

Twoja przepustka do świata C#!

Wydanie III

Rusz głową!

C#

*Doskonały podręcznik
do nauki praktycznego
programowania
w C#, XAML i .NET*



Zarządzaj swoimi
obiektami dzięki
wykorzystaniu
abstrakcji
i dziedziczenia

Napisz w pełni
funkcjonalną
staromodną
grę wideo



Dowiedz się, w jaki
sposób programowanie
asynchroniczne
pozwoło Krystynie
spełnić wymagania
klientów

Poznaj
wszystkie
tajniki wzorca
Model-Widok
-Widok-Modelu
(MVVM)



Przekonaj się, jak Janek
zastosował kolekcje
i LINQ, by ujarzmić
niesforną kolekcję
komiksów

O'REILLY®

Jennifer Greene, Andrew Stellman

Helion



Tytuł oryginału: Head First C#, 3rd Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-246-8311-6

© 2014 Helion SA

Authorized Polish translation of the English edition of **Head First C#, 3rd Edition**

ISBN 9781449343507 © 2013 Jennifer Greene, Andrew Stellman

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

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

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

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/cshru3.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/cshru3>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

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

Spis treści (skrócony)

Wstęp	31
1. Zaczynj pisać programy w C#. <i>Napisz coś fajnego, i to szybko!</i>	43
2. To tylko kod. <i>Pod maską</i>	95
3. Obiekty: zorientuj się! <i>Tworzenie kodu ma sens</i>	143
4. Typy i referencje. <i>Jest 10:00. Czy wiesz, gdzie są Twoje dane?</i>	181
Laboratorium C# numer 1. <i>Dzień na wyścigach</i>	225
5. Hermetyzacja. <i>Co ma być ukryte... niech będzie ukryte</i>	235
6. Dziedziczenie. <i>Drzewo genealogiczne Twoich obiektów</i>	275
7. Interfejsy i klasy abstrakcyjne. <i>Klasy, które dotrzymują swoich obietnic</i>	329
8. Typy wyliczeniowe i kolekcje. <i>Przechowywanie dużej ilości danych</i>	385
9. Odczyt i zapis plików. <i>Zachowaj te bajty dla mnie!</i>	441
Laboratorium C# numer 2. <i>Wyprawa</i>	495
10. Projektowanie aplikacji dla Sklepu Windows z użyciem XAML. <i>Przenosząc swoje aplikacje na wyższy poziom</i>	517
11. Async, await i serializacja kontraktu danych. <i>Przepraszam, że przerywam</i>	565
12. Obsługa wyjątków. <i>Gaszenie pożarów nie jest już popularne</i>	599
13. Kapitan Wspaniały. <i>Śmierć obiektu</i>	639
14. Przeszukiwanie danych i tworzenie aplikacji przy użyciu LINQ. <i>Przejmij kontrolę nad danymi</i>	677
15. Zdarzenia i delegaty. <i>Co robi Twój kod, kiedy nie patrzysz</i>	729
16. Projektowanie aplikacji według wzorca MVVM. <i>Świetne aplikacje od zewnątrz i od środka</i>	773
Laboratorium C# numer 3. <i>Invaders</i>	835
17. Projekt dodatkowy! <i>Napisz aplikację Windows Phone</i>	859
A Pozostałości. <i>11 najważniejszych rzeczy, które chcieliśmy umieścić w tej książce</i>	873
Skorowidz	905

Spis treści (z prawdziwego zdarzenia)



Wstęp

Przygotuj się na C#. Właśnie sobie siedzisz i próbujesz się czegoś nauczyć, ale mózg wciąż powtarza Ci, że cała ta nauka *nie jest ważna*. Twój umysł mówi: „Lepiej wyjdź z pokoju i zajmij się ważniejszymi sprawami, takimi jak to, których dzikich zwierząt unikać, oraz to, że strzelanie z łuku na golasa nie jest dobrym pomysłem”. W jaki sposób oszukać mózg, tak aby myślał, że Twoje życie naprawdę zależy od nauki C#?

Dla kogo jest ta książka?	32
Wiemy, o czym myślisz	33
Metapoznanie: myślenie o myśleniu	35
Zmuś swój mózg do posłuszeństwa	37
Czego potrzebujesz do tej książki?	38
Przeczytaj to	39
Grupa korektorów technicznych	40
Podziękowania	41

Zacznij pisać programy w C#

1

Napisz coś fajnego, i to szybko!

Czy chcesz tworzyć wspaniałe programy naprawdę szybko? Wraz z C# dostajesz do ręki **potężny język programowania** i wartościowe narzędzie. Dzięki **Visual Studio IDE** do historii przejdą sytuacje, w których musiałeś pisać jakiś nędzny kod, by ponownie zapewnić prawidłowe działanie przycisku. I to nie wszystko. Dodatkowo będziesz mógł **skupić się na faktycznym wykonywaniu naprawdę fajnych programów**, zamiast starać się zapamiętać, który parametr metody odpowiadał za *nazwę* przycisku, a który za *wyświetlany na nim tekst*. Brzmi zachęcająco? Przewróć zatem stronę i przystąpmy do programowania.

Ich unikaj.



Och! Kosmici wciągają ludzi. Niedobrze!



Dlaczego powinieneś uczyć się C#	44
C# oraz Visual Studio ułatwiają wiele czynności	45
Co robić w Visual Studio...	46
Co Visual Studio robi w naszym imieniu...	46
Obcy atakują!	50
Tylko Ty możesz uratować Ziemię	51
Oto co masz zamiar napisać	52
Zacznij od pustej aplikacji	54
Określ wymiary siatki na stronie	60
Dodaj kontrolki do siatki	62
Używaj właściwości, by zmieniać wygląd kontroltek	64
To kontrolki sprawiają, że gra działa	66
Stworzyłeś scenę, na której będzie prowadzona gra	71
Czym zajmiesz się teraz?	72
Dodaj metody, które coś zrobią	73
Podaj kod metody	74
Dokończ metodę i uruchom program	76
Oto co zrobiłeś do tej pory	78
Dodaj liczniki czasu zarządzające rozgrywką	80
Popraw działanie przycisku Start	82
Uruchom program, by zobaczyć postępy w pracy	83
Dodaj kod obsługujący interakcję użytkownika z kontrolkami	84
Dotknięcie człowiekiem wroga kończy grę	86
Teraz już można bawić się grą	87
Zadbaj, by wrogowie wyglądali jak obcy	88
Dodaj ekran startowy i tytuł	89
Opublikuj swoją aplikację	90
Użyj programu Remote Debugger, by uruchomić aplikację na innym komputerze	91
Rozpocznij zdalne debugowanie	92

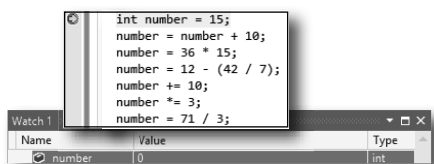
To tylko kod

2

Pod maską

Jesteś programistą, nie jedynie użytkownikiem IDE. IDE może wykonać za Ciebie wiele pracy, ale na razie jest to wszystko, co może dla Ciebie zrobić. Oczywiście istnieje wiele **powtarzalnych czynności** podczas pisania aplikacji i IDE okazuje się tu bardzo pomocne. Praca z nim to jednak *dopiero początek*. Możesz wycisnąć ze swoich programów znacznie więcej — **pisanie kodu C#** to właśnie droga, która doprowadzi Cię do tego celu. Kiedy osiągniesz mistrzowski poziom w kodowaniu, nie będzie *żadnej* rzeczy, której Twój program nie umiałby zrobić.

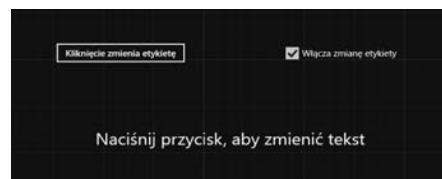
Kiedy robisz to...	96
...IDE robi to	97
Skąd się biorą programy	98
IDE pomaga Ci kodować	100
Anatomia programu	102
W tej samej przestrzeni nazw mogą być dwie klasy	107
Twoje programy używają zmiennych do pracy z danymi	108
C# używa znanych symboli matematycznych	110
Użyj debuggera, by zobaczyć, jak zmieniają się wartości zmiennych	111
Pętle wykonują czynność wielokrotnie	113
Instrukcje if/else podejmują decyzje	114
Utwórz aplikację od samego początku	115
Niech przycisk coś zrobi	117
Ustal warunki i sprawdź, czy są prawdziwe	118
Tworzenie klasycznych aplikacji Windows jest łatwe	129
Przepisz program jako klasyczną aplikację Windows	130
Twój program wie, gdzie zacząć	134
Możesz zmienić punkt wejścia programu	136
Kiedy zmieniasz coś w IDE, zmieniasz także swój kod	138



Za każdym razem, kiedy stworzysz nowy program, definiujesz dla niego przestrzeń nazw. W ten sposób jego kod jest odseparowany od innych klas platformy .NET.

Klasy zawierają **fragmenty** kodu Twojego programu (choćby istniały także bardzo małe aplikacje składające się z tylko jednej klasy).

