazw, w której umieszczona została klasa *Settings*, ułatwiająca dostęp do zdefiniowanych przed chwilą ustawień.

4. W klasie formy Form1 zdefiniuj pole o nazwie ustawienia:

```
Settings ustawienia = new Settings();
```

5. W metodzie Form_Load, którą utworzyłeś w punkcie 2., umieść polecenia odczytujące ustawienia i kopiące ich wartość do odpowiednich właściwości formy (listing 9.27).

Listing 9.27. Odczytywanie ustawień

```
private void Form1_Load(object sender, EventArgs e)
{
    this.Left = ustawienia.Left;
    this.Top = ustawienia.Top;
    this.Width = ustawienia.Width;
    this.Height = ustawienia.Height;
    this.Text = ustawienia.Text;
}
```

Po uruchomieniu aplikacji (*F5*) okno powinno znajdować się w miejscu wyznaczonym przez wartości zapisane w ustawieniach aplikacji.



Bardzo podobne ćwiczenie, ale z „ręczną” obsługą danych w pliku XML, wykonamy w rozdziale 12.

Zapisywanie ustawień z poziomu kodu

Oczywiście ustawienia aplikacji nie służą do przechowywania jednych, z góry ustalonych wartości. Aplikacja może je nie tylko odczytywać, ale również zmieniać. Chciałbym, aby nasza przykładowa aplikacja zmieniała te wartości przed zamknięciem formy, „zapamiętując” swoje położenie i wielkość. Aby to uzyskać, wróćmy do wiodu projektowania i utwórzmy metodę zdarzeniową do zdarzenia FormClosed, a następnie umieścmy w niej polecenia z listingu 9.28.

Listing 9.28. Zapisywanie ustawień do pliku

```
private void Form1_FormClosed(object sender, FormClosedEventArgs e)
{
    ustawienia.Left = this.Left;
    ustawienia.Top = this.Top;
    ustawienia.Width = this.Width;
    ustawienia.Height = this.Height;
    ustawienia.Text = this.Text;
    ustawienia.Save();
}
```

Ponieważ wszystkie ustawienia aplikacji, które zapisujemy powyższą metodą, należą do ustawień użytkownika, wykonanie tej metody spowoduje, że platforma .NET utworzy dla nich plik w katalogu domowym użytkownika (np. *C:\Users\Jacek* lub *C:\Documents and Settings\Jacek*), a dokładniej w jego podkatalogu *AppData\Local* (względnie *Ustawienia lokalne\Dane aplikacji*). Powstanie tam katalog o nazwie aplikacji, z podkatalogiem oznaczającym konkretny plik wykonywalny i jeszcze jed-

nym podkatalogiem zawierającym numer wersji aplikacji. Dopiero w tym miejscu powstanie plik XML o nazwie *user.config*.

Plik *user.config* zawiera sekcję *userSettings*, czyli ustawienia aplikacji z zakresu użytkownika. Taki sam zbiór ustawień znajdziemy w pliku *UstawieniaAplikacji.exe.config*, który umieszczony jest w katalogu aplikacji i powinien być z nią rozpowszechniany. Ustawienia z pliku *user.config* są jednak dynamicznie modyfikowane przez metodę z listingu 9.28, podczas gdy plik *UstawieniaAplikacji.exe.config* przechowuje tylko ustawienia domyślne, jakie wprowadziliśmy w projekcie. Do pliku *user.config* nie są natomiast zapisywane ustawienia z zakresu aplikacji (element *applicationSettings*) — pozostały one dla aplikacji ustawieniami tylko do odczytu.

Dywany graficzny

Jak pokazałem w aplikacji Notatnik.NET, drukowanie to nic innego jak malowanie na płótnie kartki papieru z użyciem klasy *Graphics*. Co ciekawe, w bardzo podobny sposób można rysować na płótnie formy i będziemy do tego używać tej samej klasy. Pokażę to w kolejnym projekcie.

Zdarzenie Paint formy

Zaczniemy od tego, że korzystając ze zdarzenia *Paint*, pokryjemy obszar użytkownika formy piksel po pikselu kolorem cytrynowym. Użyjemy do tego obiektu *Graphics* odczytanego z argumentu *PaintEventArgs* metody związanej ze zdarzeniem *Paint*.

1. Utwórz nowy projekt o nazwie *Arras*.
2. Znajdź zdarzenie *Paint* formy i stwórz związaną z nim metodę zdarzeniową; w niej umieść kod z listingu 9.29.

Listing 9.29. Kod metody wywoływanej w momencie odświeżania formy

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    for (int x = 0; x < this.ClientSize.Width; x++)
        for (int y = 0; y < this.ClientSize.Height; y++)
    {
        Pen pioro = new Pen(Color.LemonChiffon, 1);
        g.DrawLine(pioro, x, y, x + 1, y);
    }
}
```

3. Skompiluj i uruchom aplikację, naciskając klawisz *F5*.

Wykorzystanie zdarzenia *Paint* do naszego zadania jest naturalne. Jest ono bowiem wywoływanie po raz pierwszy tuż po utworzeniu formy, jak również zawsze wtedy, gdy forma wymaga odświeżenia. Nie musimy wobec tego martwić się o uruchamianie kodu z metody 9.29 i odnawianie zawartości formy — będzie robił to za nas system, wywołując zdarzenie *Paint* zawsze, gdy jest to potrzebne.

Jak wspomniałem wyżej, rysowanie na formie możliwe jest przy użyciu obiektu typu `Graphics`, który pobraliśmy z drugiego argumentu metody zdarzeniowej. Tego samego obiektu używaliśmy do drukowania. Wśród metod tej klasy nie ma, niestety, rysowania punktu, ale poradziliśmy sobie, rysując dwupunktowe linie od punktu wyznaczonego przez indeksy pętli do sąsiedniego punktu z prawej strony.

Warto zwrócić uwagę, że zakres pętli jest wyznaczony przez `this.ClientSize.Width` i `this.ClientSize.Height`, a nie przez `this.Width` i `this.Height`. Pierwsza para to rozmiar „wewnętrznej” części okna, na której można umieszczać kontrolki i malować (tzw. obszar użytkownika), podczas gdy druga to „zewnętrzny” rozmiar obejmujący także brzegi okna i jego pasek tytułu, po których nie można rysować.

Po skompilowaniu oczywiste staje się, że aplikacja jest daleka od optymalnej wydajności. W istocie tworzenie grafiki punkt po punkcie bezpośrednio na powierzchni formy (bez buforowania) nie nadaje się do żadnych praktycznych zastosowań. Dla efektu, jaki uzyskaliśmy po wykonaniu powyższego ćwiczenia, tj. zapelnienia całej formy jednolitym kolorem, o wiele szyszsze byłoby polecenie rysowania prostokąta, tj.:

```
g.FillRectangle(Brushes.LemonChiffon,0,0,this.Width,this.Height);
```

Należałyby umieścić je w metodzie `Form1_Paint` zamiast podwójnej pętli. Także umieszczenie polecenia tworzącego obiekt pióro wewnątrz pętli zwalnia działanie programu. Znalazło się ono tam jednak dlatego, że w następnym zadaniu kolor pióra będzie uzależniony od położenia piksela (tj. od zmiennych `x` i `y`).

Kolorowy wzór

Zastąpmy jednolity cytrynowy kolor dywanem utkanym z kolorowych nici.

1. Zastąp kod w metodzie `Form1_Paint`, a dokładnie polecenie tworzące pióro znajdujące się wewnątrz pętli wyróżnionymi poleceniami z listingu 9.30.
2. Następnie ponownie skompiluj i uruchom aplikację.

Listing 9.30. Modyfikacja metody zdarzeniowej

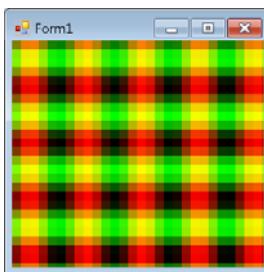
```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    for(int x=0;x<this.Width;x++)
        for(int y=0;y<this.Height;y++)
    {
        // Pen pióro=new Pen(Color.LemonChiffon,1);
        int R = (int)(255 * 0.5 * (1 + Math.Sin(x / 10)));
        int G = (int)(255 * 0.5 * (1 + Math.Sin(y / 10)));
        Pen pióro=new Pen(Color.FromArgb(R,G,0),1);
        g.DrawLine(pióro,x,y,x+1,y);
    }
}
```

Zauważmy, że w metodzie z listingu 9.30 występują zmienne `g` i `G`. Są to oczywiście różne zmienne: pierwsza jest obiektem typu `Graphics`, druga — liczbą typu `int`. To prosty dowód, że C# rozróżnia wielkość liter.

Niestety, efekt nie jest zachwycający (rysunek 9.19). Zamiast pięknego klasycznego dywanu powstał nowoczesny, nieco kubistyczny obraz, w którym zamiast płynnych zmian kolorów widzimy kilkanaście kolorowych prostokątów.

Rysunek 9.19.

Szkocka krata



Przyczyną błędu jest bardzo często zdarzające się przeoczenie. Otóż w rodzinie języków C, a więc w C/C++, Javie i C#, a w przeciwieństwie do Pascala, operator dzielenia / zwraca wynik tego samego typu jak jego argumenty. Jeżeli dzielimy przez siebie dwie liczby naturalne, wynik jest zaokrąglany w dół i zwracany w postaci liczby naturalnej.

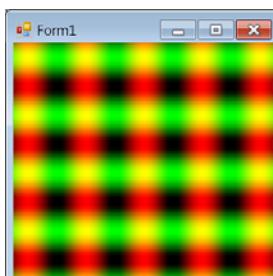
Jak sobie poradzić z tym problemem? Wystarczy dzielić nie przez 10, ale przez 10.0 lub 10f. Kompilator rozpozna w dzielниku liczbę rzeczywistą i zwróci jako wynik dzielenia również liczbę rzeczywistą. Wobec tego polecenia definiujące zmienne R i G należy poprawić w następujący sposób:

```
int R = (int)(255 * 0.5 * (1 + Math.Sin(x / 10.0)));
int G = (int)(255 * 0.5 * (1 + Math.Sin(y / 10.0)));
```

W efekcie aplikacja przybierze pożądany wygląd widoczny na rysunku 9.20. Zachęcam do eksperymentowania z funkcjami określającymi składowe koloru pióra. Można uzyskać fantastyczne wzory i zestawienia kolorów.

Rysunek 9.20.

Znacznie lepiej!



Buforowanie

Najpoważniejszą wadą tej aplikacji pozostaje jednak jej szybkość. Dywan powstaje wolno, choć oczywiście zależy to od szybkości komputera. Najprostszym sposobem na poprawienie wydajności aplikacji jest tworzenie obrazu nie bezpośrednio na powierzchni formy, ale na osobnej bitmapie i następnie szybkie jej skopiowanie. Technika taka nazywa się podwójnym buforowaniem. W tym celu modyfikujemy jeszcze raz metodę Form1_Paint zgodnie z listingu 9.31. Definiujemy również nowe pole formy — obiekt typu Bitmap, który będzie pełnił rolę bufora.

Listing 9.31. Metoda Form1_Paint z dodanym buforowaniem i nieco zmienionym wzorem

```

private Bitmap bufor = null;

private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;

    bufor=new Bitmap(this.ClientSize.Width,this.ClientSize.Height);
    for(int x=0;x<this.ClientSize.Width;x++)
        for(int y=0;y<this.ClientSize.Height;y++)
        {
            int R=(int)(255*0.5*(1+Math.Sin((x+y)/10.0)));
            int G=(int)(255*0.5*(1+Math.Sin((y-x)/10.0)));
            bufor.SetPixel(x,y,Color.FromArgb(R,G,0));
        }
    g.DrawImage(bufor,0,0);
}

```

Zdefiniowaliśmy pole bufor będące bitmapą. W naszym przykładzie jest ono inicjowane dopiero w metodzie Form1_Paint, aby jak najprościej uwzględnić ewentualne zmiany rozmiaru formy. Następnie postępując analogicznie do poprzedniej wersji metody, punkt po punkcie przygotowujemy dywan i zapisujemy go do bufora. Tym razem możemy wykorzystać metodę SetPixel zmieniającą kolor jednego piksela. I wreszcie używamy wielokrotnie przeciążonej metody Graphics.DrawImage, aby narysować tak przygotowanąbitmapę na powierzchni formy. Opóźnienie związane z rysowaniem obrazu poza ekranem jest widoczne po uruchomieniu aplikacji, ale jest znacznie krótsze niż czas potrzebny do rysowania bezpośrednio na formie.

**Wskazówka**

Rysowanie dywanu odbywa się wielokrotnie szybciej, jeżeli aplikacja jest uruchamiana poza środowiskiem Visual Studio lub uruchomiona jest w środowisku bez debugowania (*Ctrl+F5*).

Zapisywanie obrazu dywanu do pliku

Fakt, że nasz bufor jest bitmapą, wykorzystamy, aby zapisać utworzony obraz dywanu do pliku graficznego. W tym celu:

1. Przejdź do zakładki *Events* okna właściwości i utwórz metodę zdarzeniową do zdarzenia KeyPress formy.
2. W niej wpisz polecenia z listingu 9.32.

Listing 9.32. Naciśnięcie klawiszy *Ctrl+S* spowoduje zapisanie obrazu

```

private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar==(char)19 && bufor!=null)
    {
        string nazwaPliku = "Arras.png";
        bufor.Save(nazwaPliku);
        MessageBox.Show("Obraz zachowany w pliku " + nazwaPliku);
    }
}

```

3. W efekcie po naciśnięciu klawiszy *Ctrl+S* do pliku *Arras.png* w katalogu aplikacji zapisany zostanie bieżący obraz widoczny na formie.



Jeżeli na formie znajdują się jakieś kontrolki, należy ustawić własność KeyPreview formy na True, aby powyższa metoda była wywoływana.

Rozdział 10.

Przeciągnij i upuść

Typowym zagadnieniem, które wiąże się z tworzeniem aplikacji z graficznym interfejsem użytkownika, jest korzystanie z możliwości przeciągania elementów pomiędzy różnymi komponentami-pojemnikami za pomocą myszy. Obsługę tego mechanizmu z poziomu kodu C# opiszę na prostym przykładzie aplikacji z dwoma listami.

Z punktu widzenia programisty operacja *drag & drop* (z ang. *przeciagnij i upuść*) może być zrealizowana dzięki obsłudze kilku zdarzeń, przede wszystkim `MouseDown`, `DragEnter`, `DragOver` i `DragDrop`. Operacja przeniesienia i upuszczenia składa się z trzech etapów:

- 1. Rozpoczęcia przenoszenia** — wymaga wywołania metody `DoDragDrop` na rzecz komponentu, z którego przenoszony element jest zabierany.
- 2. Akceptacji** — przeciągając element nad inny komponent, wywołujemy jego zdarzenie `DragOver`; jest to właściwy moment, żeby podjąć decyzję o tym, czy przenoszony element może być w ogóle upuszczony na docelowy komponent; użytkownik aplikacji powinien zostać powiadomiony o decyzji za pomocą zmiany kształtu kurSORA myszy.
- 3. Reakcji na upuszczenie przenoszonego elementu** — w takiej sytuacji uruchamiana jest metoda związana ze zdarzeniem `DragDrop`.

W systemie Windows dla danego użytkownika możliwy jest tylko jeden proces *drag & drop* jednocześnie. To oczywiście nic dziwnego, zważywszy na to, że mamy do dyspozycji tylko jeden kurSOR. Jak na razie funkcja wielodotyku tego nie zmieniła.

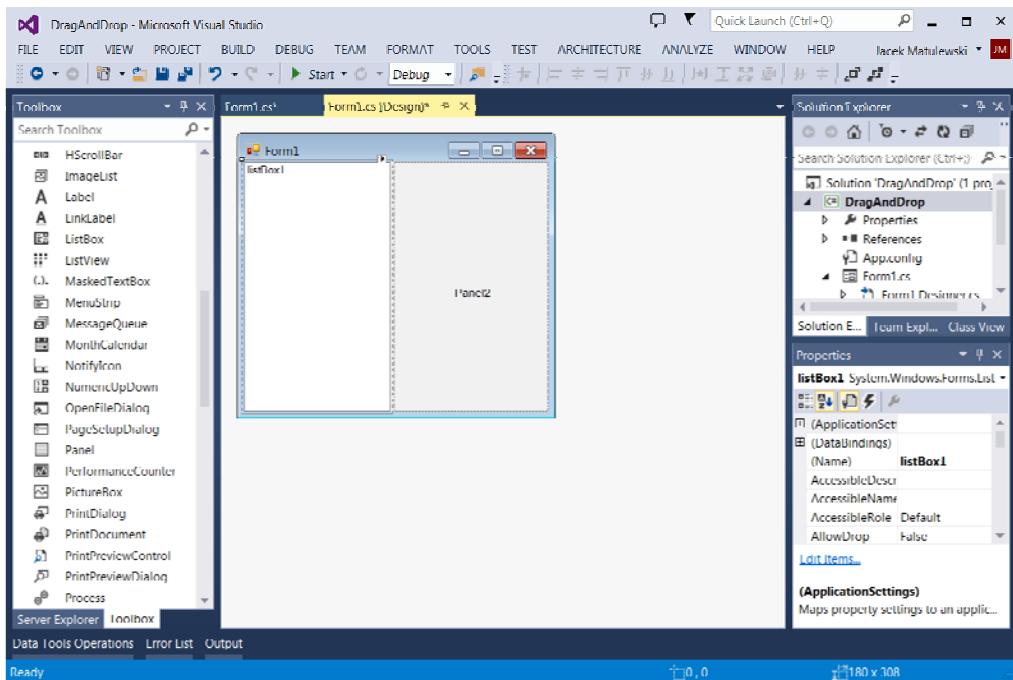
Podstawy

W opisywanym poniżej projekcie przygotujemy metody obsługujące przeciąganie i upuszczanie elementów. W miarę możliwości zrobimy to w taki sposób, żeby nie używać jawnie nazw pojemników. W zamian użyjemy argumentów przekazywanych przez metody zdarzeniowe. Dzięki temu metody będą elastyczniejsze — będzie ich można użyć dla wielu pojemników (w naszym przypadku list). A ponadto ułatwiać to będzie kopiowanie ich kodu do kolejnych projektów.

Interfejs przykładowej aplikacji

Zacznijmy od przygotowania projektu aplikacji, na której będziemy ćwiczyć. W tym celu na formie umieścimy dwie listy wypełnione kilkoma pozycjami.

1. Utwórz nowy projekt typu Windows Forms Application o nazwie *DragAndDrop*.
2. W widoku projektowania powiększ formę i umieść na niej komponent *SplitContainer* z zakładki *Containers* (z ang. *pojemniki*) podokna *Toolbox*.
3. Ustaw identyczną szerokość jego paneli (własność *SplitterDistance*).
4. W jego lewym panelu umieść listę *ListBox*.
5. Jej własność *Dock* zmień na *Fill* (rysunek 10.1). W efekcie lista wypełni cały lewy panel.



Rysunek 10.1. Dokujemy listę w panelu pojemnika

6. W podobny sposób umieść drugą listę w prawym panelu i ją również zadokuj.
7. W listach umieść po dziesięć pozycji, które będziesz mógł następnie przenosić z jednej listy do drugiej. Odpowiednie polecenia możesz dodać do konstruktora klasy formy za wywołaniem metody *InitializeComponent* (listing 10.1).

Listing 10.1. Konstruktor klasy Form1 z poleceniami dodającymi po 10 elementów do każdego komponentu ListBox

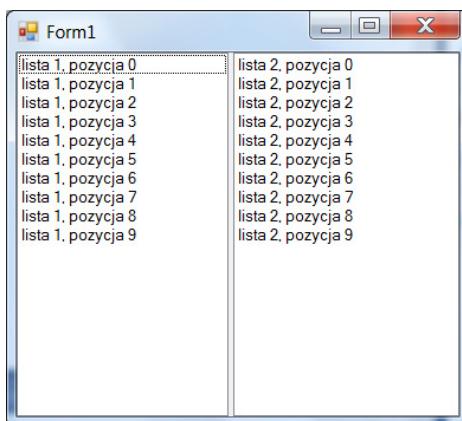
```
public Form1()
{
    InitializeComponent();

    // Wypełnianie list
    for (int i = 0; i < 10; i++)
    {
        listBox1.Items.Add("lista 1, pozycja " + i.ToString());
        listBox2.Items.Add("lista 2, pozycja " + i.ToString());
    }
}
```

- 8.** Dla wygody możesz również zmienić pozycję okna po uruchomieniu aplikacji. Wystarczy w widoku projektowania zmienić właściwość StartPosition formy na CenterScreen.

W efekcie uzyskamy widoczną na rysunku 10.2 aplikację, na której możemy testować korzystanie z mechanizmu *drag & drop*. Przy zmianie rozmiaru okna aplikacji panele również zmieniają wielkość, jednak zachowują proporcje szerokości. Można zmienić i to, przesuwając myszą obszar między panelami.

Rysunek 10.2.
Interfejs aplikacji
złożony z dwóch
komponentów-pojemników



Iinicjacja procesu przeciągania

Aby można było uruchomić procedurę *drag & drop*, konieczne jest wywołanie metody DoDragDrop na rzecz komponentu, z którego przenosimy element (w naszym przypadku jednej z list). Uruchommy zatem metodę DoDragDrop listy w przypadku wcisnięcia na jej obszarze lewego przycisku myszy.

- 1.** W edytorze kodu zdefiniuj prywatne pole klasy Form1 o nazwie dragDropSource typu object, które będzie przechowywać referencję do komponentu, z jakiego zabraliśmy przeciągany element. Nowa deklaracja musi być umieszczona w obrębie klasy Form1, ale poza jej metodami (listing 10.2).

Listing 10.2. Miejsce, w którym można wstawić deklarację pola klasy Form1

```
namespace DragAndDrop
{
    public partial class Form1 : Form
    {
        object dragDropSource = null;

        public Form1()
        {
            InitializeComponent();
            ...
        }
    }
}
```

2. Następnie przejdź do widoku projektowania i przytrzymując klawisz *Ctrl*, zaznacz jednocześnie komponenty listBox1 i listBox2.
3. Za pomocą okna *Properties* utwórz metodę zdarzeniową związaną ze zdarzeniem *MouseDown* obu komponentów. W niej umieść wywołanie metody *DoDragDrop* (listing 10.3).

Listing 10.3. Zdarzenia *MouseDown* i *MouseClick* przekazują więcej informacji niż zdarzenie *Click*

```
private void listBox1_MouseDown(object sender, MouseEventArgs e)
{
    ListBox lbSender=sender as ListBox;
    int indeks=lbSender.IndexFromPoint(e.X,e.Y);
    dragDropSource=sender; //przechowanie referencji dla DragOver

    if (e.Button==MouseButtons.Left && indeks!=-1)
    {
        lbSender.DoDragDrop(lbSender.Items[indeks],
            DragDropEffects.Copy | DragDropEffects.Move);
    }
}
```

Jak widać na listingu 10.3, wywołanie metody *DoDragDrop* obwarowaliśmy kilkoma warunkami: sprawdzamy, czy użyty przycisk myszy to przycisk lewy i czy rzeczywiście w momencie jego przyciśnięcia kurSOR myszy znajdował się nad jednym z elementów listy. Dopiero wtedy, kiedy oba warunki są spełnione, wywołujemy metodę *DoDragDrop*. Jej argumentami są obiekt, który ma być przenoszony, a więc zaznaczony element listy, oraz zbiór operacji, jakie można na nim wykonać. W przypadku list sensowne operacje ograniczają się w zasadzie do *DragDropEffects.Move* oznaczającej przenoszenie oraz *DragDropEffects.Copy* oznaczającej kopiowanie. Zezwalamy na obydwie.

Działanie metody *DoDragDrop* kończy się dopiero wtedy, gdy przenoszony obiekt zostanie upuszczony (przycisk myszy zostanie zwolniony). Z tego wynika, że zdarzenia *DragOver* i *DragDrop* muszą być wykonane w osobnym wątku, bo metoda, w której uruchomiona zostanie metoda *DoDragDrop*, nie zakończy się przed ich wywołaniem. Menedżer zadań pokazuje, że w momencie inicjacji procesu przenoszenia do aplikacji dodawane są nawet dwa dodatkowe wątki. Co z tego wynika? Otóż konsekwencje są dwie. Po pierwsze, nie należy po wywołaniu metody *DoDragDrop* umieszczać żadnych poleceń, które miałyby należeć do pierwszej fazy procesu *drag & drop*. Po drugie,

można tam umieścić polecenia należące do trzeciej fazy procesu, a szczególnie ewentualne operacje dotyczące komponentu źródłowego, np. usuwanie przenoszonego elementu.

Zgodnie z zapowiedzią w metodzie `listBox1_MouseDown`, podobnie jak i w kolejnych, unikamy jawnego używania nazw komponentów. Zamiast tego wykorzystaliśmy argument `sender` metody zdarzeniowej przesyłający referencję do komponentu, z powodu którego wywołano zdarzenie. W naszym przypadku należy go oczywiście wcześniej zrzutować na typ `ListBox` (zmienna `lbSender`). Przygotowana w taki sposób metoda będzie mogła być stosowana także do kolejnych list, które umieścimy na formie.

Należy także zwrócić uwagę na to, że metoda `listBox1_MouseDown` zapisuje do zdefiniowanego w punkcie 2. pola `dragDropSource` referencję do komponentu, z którego pobieramy element. Przechowywanie tej referencji może być przydatne w drugim etapie, jeżeli chcemy uzależnić pozwolenie upuszczenia przenoszonego elementu od tego, skąd on pochodzi. Wiąże się to ze wspomnianym już faktem, że do metody zdarzeniowej związanej ze zdarzeniem `DragOver`, która będzie wykorzystywana w tej fazie procesu, nie jest przekazywana żadna informacja o komponencie-pojemniku, z którego pochodzi przenoszony obiekt.

Akceptacja upuszczenia elementu

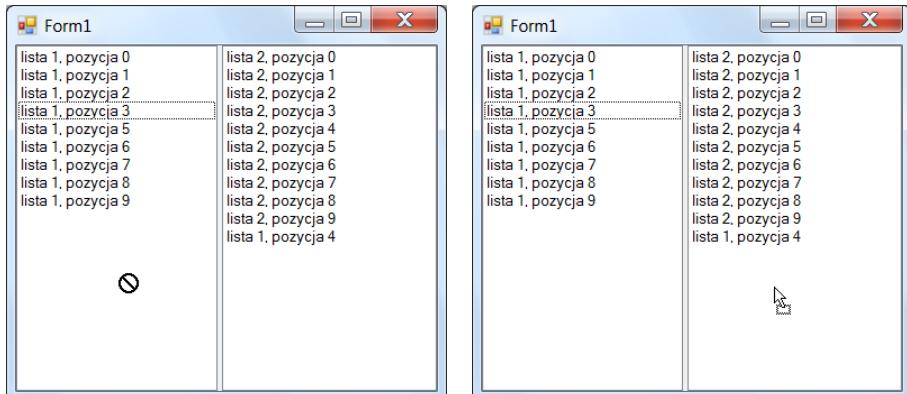
Jeżeli przenoszony element przesuniemy nad docelowy komponent (np. drugą listę), uruchomione zostanie jego zdarzenie `DragOver`. Daje to możliwość nadania kursorowi myszy kształtu, który informuje użytkownika o tym, czy upuszczenie przenoszonego obiektu na danym komponencie jest możliwe. Aby jednak to zdarzenie było w ogóle wywoływanie, własność `AllowDrop` komponentu docelowego musi być ustawiona na `true`.

1. Ponownie zaznacz obie listy na podglądzie formy.
2. Za pomocą podokna *Properties* zmień ich własność `AllowDrop` na `True`.
3. Nie zmieniając zaznaczenia, utwórz dla nich obu metodę zdarzeniową związaną ze zdarzeniem `DragOver`, w której umieścisz widoczne na listingu 10.4 instrukcje określające kształt kurSORA po przeniesieniu obiektu nad te komponenty.

Listing 10.4. Przenoszony element nie może być w naszym projekcie upuszczony na komponent, z którego został zabrany

```
private void listBox1_DragOver(object sender, DragEventArgs e)
{
    if (sender==dragDropSource)
        e.Effect=DragDropEffects.None;
    else
        if ((e.KeyState & 8)==8)
            e.Effect=DragDropEffects.Copy; //z CTRL
        else
            e.Effect=DragDropEffects.Move;
}
```

Kształt kurSORA wyznaczony jest w powyższej metodzie przez rodzaj operacji, jaką chcemy wykonać na przenoszonym elemencie (rysunek 10.3). Należy pamiętać przy tym, że wartość przypisana do decydującej o tym właściwości e.Effect musi być jedną z wartości wskazanych w drugim argumencie metody DoDragDrop (listing 10.3). My wskazaliśmy wówczas dwie: kopiowanie i przenoszenie. Z listingu 10.4 wynika, że rodzaj wykonywanej operacji zależy od tego, czy został naciśnięty klawisz *Ctrl*. Jeżeli tak, element zostanie skopiowany, jeżeli nie — przeniesiony.



Rysunek 10.3. Różne fazy przenoszenia elementu i odmienne kształty kurSORA informujące o stanie procesu

Dodatkowym warunkiem, wyrażonym w pierwszej linii kodu metody `listBox1_DragOver`, jest to, żeby pojemnik zgłaszający zdarzenie (komponent, nad którym jest kurSOR myszy przenoszący element) nie był tym samym komponentem, z jakiego obiekt został pobrany. W takim przypadku możliwość upuszczenia jest blokowana. Właśnie do tego typu warunków, i tylko dla nich, konieczne było umieszczenie w zmiennej `dragDrop` →`Source` referencji do komponentu, z którego przenoszony jest obiekt.

Reakcja na upuszczenie elementu

W momencie upuszczenia elementu aktywowane zostaje zdarzenie `DragDrop` komponentu, nad którym aktualnie znajduje się kurSOR myszy. Równocześnie w osobnym wątku dobiera końca działanie metody `DoDragDrop`. Chcemy, aby w reakcji na to zdarzenie przenoszony element był dodany do listy, na którą został upuszczony. Na listingu 10.5 prezentuję kod metody zdarzeniowej związanego ze zdarzeniem `DragDrop` realizujący tę ideę (ponownie należy ją związać ze zdarzeniem obu list).

Listing 10.5. Reakcja aplikacji na upuszczenie przenoszonego elementu

```
private void listBox1_DragDrop(object sender, DragEventArgs e)
{
    ListBox lbSender=sender as ListBox;
    int indeks=lbSender.IndexFromPoint(lbSender.PointToClient(new Point(e.X,e.Y)));
    if (indeks== -1) indeks=lbSender.Items.Count;
    lbSender.Items.Insert(indeks,e.Data.GetData(DataFormats.Text).ToString());
}
```

Na tym etapie nie powinniśmy już sprawdzać, czy dany obiekt może zostać upuszczony. To należało do zadań metody związanej ze zdarzeniem DragOver. Inaczej mogłyby dojść do sytuacji, w której pojawiłby się kursor myszy świadczący o akceptacji upuszczenia, a przenoszonego elementu wcale nie dałoby się upuścić.

W drugiej linii powyższej metody określamy pozycję elementu listy, nad którym znajdował się upuszczony element w momencie zwolnienia przycisku myszy. Pewnym utrudnieniem jest to, że tym razem położenie myszy przekazywane jest w obiekcie DragEventArgs we współrzędnych ekranu, a nie formy. Pozornie jest to szczegół, ale nieświadomy tego faktu programista może osiągnąć, próbując znaleźć powód, dla którego kod nie chce pracować, szczególnie że w obiekcie MouseEventArgs, używanym wcześniej w metodzie zdarzeniowej związanej z MouseDown, położenie myszy podawane było we współrzędnych okna. Jeżeli miejsce zwolnienia przycisku myszy i upuszczenia obiektu nie znajduje się nad żadnym elementem listy, to zwracany indeks równy jest -1. Wówczas zmieniamy jego wartość na równą liczbę elementów w liście, co wskazuje na miejsce za ostatnim elementem. Pozwoli to na użycie także i tu metody ListBox. ↳ Items.Insert, która w tym przypadku zadziała tak samo, jakbyśmy użyli metody ListBox.Items.Add.

Czynności wykonywane po zakończeniu procesu przenoszenia i upuszczania

Zauważmy, że w metodzie listBox1_DragDrop brakuje poleceń usuwających przenoszony element z komponentu-źródła. Można je tam umieścić, korzystając z referencji do pojemnika-źródła dragDropSource zrzutowanej na typ ListBox, ale wygodniej umieścić je w metodzie listBox1_MouseDown po wywołaniu (i zakończeniu) metody DoDragDrop, oczywiście jeżeli zrealizowany proces był przenoszeniem, a nie kopowaniem. Odpowiednie zmiany w metodzie listBox1_MouseDown zaznaczone zostały na listingu 10.6.

Listing 10.6. To, co znajduje się za wywołaniem DoDragDrop, należy do trzeciej fazy procesu drag & drop

```
private void listBox1_MouseDown(object sender, MouseEventArgs e)
{
    ListBox lbSender=sender as ListBox;
    int indeks=lbSender.IndexFromPoint(e.X,e.Y);
    dragDropSource=sender; //przechowanie referencji dla DragOver

    if (e.Button==MouseButtons.Left && indeks!=-1)
    {
        DragDropEffects operacja=
            lbSender.DoDragDrop(lbSender.Items[indeks],
                DragDropEffects.Copy | DragDropEffects.Move);
        if (operacja==DragDropEffects.Move)
            lbSender.Items.RemoveAt(indeks);
    }

    dragDropSource=null;
}
```

Sprawdzenia, czy użytkownik zdecydował się na kopiowanie, czy na przenoszenie, dokonujemy, korzystając z wartości typu `DragDropEffects` zwracanej przez metodę `DoDragDrop`. Jest to wartość równa tej, jaką przypisaliśmy w metodzie `listBox1_DragOver` do `e.Effect` (listing 10.4). Na tej podstawie podejmowana jest decyzja o tym, czy usunąć, czy pozostawić przenoszony element w pojemniku-źródle. Na końcu tej metody dodaliśmy także polecenie „czyszczące” pole `dragDropSource`.

Aby sprawdzić poprawność i ogólność przygotowanych metod, możemy wykonać prosty test. Dodajmy do formy trzecią listę `ListBox`, ustawmy jej własność `AllowDrop` na `true` i podepnijmy ją do wszystkich trzech metod zdarzeniowych (odpowiednio do zdarzeń `MouseDown`, `DragOver` i `DragDrop`). W ten sposób włączymy ją do układu kontrolek-pojemników, między którymi możliwe jest przenoszenie elementów.

Przenoszenie elementów między różnymi aplikacjami

Zaskakujące dla osób programujących wcześniej dla platformy Win32 może być to, że przenoszenie elementów między aplikacjami .NET nie wymaga żadnych dodatkowych modyfikacji przygotowanego już kodu. Można to przetestować, uruchamiając dwie instancje projektowanej aplikacji i przenosząc elementy między nimi. W Win32 takie przenoszenie elementów między aplikacjami wymagałoby wykorzystania komunikatorów, a w platformie .NET jest „gratis”.

Zagadnienia zaawansowane

Proste przenoszenie elementów z jednego pojemnika do drugiego, jakie opisałem w poprzednim podrozdziale, można komplikować na wiele różnych sposobów. Poniżej przedstawię trzy z nich: przeciąganie z programem opóźniającym, przenoszenie wielu elementów oraz przenoszenie plików. W dwóch pierwszych przypadkach zaczniemy od projektu, który rozwijaliśmy do tej pory, przyjmując za punkt startowy jego obecny stan. Warto zatem ów projekt teraz zapisać i zmiany wprowadzać w jego kopii. W ostatnim przypadku wykorzystamy natomiast projekt notatnika z poprzedniego rozdziału.

Opóźnione inicjowanie procesu przenoszenia

Powróćmy do pierwszej fazy procesu. Należy zwrócić uwagę na to, że proces przenoszenia nie musi być inicjowany natychmiast po naciśnięciu lewego przycisku myszy. Metodę `DoDragDrop` możemy uruchomić dopiero po przesunięciu kurSORA myszy na pewną odległość. Przy takim podejściu musimy wykorzystać nie tylko zdarzenie `MouseDown`, ale również `MouseMove`, które reaguje na ruch myszy. Wymaga ono również zdefiniowania dodatkowych pól, które będą przechowywały stan procesu i położenie kurSORA myszy w momencie naciśnięcia jej lewego przycisku, oraz liczby określającej odległość, na jaką mysz musi być przesunięta, aby proces przeciągania został rozpoczęty. Tę stałą, a dokładnie dwie: dla przesunięcia w pionie i w poziomie, można w zasadzie odczytać z ustawień systemowych. Służy do tego funkcja WinAPI `GetSystemMetrics` z argumentami `SM_CXDRAG` dla przesunięcia w poziomie i `SM_CYDRAG` — w pionie.

Uprościmy jednak wykonanie tego zadania i zamiast prostokątnego obszaru użyjemy okręgu, którego promień ustalimy samodzielnie, np. na 30 pikseli. Ponadto konieczne jest zapamiętanie indeksu elementu, który przenosimy — w argumentach metody MouseMove nie znajdziemy o nim żadnych informacji.

1. W klasie Form1 zdefiniuj nowe pola (listing 10.7).

Listing 10.7. Nowe pola klasy Form1

```
private int indeksSource=-1;
private const int promienProgowy=30;
private bool przeciąganie=false;
private Point polozenieMyszy=new Point();
```

Pole indeksSource będzie przechowywać numer przeciąganego elementu. Stała promienProgowy równa jest liczbie pikseli, na jaką będzie musiał zostać przeniesiony element, aby rozpoczął się właściwy proces *drag & drop*. Flaga przeciąganie będzie unoszona w przypadku naciśnięcia lewego przycisku myszy, a opuszczana przy jego zwolnieniu. Wreszcie w zmiennej polozenieMyszy zapisywane będą współrzędne punktu w momencie naciśnięcia lewego przycisku myszy.

2. Następnie modyfikujemy metodę listBox1_MouseDown zgodnie z wyróżnieniami widocznymi na listingu 10.8. Usuwamy z niej wywołanie metody DoDragDrop. Teraz w momencie naciśnięcia lewego przycisku zapamiętywana jest jedynie pozycja myszy i indeks zaznaczonego elementu.

Listing 10.8. Metoda inicjująca proces przenoszenia z modyfikacjami

```
private void listBox1_MouseDown(object sender, System.Windows.Forms.MouseEventArgs e)
{
    ListBox lbSender=sender as ListBox;
    int indeks=lbSender.IndexFromPoint(e.X,e.Y);
    dragDropSource=sender; //przechowanie referencji dla DragOver
    indeksSource=indeks;

    if (e.Button==MouseButtons.Left && indeks!=-1)
    {
        przeciąganie=true;
        polozenieMyszy.X=e.X;
        polozenieMyszy.Y=e.Y;
    }

    //usunięte polecenie czyszczące referencję dragDropSource
}
```

3. Inicjacja właściwego procesu *drag & drop* przeniesiona zostanie do metody związanej ze zdarzeniem MouseMove. Trzeba ją jednak najpierw utworzyć. Przejedźmy zatem do widoku projektowania, zaznaczmy obie listy, utwórzmy metodę связанную ze zdarzeniem MouseMove i umieścmy w niej blok instrukcji inicjujących proces *drag & drop* oraz usuwających element ze źródłowej listy po zakończeniu całego procesu. Proces zostanie jednak uruchomiony dopiero wtedy, gdy przesunięcie myszy z wcisniętym przyciskiem będzie większe od liczby pikseli określonej przez stałą promienProgowy (listing 10.9).

Listing 10.9. Teraz ta metoda inicjuje proces przenoszenia i upuszczania

```

private void listBox1_MouseMove(object sender, System.Windows.Forms.MouseEventArgs e)
{
    ListBox lbSource = dragDropSource as ListBox;
    if (przeciąganie)
    {
        int dx=e.X-polozenieMyszy.X;
        int dy=e.Y-polozenieMyszy.Y;
        if ((dx*dx+dy*dy)>promienProgowy*promienProgowy)
        {
            //przeniesione z listBox1_MouseDown
            DragDropEffects operacja=
                lbSource.DoDragDrop(lbSource.Items[indeksSource],
                    DragDropEffects.Copy | DragDropEffects.Move);
            if (operacja==DragDropEffects.Move)
                lbSource.Items.RemoveAt(indeksSource);
            przeciąganie=false;
            dragDropSource=null;
        }
    }
}

```

- 4.** Dla porządku utwórz także metodę zdarzeniową do MouseUp (także dla obu list), w której umieścisz polecenie opuszczające flagę przeciąganie w przypadku zwolnienia przycisków myszy (listing 10.10).

Listing 10.10. Flaga w dół

```

private void listBox1_MouseUp(object sender, MouseEventArgs e)
{
    przeciąganie = false;
}

```

Po komplikacji i uruchomieniu aplikacji stwierdzimy, że opóźniona inicjacja procesu przeciągania działa prawidłowo. Pojawił się jednak drobny problem. W trakcie przeciągania elementu, zanim rozpocznie się właściwy proces *drag & drop*, zmienia się zaznaczenie wybranego elementu. Nie wpływa to na wybór elementu dodawanego do drugiej listy (ten jest zapamiętywany w zmiennej indeksSource już w momencie wcisnięcia lewego przycisku myszy), ale może być mylące dla użytkownika aplikacji. Ten błąd można poprawić, ponawiając zaznaczenie elementu w trakcie ruchu myszy, a więc w metodzie listBox1_MouseMove (listing 10.11).

Listing 10.11. Mała latka

```

private void listBox1_MouseMove(object sender, MouseEventArgs e)
{
    ListBox lbSource = dragDropSource as ListBox;
    if (przeciąganie)
    {
        if (indeksSource>-1) lbSource.SelectedIndex = indeksSource;
        ...
    }
}

```

Przenoszenie wielu elementów

Komponent `ListBox` pozwala na zaznaczanie wielu elementów jednocześnie. Aby tę możliwość odblokować, należy ustawić właściwość `SelectionMode` na `MultiSimple` lub `MultiExtended`. W pierwszym trybie lista działa podobnie do zbioru pól opcji, tzn. kliknięcie dowolnego elementu powoduje jego zaznaczenie niezależnie od innych, a ponowne kliknięcie usuwa zaznaczenie. W trybie `MultiExtended` do zaznaczania w ten sposób pojedynczych elementów bez usuwania zaznaczania innych należy wykorzystać klawisz `Ctrl`. Poza tym można także używać klawisza `Shift`, który pozwala na zaznaczanie grupy elementów od obecnie zaznaczonego do klikanego myszą. Oba sposoby zaznaczania wielu elementów powodują kłopoty w procesie *drag & drop*. Kliknięcie myszą w celu przeciągnięcia wielu zaznaczonych elementów może zmienić zaznaczenie. Jednak po kilku eksperymentach można nabrać wprawy wystarczającej do tego, aby poradzić sobie z tą niedogodnością.

1. Wróć do projektu w stanie, w którym możliwe było przenoszenie elementów między dwoma listami bez progu opóźniającego.
2. Przejdź do widoku projektowania i zaznacz obie listy.
3. Ustaw ich właściwości `SelectionMode` na `MultiExtended`.
4. Następnie przejdź do edytora kodu. Zmiany rozpoczęliśmy od metody `listBox1_MouseDown` (listing 10.12). Traci w niej zasadność definiowanie zmiennej lokalnej `indeks`, która do tej pory przechowywała numer przenoszonego elementu. Zamiast tego należy odczytywać listę zaznaczonych elementów z właściwości `ListBox.SelectedItems`.

Listing 10.12. Metoda inicjująca proces *drag & drop*, przystosowana do obsługi wielu elementów

```
private void listBox1_MouseDown(object sender, MouseEventArgs e)
{
    ListBox lbSender = sender as ListBox;
    int indeks = lbSender.IndexFromPoint(e.X, e.Y);
    dragDropSource = sender; // przechowanie referencji dla DragOver

    if (e.Button == MouseButtons.Left && (lbSender.SelectedItems.Count > 0))
    {
        DragDropEffects operacja =
            lbSender.DoDragDrop(lbSender.SelectedItems,
                DragDropEffects.Copy | DragDropEffects.Move);
        if (operacja == DragDropEffects.Move)
            //foreach (int indeks in lbSender.SelectedItemsIndices)
            for (int i = lbSender.SelectedItems.Count - 1; i >= 0; i--)
                lbSender.Items.Remove(lbSender.SelectedItems[i]);
    }

    dragDropSource = null;
}
```

5. Teraz kolej na metodę `listBox1_DragDrop`, w której elementy zaznaczone w jednej z list są dodawane do drugiej (listing 10.13).

Listing 10.13. Do listy dodawane są wszystkie zaznaczane elementy