Klasa posiada jedną lub więcej metod. Twoje metody zawsze będą umieszczane **wewnątrz klas**, a każda z nich będzie się składała z instrukcji i wyrażeń — jak te, które do tej pory widziałeś.



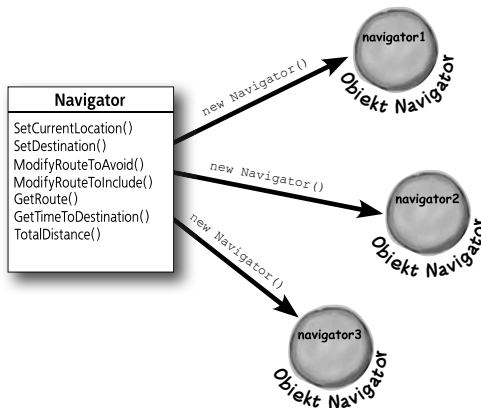
Obiekty: zorientuj się!

3

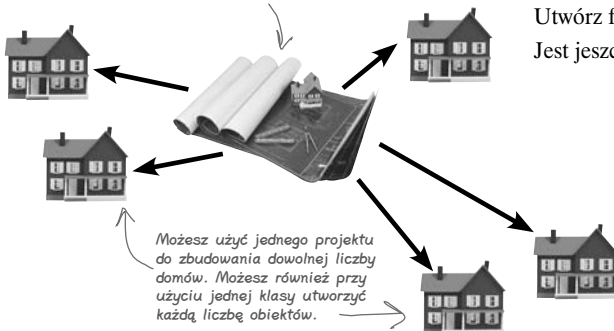
Tworzenie kodu ma sens

Każdy pisany przez Ciebie program rozwiązuje jakiś problem.

Rozpoczynając pisanie programu, zawsze warto zastanowić się, jaki *problem* ma on rozwiązywać. Właśnie do tego przydają się **obiekty**. Pozwalają one tworzyć strukturę kodu tak, by odpowiadała ona rozwiązywanemu problemowi, dzięki czemu będziesz mógł *skoncentrować się na nim samym*, a nie na mechanice tworzenia kodu. Prawidłowe użycie obiektów spowoduje, że proces pisania kodu stanie się bardziej *intuicyjny*, a jego późniejsza analiza i modyfikacja — znacznie łatwiejsze.



Kiedy definiujesz klasę, definiujesz także jej metody, podobnie jak projekt definiuje układ pomieszczeń w domu.



W jaki sposób Maciek myśli o swoich problemach	144
W jaki sposób system nawigacyjny w samochodzie Maćka rozwiązuje jego problemy	145
Klasa Navigator napisana przez Maćka posiada metody do ustalania i modyfikacji tras	146
Wykorzystaj to, czego się nauczyłeś, do napisania prostego programu używającego klas	147
Maciek ma pewien pomysł	149
Maciek może użyć obiektów do rozwiązania swojego problemu	150
Używasz klasy do utworzenia obiektu	151
Kiedy tworzysz obiekt na podstawie klasy, to taki obiekt nazywamy instancją klasy	152
Lepsze rozwiązanie... uzyskane dzięki obiektom!	153
Instancja używa pól do przechowywania informacji	158
Stwórzmy kilka instancji!	159
Dzięki za pamięć	160
Co Twój program ma na myśli	161
Możesz używać nazw klas i metod w celu uczynienia kodu bardziej intuicyjnym	162
Nadaj swojej klasie naturalną strukturę	164
Diagramy klas pozwalają w sensowny sposób zorganizować klasy	166
Utwórz klasę do pracy z kilkoma facetami	170
Utwórz projekt dla facetów	171
Utwórz formularz do interakcji z facetami	172
Jest jeszcze prostszy sposób inicjalizacji obiektów	175

Typy i referencje

4

Jest 10:00. Czy wiesz, gdzie są Twoje dane?

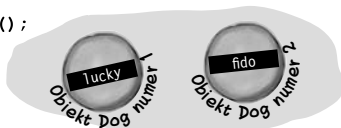
Typ danych, baza danych, dane komandora porucznika... wszystko to są ważne rzeczy. Bez danych Twoje programy są bezużyteczne. Potrzebujesz informacji dostarczanych przez użytkowników. Na ich podstawie wyszukujesz lub tworzysz nową **informację** i zwracasz im ją. W rzeczywistości prawie wszystko, co robisz podczas programowania, sprowadza się do **pracy z danymi** w taki czy w inny sposób. W tym rozdziale dowiesz się o różnych aspektach **typów danych** C#, nauczysz się pracować z danymi w programie, a nawet odkryjesz kilka pilnie strzeżonych sekretów **obiektów** (*psst... obiekty to także dane*).

Typ zmiennej określa rodzaj danych, jakie zmienna może przechowywać	182
Zmienna jest jak kubek z danymi	184
10 kilogramów danych w pięciokilogramowej torbie	185
Nawet wtedy, gdy liczba ma prawidłowy rozmiar, nie możesz przypisać jej do każdej zmiennej	186
Kiedy rzutujesz wartość, która jest zbyt duża, C# dopasowuje ją automatycznie	187
C# przeprowadza niektóre rzutowania automatycznie	188
Kiedy wywołujesz metodę, zmienne muszą pasować do typów parametrów	189
Przetestuj kalkulator zwrotu kosztów	193
Połączenie = z operatorem	194
Także obiekty używają zmiennych	195
Korzystaj ze swoich obiektów za pomocą zmiennych referencyjnych	196
Referencje są jak etykiety do Twoich obiektów	197
Jeżeli nie ma już żadnej referencji, Twoje obiekty są usuwane z pamięci	198
Referencje wielokrotne i ich efekty uboczne	199
Dwie referencje oznaczają DWA sposoby na zmianę danych obiektu	204
Specjalny przypadek: tablice	205
Tablice mogą także zawierać grupę zmiennych referencyjnych	206
Witamy w barze Niechlujny Janek — najtańsze kanapki w mieście!	207
Obiekty używają referencji do komunikacji między sobą	209
Tam, gdzie obiektów jeszcze nie było	210
Napisz grę w literki	215
Kontrolki to też obiekty, podobne do innych	219

```
Dog fido;
Dog lucky = new Dog();
```



```
fido = new Dog();
```



```
lucky = null;
```



Laboratorium C# numer 1

Dzień na wyścigach

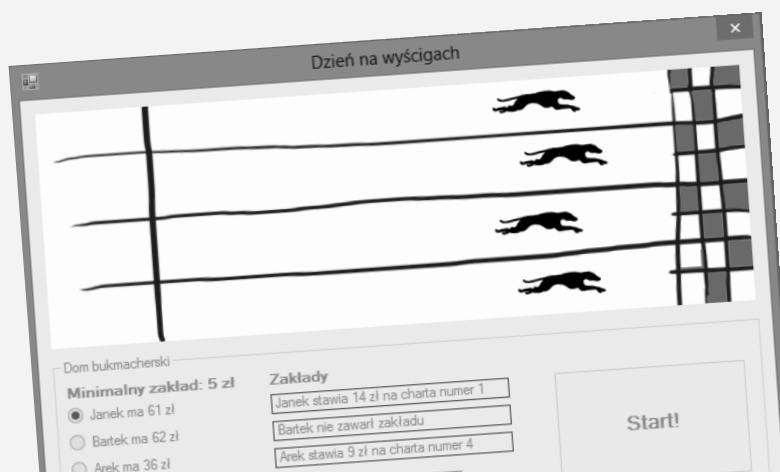
Janek, Bartek i Arek uwielbiają chodzić na tor wyścigowy, ale ciągła utrata pieniędzy powoduje u nich frustrację. Potrzebują symulatora, aby mogli określić zwycięzcę, zanim wyłożą pieniądze na zakłady. Jeśli dobrze wywiążesz się z zadania, będziesz miał procenty z ich wygranych.

Specyfikacja: stwórz symulator wyścigów

226

Końcowy produkt

234



Hermetyzacja

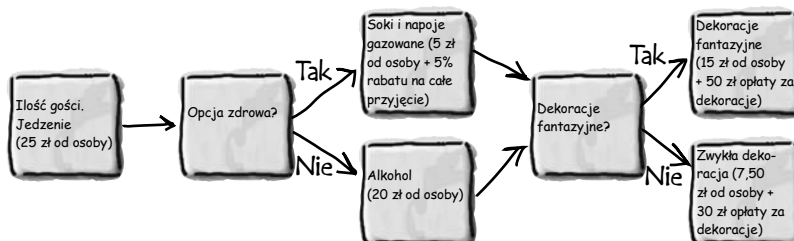
5

Co ma być ukryte... niech będzie ukryte

Czy kiedykolwiek marzyłeś o odrobinie prywatności? Czasami Twoje obiekty czują się tak samo. Na pewno nie lubisz sytuacji, w których ktoś, komu nie ufasz, czyta Twój pamiętnik lub przegląda wykazy Twoich operacji bankowych. Dobre obiekty nie pozwalają *innym* obiektom na oglądanie swoich pól. W tym rozdziale nauczysz się wykorzystywać potęgę hermetryzacji. Sprawisz, że dane obiektów będą prywatne, i dodasz metody, które umożliwią Ci zabezpieczenie dostępu do danych.



Krystyna planuje przyjęcia	236
Co powinien robić program szacujący?	237
Napiszesz program dla Krystyny	238
Jazda próbna Krystyny	244
Każda opcja powinna być obliczana osobno	246
Bardzo łatwo przez przypadek źle skorzystać z obiektów	248
Hermetyzacja oznacza, że niektóre dane w klasie są prywatne	249
Użyj hermetryzacji w celu kontroli dostępu do metod i pól Twojej klasy	250
Ale czy jego prawdziwa tożsamość jest NAPRAWDĘ chroniona?	251
Dostęp do prywatnych pól i metod można uzyskać tylko z wnętrza klasy	252
Hermetyzacja utrzymuje Twoje dane w nieskazitelnym stanie	260
Właściwości sprawiają, że hermetryzacja będzie łatwiejsza	261
Utwórz aplikację do przetestowania klasy Farmer	262
Użyj automatycznych właściwości do ukończenia klasy	263
Co wtedy, gdy chcemy zmienić pole mnożnika żywienia?	264
Użyj konstruktora do inicjalizacji pól prywatnych	265



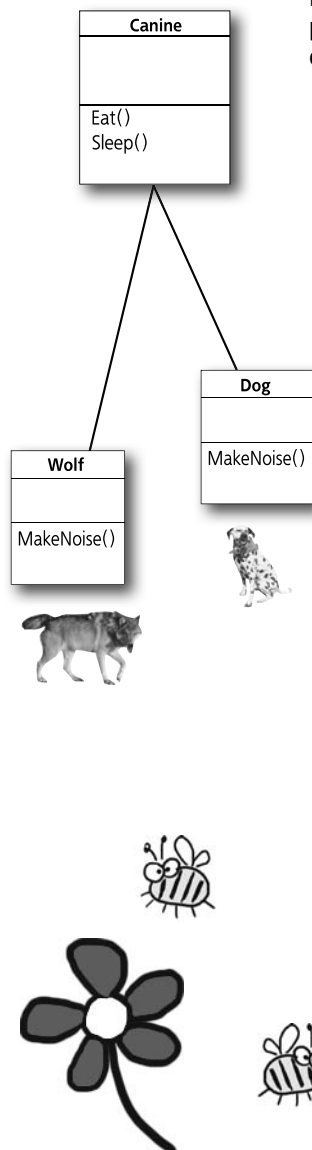
Dziedziczenie

Drzewo genealogiczne Twoich obiektów

6

Czasami **CHCIAŁBYŚ** być dokładnie taki sam jak Twoi rodzice.

Czy kiedykolwiek natknąłeś się na obiekt, który robiłby *prawie* wszystko, czego byś sobie od niego życzył? Czy kiedykolwiek znalazłeś się w takiej sytuacji, że gdybyś *zmienił dosłownie kilka rzeczy*, obiekt byłby doskonały? Cóż, to tylko jeden z wielu powodów, które sprawiają, że **dziedziczenie** zalicza się do najważniejszych koncepcji i technik w języku C#. Kiedy skończysz czytać ten rozdział, będziesz wiedział, jak **rozszerzać** obiekty, by móc wykorzystywać ich zachowania i jednocześnie dysponować **elastycznością**, która pozwoli Ci te zachowania modyfikować. Unikniesz **wielokrotnego pisania kodu**, **przedstawisz prawdziwy świat** znacznie dokładniej, a w efekcie otrzymasz kod **łatwiejszy do zarządzania**.



Krystyna organizuje także przyjęcia urodzinowe	276
Potrzebujemy klasy BirthdayParty	277
Stwórz program Planista przyjęć w wersji 2.0	278
Jeszcze jedna rzecz... Czy możesz dodać opłatę 100 zł za przyjęcia dla ponad 12 osób?	285
Kiedy klasy używają dziedziczenia, kod musi być napisany tylko raz	286
Zbuduj model klasy, rozpoczynając od rzeczy ogólnych i przechodząc do bardziej konkretnych	287
W jaki sposób zaprojektowałbyś symulator zoo?	288
Użyj dziedziczenia w celu uniknięcia powielania kodu w klasach potomnych	289
Różne zwierzęta wydają różne dźwięki	290
Pomyśl, w jaki sposób pogrupować zwierzęta	291
Stwórz hierarchię klas	292
Każda klasa pochodna rozszerza klasę bazową	293
Aby dziedziczyć po klasie bazowej, użyj dwukropka	294
Wiemy, że dziedziczenie dodaje pola, właściwości i metody klasy bazowej do klasy potomnej...	297
Klasa pochodna może przesłaniać odziedziczone metody w celu ich modyfikacji lub zmiany	298
W każdym miejscu, gdzie możesz skorzystać z klasy bazowej, możesz zamiast niej użyć jednej z jej klas pochodnych	299
Klasa pochodna może ukrywać metody klasy bazowej	306
Używaj override i virtual, by dziedziczyć zachowania	308
Klasa potomna może uzyskać dostęp do klasy bazowej, używając słowa kluczowego base	310
Jeśli Twoja klasa bazowa posiada konstruktor, klasa pochodna też musi go mieć	311
Teraz jesteś już gotowy do dokończenia zadania Krystyny	312
Stwórz system zarządzania ulem	317
Użyj dziedziczenia, aby rozszerzyć system zarządzania pszczołami	324

Interfejsy i klasy abstrakcyjne

7

Klasy, które dotrzymują swoich obietnic

Czyny potrafią powiedzieć więcej niż słowa. Czasami potrzebujesz pogrupować swoje obiekty na podstawie tego, **co robią**, zamiast tego, po jakiej klasie dziedziczą. To jest moment, w którym należy powiedzieć o **interfejsach**. Pozwalają one na pracę z każdą klasą, która jest w stanie wykonać daną czynność. Jednak wraz z **wielkimi możliwościami** **przychodzi wielka odpowiedzialność** i każda klasa, która implementuje interfejs, **musi wypełnić wszystkie swoje obowiązki**... albo kompilator połamie Ci nogi, zrozumiałeś?

* Dziedziczenie

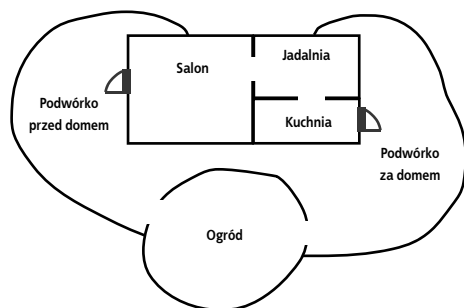
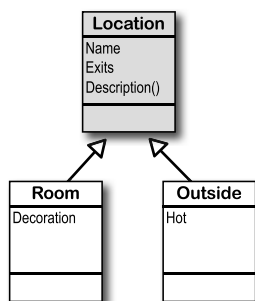


Abstrakcja

Hermetyzacja



Polimorfizm



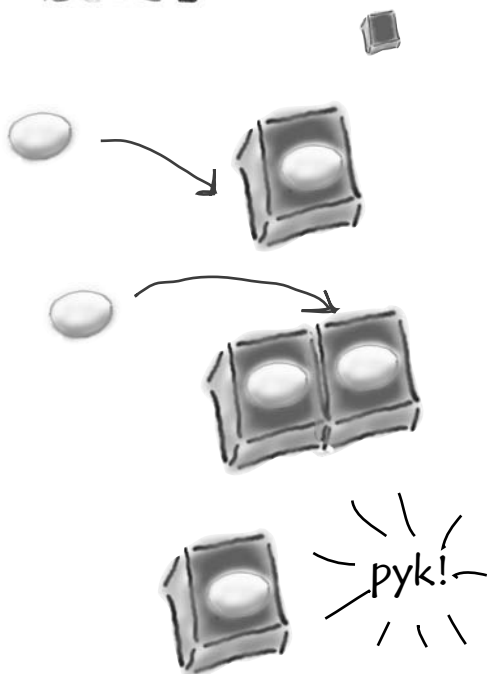
Wróćmy do pszczelej korporacji	330
Możemy użyć dziedziczenia do utworzenia klas dla różnych typów pszczół	331
Interfejs daje klasie do zrozumienia, że musi ona zaimplementować określone metody i właściwości	332
Użyj słowa kluczowego interface do zdefiniowania interfejsu	333
Teraz możesz utworzyć instancję NectarStinger, która będzie wykonywała dwa rodzaje zadań	334
Klasy implementujące interfejsy muszą zawierać WSZYSTKIE ich metody	335
Poćwicz trochę z interfejsami	336
Nie możesz utworzyć instancji interfejsu, ale możesz uzyskać jego referencję	338
Referencje interfejsów działają tak samo jak referencje obiektów	339
Za pomocą „is” możesz sprawdzić, czy klasa implementuje określony interfejs	340
Interfejsy mogą dziedziczyć po innych interfejsach	341
RoboBee 4000 może wykonywać zadania pszczół bez potrzeby spożywania cennego miodu	342
Ekspres do kawy także jest urządzeniem	344
Rzutowanie w górę działa w odniesieniu do obiektów i interfejsów	345
Rzutowanie w dół pozwala zamienić urządzenie z powrotem w ekspres do kawy	346
Rzutowanie w górę i w dół działa także w odniesieniu do interfejsów	347
Jest coś więcej niż tylko public i private	351
Modifikatory dostępu zmieniają widoczność	352
Obiekty niektórych klas nigdy nie powinny być tworzone	355
Klasa abstrakcyjna jest jak skrzyżowanie klasy i interfejsu	356
Jak wspominaliśmy, obiekty niektórych klas nigdy nie powinny być tworzone	358
Metoda abstrakcyjna nie ma ciała	359
Piekielny diament śmierci	364
Polimorfizm oznacza, że jeden obiekt może przyjmować wiele różnych postaci	367