```
private void listBox1_DragDrop(object sender, DragEventArgs e)
{
    ListBox lbSender = sender as ListBox;
    ListBox lbSource = dragDropSource as ListBox;
    int indeks = lbSender.IndexFromPoint(lbSender.PointToClient(new Point(e.X, e.Y)));
    if (indeks == -1) indeks = lbSender.Items.Count;
    lbSender.Items.Insert(indeks, e.Data.GetData(DataFormats.Text).ToString());
    for(int i=lbSource.SelectedItems.Count-1;i>=0;i--)
        lbSender.Items.Insert(indeks, lbSource.Items[lbSource.SelectedIndices[i]]);
}
```

Warto zwrócić uwagę, że w metodzie `listBox1_MouseDown` argumentem metody `DoDragDrop` jest teraz kolekcja zaznaczonych elementów. Mimo to nie skorzystamy z tego przesyłanego zbioru w metodzie `listBox1_DragDrop`. Zrobimy tak, niestety, że szkodzą dla jej ogólności, ale za to znacznie upraszczając konieczne zmiany. Zamiast tego kolekcję zaznaczonych elementów pobieramy z właściwości `SelectedItems` odczytanej z referencji `dragDropSource` (po zrzutowaniu na typ `ListBox`).

Spójrzmy także na koniec metody `listBox1_MouseDown` (listing 10.12), gdzie usuwamy elementy przeniesione do innej listy. Dlaczego użyłem pętli `for`, a nie pętli `foreach`, skoro przebiegam po wszystkich elementach kolekcji `lbSender.SelectedItems`? Dla tego, że korzystając z pętli `foreach`, usuwalibyśmy jedynie co drugi element (rozdział 3., podrozdział „Kolekcja List”). Założymy, że zaznaczyliśmy elementy od trzeciego (indeks równy 2) do szóstego (indeks równy 5). Wówczas zbiór `lbSender.SelectedItems` zawiera cztery elementy indeksowane od zera do trzech. Po usunięciu zerowego elementu pierwszy spada na zerową pozycję. Jednak pętla `foreach` przechodzi już do elementu, który obecnie znajduje się na miejscu z indeksem równym jeden. W efekcie element, który spadł z miejsca pierwszego na zerowe, nie zostanie usunięty. Usuwany jest więc każdy element o parzystym indeksie, a te o nieparzystym są pomijane. Ten sam problem pojawi się zresztą, jeżeli użyjemy pętli `for` indeksowanej od zera do liczby elementów minus jeden. Dopiero pętla `for` indeksowana od tyłu uwalnia nas od tego problemu — wówczas indeksy elementów nie ulegają zmianie po usuwaniu kolejnych pozycji z listy.

Ten sam problem pojawia się przy dodawaniu przenoszonych elementów do listy docelowej (koniec metody `listBox1_DragDrop` z listingu 10.13). Tu także najwygodniej wykorzystać pętlę `for` indeksowaną od tyłu. Inaczej należałoby w każdej iteracji obliczać indeks elementu, za którym wstawiane są nowe elementy.

Przenoszenie plików

Jak sprawdziliśmy wcześniej, przenoszenie elementów między aplikacjami w platformie .NET nie stanowi problemu. Okazuje się, że stosuje się to również do sytuacji, w której elementy przenoszone są z aplikacji Win32 do aplikacji .NET. Dla przykładu przyjrzyjmy się bardzo typowej sytuacji, w której między aplikacjami przenoszone będą pliki. Źródłem przenoszonych plików może być dowolny menedżer plików, choćby systemowy Eksplorator Windows lub popularny Total Commander. Natomiast apli-

kacją, którą nauczymy reagować na upuszczenie pliku, będzie notatnik zaprojektowany w poprzednim rozdziale. Należy wobec tego wczytać do Visual Studio jego projekt. Musimy go nauczyć reagować na drugi i trzeci etap procesu *drag & drop*. W drugim etapie aplikacja musi rozpoznać, czy przenoszone obiekty to pliki, i tylko jeżeli okaże się to prawdą, pozwalać na ich upuszczenie. Użyjemy do tego zdarzenia DragOver komponentu textBox1. Utworzymy związaną z tym zdarzeniem metodę zdarzeniową, w której zbadamy, czy przenoszony obiekt zawiera odpowiednie dane, a dokładniej dane typu DataFormats.FileDrop. Prezentuję to na listingu 10.14.

Listing 10.14. Tylko pliki będą akceptowane

```
private void textBox1_DragOver(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop))
        e.Effect = DragDropEffects.Move;
}
```

Po związaniu tej metody z polem tekstowym notatnika i przełączeniu właściwości AllowDrop na true kurSOR myszy będzie sygnalizował możliwość upuszczenia pliku. Teraz pozostaje już tylko wczytanie przenoszonego pliku. Zanim do tego przejdziemy, wyświetlimy listę przenoszonych plików, bo przenoszonych plików może być przecież kilka (listing 10.15).

Listing 10.15. Lista przenoszonych plików przekazywana jest w tablicy łańcuchów

```
private void textBox1_DragDrop(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop))
    {
        string[] nazwyPrzenoszonychPlikow =
            e.Data.GetData(DataFormats.FileDrop) as string[];

        string komunikat="Nazwy przenoszonych plików:\n";
        foreach(string nazwaPliku in nazwyPrzenoszonychPlikow)
            komunikat+=nazwaPliku+"\n";
        MessageBox.Show(komunikat);
    }
}
```

Powyższa metoda pokazuje, w jaki sposób odczytać listę nazw przenoszonych plików. A dysponując ich nazwami, możemy przecież zrobić, co tylko zechcemy, np. wczytać pierwszy z nich do pola tekstowego (listing 10.16).

Listing 10.16. Wczytanie przenoszonego pliku do notatnika

```
private void textBox1_DragDrop(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop))
    {
        string[] nazwyPrzenoszonychPlikow = e.Data.GetData(DataFormats.FileDrop)
            as string[];
        // wczytujemy pierwszy plik
    }
}
```

```
string nazwaPliku = nazwyPrzenoszonychPlikow[0];
textBox1.Lines = CzytajPlikTekstowy(nazwaPliku);
int ostatniSlash = nazwaPliku.LastIndexOf('\\');
toolStripStatusLabel1.Text=nazwaPliku.
    ↳Substring(ostatniSlash+1,nazwaPliku.Length-ostatniSlash-1);
}
```

Część III

Dane w aplikacjach dla platformy .NET

Rozdział 11.

LINQ

Od końca 2007 roku, a więc od opublikowania platformy .NET w wersji 3.5, bardzo istotną częścią języka C# stały się operatory LINQ. Akronim LINQ (wymawiany: *link*, choć niektórzy wymawiają: *lincue*) pochodzi od angielskiego *Language Integrated Query* tłumaczonego jako *zapytania zintegrowane z językiem programowania* lub krócej, jednak kosztem wierności tłumaczenia, jako *zintegrowany język zapytań*.

Biorąc pod uwagę, jak fundamentalną rolę odgrywa obecnie SQL w komunikacji między aplikacjami a relacyjnymi bazami danych, idea LINQ wydaje się oczywista, a zarazem przełomowa: wykorzystać pochodzącą z języka SQL intuicję budowania zapytań i pobierać w ten sposób dane z dowolnego źródła. To przenosi programistę na wyższy poziom abstrakcji w postrzeganiu danych, a jednocześnie jeszcze bardziej uniezależnia kod aplikacji od tego, z jakiego konkretnie źródła pochodzą wykorzystywane przez nią dane. Zapytanie LINQ zwraca kolekcję zgodną z interfejsem `IEnumerable<T>`. Kolekcja ta może być modyfikowana i zapisana z powrotem do źródła danych. Dzięki temu praca z danymi jest naturalna i w całości ogranicza się do jednego języka bez „inkluzji” w postaci łańcuchów z zapytaniami SQL. Znika zatem szereg problemów związanych ze stosowaniem osobnego języka przeznaczonego do kontaktów z bazą danych. A przede wszystkim zachowana jest pełna kontrola typów pobieranych danych i ich konwersji w poszczególnych mechanizmach pośredniczących w tym procesie. Co więcej, kontrola typów może być przeprowadzona już w momencie komplikacji.



Poniżej zaprezentuję nowe operatory LINQ na przykładzie technologii LINQ to Objects, w której źródłem danych są kolekcje. W rozdziale 12. przedstawię także LINQ to XML, a w rozdziale 15. — LINQ to SQL. To najważniejsze moduły technologii LINQ.

Operatorы LINQ

Opisane w rozdziale 3. zmiany wprowadzone do języka C# w wersji 3.0 (m.in. słowo kluczowe `var`, wyrażenia lambda, nowy sposób inicjalizacji pól publicznych tworzonego obiektu), jak również opisane w rozdziale 4. metody rozszerzające ściśle związane są z technologią LINQ — technologią będącą w 2008 roku najważniejszą nowością platformy .NET 3.5. Ujednolicila ona dostęp do różnego typu źródeł danych z poziomu

języka C#. Dzięki zmianom w języku C# zapytania LINQ mogą być formułowane w jednej linii kodu z zachowaniem prostoty i intuicyjności zapytań SQL, a jednocześnie podlegają mechanizmowi kontroli typów. Programiści Microsoft, korzystając z możliwości definiowania metod rozszerzających (metod dodawanych do istniejących już klas bez modyfikowania ich kodu), zdefiniowali w przestrzeni nazw System.Linq zbiór takich metod (m.in. Select, Where i OrderBy — tabela 11.1) dodanych do interfejsu `IEnumerable<>` implementowanego przez wszystkie kolekcje. Ich argumentami są wyrażenia lambda, które pozwolą na definiowanie kryteriów, na podstawie których wybierane i porządkowane będą dane ze źródła. Aby było wygodniej, zdefiniowano nowe operatory związane z powyższymi metodami: from, select, where i orderby. To dzięki nim struktura zapytania LINQ staje się podobna do struktury zapytania SQL.

Tabela 11.1. Metody rozszerzające składające się na technologię LINQ

Rodzaj operacji	Metody rozszerzające
Pobieranie danych	Select, SelectMany
Sortowanie	OrderBy, ThenBy, OrderByDescending, ThenByDescending, Reverse
Filtrowanie	Where
Operacje arytmetyczne	Aggregate, Average, Count, LongCount, Max, Min, Sum
Konwersja	Cast, OfType,ToArray, ToDictionary,ToList, ToLookup, ToSequence
Pobieranie elementu	Element, DefaultIfEmpty, ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Porównywanie	SequenceEqual
Tworzenie	Empty, Range, Repeat
Grupowanie	GroupBy
Łączenie	GroupJoin, Join
Wybór i pomijanie elementów	Skip, SkipWhile, Take, TakeWhile
Kwantyfikatory	All, Any, Contains
Operacje na zbiorach	Concat, Distinct, Except, Intersect, Union

Związanie rozszerzeń LINQ z interfejsem `IEnumerable<>` oznacza, że zwykłe kolekcje platformy .NET (takie jak `List<>`), które implementują ten interfejs, mogą być źródłem danych w technologii LINQ to Objects. Jeżeli zatem dysponujemy kolekcją obiektów, możemy je dowolnie filtrować, sortować, analizować w dowolny sposób, jak również łączyć z inną kolekcją.

Aby wypróbować technologię LINQ, stwórzmy nowy projekt aplikacji konsolowej i w pliku `Program.cs` zdefiniujmy klasę przechowującą poszczególne „rekordy” danych (czyli po prostu elementy kolekcji). Niech będzie to klasa `Osoba` z polami `Id`, `Imię`, `Nazwisko`, `NumerTelefonu` i `Wiek` (listing 11.1).

Listing 11.1. Klasa Osoba

```
class Osoba
{
    public int Id;
    public string Imię, Nazwisko;
```

```
    public int NumerTelefonu;
    public int Wiek;
}
```

W klasie Program zdefiniujemy statyczne pole będące listą przechowującą kilka obiektów typu Osoba. Lista ta będzie źródłem danych w dalszych przykładach z tego rozdziału (listing 11.2).

Listing 11.2. Lista osób

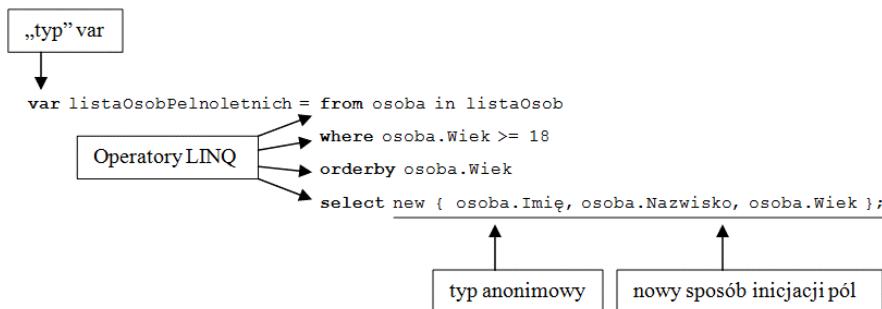
```
class Program
{
    //źródło danych
    static List<Osoba> listaOsob = new List<Osoba>
    {
        new Osoba { Id = 1, Imię = "Jan", Nazwisko = "Kowalski",
                    NumerTelefonu = 7272024, Wiek = 39 },
        new Osoba { Id = 2, Imię = "Andrzej", Nazwisko = "Kowalski",
                    NumerTelefonu = 7272020, Wiek = 29 },
        new Osoba { Id = 3, Imię = "Maciej", Nazwisko = "Bartnicki",
                    NumerTelefonu = 7272021, Wiek = 42 },
        new Osoba { Id = 4, Imię = "Witold", Nazwisko = "Mocarz",
                    NumerTelefonu = 7272022, Wiek = 26 },
        new Osoba { Id = 5, Imię = "Adam", Nazwisko = "Kowalski",
                    NumerTelefonu = 7272023, Wiek = 6 },
        new Osoba { Id = 6, Imię = "Ewa", Nazwisko = "Mocarz",
                    NumerTelefonu = 7272025, Wiek = 11 }
    };
    //ciąg dalszy klasy Program
```

Pobieranie danych (filtrowanie i sortowanie)

Teraz przećwiczymy podstawowe możliwości technologii LINQ. Rozpoczniemy od pobrania danych z listy za pomocą zapytania widocznego na rysunku 11.1. Należy umieścić je w metodzie Program.Main. Z pierwotnej listy wybierzmy tylko osoby pełnoletnie. Ponadto będą nas interesować tylko pola Imię, Nazwisko i Wiek. Wobec tego zwrażając dane, utworzymy nowe obiekty anonimowe zawierające tylko te trzy informacje.

Dzięki typowi var tworzenie typu anonimowego nie stwarza problemów przy pobieraniu danych, niezależnie od jego złożoności. Jeżeli jednak chcielibyśmy skonwertować otrzymaną listę z powrotem na kolekcję List<Osoba>, ale z mniejszą liczbą elementów, napotkamy trudności. Do takich celów lepiej zachować oryginalny typ danych (tj. rekordy typu Osoba), co prowadzi do następującej formy zapytania:

```
var listaOsobPelnoletnich = from osoba in listaOsob
                            where osoba.Wiek >= 18
                            orderby osoba.Wiek
                            select osoba;
```



Rysunek 11.1. Struktura zapytania LINQ

Wówczas możliwe jest użycie metody rozszerzającej `ToList`, konwertującej otrzymany wynik z powrotem na listę:

```
List<Osoba> podlista = listaOsobPelnoletnich.ToList<Osoba>();
```

Najprostsza rzecz, jaką możemy zrobić z pobranymi danymi, to oczywiście pokazanie ich na ekranie (w konsoli):

```
Console.WriteLine("Lista osób pełnoletnich:");
foreach (var osoba in listaOsobPelnoletnich)
    Console.WriteLine(osoba.Imię + " " + osoba.Nazwisko + " (" + osoba.Wiek + ")");
```

Analiza pobranych danych

Korzystając z metod rozszerzających interfejs `IEnumerable<>`, możemy zbadać, jaki jest maksymalny wiek otrzymanych w wyniku zapytania osób, jaki jest ich wiek średni lub jaka jest suma ich lat. Służą do tego funkcje `Max` (oczywiście jest również `Min`), `Average` i `Sum`. Należy jedynie wskazać, które pole analizowanych obiektów lub jaka kombinacja tych pól ma być sumowana lub uśredniana. Korzystamy tu z przedstawionych wyżej wyrażeń lambda. W poniższym przykładzie analizowanym polem jest `Wiek`:

```
Console.WriteLine("Wiek najstarszej osoby: " +
    listaOsobPelnoletnich.Max(osoba => osoba.Wiek));
Console.WriteLine("Średni wiek osób pełnoletnich: " +
    listaOsobPelnoletnich.Average(osoba => osoba.Wiek));
Console.WriteLine("Suma lat osób pełnoletnich: " +
    listaOsobPelnoletnich.Sum(osoba => osoba.Wiek));
```

Wybór elementu

Mozemy również zidentyfikować osobę, która ma najwięcej lat. Najwygodniej użyć do tego metody `Single`, której argumentem jest delegacja do metody, zwracającej prawdę, gdy właściwości badanego elementu są zgodne z naszymi oczekiwaniemi — w naszym wypadku gdy wiek osoby jest równy maksymalnemu wiekowi osób w kolekcji:

```
var najstarszaOsoba=listaOsobPelnoletnich.Single(  
    osoba1=>(osoba1.Wiek==listaOsobPelnoletnich.Max(osoba => osoba.Wiek)));  
Label8.Text += "<p>Najstarsza osoba: " + najstarszaOsoba.Imie + " " +  
    najstarszaOsoba.Nazwisko + " (" + najstarszaOsoba.Wiek + ")";
```

Weryfikowanie danych

Korzystając z metod rozszerzających, możemy sprawdzić, czy lista osób (wynik zapytania LINQ lub pierwotna lista osób) spełnia jakiś warunek, np. czy wszystkie osoby z listy mają więcej niż 18 lat:

```
bool czyWszystkiePełnoletnie = listaOsobPelnoletnich.All(osoba => (osoba.Wiek > 18));
```

Można też sprawdzić, czy jakakolwiek osoba spełnia ten warunek:

```
bool czyZawieraPełnoletnią = listaOsob.Any(osoba => (osoba.Wiek > 18));
```

Można również sprawdzić, czy w podzbiorze otrzymanym jako wynik zapytania znajduje się konkretny obiekt. Służy do tego metoda Contains.

Prezentacja w grupach

Załóżmy, że w tabeli mamy dane osób z wielu rodzin. Wówczas wygodne może być pokazanie osób o tym samym nazwisku w osobnych grupach. W prostym przykładzie widocznym na listingu 11.3 zaprezentujemy osobno osoby o tym samym nazwisku, ale w jego męskiej i żeńskiej formie. Warunek umieszczający w osobnych grupach może być jednak dowolną funkcją C#, moglibyśmy zatem sprawdzać, czy sam rdzeń nazwisk jest taki sam (po odjęciu końcówek -ski i -ska).

Listing 11.3. Grupowanie danych w zapytaniu

```
var grupyOsobO TymSamymNazwisku = from osoba in listaOsob  
                                         group osoba by osoba.Nazwisko into grupa  
                                         select grupa;  
Console.WriteLine("Lista osób pogrupowanych według nazwisk:");  
foreach (var grupa in grupyOsobO TymSamymNazwisku)  
{  
    Console.WriteLine("Grupa osób o nazwisku " + grupa.Key);  
    foreach (Osoba osoba in grupa) Console.WriteLine(osoba.Imię  
        + " " + osoba.Nazwisko);  
    Console.WriteLine();  
}
```

Łączenie zbiorów danych

Możliwe jest również łączenie zbiorów danych. Połączmy dla przykładu listę osób pełnoletnich i listę kobiet. Te ostatnie rozpoznamy po ostatniej literze imienia — zobacz warunek za operatorem where w poleceniu tworzącym zbiór listakobiet na linii 11.4. W ogólności jest to niepewna metoda rozpoznawania płci (przykłady to Kuba, Barnaba czy Bonawentura), ale dla naszych potrzeb wystarczająca.

Listing 11.4. Dwa pierwsze zapytania tworzą dwa zbiorы, które łączone są w trzecim zapytaniu

```
var listaOsobPelnoletnich = from osoba in listaOsob
                               where osoba.Wiek>=18
                               orderby osoba.Wiek
                               select new { osoba.Imię, osoba.Nazwisko, osoba.Wiek };
var listaKobiet = from osoba in listaOsob
                   where osoba.Imię.EndsWith("a")
                   select new { osoba.Imię, osoba.Nazwisko, osoba.Wiek };

var listaPelnoletnich_I_Kobiet = listaOsobPelnoletnich.Concat(listaKobiet);
```

Do łączenia zbiorów użyliśmy metody rozszerzającej Concat. W efekcie uzyskamy zbiór o wielkości równej sumie wielkości obu łączonych zbiorów, zawierający wszystkie elementy z obu zbiorów. Jeżeli jakiś element powtarza się w obu zbiorach, będzie także powtarzał się w zbiorze wynikowym. Jeżeli chcemy się pozbyć zdublowanych elementów, należy użyć metody rozszerzającej Distinct:

```
var listaPelnoletnich_I_Kobiet = listaOsobPelnoletnich.Concat(listaKobiet).Distinct();
```

Ten sam efekt, tj. sumę mnogościową z wyłączeniem powtarzających się elementów, można uzyskać, tworząc unię kolekcji za pomocą metody rozszerzającej Union:

```
var listaPelnoletnich_I_Kobiet = listaOsobPelnoletnich.Union(listaKobiet);
```

Oprócz sumy mnogościowej można zdefiniować również iloczyn dwóch zbiorów (część wspólną). Służy do tego metoda rozszerzająca Intersect:

```
var listaKobietPelnoletnich = listaOsobPelnoletnich.Intersect(listaKobiet);
```

Łatwo się domyślić, że możliwe jest również uzyskanie różnicy zbiorów. Dla przykładu znajdzmy listę osób pełnoletnich niebędących kobietami:

```
var listaPelnoletnichNiekobiet = listaOsobPelnoletnich.Except(listaKobiet);
```

Łączenie danych z różnych źródeł w zapytaniu LINQ — operator join

Łączenie zbiorów może dotyczyć nie tylko dwóch kolekcji o takiej samej strukturze encji. Możemy również utworzyć nową kolekcję z dwóch kolekcji zawierających różne informacje na temat tych samych obiektów. Założymy, że mamy dwa zbiorы przechowy-

wujące dane o tych samych osobach. W jednym mamy numery telefonów, w drugim personalia osób. Na potrzeby ilustracji obie kolekcje możemy utworzyć z listy listaOsob za pomocą następujących zapytań:

```
var listaTelefonów = from osoba in listaOsob
                      select new {osoba.Id, osoba.NumerTelefonu};
var listaPersonaliów = from osoba in listaOsob
                      select new {osoba.Id, osoba.Imię, osoba.Nazwisko};
```

Relacja między nimi opiera się na wspólnym polu-kluczu Id jednoznacznie identyfikującym właściciela numeru telefonu i osobę o danym imieniu i nazwisku.

Załóżmy teraz, że z tych dwóch źródeł danych chcemy utworzyć jedną spójną kolekcję zawierającą zarówno numery telefonów, jak i personalia ich właścicieli. Do tego możemy użyć operatora join w zapytaniu LINQ:

```
var listaPersonaliówZTelefonami = from telefon in listaTelefonów
                                      join personalia in listaPersonaliów
                                      on telefon.Id equals personalia.Id
                                      select new
                                      {
                                          telefon.Id,
                                          personalia.Imię,
                                          personalia.Nazwisko,
                                          telefon.NumerTelefonu
                                      };
;
```

Składnia zapytania zrobiła się nieco skomplikowana, ale nie na tyle, żeby przy odrobinie wysiłku nie dało się jej zrozumieć. Za operatorem from znajdują się teraz dwie sekcje element in źródło rozdzielone operatorem join. W powyższym przykładzie są to telefon in listaTelefonów oraz personalia in listaPersonaliów. Za nimi następuje sekcja on, w której umieszczamy warunek porównujący wybrane pola z obu źródeł. Sprawdzamy równość pól Id w obu kolekcjach, tj. telefon.Id equals personalia.Id. Jeżeli są równe, to z tych dwóch rekordów (z różnych źródeł) tworzony jest obiekt będący elementem zwracanej przez zapytanie kolekcji. Powstaje więc iloczyn kartezjański dwóch zbiorów, a raczej jego podzbiór wyznaczony przez warunek znajdujący się w zapytaniu za słowem kluczowym on.

Możliwość modyfikacji danych źródła

Jeżeli w wyniku zapytania LINQ utworzymy listę osób pełnoletnich:

```
var nowaListaOsobPełnoletnich = from osoba in listaOsob
                                       where osoba.Wiek >= 18
                                       orderby osoba.Wiek
                                       select osoba;
```

to dane zgromadzone w nowej kolekcji listaOsobPełnoletnich nie są kopiowane. Warunkiem jest jednak, że elementy kolekcji-źródła, czyli w powyższym przykładzie klasa Osoba jest typu referencyjnego, czyli jest właściwie klasą, a nie strukturą. Do nowej kolekcji dołączane są referencje z oryginalnego zbioru. To oznacza, że dane można

modyfikować, a zmiany będą widoczne także w oryginalnej kolekcji. Dla przykładu zmieńmy pola pierwszej pozycji na liście (tj. ze względu na sortowanie wyniku zapytania według wieku — najmłodszej):

```
Osoba pierwszyNaLiscie = nowaListaOsobPełnoletnich.First<Osoba>();  
pierwszyNaLiscie.Imię = "Karol";  
pierwszyNaLiscie.Nazwisko = "Bartnicki";  
pierwszyNaLiscie.Wiek = 31;
```

Po tej zmianie wyświetlimy dane z oryginalnego źródła (tj. listaOsob), a przekonamy się, że Witolda Mocarza (26) zastąpił Karol Bartnicki (31).

Ciekawe jest, że jeżeli po tej zmianie wyświetlimy listę osób z kolekcji nowaLista<OsobaPełnoletnich> polecением:

```
foreach (var osoba in nowaListaOsobPełnoletnich)  
    Console.WriteLine(osoba.Imię + " " + osoba.Nazwisko + " (" + osoba.Wiek + ")");
```

to mimo zmiany w oryginalnej kolekcji nadal otrzymane wyniki będą prawidłowo posortowane według wieku (nowa osoba wskoczy na drugą pozycję). Zapytanie jest bowiem ponawiane przy każdej próbie odczytu danych z kolekcji nowaListaOsobPełnoletnich.

W przypadku LINQ to Objects polecenie umieszczone za operatorem `select` można zmienić w taki sposób, aby dane były kopiowane do nowych obiektów, tj.

```
var nowaListaOsobPełnoletnich = from osoba in listaOsob  
        where osoba.Wiek >= 18  
        orderby osoba.Wiek  
        select new Osoba { Id = osoba.Id, Imię =  
            →osoba.Imię,  
            Nazwisko = osoba.Nazwisko,  
            NumerTelefonu =  
            →osoba.NumerTelefonu,  
            Wiek = osoba.Wiek };
```

Wówczas cała ta filozofia bierze w łeb i edycja kolekcji będącej wynikiem zapytania nie wpływa na zawartość oryginału.

* * *

Jak widać, z technicznego punktu widzenia technologia LINQ to przede wszystkim metody rozszerzające, zdefiniowane dla interfejsu `IEnumerable<T>`. Jej możliwości są jednak spore. Zapytania LINQ pozwalają na pobieranie danych, a metody rozszerzające — na ich analizę i przetwarzanie. W powyższym rozdziale przedstawiony został przykład, w którym źródłem danych jest kolekcja obiektów (LINQ to Objects). To jednak zaledwie jeden element z całego zbioru technologii LINQ. Pozostałe to LINQ to DataSet, w której źródłem danych jest komponent `DataSet` reprezentujący dane z dowolnej bazy danych, LINQ to SQL (rozdział 15.), w której dostęp do danych z tabel SQL Servera jest niemal tak samo prosty jak w przypadku LINQ to Objects, i LINQ to XML, w której źródłem danych są pliki XML (rozdział 12.). Należy wspomnieć również o LINQ to Entities związanej z Entity Framework (rozdział 17.). W każdej z technologii LINQ z punktu widzenia użytkownika dostęp do danych realizowany jest w podobny sposób.

Rozdział 12.

Przechowywanie danych w plikach XML. LINQ to XML

W tym rozdziale zajmiemy się zapisywaniem i odczytywaniem plików XML (z ang. *Extensible Markup Language*) za pomocą technologii LINQ to XML. XML jest językiem pozwalającym zapisać do pliku dane w strukturze hierarchicznej. Pliki te są plikami tekstowymi, a więc są w pełni przenośne między różnymi systemami. Co więcej, parsery XML są już na tyle powszechnie, że ta przenośność ma istotne znaczenie praktyczne.

Podstawy języka XML

Podstawowa zasada organizacji pliku XML jest podobna do HTML, tzn. struktura pliku opiera się na znacznikach z możliwością ich zagnieżdżania. Wewnątrz pary znaczników (otwierającego i zamkajającego) mogą być przechowywane dane. Jednak — w odróżnieniu od języka HTML — tutaj nazwy znaczników określamy sami. Nie ma zbioru predefiniowanych znaczników z określona semantyką.

Przedstawię najpierw zasadnicze elementy języka XML i strukturę prostego pliku, aby następnie przejść do przykładu, w którym w pliku XML przechowamy ustawienia aplikacji.

Deklaracja

W dokumencie XML może znajdować się tylko jedna deklaracja i jeżeli istnieje, musi być w pierwszej linii pliku. Deklaracja identyfikuje plik jako dokument XML:

```
<?xml version="1.0"?>
```

Może zawierać wersję specyfikacji XML, z którą plik jest zgodny, oraz ewentualnie sposób kodowania i inne własności dokumentu.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

Elementy

Element to część dokumentu XML, która przechowuje dane lub inne zagnieżdżone elementy. Tworzy go para znaczników: otwierający i zamykający, oraz ewentualne dane między nimi. Każdy dokument musi zawierać przynajmniej jeden element. Oto przykładowy pusty, tj. niezawierający żadnych danych dokument XML z jednym elementem o nazwie opcje:

```
<?xml version="1.0"?>
<opcje>
</opcje>
```

A oto przykład „wielopoziomowego” dokumentu XML zawierającego dane o położeniu i rozmiarach okna:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<opcje>
  <okno>
    <pozycja>
      <X>189</X>
      <Y>189</Y>
    </pozycja>
    <wielkość>
      <Szer>300</Szer>
      <Wys>300</Wys>
    </wielkość>
  </okno>
</opcje>
```

Wszystkie dane zamknięte są w obowiązkowym elemencie, który nazwaliśmy opcje. Jego podelement okno zawiera dwa podpodelementy pozycja i wielkość, dopiero w nich znajdują się elementy przechowujące dane, m.in. X i Y¹.

Atrybuty

Atrybuty są tym, czym parametry w znacznikach HTML. Do każdego elementu można dodać kilka atrybutów przechowujących dodatkowe informacje, np.:

```
<okno nazwa="Form1">
```

Komentarze

Podobnie jak w HTML-u komentarze to znaczniki postaci:

```
<!--Parametry aplikacji -->
```

* * *

¹ W przeciwieństwie do HTML-a w języku XML wielkość liter ma znaczenie.

To oczywiście tylko wstęp do języka XML. Znamy już wprawdzie najprostsze, a zazwyczaj najczęściej spotykane jednostki składniowe tego języka, ale w ogóle nie wspomniałem o takich terminach jak instrukcje przetwarzania, sekcje CDATA, *encje* (ang. *entity*), przestrzenie nazw czy szablony. Tych kilka przedstawionych wyżej terminów wystarczy do typowych zastosowań, w tym do przechowywania w plikach XML danych aplikacji.

LINQ to XML

Najpoważniejszym z atutów technologii LINQ, podkreślonym już w poprzednim rozdziale, jest unifikacja sposobów dostępu do różnego typu źródeł danych. Dzięki temu osoba, która pozna operatory LINQ i nauczy się budować zapytania LINQ, potrafi pobrać dane z kolekcji, tabeli bazy danych czy pliku XML. Jednak w przypadku plików XML twórcy LINQ poszli o krok dalej: pozwolili nie tylko na budowanie zapytań, ale również zaproponowali nowy sposób zapisywania danych. I od tej części LINQ to XML zaczniemy.

Tworzenie pliku XML za pomocą klas XDocument i XElement

Stwórzmy projekt aplikacji Windows Forms. Wykorzystamy w nim pliki XML do przechowywania położenia i rozmiaru okna (podrozdział „Ustawienia aplikacji” z rozdziału 9.). Informacje te będą zapisywane przy zamknięciu aplikacji i odtwarzane przy jej uruchamianiu. Zaczniemy od zapisywania. Na listingu 12.1 przedstawiam realizujący to zadanie fragment kodu. Umieściłem go w metodzie związanego ze zdarzeniem FormClosed. Oznacza to, że kod z listingu 12.1 zostanie uruchomiony w momencie zamknięcia okna. Wszystkie klasy użyte w poniższym listingu należą do przestrzeni nazw System.Xml.Linq, zatem do grupy poleceń `using` należy dodać jeszcze jedno, które dołączy także tę przestrzeń.

Listing 12.1. Plik XML tworzony za pomocą klas LINQ to XML

```
private void Form1_FormClosed(object sender, FormClosedEventArgs e)
{
    XDocument xml = new XDocument(
        new XDeclaration("1.0", "utf-8", "yes"),
        new XComment("Parametry aplikacji"),
        new XElement("opcje",
            new XElement("okno",
                new XAttribute("nazwa", this.Text),
                new XElement("pozycja",
                    new XElement("X", this.Left),
                    new XElement("Y", this.Top)
                ),
                new XElement("wielkość",
                    new XElement("Szer", this.Width),
                    new XElement("Wys", this.Height)
                )
            )
        )
    );
}
```

```
        )  
    );  
  
    xml.Save("Ustawienia.xml");  
}
```

Na pierwszy rzut oka kod ten może wydawać się dość zawijły. W orientacji powinny pomóc wcięcia w kodzie. Niemal cały kod stanowi konstruktor klasy `XDocument`, który podobnie jak widoczna w kodzie klasa `XElement` może przyjmować dowolną liczbę argumentów². W przykładzie z listingu 12.1 konstruktor klasy `XDocument` przyjmuje trzy argumenty: deklarację (obiekt typu `XDeclaration`), komentarz (obiekt typu `XComment`) i element główny o nazwie opcje (obiekt `XElement`). Nazwa elementu podawana jest w pierwszym argumencie konstruktora klasy `XElement`. Jego argumentem jest natomiast tylko jeden podelement o nazwie okno. Ten ma z kolei dwa podpodelementy (pozycja i wielkość), a każdy z nich po dwa podpodpodelementy. Tak utworzone drzewo można zapisać do pliku XML, korzystając z metody `Save` (ostatnia linia metody z listingu 12.1). W efekcie po uruchomieniu aplikacji i jej zamknięciu na dysku w podkatalogu `bin\Debug` katalogu, w którym zapisaliśmy projekt, pojawi się plik `Ustawienia.xml` widoczny na listingu 12.2.

Listing 12.2. Plik XML uzyskany za pomocą metody z listingu 12.1

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>  
<!--Parametry aplikacji-->  
<opcje>  
  <okno nazwa="Przykład wykorzystania LINQ to XML">  
    <pozycja>  
      <X>22</X>  
      <Y>29</Y>  
    </pozycja>  
    <wielkość>  
      <Szer>300</Szer>  
      <Wys>300</Wys>  
    </wielkość>  
  </okno>  
</opcje>
```

Taki sposób przygotowania kodu odpowiedzialnego za tworzenie pliku XML, w którym struktura tego pliku odwzorowana jest w argumentach konstruktora, nie zawsze jest wygodny. Nie można go zastosować, np. jeżeli struktura pliku XML nie jest z góry ustalona i zależy od samych danych, co wpływa np. na liczbę elementów. Wówczas poszczególne elementy możemy utworzyć osobno, deklarując obiekty, a następnie zbudować z nich drzewo z wykorzystaniem metod `Add` obiektów typu `XDocument` i `XElement`. Pokazuję to na listingu 12.3. Wówczas jest miejsce na instrukcje warunkowe lub instrukcje wielokrotnego wyboru, którymi możemy wpływać na strukturę tworzzonego pliku XML.

² Zobacz rozdział 3., podrozdział „Tablice jako argumenty metod oraz metody z nieokreślona liczbą argumentów”.

Listing 12.3. Osobne tworzenie obiektów odpowiadających poszczególnym elementom

```
private void Form1_FormClosed(object sender, FormClosedEventArgs e)
{
    //definiowanie obiektów
    XDocument xml = new XDocument();
    XDeclaration deklaracja = new XDeclaration("1.0", "utf-8", "yes");
    XComment komentarz = new XComment("Parametry aplikacji");
    XElement opcje = new XElement("opcje");
    XElement okno = new XElement("okno");
    XAttribute nazwa = new XAttribute("nazwa", this.Text);
    XElement pozycja = new XElement("pozycja");
    XElement X = new XElement("X", this.Left);
    XElement Y = new XElement("Y", this.Top);
    XElement wielkość = new XElement("wielkość");
    XElement Szer = new XElement("Szer", this.Width);
    XElement Wys = new XElement("Wys", this.Height);

    //budowanie drzewa (od gałęzi)
    pozycja.Add(X);
    pozycja.Add(Y);
    wielkość.Add(Szer);
    wielkość.Add(Wys);
    okno.Add(nazwa);
    okno.Add(pozycja);
    okno.Add(wielkość);
    opcje.Add(okno);

    xml.Declaration=deklaracja;
    xml.Add(komentarz);
    xml.Add(opcje);

    //zapis do pliku
    xml.Save("Ustawienia.xml");
}
```

Pobieranie wartości z elementów o znanej pozycji w drzewie

Plik XML jest zapisywany w momencie zamknięcia aplikacji. Zapisane w nim informacje odczytamy w momencie uruchamiania aplikacji i użyjemy ich do odtworzenia pozycji i wielkości okna sprzed zamknięcia. Klasy LINQ to XML pozwalają na szybkie odczytanie tylko tych elementów, które są nam potrzebne, bez konieczności analizowania całej kolekcji elementów i śledzenia znaczników otwierających i zamykających elementy, do czego zmusza np. klasa `XmlTextReader`. Różnica bierze się z faktu, że w LINQ to XML cały plik wczytywany jest do pamięci i automatycznie parsowany. Metoda z listingu 12.4 prezentuje sposób odczytania nazwy okna i czterech liczb opisujących jego położenie i wielkość.

Listing 12.4. Odczyt atrybutu elementu okno i czterech jego podelementów

```
private void Form1_Load(object sender, EventArgs e)
{
    try
    {
```

```

XDocument xml = XDocument.Load("Ustawienia.xml");

// odczytanie tytułu okna
this.Text = xml.Root.Element("okno").Attribute("nazwa").Value;

// odczytanie pozycji i wielkości
 XElement pozycja = xml.Root.Element("okno").Element("pozycja");
 this.Left = int.Parse(pozycja.Element("X").Value);
 this.Top = int.Parse(pozycja.Element("Y").Value);

 XElement wielkość = xml.Root.Element("okno").Element("wielkość");
 this.Width = int.Parse(wielkość.Element("Szer").Value);
 this.Height = int.Parse(wielkość.Element("Wys").Value);
}
catch (Exception exc)
{
    MessageBox.Show("Błąd podczas odczytywania pliku XML:\n" + exc.Message);
}
}

```

Statyczna metoda Load klasy XDocument pozwala na wczytanie całego pliku XML³. Następnie jego zawartość (drzewo elementów) udostępniana jest za pomocą właściwości obiektu XDocument. Element główny dostępny jest dzięki właściwości Root, jego zawartość zaś przy użyciu metod Elements, Nodes i Descendants zwracających kolekcje elementów. Ostatnia z nich pozwala na uzyskanie kolekcji elementów z różnych poziomów drzewa, przefiltrowanych za pomocą nazwy podanej w argumencie tej metody. Poszczególne elementy o znanych nazwach można odczytać, korzystając z metody Element, której argumentem jest nazwa elementu. Zwracana wartość to referencja do obiektu typu XElement. Na tej metodzie bazuje kod z listingu 12.4. Aby odczytać wartość elementu tym sposobem, musimy znać jedynie ścieżkę do odpowiedniej gałęzi drzewa. Równie łatwo można odczytać inne informacje. Na listingu 12.5 pokazuję trzy przykłady.

Listing 12.5. Odczytywanie nazwy elementu głównego, nazw wszystkich podelementów i wersji XML

```

private void button1_Click(object sender, EventArgs e)
{
    XDocument xml = XDocument.Load("Ustawienia.xml");

    //wersja XML
    string wersja = xml.Declaration.Version;
    MessageBox.Show("Wersja XML: " + wersja);

    //odczytanie nazwy głównego elementu
    string nazwaElementuGłownego = xml.Root.Name.LocalName;
    MessageBox.Show("Nazwa elementu głównego: " + nazwaElementuGłownego);

    //kolekcja podelementów ze wszystkich poziomów drzewa
    IEnumerable< XElement> wszystkiePodelementy = xml.Root.Descendants();
    string s = "Wszystkie podelementy:\n";
    foreach ( XElement podelement in wszystkiePodelementy) s += podelement.Name + "\n";
    MessageBox.Show(s);
}

```

³ Argumentem tej metody może być nie tylko ścieżka pliku na lokalnym dysku, ale również adres internetowy (URI), np. <http://www.nbp.pl/kursy/xml/LastC.xml>.

```
// kolekcja podelementów elementu okno
IEnumerable< XElement > podelementyOkno = xml.Root.Element("okno").Elements();
s = "Podelementy elementu okno:\n";
foreach ( XElement podelement in podelementyOkno ) s += podelement.Name + "\n";
MessageBox.Show(s);
}
```

W ostatnim przykładzie pokazuję, w jaki sposób pobrać nazwy i wartości wszystkich podelementów konkretnego elementu. Umożliwia to przeprowadzenie samodzielnej analizy ich zawartości i ewentualny odczyt informacji z plików XML, których struktura nie jest ustalona z góry.

Odwzorowanie struktury pliku XML w kontrolce TreeView

Elementy w pliku XML tworzą strukturę drzewa. Wobec tego sam narzuca się pomysł, aby zaprezentować je w kontrolce TreeView. Będzie to przy okazji doskonała okazja, aby tę kontrolkę poznać. Do realizacji tego zadania użyjemy rekurencyjnego wywoływania metody DodajElementDoWęzla, które znakomicie ułatwia wędrówkę po gałęziach drzewa pliku XML i jednocześnie budowanie drzewa węzłów dla kontrolki TreeView (listing 12.6).

Listing 12.6. Zmienna poziom nie jest w zasadzie używana, ale umożliwia łatwą orientację w ilości rozgałęzień, jakie mamy już za sobą, wędrując po drzewie

```
private void DodajElementDoWęzla(XElement elementXml, TreeNode wezelDrzewa, int poziom)
{
    poziom++;

    IEnumerable< XElement > elementy = elementXml.Elements();
    foreach ( XElement element in elementy )
    {
        string opis = element.Name.LocalName;
        if (!element.HasElements) opis += " (" + element.Value + ")";
        TreeNode nowyWęzeł = new TreeNode(opis);
        wezelDrzewa.Nodes.Add(nowyWęzeł);
        DodajElementDoWęzla(element, nowyWęzeł, poziom);
    }
}

private void button6_Click(object sender, EventArgs e)
{
    try
    {
        XDocument xml = XDocument.Load("Ustawienia.xml");

        treeView1.BeginUpdate();
        treeView1.Nodes.Clear();

        TreeNode wezelGlowny = new TreeNode(xml.Root.Name.LocalName);
        DodajElementDoWęzla(xml.Root, wezelGlowny, 0);
        treeView1.Nodes.Add(wezelGlowny);
    }
}
```

```

        wezelGlowny.ExpandAll();
        treeView1.EndUpdate();
    }
    catch (Exception exc)
    {
        MessageBox.Show("Błąd podczas odczytywania pliku XML:\n" + exc.Message);
    }
}

```

Warto sprawdzić, co zobaczymy w kontrolce TreeView, jeżeli zamiast pliku *Ustawienia.xml* wczytamy plik <http://www.nbp.pl/kursy/xml/LastC.xml>. Aby się o tym przekonać, pierwszą linię metody button6_Click należy zastąpić przez:

```
XDocument xml = XDocument.Load("http://www.nbp.pl/kursy/xml/LastC.xml");
```

Przenoszenie danych z kolekcji do pliku XML

Plik XML może służyć zarówno do zapisu danych, które mają wielopoziomową strukturę drzewa, a elementy się nie powtarzają, jak i do przechowywania danych zgromadzonych w wielu rekordach (encjach) o tej samej strukturze — odpowiedników tabel. Taką strukturę ma chociażby plik pobierany ze strony NBP. Przechowywanie tabel nie jest tym, do czego obiektowe bazy danych zostały wymyślone, ale w praktyce takie struktury plików XML spotyka się bardzo często. Może to być wygodne, gdy chcemy przenieść dane z jednej bazy danych do drugiej lub przeprowadzić wymianę danych między aplikacjami. Ważna jest przy tym przenośność plików XML, które dla systemów operacyjnych są zwykłymi plikami tekstowymi. Co prawda do tego celu zazwyczaj wykorzystywana jest serializacja, ale można ją także z łatwością zrealizować w ramach LINQ to XML.

Podczas budowania programu pozwalającego na przenoszenie danych między bazami danych lub aplikacjami za pomocą plików XML zasadniczym problemem jest zatem konwersja kolekcji lub tabeli do pliku XML i późniejsza konwersja odwrotna pliku XML do kolekcji lub tabeli. Zajmiemy się pierwszym typem konwersji, a więc z kolekcji do pliku XML. Przykład takiej konwersji widoczny jest na listingu 12.7. Wykorzystałem w nim kolekcję osób wraz z ich personaliami, wiekiem i numerami telefonów przygotowaną w poprzednim rozdziale (listingi 11.1 i 11.2).

Listing 12.7. Konwersja kolekcji do pliku XML

```

private void button2_Click(object sender, EventArgs e)
{
    XDocument xml = new XDocument(
        new XDeclaration("1.0", "utf-8", "yes"),
        new XElement("ListaOsob",
            from osoba in listaOsob
            orderby osoba.Wiek
            select new XElement("Osoba",
                new XAttribute("Id", osoba.Id),
                new XElement("Imię", osoba.Imię),
                new XElement("Nazwisko", osoba.Nazwisko),
                new XElement("NumerTelefonu", osoba.NumerTelefonu),
                new XElement("Wiek", osoba.Wiek)
            )
        )
    );
}

```

```

        )
    );
}

xml.Save("ListaOsob.xml");
}

```

W powyższej metodzie wykorzystaliśmy fakt, że argumentem konstruktora `XElement` może być kolekcja elementów (obiektów `XElement`). Kolekcję tę utworzyliśmy za pomocą zapytania LINQ to Object. Każdy element tej kolekcji (o nazwie `Osoba`) składa się z czterech podelementów (`Imię`, `Nazwisko`, `NumerTelefonu` i `Wiek`) odpowiadających polom tabeli. Identyfikator rekordu zapisałem w atrybutie o nazwie `Id` elementów `Osoba`.

Zapytania LINQ, czyli tworzenie kolekcji na bazie danych z pliku XML

Dysponując plikiem XML zawierającym tabelę, możemy pobierać z niej dane z wykorzystaniem zapytań LINQ⁴. Jeżeli nie uwzględnimy w zapytaniu filtrowania i sortowania elementów, zapytanie takie będzie najprostszym sposobem konwersji pliku XML na kolekcję. W listingu 12.8 zawartość pliku XML (posortowana i przefiltrowana) zapisywana jest w kolekcji złożonej z obiektów typu `Osoba` (listing 11.1). Korzystając z typów anonimowych, moglibyśmy również zapisać tylko niektóre pola odczytane z tabeli lub wręcz zwrócić kolekcję łańcuchów, np. zawierających jedynie personalia. Zależy to jedynie od tego, jakie dane i w jakiej postaci są nam potrzebne.

Listing 12.8. Konwersja pliku XML do kolekcji

```

private void button3_Click(object sender, EventArgs e)
{
    //pobieranie danych
    XDocument xml = XDocument.Load("ListaOsob.xml");
    IEnumerable<Osoba> listaOsobPelnoletnich =
        from osoba in xml.Descendants("Osoba")
        select
            new Osoba() {

```

⁴ Takie zapytania można oczywiście konstruować także, gdy dane nie są ułożone w serie rekordów, jak było w przypadku pliku zawierającego położenie i wielkość okna, ale wówczas wynikiem zapytania są zazwyczaj kolekcje jednoelementowe. Korzystanie z zapytania LINQ w takiej sytuacji to raczej przerost formy nad treścią. Na poniższym listingu pokazuję jednak zapytanie LINQ wyświetlające informacje o składowej `x` położenia okna uzyskane z pliku `Ustawienia.xml` za pomocą zapytania LINQ.

```

private void button2_Click(object sender, EventArgs e)
{
    XDocument xml = XDocument.Load("Ustawienia.xml");
    var cos = from element in xml.Descendants() where element.Name == "X" select element;
    int ile = cos.Count();
    MessageBox.Show(ile.ToString());
    string s = "Wartości elementów:\n";
    foreach ( XElement element in cos) s += element.Value;
    MessageBox.Show(s);
}

```

```

        Id = int.Parse(osoba.Attribute("Id").Value),
        Imię = osoba.Element("Imię").Value,
        Nazwisko = osoba.Element("Nazwisko").Value,
        NumerTelefonu = int.Parse(osoba.Element("NumerTelefonu").Value),
        Wiek = int.Parse(osoba.Element("Wiek").Value) };

    // wyświetlanie danych
    string s = "Lista osób pełnoletnich:\n";
    foreach (Osoba osoba in listaOsobPelnoletnich) s += osoba.Imię + " " + osoba.Nazwisko
                                                + " (" + osoba.Wiek + ")\n";
    MessageBox.Show(s);
}

```

Na listingu 12.8 pokazuję, w jaki sposób utworzyć kolekcję łańcuchów zawierających imiona i nazwiska tych osób spośród zapisanych w pliku XML, które są pełnoletnie. W dodatku sortuję je zgodnie z alfabetyczną kolejnością imion. Uzupełnijmy zatem zapytanie LINQ o operatory where i orderby, a po operatorze select zbudujmy odpowiedni łańcuch. Prezentuję to na listingu 12.9.

Listing 12.9. Budowanie kolekcji łańcuchów na podstawie informacji z pliku XML

```

private void button4_Click(object sender, EventArgs e)
{
    // pobieranie danych
    XDocument xml = XDocument.Load("ListaOsob.xml");
    IEnumerable<string> listaOsobPelnoletnich =
        from osoba in xml.Descendants("Osoba")
        where int.Parse(osoba.Element("Wiek").Value) >= 18
        orderby osoba.Element("Imię").Value
        select osoba.Element("Imię").Value + osoba.Element("Nazwisko").Value;

    // wyświetlanie danych
    string s = "Lista osób pełnoletnich:\n";
    foreach (string personalia in listaOsobPelnoletnich) s += personalia + "\n";
    MessageBox.Show(s);
}

```

Modyfikacja pliku XML

Załóżmy, że chcemy zmodyfikować istniejący plik XML, a dokładniej zmienić wartość w tylko jednym z jego elementów. W takiej sytuacji możemy wczytać plik XML do obiektu XDocument, pobrać interesujący nas element (przechowując jego referencję, a nie wartość), zmodyfikować go i z powrotem zapisać do pliku XML. Wszystkie te czynności realizuję za pomocą metody z listingu 12.10, w której zwiększałem o jeden wiek wybranej osoby.

Listing 12.10. Modyfikacja wybranego elementu w pliku XML

```

private void button5_Click(object sender, EventArgs e)
{
    XDocument xml = XDocument.Load("ListaOsob.xml");
    IEnumerable< XElement > listaOsob = xml.Descendants("Osoba").Where(osoba =>
        (osoba.Element("Imię").Value == "Jan" && osoba.Element("Nazwisko").Value ==
        "Kowalski"));

```

```
if (listaOsob.Count() > 0) listaOsob.First().Element("Wiek").Value =  
    ↳(int.Parse(listaOsob.First().Element("Wiek").Value) + 1).ToString();  
else  
{  
    MessageBox.Show("Brak osób o podanym imieniu i nazwisku");  
    return;  
}  
xml.Save("ListaOsob.xml");  
}
```

Modyfikację nazwy elementu można przeprowadzić w analogiczny sposób — jedyna zmiana w powyższym kodzie to zamiana właściwości Value na Name, np. listaOsob.
First().Element("Wiek").Name. Dla zmiany nazwy elementu nie widzę jednak praktycznego zastosowania. Może je mieć natomiast zmiana wartości atrybutu. Dostęp do niego możemy uzyskać dzięki wyrażeniu listaOsob.First().Attribute("Id").Value.

Rozdział 13.

Baza danych SQL Server w projekcie Visual Studio

Odwzorowanie obiektowo-relacyjne

Programista zamierzający korzystać w swoim projekcie z baz danych może wybrać jeden z trzech oferowanych przez Microsoft mechanizmów odwzorowania zawartości baz danych w klasach .NET lub jeden z wielu framework'ów niezależnych firm, spośród których warto wspomnieć przede wszystkim darmowy nHibernate. W dalszych rozdziałach opiszę jednak tylko te techniki *odwzorowania obiektowo-relacyjnego* (ang. *object-relational mapping*, w skrócie ORM), które zostały przygotowane przez Microsoft i dostępne są w platformie .NET bez konieczności dodatkowej instalacji. Czym są w ogóle technologie ORM? Mówiąc najprościej, jest to mechanizm, który umożliwia aplikacji dostęp do danych z tabel i widoków bazy danych bez konieczności nawiązywania niskopoziomowego „kontaktu” z bazą danych i wysyłania do niej zapytań SQL. ORM udostępnia dane w kolekcjach platformy .NET, z których mogą być nie tylko odczytywane, ale również w nich modyfikowane. Jest to więc pomost między światem aplikacji a światem bazy danych, warstwa abstrakcji zwalniająca programistę z obowiązku znajomości szczegółów dotyczących konkretnego typu używanej przez niego bazy danych.

Jako pierwszą wymienię tradycyjną, ale nadal dostępną technikę ADO.NET opartą na kontrolce DataSet, która pozwala na tworzenie aplikacji bazodanowych przy minimalnej liczbie samodzielnie napisanych linii kodu. Pozwala na łączenie z bazami danych SQL Server, Microsoft Access poprzez mechanizm OLE DB, z obiektowymi bazami danych XML, a także bazami dostępnymi poprzez pomosty ODBC. Przez wiele kolejnych wersji platformy .NET była to jedyna oficjalna technologia Microsoftu, w wyniku czego nadal jest wiele projektów, które z tej technologii korzystają. Z tego powodu poświęciłem tej technologii rozdział 16. Za wadę tej technologii można uznać jej niską abstrakcyjność, choć w praktyce nie zawsze to rzeczywiście jest wada. Kontrolka DataSet jest w zasadzie bezpośrednią reprezentacją tabel bazy danych, a dodatkowo jest

zarządcą poleceń SQL wysyłanych do bazy danych. Zajmuje się także buforowaniem odbieranych z niej danych. W efekcie programista przymuszany jest do myślenia w kategoriach baz danych i języka SQL.

Drugim rozwiązaniem jest LINQ to SQL i jego klasa `DataContext`. To idealne rozwiązanie dla programistów: proste, lekkie i eleganckie. Moje ulubione. Niestety ograniczone wyłącznie do baz danych SQL Server. Nie jest już wprawdzie rozwijane, ale jest na tyle dojrzałe, że nie widzę niczego złego w korzystaniu z niego także w nowych projektach. Rozwiązanie to opisuję dość szeroko w rozdziałach 14. i 15. Zarzucenie tego projektu było zresztą bardziej decyzją „polityczną” niż merytoryczną: projekt ten był rozwijany przez zespół programistów pracujących pod kierunkiem Andersa Hejlsberga nad rozwojem języka C# i LINQ (w czasie przygotowywania platformy .NET 3.5). Był sztandarową technologią z rodziny LINQ, dowodzącą jej skuteczności. Po udostępnieniu platformy .NET w wersji 3.5 projekt LINQ to SQL przeszedł pod skrzydła zespołu zajmującego się bazami danych. Tam rozwijany był już jednak inny projekt — Entity Framework. I to on „wygrał”, stając się sztandarową techniką odwzorowywania obiektowo-relacyjnego Microsoftu.

Wspomniane Entity Framework (EF) jest trzecim, obecnie najbardziej promowanym mechanizmem ORM dostępnym w platformie .NET. Od wersji 6.0 dołączanej do Visual Studio 2013 jego rozwój został „uwolniony” i zajmuje się nim teraz grupa programistów *open source* (podobnie jak np. ASP.NET MVC). Źródła EF można znaleźć na stronie *CodePlex* (<http://entityframework.codeplex.com/>). Warto zwrócić uwagę, że choć z EF można korzystać podobnie jak z dwóch pozostałych narzędzi ORM, a więc automatycznie tworzyć klasy-modele odwzorowujące strukturę istniejącej bazy danych, to możliwy jest również scenariusz odwrotny, w którym najpierw projektujemy model, na podstawie którego utworzona zostanie baza danych. To stwarza nowe możliwości. Martwi mnie jednak mniejsza niż w przypadku LINQ to SQL wydajność tego framework'u. Nie potrafię wprawdzie tej oceny podeprzeć uczciwymi badaniami, jednak np. inicjacja aplikacji, w której ładowane są dane *via* EF, wydaje się wyraźnie dłuższa.

Warto dodać, że w przypadku wszystkich trzech technologii po utworzeniu obiektowej reprezentacji zawartości bazy danych możliwe jest utworzenie tzw. źródeł danych (podokno *Data Sources* w Visual Studio). Nie jest to kolejny zbiór klas, a jedynie pewna „lekka” warstwa abstrakcji pozwalająca m.in. na wygodne projektowanie interfejsu aplikacji. Dysponując źródłami danych, możemy za pomocą myszy utworzyć siatkę prezentującą zawartość tabeli lub widoku, formularze lub w łatwy sposób wykorzystać relację między tabelami do utworzenia układu siatek typu *master-details*. Te wszystkie czynności — opisuję je w rozdziale 15. na przykładzie technologii LINQ to SQL — w niewielkim stopniu zależą od tego, z jakiej technologii odwzorowania obiektowo-relacyjnego korzystamy. Możliwość stosowania źródeł danych nie ogranicza się zresztą tylko do tabel w bazach danych. Mogą one także udostępniać dane pobierane z usług sieciowych, kolekcji czy innych obiektów dostępnych w aplikacji.

Szalenie krótki wstęp do SQL

Zgodnie z zapowiedzią: programując aplikacje korzystające z baz danych, z założenia będziemy starali się nie zniżać do poziomu, na którym konieczne będzie samodzielne redagowanie poleceń w języku SQL. Jak obiecałem wyżej, w Visual Studio w przypadku bazy danych SQL Server możemy uniknąć budowania poleceń języka SQL zarówno podczas tworzenia samej bazy danych, jak i przy dodawaniu do niej tabel i zapelnianiu ich danymi. Wszystkie te czynności można wykonać przy użyciu myszy, choć pokazywane będą polecenia SQL i będziemy mieli możliwość ich edycji. Natomiast dzięki mechanizmom ORM będziemy uwolnieni od SQL-a także przy pobieraniu i edycji danych z poziomu aplikacji. Od SQL-a nie uciekniemy jednak, jeżeli zechcemy zdefiniować procedurę składowaną lub widok. Dlatego chciałbym przybliżyć Czytelnikowi chociażby podstawy tego języka w minimalnym potrzebnym nam zakresie.

SQL to język pozwalający na dostęp do danych w bazach danych i ich modyfikację. Transact SQL (T-SQL) to microsoftowy dialekt tego języka, stosowany w SQL Server i Access. Liczba nowości, o jakie rozszerzony został T-SQL w najnowszych wersjach SQL Servera, świadczy o jego bardzo dynamicznym rozwoju. My jednak pozostaniemy przy trzech najbardziej podstawowych instrukcjach tego języka. Powinniśmy koniecznie poznać polecenie SELECT pozwalające na pobieranie danych z tabeli, polecenie INSERT pozwalające na dodanie rekordu do tabeli oraz polecenie DELETE, które umożliwia jego usunięcie.

Select

Instrukcja SELECT pozwala na tworzenie zapytań pobierających dane z bazy danych. Podobnie jak w przypadku pozostałych poleceń SQL składnia tej instrukcji zbliżona jest do składni języka naturalnego, np. wybierz [kolumny] z [tabeli], co tłumaczy się na:

```
SELECT [kolumna1],[kolumna2] FROM [tabela]
```

Nawiąsy klamrowe otaczają nazwy kolumn i tabel, co pozwala na swobodne używanie spacji. W przykładzie opisywanym niżej w tym rozdziale tabela ma nazwę Osoby, a dwie z jej kolumn — Imię i Nazwisko, zatem zapytanie może mieć postać:

```
SELECT [Imię],[Nazwisko] FROM [Osoby]
```

Jeżeli chcemy pobrać wszystkie kolumny tabeli, zamiast je wymieniać, możemy użyć gwiazdki:

```
SELECT * FROM [Osoby]
```

Powyższą instrukcję można skomplikować, dodając warunek filtrujący zawartość pobieranych danych. Założmy, że tabela ma jeszcze jedną kolumnę. Jest w niej zapisywany wiek osoby. Pobierzmy z tabeli jedynie te personalia, które należą do osób pełnoletnich:

```
SELECT [Imię],[Nazwisko] FROM [Osoby] WHERE Wiek>=18
```

Dane mogą być nie tylko filtrowane, ale również sortowane. W tym celu do zapytania należy dodać klauzulę ORDER BY z następującą po niej nazwą kolumny, według której dane mają być uporządkowane, np.

```
SELECT [Imię],[Nazwisko] FROM [Osoby] WHERE Wiek>=18 ORDER BY [Nazwisko]
```

Insert

Drugie z podstawowych poleceń języka SQL to INSERT. Pozwala na dodanie do tabeli nowego rekordu z danymi. Oto jego podstawowa składnia w T-SQL:

```
INSERT INTO tabela ([kolumna1],[kolumna2]) VALUES (wartość1,wartość2)
```

Odpowiada to prostej składni języka naturalnego: wstaw do tabeli, we wskazane kolumny, podane wartości. W prostym przykładzie, w którym chcemy zapełnić tylko dwa pola tabeli, instrukcja INSERT może wyglądać następująco:

```
INSERT INTO Osoby ([Imię],[Nazwisko]) VALUES ('Jacek','Matulewski')
```

To jednak zadziała tylko w sytuacji, w której pozostałe pola mogą pozostać puste. W przeciwnym wypadku dodanie rekordu nie powiedzie się.

Delete

Kolejne polecenie usuwa wiersz z bazy danych. Jego podstawowa konstrukcja też nie jest specjalnie zawiła. Usuń z tabeli ListaAdresow wszystkie te wiersze, które w kolumnie Email zawierają podany adres — takie polecenie w języku SQL należy zapisać jako:

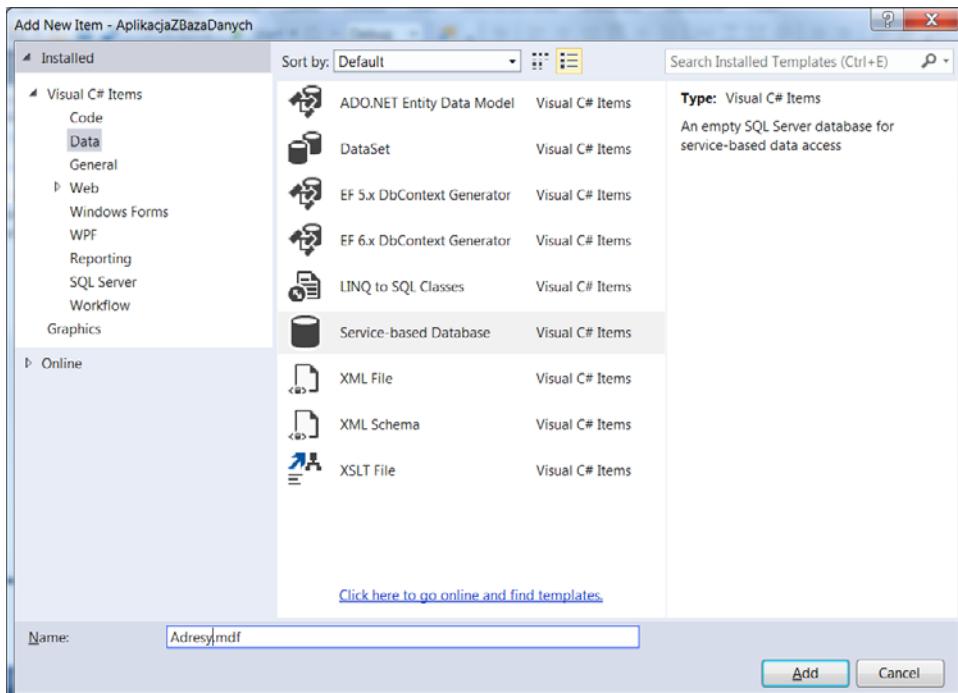
```
DELETE FROM ListaAdresow WHERE Email='jacek@phys.uni.torun.pl'
```

Projekt aplikacji z bazą danych

W dalszej części rozdziału opiszę, w jaki sposób z poziomu Visual Studio można utworzyć i dodać do projektu aplikacji instancję bazy danych SQL Server. SQL Server jest wyróżniony. Projekty .NET mogą wprawdzie korzystać z wielu rodzajów baz danych, ale tylko w przypadku SQL Servera Visual Studio pozwala na tworzenie bazy danych od zera bez konieczności używania dodatkowych narzędzi.

Dodawanie bazy danych do projektu aplikacji

1. Pierwszym krokiem niech będzie utworzenie nowego projektu aplikacji Windows Forms o nazwie *AplikacjaZBazaDanych*.
2. Następnie dodajmy do projektu bazę danych SQL Server. W tym celu z menu *Project* wybieramy pozycję *Add New Item....*
3. W otwartym w ten sposób oknie (rysunek 13.1) zaznaczamy ikonę *Service-based Database*, a w polu *Name* wpisujemy nazwę *Adresy.mdf*. Klikamy przycisk *Add*.



Rysunek 13.1. Tworzenie bazy danych SQL Server w Visual Studio

4. Do plików wymienionych w *Solution Explorer* dodana zostanie pozycja *Adresy.mdf* wraz z towarzyszącym jej plikiem *Adresy.ldf*.

Łańcuch połączenia (ang. connection string)

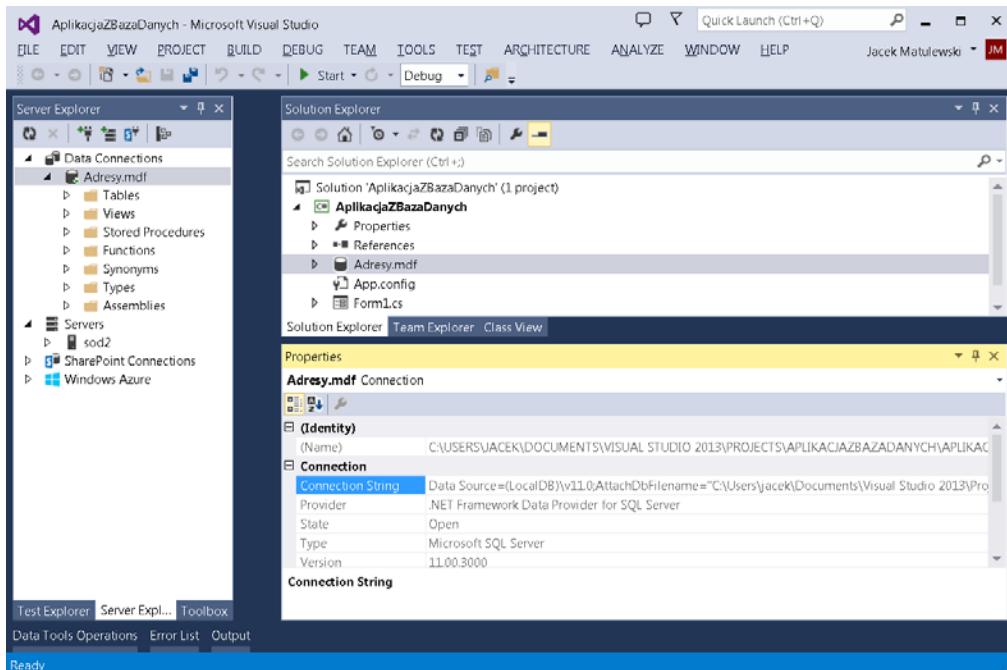
Kliknijmy dwukrotnie pozycję *Adresy.mdf* w *Solution Explorer*. W miejscu, gdzie zwykle jest podokno *Toolbox*, pojawi się *Server Explorer* — wygodne narzędzie, które umożliwia także przeglądanie i edycję baz danych SQL Server. Pierwsze otwarcie bazy danych w tym podoknie wiąże się z uruchomieniem serwera bazy danych i może potrwać dłuższą chwilę, czasem dłuższą niż przewidziany na to czas. Zatem jeżeli się nie powiedzie, warto spróbować jeszcze raz. Po rozwinięciu gałęzi *Adresy.mdf* (rysunek 13.2) zobaczymy podgałęzie odpowiadające tabelom, widokom, procedurom składowanym itd. Wszystkie są na razie puste.



Zawartość bazy danych SQL Server można też przeglądać w podoknie *SQL Server Object Explorer* (dostępne z menu kontekstowego połączonej bazy danych w podoknie *Server Explorer*).

Zwróćmy uwagę na pewien szczegół. Jeżeli w podoknie *Server Explorer* zaznaczymy gałąź *Adresy.mdf*, w podoknie *Properties* pojawią się szczegóły, m.in. dotyczące połączenia z tą bazą danych. Ważna dla nas własność to *Connection String*, czyli łańcuch połączenia, który określa szczegóły połączenia z bazą danych. Po drobnej modyfikacji będziemy go używali w kolejnych rozdziałach. W tej chwili ma on następującą postać:

```
Data Source=(LocalDB)\v11.0;AttachDbFilename="C:\Users\jacek\Documents\Visual Studio
➥2013\Projects\AplikacjaZBazaDanych\AplikacjaZBazaDanych\Adresy.mdf";Integrated
➥Security=True
```



Rysunek 13.2. Użycie okna *Server Explorer* do przeglądania i edycji bazy danych

Jak widać, łańcuch ten zawiera bezwzględną ścieżkę do pliku *Adresy.mdf*. Aby umożliwić swobodne przenoszenie projektu, warto będzie tę ścieżkę zrelatywizować względem katalogu projektu, tj.:

```
Data Source=(LocalDB)\v11.0;AttachDbFilename="|DataDirectory|
➥\Adresy.mdf";Integrated Security=True
```

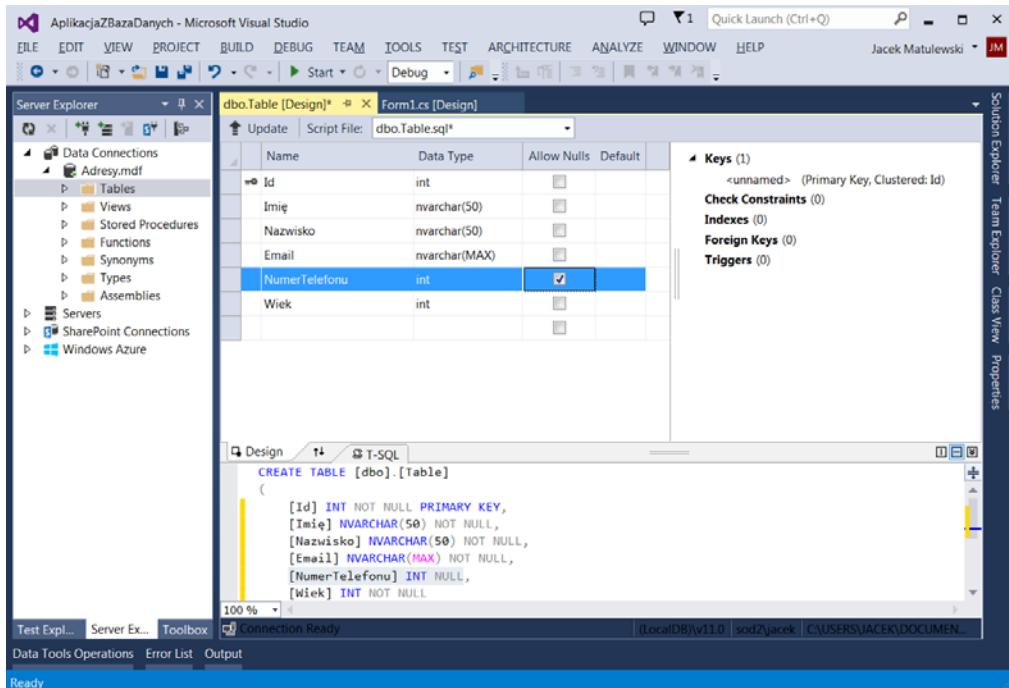
Poza tym w podoknie *Properties* możemy sprawdzić informacje o użytym sterowniku (*.NET Framework Data Provider for SQL Server*) i aktualnym stanie połączenia. Ten ostatni może być sygnalizowany lapidarnym *Closed* — rozłączone. Wystarczy jednak rozwiniąć gałąź w *Server Explorer*, aby stan zmienił się na *Open*. Stan jest też sygnalizowany małą ikoną przy pliku bazy danych w *Server Explorerze*. Po otwarciu połączenia w podoknie *Properties* możemy zobaczyć także typ i wersję mechanizmu odpowiedzialnego za dostęp do bazy danych. W przypadku SQL Servera powinno to być *Microsoft SQL Server*, a w przypadku bazy danych Access — *MS Jet*.

Dodawanie tabeli do bazy danych

Aby do bazy danych dodać tabelę:

1. W podoknie *Server Explorer* zaznaczmy pozycję *Tables* i z jej menu kontekstowego wybierzmy *Add New Table*. W głównej części okna pojawi

się edytor pozwalający na zdefiniowanie kolumn (lub jak kto woli, pól) nowej tabeli. Proponuję zdefiniować tabelę, ustalając nazwy pól i ich typy zgodnie ze wzorem przedstawionym na rysunku 13.3.



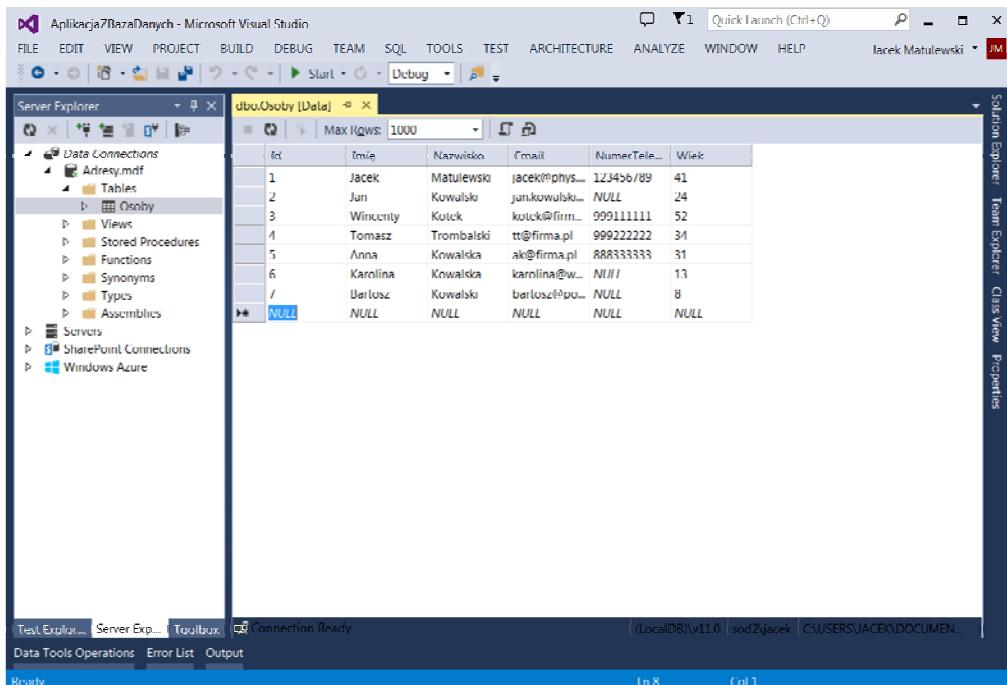
Rysunek 13.3. Propozycja tabeli, na której będziemy ćwiczyć korzystanie z ADO.NET

2. Jedno z pól powinno być kluczem głównym. Domyslnie jest nim automatycznie dodane do tabeli pole *Id* (vide ikona z kluczem przy tym polu). Proponuję tak pozostawić. Rolę unikatowego klucza mogłyby również pełnić pole *Email* zawierające adres e-mail.
3. Jedynie przy polu *NumerTelefonu* pozostawiamy zaznaczoną opcję *Allow Nulls*. Wszystkie pozostałe będą musiały posiadać wartości.
4. Pod edytorem tabeli widoczny jest kod T-SQL, który powstaje w konsekwencji definiowania kolejnych kolumn. Aby utworzyć tabelę, musimy go uruchomić. Służy do tego przycisk *Update* słabo widoczny nad edytorem tabeli. Wcześniej zmieńmy jednak nazwę tabeli w kodzie T-SQL z *Table* na *Osoby*.
5. Teraz kliknijmy przycisk *Update*.
6. Pojawi się okno *Preview Database Updates* z podglądem zmian wprowadzanych do bazy danych. Zobaczymy w nim, że jedyną zmianą będzie utworzenie nowej tabeli. Aby potwierdzić, kliknijmy *Update Database*.
7. W podoknie *Database Tools Operations* zobaczymy potwierdzenie zmian wykonanych w bazie danych. Natomiast w podoknie *Server Explorer* w gałęzi *Tables* zobaczymy tabelę *Osoby*.

Edycja danych w tabeli

Następnym krokiem może być wypełnienie tabeli przykładowymi danymi:

1. W *Server Explorer* rozwińmy gałąź *Tables* i zaznaczmy pozycję *Osoby*.
2. Z jej menu kontekstowego wybierzmy polecenie *Show Table Data*.
3. Zobaczmy siatkę pozwalającą na edycję danych. Umieścmy w tabeli kilka przykładowych rekordów (rysunek 13.4).

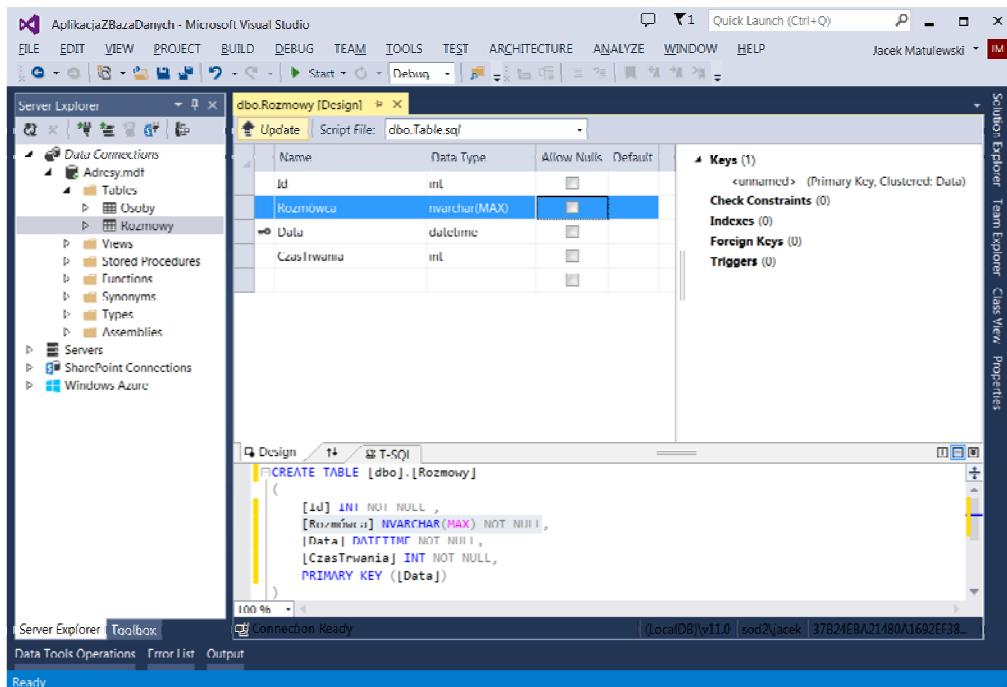


Rysunek 13.4. Edycja bazy danych za pomocą narzędzi Visual Studio

Druga tabela

Postępując podobnie, tj. korzystając z narzędzi dostępnych w podoknie *Server Explorer*, dodajmy do bazy danych *Adresy.mdf* jeszcze jedną tabelę, która będzie zawierać identyfikator osoby, łańcuch opisujący jej rozmówce, datę przeprowadzenia rozmowy i czas jej trwania w minutach (rysunek 13.5). Pola o nazwie Id i CzasTrwania będą typu int, pole Rozmowca — typu nvarchar(MAX), a pole Data — typu datetime. Żadne z pól nie będzie dopuszczać pustych wartości. Kluczem głównym może zostać tylko pole Data (obecność klucza głównego stanie się ważna w momencie tworzenia relacji między tą a innymi tabelami bazy danych). Następnie zapiszmy tabelę, nazywając ją *Rozmowy*. Pole Id w nowej tabeli będzie zawierać te same identyfikatory osoby co pole Id w tabeli *Osoby*. Nie jest to w pełni zgodne z konwencją, według której Id powinno być unikalnym identyfikatorem rozmowy, a jednocześnie kluczem głównym tej tabeli, a klucz obcy identyfikujący osobę powinien nazywać się np. *OsobaId*. Każdej osobie będzie

można przyporządkować listę rozmów, a każdej rozmowie — tylko jednego rozmówcę. Tabela Osoby będzie więc naszą tabelą-rodzicem, a tabela Rozmowy — tabelą zawierającą szczegóły, czyli tabelą-dzieckiem. Nie będziemy jednak definiować relacji między tymi tabelami na poziomie bazy danych. Da nam to pretekst do tworzenia tych relacji już na poziomie klas ORM. Tabelę należy oczywiście wypełnić jakimiś przykładowymi danymi, choć takimi jak na rysunku 13.6.



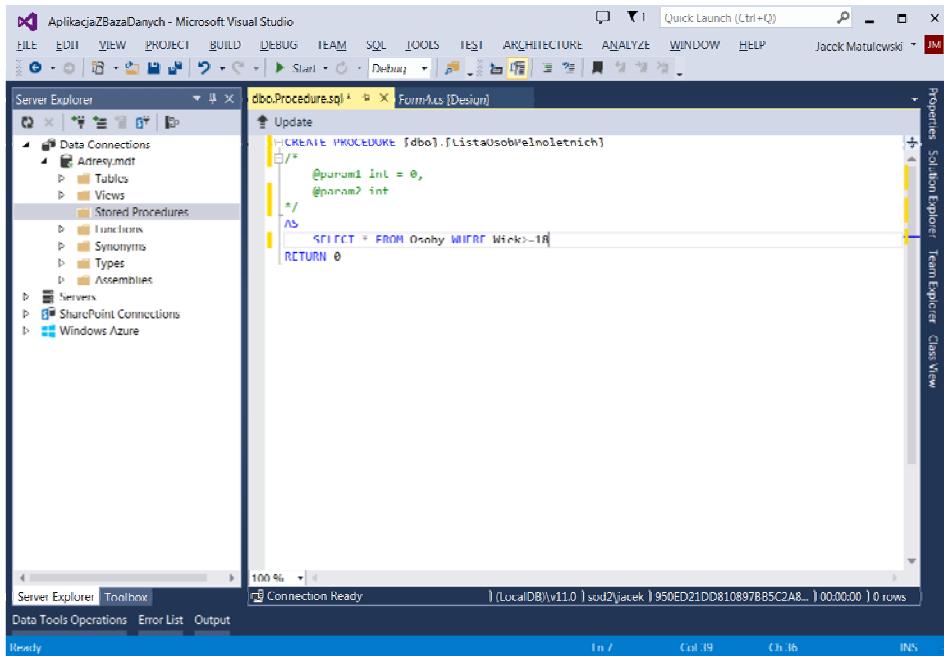
Rysunek 13.5. Projekt tabeli

dbo.Rozmowy (Data)			
	Rozmówca	Data	CzasTrwania
1	KM	2013-11-20 14:16:00	10
3	GM	2013-11-20 17:10:24	13
3	GM	2013-11-21 10:01:20	177
3	GM	2013-11-21 12:10:56	11
1	AD	2013-11-21 12:11:03	56
2	GM	2013-11-22 21:00:04	734
	NUL	NUL	NUL

Rysunek 13.6. Przykładowe dane w tabeli Rozmowy

Procedura składowana — pobieranie danych

Dodatkowo umieścmy w bazie danych trzy procedury składowane. Pierwsza będzie zapytaniem służącym do pobierania danych, druga posłuży do ich modyfikacji, a trzecia do tworzenia nowej tabeli. Zaczniemy od zapytania SQL pobierającego personalia osób pełnoletnich zapisanych w tabeli Osoby. W tym celu w podoknie *Server Explorer*, w gałęzi *Data Connections*, *Adresy.mdf* przejdźmy do pozycji *Stored Procedures* i prawym przyciskiem myszy rozwińmy jej menu kontekstowe. Wybierzmy z niego polecenie *Add New Stored Procedure*. Pojawi się edytor, w którym możemy wpisać kod SQL (rysunek 13.7). Za słowem kluczowym AS wpiszmy proste zapytanie SQL: `SELECT * FROM Osoby WHERE Wiek>=18`. Zmieńmy też nazwę procedury na `[dbo].[ListaOsobPelnoletnich]` (poszczególne fragmenty w nawiasach kwadratowych). Usuńmy też argumenty procedury. Wreszcie zapiszmy zmiany w bazie danych, klikając przycisk *Update*. W podoknie *Server Explorer*, w gałęzi *Stored Procedures* bazy *Adresy.mdf*, powinna pojawić się pozycja *ListaOsobPelnoletnich* (aby tak się stało, konieczne będzie kliknięcie ikony *Refresh*). Procedurę tę można uruchomić z poziomu podokna *Server Explorer*. Wystarczy z jej menu kontekstowego wybrać polecenie *Execute*. Pojawi się skrypt SQL (na jego pasku narzędzi jest ikona pozwalająca na ponowne uruchomienie), a pod nim dane będące wynikiem zapytania.



Rysunek 13.7. Edycja zapytania SQL procedury umieszczonej w bazie danych

Procedura składowana — modyfikowanie danych

Przygotujmy także procedurę składowaną zwiększającą wiek wszystkich osób niebędących pełnoletnimi kobietami (listing 13.1). Zapiszmy ją w bazie pod nazwą *AktualizujWiek* (przycisk *Update*) i dodajmy do prawego panelu na zakładce *Adresy.dbml*.

Listing 13.1. Procedura składowana modyfikująca dane w tabeli Osoby

```
CREATE PROCEDURE [dbo].[AktualizujWiek]
AS
    UPDATE Osoby SET Wiek=Wiek+1 WHERE NOT SUBSTRING(Imię,LEN(Imię),1)='a' OR Wiek<18
    RETURN 0
```

W zapytaniu użyliśmy procedury SUBSTRING zwracającej fragment łańcucha i negacji NOT. Sprawdźmy jej działanie polecienniem *Execute* z menu kontekstowego. Pojawi się wówczas skrypt SQL. Zmiany zostaną wprowadzone do bazy danych z katalogu projektu (nie tej skopiowanej do podkatalogu *bin\Debug* lub *bin\Release*).

Procedura składowana — dowolne polecenia SQL

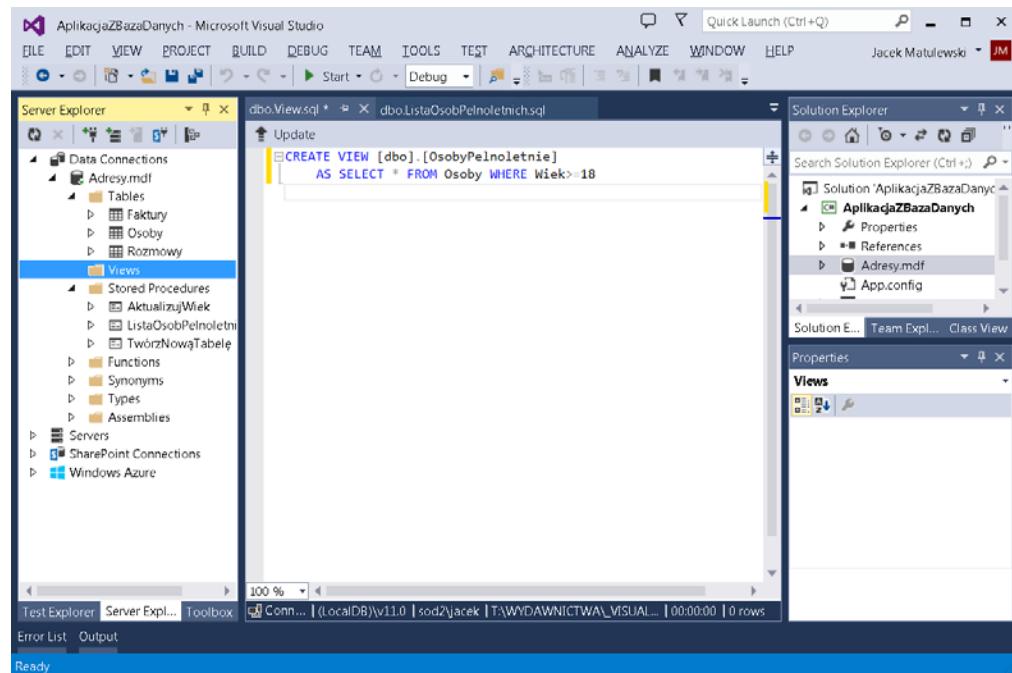
Pobieranie danych i ich modyfikacja za pomocą składowanych poleceń SQL nie jest może tym, co programiści, szczególnie znający LINQ, docenią najbardziej. Warto jednak zwrócić uwagę, że istnieje grupa czynności, które czasem najprościej wykonać bezpośrednio w SQL. Dla przykładu: jeżeli w bazie danych tworzymy zbiór tabel dla każdego nowo zarejestrowanego użytkownika, to nie do przecenienia jest możliwość przygotowania składowanej procedury zawierającej kod SQL tworzący tabelę i wypełniający ją danymi. W tym celu wystarczy przygotować nową procedurę składowaną, choćby taką jak na listingu 13.2. W ogólności nazwę tabeli i nazwy pól możemy przekazać przez parametry procedury. Pamiętajmy jednak, że utworzona w taki sposób tabela nie będzie odwzorowana w żadnym mechanizmie mapowania obiektowo-relacyjnego (ORM), co utrudnia korzystanie z niej z poziomu kodu C#.

Listing 13.2. Przykładowa procedura składowana tworząca tabelę