Typy wyliczeniowe i kolekcje

8

Przechowywanie dużej ilości danych

Z deszczu pod rynnę. W rzeczywistym świecie nie musisz się zwykle zajmować danymi w małych ilościach i w niewielkich fragmentach. Nie, Twoje dane przychodzą do Ciebie w **grupach, stosach, pękach, kopach**. Potrzebujesz jakiegoś potężnego narzędzia do ich zorganizowania. Nadszedł czas, aby przedstawić **kolekcje**. Pozwalają one **przechowywać i sortować dane, a także zarządzać** tymi z nich, które Twój program musi przeanalizować. Dzięki temu możesz myśleć o pisaniu programów do pracy z danymi, a samo ich przechowywanie zostawić kolekcjom.



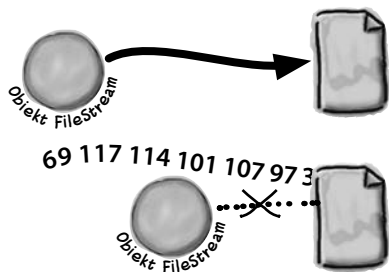
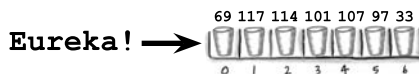
Łańcuchy znaków nie zawsze sprawdzają się przy kategoryzowaniu danych	386
Typy wyliczeniowe pozwalają Ci wyliczyć prawidłowe wartości	387
Typy wyliczeniowe pozwalają na reprezentowanie liczb za pomocą nazw	388
Z tablicami ciężko się pracuje	392
Listy ułatwiają przechowywanie kolekcji... czegokolwiek	393
Listy są bardziej elastyczne niż tablice	394
Listy kurczą się i rosną dynamicznie	397
Typy generyczne mogą przechowywać każdy typ	398
Inicjatory kolekcji działają tak samo jak inicjatory obiektu	402
Stwórzmy listę kaczek	403
Listy są proste, ale SORTOWANIE może być skomplikowane	404
IComparable<T> pomoże Ci posortować listę kaczek	405
Użyj interfejsu IComparer, aby powiedzieć liście, jak ma sortować	406
Utwórz instancję obiektu porównującego	407
IComparer może wykonywać złożone porównania	408
Przesłonięcie metody ToString() pozwala obiektom przedstawiać się	411
Zmień pętlę foreach tak, by obiekty Duck i Card same się opisywały	412
Pisząc pętlę foreach, używasz IEnumerable<T>	413
Używając IEnumerable, możesz rzutować całą listę w górę	414
Możesz tworzyć własne przeciążone metody	415
Użyj słownika do przechowywania kluczy i wartości	421
Wybrane funkcjonalności słownika	422
Napisz program korzystający ze słownika	423
I jeszcze WIĘCEJ typów kolekcji...	435
Kolejka działa według reguły: pierwszy przyszedł, pierwszy wyszedł	436
Stos działa według reguły: ostatni przyszedł, pierwszy wyszedł	437

Odczyt i zapis plików

9

Zachowaj te bajty dla mnie!

Czasem oplaca się być trwałym. Do tej pory wszystkie programy były krótkotrwałe. Uruchamiały się, działały przez chwilę i były zamykane. Czasami nie jest to wystarczające, zwłaszcza jeżeli zajmujesz się ważnymi danymi. Musisz mieć możliwość **zapisania swojej pracy**. W tym rozdziale pokażemy sposób **zapisywania danych do pliku**, a następnie **wczytywania tych informacji z powrotem** do programu. Dowiesz się co nieco o **klasach strumieni .NET** i zetkniesz się z tajemnicami systemów **szesnastkowego i dwójkowego**.



C# używa strumieni do zapisu i odczytu danych	442
Różne strumienie zapisują i odczytują różne rzeczy	443
FileStream odczytuje dane z pliku i zapisuje je w nim	444
W jaki sposób zapisać tekst do pliku w trzech prostych krokach	445
Kanciarz wymyślił nowy diabelski plan	446
Zapis i odczyt wymaga dwóch obiektów	449
Dane mogą przechodzić przez więcej niż jeden strumień	450
Użyj wbudowanych obiektów do wyświetlenia standardowych okien dialogowych	453
Okna dialogowe są kolejnymi kontrolkami .NET	454
Okna dialogowe także są obiektami	455
Używaj wbudowanych klas File oraz Directory do pracy z plikami i katalogami	456
Używaj okien dialogowych do otwierania i zapisywania plików (wszystko za pomocą kilku linijek kodu)	459
Dzięki IDisposable obiekty usuwane są prawidłowo	461
Unikaj błędów systemu plików, korzystając z instrukcji using	462
Zapisywanie danych do plików wymaga wielu decyzji	468
Użyj instrukcji switch do wybrania właściwej opcji	469
Dodaj przeciążony konstruktor Deck(), który wczytuje karty z pliku	471
Kiedy obiekt jest serializowany, serializowane są także wszystkie obiekty z nim powiązane...	475
Serializacja pozwala Ci zapisywać lub odczytywać całe grafy obiektów naraz	476
.NET automatycznie konwertuje tekst do postaci Unicode	481
C# może użyć tablicy bajtów do przesyłania danych	482
Do zapisywania danych binarnych używaj klasy BinaryWriter	483
Pliki utworzone dzięki serializacji można także zapisywać i odczytywać ręcznie	485
Sprawdź, gdzie pliki się różnią, i użyj tej informacji do ich zmiany	486
Praca z plikami binarnymi może być skomplikowana	487
Użyj strumieni plików do utworzenia widoku w postaci szesnastkowej	488
StreamReader i StreamWriter będą do tego odpowiednie	489

Laboratorium C# numer 2

Wyprawa

Twoim zadaniem jest stworzenie gry przygodowej, w której potężny wojownik wyrusza na wyprawę i dzielnie walczy, poziom za poziomem, ze śmiertelnie niebezpiecznymi wrogami. Stworzysz system turowy. Oznacza to, że najpierw gracz wykonuje jeden ruch, a następnie ruch wykonuje przeciwnik. Gracz może przesunąć się lub zaatakować; potem możliwość ruchu i ataku dostaje każdy z wrogów. Gra toczy się do czasu, aż gracz pokona wszystkich przeciwników na wszystkich siedmiu poziomach lub zginie.

Specyfikacja: utwórz grę przygodową

496

Zabawa dopiero się zaczyna!

516



Projektowanie aplikacji dla Sklepu Windows z użyciem XAML

10

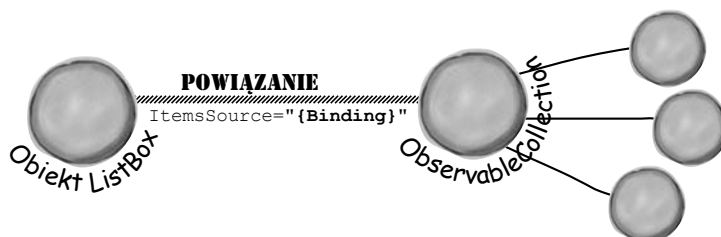
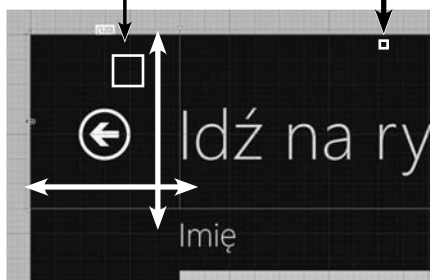
Przenosząc swoje aplikacje na wyższy poziom

Jesteś już gotów, by wkroczyć do zupełnie nowego świata tworzenia aplikacji.

Korzystanie z technologii WinForms do tworzenia klasycznych aplikacji dla systemu Windows jest doskonałym sposobem nauki ważnych rozwiązań języka C#, niemniej jednak możesz pójść *znacznie dalej*. W tym rozdziale dowiesz się, jak używać języka **XAML** do projektowania aplikacji przeznaczonych dla Sklepu Windows, nauczysz się tworzyć aplikacje **działające na dowolnych urządzeniach, integrować** dane ze stronami przy użyciu **wiązania danych** i używać Visual Studio do ujawniania tajemnic stron XAML poprzez badanie obiektów tworzonych na podstawie kodu XAML.

Siatka składa się z kwadratów o wielkości 20 pikseli, nazywanych jednostkami.

Każda jednostka jest podzielona na podjednostki o wielkości 5 pikseli.