```
ALTER PROCEDURE dbo.TwórzNowąTabelę
AS
    CREATE TABLE "Faktury" (Id INT, NrFaktury INT, Opis NVARCHAR(MAX))
    INSERT INTO "Faktury" VALUES(1, 1022, 'Faktura za styczeń 2008')
    INSERT INTO "Faktury" VALUES(1, 1023, 'Faktura za luty 2008')
    RETURN
```

Widok

Zamiast procedury składowanej będącej zapytaniem SQL do pobierania danych lepiej użyć widoku. Z punktu widzenia programisty widok zachowuje się tak jak tabela, ale tak naprawdę dostępne w nim dane są generowane dynamicznie. Można w ten sposób ograniczyć dostępność danych lub przygotować pseudotabelę łączącą dane z kilku tabel. Na rysunku 13.8 widoczne jest polecenie SQL tworzące widok udostępniający listę osób pełnoletnich z tabeli Osoby (aby otworzyć edytor, klikamy *Add New View* na gałęzi *Views*). Po jego przygotowaniu musimy jak zwykle nacisnąć przycisk *Update*, a w podoknie *Server Explorer*, w gałęzi *Views*, po kliknięciu ikony *Refresh* pojawi się pozycja *OsobyPełnoletnie*. Z jej menu kontekstowego możemy wybrać polecenie *Show Results*, aby zweryfikować działanie kryjącego się pod widokiem zapytania. We wszystkich rekordach pole *Wiek* powinno być większe lub równe 18.



Rysunek 13.8. Tworzenie widoku udostępniającego podzbior rekordów tabeli Osoby

* * *

Jak widać, Visual Studio wyposażone jest w narzędzia pozwalające na edycję tabel bazy SQL Server, a nawet zawartych w nich danych. Wsparcie dla baz danych Access jest mniejsze, tzn. nie jest możliwe tworzenie baz tego typu ani ich edycja. Trudno się jednak temu dziwić. Aplikacja Access, stanowiąca część pakietu Microsoft Office, w przeciwieństwie do SQL Server i SQL Server Compact nie jest programem darmowym. Nie oznacza to jednak, że bazy danych Access nie możemy użyć w aplikacji. Wręcz przeciwnie.



Zapiszmy projekt w obecnym stanie i zróbcmy kopię całego jego katalogu. To ważne, bo projekt ów będzie punktem wyjścia dalszych ćwiczeń w kolejnych rozdziałach.

Rozdział 14.

LINQ to SQL

Wspominając o technologii LINQ, mówi się zazwyczaj o zanurzeniu języka SQL w języku C#. W LINQ to SQL zanurzenie to można rozumieć niemal dosłownie — zapytanie LINQ jest w tym przypadku tłumaczone na zapytanie w języku T-SQL, które jest wysyłane do bazy danych SQL Server. Jest to element bardzo przemyślnego mechanizmu ORM, który chciałbym przedstawić jako pierwszy. Należy podkreślić, że technologia LINQ to SQL współpracuje wyłącznie z bazami danych SQL Server (nie obsługuje nawet lokalnych baz SQL Server Compact, a jedynie jej „pełne” wersje). Była swojego rodzajem eksperymentem, dowodem na to, że technologia LINQ może być użyteczna w praktyce. I choć projekt rozwoju LINQ to SQL został zakończony, jest dostępny wśród rozwiązań oferowanych przez Visual Studio i ze względu na swoją wygodę jest nadal często używany w nowych projektach.

Podstawowym elementem LINQ to SQL jest klasa `DataContext`, której towarzyszy tzw. klasa encji, tj. klasa C#, która opisuje zawartość rekordu (encji) z tabeli bazy danych. To ta klasa jest kluczowa dla odwzorowania obiektowo-relacyjnego (ORM) realizowanego przez LINQ to SQL. Dzięki tej klasie możliwe jest zbudowanie pomostu między tabelą odczytywaną z bazy danych a tworzoną na podstawie jej danych kolekcją zwracaną przez zapytanie LINQ. Dzięki klasie encji możemy te dwie rzeczy w pewnym stopniu utożsamiać. Obiekt reprezentujący tabelę może być źródłem w zapytaniu LINQ. Visual Studio udostępnia wygodne narzędzia do tworzenia klasy encji, z których należy korzystać. Do tego też dojdziemy, ale wpierw proponuję stworzyć taką klasę samodzielnie — przynajmniej raz warto to zrobić, aby w pełni zrozumieć, w jaki sposób zrealizowane jest odwzorowanie pól tabeli w pola klasy encji.

Zanim przejdziemy do konkretów, chciałbym jeszcze zwrócić uwagę Czytelnika na jeden istotny fakt. W LINQ to Object, które poznaliśmy w rozdziale 11., źródłem danych była kolekcja istniejąca w pamięci i w pełni dostępna z poziomu programu. Nie było zatem konieczności odwoływanego się do zasobów zewnętrznych (np. plików bazy danych). W związku z tym cały kod aplikacji mógł być skompilowany do kodu pośredniego. W LINQ to SQL, którym zajmiemy się w tym rozdziale, lub LINQ to XML, omówionym w rozdziale 12., taka pełna komplikacja nie jest możliwa. Analiza danych pobranych ze źródła danych może być przeprowadzona dopiero w trakcie działania programu, do czego używane są tzw. drzewa zapytań¹.

¹ Więcej na ten temat w książce *Visual Studio 2010 dla programistów C#*, Helion, 2010.

Klasa encji

Prezentację LINQ to SQL zaczniemy od projektu z poprzedniego rozdziału, w którym do aplikacji Windows Forms dołączylismy bazę danych SQL Server *Adresy.mdf*. Naszym pierwszym celem będzie pobranie danych z jej tabeli *Osoby*, zatem zaczniemy od zdefiniowania klasy encji opisującej rekordy właśnie tej tabeli.

Warto powtórzyć, że *klasa encji* (ang. *entity class*) to zwykła klasa C#, w której pola klasy powiązane są za pomocą atrybutów z polami tabeli (kolumnami). Mamy więc do czynienia z modelowaniem zawartości relacyjnej bazy danych w typowanych klasach języka, w którym przygotowujemy program. W tym kontekście używany jest angielski termin *strongly typed*, który podkreśla izomorficzność relacji klasy encji i struktury tabeli. Ów związek jest niezwykle istotny — przede wszystkim umożliwia kontrolę typów, która w pewnym sensie rozciąga się na połączenie z bazą danych. Dzięki tej relacji programista może w pewnym stopniu utożsamiać tę klasę z reprezentowaną przez nią tabelą. To pozwala również przygotowywać zapytania LINQ z wykorzystaniem nazw pól tabeli — ich przetłumaczenie na zapytania SQL jest wtedy szczególnie proste, a jednocześnie w pełni zachowana zostaje kontrola typów. Domyślne wiązanie realizowane jest na podstawie nazw obu pól. Możliwa jest jednak zmiana tego domyślnego sposobu wiązania oraz zmiana własności poszczególnych pól.



Wskazówka

Atrybuty, których użyjemy do „oznakowania” klasy encji, należą do przestrzeni nazw *System.Data.Linq.Mapping* i umieszczone są w osobnej bibliotece platformy .NET o nazwie *System.Data.Linq.dll*. W przypadku „ręcznego” przygotowywania klasy encji, od czego zaczniemy, bibliotekę tę należy samodzielnie dodać do projektu przy użyciu polecenia *Project/Add Reference....* Omówione niżej narzędzie wizualnego projektowania klas encji zrobiło to za nas. Pamiętajmy, aby po dodaniu referencji zadeklarować użycie jej przestrzeni nazw instrukcją `using System.Data.Linq; →Mapping;`.

Przed całą klasą encji powinien znaleźć się atrybut *Table*, w którym wskazujemy nazwę tabeli z bazy danych. Natomiast przed polami klasy odpowiadającymi kolumnom z tabeli należy umieścić atrybut *Column*, w którym możemy wskazać m.in. nazwę kolumny w tabeli (parametr *Name*), zaznaczyć, że kolumna ta jest kluczem głównym (*IsPrimaryKey*) lub że może przyjmować puste wartości (*CanBeNull*). Argumentu *Name* używam tylko przy polu *Id*, i to tylko dla przykładu. Nie jest on konieczny, jeżeli nazwy kolumn są takie same jak nazwy pól w klasie encji. W listingu 14.1 przytaczam przykład klasy encji, w której definiuję typ rekordu z tabeli *Osoby* z bazy danych *Adresy.mdf* znanej z wcześniejszych rozdziałów (zobacz też klasę encji z rozdziału 11., listing 11.1). Klasę też umieściłem w pliku *Form1.cs* w przestrzeni nazw *AplikacjaZ→BazaDanych*.

Listing 14.1. Klasa encji związana z tabelą *Osoby* z bazy *Adresy.mdf*

```
[Table(Name = "Osoby")]
public class Osoba
{
    [Column(Name = "Id", IsPrimaryKey = true, CanBeNull = false)]
    public int Id;
    [Column(CanBeNull = false)]
```

```
public string Imię;
[Column(CanBeNull = false)]
public string Nazwisko;
[Column(CanBeNull = true)]
public string Email;
[Column(CanBeNull = true)]
public int? NumerTelefonu;
[Column(CanBeNull = false)]
public int Wiek;
}
```

Tworząc tabelę w bazie SQL Server, zezwoliliśmy, aby jej pola NumerTelefonu dopuszczały pustą wartość. Definiując klasę encji, należy zadeklarować odpowiadające im pola klasy w taki sposób, aby dopuszczały przypisanie wartości null. W przypadku łańcuchów nie ma problemu — typ String jest typem referencyjnym i zawsze można mu przypisać wartość null. Inaczej wygląda to np. w przypadku typu int, który jest typem wartościowym. Należy wówczas w deklaracji pola wykorzystać typ parametryczny Nullable<int>, który równoważnie może być zapisany jako int? (zobacz rozdział 3., podrozdział „Nullable”). Tak właśnie zrobiliśmy w przypadku pola NumerTelefonu.

W powyższym listingu wiązanie klasy z tabelą bazy danych przeprowadzane zostaje na podstawie atrybutów dołączanych do definicji klasy. W terminologii Microsoft nazywane jest to *mapowaniem opartym na atrybutach* (ang. *attribute-based mapping*). Możliwe jest również podejście alternatywne, w którym mapowanie odbywa się na podstawie struktury zapisanej w pliku XML. Takie podejście nosi nazwę mapowania zewnętrznego i nie będę się nim tu zajmował. Po jego opis warto zajrzeć na stronę MSDN: <http://msdn2.microsoft.com/en-us/library/bb386907.aspx>.

Pobieranie danych

Jak wspomniałem wcześniej, klasa encji zwykle nie jest tworzona ręcznie. Zazwyczaj do jej projektowania wykorzystywany jest edytor O/R Designer, który omówię w dalszej części tego rozdziału. Wydaje mi się jednak, że przy pierwszych próbach korzystania z technologii LINQ to SQL warto zrobić wszystko samodzielnie. Dotyczy to również powoływanego instancji klasy DataContext, choć korzystając z O/R Designera, uzyskalibyśmy klasę potomną względem DataContext, w której zdefiniowane byłyby m.in. właściwości odpowiadające tabelom i widokom oraz metody opakowujące procedury składowane. Wszystko mielibyśmy podane jak na tacy. Bez O/R Designera skazani jesteśmy na „czystą” klasę DataContext. A mimo to ilość kodu, jaki będziemy musieli przygotować, aby pobrać dane z tabeli, i tak będzie niewielka.

Zacznijmy od zdefiniowania pól przechowujących referencje do instancji klasy `DataContext` i jej tabeli `Osoby`:

```
const string connectionString = @"Data Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\Adresy.mdf;Integrated Security=True";
static DataContext bazaDanychAdresy = new DataContext(connectionString);
static Table<Osoba> listaOsob = bazaDanychAdresy.GetTable<Osoba>();
```

Tworzymy obiekt `DataContext` i pobieramy z niego referencję do tabeli, a więc jej reprezentacji w LINQ to SQL, czyli klasy `Table` parametryzowanej klasą encji `Osoba`. Klasa `DataContext` znajduje się w przestrzeni nazw `System.Data.Linq`, należy więc uwzględnić ją w grupie poleceń `using`². Całość wymaga tylko dwóch linii kodu. W pierwszej tworzymy sam obiekt `DataContext` reprezentujący bazę danych, a w drugiej tworzymy pole o nazwie `listaOsob` reprezentujące tabelę i umożliwiające dostęp do danych pobranych z tabeli `Osoby`. Pole to stanowi kluczową informację, będzie mogło być używane jako źródło danych w zapytaniach LINQ! Nazwa pobieranej tabeli wskazana została w atrybutie `Table`, którym poprzedziliśmy klasę encji. Dlatego jej nazwa nie pojawia się w powyższych instrukcjach.

Konstruktor klasy `DataContext` (pierwsza instrukcja) wymaga podania łańcucha połączenia konfigurującego połaczenie z bazą danych. Postać tego łańcucha ustaliliśmy w poprzednim rozdziale. Wcześniej, tj. do wersji 2010 środowiska Visual Studio, wystarczało podać bezwzględną ścieżkę do pliku bazy danych. Od wersji 2012, tj. od momentu zastąpienia baz danych SQL Server Express dołączanych do Visual Studio przez wersje LocalDB³, niezbędne jest wskazanie bazy danych za pomocą pełnego łańcucha połączenia.

Jak wspomniałem, obiekt typu `Table<Osoba>` może być źródłem w zapytaniach LINQ. Niczego więcej nam nie trzeba. Dzięki temu możemy swobodnie pobrać interesujące nas dane z tabeli. Prezentuję to na listingu 14.2, na którym widoczna jest metoda zdarzeniowa przycisku umieszczonego na formie.

Listing 14.2. Pobieranie danych z tabeli w bazie SQL Server za pomocą LINQ to SQL

```
private void button1_Click(object sender, EventArgs e)
{
    // pobieranie kolekcji
    var listaOsobPełnoletnich = from osoba in listaOsob
                                where osoba.Wiek >= 30
                                orderby osoba.Nazwisko
                                select osoba;

    // wyświetlanie pobranej kolekcji
    string s = "Lista osób pełnoletnich:\n";
    foreach (Osoba osoba in listaOsobPełnoletnich)
        s += osoba.Imię + " " + osoba.Nazwisko + " (" + osoba.Wiek + ")\n";
    MessageBox.Show(s);
}
```

Zapytanie LINQ (pierwsza instrukcja metody widocznej w listingu 14.2) pobiera z tabeli listę osób, które mają co najmniej 18 lat, i sortuje ich nazwiska alfabetycznie. Zwróciły uwagę na podobieństwo tego zapytania do tego, które stosowaliśmy w przypadku LINQ to Objects w rozdziale 11.

² Tu stosuje się również uwaga z poprzedniego podrozdziału na temat dodawania referencji do biblioteki `System.Data.Linq.dll`.

³ Zobacz <http://technet.microsoft.com/en-us/library/hh510202.aspx>.

W listingu 14.2 klasa encji Osoba pojawia się tylko raz w pętli foreach. Co więcej, skoro jest jasne, że każdy rekord jest obiektem typu Osoba, a to wiadomo dzięki parametryzacji pola listaOsob, jawne wskazanie typu w tej pętli nie jest wcale konieczne i z równym skutkiem moglibyśmy użyć słowa kluczowego var — kompilator sam wywnioskowałby, z jakim typem ma do czynienia.

Prezentacja danych w siatce DataGridView

Oczywiście pobrane w ten sposób dane można prezentować nie tylko w okienkach komunikatu. Wygodniej użyć do tego kontrolki. Dla przykładu umieścmy na formie siatkę, czyli kontrolkę DataGridView. Następnie w metodzie zdarzeniowej przycisku umieścmy kod:

```
var osobyPelnoletnie = from o in listaOsob
    where o.Wiek >= 18
    orderby o.Nazwisko
    select new { o.Id, o.Imię, o.Nazwisko, o.Email,
        ↳o.NumerTelefonu, o.Wiek };
dataGridView1.DataSource = osobyPelnoletnie;
```

W zapytaniu pobieramy rekordy odpowiadające osobom pełnoletnim posortowane alfabetycznie. Wynik zapytania, czyli kolekcję, wskazujemy następnie jako źródło danych kontrolki. Zwróćmy jednak uwagę, że wymusiłem skopiowanie danych, co uszykałem dzięki użyciu typu anonimowego w zwracanej kolekcji. Bez tego przy tak prostopnym podejściu nie uzyskalibyśmy pożdanego efektu. Łatwiejsze to będzie po utworzeniu klas za pomocą O/R Designer'a, a jeszcze łatwiejsze po dodaniu warstwy źródeł danych (kolejny rozdział).

Aktualizacja danych w bazie

Pobieranie danych, wizytówka LINQ, to zwykle pierwsza czynność wykonywana przez aplikacje. Jednak technologia LINQ to SQL nie poprzestaje tylko na tym. Zmiany wprowadzone w pobranej zapytaniem LINQ kolekcji można w łatwy sposób przesłać z powrotem do bazy danych. Służy do tego metoda SubmitChanges obiektu DataContext. Tym samym wszelkie modyfikacje danych stają się bardzo naturalne: nie ma konieczności przygotowywania poleceń SQL odpowiedzialnych za aktualizację tabel w bazie danych — wszystkie operacje wykonujemy na obiektach C#. Zachowana jest w ten sposób spójność programu, co pozwala na kontrolę typów i pełną weryfikację kodu już w trakcie komplikacji.

Modyfikacje istniejących rekordów

Pokażmy to na prostym przykładzie. Założmy, że mija Nowy Rok i z tej okazji zwiększamy wartość w polach *Wiek* wszystkich osób. Rzecznik jasna wszystkich poza kobietami, które są już pełnoletnie. Pobieramy zatem kolekcję osób, które są mężczyznami lub są niepełnoletnie. Spójnik „lub” użyty w poprzednim zdaniu rozumiany powinien być tak, jak zdefiniowany jest operator logiczny OR — powinniśmy uzyskać sumę mnogościową zbiorów osób niepełnoletnich i zbioru mężczyzn. Następnie zwiększamy wartość pola *Wiek* każdej osoby z tak uzyskanego zbioru. I najmilsza rzecz: aby zapisać nowe wartości do pliku bazy danych, wystarczy tylko wywołać metodę *SubmitChanges* na rzecz obiektu *bazaDanychAdresy*. Powyższe czynności zapisane w języku C# prezentuję na listingu 14.3.

Listing 14.3. Modyfikowanie istniejącego rekordu

```
private void button2_Click(object sender, EventArgs e)
{
    // pobieranie kolekcji
    var listaOsobDoZmianyWieku = from osoba in listaOsob
                                    where (osoba.Wiek<18 || !osoba.Imię.EndsWith("a"))
                                    select osoba;

    // wyświetlanie pobranej kolekcji
    string s = "Lista osób niebędących pełnoletnimi kobietami:\n";
    foreach (Osoba osoba in listaOsobDoZmianyWieku) s += osoba.Imię + " " +
        osoba.Nazwisko + " (" + osoba.Wiek + ")\n";
    MessageBox.Show(s);

    // modyfikowanie kolekcji
    foreach (Osoba osoba in listaOsobDoZmianyWieku) osoba.Wiek++;

    // wyświetlanie pełnej listy osób kolekcji po zmianie
    s = "Lista wszystkich osób:\n";
    foreach (Osoba osoba in listaOsob) s += osoba.Imię + " " +
        osoba.Nazwisko + " (" + osoba.Wiek + ")\n";
    MessageBox.Show(s);

    // zapisywanie zmian
    bazaDanychAdresy.SubmitChanges();
}
```

Musi pojawić się pytanie, skąd obiekt *bazaDanychAdresy* (obiekt typu *DataContext* reprezentujący w naszej aplikacji bazę danych) „wie” o modyfikacjach wprowadzonych do kolekcji *listaOsobDoZmianyWieku* otrzymanej zapytaniem LINQ z kolekcji uzyskanej metodą *bazaDanychAdresy.GetTable<Osoba>()*. Otóż stąd, że kolekcja *listaOsobDoZmianyWieku* przechowuje tylko referencje do obiektów tej tabeli, a nie kopie tych obiektów (zobacz zapytania LINQ w podrozdziale „Możliwość modyfikacji danych źródła” w rozdziale 11.). Zresztą w przypadku LINQ to SQL próba uruchomienia zapytania, w którym tworzymy kopie obiektów, skończy się zgłoszeniem wyjątku *NotSupportedException*. Tak więc wszystkie zmiany wprowadzane do kolekcji automatycznie wprowadzane są w buforze obiektu *bazaDanychAdresy*, przechowującym dane pobrane z bazy danych. A obiekt ten już potrafi zapisać je z powrotem do tabeli bazy SQL Server. Warto przy tym pamiętać, że wszystkie zmiany wprowadzane w ko-

lekcijsą przechowywane tylko w niej aż do momentu wywołania metody `SubmitChanges`. Możemy więc dowolnie (wielokrotnie) modyfikować kolekcję bez obawy, że nadużywamy zasobów bazy danych i komputera, obniżając tym samym wydajność programu.

**Wskazówka**

W trakcie projektowania aplikacji należy zwrócić uwagę, czy plik bazy danych widoczny w podoknie *Solution Explorer* jest kopiowany do katalogu, w którym umieszczana jest skompilowana aplikacja (podkatalog *bin/Debug* lub *bin/Release*). Odpowiada za to opcja *Copy to Output Directory* widoczna w podoknie *Properties* po zaznaczeniu pliku bazy danych w *Solution Explorer*. Brak kopiowania powoduje, że wersja widoczna w *Server Explorer* jest inna od tej, której używa uruchomiony program. Z kolei kopiowanie oznacza, że wszystkie zmiany wprowadzone przez program przestaną być widoczne po komplikacji, w trakcie której plik bazy danych jest nadpisywany.

Dodawanie i usuwanie rekordów

Co jeszcze możemy zmienić w tabeli? Ważna jest możliwość dodawania nowych i usuwania istniejących rekordów. Również to zadanie jest dzięki LINQ to SQL bardzo proste. W tym przypadku zmiany muszą być jednak wprowadzane wprost w obiekcie reprezentującym tabelę — w instancji klasy `DataContext`, a nie w kolekcji pobranej z niego zapytaniem LINQ. Na listingu 14.4 pokazane jest, jak dodać do tabeli nowy rekord. Wyznaczamy wartość pola `Id` dla nowego rekordu, dodając jednościodziobową wartość do największej wartości tego pola odczytanej z tabeli⁴ — korzystamy przy tym z metody rozszerzającej `Max`. Następnie tworzymy obiekt typu `Osoba` i polecamy `InsertOnSubmit` dodać go do tabeli. Rzeczywista zmiana nastąpi w momencie najbliższego wywołania metody `SubmitChanges` obiektu `DataContext`.

Listing 14.4. Dodawanie rekordu do tabeli

```
private void button3_Click(object sender, EventArgs e)
{
    // dodawanie osoby do tabeli
    int noweId = listaOsob.Max(osoba => osoba.Id) + 1;
    MessageBox.Show("Nowe Id: " + noweId);
    Osoba noworodek = new Osoba { Id = noweId, Imię = "Nela", Nazwisko = "Boderska",
        Email = "nb@bocian.pl", NumerTelefonu = null, Wiek = 0 };
    listaOsob.InsertOnSubmit(noworodek);

    // zapisywanie zmian
    bazaDanychAdresy.SubmitChanges(); // w bazie dodawany jest nowy rekord

    // wyświetlanie tabeli
    string s = "Lista osób:\n";
    foreach (Osoba osoba in listaOsob) s += osoba.Imię + " "
        + osoba.Nazwisko + " (" + osoba.Wiek + ")\n";
    MessageBox.Show(s);
}
```

⁴ Właściwie byłoby pewnie wyszukanie najmniejszej wolnej wartości, ale nie chciałem niepotrzebnie komplikować kodu. Zresztą zwykle tak postępuję w myśl zasad „prosty kod oznacza mniejsze ryzyko błędu”.

Równie łatwo usunąć z tabeli rekord lub całą grupę rekordów. Należy tylko zdobyć referencję do odpowiadającego im obiektu (względnie grupy obiektów). Dla pojedynczego rekordu należy użyć metody `DeleteOnSubmit`, dla kolekcji — `DeleteAllOnSubmit`. W obu przypadkach rekordy zostaną oznaczone jako przeznaczone do usunięcia i rzeczywiście usunięte z bazy danych przy najbliższym wywołaniu metody `SubmitChanges`. Na listingu 14.5 prezentuję metodę usuwającą z tabeli wszystkie osoby o imieniu Nela.

Listing 14.5. Wszystkie modyfikacje w bazie danych wykonywane są w momencie wywołania metody `SubmitChanges`

```
private void button4_Click(object sender, EventArgs e)
{
    // wybieranie elementów do usunięcia i ich oznaczanie
    IEnumerable<Osoba> doSkasowania = from osoba in listaOsob
                                         where osoba.Imię == "Nela"
                                         select osoba;
    listaOsob.DeleteAllOnSubmit(doSkasowania);

    // zapisywanie zmian
    bazaDanychAdresy.SubmitChanges();

    // wyświetlanie tabeli
    string s = "Lista osób:\n";
    foreach (Osoba osoba in listaOsob) s += osoba.Imię + " "
                                              + osoba.Nazwisko + " (" + osoba.Wiek + ")\n";
    MessageBox.Show(s);
}
```

Inne operacje

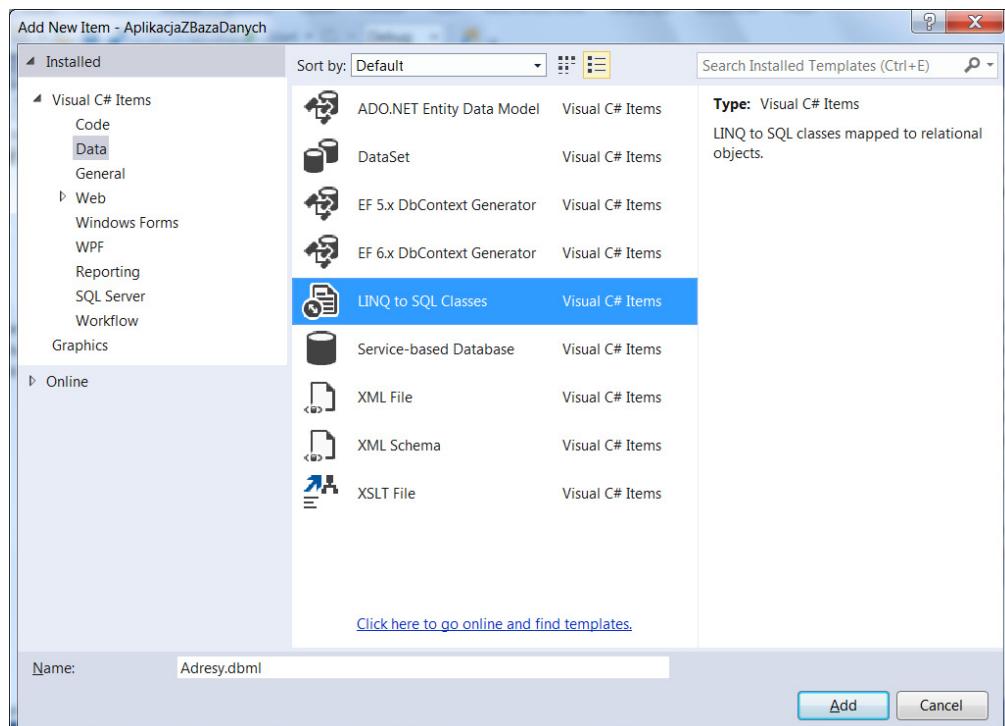
Klasa `DataContext` służy nie tylko do pobierania i modyfikacji zawartości bazy danych. Za pomocą jej metod `CreateDatabase` i `DeleteDatabase` można także tworzyć i usuwać bazy danych SQL Server z poziomu kodu. Przy użyciu metody `ExecuteCommand` można wykonać podane w argumencie polecenie SQL, a z zastosowaniem `ExecuteQuery` — uruchomić zapytanie SQL i odczytać pobrane przez nie dane.

Wizualne projektowanie klasy encji

Ręczne tworzenie klasy encji i używanie „czystego” obiektu `DataContext`, choć nie specjalnie skomplikowane, to jednak pozbawione jest szeregu zalet, z jakich będziemy mogli skorzystać, gdy użyjemy narzędzia, którego nazwa w dosłownym tłumaczeniu to *projektant obiektowo-relacyjny* (ang. *Object Relational Designer*, w skrócie: O/R Designer) i które w istocie umożliwia automatyczne odwzorowanie zawartości bazy danych w klasach C#. Co to za zalety? Automatyczna konfiguracja połączenia, automatycznie tworzone klasy encji dla każdej tabeli lub widoku i ich łatwa aktualizacja, klasa rozszerzająca klasę `DataContext` i udostępniająca tabele, widoki i procedury składowane. To moim zdaniem wystarczająca rekomendacja do używania O/R Designer. Co prawda kod samodzielnie przygotowanej klasy encji jest zwykle bardziej zwarty, ale pisząc ją, możemy popełniać błędy, których unikniemy, tworząc klasy z użyciem myszy. Zysk w czasie tworzenia jest dzięki temu ogromny.

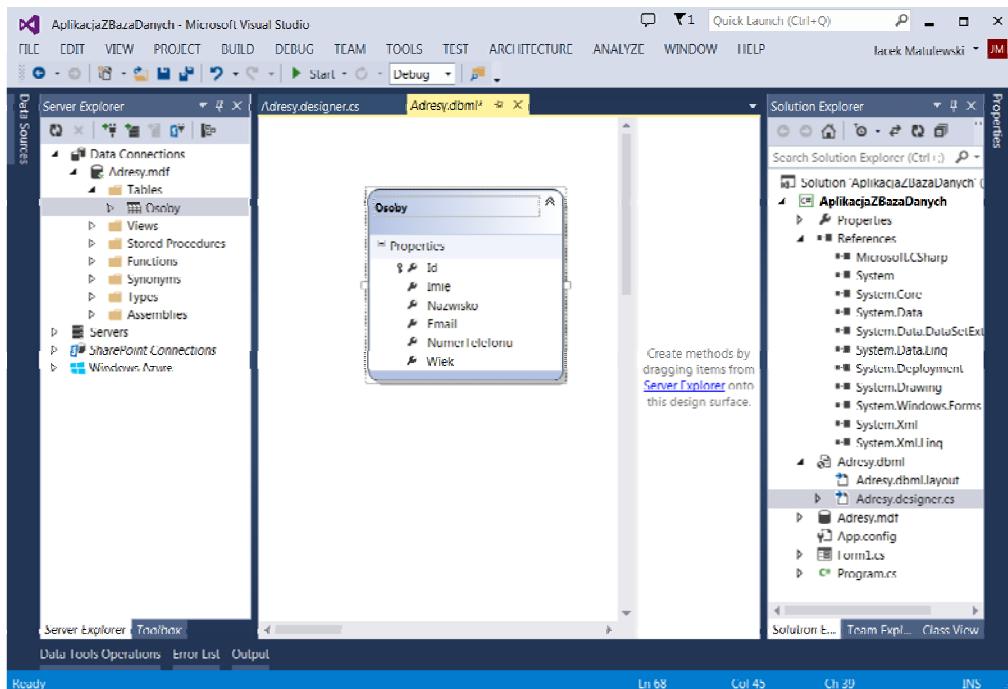
O/R Designer

Zacznijmy znów od zera, tj. od projektu z poprzedniego rozdziału, w którym do aplikacji Windows Forms dołączona jest baza SQL Server o nazwie *Adresy.mdf*. Aby utworzyć klasy ORM, wybieramy z menu *Project* polecenie *Add New Item*. Pojawi się okno widoczne na rysunku 14.1, w którym przechodzimy do kategorii *Data*. Zaznaczamy w nim ikonę *LINQ to SQL Classes*. W polu *Name* podajemy nazwę nowego pliku, tj. *Adresy.dbml*, i klikamy *Add*. Po zamknięciu okna dialogowego zobaczymy nową zakładkę. Na razie pustą, nie licząc napisów informujących, że na dwa widoczne na niej panele należy przenosić elementy z podokna *Server Explorer*.



Rysunek 14.1. Tworzenie klasy reprezentującej dane

Przenieśmy na lewy panel tabelę *Osoby* (rysunek 14.2) z podokna *Server Explorer*. Angielska konwencja nazw stosowana w kreatorze klas z pliku *Adresy.dbml* nie przystaje niestety do polskich nazw, jakich użyłem w bazie *Adresy.mdf*. Automatycznie tworzona klasa encji przejmuje bowiem nazwę tabeli, tj. *Osoby* (nasza ręcznie przygotowana klasa encji nazywała się *Osoba*). Z kolei własność tylko do odczytu zdefiniowana w klasie *AdresyDataContext*, która zwraca tabelę (a konkretnie kolekcję typu *Table ↳<0soby>*), nazywa się *Osobies* z „ies” dodanym na końcu (automatycznie stworzona angielska liczba mnoga od polskiego „Osoby”). Gdyby tabela nazywała się np. *Person*, klasa encji także nazywałaby się *Person*, a własność reprezentująca tabelę — *Persons*. Wówczas nazwy brzmiałyby dużo lepiej. A tak mamy *Osobies*.



Rysunek 14.2. Zakładka z O/R Designer

Na szczęście nazwy utworzonych przez O/R Designer klas można łatwo zmienić — wystarczy na zakładce *Adresy.dbml* kliknąć nagłówek tabeli i wpisać nową nazwę. To samo dotyczy nazw pól domyślnie ustalonych na identyczne z nazwami znalezionymi w tabeli SQL Server. Jeżeli wpiszemy nazwę różną od tej w tabeli, O/R Designer zmodyfikuje atrybut `Column`, który wskaże pole tabeli, z jakim definiowane pole obiektu ma być związane. Proponuję, abyśmy zmienili nazwę klasy encji z *Osoby* na *Osoba*. To spowoduje automatyczną zmianę nazwy własności udostępniającej tę tabelę z *Osobies* na *Osobas*. Tu się nie poprawiło, za to nazwa klasy encji będzie bardziej intuicyjna.

Utworzona przez O/R Designer klasa *Osoba* (można ją znaleźć w pliku *Adresy.designer.cs*) jest klasą encji o funkcji identycznej z funkcją zdefiniowanej przez nas wcześniej klasą *Osoba* (listing 14.1). Nie jest jednak z nią tożsama. Tę pierwszą O/R Designer wyposażył poza polami, które i my zdefiniowaliśmy, również w zbiór metod i zdarzeń. Zawiaduje nimi O/R Designer i poza wyjątkowymi sytuacjami raczej nie należy samodzielnie edytować ich kodu. Same pola, inaczej niż w naszej prostej implementacji, są w nowej klasie prywatne i poprzedzone znakiem podkreślenia. Zwróćmy uwagę, że te z nich, które odpowiadają tym polom tabeli, które dopuszczały pustą wartość, zdefiniowane zostały przy użyciu typu `System.Nullable<typ>`. Dostęp do pól możliwy jest poprzez publiczne własności. Przykład takiej własności odpowiadającej polu *Id* widoczny jest na listingu 14.6. Własność poprzedzona jest atrybutem `Column`, którego używaliśmy też w naszej wcześniejszej klasie encji. Jej argumenty, poza `IsPrimaryKey` ustawionym na `true`, są jednak inne. Ze względu na identyczną nazwę pola tabeli i własności klasy argument `Name` został pominięty. Dwa nowe argumenty to `Storage`, który wskazuje prywatne pole przechowujące wartość komórki, oraz `DbType`. W tym ostatnim

przechowywana jest informacja o oryginalnym typie pola tabeli (kolumny). Tworzą go nazwa typu dozwolonego przez SQL Server (np. Int lub NVarChar(MAX)), uzupełniona ewentualnie frazą NOT NULL, jeżeli baza danych nie dopuszcza pustych wartości dla tego pola.

Listing 14.6. Własność `Id` klasy `Osoba`. Towarzyszy jej pole zdefiniowane jako `private int _Id`

```
[global::System.Data.Linq.Mapping.ColumnAttribute(Storage="_Id", DbType="Int NOT
➥NULL", IsPrimaryKey=true)]
public int Id
{
    get
    {
        return this._Id;
    }
    set
    {
        if ((this._Id != value))
        {
            this.OnIdChanging(value);
            this.SendPropertyChanging();
            this._Id = value;
            this.SendPropertyChanged("Id");
            this.OnIdChanged();
        }
    }
}
```

Wspomniana klasa reprezentująca całą bazę danych zawiera własności odpowiadające tabelom bazy. Jest zatem, to modne ostatnio określenie, *strongly typed*. Pozwala dzięki temu na bezpieczny, tj. zachowujący typy danych, dostęp do zewnętrznego zasobu — do danych w bazie SQL Server.

Nowe klasy `AdresyDataContext` i `Osoba` można wykorzystać tak, jak robiliśmy do tej pory z klasą `Osoba` i „ręcznie” tworzoną instancją „czystej” klasy `DataContext`. Na listingu 14.7 znajduje się kod metody, w której zmiany dotyczą w zasadzie wyłącznie nazw klas: zmienionych z wcześniej używanych `DataContext` i `Osoba` na `AdresyData
➥Context` i `Osoba`. Skorzystałem także z własności `Osobas` (z „s” na końcu) dodanej w klasie `AdresyDataContext`, aby pobrać referencję do kolekcji zawierającej dane z tabeli `Osoby` (wyróżnienie w listingu 14.7). Podobnie jak w przypadku klasy `DataContext` możemy w konstruktorze klasy `AdresyDataContext` jawnie podać łańcuch połączenia. Nie jest to już jednak konieczne — ścieżka ta jest bowiem przechowywana w ustawieniach projektu (zobacz plik `App.config`).

Listing 14.7. Korzystanie z automatycznie utworzonej klasy encji do pobierania i modyfikacji danych z tabeli `SQL Server`

```
private void button1_Click(object sender, EventArgs e)
{
    // pobieranie danych z tabeli
    AdresyDataContext bazaDanychAdresy = new AdresyDataContext();
    var listaOsob = bazaDanychAdresy.Osobas;
```

```

// pobieranie kolekcji
var listaOsobPelnoletnich=from osoba in listaOsob
    where osoba.Wiek>=18
    select osoba;

// wyświetlanie pobranej kolekcji
string s = "Lista osób pełnoletnich:\n";
foreach (Osoba osoba in listaOsobPelnoletnich)
    s += osoba.Imię + " " + osoba.Nazwisko + " (" + osoba.Wiek + ")\n";
MessageBox.Show(s);

// informacje o pobranych danych
MessageBox.Show("Typ: " + listaOsobPelnoletnich.GetType().FullName);
MessageBox.Show("Ilość pobranych rekordów: " +
    listaOsobPelnoletnich.Count().ToString());
MessageBox.Show("Suma wieku wybranych osób: "
    + listaOsobPelnoletnich.Sum(osoba => osoba.Wiek).ToString());
MessageBox.Show("Imię pierwszej osoby: " + listaOsobPelnoletnich.First().Imię);

s = "Pełna lista osób:\n";
foreach (Osoba osoba in listaOsob) s += osoba.Imię + " " + osoba.Nazwisko +
    " (" + osoba.Wiek + ")\n";
MessageBox.Show(s);
}

```

Współpraca z kontrolkami tworzącymi interfejs aplikacji

Ewidentnie w tym rozdziale nadużywam metody `MessageBox.Show`. Bardziej naturalne do prezentacji danych jest korzystanie z kontrolek, np. `DataGridView`. Bez trudu można taki komponent wypełnić danymi udostępnianymi przez klasę `AdresyDataContext` bez konieczności triku z kopiowaniem, jaki musielibyśmy zastosować w przypadku „czystej” klasy `DataContext`.

Aby to pokazać, dodajmy do projektu nową formę (polecamie *Add Windows Form* z menu *Project*), pozostawiając jej domyślną nazwę, czyli *Form2.cs*. By nowa forma została utworzona i pokazana po uruchomieniu aplikacji, należy w konstruktorze klasy *Form1* (pierwszej formy) dodać polecenie: `new Form2().Show();`. W klasie *Form2* reprezentującej nową formę definiujmy pole — instancję klasy przygotowanej przez narzędzia O/R Designer'a:

```
AdresyDataContext bazaDanychAdresy = new AdresyDataContext();
```

Następnie wystarczy umieścić na formie kontrolkę `DataGridView` i przypisać jej właściwości `DataSource` odpowiednią tabelę (np. w konstruktorze za poleceniem `InitializeComponent();`):

```
dataGridView1.DataSource = bazaDanychAdresy.Osobas;
```

Jeżeli nie chcemy prezentować pełnej zawartości tabeli, właściwości `DataSource` możemy przypisać kolekcję zwracaną przez zapytanie LINQ:

```
dataGridView1.DataSource = from osoba in bazaDanychAdresy.Osobas  
    where osoba.Wiek >= 18  
    orderby osoba.Imię  
    select new { osoba.Imię, osoba.Nazwisko, osoba.Wiek };
```

Kontrolka DataGridView pozwala na edycję danych z kolekcji, jednak pod warunkiem że prezentowana w niej kolekcja nie jest zbudowana z obiektów anonimowych typów (jak jest w poprzednim punkcie), lecz z referencji do obiektów ze źródła danych (instancji klas encji):

```
dataGridView1.DataSource = from osoba in bazaDanychAdresy.Osobas  
    where osoba.Wiek >= 18  
    orderby osoba.Imię  
    select osoba;
```

Aby wprowadzone w ten sposób modyfikacje danych wysłać z powrotem do bazy, należy tylko wywołać metodę SubmitChanges instancji klasy AdresyDataContext, tj. obiektu bazaDanychAdresy.

Do formy możemy dodać także rozwijaną listę ComboBox i „podłączyć” ją do tego samego wiązania, którym z bazą połączona jest kontrolka dataGridView1. W tym celu do konstruktora drugiej formy należy dopisać kolejne polecenia:

```
comboBox1.DataSource = dataGridView1.DataSource;  
comboBox1.DisplayMember = "Email";  
comboBox1.ValueMember = "Id";
```

Pierwsze wskazuje na źródło danych rozwijanej listy — jest nim ta sama kolekcja, która prezentowana jest w kontrolce dataGridView1. Drugie preczytuje, którą kolumnę danych rozwijana lista ma prezentować w swoim interfejsie. Trzecie wskazuje kolumnę, której wartości będą dostępne we właściwości SelectedValue rozwijanej listy. Ze względu na korzystanie ze wspólnego wiązania zmiana aktywnego rekordu w siatce spowoduje zmianę rekordu w rozwijanej liście i odwrotnie.

Większość kontrolek, np. pole tekstowe TextBox czy kontrolka NumericUpDown, nie ma właściwości DataSource. Dlatego w ich przypadku wiązanie z danymi wygląda nieco inaczej:

```
if (textBox1.DataBindings.Count == 0)  
    textBox1.DataBindings.Add("Text", bazaDanychAdresy.Osobas, "Email");  
if (numericUpDown1.DataBindings.Count == 0)  
    numericUpDown1.DataBindings.Add("Value", bazaDanychAdresy.Osobas, "Wiek");
```

Instrukcje warunkowe mają zapobiec próbie zduplikowania wiązania.

Korzystanie z widoków

W identyczny sposób jak z tabel możemy pobierać dane z widoków. Aby się o tym przekonać, z podokna Server Explorer, z gałęzi *Adresy.mdf*, *Views*, przeciągnijmy do lewego panelu w zakładce *Adresy.dbml* widok *OsobyPełnoletnie*. Następnie zaprezentujmy jego zawartość, zmieniając w konstruktorze Form2 źródło danych siatki dataGridView1:

```
dataGridView1.DataSource = bazaDanychAdresy.OsobyPełnoletnies;
```

Łączenie danych z dwóch tabel — operator join

Baza danych *Adresy.mdf* oprócz tabeli *Osoby* ma także tabelę *Rozmowy* zawierającą spis rozmów, jakie odbyły osoby z tabeli *Osoby*. Jest kilka scenariuszy, zgodnie z którymi możemy postępować z tak rozdzielonymi danymi. W pierwszym postaramy się połączyć dane z obu tabel, wzbogacając dane o rozmowach informacjami o rozmówcy.

Przejdzmy na zakładkę *Adresy.dbml* (w razie potrzeby można ją otworzyć, klikając plik *.dbml* w podoknie *Solution Explorer*) i przeniesmy tabelę *Rozmowy* na lewy panel, obok tabeli *Osoby* (przemianowanej na *Osoba*). Do klasy *AdresyDataContext* dodana zostanie nowa własność o nazwie *Rozmowies* (znowu nie dopasowaliśmy się do angielskiego schematu nazw).

Pobierzmy teraz listę rozmów trwających dłużej niż dziesięć sekund wraz z personaliemi dzwoniącej osoby. W tym celu użyjemy zapytania LINQ zawierającego operator *join*, który wybierać będzie rekordy o tych samych wartościach pól *Id* w obu tabelach. Pokazuję to na listingu 14.8.

Listing 14.8. Pominąłem polecenia tworzące obiekt *bazaDanychAdresy*

```
var listaOsob = bazaDanychAdresy.Osobas;
var rozmowy = bazaDanychAdresy.Rozmowies;

IEnumerable<string> listaDlugichRozmow =
    from osoba in listaOsob
    join rozmowa in rozmowy on osoba.Id equals rozmowa.Id
    where rozmowa.CzasTrwania > 10
    select osoba.Imię + " " + osoba.Nazwisko + ", " + rozmowa.Data.ToString() +
        " (" + rozmowa.CzasTrwania + ")";

string s = "Lista rozmów trwających dłużej niż 10 sekund:\n";
foreach (string opis in listaDlugichRozmow) s += opis + "\n";
MessageBox.Show(s);
```

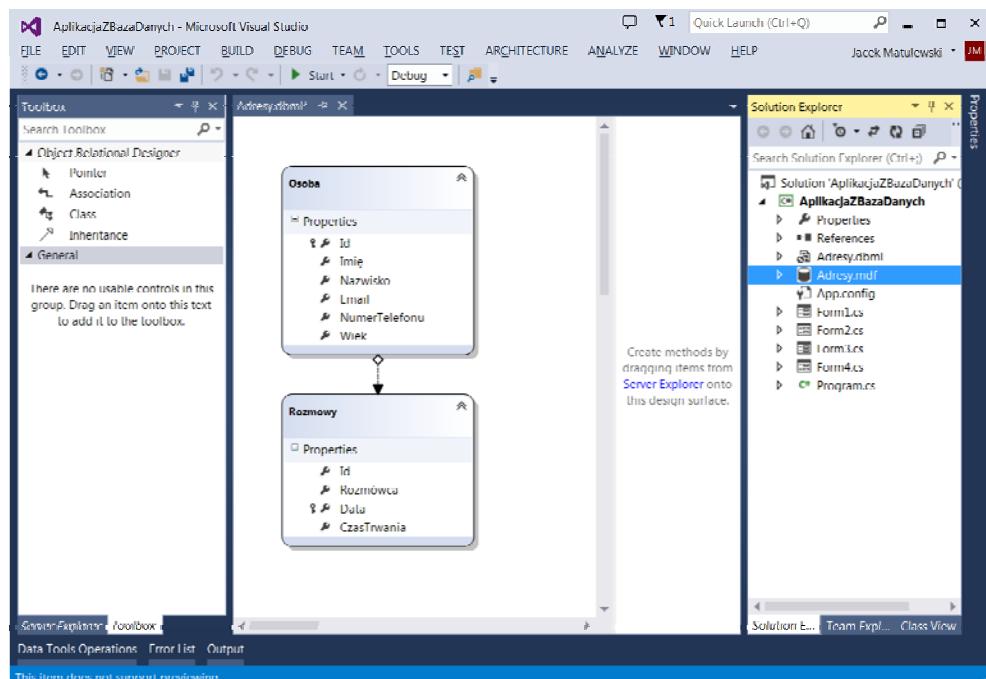
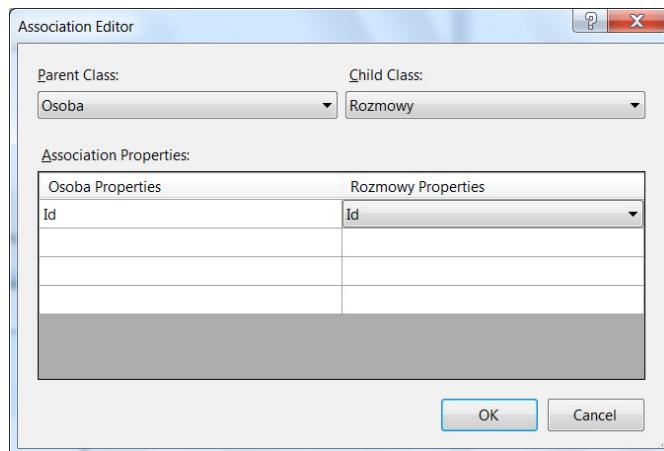
Relacje (Associations)

W innym scenariuszu możemy połączyć obie tabele w O/R Designerze. Nie jest to połączenie między tabelami w bazie danych, a jedynie połączenie między reprezentującymi te tabele klasami. W tym celu przejdźmy na zakładkę *Adresy.dbml*, kliknijmy prawym przyciskiem myszy nagłówek tabeli *Osoba* i z kontekstowego menu wybierzmy polecenie *Add, Association*. Pojawi się edytor relacji (rysunek 14.3). Z rozwijanej listy *Child Class* wybieramy *Rozmowy*. Następnie w tabeli *Association Properties* pod spodem wybieramy w obu kolumnach pole *Id* i klikamy *OK*. Ustanowione połączenie zaznaczone zostanie w O/R Designerze strzałką łączącą dwie tabele (rysunek 14.4).

Dzięki relacji łączącej obie tabele możemy uprościć zapytanie LINQ:

```
var listaDlugichRozmow = from rozmowa in rozmowy
    where rozmowa.CzasTrwania > 10
    select rozmowa.Osoba.Imię + " " +
        rozmowa.Osoba.Nazwisko + ", " +
        rozmowa.Data.ToString() +
        " (" + rozmowa.CzasTrwania + ")";
```

Rysunek 14.3.
Edytor połączenia
miedzy obiektami
reprezentującymi
tabele



Rysunek 14.4. Relacja rodzic – dziecko wiążąca dwie tabele

Taka postać sekcji select jest możliwa, ponieważ do klasy encji `Rozmowy` dodana została własność `Osoba`, która wskazuje na jeden pasujący rekord tabeli `Osoby`. Z kolei klasa encji `Osoba` zawiera własność `Rozmowies` obejmującą pełny zbiór rekordów, w których wartość pola `Id` równa jest tej, jaka znajduje się w bieżącym wierszu tabeli `Osoby`. To ostatnie pozwala na proste filtrowanie danych bez konieczności przygotowywania zapytania LINQ. Tak więc kolekcja uzyskana poleceniem:

```
var listaRozmowWincentegoKotka =
    bazaDanychAdresy.ListaOsobs.Single(osoba => osoba.Id == 3).Rozmowies;
```

zawiera rekordy identyczne z rekordami kolekcji uzyskanej przy użyciu zapytania:

```
var listaRozmowWincentegoKotka = from rozmowa in rozmowa
                                         where rozmowa.Id == 3
                                         select rozmowa;
```

Korzystanie z procedur składowanych

W momencie uruchamiania metody `SubmitChanges` obiekt klasy `DataContext` automatycznie tworzy zbiór poleceń SQL, które modyfikują zawartość bazy danych. Podobnie jest zresztą w czasie pobierania danych. Wówczas zapytanie LINQ tłumaczone jest na zapytanie SQL i wysyłane do bazy danych. To jest podstawowa idea technologii LINQ to SQL.

Zamiast korzystać z tych tworzonych automatycznie procedur i zapytań, możemy użyć własnych procedur składowanych w bazie danych (ang. *stored procedures*). Jest to w pewnym sensie krok wstecz, wymaga bowiem programowania w SQL, ale za to pozwala na zachowanie pełnej elastyczności podczas korzystania z SQL Server. I jeśli nawet z procedur składowanych nie ma sensu korzystać do pobierania danych, do tego lepiej nadają się widoki, warto je poznać, aby uruchamiać dowolne polecenia SQL, które pozwolą np. tworzyć nowe tabele w bazie (choć i to lepiej robić z poziomu klasy `DataContext`). Czasem korzystanie z procedur składowanych jest również podyktowane względami bezpieczeństwa.

Pobieranie danych za pomocą procedur składowanych

W bazie danych *Adresy.mdf* zdefiniowane są trzy procedury składowane: zapytanie `ListaOsobPełnoletnich`, `AktualizujWiek` i `TwórzNowąTabelę`. Wystarczy przeciągnąć je z podokna *Server Explorer* do pustego prawego panelu w zakładce *Adresy.dbml*. Zaczniemy od metody zapytania `ListaOsobPełnoletnich`. Spowoduje to dodanie do klasy `AdresyDataContext` metody `ListaOsobPełnoletnich`. Zwraca ona dane pobrane przez zapytanie SQL w postaci kolekcji elementów typu `ListaOsobPełnoletnichResult`.

Typ `ListaOsobPełnoletnichResult` został automatycznie utworzony za pomocą O/R Designer. W przypadku pobierania wszystkich pól z tabeli, tj. gdy w zapytaniu SQL po słowie `SELECT` widoczna jest gwiazdka (*), nowa klasa będzie równoważna typowi `Osoba`. Zapytanie SQL może jednak zwracać tylko wybrane pola, a wówczas tworzenie nowego typu jest bardziej przydatne.



Procedurę składowaną można przetestować bez konieczności wywoływania jej w kodzie C#. Wystarczy w podoknie *Server Explorer* wywołać z jej menu kontekstowego polecenie *Execute*. Wówczas w głównej części okna pojawi się faktycznie wykonany kod SQL, a pod spodem uzyskane w ten sposób dane.

Dzięki nowej metodzie uruchomienie składowanego zapytania SQL i odebranie zwracanych przez nie danych jest dziedzinie proste:

```
IEnumerable<ListaOsobPelnoletnichResult> listaOsobPelnoletnich =  
    bazaDanychAdresy.ListaOsobPelnoletnich();  
  
    string s = "Lista osób pełnoletnich (procedura składowana):\n";  
    foreach (var osoba in listaOsobPelnoletnich)  
        s += osoba.Imię + " " + osoba.Nazwisko + " (" + osoba.Wiek + ")\n";  
    MessageBox.Show(s);
```

Modyfikowanie danych za pomocą procedur składowanych

Równie niewiele kłopotu sprawia uruchomienie procedury *AktualizujWiek*. Przeciągamy ją na prawy panel zakładki *Adresy.dbml* i od razu możemy ją uruchomić z poziomu kodu C#, wywołując instrukcję:

```
bazaDanychAdresy.AktualizujWiek();
```

Wówczas zmiany zostaną wprowadzone do kopii bazy danych z katalogu roboczego uruchomionej aplikacji.

Zmiany w tabeli bazy danych będą wprowadzone natychmiast i bezpośrednio, tj. np. bez konieczności uruchamiania metody *SubmitChanges* na rzecz instancji klasy *AdresyDataContext*. Z tego wynika też, że jeżeli wcześniej pobraliśmy kolekcję danych zapytaniem LINQ, należy to teraz powtórzyć, aby uwzględnić aktualizacje.



Pamiętajmy, że w trakcie działania programu zmiany wprowadzane są do pliku bazy danych skopiowanego do podkatalogu *bin/Debug*, a nie do pliku widocznego np. w narzędziach projektowania Visual Studio.

Podobnie będzie z procedurą *TwórzNowąTabelę*. W jej przypadku jest jednak poważny kłopot. Może być bowiem wywołana tylko raz. Wówczas stworzy tabelę o nazwie *Faktury*, przez co kolejne wywołanie już nie może się powieść. Niestety nowa tabela nie będzie odwzorowana w klasie *AdresyDataContext*, wobec czego nie ma prostego sposobu, żeby sprawdzić, czy tabela *Faktury* już istnieje, czy jeszcze nie.

```
try  
{  
    bazaDanychAdresy.TwórzNowąTabelę();  
    MessageBox.Show("Tabela 'Faktury' została utworzona");  
}  
catch(Exception exc)  
{  
    MessageBox.Show("Błąd podczas tworzenia tabeli 'Faktury': " + exc.Message);  
}
```


Rozdział 15.

Kreator źródeł danych

Ponownie wróćmy do projektu z rozdziału 13. W nim szybko odtworzymy klasy LINQ to SQL z odwzorowanymi za pomocą O/R Designer tabelami *Osoby* i *Rozmowy*. W tym celu:

1. Wczytaj projekt z bazą danych *Adresy.mdf* — w stanie, jaki miał on na końcu rozdziału 13.
2. Z menu *Project* wybierz *Add New Item...*.
3. W otwartym oknie *Add New Item* — *AplikacjaZBazaDanych*, w lewym panelu, zaznacz gałąź *Data*, a następnie w środkowym — pozycję *LINQ to SQL Classes*. Zmień nazwę nowego pliku na *Adresy.dbml* i zatwierdź jego utworzenie, klikając przycisk *Add*.
4. Otwarta zostanie zakładka *Adresy.dbml*, na którą z podokna *Server Explorer* przeciągamy tabele *Osoby* i *Rozmowy*.
5. Tworzymy relację między tymi tabelami, wybierając z menu kontekstowego tabeli *Osoby* (na zakładce *Adresy.dbml*) polecenie *Add, Association*. Pojawi się okno *Association Editor*. W jego lewej kolumnie z rozwijanej listy wybieramy *Osoby*, a w prawej — *Rozmowy*. Niżej w siatce wybieramy z lewej i prawej kolumny pola *Id* (rysunek 14.6) i klikamy *OK*.

Wskazaliśmy obie tabele *Osoby* i *Rozmowy*, ale tak naprawdę korzystać będziemy tylko z tej pierwszej. Obecność drugiej nie wnosi zasadniczo niczego nowego. Ważna będzie natomiast relacja wiążąca te dwie tabele i ją wykorzystamy.

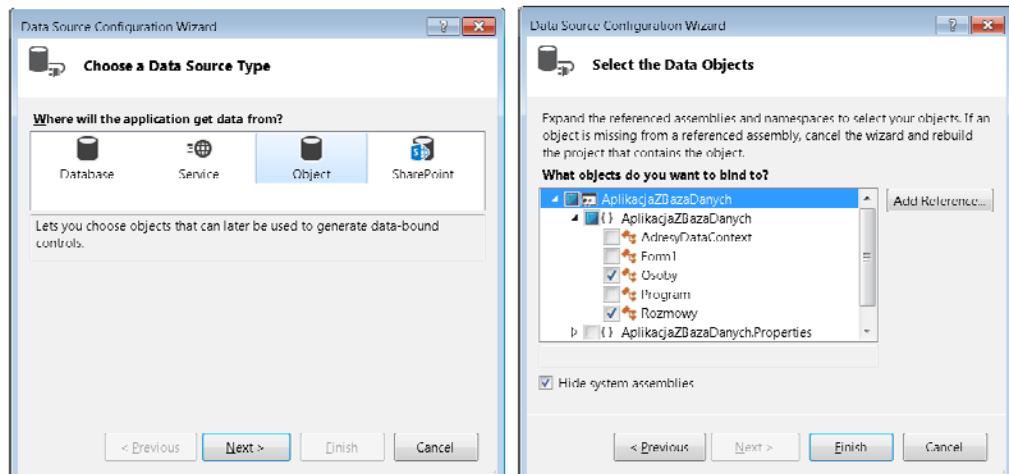
Kreator źródła danych

O/R Designer ponownie przygotował klasę *AdresyDataContext* udostępniającą wskażane przez nas elementy bazy danych i zawierającą definicje klas encji dla dwóch wskazanych przez nas tabel. W poprzednim rozdziale klasę tą wykorzystywaliśmy, samodzielnie tworząc jej instancje, których następnie używaliśmy do pobierania i modyfikowania danych. Do edycji danych używaliśmy także kontrolek Windows Forms. Możemy jednak pójść o krok dalej. Jeżeli w Visual Studio utworzymy tzw. *źródła*

danych (ang. *data sources*), to będziemy mogli myślą bardzo szybko i bezpiecznie tworzyć interfejs użytkownika; Visual Studio wspiera wiązanie danych z kontrolkami tworzącymi interfejs aplikacji. Co ciekawe, poziom, na którym posługujemy się źródłem danych, i narzędzia pozwalające na szybkie tworzenie interfejsu użytkownika nie zależą już od tego, jaki mechanizm ORM w rzeczywistości dostarcza dane. Zatem narzędzia, które poznamy w tym rozdziale, będą działały równie dobrze dla tradycyjnego ADO.NET z klasą *DataSet* jako reprezentantem bazy danych (zobacz następny rozdział), jak i z Entity Framework (rozdział 17.).

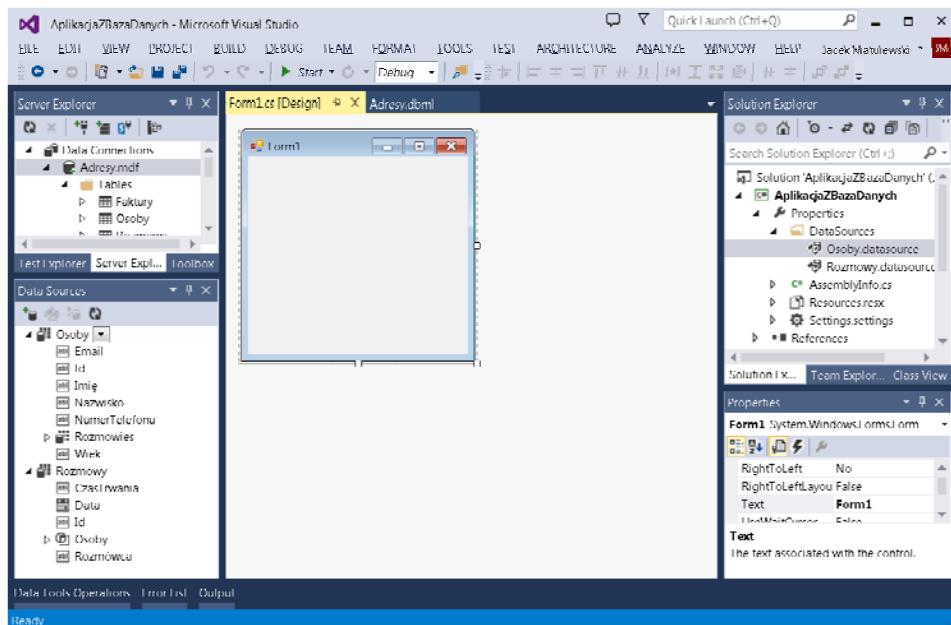
Stwórzmy zatem źródło danych:

1. Z menu *Project* wybierz polecenie *Add New Data Source....*
2. Pojawi się kreator *Data Source Configuration Wizard*. W jego pierwszym kroku należy wskazać, jaki typ danych będzie udostępniany przez źródło danych (rysunek 15.1, lewy); w przypadku LINQ to SQL należy zaznaczyć ikonę *Object*. Kliknij *Next >*.
3. W kolejnym kroku kreatora zaznacz klasy, które zawierają informacje na temat struktury tabel *Osoba* i *Rozmowy* (rysunek 15.1, prawy). Czasem zaraz po utworzeniu pliku *.dbml* klasy te są niewidoczne — wówczas pomaga ponowne wczytanie projektu.



Rysunek 15.1. Pierwszy krok kreatora obiektu — źródła danych LINQ to SQL

4. Kliknij przycisk *Finish*.
5. Po zamknięciu kreatora przejdź do podokna *Data Sources* (dostępne w menu *View, Other Windows*), w którym widoczne będzie utworzone przed chwilą źródło danych. Widok tego okna zmienia się nieco w zależności od aktywnej zakładki. Nas będzie ono interesowało w kontekście widoku projektowania formy (rysunek 15.2).



Rysunek 15.2. Podokno Data Sources z dwoma źródłami danych

Źródła danych nie są klasami zdolnymi do buforowania danych ani w żaden sposób nie odwzorowują danych. Są warstwą, która pozwala na abstrahowanie od konkretnego mechanizmu ORM i m.in. wygodne tworzenie interfejsu użytkownika. Zdefiniowane są w plikach XML zapisanych w podkatalogu *Properties/DataSources* (zobacz podokno *Solution Explorer* na rysunku 15.2). Po usunięciu komentarza ostrzegającego przed samodzielnią edycją tego pliku zawiera on bardzo proste drzewo elementów (listing 15.1), które wskazuje na klasę *Osoba* z pliku *Adres.designer.cs* jako rzeczywiste źródło danych.

Listing 15.1. Źródło danych w przypadku tabeli Osoba

```
<?xml version="1.0" encoding="utf-8"?>
<GenericObjectDataSource DisplayName="Osoby" Version="1.0"
  xmlns="urn:schemas-microsoft-com:xml-msdatasource">
  <TypeInfo>AplikacjaZBazaDanych.Osoby, Adres.designer.cs, Version=0.0.0.0,
    Culture=neutral, PublicKeyToken=null</TypeInfo>
</GenericObjectDataSource>
```



Wskazówka

Czasem, raczej rzadko, po wczytaniu projektu do Visual Studio w podoknie *Data Sources* nie są widoczne żadne źródła danych, choć wiemy, że dodaliśmy je do projektu. W takiej sytuacji należy w podoknie *Solution Explorer* przejść do podkatalogu *Properties/DataSources* i dwukrotnie kliknąć któryś z plików z rozszerzeniem *.datasource*.

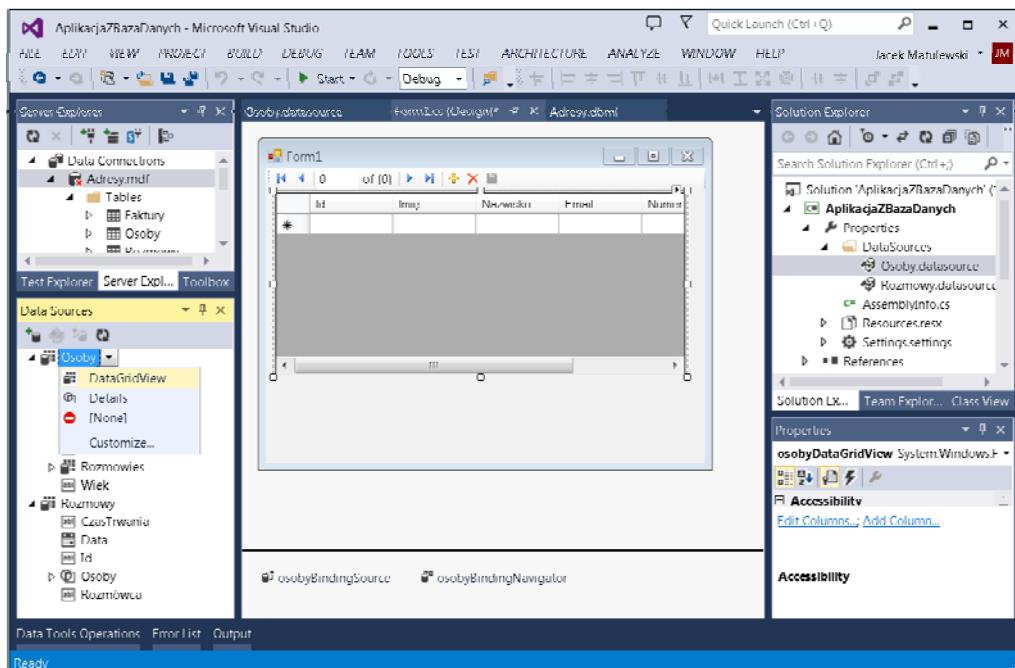
W opisany wyżej przypadku „pod” źródłem danych leży klasa *Osoba*, która odwzorowuje tabelę *Osoby* z bazy danych *Adresy.mdf*. W poniższym tekście nie będę akcentował tej wielowarstwowej struktury i często będę pisał, że np. w źródle danych widoczne są pola tabeli.

Zautomatyzowane tworzenie interfejsu użytkownika

Tak zdefiniowane źródło danych nie zawiera samych danych, a jedynie informacje o ich strukturze. To jednak wystarczy, aby bardzo przyspieszyć projektowanie interfejsu użytkownika w aplikacjach bazodanowych. Jeżeli bowiem dysponujemy źródłem danych, to możemy zaprojektować interfejs aplikacji, korzystając z metody *przeciągnij i upuść*. Ograniczę się do najczęściej spotykanych przypadków: prezentacji tabeli w siatce, tworzenia formularza dla wszystkich pól tabeli oraz konfigurowania pary siatek w układzie *master-details*.

Prezentacja tabeli w siatce

Najprostsze co można zrobić, to przeciągnąć z podokna *Data Sources* na podgląd formy element *Osoby* lub *Rozmowy* (ja wybrałem ten pierwszy). Spowoduje to umieszczenie na formie siatki *DataGridView*, w której widoczne będą kolumny odpowiadające polom tabeli. Siatka nie będzie wypełniona danymi (także po uruchomieniu aplikacji), ale będzie zawierać kolumny odpowiadające polom tabeli. Wraz z pierwszą kontrolką umieszczoną w ten sposób na formie na pasku pod formą pojawi się również komponent wiązania (*osobyBindingSource*) oraz kontrolka nawigacyjna (*osobyBindingNavigator*). Tę ostatnią można zobaczyć także na szczytce podglądu formy (rysunek 15.3).



Rysunek 15.3. Siatka i kontrolka nawigacyjna utworzona poprzez przeniesienie elementu *Osoby* z podokna *Data Sources*

Klasa BindingSource

Aby po uruchomieniu aplikacji siatka zawierała dane, należy odpowiednio „podpiąć” kontrolkę wiązania osobaBindingSource do klasy AdresyDataContext (klasa ORM w technologii LINQ to SQL zdefiniowana w pliku *Adresy.dbml*, zobacz poprzedni rozdział). Zaczniemy od zdefiniowania pola formy będącego instancją klasy AdresyDataContext. Następnie w konstruktorze formy powiążemy udostępnianą przez ten obiekt tabelę Osoby z obiektem wiązania osobyBindingSource (listing 15.2). Obiekt ten jest już źródłem danych dla siatki, dzięki czemu po uruchomieniu aplikacji zobaczymy w siatce rekordy z tabeli *Osoby* (rysunek 15.4). To wiązanie powoduje, że zmiany w danych wprowadzone za pomocą siatki (lub innych kontrolek) są automatycznie przenoszone do kolekcji przechowywanych w instancji klasy AdresyDataContext. Aby więc zmodyfikowane dane zapisać do pliku bazy danych, wystarczy wywołać metodę *bazaDanychAdresy.SubmitChanges*.

Listing 15.2. Wiązanie danych między klasami LINQ to SQL a kontrolkami

```
public partial class Form1 : Form
{
    AdresyDataContext bazaDanychAdresy = new AdresyDataContext();

    public Form1()
    {
        InitializeComponent();
        osobyBindingSource.DataSource = bazaDanychAdresy.Osobies;
    }
}
```

Rysunek 15.4.

Warto zająć się zakotwiczeniem siatki, aby wygodnie oglądać w niej dane

	Id	Imię	Nazwisko	Email	Numer
▶	1	Jacek	Matulewski	jacek@phys...	123
	2	Jan	Kowalski	jan.kowalski...	
	3	Wincenty	Kotek	kotek@firma...	999
	4	Tomasz	Trombalski	tt@firma.pl	999
	5	Anna	Kowalska	ak@firma.pl	888
	6	Karolina	Kowalska	karolina@w...	
	7	Bartosz	Kowalski	bartosz@po...	

Sortowanie

Rekordy prezentowane są w siatce w kolejności, w jakiej dodawane były do bazy danych. Nic jednak nie stoi na przeszkodzie, aby rekordy zbuforowane w kolekcjach przechowywanych przez aplikację posortować np. w kolejności alfabetycznej nazwisk. W tym celu należy:

1. Przejść do widoku projektowania formy (zakładka *Form1.cs [Design]*).
2. Zaznaczyć komponent *osobyBindingSource*.
3. Korzystając z okna właściwości, przypisać właściwości *Sort* nazwę kolumny, względem której dane mają być posortowane. Wpisujemy tam *Nazwisko*.

Po ponownym uruchomieniu aplikacji przekonamy się, że teraz zawartość siatki ułożona jest już alfabetycznie w kolejności wyznaczonej przez nazwiska zapisanych osób.

Filtrowanie

Zachęcamy łatwym wprowadzeniem sortowania, możemy zechcieć użyć także właściwości *Filter* obiektu *osobyBindingSource*. Możemy tej właściwości przypisać łańcuch definiujący filtr identyczny z tym, jaki stosuje się w zapytaniu SQL po słowie kluczowym *WHERE*, np. *Wiek < 50*. Okazuje się jednak, że to nie zadziała. W przypadku technologii LINQ to SQL źródło danych nie wspiera sortowania (właściwość *osobyBindingSource.SupportsFiltering* równa jest *false*). Inaczej będzie w przypadku tradycyjnego ADO.NET i klasy *DataSet*, co zobaczymy w kolejnym rozdziale.

Co możemy wobec tego zrobić? Rozwiążanie jest bardzo proste. Wystarczy zmodyfikować polecenie, jakie umieściliśmy w konstruktorze, dodając do niego filtrowanie (listing 15.3). Zresztą tak samo można by było dodać sortowanie. W ogóle instrukcję tę można zmienić na bardziej rozbudowane zapytanie LINQ. W dalszych przykładach używam jednak danych niefiltrowanych.

Listing 15.3. Zmodyfikowane polecenie ładujące dane

```
public Form1()
{
    InitializeComponent();

    osobyBindingSource.DataSource = bazaDanychAdresy.Osobies.Where(o => o.Wiek < 50);
}
```

Prezentacja rekordów tabeli w formularzu

Siatka jest wygodna, bo pozwala na obejrzenie całej tabeli. Jednak czasem wygodniej jest użyć formularza — pozwala to na narzucenie użytkownikowi aplikacji skoncentrowania się na wybranym rekordzie tabeli, dzięki czemu unikamy przypadkowych zmian. Aby utworzyć taki formularz, wystarczy w pozycji *Osoby* w podoknie *Data Sources* z rozwijanej listy widocznej na rysunku 15.3 wybrać *Details* zamiast domyślnego *DataGridView*. Następnie wybierzmy typy kontrolek odpowiadające poszczególnym polom tabeli. Domyślnym jest pole tekstowe *TextBox*. Warto je zmienić na *ComboBox* w przypadku pola *Email* i *NumericUpDown* w przypadku pola *Wiek*. W efekcie po przeniesieniu elementu *Osoby* na formę (np. poniżej siatki) umieszczony zostanie na niej cały zbiór kontrolek (rysunek 15.5). Obok kontrolek umieszczone zostaną etykiety z opisami.

Rysunek 15.5.

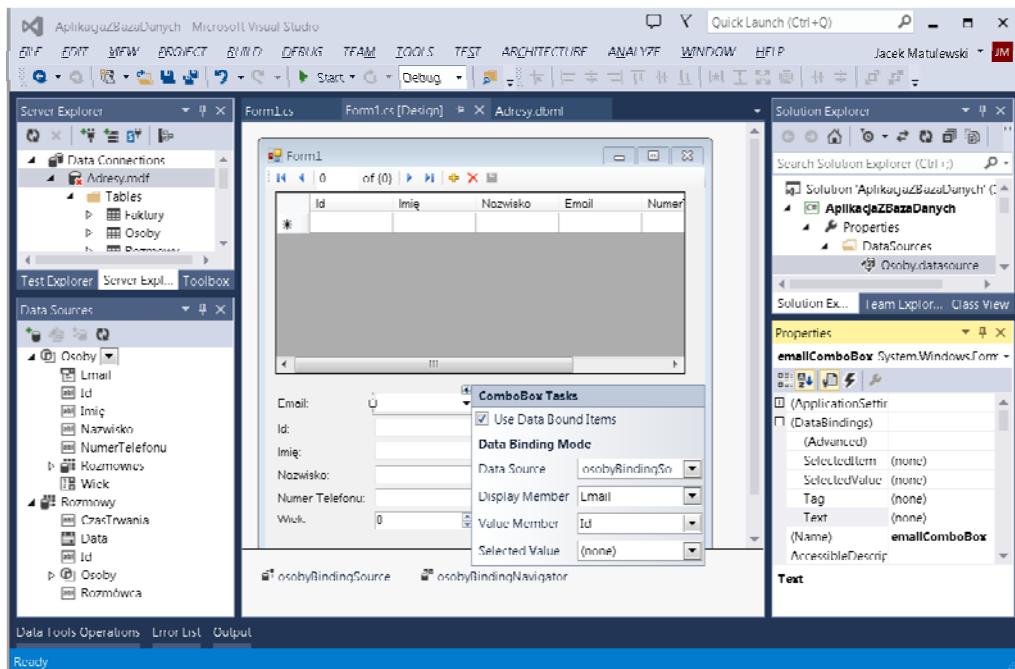
Formularz utworzony dzięki źródłom danych

Id	Imię	Nazwisko	Email	Numer Telefonu
1	Jacek	Matulewski	jacek@phys...	123
2	Jan	Kowalski	jan.kowalski...	
3	Wincenty	Kotek	kotek@firma...	999
4	Tomasz	Trombalski	tt@firma.pl	999
5	Anna	Kowalska	ak@firma.pl	888
6	Karolina	Kowalska	karolina@w...	
7	Bartosz	Kowalski	bartosz@po...	

Wszystkie kontrolki tworzące interfejs użytkownika związane są z jednym obiektem wiązania osobyBindingSources, a za jego pośrednictwem z jednym obiektem typu Adresy ↵DataContext. W związku z tym zmiana aktualnego rekordu w siatce spowoduje automatyczną zmianę danych w pozostałych kontrolkach.

Rozczarowane mogą być jednak osoby, które miały nadzieję, że w kontrolce ComboBox, którą wybraliśmy dla pola Email, po jej rozwinięciu widoczne będą adresy e-mail wszystkich osób z tabeli. Niestety tak nie jest. Możemy to jednak łatwo uzyskać:

1. Wystarczy w widoku projektowania zaznaczyć tę rozwijaną listę (obiekt emailComboBox).
2. W podoknie *Properties* rozwinąć zbiór (*Data Bindings*), a w nim usunąć wiązanie własności *Text* z polem *Email* udostępnianym przez osobyBindingSource (należy ustawić na *None*).
3. Następnie na podglądzie formy, w liście bocznej tej kontrolki zaznaczamy pole opcji *Use Data Bound Items* (rysunek 15.6)
4. Wówczas pojawią się dodatkowe rozwijane listy, w których wskazujemy obiekt osobyBindingSource jako źródło danych (lista *Data Source*). Wybieramy także, że w liście wyświetlane będą adresy e-mail (pozycja *Email* w rozwijanej liście *Display Member*), a dodatkowo, że z własności *SelectedValue* kontrolki będziemy mogli odczytać identyfikator Id osoby, której adres e-mail został wybrany (pozycja *Id* w liście *Value Member*).
5. Dodatkowo można zmienić własność *DropDownStyle* rozwijanej listy na *DropDownList*. Uniemożliwi to wprawdzie edycję adresu e-mail, ale za to ułatwi rozwijanie listy.



Rysunek 15.6. Zmiana sposobu wiązania danych w przypadku kontrolki ComboBox

Teraz po uruchomieniu programu i rozwinięciu listy z opisem *Email* widoczne będą w niej adresy e-mail wszystkich osób z tabeli. Wybranie jednego z nich spowoduje zmianę aktywnego rekordu zaznaczonego w siatce i widocznego w pozostałych kontrolkach.



Wskazówka Możesz przyciągać na formę także poszczególne pola tabeli *Osoba* widoczne w podoknie *Data Sources*. Powstanie wówczas pojedyncza kontrolka wybranego typu wraz z automatycznie dodanym opisem. Kontrolki te również będą związane ze źródłem *osobaBindingSource*.

Dwie siatki w układzie master-details

W klasie ORM *AdresyDataContext* tabele *Osoby* i *Rozmowy* połączone są relacją jeden do wielu poprzez pole *Id*. Ten fakt znajduje swoje odbicie w źródle danych: w elemencie *Osoby* widoczna jest gałąź *Rozmowies*. Reprezentuje ona zbiór rozmów wybranej osoby, a więc zbiór rekordów z tabeli *Rozmowy*, w których pole *Id* równe jest polu *Id* z aktywnego rekordu tabeli *Osoby*. Natomiast w tabeli *Rozmowy* widoczna jest gałąź *Osoby*, która zawierać będzie dane jednego rekordu z tabeli *Osoby* o identyfikatorze *Id* odpowiadającym temu zapisanemu w rekordzie rozmowy. Wykorzystajmy podelementy *Rozmowies* do utworzenia dwóch związanych ze sobą siatek. W pierwszej siatce pokazywane będą wszystkie osoby z tabeli *Osoby*. Tę siatkę już *de facto* przygotowaliśmy wcześniej. Dodamy do niej drugą, w której pokazywane będą dodatkowe szczegóły dotyczące wybranej w pierwszej siatce osoby; w naszym przykładzie

— rozmowy tej osoby zapisane w tabeli *Rozmowy*. Aby to osiągnąć, wystarczy na formę przeciągnąć element *Osoby/Rozmowies* i voilà. Jeżeli nie chcemy oglądać wszystkich pól tabeli, z menu kontekstowego siatki wybierzmy polecenie *Edit Columns....* Pojawi się okno dialogowe pozwalające na wybór prezentowanych w siatce kolumn.

Rozdział 16.

Tradycyjne ADO.NET (DataSet)

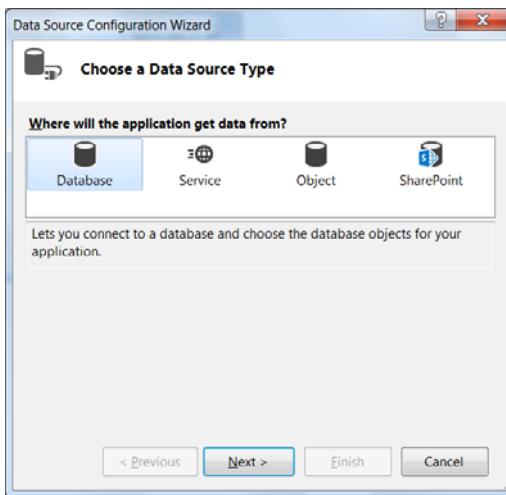
W rozdziale 13. przedstawiłem panoramę technologii pozwalających na wykorzystanie baz danych w aplikacjach .NET. Wynika z niej, że tradycyjne rozwiązania oparte na kontrolkach DataSet są obecnie *passé*. Główny zarzut stawiany tej technologii to konieczność pracy na stosunkowo niskim poziomie, z edycją kodu T-SQL włącznie. Wypierają je nowsze rozwiązania oferowane w Visual Studio, w szczególności LINQ to SQL, który jest jednak ograniczony jedynie do współpracy z bazami danych SQL Server oraz najnowszym Entity Framework. Bardzo popularny jest również nHibernate. Pomimo tego chciałbym w tym rozdziale zaprezentować także tradycyjne rozwiązanie ADO.NET opierające się na klasie DataSet. Głównym powodem jest jego wcześniejsza popularność, która przekłada się na znaczną ilość projektów, w których jest nadal używane.

Konfiguracja źródła danych DataSet

Kolejny raz cofamy się do projektu aplikacji z bazą danych SQL Server *Adresy.mdf* przygotowanego w rozdziale 13. Do projektu dodamy klasę ORM dziedziczącą z klasy DataSet, w której udostępnione zostaną dane z tabel *Osoby* i *Rozmowy* bazy danych. Proszę zwrócić uwagę, że tym razem utworzymy klasę DataSet za pomocą kreatora, który utworzy też źródło danych. Te dwa elementy nie są w tradycyjnym ADO.NET związane.

1. Z menu *Project* wybieramy polecenie *Add New Data Source....* Pojawi się kreator *Data Source Configuration Wizard*.
2. W pierwszym kroku kreatora wybieramy źródło danych. Tym razem powinniśmy zaznaczyć ikonę *Database* (rysunek 16.1) i kliknąć przycisk *Next*.
3. W kolejnym kroku wybieramy model dostępu do bazy danych, który nas interesuje. Dostępny jest jednak tylko *Dataset*. Klikamy wobec tego *Next*.

Rysunek 16.1.
Kreator komponentu
DataSet



4. Teraz mamy możliwość wyboru egzemplarza bazy danych, z którego dane chcemy pobierać. W naszym projekcie jest jednak tylko jeden (*Adresy.mdf*). Warto natomiast zwrócić uwagę na łańcuch konfigurujący połączenie (ang. *connection string*), który powinien być widoczny w dolnej części okna kreatora (należy kliknąć przycisk z plusem). Powinien on być podobny do tego, którego używaliśmy w rozdziale 14. do skonfigurowania klasy *DataContext* (zobacz także podrozdział „Łańcuch połączenia” z rozdziału 13.):

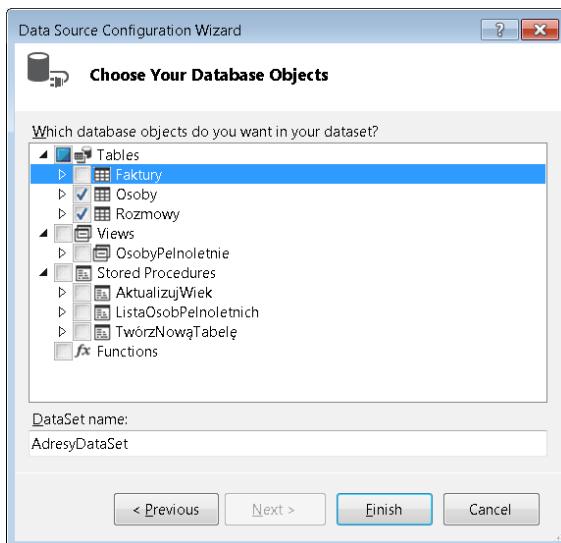
```
Data Source=(LocalDB)\v11.0;AttachDbFilename=
→|DataDirectory|\Adresy.mdf;Integrated Security=True
```

5. Klikamy *Next*. Kolejny krok kreatora zawiera pytanie o to, czy łańcuch konfigurujący połączenie z bazą danych powinien być zapisany do pliku konfiguracyjnego aplikacji. Pozostanmy przy domyślnym ustawieniu (tj. łańcuch zostanie tam zapisany pod nazwą *AdresyConnectionString*) i ponownie kliknijmy *Next*.
6. Teraz zobaczymy okno pozwalające wybrać, które tabele bazy danych mają być udostępnione w aplikacji. Należy być jednak cierpliwym, bo pobranie informacji o tabelach może zajść dłuższą chwilę. Jak już się tego doczekamy, rozwińmy gałąź *Tables* i zaznaczmy tabele *Osoby* i *Rozmowy* (rysunek 16.2).
7. To ostatni krok kreatora. Teraz możemy kliknąć przycisk *Finish*.

W efekcie do projektu zostanie dodany plik *AdresyDataSet.xsd* wraz z plikami towarzyszącymi. W pliku tym zdefiniowana jest klasa *AdresyDataSet*, która w naszym projekcie będzie reprezentowała bazę danych. Została ona zaprojektowana w taki sposób, że za jej pomocą dostępne będą dane z tabel *Osoby* i *Rozmowy* bazy *Adresy.mdf*. *AdresyDataSet* to klasa potomna względem *System.Data.DataSet*; jej definicja znajduje się w pliku *AdresyDataSet.Designer.cs*. Plik ten nie powinien być jednak edytowany bezpośrednio przez programistę — zamiast tego Visual Studio udostępnia narzędzia projektowania wizualnego (o tym niżej). Wbrew temu zaleceniu zajrzyjmy do tego pliku. Zobaczmy, że klasa *AdresyDataSet* zawiera prywatne pola *tableOsoby* typu *OsobyDataTable* i *tableRozmowy* typu *RozmowyDataTable*. Typy te także zdefiniowane są

Rysunek 16.2.

Możemy wybrać elementy tabeli, które zostaną udostępnione aplikacji



w pliku *AdresyDataSet.Designer.cs* i dziedziczą z `System.Data.DataTable`. Oba pola udostępniane są przez własności tylko do odczytu (tj. zawierają jedynie sekcje get) o nazwach *Osoby* i *Rozmowy*.

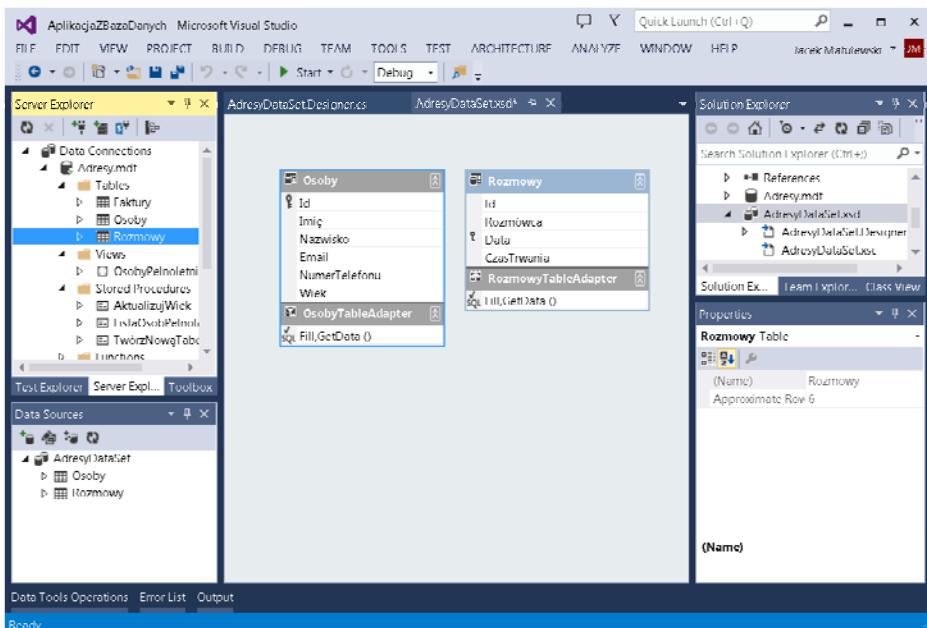
Wskazówka

W odróżnieniu od poznanej w poprzednich rozdziałach klasy `DataContext` i technologii LINQ to SQL tradycyjne ADO.NET z klasą `DataSet` wspiera nie tylko bazy danych SQL Server, ale także Microsoft Access czy bazy łączone poprzez mostek ODBC. Po przygotowaniu komponentu `DataSet` dla dalszego rozwoju projektu jest w dużym stopniu obojętne, z jakiego typu bazą danych mamy do czynienia. Aby to zilustrować, do kodów źródłowych dołączonych do książki dodany został projekt podobny do opisywanego w tym rozdziale, ale oparty na pliku `.accdb` bazy Microsoft Access.

Tworząc klasę ORM, a więc klasę `AdresyDataSet`, utworzyliśmy także źródła danych. A właściwie klasa `AdresyDataSet` jest źródłem danych. Możemy się o tym przekonać, otwierając podokno *Data Sources*. Widoczna jest w nim pozycja *AdresyDataSet*, a w niej dwie gałęzie odpowiadające tabelom *Osoby* i *Rozmowy* (rysunek 16.3).

Edycja klasy DataSet i tworzenie relacji między tabelami

Wspomniałem już, że klasa `AdresyDataSet` zdefiniowana jest w pliku *AdresyDataSet.Designer.cs*. To jest plik towarzyszący plikowi *AdresyDataSet.xsd*. Znajdzmy go w podoknie *Solution Explorer* i dwukrotnie kliknijmy. Pojawi się zakładka — bardzo podobna do tej, za pomocą której konfigurowaliśmy w rozdziale 14. klasę `AdresyData-Context` (rysunek 16.3). Również i w tym przypadku możemy z podokna *Server Explorer* przeciągać tabele, widoki i procedury składowane, jak również je usuwać.



Rysunek 16.3. Wizualne projektowanie klasy ORM AdresyDataSet

Możemy również stworzyć relację między tabelami. Połączmy tabelę *Osoby* i *Rozmowy* przez utożsamienie ich pól *Id*. W tym celu z menu kontekstowego rozwijanego na rzecz tabeli *Osoby* wybierzmy polecenie *Add/Relation...*. Pojawi się okno dialogowe *Relation*, które konfigurujemy zgodnie ze wzorem widocznym na rysunku 16.4. Po zatwierdzeniu przyciskiem *OK* tabele z klasy *AdresyDataSet* zostaną połączone relacją, a ich reprezentacje na zakładce *AdresyDataSet.xsd* — odpowiednią strzałką.

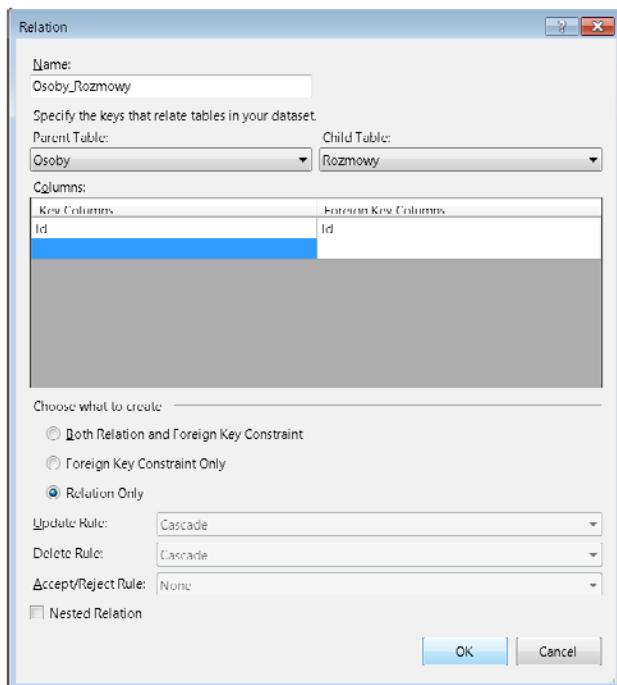
Podgląd danych udostępnianych przez komponent DataSet

Server Explorer łączy się z bazą danych bezpośrednio, a więc nie wymaga pośrednictwa utworzonej w poprzednim ćwiczeniu klasy *AdresyDataSet*. Jednak gdy zechcemy upewnić się, że ta klasa działa prawidłowo, wygodna byłaby możliwość oglądania danych importowanych za pośrednictwem tej klasy. Takie narzędzia również jest dostępne.

1. Z menu *View* wybieramy podmenu *Other Windows*, a w nim *Data Sources*.

Obok podokien *Toolbox* i *Server Explorer* z lewej strony okna Visual Studio pojawi się poznane już w poprzednim rozdziale podokno *Data Sources*. Jednak w odróżnieniu od omawianej w poprzednim rozdziale sytuacji, w której źródła danych korzystały z danych udostępnianych przez klasy LINQ to SQL zamiast z osobnych źródeł dla każdej tabeli, teraz w podoknie widoczne jest tylko jedno źródło — *AdresyDataSet*. Jeżeli je rozwinieśmy, zobaczymy dwie własności tej klasy reprezentujące tabele *Osoby* i *Rozmowy*, a w nich zaimportowane pola.

Rysunek 16.4.
Definiowanie relacji
między tabelami
w klasie AdresyDataSet



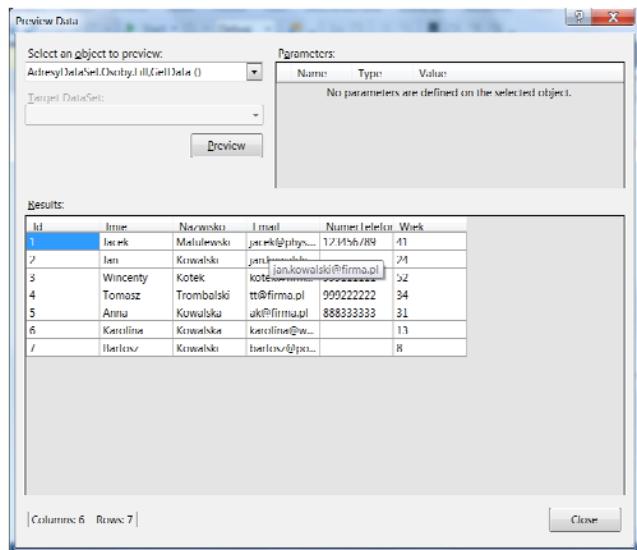
2. Z menu kontekstowego pozycji *AdresyDataSet* wybieramy polecenie *Preview Data*.... Takiego polecenia nie ma w przypadku źródeł opartych na LINQ to SQL.
3. Z rozwijanego menu *Select an object to preview* wybieramy np. tabelę *Osoby* z klasy *AdresyDataSet*, a w niej jedyną pozycję *Fill, GetData()*.
4. Teraz wystarczy kliknąć przycisk *Preview*, aby zobaczyć siatkę z danymi (rysunek 16.5). To są dane, jakie będą udostępniane w aplikacji przez klasę *AdresyDataSet*.
5. Aby zamknąć okno *Preview Data*, klikamy przycisk *Close*.

Prezentacja danych w siatce

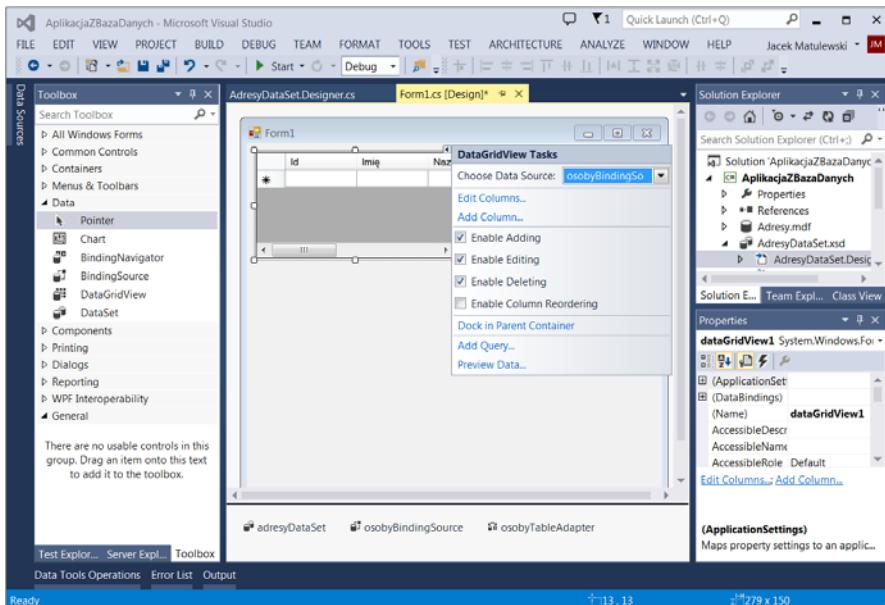
Myszę, że to dobry moment, aby pokazać zaimportowane dane w oknie aplikacji. Začniemy od najprostszej formy prezentacji danych, a mianowicie od siatki (komponent *DataGridView*). Umieścmy wobec tego na formie kontrolkę *DataGridView* i skonfigurujmy ją w taki sposób, aby prezentowała dane z tabeli *Osoby*.

1. Kliknij dwukrotnie plik *Form1.cs* w podoknie *Solution Explorer*. W edytorze pojawi się wówczas widok projektowania formy aplikacji (zakładka *Form1.cs [Design]*).
2. Na podglądzie formy umieszczamy komponent *DataGridView* z zakładki *Data* podokna *Toolbox*.

Rysunek 16.5.
Podgląd danych importowanych przez komponenty typu *DataSet*



3. Poza podglądem umieszczonego komponentu zobaczymy także okienko *DataGridView Tasks* (rysunek 16.6)¹. Zawiera ono niektóre informacje widoczne także w podoknie *Properties* oraz listę dodatkowych czynności, które można wykonać dla zaznaczonego komponentu.



Rysunek 16.6. Wygląd rozwijanej listy po wybraniu pozycji z punktu 4. i utworzeniu związkanych z nią obiektów

¹ Można je w każdej chwili schować lub otworzyć za pomocą niewielkiej strzałki widocznej po zaznaczeniu komponentu na górnjej krawędzi ramki sygnalizującej zaznaczenie komponentu, z jej prawej strony.

4. Z rozwijanej listy *Choose Data Source* w tym okienku wybieramy *Other Data Sources/Project Data Source/AdresyDataSource/Osoby*. O poprawnym połączeniu najlepiej będą świadczyć kolumny widoczne w siatce; powinny one odpowiadać polom tabeli *Osoby*.



Zamiast wykonywać czynności z punktów 2 – 4, możemy po prostu z podokna *Data Sources* przeciągnąć element *Osoby* na podgląd formy. Wówczas przy domyślnych ustawieniach również powstanie siatka prezentująca dane.

5. Aby ułatwić przeglądanie danych, zmieńmy jeszcze własność *Dock* komponentu *dataGridView1* na *Fill*; w ten sposób siatka prezentująca dane będzie zajmowała cały obszar użytkownika w oknie aplikacji.
6. Możemy skompilować i uruchomić aplikację (*F5*), aby zobaczyć dane prezentowane w siatce (rysunek 16.7).

Rysunek 16.7.

Komponent
DataGridView
w działaniu

	Id	Imię	Nazwisko	Email	Numer Telefonu	Wiek
▶	1	Jacek	Matulewski	jacek@phys...	123456789	41
	2	Jan	Kuwalski	jan.kuwalski...		21
	3	Wojciech	Kostecki	wojtek@firma...	999111111	52
	4	Tomasz	Trubalski	tl@firma.pl	999222222	34
	5	Anna	Kowalska	nk@firma.pl	888333333	31
	6	Karolina	Kuwalik	karolina@w...		13
*	7	Bartosz	Kowalski	bartosz@pn...		8

W tej chwili na formie poza dodanym wcześniej komponentem *dataGridView1* obecne są jeszcze trzy inne obiekty. Pierwszym z nich jest *adresyDataSet*, czyli instancja klasy *AdresyDataSet* zawierającej informacje o konfiguracji naszego połączenia z bazą danych *Adresy.mdf*. Kolejnym obiektem jest *osobyBindingSource*, który wskazuje na tabelę *Osoby* i który poznaliśmy już w poprzednim rozdziale, oraz *osobyTableAdapter*. Temu ostatniemu możemy się przyjrzeć dokładniej, wybierając z jego menu kontekstowego polecenie *Edit Quereis in DataSet Designer...* lub klikając dwukrotnie plik *AdresyDataSet.xsd* w *Solution Explorer*.

Zwróciły uwagę, że w odróżnieniu od sytuacji opisanej w poprzednim rozdziale nie musielibyśmy napisać ani jednej linii kodu, aby w siatce widoczne były dane (listing 15.2). Jest tak, ponieważ instrukcję taką dodał już kreator. W metodzie zdarzeniowej związanej ze zdarzeniem *Form1.Load* umieścił polecenie widoczne na listingu 16.1. Usunięcie tej linii spowoduje, że po uruchomieniu aplikacji dane w siatce już się nie pojawią, choć nagłówki kolumn pozostaną.

Listing 16.1. Konstruktor klasy okna z automatycznie dodanym poleceniem wczytania danych z tabeli