Damian używa Windows 8	518
Technologia Windows Forms korzysta z grafu obiektów stworzonego przez IDE	524
Użyj IDE do przejrzania grafu obiektów	527
Aplikacje dla Sklepu Windows używają XAML do tworzenia obiektów interfejsu użytkownika	528
Przeprojektuj formularz Idź na ryby!, zmieniając go w aplikację dla Sklepu Windows	530
Określanie postaci strony rozpoczyna się od dodania kontrolki	532
Wiersze i kolumny mogą zmieniać wielkość, dostosowując się do rozmiarów strony	534
Skorzystaj z systemu siatki, by określić układ stron aplikacji	536
Wiązanie danych kojarzy strony XAML z klasami	542
Kontrolki XAML mogą zawierać tekst... i nie tylko	544
Użyj wiązania danych, by usprawnić aplikację Niechlujnego Janka	546
Korzystaj z zasobów statycznych, by deklarować obiekty w kodzie XAML	552
Wyświetlaj obiekty, używając szablonów danych	554
Interfejs INotifyPropertyChanged pozwala powiązanym obiektom przesyłać aktualizacje	556
Zmodyfikuj klasę MenuMaker, by informowała Cię, gdy zmieni się właściwość GeneratedDate	557

Async, await i serializacja kontraktu danych

11

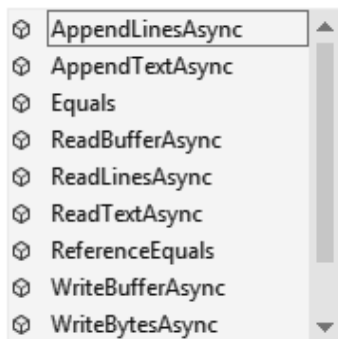
Przepraszam, że przerywam

Nikt nie lubi być zmuszanym do oczekiwania... zwłaszcza użytkownicy.

Komputery są doskonałe w wykonywaniu wielu rzeczy jednocześnie, nie ma zatem żadnego powodu, aby Twoje aplikacje nie mogły tego robić. W tym rozdziale dowiesz się, jak sprawić, by dzięki **zastosowaniu metod asynchronicznych** Twoje aplikacje reagowały błyskawicznie. Nauczysz się także korzystać z **wbudowanych narzędzi do wybierania plików**, wyświetlać **okienka z komunikatami** oraz **asynchronicznie zapisywać i odczytywać dane z plików** bez „zawieszania” aplikacji. Połączysz te wszystkie możliwości z serializacją kontraktu danych i opanujesz tworzenie bardzo nowoczesnych aplikacji.

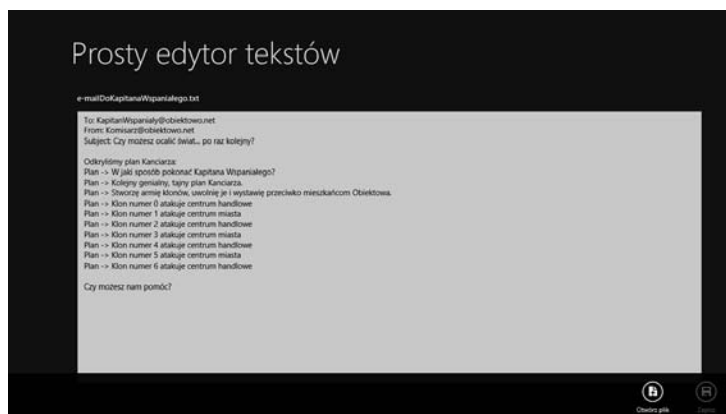


FileIO.



Damian ma problemy z plikami	566
Aplikacje dla Sklepu Windows używają await, by błyskawicznie reagować	568
Używaj klasy FileIO do odczytywania i zapisywania plików	570
Napisz nieco mniej prosty edytor tekstów	572
Kontrakt danych jest abstrakcyjną definicją danych obiektu	577
Do odnajdywania i otwierania plików używaj metod asynchronicznych	578
Klasa KnownFolders ułatwia dostęp do najczęściej używanych folderów	580
W kodzie XML jest serializowany cały graf obiektów	581
Prześlij kilka obiektów Guy do lokalnego folderu aplikacji	582
Wypróbujmy działanie aplikacji	586
Używaj klasy Task, by wywoływać jedną metodę asynchroniczną w innej	587
Napisz dla Damiana nową aplikację do zarządzania wymówkami	588
Odrębna strona, wymówka i ExcuseManager	589
Utwórz stronę główną aplikacji Menedżera wymówek	590
Dodaj pasek aplikacji do strony głównej	591
Napisz klasę ExcuseManager	592
Dodaj kod obsługujący stronę	594