```
private void Form1_Load(object sender, EventArgs e)
{
    this.osobyTableAdapter.Fill(this.adresyDataSet.Osoby);
}
```

Zapisywanie zmodyfikowanych danych

Siatka umożliwia nie tylko przeglądanie, ale również edycję i wprowadzanie nowych danych. Zmiany nie są jednak automatycznie zapisywane w tabeli bazy danych — to wymaga wywołania przez programistę metody `Update` obiektu `osobyTableAdapter`. Aktualizację danych z bazy danych przeprowadzimy w momencie zamknięcia okna aplikacji. Wcześniej poprosimy jednak użytkownika o potwierdzenie zamiaru zapisania zmienionych danych.

1. Instalujemy „system” wykrywania zmian w tabeli:

- a) Klikamy klawisz *F7*, aby przejść do edycji kodu pliku *Form1.cs*. Następnie w klasie *Form1* definiujemy pole typu `bool` o nazwie `daneZmienione`:
- ```
private bool daneZmienione = false;
```
- b) Na końcu metody *Form1\_Load*, tj. po wczytaniu danych do komponentu `adresyDataSet.Osoby`, dodajemy polecenie opuszczające flagę (wyróżnione w listingu 16.2).

**Listing 16.2.** Opuszczanie flagi

```
private void Form1_Load(object sender, EventArgs e)
{
 this.osobyTableAdapter.Fill(this.adresyDataSet.Osoby);
 daneZmienione = false;
}
```

- c) Przechodzimy do widoku projektowania, zaznaczamy komponent `dataGridView1` i za pomocą podokna *Properties* tworzymy metodę zdarzeniową związaną z jego zdarzeniem `CellValueChanged` z poleceniem unoszącym flagę (listing 16.3).

**Listing 16.3.** Podniesienie flagi

```
private void dataGridView1_CellValueChanged(object sender,
DataGridEventArgs e)
{
 daneZmienione = true;
}
```

2. W widoku projektowania formy zaznaczamy okno *Form1*, klikając jego pasek tytułu.
3. Za pomocą okna właściwości tworzymy metodę zdarzeniową do zdarzenia `FormClosing` i umieszczamy w niej polecenie zapisujące zmiany w tabeli (listing 16.4).

**Listing 16.4.** Polecenie aktualizujące zawartość bazy danych przed zamknięciem aplikacji

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
 if (!daneZmienione) return;
 switch(MessageBox.Show("Czy zapisać zmiany do bazy danych?",
 this.Text, MessageBoxButtons.YesNoCancel))
```

```
{
 case DialogResult.Cancel:
 e.Cancel = true;
 break;
 case DialogResult.Yes:
 try
 {
 this.Validate();
 this.osobyBindingSource.EndEdit();
 this.osobyTableAdapter.Update(adresyDataSet.Osoby);
 //MessageBox.Show("Dane zapisane do bazy");
 }
 catch (Exception exc)
 {
 MessageBox.Show("Zapisanie danych nie powiodło się (" + exc.Message + ")");
 }
 break;
 case DialogResult.No:
 break;
}
}
}
```

4. Zaznaczamy plik bazy danych *Adresy.mdf* w podoknie *Solution Explorer*. W podoknie *Properties* zmieniamy związaną z nim opcję *Copy to Output Directory* na *Copy if newer*.

Najpierw wy tłumaczę się ze zmiany wprowadzonej w ostatnim punkcie powyższego ćwiczenia. Jeżeli pozostawimy domyślne ustawienie projektu, w którym opcja *Copy to Output Directory* ustawiona jest na *Copy always*, to przy każdej komplikacji oryginał bazy danych z katalogu projektu (tj. katalogu, w którym są także pliki źródłowe .cs) będzie kopowany do katalogu *bin\Debug* lub *bin\Release*. W efekcie zmiany wprowadzone do tej kopii bazy będą po każdej komplikacji zamazywane, bo przywracana będzie stara wersja pliku bazy danych.

Polecenia wykonywane w metodzie *Form1\_Closing* zapewnią możliwość zapisania zmian w danych przed zamknięciem formy (jeżeli użytkownik kliknie przycisk *Tak* w prostym oknie dialogowym), a dodatkowo umożliwią także anulowanie zamknięcia okna (jeżeli użytkownik kliknie *Anuluj*). Dzięki fladze *daneZmienione* pytanie o zapisanie danych pojawi się jedynie wtedy, gdy dane rzeczywiście zostaną zmodyfikowane.

Moment zapisania zmian należy wybrać uważnie. Można tak jak w powyższym przykładzie zapisywać zmiany dopiero wtedy, gdy mamy zamiar zakończyć pracę z programem. Jednak wówczas podejmujemy ryzyko utraty zmian w razie awarii systemu (wy starczy niespodziewane zawieszenie systemu operacyjnego). Można również zapisywać zmiany po każdej akceptacji edytowanej komórki lub wiersza. Inne możliwe rozwiązań to autozapis co określony czas (należy wówczas wykorzystać komponent *Timer*).



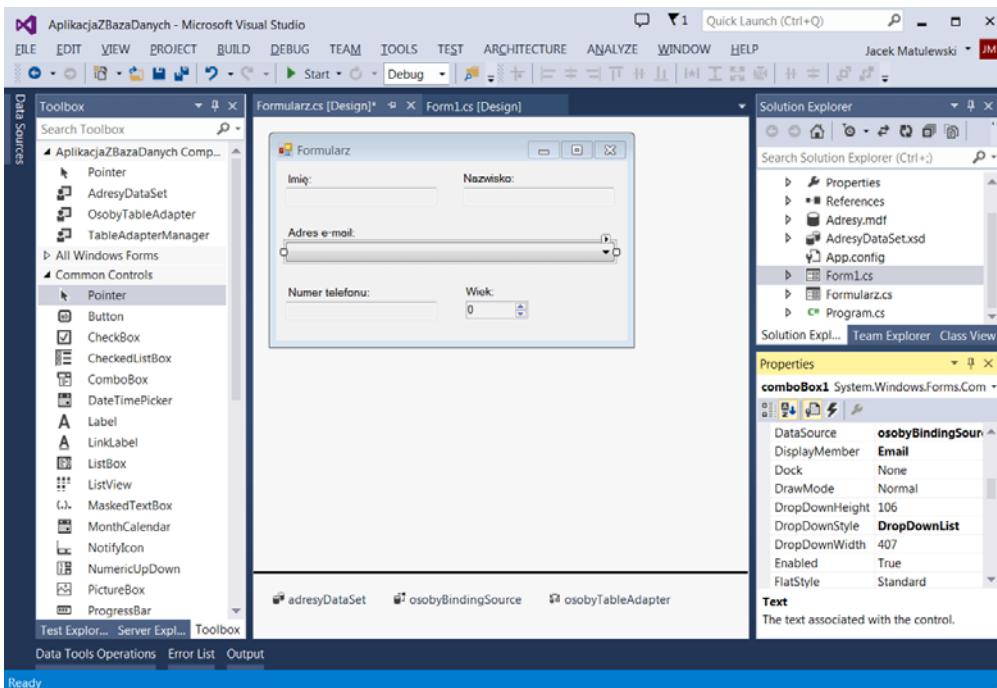
Wskazówka

Należy zwrócić uwagę na to, że zarówno klasa *AdresyDataSet*, jak i *OsobyTableAdapter* są komponentami. Visual C# udostępnia je w zakładce *AplikacjaZBaza Danych Components* widocznej w widoku projektowania na szczytce listy komponentów *Toolbox*. Można ich użyć np. na innych formach bieżącego projektu. Czasem, aby ta zakładka pojawiła się w podoknie *Toolbox*, należy zamknąć i ponownie uruchomić Visual Studio, a następnie wczytać projekt.

## Prezentacja danych w formularzu

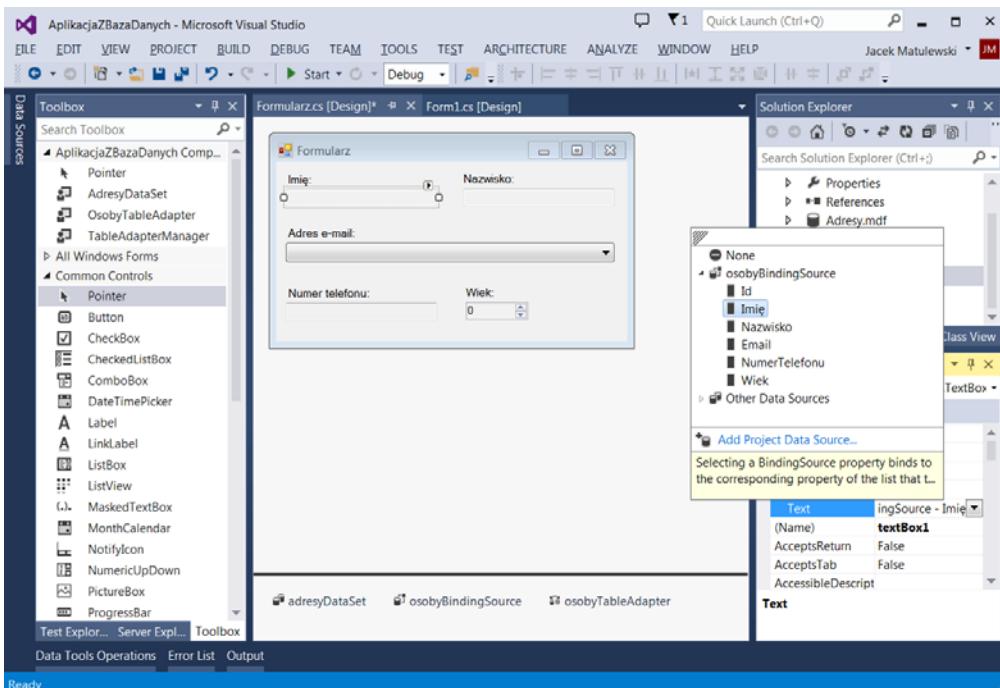
Tabela nie jest zwykle najbardziej optymalnym sposobem przeglądania danych, mimo że jest to jedyny sposób, za pomocą którego możemy zobaczyć jednocześnie wiele rekordów. Zazwyczaj wygodniejsze jest korzystanie z różnego typu formularzy. Przygotujemy zatem formularz. Ponieważ wiemy już, jak szybko przygotować formularz za pomocą podokna *Data Source* — wystarczy przeciągać na formę widoczne w nim elementy — tu chyba tylko z przekory opiszę, jak zrobić to „ręcznie”. W formularzu do wyboru rekordu użyjemy rozwijanej listy zawierającej adresy e-mail.

1. Formularz umieścimy w nowym oknie. Aby je utworzyć:
  - a) Z menu *Project* wybieramy pozycję *Add Windows Form*....
  - b) Pojawi się okno *Add New Item* — *AplikacjaZBazaDanych*. Zaznaczmy ikonę *Windows Form*.
  - c) W polu *Name* wpisujemy nazwę *Formularz*.
  - d) Wybór potwierdzamy, klikając *Add*.
2. Za pomocą okna właściwości konfigurujemy nowe okno w następujący sposób:
  - a) właściwość *FormBorderStyle* zmieniamy na *FixedSingle*;
  - b) *imizeBox* ustawiamy na *False*;
  - c) *ShowInTaskbar* zmieniamy na *False*.
3. Na podglądzie okna umieszczamy rozwijaną listę *ComboBox*, trzy pola edycyjne *TextBox* i komponent *NumericUpDown* (wszystkie z zakładki *Common Controls* podokna *Toolbox*).
4. Formę uzupełniamy etykietami *Label* umieszczonymi zgodnie ze wzorem zaprezentowanym na rysunku 16.8.
5. Właściwość *Maximum* komponentów *NumericUpDown* zwiększamy do np. 1000.
6. Właściwość *ReadOnly* pól edycyjnych i komponentu *NumericUpDown* ustawiamy na *true*.
7. Zaznaczamy rozwijaną listę. Zmieniamy jej właściwość *DropDownStyle* na *DropDownList*, przez co zablokujemy możliwość edycji i w tym komponencie.
8. Aby rozwijana lista zawierała adresy e-mail z pola *Email* tabeli *Osoby*, należy:
  - a) W podglądzie formy rozwiniąć okienko *ComboBox Tasks* i zaznaczyć pozycję *Use data bound items*.
  - b) Następnie z rozwijanej listy *DataSource* wybrać *Other Data Sources*, *Project Data Sources*, *AdresyDataSet* i wreszcie tabelę *Osoby*. Podobnie jak w przypadku siatki powstanie wówczas obiekt *osobyBindingSource*, *adresyDataSet* oraz *osobyTableAdapter*.
  - c) Z rozwijanej listy *Value Member* wybieramy pole *Id*, a z rozwijanej listy *Display Member* — pole *Email*.



Rysunek 16.8. Propozycja prostego formularza pozwalającego na przeglądanie danych z tabeli Osoby

9. Niestety pozostałych kontrolek, w tym pól edycyjnych TextBox i kontrolki NumericUpDown, nie można związać z danymi z podrócznej listy zadań. Pozwala na to jednak podokno własności.
  - a) Zaznaczmy na początek kontrolkę textBox1 (z etykietą *Imię*).
  - b) W podoknie *Properties* odnajdujemy grupę (*DataBindings*) i jej podelement *Text*.
  - c) Rozwijamy listę obok tej własności, a także widoczny w niej element *osobyBindingSource*.
  - d) Wreszcie wybieramy pozycję *Imię* (rysunek 16.9).
10. W analogiczny sposób wiążemy pozostałe pola edycyjne.
11. Analogicznie wiążemy także pole *Wiek* tabeli *Osoby* z własnością *DataBindings, Value* komponentu numericUpDown1.
12. Aby dodane okno formularza było widoczne po uruchomieniu aplikacji, należy stworzyć instancję klasy *Formularz* i wywołać jej metodę *Show*. Jednak aby nowe okno było widoczne na pierwszym planie, należy wywołać metodę *Show* drugiego okna już po pojawienniu się na ekranie pierwszego okna.
  - a) Przechodzimy do widoku projektowania formy *Form1* (plik *Form1.cs*).
  - b) Za pomocą podokna *Properties* tworzymy metodę zdarzeniową do zdarzenia *Shown* i umieszczać w niej polecenie widoczne na listingu 16.5.



Rysunek 16.9. Edytor wiązania własności z rekordami tabeli

Listing 16.5. Polecenie tworzące i pokazujące drugie okno aplikacji

```
namespace AplikacjaZBazaDanych
{
 public partial class Form1 : Form
 {
 public Form1()
 {
 InitializeComponent();
 }

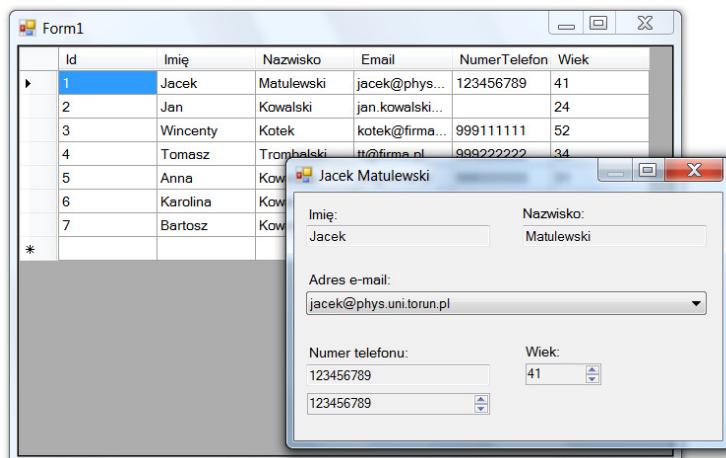
 ...

 private void Form1_Shown(object sender, EventArgs e)
 {
 new Formularz().Show();
 }
 }
}
```

Po uruchomieniu aplikacji zobaczymy dwie formy. Nas będzie teraz interesować oczywiście forma z formularzem (rysunek 16.10). Spróbujmy rozwinąć listę *Adres e-mail*. Zauważmy, że jeżeli wybierzemy z listy nowy adres, to zawartość pozostałych pól zostanie zaktualizowana w taki sposób, iż widoczne w nich będą zawartości komórek z wybranego w rozwijanej liście rekordu tabeli. Odpowiedzialne za to jest wspólne wiązanie do obiektu *osobyBindingSource*. W tym komponencie określony jest tzw. aktywny rekord, tj. wiersz, który jest aktualnie udostępniany przez tabelę (w przypadku

**Rysunek 16.10.**

Przeglądanie danych w formularzu ułatwia skupienie się na wybranych polach tabeli



DataGridView i ComboBox chodzi o wiersz zaznaczony). Wybór aktywnego rekordu w oknie formularza i w oknie z siatką są niezależne, ponieważ oba okna korzystają z połączenia udostępnianego przez własne instancje komponentu BindingSource.

## Sortowanie i filtrowanie

Jeżeli chcemy posortować dane widoczne w siarce lub formularzu, wystarczy użyć właściwości Sort obiektów osobyBindingSource (zobacz poprzedni rozdział). Jeżeli tej właściwości przypiszemy łańcuch „Nazwisko”, dane zostaną posortowane zgodnie z alfabetyczną kolejnością nazwisk.

W odróżnieniu od technologii LINQ to SQL w przypadku DataSet działa także filtrowanie danych. Wystarczy właściwości osobyBindingSource.Filter przypisać łańcuch definijący filtr identyczny z tym, jaki stosuje się w zapytaniu SQL po słowie kluczowym WHERE, np. Wiek<50. Wówczas dane prezentowane na formie ograniczą się do rekordów, w których pole Wiek ma wartość mniejszą niż 50. Pozostałe osoby nie zostaną udostępnione przez źródło danych (a raczej reprezentanta tabeli na formie, a więc obiekt osobyBindingSource) i w konsekwencji nie będą widoczne w kontrolach.

## Odczytywanie z poziomu kodu wartości przechowywanych w komórkach tabeli

Definicja tabeli w naszej bazie danych dopuszcza, żeby pole *NumerTelefonu* w rekordzie było puste (zobacz rysunek 13.3 z rozdziału 13.). Jeżeli puste byłoby pole zawierające tekst, to w odpowiadającej mu kontrolce TextBox związanego z tym polem pojawiłby

się po prostu pusty tekst. Jednak komponent NumericUpDown nie radzi sobie w takiej sytuacji. Aby się o tym przekonać, dodajmy do formularza jeszcze jeden komponent NumericUpDown i zwiążmy jego własność Value z polem *NumerTelefonu*. Pamiętajmy tylko o ustawieniu własności Maximum na tyle dużej, aby była większa od numerów telefonów. Po uruchomieniu aplikacji zobaczymy, że wartość pokazywana w tej kontrolce nie jest aktualizowana, gdy wybierany jest rekord, w którym brakuje danych z pola *NumerTelefonu*.

Można tę kontrolkę wspomóc, sprawdzając wartość zapisaną w komórce, a jeżeli jest pusta, wymusić pokazanie zera lub np. ukrycie komponentu. To wiąże się z ważnym zagadnieniem: jak odczytać z poziomu kodu C# wartości bieżącego rekordu tabeli. Umiejętność ta może być przydatna nie tylko do wspomagania kontrolki NumericUpDown. W ten sposób można chociażby wyświetlać dane obliczone na podstawie wartości z kilku pól tabeli lub w inny sposób od nich zależne.

Na szczęście odczytywanie wartości z dowolnej komórki tabeli jest proste. Można do tego wykorzystać instancję komponentu DataSet, która reprezentuje tabelę w aplikacji, i z niej odczytać potrzebną wartość:

```
int ir=1; //indeks odczytywanego rekordu
string personalia=adresyDataSet.Osoby[ir].Imię + " " +
adresyDataSet.Osoby[ir].Nazwisko;
```

Należy oczywiście pamiętać o tym, że dane są obecne w adresyDataSet dopiero po wywołaniu metody osobyTableAdapter.Fill wstawionej do metody Form1\_Load. Korzystanie z adresyDataSet ma jednak tę podstawową wadę, że dane tam przechowywane nie są czułe na sortowanie i filtrowanie, które realizowane jest dopiero na poziomie osobyBindingSource, a tym bardziej nie potrafią dostarczyć informacji o tym, który rekord jest rekordem bieżącym (tj. widocznym w formularzu lub zaznaczonym w siatce DataGridView). Z tych względów często wygodniej jest korzystać z tego samego źródła danych, z którego korzystają kontrolki udostępnione użytkownikowi aplikacji, a więc w naszym przypadku z osobyBindingSource. Dzięki temu znajdziemy się w tym samym kontekście wiązania.

A zatem jeżeli chcemy odczytać bieżący rekord, powinniśmy odczytać własność osobyBindingSource.Current, a następnie zrzutować go na typ OsobyRow. Jest to klasa zagnieżdżona, zdefiniowana w klasie AdresyDataSet. Rzutowanie nie jest jednak proste, bo najpierw należy zrzutować odczytany rekord na typ DataRowView, a dopiero z niego odczytać zawartość rekordu:

```
AdresyDataSet.OsobyRow rekord = (osobyBindingSource.Current as DataRowView).Row
as AdresyDataSet.OsobyRow;
```

W ten sposób uzyskamy dostęp do bieżącego rekordu prezentowanego przez obiekt typu OsobyRow, który udostępnia własności o nazwach odpowiadających nazwom pól tabeli:

```
string personalia = rekord.Imię + " " + rekord.Nazwisko;
```

W ten sposób możemy również rozwiązać problem kontrolki NumericUpDown. Próba odczytania wartości, która w tabeli zapisana jest jako NULL, powoduje zgłoszenie wyjątku. Wystarczy zatem w konstrukcji try..catch sprawdzić zawartość pola *NumerTelefonu*

i jeżeli nie da się go odczytać, wymusić pokazanie zera w kontrolkach NumericUpDown (listing 16.6). Przy takim podejściu, w którym *de facto* kod C# przejmuje odpowiedzialność za wypełnienie kontrolki danymi, lepiej jest usunąć wiązanie ze źródłem danych — inaczej zachowanie kontrolki może być trudne do przewidzenia.

**Listing 16.6.** Osobna konstrukcja try..catch dla każdego z pól numerycznych

```
private void osobyBindingSource_CurrentChanged(object sender, EventArgs e)
{
 AdresyDataSet.OsobyRow rekord =
 (osobyBindingSource.Current as DataRowView).Row as AdresyDataSet.OsobyRow;
 Text = rekord.Imię + " " + rekord.Nazwisko;
 try
 {
 numericUpDown2.Value = rekord.NumerTelefonu;
 }
 catch
 {
 textBox3.Text = "Brak";
 numericUpDown2.Value = 0;
 }
}
```

Możliwe są także scenariusze, w których wygodne byłoby odwzorowanie w odczytanej zmiennej faktu, że w bazie danych pole w rekordzie nie ma przypisanej wartości. Idealnie nadaje się do tego typ *Nullable* (zobacz rozdział 3.), który np. w klasach enklji LINQ to SQL automatycznie stosowany jest w przypadku pól oznaczonych jako zezwalające na pustą zawartość (*Allow Null*). W przypadku DataSet, które przecież powstało, nim do platformy .NET dodany został typ *Nullable*, sprawa wymaga nieco większej ilości kodu (listing 16.7).

**Listing 16.7.** Odwzorowanie z użyciem typu *Nullable<int>*, czyli *int?*

```
private void osobyBindingSource_CurrentChanged(object sender, EventArgs e)
{
 AdresyDataSet.OsobyRow rekord =
 (osobyBindingSource.Current as DataRowView).Row as AdresyDataSet.OsobyRow;
 Text = rekord.Imię + " " + rekord.Nazwisko;

 int? numerTelefonu = null;
 if (!rekord.IsNumerTelefonuNull()) numerTelefonu = rekord.NumerTelefonu;

 if (numerTelefonu.HasValue)
 {
 numericUpDown2.Value = numerTelefonu.Value;
 }
 else
 {
 textBox3.Text = "Brak";
 numericUpDown2.Value = 0;
 }
}
```

## Zapytania LINQ do danych z DataSet

Dysponując obiektem DataSet wypełnionym danymi, możemy pobierać z niego dane, także korzystając z zapytań LINQ. Technologia ta nazywa się LINQ to DataSet. Listing 16.8 pokazuje prosty przykład jej użycia, w którym pobierana i prezentowana jest lista osób pełnoletnich z tabeli *Osoby*. Ostatnia instrukcja tej metody pokazuje również, że na rzecz tabeli można wywoływać rozszerzenia LINQ, w tym przypadku rozszerzenie *First*.

**Listing 16.8.** Przykłady użycia LINQ to DataSet

```
private void button1_Click(object sender, EventArgs e)
{
 var wynikZapytania = from osoba in adresyDataSet.Osoby
 where osoba.Wiek > 18
 orderby osoba.Wiek
 select osoba.Imię + " " + osoba.Nazwisko + " (" +
 osoba.Wiek.ToString() + ")";
 string s = "Osoby pełnoletnie:\n";
 foreach (string element in wynikZapytania) s += element + "\n";
 MessageBox.Show(s);

 var pierwszyPełnoletni = adresyDataSet.Osoby.First(o => o.Wiek > 18);
}
```

Danych pobranych zapytaniem LINQ można także użyć jako źródła danych np. dla kontrolek, w szczególności kontrolki *DataGridView*:

```
var wynikZapytania = from osoba in adresyDataSet.Osoby
 where osoba.Wiek > 18
 orderby osoba.Wiek
 select new
 {
 osoba.Imię,
 osoba.Nazwisko,
 osoba.Wiek
 }
dataGridView1.DataSource = wynikZapytania.ToList();
```

# Rozdział 17.

# Entity Framework

Entity Framework (EF) jest najnowszym mechanizmem odwzorowania obiektowo-relacyjnego w platformie .NET. Jest zdecydowanie najbardziej elastyczny, ogólny, ale jednocześnie wydaje się najwolniejszy<sup>1</sup>. Do Visual Studio 2013 dołączona jest wersja EF 6.0.

W przypadku EF możliwych jest kilka scenariuszy, zgodnie z którymi możemy przygotować aplikację bazodanową. Wybór jednego z nich zależy od tego, czy preferujemy modelowanie klas ORM za pomocą myszy, czy z poziomu kodu (samodzielne definiowanie klas encji) oraz czy modelowana baza danych już istnieje, czy też chcemy ją właśnie utworzyć na podstawie modelu<sup>2</sup>. Wszystkie możliwości przedstawia tabela 17.1. W poprzednich rozdziałach rozwijaliśmy projekt z gotową bazą danych; tak będzie także w przypadku EF. W efekcie ograniczymy się tylko do rozwiązań z pierwszego wiersza tabeli.

**Tabela 17.1.** Scenariusze tworzenia projektu korzystającego z Entity Framework

|                               | Narzędzia projektowania ORM                                                                                                         | Definiowanie klasy encji                                                                                                                           |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| Istniejąca baza danych        | <i>Database first</i><br>automatyczne odwzorowanie struktury bazy danych w klasach ORM                                              | <i>Code first, existing database</i><br>wspariane przez narzędzia Visual Studio tworzenie klas ORM z poziomu kodu                                  |
| Baza tworzona przez aplikację | <i>Model first</i><br>definiowanie klas ORM (modelu bazy danych) i po uruchomieniu aplikacji tworzenie zgodnej z tym projektem bazy | <i>Code first, new database</i><br>definiowanie klas modelu ORM „ręcznie” i używanie narzędzi migracji do utworzenia odpowiadającej im bazy danych |

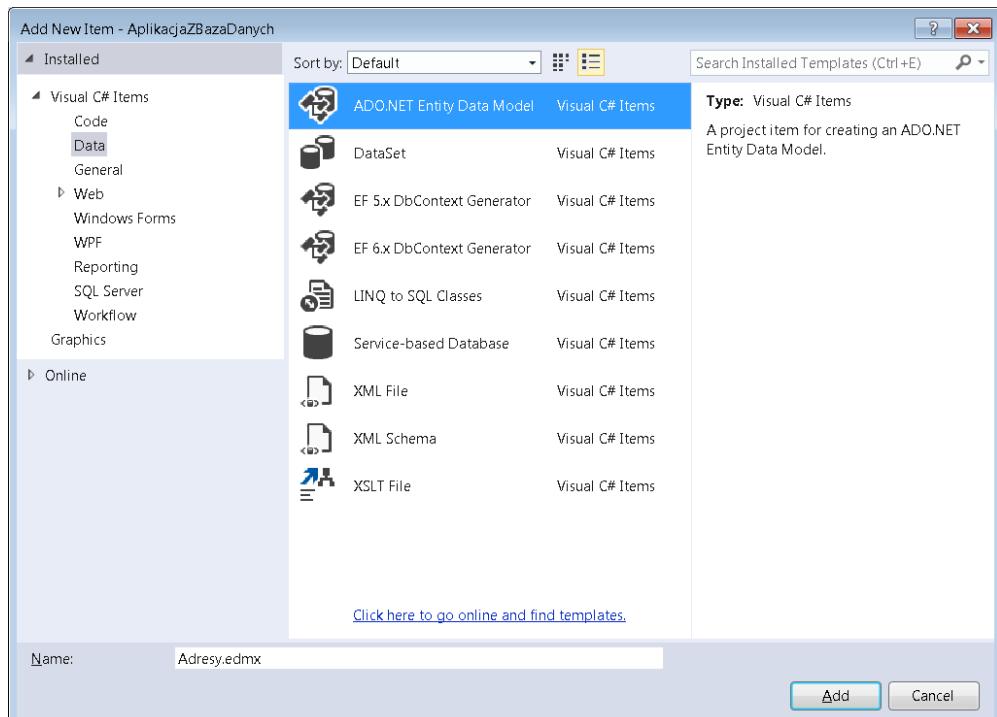
<sup>1</sup> To stwierdzenie oparte jest na moich wrażeniach z jego używania. Nie wykonywałem ani nie widziałem w Internecie żadnych miarodajnych testów. Te, które można tam znaleźć, odnoszą się do pierwszych wersji EF.

<sup>2</sup> Zobacz omówienie tego tematu w filmie ze strony <http://msdn.microsoft.com/en-US/data/jj590134> oraz umieszczone na tej stronie linki do stron omawiających poszczególne scenariusze.

# Tworzenie modelu danych EDM dla istniejącej bazy danych

Kolejny raz wracamy do przygotowanego w rozdziale 13. projektu aplikacji Windows Forms, do której dodany jest plik *Adresy.mdf* bazy danych SQL Server. Tym razem dla tej bazy danych stworzymy model danych EF (ang. *Entity Data Model*, w skrócie EDM).

1. Z menu *Project* wybieramy *Add New Item....*
2. Następnie w otwartym oknie dialogowym (rysunek 17.1) z zakładki *Data* wybieramy *ADO.NET Entity Data Model*.



Rysunek 17.1. Tworzenie modelu danych EF

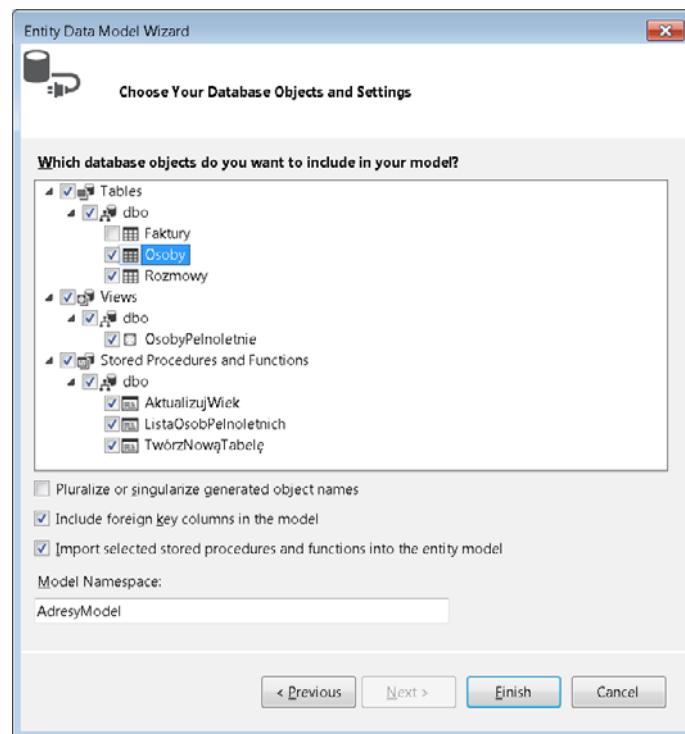
3. W polu *Name* wpisujemy nazwę pliku *Adresy.edmx* i klikamy *Add*.
4. Uruchomiony zostanie kreator EDM. W pierwszym kroku możemy wybrać pusty model, co należałoby zrobić w przypadku projektu realizowanego w scenariuszu *Model first*, lub model tworzony na bazie istniejącej bazy danych (pozycja *Generate from database*). Wybieramy tę drugą możliwość i klikamy *Next*.
5. W drugim kroku musimy stworzyć połączenie z bazą danych. W rozwijanej liście wybrana jest już jedyna baza danych obecna w projekcie, czyli *Adresy.mdf*. W tym momencie można również wskazać dowolną inną bazę.

danych, nie tylko SQL Server. W dolnej części okna kreatora widoczny jest łańcuch połączenia. Zwróćmy uwagę, że wygląda nieco inaczej niż ten, do którego zdążyliśmy się przyzwyczaić w poprzednich rozdziałach.

6. Jeszcze niżej widoczne jest zaznaczone pole opcji i pole tekstowe z nazwą, pod którą zapisane będą ustawienia połączenia. Nazwa ta, a konkretnie AdresyEntities, będzie też nazwą klasy kontekstu EF — odpowiednika DataSet i DataContext z poznanych wcześniejszej technologii ORM.
7. W trzecim kroku zostaniemy zapytani o wersję EF, której chcemy użyć. W Visual Studio 2013 możemy wybrać między wersjami 5.0 i 6.0. Możliwa jest także instalacja innych wersji. Ja wskazałem najnowsze Entity Framework 6.0 i kliknąłem *Next*.
8. W ostatnim kroku kreatora możemy wskazać elementy z bazy danych, które mają być udostępniane przez EF (rysunek 17.2). Wygląda to bardzo podobnie jak przy tworzeniu klas LINQ to SQL lub klasy DataSet. Udostępnijmy tabele *Osoby* i *Rozmowy* oraz widok i procedury składowane.