Prosty edytor tekstów

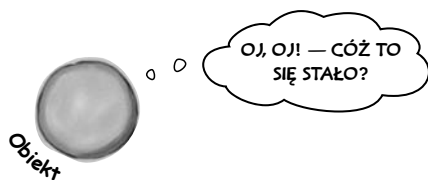


Obsługa wyjątków

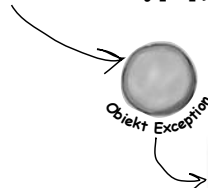
12

Gaszenie pożarów nie jest już popularne

Programiści nie mają być strażakami. Pracowałeś jak wół, przebrnąłeś przez dokumentację techniczne i kilka zajmujących książek *Rusz głową!*, wspiąłeś się na szczyt swoich możliwości: jesteś **mistrzem programowania**. W dalszym ciągu musisz jednak odrywać się od pracy, ponieważ **program wyłącza się** lub **nie zachowuje się tak, jak powinien**. Nic nie wybija Cię z rytmu tak, jak obowiązek naprawienia dziwnego błędu... Z **obsługą wyjątków** możesz jednak napisać kod, który **poradzi sobie z pojawiającymi się problemami**. Jest nawet lepiej, możesz bowiem zareagować na ich pojawienie się i sprawić, że wszystko **będzie dalej działało**.



```
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```



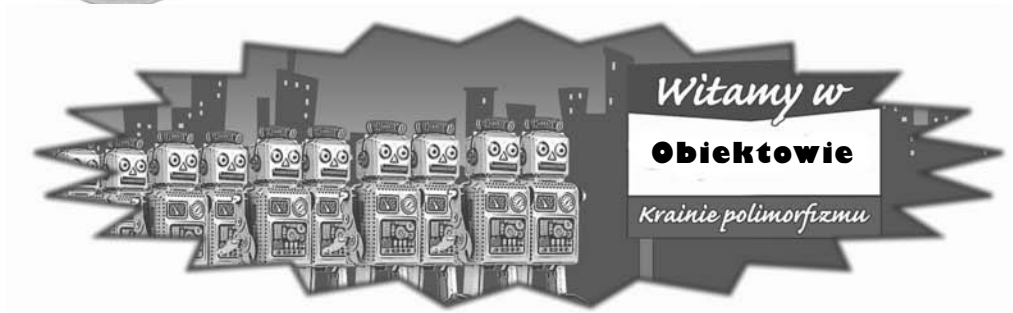
Damian potrzebuje swoich wymówek, aby być mobilnym	600
Kiedy program zgłasza wyjątek, .NET tworzy obiekt Exception	604
Kod Damiana zrobił coś nieoczekiwano	606
Wszystkie obiekty wyjątków dziedziczą po Exception	608
Debugger pozwala Ci wysledzić wyjątki w kodzie i zapobiec im	609
Użyj debuggera wbudowanego w IDE, aby znaleźć problem w programie do zarządzania wymówkami	610
Oj, oj! — w kodzie dalej są błędy...	613
Obsłuż wyjątki za pomocą try i catch	615
Co się stanie, jeżeli wywoływana metoda będzie niebezpieczna?	616
Użyj debuggera do przesłedzenia przepływu w blokach try/catch	618
Jeśli posiadasz kod, który ZAWSZE musi zostać wykonany, zastosuj finally	620
Użyj obiektu Exception w celu uzyskania informacji o problemie	625
Użyj więcej niż jednego bloku catch do wyłapania różnych typów wyjątków	626
Jedna klasa zgłasza wyjątek, inna klasa go przechwytuje	627
Łatwy sposób na uniknięcie licznych problemów: using umożliwia Ci stosowanie try i finally za darmo	631
Unikanie wyjątków: zaimplementuj IDisposable, aby przeprowadzić własne procedury sprzątnięcia	632
Najgorszy z możliwych bloków catch: komentarze	634
Kilka prostych wskazówek dotyczących obsługi wyjątków	636

13

Kapitan Wspaniały

Śmierć obiektu

Twoją ostatnią szansą na ZROBIENIE czegoś... jest użycie finalizatora	646
Kiedy DOKŁADNIE wywoływany jest finalizator?	647
Dispose() działa z using, a finalizatory działają z mechanizmem oczyszczania pamięci	648
Finalizatory nie mogą polegać na stabilności	650
Spraw, aby obiekt serializował się w Dispose()	651
Struktura jest podobna do obiektu...	655
...ale nie jest obiektem	655
Wartości są kopiowane, referencje są przypisywane	656
Struktury traktowane są jak typy wartościowe, obiekty jak typy referencyjne	657
Stos i sarta: więcej na temat pamięci	659
Używaj parametrów wyjściowych, by zwracać z metody więcej niż jedną wartość	662
Przekazuj referencje, używając modyfikatora ref	663
Używaj parametrów opcjonalnych, by określać wartości domyślne	664
Jeśli musisz używać wartości pustych, stosuj typy, które je akceptują	665
Typy akceptujące wartości puste poprawiają odporność programów	666
„Kapitan” Wspaniały... nie tak bardzo	669
Metody rozszerzające zwiększają funkcjonalność ISTNIEJĄCYCH klas	670
Rozszerzanie podstawowego typu: string	672



Przeszukiwanie danych i tworzenie aplikacji przy użyciu LINQ

14

Przejmij kontrolę nad danymi

To świat przepelniony danymi... Lepiej, żebyś wiedział, jak w nim żyć.

Czasy, gdy mogłeś programować kilka dni, a nawet kilka tygodni, bez konieczności pracy z **ogromem danych**, minęły już bezpowrotnie. Nadeszła epoka, w której **wszystko opiera się na nich**. W tym miejscu do akcji wkracza LINQ. To nie tylko sposób na **pobieranie danych** w prosty, intuicyjny sposób. Pozwala on także **grupować i łączyć dane pochodzące z różnych źródeł**. A kiedy już podzielisz dane na fragmenty, którymi można łatwo zarządzać, Twoje aplikacje dla Sklepu Windows będą **korzystać z kontrolek do nawigowania**, pozwalających poruszać się po danych, przeglądać je, a nawet powiększać i wyświetlać szczegółowe informacje na ich temat.

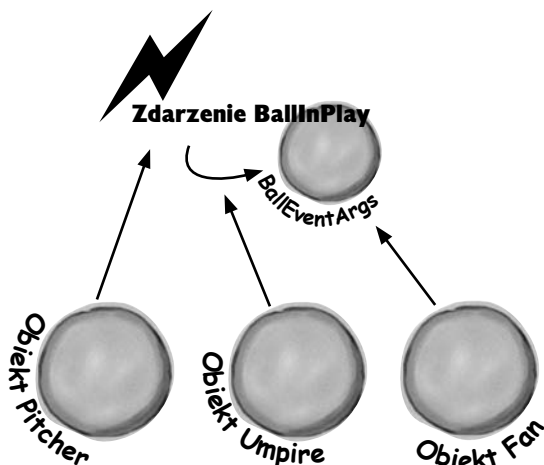
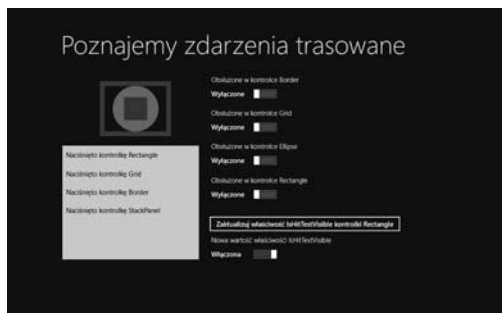
Janek jest superfanem Kapitana Wspaniałego...	678
...ale jego kolekcja zajmuje każde wolne miejsce	679
Dzięki LINQ możesz pobrać dane z różnych źródeł	680
Kolekcje .NET są przystosowane do działania z LINQ	681
LINQ ułatwia wykonywanie zapytań	682
LINQ jest prosty, ale Twoje zapytania wcale takie być nie muszą	683
Janek chętnie skorzystałby z pomocy	686
Zacznij pisać aplikację dla Janka	688
Używaj słowa kluczowego new, by tworzyć typy anonimowe	691
LINQ ma wiele zastosowań	694
Dodaj nowe zapytania do aplikacji Janka	696
LINQ może połączyć Twoje wyniki w grupy	701
Połącz wartości Janka w grupy	702
Użyj join do połączenia dwóch kolekcji w jedną sekwencję	705
Janek zaoszczędził mnóstwo szmalu	706
Użyj semantycznego powiększenia, aby przejść do danych	712
Dodaj zoom semantyczny do aplikacji Janka	714
Zrobiłeś na Janku wielkie wrażenie	719
Szablon Split App ułatwia tworzenie aplikacji służących do przeglądania danych	720

Zdarzenia i delegaty

15

Co robi Twój kod, kiedy nie patrzysz

Twoje obiekty zaczynają myśleć o sobie. Nie możesz zawsze kontrolować tego, co one robią. Czasami różne rzeczy... zdarzają się. Kiedy to następuje, chciałbyś, aby Twoje obiekty były wystarczająco sprytnie i odpowiednio **reagowały**. To miejsce, w którym do akcji wkraczają zdarzenia. Jeden obiekt *udostępnia* zdarzenie, inny je *obsługuje* i wszystko pracuje razem, aby całość działała sprawnie. Jest to wspaniałe, o ile nie chcesz, by Twój obiekt mógł kontrolować, kto będzie mógł nasłuchiwać jego zdarzeń. Wtedy bardzo pomocne okazują się **funkcje zwrotne**.



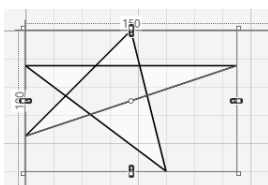
Czy kiedykolwiek marzyłeś o tym, aby Twoje obiekty potrafiły samodzielnie myśleć?	730
Ale skąd obiekt WIE, że ma odpowiedzieć?	730
Kiedy wystąpi ZDARZENIE... obiekty nasłuchują	731
Jeden obiekt wywołuje zdarzenie, inne nasłuchują...	732
Potem inne obiekty obsługują zdarzenie	733
Łącząc punkty	734
IDE automatycznie tworzy za Ciebie procedury obsługi zdarzeń	738
Ogólny typ EventHandler pozwala definiować własne typy zdarzeń	744
Formularze używają wielu różnych zdarzeń	745
Jedno zdarzenie, wiele procedur obsługi	746
Aplikacje dla Sklepu Windows używają zdarzeń do zarządzania cyklem życia procesu	748
Dodaj zarządzanie cyklem życia procesu do aplikacji Janka	749
Kontrolki XAML korzystają ze zdarzeń trasowanych	752
Utwórz aplikację do badania zdarzeń trasowanych	753
Połączenie nadawców zdarzenia z jego odbiorcami	758
Delegat ZASTĘPUJE właściwą metodę	759
Delegat w akcji	760
Każdy obiekt może subskrybować publiczne zdarzenie...	763
Użyj funkcji zwrotnej, by wiedzieć, kto nasłuchuje	764
Funkcje zwrotne są jedynie sposobem używania delegatów	766
Możesz używać funkcji zwrotnych w oknach dialogowych MessageDialog	768
Użyj delegatów, by skorzystać z panelu Ustawienia	770

Projektowanie aplikacji według wzorca MVVM

16

Świetne aplikacje od zewnątrz i od środka

Twoje aplikacje muszą być nie tylko wspaniałe wizualnie. Kiedy mówimy o *projekcie*, co Ci przychodzi do głowy? Przykład jakiejś wspaniałej architektury budowlanej? Doskonale rozplanowana strona WWW? Produkt zarówno estetyczny, jak i dobrze zaprojektowany? Dokładnie te same zasady odnoszą się do aplikacji. W tym rozdziale poznasz **wzorzec Model-View-ViewModel** (MVVM, model-widok-widok modelu) i dowiesz się, jak używać go do tworzenia dobrze zaprojektowanych aplikacji o luźnych powiązaniach. Przy okazji poznasz **animacje, szablony kontrolki** używane do projektowania wyglądu aplikacji, nauczysz się stosować **konwertery**, by ułatwić korzystanie z techniki wiązania danych, a w końcu zobaczysz, jak połączyć te wszystkie elementy, by **stworzyć solidne podstawy** do tworzenia dowolnych aplikacji w języku C#.



Liga „Koszykówka. Rusz głową” potrzebuje swojej aplikacji	774
Jednak czy wszyscy uzgodnią, jak napisać tę aplikację?	775
Czy projektujesz pod kątem wiązania danych, czy łatwości pracy z danymi?	776
Wzorzec MVVM pozwala projektować, uwzględniając zarówno wiązanie, jak i dane	777
Użyj wzorca MVVM, by rozpocząć tworzenie aplikacji dla ligi koszykówki	778
Kontrolki użytkownika pozwalają tworzyć swoje własne kontrolki	781
Sędziowie potrzebują stopera	789
Wzorzec MVVM oznacza myślenie o stanie aplikacji	790
Zacznij tworzenie modelu aplikacji stopera	791
Zdarzenia ostrzegają resztę aplikacji o zmianie stanu	792
Utwórz widok prostej aplikacji stopera	793
Dodaj model widoku aplikacji stopera	794
Konwertery automatycznie konwertują wartości na potrzeby powiązań	798
Konwertery mogą operować na wielu różnych typach danych	800
Stan wizualny sprawia, że kontrolki odpowiadają na zmiany	806
Używaj DoubleAnimation, by animować wartości zmiennoprzecinkowe	807
Używaj animacji obiektów do animowania wartości obiektów	808
Stwórz stoper wskazówkowy, używając tego samego modelu widoku	809
Kontrolki interfejsu użytkownika można także tworzyć w kodzie C#	814
C# pozwala także na tworzenie „prawdziwych” animacji	816
Użyj kontrolki użytkownika, by wyświetlać rysunki tworzące animację	817
Niech Twoje pszczoły latają po stronie	818
Użyj ItemsPanelTemplate, by powiązać pszczoły z kontrolką Canvas	821
Gratulujemy! (Choć jeszcze nie skończyłeś...)	834

Laboratorium C# numer 3

Invaders

Dzięki temu laboratorium oddasz hołd jednej z najbardziej popularnych, czczonych i powielanych ikon w historii gier komputerowych. Nie potrzebuje ona żadnego wprowadzenia. Czas utworzyć grę Invaders.

Dziadek wszystkich gier	836
Można zrobić znacznie więcej...	857



Projekt dodatkowy!

17

Napisz aplikację Windows Phone

Klasy, obiekty, XAML, hermetyzacja, dziedziczenie, polimorfizm, LINQ, MVVM... dysponujesz już wszystkimi narzędziami niezbędnymi do pisania wspaniałych aplikacji dla Sklepu Windows oraz tradycyjnych aplikacji okienkowych. Czy jednak wiesz, że **tych samych narzędzi możesz użyć do pisania aplikacji dla *Windows Phone***? Tak, to prawda! W tym dodatkowym projekcie poznasz proces tworzenia gry dla systemu Windows Phone. Jeśli nie posiadasz odpowiedniego urządzenia, to i tak nie masz się czym przejmować — będziesz mógł w nią grać na **emulatorze Windows Phone**. A zatem zaczynamy!



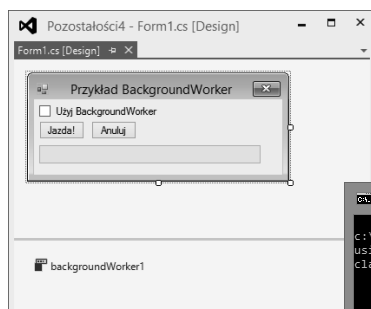
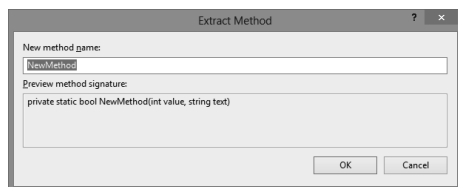
Atak pszczół!	860
Zanim zaczniesz...	861

Pozostałości

A

11 najważniejszych rzeczy, które chcieliśmy umieścić w tej książce

Zabawa dopiero się zaczyna! Pokazaliśmy Ci mnóstwo wspariających narzędzi do tworzenia naprawdę **potężnych programów** w C#. Nie było jednak możliwe, abyśmy w tej książce zmieścili **każde narzędzie, technologię i technikę** — nie ma ona po prostu tylu stron. Musieliśmy podjąć *naprawdę przemyślaną decyzję*, co umieścić, a co pominąć. Oto kilka tematów, których nie mogliśmy przedstawić. Pomimo tego, że nie zajęliśmy się nimi, w dalszym ciągu myślimy, że są one **ważne i przydatne**. Należałoby więc chociaż o nich wspomnieć. Tak też zrobiliśmy.