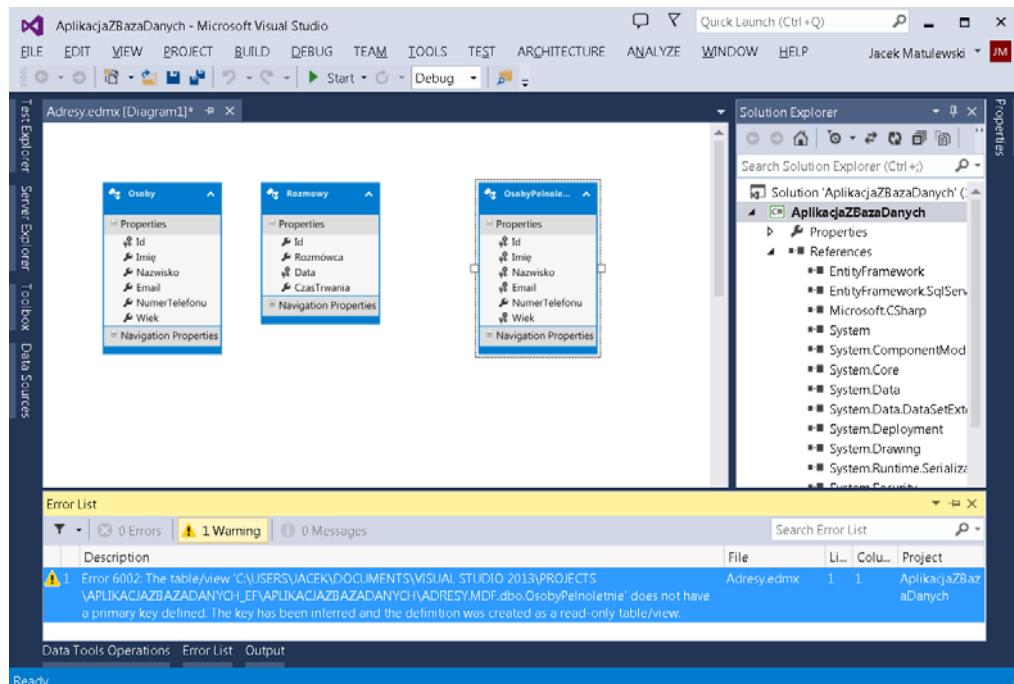
**Rysunek 17.2.**

Elementy  
bazy danych  
udostępniane  
w modelu  
danych EDM



9. Bez zmian pozostawmy nazwę przestrzeni nazw modelu *AdresyModel* i kliknijmy *Finish*. Dzięki niezaznaczeniu opcji *Pluralize and singularize generated object names* kreator nie będzie próbował tworzyć liczb mnogich dla nazw z naszej tabeli. W zamian zrobimy to za chwilę sami.

Do projektu zostanie dodana grupa plików *Adresy.edmx*. W trakcie jej tworzenia pojawi się ostrzeżenie, że widok *OsobyPelnoletnie* nie ma zdefiniowanego klucza głównego, w związku z czym będzie on udostępniany w trybie tylko do odczytu (zobacz ostrzeżenie widoczne na rysunku 17.3). W trakcie tworzenia plików pojawi się także ostrzeżenie, że uruchomiony zostanie skrypt, który może nawet uszkodzić komputer! Mojemu komputerowi nic się nie stało, więc żeby ciągle nie odpowiadać na to pytanie, warto zaznaczyć opcję, dzięki której komunikat z tym ostrzeżeniem nie będzie pojawiał się ponownie. Po zakończeniu działania kreatora model zostanie zaprezentowany na zakładce *Adresy.edmx [Diagram1]*, zawierającej obie udostępniane tabele i widok (rysunek 17.3). Przypomina to jako żywo zakładki z klasami ORM, jakie poznaliśmy w poprzednich rozdziałach. Nie powinniśmy wobec tego poczuć się zbytnio zagubieni.



Rysunek 17.3. Model danych EDM i ostrzeżenie dotyczące braku klucza głównego w widoku

Po utworzeniu modelu EDM do katalogu projektu dodany zostanie podkatalog *packages/EntityFramework.6.0.0*, w którym umieszczone zostaną biblioteki EF. To zapewnia łatwą przenośność projektu. Biblioteki te, a konkretnie *EntityFramework.dll* i *EntityFramework.SqlServer.dll* wraz z towarzyszącymi im plikami XML, zostaną również skopiowane do katalogu docelowego skompilowanej aplikacji.

Warto również zwrócić uwagę na pliki zawierające klasy encji zbudowane z domyślnie implementowanych własności (zobacz rozdział 3.). Dostępne są w podoknie *Solution Explorer*, w gałęzi *Adresy.edmx\Adresy.tt*. W naszym przypadku będą to trzy pliki: *Osoby.cs* (listing 17.1), *Rozmowy.cs* i *OsobyPelnoletnie.cs*.

**Listing 17.1.** Klasa encji EDM tabeli Osoby. Komentarz w nagłówku radzi nie edytować tego pliku

```
//-----
//<auto-generated>
// This code was generated from a template.
//
// Manual changes to this file may cause unexpected behavior in your application.
// Manual changes to this file will be overwritten if the code is regenerated.
//</auto-generated>
//-----

namespace AplikacjaZBazaDanych
{
 using System;
 using System.Collections.Generic;

 public partial class Osoby
 {
 public int Id { get; set; }
 public string Imię { get; set; }
 public string Nazwisko { get; set; }
 public string Email { get; set; }
 public Nullable<int> NumerTelefonu { get; set; }
 public int Wiek { get; set; }
 }
}
```

## Użycie klasy kontekstu z modelu danych EF

Instancja klasy `AdresyEntities`, która należy do zbioru klas ORM utworzonych przez kreator modelu danych EDM, udostępnia własności reprezentujące tabele i widoki oraz metody odpowiadające procedurom składowanym obecnym w bazie danych. Konieczne będzie wobec tego utworzenie instancji tej klasy.

1. Jednak zanim zaczniemy tej klasy używać, zmieńmy nazwy udostępnionych w niej tabel, a jednocześnie odpowiadających im klas encji. Na zakładce *Adresy.edmx [Diagram1]* zaznaczmy kolejno tabele *Osoby* i *Rozmowy* i zmieńmy ich nazwy na *Osoba* i *Rozmowa*. Tym samym zmienimy także nazwę pliku z listingu 17.1 na *Osoba.cs* i konsekwentnie zmienimy także nazwę zdefiniowanej w nim klasy na *Osoba*.
2. Wymuśmy skompilowanie projektu (*Ctrl+Shift+B* lub *F6*).
3. Teraz kliknijmy dwukrotnie pozycję *Form1.cs* w podoknie *Solution Explorer*, aby przejść do widoku projektowania formy. Umieśćmy na niej przycisk i utwórzmy jej domyślną metodę zdarzeniową.
4. Metodę tę wykorzystamy jako „piaskownicę”, w której przećwiczymy używanie klasy kontekstu `AdresyEntities`. Zaczniemy od elementarza, a więc odczytania i prezentacji wszystkich rekordów z tabeli *Osoby*. Po uruchomieniu kodu

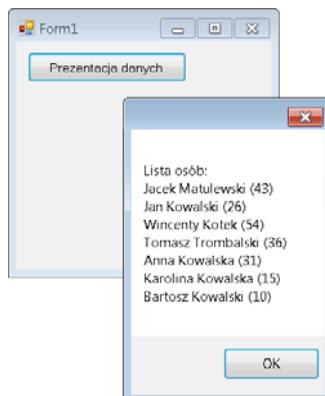
z listingu 17.2 zobaczymy okno dialogowe jak na rysunku 17.4. Pojawi się po wyraźnie dłuższym czasie, niż było to w przypadku LINQ to SQL!

### **Listing 17.2.** Znowu MessageBox

```
private void button1_Click(object sender, EventArgs e)
{
 using(AdresyEntities bazaDanychAdresy = new AdresyEntities())
 {
 string s = "Lista osób:\n";
 foreach (Osoba o in bazaDanychAdresy.Osoby)
 s += o.Imię + " " + o.Nazwisko + " (" + o.Wiek + ")\n";
 MessageBox.Show(s);
 }
}
```

**Rysunek 17.4.**

Dane odczytane  
z tabeli *Osoby*  
udostępnianej  
przez EDM



5. A co z modyfikowaniem danych? Okazuje się, że jest to równie proste jak w LINQ to SQL. Listing 17.3 pokazuje zmodyfikowany kod metody zdarzeniowej przycisku, w którym po prezentacji danych dodawany jest do tabeli *Osoby* nowy rekord, a potem całość jest jeszcze raz pokazywana.

### **Listing 17.3.** Dodawanie nowego rekordu z poziomu kodu

```
private void button1_Click(object sender, EventArgs e)
{
 using(AdresyEntities bazaDanychAdresy = new AdresyEntities())
 {
 //prezentacja oryginalnych danych
 string s = "Lista osób:\n";
 foreach (Osoba o in bazaDanychAdresy.Osoby)
 s += o.Imię + " " + o.Nazwisko + " (" + o.Wiek + ")\n";
 MessageBox.Show(s);

 //tworzenie nowego rekordu
 int noweId = bazaDanychAdresy.Osoby.Max(o => o.Id) + 1;
 Osoba nowaOsoba = new Osoba()
 {
 Id = noweId,
 Imię = "Antoni",
 }
 }
}
```

```
Nazwisko = "Gburek",
Wiek = 45,
Email = "ag@firma.pl",
NumerTelefonu = 123456789
};

//dodanie rekordu do bazy i zapisanie zmian
bazaDanychAdresy.Osoby.Add(nowaOsoba);
bazaDanychAdresy.SaveChanges();

//prezentacja zmodyfikowanych danych
s = "Lista osób:\n";
foreach (Osoba o in bazaDanychAdresy.Osoby)
 s += o.Imię + " " + o.Nazwisko + "(" + o.Wiek + ")\n";
MessageBox.Show(s);
}

}
```

Jak widać, dodawanie nowych rekordów do bazy danych *Osoby* polega na dodaniu kolejnych elementów do kolekcji *bazaDanychAdresy.Osoby*. Trzeba tylko pamiętać o przekazaniu zmian z powrotem do bazy danych, tj. o wywołaniu metody *bazaDanychAdresy.SaveChanges()*.

## LINQ to Entities

W kodzie z listingu 17.3 do ustalenia identyfikatora nowego rekordu użyliśmy rozszerzenia *Max* bezpośrednio na właściwości *Osoby* instancji klasy *AdresyEntities*. Jest to obiekt typu *DbSet<Osoba>* zdefiniowanego w przestrzeni nazw *System.Entity.Data*. Rozszerzenie *Max*, jak pamiętamy z rozdziału 11., to element technologii LINQ. Można wobec tego podejrzewać, że obiekty typu *DbSet<>* mogą być źródłami danych także dla zapytań LINQ. I tak jest w rzeczywistości! Umożliwia to LINQ to Entities.

Aby się o tym przekonać, przygotujmy drugą metodę związaną z przyciskiem; zaprezentujmy w niej dane pobrane za pomocą zapytania LINQ. Tradycyjnie będą to osoby pełnoletnie posortowane alfabetycznie wg nazwisk (listing 17.4).

**Listing 17.4.** Przykład użycia LINQ to Entities

```
private void button2_Click(object sender, EventArgs e)
{
 using (AdresyEntities bazaDanychAdresy = new AdresyEntities())
 {
 //zapytanie LINQ to Entities
 var osobyPelnoletnie = from o in bazaDanychAdresy.Osoby
 where o.Wiek >= 18
 orderby o.Nazwisko
 select o;

 //prezentacja wyników zapytania
 string s = "Lista osób pełnoletnich:\n";
 foreach (Osoba o in osobyPelnoletnie)
 s += o.Imię + " " + o.Nazwisko + "(" + o.Wiek + ")\n";
 }
}
```

```

 MessageBox.Show(s);
 }
}

```

Pobrane w ten sposób dane można także modyfikować, a ponieważ wynik zapytań zawiera referencję do oryginalnych rekordów, zapisać do bazy danych, wywołując metodę `bazaDanychAdresy.SaveChanges`. Taka możliwość znika, jeżeli zaznaczymy zakres pobieranych pól tabeli, korzystając z obiektów anonimowych (listing 17.5).

**Listing 17.5.** Użycie klasy anonimowej w zapytaniu LINQ to Entities

```

//zapytanie LINQ to Entities
var zredukowaneOsobyPelnoletnie = from o in bazaDanychAdresy.Osoby
 select new { o.Imię, o.Nazwisko, o.Wiek };

//prezentacja wyników zapytania
s = "Lista osób pełnoletnich:\n";
foreach (var o in zredukowaneOsobyPelnoletnie)
 s += o.Imię + " " + o.Nazwisko + " (" + o.Wiek + ")\n";
MessageBox.Show(s);

```

Zaskoczeniem może być, że nie uda się wykonanie zapytania, w którym tworzona ma być kolekcja łańcuchów (pojawia się wyjątek przy pierwszej próbie użycia wyniku zapytania, czyli w miejscu, gdzie tak naprawdę zapytanie jest wykonywane):

```

var personaliaOsobPelnoletnich = from o in bazaDanychAdresy.Osoby
 select o.Imię + " " + o.Nazwisko + " (" + o.Wiek + ")";

```

Komunikat wyjątku obwieści nam, że LINQ to Entities wspiera tylko rzutowanie wyników zapytania na typy z modelu danych EDM lub typów wyliczeniowych. Najprostszym, choć może nie najbardziej kanonicznym sposobem, żeby ten problem obejść, jest zamiana LINQ to Entities na LINQ to Objects:

```

var personaliaOsobPelnoletnich = from o in bazaDanychAdresy.Osoby.ToList()
 select o.Imię + " " + o.Nazwisko + " (" + o.Wiek + ")";

```

## Prezentacja i edycja danych w siatce

Spróbujmy teraz zaprezentować dane pobrane z tabeli w siatce `DataGridView`. Od razu postarajmy się, aby dane zmodyfikowane za pomocą tej siatki mogły być z powrotem zapisane do bazy danych np. przy zamknięciu aplikacji (wcześniej poprosimy użytkownika o potwierdzenie).

1. Zaczniemy od umieszczenia kontrolki `DataGridView` na podglądzie okna w widoku projektowania. Wygodne będzie zakotwiczenie jej do brzegów formy (własność `Anchor`).
2. Następnie przejdźmy do edycji kodu pliku `Form1.cs` i w klasie `Form1` zdefiniujmy dwa pola:

```

AdresyEntities bazaDanychAdresy;
bool daneZmienione;

```

Przeznaczenie pierwszego jest już oczywiste. Drugie będzie służyło jako flaga podnoszona, gdy użytkownik zmieni dane wyświetlane w siatce (zobacz podrozdział „Zapisywanie zmodyfikowanych danych” z poprzedniego rozdziału).

- Następnie utwórzmy metodę zdarzeniową do zdarzenia Load formy (wystarczy kliknąć dwukrotnie formę w widoku projektowania) i wczytajmy w niej dane (listing 17.6). W tym momencie po uruchomieniu aplikacji siatka powinna już być zapełniona danymi (rysunek 17.5).

**Listing 17.6.** Łączenie siatki z danymi

```
private void Form1_Load(object sender, EventArgs e)
{
 bazaDanychAdresy = new AdresyEntities();
 bazaDanychAdresy.Osoby.Load();

 dataGridView1.DataSource = bazaDanychAdresy.Osoby.Local.ToBindingList<Osoba>();

 daneZmienione = false;
}
```

**Rysunek 17.5.**

Dane z tabeli

Osoby prezentowane  
w siatce DataGridView

|   | Id | Imię     | Nazwisko   | Email           | NumerTelefon | Wiek |
|---|----|----------|------------|-----------------|--------------|------|
| ▶ | 1  | Jacek    | Matulewski | jacek@phys...   | 123456789    | 43   |
|   | 2  | Jan      | Kowalski   | jan.kowalski... |              | 26   |
|   | 3  | Wojciech | Kotek      | kotek@firma...  | 999111111    | 54   |
|   | 4  | Tomasz   | Trombalski | tt@firma.pl     | 888222222    | 36   |
|   | 5  | Ania     | Kowalska   | ak@firma.pl     | 888333333    | 31   |
|   | 6  | Karolina | Kowalska   | karolina@w...   |              | 15   |
|   | 7  | Bartosz  | Kowalski   | bartosz@po...   |              | 10   |
| * |    |          |            |                 |              |      |
| ◀ |    |          |            |                 |              |      |

- Metoda rozszerzająca `ToBindingList<>` zdefiniowana jest w przestrzeni nazw `System.Data.Entity`, aby więc była widoczna (także w *IntelliSense*), należy w pliku *Form1.cs*, w bloku poleceń `using`, dodać instrukcję `using System.Data.Entity;`.
- Dalsze czynności będą związane z zapisywaniem zmodyfikowanych danych. Przede wszystkim zadbajmy o to, aby podnieść flagę w razie modyfikacji danych w komórkach siatki (zdarzenie `DataGridView.CellValueChanged`). Pokazuje to listing 17.7.
- I wreszcie stwórzmy metodę zdarzeniową związaną ze zdarzeniem `Form1.FormClosing`, widoczną na listingu 17.8, w której dajemy użytkownikowi wybór, czy chce zapisać zmiany, czy je odrzucić. Może również anulować zamknięcie okna.

**Listing 17.7.** Podniesienie flagi

```
private void dataGridView1_CellValueChanged(object sender,
DataEventArgs e)
{
 daneZmienione = true;
}
```

**Listing 17.8.** Typowa dla edytorów obsługa zamknięcia okna

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
 if (!daneZmienione) return;
 switch (MessageBox.Show("Czy zapisać zmiany do bazy danych?", this.Text,
 MessageBoxButtons.YesNoCancel))
 {
 case DialogResult.Cancel:
 e.Cancel = true;
 break;
 case DialogResult.Yes:
 try
 {
 this.Validate();
 bazaDanychAdresy.SaveChanges();
 bazaDanychAdresy.Dispose();
 }
 catch (Exception exc)
 {
 MessageBox.Show("Zapisanie danych
 →nie powiodło się ("+exc.Message+ ")");
 }
 break;
 case DialogResult.No:
 break;
 }
}
```

Dzięki tym wszystkim czynnościom po uruchomieniu formy, które może trwać wyraźnie dłużej, w siatce prezentowane są dane z tabeli *Osoby*, które można edytować, a zmiany mogą być zapisane przed zamknięciem aplikacji.

## Asynchroniczne wczytywanie danych

Inicjacja aplikacji trwa dość długo, głównie ze względu na wywołanie metody `bazaDanychAdresy.Osoby.Load`. Wobec tego warto zwrócić uwagę na jej wersję asynchroniczną, tj. `LoadAsync`. Dokładniej rzecz ujmując: jest to metoda rozszerzająca dodana do

EF w wersji 6.0, widoczna po deklaracji użycia przestrzeni nazw `System.Data.Entities`<sup>3</sup>. Metoda ta zwraca referencję do zadania, zatem idealnym rozwiązaniem będzie użycie poznanego w rozdziale 7. operatora `await`. Jak pamiętamy z rozdziału 7., operator ten sprawia, że kompilator potnie metodę, w której jest umieszczony, w taki sposób, że polecenia znajdujące się za instrukcją `await` zostaną wykonane dopiero po zakończeniu zadania (zostaną *de facto* umieszczone w wywoływanej wówczas metodzie zwrotnej). Do tego czasu wątek nie będzie jednak blokowany i będzie mógł dokończyć m.in. inicjację okna. Tym samym okno będzie pokazane i będzie „responsible”. Listing 17.9 pokazuje, w jaki sposób zmodyfikować metodę `Form1_Load`, aby użyć asynchronicznego wczytywania danych.

---

**Listing 17.9. Użycie asynchronicznej metody do wczytywania danych**

---

```
private async void Form1_Load(object sender, EventArgs e)
{
 bazaDanychAdresy = new AdresyEntities();

 await bazaDanychAdresy.Osoby.LoadAsync();
 dataGridView1.DataSource = bazaDanychAdresy.Osoby.Local.ToBindingList<Osoba>();

 daneZmienione = false;
}
```

---

Również zapisywanie danych może być przeprowadzone asynchronicznie (metoda `SaveChangesAsync`). Jednak ponieważ my dane zapisujemy tuż przed zamknięciem aplikacji, próba zapisania ich asynchronicznie nie miałaby praktycznego znaczenia dla działania aplikacji, a źle przeprowadzona (bez wymuszenia synchronizacji) mogłaby doprowadzić do niezapisania danych.

## Użycie widoku i procedur składowanych

Poza tabelami klasa `AdresyEntities` udostępnia także widok (własność `OsobyPełnoletnie`) i trzy procedury składowane. Te ostatnie udostępniane są w postaci wygodnych metod. Zaczniemy od widoku. Jak pamiętamy, z powodu braku klucza głównego jest on dostępny tylko do odczytu. Pokażemy jego zawartość w drugiej siatce. Umieścimy wobec tego siatkę `DataGridView` na stronie (można zmienić jej właściwość `ReadOnly` na `true`) i wczytajmy do niej dane poleceniami dodanymi do metody `Form1_Load`:

---

<sup>3</sup> We wcześniejszych wersjach możemy ją z łatwością zdefiniować sami:

```
public static class Rozszerzenia
{
 public static Task LoadAsync<T>(this DbSet<T> ds) where T : class
 {
 Task zadanie = new Task(ds.Load);
 zadanie.Start();
 return zadanie;
 }
}
```

```
await bazaDanychAdresy.OsobyPelnoletnie.LoadAsync();
dataGridView2.DataSource =
 bazaDanychAdresy.OsobyPelnoletnie.Local.ToBindingList<OsobyPelnoletnie>();
```

Z bardzo podobnym skutkiem możemy użyć procedury składowej — zapytania Lista OsobPelnoletnich:

```
dataGridView2.DataSource = bazaDanychAdresy.ListaOsobPelnoletnich();
```

Metoda ta zwraca kolekcję parametryzowaną typem ListaOsobPelnoletnich\_Result.

Równie łatwe jest wywołanie procedury AktualizujWiek. Zróbcmy to, korzystając z dodatkowego przycisku. Po wywołaniu procedury odświeżmy zawartość pierwszej siatki, aby móc zobaczyć wynik jej działania (listing 17.10). Nie musimy martwić się o zapisanie danych w pliku bazy danych — procedura składowa działa w końcu bezpośrednio na danych w bazie. Problemem jest natomiast aktualizacja kontrolek — wymuszamy ją w dość brutalny sposób, wywołując metodę Form1\_Load.

**Listing 17.10.** Wywołanie metody Form1\_Load nie jest może najbardziej eleganckim rozwiązaniem

---

```
private void button3_Click(object sender, EventArgs e)
{
 bazaDanychAdresy.AktualizujWiek();
 Form1_Load(null, null);
}
```

---

W podobny sposób można uruchomić procedurę składowaną TwórzNowąTabelę. Powstanie wówczas tabela *Faktury*, która nie jest odwzorowana w plikach EDM. Można to jednak łatwo zmienić, korzystając z polecenia *Update Model from Database...* z menu kontekstowego rozwijanego na zakładce *Adresy.edmx [Diagram1]*.

## Połączenie między tabelami

Źródła danych (ang. *data sources*), których używaliśmy w poprzednich rozdziałach i które okazały się nad wyraz wygodne, dostępne są również w przypadku *Entity Framework*. Przekonajmy się o tym, tworząc interfejs w stylu *master-details* dla tabeli osób i przypisanych do nich rozmów<sup>4</sup>. Tabele w bazie danych nie są powiązane. To dawało nam pretekst do łączenia ich reprezentacji na poziomie modelowania ORM. Zrobimy tak i w przypadku EF. Stworzymy „zwykłe” połączenie jeden do wielu, w którym wykorzystamy pole *Id* tabeli *Osoby* i *Rozmowy*. Nie wykorzystamy przy tym oferowanej domyślnie możliwości utworzenia klucza obcego (ang. *foreign key*) w tabeli zawierającej szczegóły.

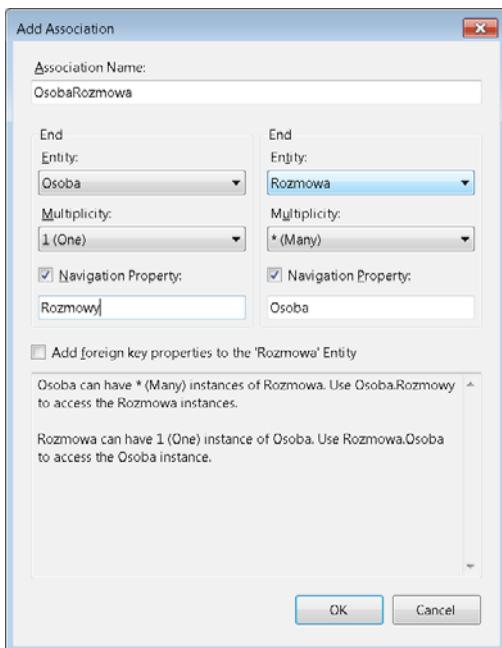
1. Przejdz na zakładkę *Adresy.edmx [Diagram1]*.
2. Z menu kontekstowego tabeli/obiektu *Osoba* wybierz *Add New, Association...*

---

<sup>4</sup> Zobacz <http://msdn.microsoft.com/en-us/data/jj706685.aspx> i <http://msdn.microsoft.com/en-us/data/jj713299.aspx>.

3. Pojawi się okno *Add Association*, w którym wprowadzamy ustawienia analogiczne jak na rysunku 17.6. Zwróć uwagę, że pole *Navigation Property* w lewej kolumnie zmieniłem z domyślnego *Rozmowa* na *Rozmowy*. To lepiej odda fakt, że jednej osobie może odpowiadać wiele rozmów. Ważną zmianą jest również usunięcie zaznaczenia pola opcji *Add foreign key properties to the 'Rozmowa' Entity*.

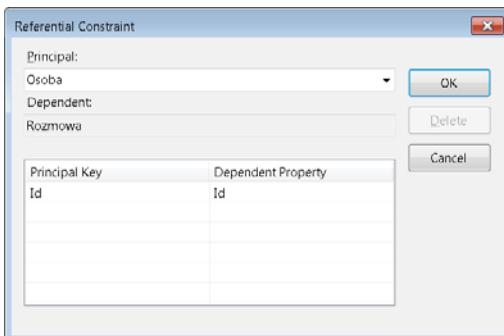
**Rysunek 17.6.**  
Tworzenie połączenia  
między obiektami  
reprezentującymi tabele



4. Po kliknięciu *OK* tabele w widoku *Adresy.edmx [Diagram1]* powinny zostać połączone strzałką.
5. Zwróćmy jednak uwagę, że nie mieliśmy możliwości wskazania, jakie pola obu tabel mają być podstawą związku między nimi. Aby to zrobić, kliknijmy dwukrotnie strzałkę łączącą tabele *Osoba* i *Rozmowa*. Pojawi się okno *Referential Constraint*. Musimy wskazać tabelę *podstawową* (ang. *principal*). Powinna to być tabela *Osoba*. Jeżeli ją wskażemy, automatycznie ustawione zostaną klucz podstawowy i zależny, tak aby wskazywały pola *Id* w obu tabelach (rysunek 17.7).

Pamiętajmy o tym, żeby przed przystąpieniem do tworzenia źródła zapisać zmiany wprowadzone w obiektach EDM. Warto również przekompilować projekt. Czasem pojawiają się w tym momencie błędy związane z odwzorowywaniem relacji między tabelami, a raczej klasami, które je reprezentują. Jeżeli jednak wszystko jest w porządku, po zapisie zmian warto zanjrzeć do pliku *Osoba.cs* (listing 17.11). Utworzona relacja została odzwierciedlona w klasie encji jako kolekcja *Rozmowy* zawierająca wszystkie rozmowy przeprowadzone przez daną osobę. Będzie też widoczna w źródle danych, które zaraz utworzymy.

**Rysunek 17.7.**  
Szczegóły połączenia  
tabel



**Listing 17.11.** Połączenie obiektów reprezentujących tabele powoduje dodanie kolekcji do klasy encji

```

// -----
// <auto-generated>
// This code was generated from a template.
//
// Manual changes to this file may cause unexpected behavior in your application.
// Manual changes to this file will be overwritten if the code is regenerated.
// </auto-generated>
// -----

namespace AplikacjaZBazaDanych
{
 using System;
 using System.Collections.Generic;

 public partial class Osoba
 {
 public Osoba()
 {
 this.Rozmowy = new HashSet<Rozmowa>();
 }

 public int Id { get; set; }
 public string Imię { get; set; }
 public string Nazwisko { get; set; }
 public string Email { get; set; }
 public Nullable<int> NumerTelefonu { get; set; }
 public int Wiek { get; set; }

 public virtual ICollection<Rozmowa> Rozmowy { get; set; }
 }
}

```

## Tworzenie źródła danych

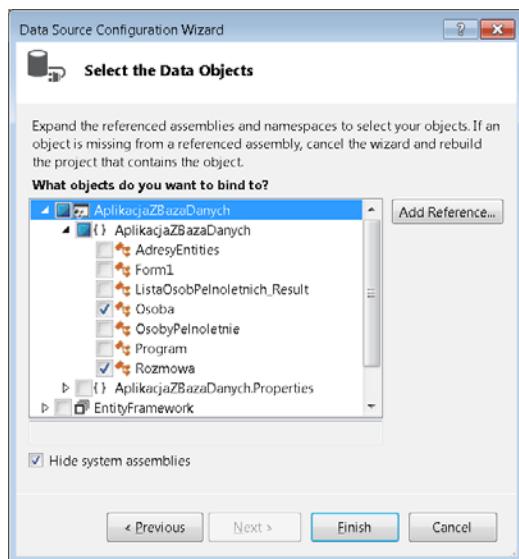
Wreszcie przejdźmy do tworzenia źródła danych. Jest to równie proste jak w przypadku LINQ to SQL:

1. Z menu *Project* wybieramy *Add New Data Source....*. Uruchamiamy w ten sposób kreator — okno zatytułowane *Data Source Configuration Wizard*.

2. W pierwszym kroku kreatora zaznaczamy ikonę *Object* i klikamy przycisk *Next*.
3. W drugim — wskazujemy obiekty, które mają być źródłem danych (rysunek 17.8). Powinien to być przede wszystkim obiekt *Osoba*. Ze względu na późniejsze problemy dodajmy jednak także obiekt *Rozmowa*. Oba znajdują się w przestrzeni nazw *AplikacjaZBazaDanych*.

**Rysunek 17.8.**

*Wskazywanie  
źródeł danych*

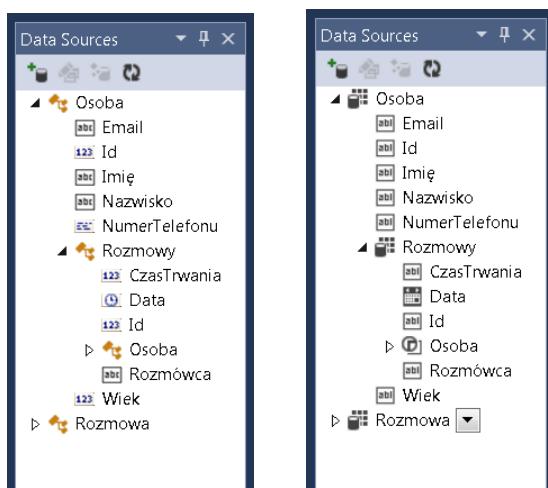


4. Klikamy *Finish*.

W podoknie *Data Sources* pojawią się dwa nowe źródła danych (rysunek 17.9, lewy i prawy). Należy zwrócić uwagę, czy w źródle *Osoba* widoczny jest element *Rozmowy*. W moim przypadku konieczne było wcześniejsze zapisanie zmodyfikowanego modelu EDM i skompilowanie projektu bez błędów.

**Rysunek 17.9.**

*Podokno źródła danych.  
Z lewej widok tego okna,  
gdy w głównej części  
okna VS widoczne są  
klasy EDM. Z prawej  
— przy otwartym widoku  
projektowania interfejsu  
formy*



## Automatyczne tworzenie interfejsu

Wiemy już, że gdy dysponujemy się źródłami danych widocznymi w podoknie *Data Sources*, tworzenie interfejsu jest proste i przyjemne. Interfejs będzie składał się z dwóch części: formularza prezentującego bieżącą osobę (zmianę osoby umożliwi kontrolka typu *BindingNavigator*) oraz siatki zawierającej zbiór rozmów tej osoby. Umieścimy je na nowej formie.

1. Z menu *Project* wybieramy polecenie *Add Windows Form...*. Pojawi się okno dialogowe, w którym możemy od razu kliknąć przycisk *Add*. Do projektu dodana zostanie forma o nazwie *Form2* zapisana m.in. w pliku *Form2.cs*.
2. W widoku projektowania nowej formy zwiększymy jej rozmiar tak, żeby zmieścił się na niej formularz i siatka. Następnie w podoknie *Data Sources* (rysunek 17.9, prawy) rozwijamy listę związaną z pozycją *Osoba* i wybieramy *Details*. Postępując podobnie przy polu *Wiek*, zmieniamy typ kontrolki na *NumericUpDown*.
3. Przeciągamy pozycję *Osoba* z podokna *Data Sources* na podgląd okna widocznego na zakładce *Form2.cs [Design]*. Na formie pojawi się zbiór kontrolek z etykietami oraz zadokowana do górnej krawędzi kontrolka *osobaBindingNavigator* wspomagająca nawigację po rekordach tabeli.
4. Umieszczone na formie kontrolki odpowiadają wszystkim elementom obiektu *Osoba* poza kolekcją *Rozmowy* (listing 17.12). Tak naprawdę nie wszystkie są potrzebne; można, a nawet należy usunąć np. element *Id*. Wspomniany podelement *Rozmowy* przeciągamy osobno. Zgodnie z domyślnymi ustawieniami na formie powstanie wówczas siatka. Jednak zamiast spodziewanych kolumn ja zobaczyłem tylko dwie: *Count* i *IsReadOnly*. To oczywiście błęd, który pojawił się w wersji Entity Framework 5.0 i nadal nie został poprawiony. W tej sytuacji konieczne jest obejście problemu. Tym zajmiemy się jednak za chwilę, a najpierw sprawdźmy, czy kontrolki formularza zapełniają się danymi. To wymaga wczytania tych danych.
5. Naciśnijmy *F7*, aby przejść do edycji kodu pliku *Form2.cs*. W klasie *Form2* zdefiniujmy pole *bazaDanychAdresy* typu *AdresyEntities*. W konstruktorze zainicjujmy to pole, tworząc obiekt typu *AdresyEntities*. I wreszcie jako źródło danych dla *osobaBindingSource* wskażmy tabelę *Osoby* z obiektu *bazaDanychEntities* (listing 17.12).

**Listing 17.12.** Wczytywanie danych

```
public partial class Form2 : Form
{
 AdresyEntities bazaDanychAdresy;
 public Form2()
 {
 InitializeComponent();
 bazaDanychAdresy = new AdresyEntities();
 osobaBindingSource.DataSource = bazaDanychAdresy.Osoby.ToList();
 }
}
```

6. Na koniec dopilnujmy, aby powstała instancja klasy opisującej nową formę. Przejdźmy na zakładkę *Form1.cs* i do konstruktora klasy *Form1* dodajmy instrukcję `new Form2().Show();`.

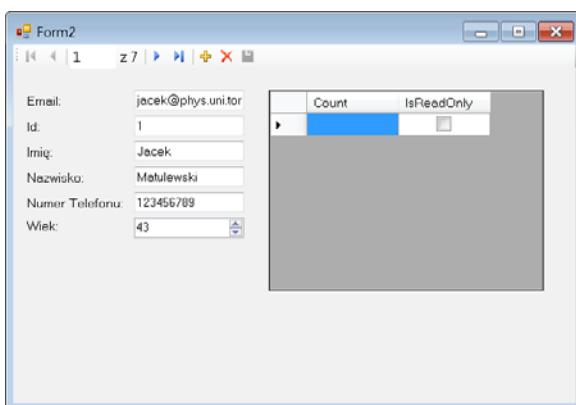
Jeżeli chcemy ograniczyć wyświetlane w formularzu osoby do tych, które przeprowadziły jakieś rozmowy, możemy zmodyfikować zapytanie określające zakres danych udostępnianych przez *osobyBindingSource*:

```
osobaBindingSource.DataSource =
 bazaDanychAdresy.Osoby.Where(o => o.Rozmowy.Any()).ToList();
```

Po uruchomieniu aplikacji powinniśmy zobaczyć formularz wypełniony danymi (rysunek 17.10). Natomiast siatka powinna zawierać listę rozmów przeprowadzonych przez bieżącą osobę. Tak się jednak nie dzieje. W zamian zobaczymy tylko publiczne właściwości kolekcji *HashSet*, tj. *Count* i *IsReadOnly*<sup>5</sup>.

Rysunek 17.10.

Błąd w przypadku umieszczenia na formie siatki zależnej



Możemy to naprawić na dwa sposoby. Pierwszym jest proste obejście problemu. Możemy po prostu niezależnie od formularza podłączyć siatkę do drugiego źródła danych, filtrując je tak, aby wyświetlane były tylko rozmowy dla wskazanego *Id* osoby. Użyłem do tego zdarzenia *CurrentChange* komponentu *osobyBindingSource* informującego o zmianie bieżącego rekordu. W powiązanej z tym zdarzeniem metodzie zmieniam źródło danych drugiej siatki.

1. W widoku projektowania formy *Form2* usuń dodaną przed chwilą siatkę oraz jej źródło *rozmowyBindingSource*.
2. W zamian przeciągnij tam źródło *Rozmowa* (nie gałąź źródła *Osoby*, a drugie osobne źródło).
3. Zaznacz kontrolkę *osobyBindingSource* widoczną na pasku pod podglądem formy. Korzystając z podokna *Properties*, utwórz metodę zdarzeniową związaną ze zdarzeniem *CurrentChange* tego obiektu, a w niej umieść polecenia widoczne na listingu 17.13.

<sup>5</sup> Por. zgłoszenie tego błędu: <http://social.msdn.microsoft.com/Forums/en-US/f6ac20c1-02b0-44e7-9ee8-65a3a900a063/entity-framework-child-entity-on-gridview-only-count-is-read-only?forum=adodotnetentityframework>. Ten błąd pojawia się także w wersji 5.0.

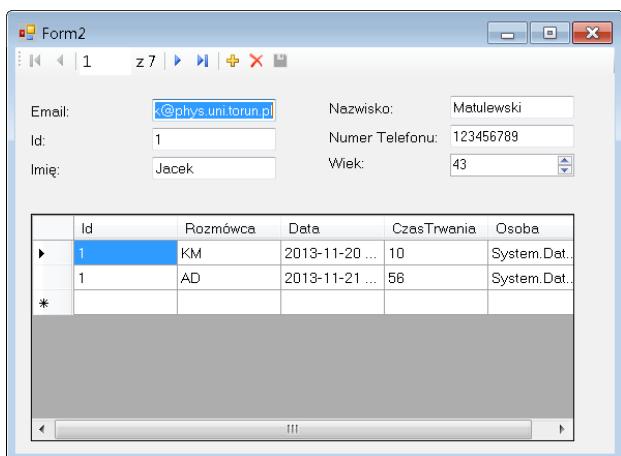
**Listing 17.13.** Filtrowanie danych widocznych w siatce

```
private void osobaBindingSource_CurrentChanged(object sender, EventArgs e)
{
 var rozmowyOsoba = bazaDanychAdresy.Rozmowy.
 Where(r => r.Id == ((Osoba)osobaBindingSource.Current).Id);
 rozmowaBindingSource.DataSource = rozmowyOsoba.ToList();
}
```

Możemy teraz uruchomić aplikację. W siarce powinny pojawiać się rozmowy osoby widocznej w formularzu (rysunek 17.11).

**Rysunek 17.11.**

Formularz prezentujący osobę i siatkę zawierającą listę rozmów tej osoby



Drugi sposób, nieco bardziej wyrafinowany, wymaga modyfikacji skryptu generującego klasy w modelu EDM. Jego celem jest zmiana typu kolekcji Rozmowy w klasie Osoba (listing 17.11) z HashSet na ObservableCollection<sup>6</sup>. Taką zmianę możemy zresztą wprowadzić do klasy Osoba ręcznie (należy zmienić zarówno deklarację referencji, jak i typ tworzonego obiektu) — wówczas problem zniknie i przeciagnięcie pudełka Rozmowy z okna *Data Sources* na formę da prawidłowy wynik. Kłopot w tym, że klasa ta jest generowana automatycznie i każda zmiana w modelu EDM spowoduje zamazanie naszych poprawek. Dlatego należy zmodyfikować skrypt generujący te klasy, który znajdziemy w pliku *Adresy.tt*, a konkretnie należy wprowadzić do niego następujące trzy zmiany:

linia 50.:

```
this.<#=code.Escape(navigationProperty)#> = new ObservableCollection<#=typeMapper.
GetTypeName(navigationProperty.ToEndMember.GetEntityType())#>();
```

linia 296.:

```
navigationProperty.ToEndMember.RelationshipMultiplicity == Relationship-
Multiplicity.Many ? ("ObservableCollection<" + endType + ">") : endType,
```

<sup>6</sup> Por. <http://stackoverflow.com/questions/12695754/unable-to-create-gridview-by-dragging-objectdatasource-to-winform> i <http://msdn.microsoft.com/en-us/data/jj682076.aspx>.

po linii 424. dodajemy linię:

```
includeCollections ? (Environment.NewLine + "using
→System.Collections.ObjectModel;") : "",
```

Po tych zmianach i zapisaniu skryptu, co spowoduje jego ponowne uruchomienie, klasa Osoba zostanie wygenerowana w taki sposób, że tworzenie tabeli zawierającej szczegóły rozmów będzie działało prawidłowo.

## Edycja i zapis zmian

Po połączeniu kontrolek z danymi jedyną rzeczą, jaką należy zrobić, aby zmiany wprowadzone w formularzu i siatce były zapisywane do pliku bazy danych, jest wywołanie metody bazaDanychAdresy.SaveChanges. Zwykle robiliśmy to przy zamknięciu aplikacji. Teraz proponuję jednak użyć do tego ikony dyskietki widocznej w kontrolce osobyBindingNavigator. W tym celu:

1. W widoku projektowania formy Form2 kliknij prawym klawiszem myszy nieaktywną ikonę dyskietki w kontrolce osobyBindingNavigator widocznej przy górnej krawędzi podglądu formy.
2. Z jej menu kontekstowego wybierz polecenie *Enabled*.
3. Dwukrotnie kliknij odblokowaną ikonę, aby utworzyć jej domyślną metodę zdarzeniową, i umieść w tak utworzonej metodzie polecenie zapisania zmian (listing 17.14).

**Listing 17.14.** Zapisywanie zmian wprowadzonych w kontrolkach

```
private void osobaBindingNavigatorSaveItem_Click(object sender, EventArgs e)
{
 osobaBindingSource.EndEdit();
 rozmowaBindingSource.EndEdit();
 bazaDanychAdresy.SaveChanges();
}
```

Jak wspomniałem wyżej, zapisywanie można także przeprowadzić asynchronicznie. Wówczas należy jednak przypilnować, aby zadanie zapisywania zakończyło się przed ewentualnym zamknięciem aplikacji.

\* \* \*

Entity Framework to już dojrzałe narzędzie ORM, choć jak zaznaczyłem we wstępie do tego rozdziału, martwi mnie jego wydajność. Jest to jednak zagadnienie ważne tylko przy intensywnym zapisie i odczytcie; w typowych scenariuszach ma mniejszy wpływ na działanie aplikacji, szczególnie jeżeli wykorzystamy możliwości asynchronicznego wczytywania danych.



Część IV

## **Dodatki**



# Zadania

## Język C#, programowanie obiektowe, LINQ

Wszystkie zadania z tej części należy wykonać w projekcie aplikacji konsolowej.

1. Przygotuj metodę o sygnaturze `static long Silnia(byte argument);` obliczającą silnię liczby podanej w argumencie. Silnia z liczby 3 to  $3! = 1 \cdot 2 \cdot 3$ . Silnia z liczby  $n$  to  $n! = 1 \cdot 2 \cdots \cdot (n - 1) \cdot n$ . Sprawdź, czy wartości zwracane przez metodę są prawidłowe, oraz ustal, dla jakich wartości argumentów wartość tej metody pozostaje w zakresie liczby `long` (maks. wartość to 9 223 372 036 854 775 807). Dodaj do metody warunek sprawdzający, czy argument jest z prawidłowego zakresu (w przeciwnym przypadku należy zgłosić wyjątek `ArgumentOutOfRangeException`).
2. Przygotuj metodę `CiągFibonacciego`, która zapełnia podaną w argumencie tablicę kolejnymi wyrazami ciągu Fibonacciego, zaczynając od 1.
3. Znajdź metodę obliczającą podatek od podanej w argumencie kwoty zgodnie z poniższym wzorem:
  - a) za mniej niż 10 tys. — 10%,
  - b) za sumę z przedziału 10 tys. – 30 tys. — 15%,
  - c) za kwotę większą niż 30 tys. — 20%.
4. Korzystając z wielokrotnej pętli `for` i instrukcji `Console.WriteLine`, wyświetl w konsoli następujące wzory:

|      |       |        |                 |
|------|-------|--------|-----------------|
| **** | 54321 | 121212 | 12233           |
| ***  | 65432 | 212121 | 223334444       |
| **   | 76543 | 121212 | 333444455555    |
| *    | 87654 | 212121 | 444455555666666 |

5. Korzystając z pętli `do..while`, napisz program-zabawę w zgadywanie liczby „pomyślanej” przez komputer (z zakresu 1 – 10).
6. Miasto T. ma obecnie 100 tys. mieszkańców, ale jego populacja zwiększa się o 3% rocznie. Miasto B. ma 300 tys. mieszkańców i ta liczba rośnie w tempie 2% na rok. Wykonaj symulację prezentującą liczbę mieszkańców w obu miastach i zatrzymującą się, gdy liczba mieszkańców miasta T. przekroczy liczbę z miasta B.
7. Napisz program losujący zadaną liczbę razy liczbę z przedziału [1,6] (rzut kostką). Sprawdź rozkład wyników (liczbę rzutów, w których wylosowane zostały poszczególne liczby oczek), oblicz średnią, medianę i wariancję.
8. Napisz program proszący użytkownika o napisanie liczby z zakresu od 1 do 10. Wyświetl kwadrat tej liczby. W razie wpisania liczby spoza zakresu wyświetl komunikat i umożliwi ponowne jej wpisanie.
9. Program z poprzedniego punktu wyposaź w menu pozwalające na:
  - a) dodanie do liczby 10,
  - b) pomnożenie liczby przez 2,
  - c) odjęcie od liczby 1,
  - d) wyjście z programu.

Wybór pozycji z menu powinien następować poprzez wpisanie numeru pozycji. Skorzystaj z instrukcji `switch`.

1. Przygotuj program wyświetlający na ekranie konsoli wskazany w linii poleceń plik tekstowy.
2. W programie z poprzedniego punktu użij formatowania (kolory) do zaznaczenia wybranych słów kluczowych (`for`, `while`, `do`, `if`, `else`, `switch` itp.) przy wyświetlaniu pliku z kodem C#.
3. Napisz program pozwalający na dynamiczne tworzenie listy łańcuchów i jej wyświetlanie na ekranie. Należy uwzględnić możliwość dodania kolejnej pozycji do listy, sortowanie pozycji w liście i usuwanie z listy łańcucha na wskazanej pozycji.
4. W aplikacji konsolowej przygotuj słownik (kolekcja `Dictionary` — bardzo podobna do omówionej w rozdziale 3. kolekcji `SortedList`) o następujących elementach:

| klucz   | wartość  |
|---------|----------|
| jest    | is       |
| ma      | has      |
| lubi    | likes    |
| oglądać | to watch |
| kot     | cat      |
| włosy   | hair     |
| filmy   | movies   |
| bardzo  | very     |
| rudy    | red      |
| rude    | red      |

Korzystając z tego słownika, przetłumacz poniższe zdania (wyszukuj wyrazy klucze i zastępuj je wyrazami wartościami):

Ala ma kota.  
Bartek jest bardzo wysoki.  
Kasia ma rude włosy.  
Karolina lubi oglądać filmy.  
Jacek lubi C#.

Przygotuj kod tłumaczący zdania z powrotem na polski.

**5.** W aplikacji konsolowej zdefiniuj tablicę łańcuchów:

```
string[] słowa = { "czereśnia", "jabłko", "borówka", "wiśnia", "jagoda",
"gruszka", "śliwka", "malina" };
```

Korzystając z metod-rozszerzeń LINQ, wyświetli:

- a)** najdłuższą i najkrótszą długość słowa (Min i Max z odpowiednimi wyrażeniami lambda);
- b)** średnią długość słów (Average);
- c)** całkowitą liczbę liter we wszystkich słowach (Sum).

Przygotuj zapytania LINQ, które:

- a)** zwraca wszystkie słowa o długości większej niż 6 liter posortowane alfabetycznie;
- b)** zwraca wszystkie słowa o długości większej niż 6 liter posortowane według ich długości;
- c)** zwraca wszystkie słowa kończące się na „a” posortowane według ostatniej litery;
- d)** zwraca długości poszczególnych słów posortowane według alfabetycznej kolejności tych słów;
- e)** jak w podpunkcie d, ale tylko dla słów, które zawierają literę „o”;
- f)** zwraca słowa z tablicy ze zmienionymi literami na duże.

**6.** W nowym projekcie zdefiniuj klasę Nagrywarka, która posiada następujące pola: `miejsceNagrywania` (typ wyliczeniowy o wartościach DVD i HDD), `stan` (typ wyliczeniowy, możliwe wartości: Wyłączone, Zatrzymane, Nagrywanie, Odtwarzanie). Wartość pola `stan` niech będzie udostępniana przez własność tylko do odczytu `Stan`, a `miejsceNagrywania` przez własność umożliwiającą zapis i odczyt. Klasa Nagrywarka powinna mieć metody: `Włącz`, `Odtwarzaj`, `Nagrywaj`, `Zatrzymaj`. Odtwarzanie i nagrywanie są czynnościami wzajemnie wykluczającymi się, tzn. nie można włączyć nagrywania, gdy włączone jest odtwarzanie i odwrotnie. Obie czynności możliwe są dopiero po włączeniu urządzenia (przełączenie ze stanu `Wyłączone` do `Zatrzymane`).

**7.** Przygotuj klasę `Nagranie`, która opisuje nagrany przez nagrywarkę film (numer nagrania oraz data i czas rozpoczęcia i zakończenia nagrywania), a następnie użyj jej w klasie `NagrywarkaZPamięcią` (dziedziczącej z klasy

Nagrywarka z poprzedniego zadania), w której przechowywana jest lista nagrań (`List<Nagranie>`). Dodaj funkcjonalność pozwalającą na odtwarzanie wybranego nagrania i jego usuwanie.

8. Zdefiniuj rozszerzenie dla klasy `NagrywarkaZPamięcią`, które wyświetla jej pełny stan z listą nagrań.
9. Zdefiniuj metodę rozszerzającą dla wszystkich obiektów .NET (dla klasy `Object`), która korzystając z metody `ToString`, konwertuje obiekt do łańcucha i pokazuje go w konsoli.
10. Przygotuj klasę `KodPocztowy` zawierającą pola `OkrągPocztowy` (dwie cyfry) i `SektorKodowy` (trzy cyfry) z nadpisaną metodą `ToString` zwracającą obie części kodu połączone myślnikiem.
11. Przygotuj klasę `Adres` zawierającą publiczne pola `Kraj`, `Miasto`, `Ulica`, `NumerDomu` i `NumerMieszkania` typu `string` oraz pole `KodPocztowy` typu `KodPocztowy` (klasa z poprzedniego zadania). Utwórz instancję klasy przechowującą Twój adres zamieszkania.
12. Przygotuj klasę `OsobaZameldowana` dziedziczącą z klasy `Osoba` z rozdziału 11. Przygotuj kolekcję obiektów tego typu. Dla tej kolekcji zredaguj zapytania LINQ zwracające:
  - a) osoby pełnoletnie mieszkające w Twoim mieście zamieszkania posortowane wg wieku malejąco;
  - b) kobiety mające więcej niż 40 lat zgrupowane wg województw (pierwsza część kodu pocztowego);
  - c) mężczyzn pełnoletnich niemieszkających w Twoim mieście zamieszkania posortowanych według kodu pocztowego;
  - d) kobiety, których nazwisko nie kończy się na literę „a”;
  - e) łańcuch zawierający imię i nazwisko oraz wiek w nawiasie dla osób z pierwszego i drugiego zapytania;
  - f) wiek (liczba całkowita typu `int`) osób niepełnoletnich, posortowany wg kodu pocztowego (nieobecnego w zwracanych danych).
13. Zdefiniuj interfejs `IPosiadajacyAdresEmail` wymuszający zdefiniowanie właściwości `Email` i użyj go w klasie `Osoba`.
14. Przygotuj strukturę `LiczbaZespolona` implementującą liczby zespolone. Struktura powinna zawierać właściwości *auto-implemented*: `Real` i `Imag` typu `double` (część rzeczywista i urojona), metody: `Conj` (sprzężenie zespolone zmieniające znak części urojonej) i `ToString`, stałe: `Zero (0,0)`, `Jeden (1,0)` i `I (0,1)`, oraz operatory implementujące podstawowe operacje arytmetyczne: `+`, `-`, `*` (nie trzeba definiować dzielenia). Więcej informacji (m.in. definicje działań dla liczb zespolonych) na stronie: [http://pl.wikipedia.org/wiki/Liczby\\_zespolone](http://pl.wikipedia.org/wiki/Liczby_zespolone).
15. Zdefiniuj rozszerzenie klasy `LiczbaZespolona`, które oblicza normę liczby zespolonej (suma kwadratów części rzeczywistej i urojonej).

- 16.** Klasę `LiczbaZespolona` z wcześniejszego zadania przenieś do biblioteki PCL i przygotuj dla niej przynajmniej dziesięć testów jednostkowych.
- 17.** Napisz i przetestuj metodę szukającą największego wspólnego dzielnika o sygnaturze `int NWD(int a, int b);` (<http://pl.wikipedia.org/wiki/NWD>), korzystając z algorytmu Euklidesa ([http://pl.wikipedia.org/wiki/Algorytm\\_Euklidesa](http://pl.wikipedia.org/wiki/Algorytm_Euklidesa)). Przygotuj testy jednostkowe sprawdzające poprawność metody dla z góry ustalonych i dla losowo wybranych argumentów.
- 18.** Przygotuj program, w którym tablica liczb całkowitych porządkowana jest metodą sortowania bąbelkowego. Po każdym kroku wyświetlaj zawartość tablicy w konsoli.
- 19.** Zmodyfikuj projekt z poprzedniego zadania tak, żeby metody służące do sortowania i wyświetlania tablicy były metodami parametrycznymi działającymi dla dowolnego parametru T implementującego interfejs `IComparable` i osobno `IComparable<T>`.
- 20.** Zdefiniuj klasę o nazwie `KlasaZLicznikiem`, która w statycznym prywatnym polu przechowuje liczbę utworzonych instancji. Udostępnij tę liczbę w publicznej własności.
- 21.** Zdefiniuj klasę A i dziedziczącą z niej klasę B. W klasie A zdefiniuj dwie metody wirtualne o nazwach M1 i M2. W klasie potomnej nadpisz metodę M1 i ukryj metodę M2. We wszystkich metodach umieść polecenia wyświetlające łańcuch postaci `NazwaKlasy.NazwaMetody`. Następnie utwórz instancje klas A i B następującymi poleceniami:

```
A aa = new A();
A ab = new B();
B bb = new B();
B ba = new A();
```

i wywołaj na ich rzecz metody M1 i M2. Sprawdź, jakie komunikaty będą wyświetlane.

- 22.** W klasie A i B zdefiniuj konstruktory domyślne oraz konstruktory pozwalające na określenie koloru, w jakim w konsoli wyświetlane są napisy w metodach M1 i M2 (konieczne będzie zdefiniowanie pola, które będzie przechowywać kolor). W klasie B konstruktor z argumentem powinien wywoływać konstruktor z argumentem z klasy A (bez samodzielnego modyfikowania nowego pola).
- 23.** Napisz klasę Timer z właściwościami *auto-implemented* `Interval` typu `int` i `Enabled` typu `bool`. Jeżeli właściwość `Enabled` równa jest `true`, co liczbę milisekund określoną przez właściwość `Interval` wywołuj metodę przekazaną przez konstruktor (jego argumentem ma być delegacja).
- 24.** Przygotuj dwie metody przyjmujące jako argument delegacje typu `double f(double x);` i obliczające pierwszą i drugą pochodną funkcji f w zadanym punkcie x (drugi argument metod):

```
delegate double Funkcja(double x); //Func<double,double>
private static double pierwszaPochodna(Funkcja f, double x, double h);
private static double drugaPochodna(Funkcja f, double x, double h);
```

Do obliczania pochodnych użyj wzorów:

$$f'(x) = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} \text{ i } f''(x) = \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{\Delta x^2}.$$

**25.** Przygotuj metodę o sygnaturze

```
private static double całkuj(Funkcja f,double a,double b,uint n);
```

która oblicza całkę funkcji  $f$  w zakresie  $[a,b]$ , dzieląc go na  $n$  przedziałów.

**26.** Przygotuj klasę o nazwie `AutoSumowanie` zawierającą:

- a) metodę `Dodaj`, w której podajemy jako argument liczbę typu `decimal` — w przypadku podania ujemnej wartości należy zgłosić wyjątek;
- b) własność tylko do odczytu o nazwie `Suma` pozwalającą na odczytanie sumy wartości podanych metodą `Dodaj`;
- c) konstruktor pozwalający na ustalenie wartości początkowej oraz wartości (limitu), której przekroczenie powinno być sygnaлизowane.

**27.** Stwórz projekt aplikacji konsolowej. Do jego rozwiązania dodaj projekt biblioteki DLL lub PCL z klasą `RownanieKwadratowe`, która ma:

- a) prywatne pola `a`, `b` i `c` typu `double` (współczynniki trójmianu) — ich własność powinna być ustalana dzięki argumentom konstruktora i niemożliwa do późniejszej zmiany;
- b) prywatne pole `delta_pierwiastek` typu `double?` przechowujące pierwiastek równania kwadratowego obliczanego w konstruktorze;
- c) publiczną własność `public bool CzyPierwiastkiIstnieja { get; private set; }`, której wartość ustalana jest w konstruktorze na podstawie wartości wyróżnika równania kwadratowego (delta);
- d) prywatną metodę `delta` obliczającą wyróżnik;
- e) publiczne własności `X1` i `X2` zwracające pierwiastki równania, jeżeli te istnieją (lub zgłaszające wyjątki, jeżeli wyróżnik jest mniejszy od zera).

**28.** Do rozwiązania z poprzedniego projektu dodaj projekt testów jednostkowych i zdefiniuj testy sprawdzające wartości pierwiastków dla współczynników, dla których istnieją dwa różne pierwiastki, podwójny pierwiastek lub równanie nie ma rozwiązań.

**29.** Zmodyfikuj klasę `Ułamek` z rozdziału 4. w taki sposób, żeby implementowała interfejs `IConvertible`.

**30.** Przygotuj własną implementację typów `Lazy<>` i `Nullable<>`.

**31.** Przygotuj singleton z opóźnioną inicjacją (rozdział 4.), ale oparty na typie `Lazy<>`.

**32.** Przygotuj zbiór klas realizujący system bankowy (jeden bank). Potrzebne są przynajmniej dwie klasy. Klasa `Konto`, która zawiera numer konta, saldo

i metody pozwalające na wpłatę i wypłatę podanej w argumencie kwoty, oraz klasa Przelew, która przechowuje wszystkie informacje potrzebne do przelania pieniędzy z jednego konta na drugie (wzorzec *command*).

33. Przygotuj strukturę Kąt. Konstruktory powinny umożliwiać inicjowanie stanu wartością wyrażoną w radianach (double), stopniach (double), stopniach ( $4 \times \text{int } d:m:s:ms$ ) lub godzinach ( $4 \times \text{int } h:m:s:ms$ ). Struktura powinna także umożliwiać odczyt wartości w takich jednostkach. Należy uwzględnić możliwość redukcji do wskazanego zakresu: [0,360] lub [-180,180]. Przygotuj również operatory dodawania i odejmowania oraz mnożenia przez liczbę double. Nadpisz metodę string `ToString()` oraz zdefiniuj metodę `ToString(string format)`.
34. Przygotuj zbiór klas zarządzających dwoma windami w bloku. Obie windy są sterowane jednym panelem przycisków na piętro. Jak zoptymalizować zarządzanie windami, aby przyjazd windy do wezwania był jak najszybszy? Jakie przyciski powinien posiadać panel?
35. Napisz program przeszukujący tysiącelementową tablicę w poszukiwaniu minimalnej i maksymalnej wartości. Przyspiesz działanie owego programu, korzystając z pętli `Parallel.For` lub `Parallel.ForEach`.
36. Przygotuj klasę implementującą drzewo z elementami typu int i dowolną ilością dzieci w każdym węźle. Przygotuj metodę przeszukującą drzewo w taki sposób, żeby przeszukiwanie każdego węzła-dziecka było uruchamiane w osobnym zadaniu. Użyj do tego pętli `Parallel.For`.

## Windows Forms

Wszystkie zadania z tej części należy wykonać w projektach typu Windows Forms Application. Można korzystać z metod i klas przygotowanych w zadaniach z poprzedniej części (np. metody `Silnia` lub klasy `RownanieKwadratowe`).

1. Przygotuj aplikację, w której na formie znajduje się pole edycyjne (TextBox), etykieta (Label) oraz przycisk (Button). Po kliknięciu przycisku należy obliczyć silnię liczby wpisanej w polu tekstowym. Do „parsowania” ciągu znaków do liczby należy użyć metody `byte.Parse`. Metoda ta zgłasza wyjątek, jeżeli łańcuch nie zawiera poprawnej liczby typu byte. Należy obsłużyć ten wyjątek konstrukcją `try..catch`, wyświetlając komunikat w razie błędu (`MessageBox.Show`). Wynik obliczeń wyświetli na etykiecie (Label).
2. Przygotuj aplikację, która utworzy dynamicznie tabelę szesnastu przycisków i rozmieści je na formie w postaci siatki  $4 \times 4$ . Referencje do przycisków należy przechować w dwuwymiarowej tabeli. Poszczególne przyciski po kliknięciu powinny zmieniać etykietę na *Hello!*.
3. W aplikacji Kolory z rozdziału 8. należy umożliwić wybór predefiniowanego koloru za pomocą klawiszy funkcyjnych. Na szczycie okna umieść panele z numerami klawiszy funkcyjnych ( $F2 - F12$ ) z odpowiednimi kolorami tła. Należy unikać powielania kodu metod.

4. W aplikacji Notatnik.NET z rozdziału 9. do menu dodaj ikony. Można użyć obrazów dołączonych do Visual Studio w katalogu *C:\Program Files\Microsoft Visual Studio 11.0\Common7\VS2012ImageLibrary\1033* lub dostępnych pod adresem: <http://www.famfamfam.com/lab/icons/silk/>.
5. W tej samej aplikacji przygotuj pasek narzędzi zawierający pozycje z menu *Plik*.
6. Pytanie o zapisanie tekstu do pliku wyświetlane w aplikacji Notatnik.NET przed jej zamknięciem nie ma sensu, jeżeli ów tekst nie został zmieniony. Dlatego należy w klasie *Form1* zdefiniować pole *tekstZmieniony* typu *bool*, którego wartość powinna być ustalona na *true* w przypadku modyfikacji zawartości notatnika (pomocne będzie zdarzenie *TextChanged* komponentu *TextBox*). Jeżeli podczas zamykania formy jest ono równe *false*, komunikat z pytaniem nie powinien zostać pokazany. Zaimplementuj ten pomysł.
7. W projekcie Notatnik.NET zmodyfikuj metodę *printDocument1\_PrintPage* w taki sposób, żeby zmniejszyć o 4 liczbę linii drukowanych na stronie, a dostawić nagłówek i stopkę. Nagłówek powinien zawierać nazwę pliku, a stopka — numer strony.

8. Przygotuj dwie metody rozszerzające dla klasy *TextBox* (kontrolka Windows Forms), które pozwolą na wczytywanie i zapisywanie jej zawartości z pliku tekstowego (*LoadFromFile* i *SaveToFile*).
9. W projekcie Notatnik.NET używamy metod pozwalających na zapis i odczyt plików tekstowych z dysku. Przygotuj ich wersje asynchroniczne, czyli tworzące zadania:

```
public static Task<string[]> CzytajPlikTekstowyAsync(string nazwaPliku);
public static Task ZapiszDoPlikuTekstowego(string nazwaPliku, string[] tekst);
```

Użyj ich w programie, korzystając z operatora *await*. Postaraj się na poziomie interfejsu zapobiec próbie jednoczesnego wczytania i zapisania tekstu do tego samego pliku (przełączając własność *Enabled* kontrolek). Spróbuj w podobny sposób przygotować kod służący do drukowania zawartości notatnika.

10. Przygotuj klasę, która na podobieństwo obecnej w projektach Windows Forms klasy *Settings* umożliwia zapisywanie i odczytywanie do plików XML położenia i wielkości okna (właściwości *Left*, *Top*, *Width*, *Height*) oraz jego etykiety (właściwość *Text*). Klasa powinna odczytywać ustawienia w konstruktorze, a zapisywać po wywołaniu metody *Save*.
11. Do aplikacji *Kolory* (projekt z rozdziału 8.) dodaj zapisywanie składowych RGB koloru ustalonego za pomocą suwaków, korzystając z ustawień aplikacji (klasa *Settings*).
12. Pobierz ze strony NBP aktualny kurs walut w pliku XML ([http://www.nbp.pl/home.aspx?f=/kursy/instrukcja\\_pobierania\\_kursow\\_walut.html](http://www.nbp.pl/home.aspx?f=/kursy/instrukcja_pobierania_kursow_walut.html), zobacz rozdział 12.). Następnie przygotuj aplikację Windows Forms o nazwie *Kantor*, która zawiera:
  - a) klasę *KursyWalutNBP* odczytującą z pobranego pliku XML oficjalne kursy walut NBP i udostępniającą właściwości pozwalające odczytać kursy dla dolara amerykańskiego (USD) i euro;

- b) klasę *KursyWalutKantoru*, która pozwala na przeliczanie między złotymi, euro i USD na podstawie odczytanych w poprzedniej klasie kursów NBP przy ustalonych „widelkach” (procent „widełek” jest argumentem konstruktora klasy *KursyWalutKantoru*).

Przygotuj podstawowe testy jednostkowe dla obu klas. Należy także przygotować interfejs Windows Forms, który pozwala na obsługę kupna i sprzedaży USD i euro przez kantor.

- 13.** Korzystając z klasy o nazwie *AutoSumowanie* (zadanie z pierwszej części), przygotuj aplikację Windows Forms o nazwie *AsystentSklepowy* z jednym oknem zawierającym kontrolki *Label*, *TextBox* i *Button*. Po kliknięciu przycisku kwota wpisana do pola tekstowego powinna być dodawana do sumy wyświetlanej w kontrolce *Label* (to zadanie powinna realizować klasa *AutoSumowanie*). Uwzględnij możliwość zgłoszania wyjątków przez metodę *AutoSumowanie.Dodaj*. Aplikacja powinna powiadamiać o przekroczeniu założonego limitu.
- 14.** W aplikacji *AsystentSklepowy* z poprzedniego zadania dodaj:
- a) mechanizm zapamiętywania stanu z użyciem ustawień aplikacji — wyświetlana w programie suma powinna być przywracana po ponownym uruchomieniu aplikacji;
  - b) korzystając z półtona formy (rozdział 9.), narysuj słupek, którego wysokość odpowiada stosunkowi bieżącej wartości sumy do założonego limitu — słupek powinien być zielony, jeżeli suma jest mniejsza od limitu, a czerwony po jego przekroczeniu;
  - c) moduł przechowujący kolejne dodawane kwoty z możliwością drukowania ich zestawienia i ostatecznej sumy.
- 15.** Przygotuj interfejs Windows Forms umożliwiający korzystanie z klasy *RównanieKwadratowe* z pierwszej części zadań. Interfejs powinien zawierać pola tekstowe umożliwiające podanie współczynników *a*, *b* i *c* oraz kontrolki wyświetlające wartości obliczonych pierwiastków.
- 16.** Przygotuj projekt biblioteki DLL z zaprojektowaną przez siebie kontrolką (projekt typu Windows Forms *Control Library*) wyświetlającą bieżącą datę i godzinę, aktualizowaną domyślnie co pół sekundy. Do wyświetlania daty użyj kontrolki *Label*, a do cyklicznego odświeżania użyj komponentu *Timer*. Przygotuj własność typu *int* pozwalającą na określenie, co ile czas ma być zmieniany, oraz zdarzenie informujące o pełnych godzinach. W aplikacji testującej wykorzystaj to zdarzenie do odtworzenia jakiegoś dźwięku.
- 17.** Zmodyfikuj poprzednią kontrolkę w taki sposób, żeby oprócz kontrolki *Label* wyświetlającej tym razem tylko datę (bez godziny) na kontrolce widoczny był zegar analogowy z dwoma lub trzema wskaźówkami. Można go łatwo przygotować, korzystając ze zdarzenia *Paint* i dostępnego w nim obiektu *e.Graphics* (analogicznie jak rysowanie na obszarze klienta formy).
- 18.** Odtwórz czynności z rozdziałów od 14. do 17. dla bazy danych Microsoft Access.



# Skorowidz

## A

ADO.NET, 267, 307, 309  
algorytm Euklidesa, 101  
alias, 115  
aplikacja  
    błąd, 27  
    debugowanie, 27, 30, 31  
        obsługa wyjątków, 33  
Entity Framework, 162  
ikona w zasobniku, 217, 219  
inicjacja asynchroniczna, 332  
interfejs, *Patrz:* interfejs  
    aplikacji  
konsolowa, 15, 26, 76  
menu  
    Edycja, 204  
    główne, 194  
    Plik, 194, 196, 202, 205  
    Wikad, 203  
projekt, 190  
punkt wejściowy, 17  
środowisko, 23  
tryb pojedynczego wątku, 181  
uruchamianie  
    bez debugowania, 18  
    breakpoint, 30  
    do kurSORA, 30  
    obsługa wyjątków, 35  
    w trybie śledzenia, 28  
    z debugowaniem, 19  
ustawienia, 222, 223, 224  
Windows Forms, 26, 76, 171, 176, 257  
interfejs, *Patrz:* interfejs  
    aplikacji  
    Windows Forms  
Windows Phone, 39, 141

Windows Store, 39, 40,  
    162, 165  
WPF, 26, 39  
z bazą danych SQL Server,  
    270  
zamykanie, 196, 197  
auto-implemented properties,  
*Patrz:* właściwość domyślnie  
implementowana

## B

balloon, *Patrz:* dymek  
baza danych, 40, 267, 323  
aktualizacja, 283  
Microsoft Access, 309  
obiektowa, 267  
odwzorowanie  
    obiektowo-relacyjne,  
        *Patrz:* ORM  
    w klasie .NET, 267  
rekord, 285  
SQL Server, 267, 271, 279,  
    309  
SQL Server Compact, 279  
biblioteka, 72  
ASP.NET, 193  
DLL, 137, 144  
    przenośna, *Patrz:*  
        biblioteka PCL  
    referencja, 139  
Entity Framework, 193, 326  
EntityFramework.SqlServer.d  
    ll, 326  
Forms.dll, 21  
kontrolek, 171  
łączenie  
    dynamiczne, 140  
    statyczne, 137, 140  
PCL, 137, 140, 144

STL, 111  
System.Data.Linq.dll, 280  
TPL, 159  
WCF, 193  
Windows Forms, 171, 193  
    kontrolka, 189  
WPF, 171, 193  
biblioteka DLL, 138  
blok, 79  
boxing, *Patrz:* pudelkowanie  
buforowanie podwójne, 227

## C

callback, *Patrz:* funkcja zwrotna  
Caller Information, 93  
CAS, 43  
CIL, 41, 43  
CLR, 39, 40, 43  
CLS, 43  
Code Access Security, *Patrz:*  
    CAS  
Common Intermediate  
    Language, *Patrz:* CIL  
Common Language Runtime,  
    *Patrz:* CLR  
Common Language  
    Specification, *Patrz:* CLS  
Common Type System, *Patrz:*  
    CTS  
connection string, *Patrz:*  
    łańcuch konfigurujący  
    połączenie, *Patrz:* łańcuch  
    połączenia  
CTS, 43  
czcionka, 203

**D**

dane  
 baza, *Patrz*: baza danych  
 filtrowanie, 302, 319  
 łączenie zbiorów, 252  
 modyfikacja, 276, 295  
 pobieranie, 276, 283  
 sortowanie, 301, 319  
 typ, *Patrz*: zmienna typ  
 źródło, 298, 299, 304, 307,  
 334, 336  
 kreator, 297  
 data source, *Patrz*: dane źródło  
 delegacja, 61, 64, 95, 118  
 DLR, 40, 43  
 drag & drop, *Patrz*:  
 przeciagnij i upuść  
 drukowanie, 205, 208, 209  
 długich linii, 210, 212  
 w tle, 213  
 drzewo, 81  
 dymek, 219  
 Dynamic Language Runtime,  
*Patrz*: DLR  
 dyrektywa  
 #define, 79  
 #endregion, 79, 80  
 #if, 78  
 #region, 79, 80  
 dziedziczenie preprocesora, 77, 78  
 dziedziczenie, 117, 120, 122,  
 129, 131  
 wielokrotne, 134

**E**

edytor  
 Create Unit Test, 144  
 kodu, 17, 80  
 O/R Designer, 281, 286, 287  
 relacji, 292  
 EF, *Patrz*: Entity Framework  
 ekran powitalny, 214, 215  
 entity class, *Patrz*: klasa encji  
 Entity Framework, 268, 307,  
 323, 326, 341  
 entry point, *Patrz*: aplikacja  
 punkt wejściowy  
 Euklidesa algorytm, *Patrz*:  
 algorytm Euklidesa  
 exception, *Patrz*: wyjątek  
 extension method, *Patrz*:  
 rozszerzenie

**F**

FIFO, 88  
 FILO, 88  
 formularz, 302, 316  
 funkcja, 56  
 Average, 250  
 GetSystemMetrics, 238  
 haszująca, 108  
 Max, 250  
 Min, 250  
 Sum, 250  
 trygonometryczna, 159  
 zwrotna, 64

**G**

garbage collector, *Patrz*:  
 odśmiecacz  
 generator liczb pseudolosowych,  
 73  
 generic types, *Patrz*: zmienna  
 typ ogólny  
 graphical user interface, *Patrz*:  
 GUI  
 GUI, 171, 231

**H**

Hejlsberg Anders, 39, 268

**I**

indeksator, 95  
 inheritance, *Patrz*: dziedziczenie  
 instrukcja  
 break, 74, 101  
 continue, 74  
 DELETE, 270  
 INSERT, 270  
 return, 76  
 SELECT, 269  
 using  
 System.Data.Linq.Mapping,  
 280  
 warunkowa  
 if..else, 72  
 wybór switch, 73  
 IntelliSense, 18, 72, 123, 331  
 tryb, 18  
 interfejs, 95, 111, 134, 135  
 aplikacji, 189, 190  
 kontrolka, *Patrz*: kontrolka  
 Windows Forms, 171, 172

graficzny użytkownika,  
*Patrz*: GUI  
 IComparable, 84, 109, 114  
 IConvertible, 100  
 IDictionary, 87  
 IDirectory, 25  
 IEnumerable, 247, 248, 250  
 implementacja, 109, 131  
 przez typ ogólny, 114  
 master-details, 334  
 Modern UI, 165  
 tworzenie, 338  
 użytkownika, 298, 300

**J**

Java, 39  
 język  
 C#, 17, 39, 40, 95  
 wielkość liter, 18  
 C++, 17, 39  
 dynamiczny, 40  
 Java, *Patrz*: Java  
 Python, 40  
 Ruby, 40  
 SQL, *Patrz*: SQL  
 Transact SQL, *Patrz*: język  
 T-SQL  
 T-SQL, 269, 279  
 XML, 255  
 atrybut, 256  
 deklaracja, 255, 258  
 dokument, 255  
 element, 256  
 komentarz, 256, 258  
 zapytań, *Patrz*: LINQ  
 JIT, 43  
 Just-In-Time, *Patrz*: JIT

**K**

katalog  
 domowy, 25  
 specjalny, 24  
 klasa, 15, 66, 67  
 abstrakcyjna, 124, 125,  
 127, 135  
 Array, 65, 81  
 bazowa, 98, 100, 112, 113,  
 115, 120, 123, 127, 134  
 bazująca na typie, 110  
 BindingSource, 301

- DataContext, 268, 279, 281, 286, 294  
DataSet, 307, 309  
definiowanie, 95  
encji, 279, 280, 281, 286, 287, 293, 323  
Enum, 54  
Environment, 23, 24  
Graphics, 205, 225  
HttpClient, 165  
instancja, *Patrz*: obiekt  
Lazy, 55  
List, 81  
MessageBox, 21  
opakowująca, 45  
ORM, 287, 309, 323  
Panel, 175  
Parallel, 160  
ParallelLoopState, 161  
pole, *Patrz*: pole  
potomna, 117, 120, 123, 127  
PrivateObject, 147  
Queue, 87  
Random, 73  
SoundPlayer, 221  
Stack, 87  
statyczna, 131  
    dziedziczenie, 131  
    instancja, 131  
StorageFile, 165  
StreamReader, 165  
StreamWriter, 165  
String, 51, 52  
StringBuilder, 53  
SystemColor, 189  
Task, 159  
Trace, 93  
TrackBar, 175  
WCF, 165  
właściwość, *Patrz*:  
    właściwość  
XDocument, 258  
XmlReader, 165  
klawiatura, 187  
    odeczytywanie danych, 20  
klawisz, 20  
    Alt, 187  
    Ctrl, 187  
    Esc, 187  
    F10, 28, 29  
    F11, 28, 29  
    F4, 174  
    F5, 19, 184  
    F7, 176
- F9, 30  
Shift, 175, 187  
specjalny, 187  
Tab, 180  
klawisze skrótów  
    debugera, 29  
    edytora, 19  
klucz — wartość, 87  
kod  
    maszynowy, 43  
    oparty na refleksji, 72  
    pośredni, *Patrz*: CIL  
    źródłowy, 79, 80  
kolejka, 81, 87, 88  
kolekcja, 44, 75, 81, 85  
    Dictionary, 87  
    List, 85  
    SortedDictionary, 87  
    SortedList, 87, 88  
kompilacja, 80  
    atrybut, 80  
    dwustopniowa, 40  
    warunkowa, 78, 79  
kompilator, 40, 43  
    jako usługa, 40  
komponent, *Patrz*: kontrolka  
konstruktor, 129, 177  
    bezargumentowy, 98, 130  
    domyślny, 98, 104, 130, 132  
    prywatny, 132  
kontrolka, 175, 189, 290  
    ComboBox, 302, 303  
    DataGridView, 283, 290, 291, 300, 311, 330  
    DataSet, 267, 307  
    Label, 214  
    panel, 175  
    suwak, 175  
    TextBox, 302  
    TreeView, 261  
    zdarzenie domyślne, 196  
kreator  
    modelu danych EDM, 327  
    źródła danych, 297  
kursor myszy, 231, 234, 236  
    kształt, 235, 236  
    położenie, 238
- LINQ to SQL, 268, 279, 280, 281, 282, 284, 285, 294, 297, 307  
LINQ to XML, 257, 259  
Linux, 40  
lista, 81, 85  
    dwukolumnowa, 87  
    dysków logicznych, 26  
    szablonów, 100  
    z kluczem, 88  
literał liczbowy,  
    *Patrz*: stała liczbową
- L**
- łańcuch, 51, 53  
konfigurujący połączenie, 308  
połączenia, 271  
znaków, 20, 199
- M**
- makro, 78  
metoda, 44, 95  
    abstrakcyjna, 125, 127  
    anonimowa, 64, 65  
Append, 54  
argument, 57, *Patrz też*:  
    metoda parametr  
asynchroniczna, 165  
atrybut  
    ClassCleanup, 150  
    ClassInitialize, 150  
    TestCleanup, 150  
    TestInitialize, 150  
Break, 161  
CompareTo, 84  
Console, 18, 21  
CreateDatabase, 286  
DeleteDatabase, 286  
DoDragDrop, 231, 233, 234, 236, 238  
Equals, 106  
ExecuteCommand, 286  
GetEnvironmentVariable, 25  
GetEnvironmentVariables, 25  
GetHashCode, 106, 107  
GetValueOrDefault, 67  
głowa, 57  
Insert, 53  
LoadAsync, 332  
MessageBox, 200  
Min, 65
- L**
- Language Integrated Query,  
    *Patrz*: LINQ, zapytanie LINQ  
LINQ, 247, 248, 249  
LINQ to Entities, 329

metoda  
 nadpisywanie, 123, 126,  
 127, 180  
 OrderBy, 248  
 Parallel.For, 161  
 parametr, 57, 113, *Patrz też:*  
     metoda argument  
     nazwany, 59  
     opcjonalny, 58  
     tablica, 89  
     typ referencyjny, 58, 60  
     typ wartościowy, 58, 60  
     wartość domyślna, 46, 58  
     zwracana wartość, 59  
 PrivateObject, 147  
 przeciągnij i upuść, *Patrz:*  
     przeciągnij i upuść  
 przeciążanie, 57  
 przesłanianie, 124  
 Read, 20  
 ReadKey, 20  
 ReadLine, 20  
 Remove, 53  
 Replace, 53  
 rozszerzająca, 65, 117, 118,  
 248, 250, 332  
 SaveChangesAsync, 333  
 Select, 248  
 SetError, 21  
 SetIn, 21  
 SetOut, 21  
 Show, 21  
 Single, 250  
 Sort, 84  
 statyczna, 17, 18, 56, 99  
 Stop, 161  
 SubmitChanges, 283, 294  
 sygnatura, 57  
 Where, 248  
 wirtualna, 126, 127  
 WriteLine, 20, 21  
 wywołanie, 93  
 zdarzeniowa, 95, 182, 183,  
 185, 196  
 testowanie, 183  
 wywoływanie z poziomu  
     kodu, 186  
 zwracana wartość, 59  
 zwrotna, 166  
 mock object, *Patrz:* zaślepkę  
 modyfikator  
     async, 165  
     const, 99  
     event, 63

explicit, 109  
 implicit, 109  
 internal, 97  
 new, 124  
 override, 124, 126  
 private, 97, 123  
 protected, 97  
 protected internal, 97  
 public, 97, 123  
 readonly, 98, 99  
 sealed, 123  
 static, 56, 98, 99  
 virtual, 124, 127  
 MSIL, *Patrz:* IL

**N**

nadawca, 186  
 największy wspólny dzielnik, 101  
 namespace, *Patrz:* przestrzeń  
     nazw  
 nHibernate, 267, 307  
 NUnit, 143

**O**

obiekt, 66, 95  
 anonimowy, 249  
 formy, 173  
 inicjacja, 92  
 klonowanie, 53  
 kopiowanie, 66  
 sortowanie, *Patrz:* sortowanie  
     zastępczy, 157  
 object-relational mapping,  
*Patrz:* ORM  
 obszar powiadamiania, *Patrz:*  
     zasobnik  
 odśmiecacz, 67, 96, 180  
 odwzorowanie obiektowo-  
     relacyjne, *Patrz:* ORM  
 okno, 21  
     aplikacji, 173  
     ikona, 191  
     Locals, 31  
     nazwa, 191  
     Properties, 174  
     Toolbox, 174, 175  
     Watch, 31  
     własności, 174  
 OLE DB, 267  
 operator, 47, 48, 49  
     !=, 106, 107

**P**

pamięć  
 wyciek, 67  
 zarządzanie, 39  
 Parallel Extensions, 159  
 pasek stanu, 192, 198  
 pętla, 74  
     do..while, 74  
     for, 74  
         równoległa, 159, 161  
         foreach, 75, 83  
         while, 74  
 platforma, 142  
     .NET, 39, 40, 140  
         historia, 41  
         wersja, 41, 193  
         wydajność, 96  
     Mono, 40, 137

WinRT, 40, 140  
Xbox, 141  
XNA, 40, 140

plik

.bmp, 214  
.dll, 41  
.emf, 214  
.exe, 41, 137  
.gif, 214  
.ico, 191

.jpeg, 214  
.png, 214  
.wav, 220, 221  
.wmf, 214

AssemblyInfo.cs, 80  
dźwiękowy, 220  
przenoszenie, 242  
tekstowy, 199, 202  
wybór, 200  
XML, 255, 257, 260  
modyfikowanie, 264  
przenośność, 262

pole, 95

deklaracja, 98  
 prywatne, 97, 147, 180  
 statyczne, 98  
 tylko do odczytu, 98

polimorfizm, 127

Portable Class Library, *Patrz:*  
 biblioteka PCL, *Patrz:*  
 biblioteka PCL

preprocesor

dyrektywa, *Patrz:* dyrektywa  
 preprocesora  
 stała, *Patrz:* stała  
 preprocesora

procedura składowana, 276, 277,  
 294, 295, 327, 333

testowanie, 294

programowanie

asynchroniczne, 40, 162  
 dynamiczne, 43  
 obiektowe, 95, 119  
 wizualne, 172  
 współbieżne, 40, 159  
 zdarzeniowe, 196

projektowanie wizualne, 308  
 przeciagnij i upuść, 231, 237, 238  
 wiele elementów, 241

przestrzeń nazw, 17

System.Collections, 81  
 System.Collections.Specialized,  
 81

System.Data.Entities, 333

System.Data.Entity, 331  
 System.Data.Linq.Mapping,  
 280  
 System.Entity.Data, 329  
 System.Linq, 248  
 System.Xml.Linq, 257  
 pudelkowanie, 68

## Q

queue, *Patrz:* kolejka

## R

relacja, 292  
 jeden do wielu, 304  
 ReSharper, 72  
 rozszerzenie, 65, 117, 118, 248,  
 250, 332

## S

schowek, 204  
 sender, *Patrz:* nadawca  
 siatka DataGridView, *Patrz:*  
 kontrolka DataGridView  
 singleton, 123, 131  
 słownik, 81, 87, 88  
 słowo kluczowe  
 abstract, 125  
 break, 74  
 checked, 33, 37  
 class, 97  
 continue, 74  
 default, 46  
 delegate, 61  
 dynamic, 69, 72  
 event, 62, 63  
 namespace, 17  
 out, 61  
 override, 100  
 params, 89  
 ref, 61  
 return, 59  
 struct, 66, 97  
 this, 98, 118, 180  
 throw, 77  
 try, 75  
 using, 17, 200  
 var, 47, 72, 247  
 void, 57  
 yield, 90  
 sortowanie, 84

splash screen, *Patrz:* ekran  
 powitalny  
 SQL, 247, 269, 277  
 stack, *Patrz:* stos  
 stała, 66

liczbową, 46  
 preprocesora, 78  
 stored procedure, *Patrz:*  
 procedura składowana

stos, 66, 87, 88, 96  
 wywołań, 33  
 strongly typed, 280  
 struktura, 66, 67, 96, 120  
 bazująca na typie, 110  
 strumień  
 błędów, 21  
 przekierowanie, 21  
 standardowy wyjście, 20  
 StringReader, 208  
 szablon, 100, 111

## Ś

środowisko CLR, *Patrz:* CLR

## T

tabela, 327  
 aktualizacja, 283  
 dodawanie do bazy danych,  
 272  
 edycja danych, 274  
 łączenie, 292  
 prezentacja w formularzu,  
 302  
 relacja, 310  
 tablica, 81, 85, 109  
 jako argument metody, 89  
 łańcuchów, 199  
 wielowymiarowa, 84  
 Target Framework, 193  
 Task Parallel Library, *Patrz:*  
 biblioteka TPL  
 technologia LINQ, *Patrz:* LINQ  
 test  
 funkcjonalny aplikacji, 143  
 integracyjny, 143  
 jednostkowy, 143, 144, 152  
 konstruktora, 147  
 pola prywatnego, 147  
 projekt, 144  
 tworzenie, 145  
 uruchamianie, 146  
 wyjątków, 148

## test

- losowy, 151
- metody zdarzeniowej, 183
- operatorów arytmetycznych, 155
- procedury składowanej, 294
- systemowy, 143
- wydajnościowy, 143
- typ, *Patrz:* zmienna typ

**U**

- użytkownika profil
- katalog domowy, 25
- katalog specjalny, 24

**W**

- widok, 16, 277, 291, 327, 333
- Windows, 39
- Windows Presentation Foundation, *Patrz:* biblioteka WPF
- właściwość, 95, 102
  - domyślnie implementowana, 103, 104
- wyjątek, 75
  - DivideByZeroException*, 50
  - filtrowanie, 200
  - InvalidOperationException*, 109
  - nieobsłużony, 76
  - obsługa, 76
  - OverflowException*, 37
  - zgłaszanie, 77
- wyrażenie lambda, 64, 65, 248, 250

**Z**

- zapytanie
  - LINQ, 39, 65, 72, 90, 248, 257, 263, 279, 329
  - SQL, 248, 267, 277
  - T-SQL, 279
- zintegrowane z językiem programowania, *Patrz:* LINQ, zapytanie LINQ
- zasobnik, 217
- zaślepka, 157
- zdarzenie, 62, 95, 186, 196
  - domyślne, 196
  - DragDrop*, 231, 234
  - DragEnter*, 231
  - DragOver*, 231, 234, 235
  - FormClosed*, 257
  - KeyPress*, 187
  - MouseDown*, 231
  - Paint*, 208, 225
- zintegrowany język zapytań, *Patrz:* LINQ
- zmienna, 44
  - całkowita, 44, 49
  - deklaracja, 44
  - globalna, 131
  - inicjowanie leniwe, 55
  - int, 33
  - łańcuchowa, 44
  - null, 67, 68
  - obiektowa, 180
  - środowiskowa
    - USERPROFILE*, 25
  - typ, 44, 45, 47
    - anonimowy, 119
    - Delegacja, 61
    - dynamiczny, 47, 69, 71
    - Graphics*, 226
    - konwersja, 49, 100, 108, 127
    - Nullable*, 67

- object, 71
- ogólny, 110, 111, 116
- Panel, 180
- parametryczny, *Patrz:*
  - zmienna typ ogólny
  - referencyjny, 47, 58, 60, 66, 67, 71, 83, 96, 120, 180
- Task, 165
- wartościowy, 47, 55, 58, 60, 66, 67, 68, 83, 96, 120
- wartość domyślna, 46, 58
- wyliczeniowy, 54
- XComment, 258
- XDeclaration, 258
- znakowy, 46
- zmiennoprzecinkowa, 44, 49, 50
- znak
  - `!=`, 106, 107
  - `+=`, 53, 54, 62
  - `<`, 106, 107
  - `<=`, 106
  - `=`, 62
  - `:=`, 44
  - `==`, 106, 107
  - `>`, 106, 107
  - `>=`, 106
  - `\b`, 51
- backslash, *Patrz:* znak lewego ukośnika
- cudzysłów, 18
- końca linii, 18, 51
- lewego ukośnika, 51
- łańcuch, *Patrz:* łańcuch znaków
- `\n`, *Patrz:* znak końca linii
- spacji, 18
- `\u`, 51
- zapytania, 68

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJE

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>