```

Developer Command Prompt for VS2012

c:\Users\Public\Documents>type HelloWorld.cs
using System;
class HelloWorld {
    public static void Main(string[] args) {
        Console.WriteLine("Witaj, świecie!");
    }
}

c:\Users\Public\Documents>csc HelloWorld.cs
Kompilator Microsoft (R) Visual C# w wersji 4.0.30319.33440
dla programu Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. Wszelkie prawa zastrzeżone.

c:\Users\Public\Documents>HelloWorld.exe
Witaj, świecie!

c:\Users\Public\Documents>

```

1. Na temat aplikacji dla Sklepu Windows można dowiedzieć się znacznie więcej 874
 2. Podstawy 876
 3. Przestrzenie nazw i złożenia 882
 4. Użyj BackgroundWorker, by poprawić działanie interfejsu użytkownika 886
 5. Klasa Type oraz metoda GetType() 889
 6. Równość, IEquatable oraz Equals() 890
 7. Stosowanie yield return do tworzenia obiektów umożliwiających iterację 893
 8. Refaktoryzacja 896
 9. Anonimowe typy i metody oraz wyrażenia lambda 898
 10. Zastosowanie LINQ to XML 900
 11. Windows Presentation Foundation 902
- Czy wiesz, że C# i .NET Framework potrafią... 903

S

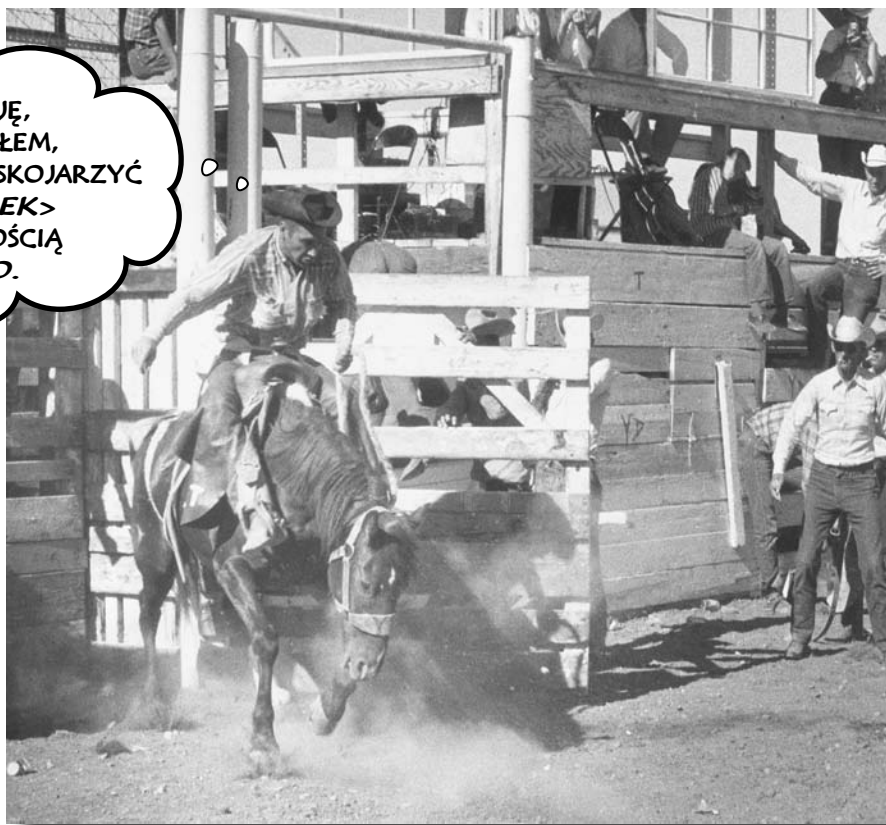
Skorowidz

905

10. Projektowanie aplikacji dla Sklepu Windows z użyciem XAML

Przenieś swoje aplikacje na wyższy poziom

MAM NADZIEJĘ,
ŻE PAMIĘTAŁEM,
BY PRAWIDŁOWO SKOJARZYĆ
SWÓJ <TYŁEK>
Z WŁAŚCIWOŚCIĄ
SIODŁO.



Jesteś już gotów, by wkroczyć do zupełnie nowego świata tworzenia aplikacji.

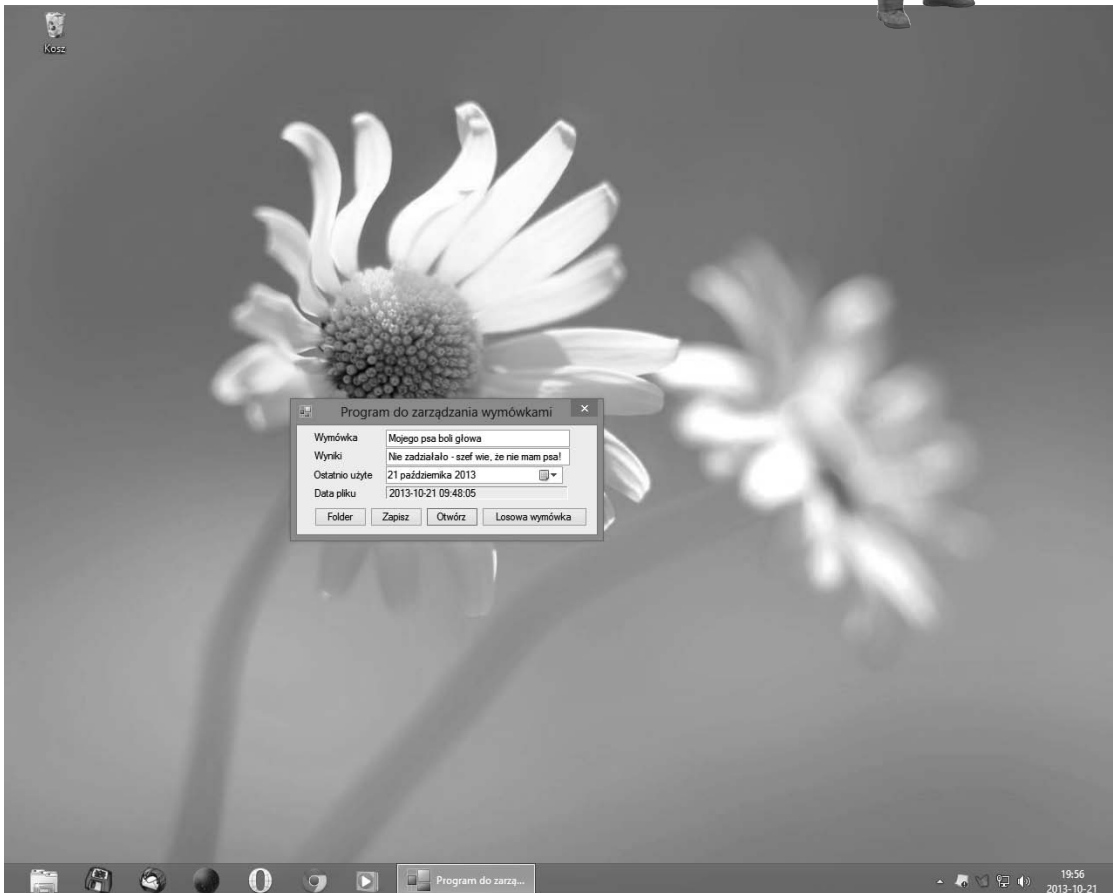
Korzystanie z technologii WinForms do tworzenia klasycznych aplikacji dla systemu Windows jest doskonałym sposobem nauki ważnych rozwiązań języka C#, niemniej jednak możesz pójść *znacznie dalej*. W tym rozdziale dowiesz się, jak używać języka **XAML** do projektowania aplikacji przeznaczonych dla Sklepu Windows, nauczysz się tworzyć aplikacje **działające na dowolnych urządzeniach**, **integrować** dane ze stronami przy użyciu **wiązania danych** i używać Visual Studio do ujawniania tajemnic stron XAML poprzez badanie obiektów tworzonych na podstawie kodu XAML.

Damian używa Windows 8

Staromodne aplikacje Damiana dla systemu Windows wyglądają archaicznie! Damian ma już dosyć ciągłego klikania malutkich pól wyboru. A chciałby, żeby jego program do zarządzania wymówkami był aplikacją z prawdziwego zdarzenia. Czy możemy mu w tym pomóc?

Program do zarządzania wymówkami Damiana działa, jednak przestarzały klasyczny program dla systemu Windows nie jest żadną konkurencją dla prawdziwej, fantastycznej aplikacji, w stu procentach przeznaczonej dla Sklepu Windows.

CO TO NIBY JEST, JAKIŚ
ZAMIERZCHŁY ROK 2003?
LUDZIE, ZAJMIJMY SIĘ TYM
PROGRAMEM!





Zrób to!

Chcesz szybko wtłoczyć te pomysły do swojej głowy? Zatem zrób opisane poniżej rzeczy, zanim zaczniesz czytać ten rozdział!

Aplikacje przeznaczone dla Sklepu Windows są bardziej skomplikowane niż programy WinForms. Dlatego też programy WinForms są naprawdę efektywnym narzędziem do nauki, jednak nie radzą sobie równie dobrze, gdy chodzi o napisanie niezwykle odlotowej aplikacji. Czasami warto cofnąć się o krok i pomyśleć o tym, jak się uczyć, gdyż to pomoże nam robić to bardziej efektywnie. A zatem spróbujmy to zrobić teraz.

Stworzyłeś już sobie doskonałe podstawy, zdobywając podstawową wiedzę o języku C#, obiektach, kolekcjach i innych narzędziach .NET. Teraz jednak wrócimy do tworzenia aplikacji dla Sklepu Windows przy wykorzystaniu języka XAML. Na kilku następnych stronach spróbujemy skorzystać z możliwości IDE do badania obiektów, które tworzą i którymi zarządzają programy WinForms. Kiedy później użyjesz IDE do badania obiektów tworzonych w aplikacjach dla Sklepu Windows pisanych przy użyciu języka XAML, staraj się wychwytywać różnice — oraz, co jest równie ważne, podobieństwa pomiędzy nimi.

Wykonaj poniższe czynności, zanim zaczniesz dalszą lekturę tego rozdziału

Cały rozdział 1. i znaczną część rozdziału 2. przeznaczaliśmy na przedstawienie tworzenia aplikacji przeznaczonych dla Sklepu Windows przy wykorzystaniu języka XAML oraz platformy .NET Framework for Windows Store. Teraz wykorzystamy wiedzę zdobytą w tamtych dwóch rozdziałach. Jeśli chcesz w możliwie jak największym stopniu skorzystać z lektury tego rozdziału, sugerujemy, żebyś wykonał kilka przedstawionych poniżej czynności.

- ★ Dla niektórych przesiadka z WinForms na stosowanie języka XAML jest całkowicie bezproblemowa, innych natomiast może drażnić. **Te czynności przygotowawcze pomogą Ci szybciej przyswoić sobie najważniejsze idee.**
- ★ Wróć do rozdziału 1. i ponownie, od samego początku stwórz aplikację *Ratuj ludzi*. Tym razem upewnij się, że wpisałeś ręcznie cały kod.
- ★ I nie zapomnij **przeanalizować kodu aplikacji!** Wciąż znajdują się w nim fragmenty, którymi jeszcze nie zajmowaliśmy się szczegółowo, choć powinieneś już rozpoznawać je na tyle dobrze, byś mógł rozpocząć tworzenie mentalnych podstaw.
- ★ Spróbuj dobrze zrozumieć tajniki działania gry. Nie staraj się jednak robić tego za wszelką cenę. Jak już wspominaliśmy — wciąż pozostało sporo zagadnień, którymi w tej książce jeszcze nie zajmowaliśmy się.
- ★ Zwróć szczególną uwagę na zmiany wprowadzane w szablonie *Basic Page* oraz na sposób, w jaki zastąpiliśmy nim domyślną stronę *MainPage.xaml* — w tym rozdziale będziesz bowiem to robił kilka razy.
- ★ **Jeszcze raz wykonaj projekt XAML z rozdziału 2.** Nie zapomnij zrobić także ćwiczenia. Teraz powinieneś być gotowy!

I jeszcze jedno...

W tej książce nie przedstawiliśmy wszystkich możliwości aplikacji WinForms. W rzeczywistości korzystają one z silnika graficznego określanego jako GDI+, potrafiącego generować zaskakująco dobry interfejs graficzny i materiały do druku oraz doskonale obsługiwać interakcję z użytkownikiem (choć wymaga ona znacznie większego nakładu pracy niż w przypadku korzystania z języka XAML). Jednym z najważniejszych sposobów nauki podstawowych zasad programowania jest przeanalizowanie tej samej rzeczy wykonanej na dwa różne sposoby.



Defektywistyczne poszukiwania!

Od czasu pierwszego projektu, *Ratuj ludzi*, przedstawionego w rozdziale 1., poznałeś już całkiem sporo ważnych pojęć związanych z językiem C# i nabyłeś dużo praktyki w posługiwaniu się nimi. Teraz nadszedł czas, by założyć kapelusz, wziąć lupę i przetestować swoje detektywistyczne umiejętności. Sprawdź, czy będziesz w stanie odszukać wszystkie elementy C# w kodzie aplikacji *Ratuj ludzi*. Aby ułatwić Ci początki poszukiwań, podaliśmy jedną z odpowiedzi. Czy jesteś w stanie znaleźć pozostałe?

(Na niektóre z pytań jest więcej niż jedna prawidłowa odpowiedź).

- Zastosowanie łańcucha znaków jako klucza podczas pobierania obiektu z kolekcji Dictionary.

.....
.....
.....

- Zastosowanie inicjalizatora.

.....
.....
.....

- Dodanie do tej samej kolekcji obiektów dwóch różnych typów.

.....
.....
.....

- Wywołanie metody statycznej.

.....
.....
.....

Zastosowanie metody procedury obsługi zdarzeń.

.....
.....
.....

Użycie słowa kluczowego `as` w celu rzutowania obiektu w dół.

.....
.....
.....

Przekazanie do metody referencji do obiektu.

.....
.....
.....

Utworzenie instancji obiektu i wywołanie jednej z jego metod.

.....
.....*W metodzie `AnimateEnemy()` tworzony jest obiekt klasy `Storyboard`, a następnie wywoływana jest jego metoda `Begin()`.*
.....

Użycie typu wyliczeniowego w celu przypisania wartości.

.....
.....
.....





Defektywistyczne poszukiwania!

Rozwiązanie

Od czasu pierwszego projektu, *Ratuj ludzi*, przedstawionego w rozdziale 1., poznałeś już całkiem sporo ważnych pojęć związanych z językiem C# i nabyłeś dużo praktyki w posługiwaniu się nimi. Teraz nadszedł czas, by założyć kapelusz, wziąć lupę i przetestować swoje detektywistyczne umiejętności. Sprawdź, czy będziesz w stanie odszukać wszystkie elementy C# w kodzie aplikacji *Ratuj ludzi*. Aby ułatwić Ci początki poszukiwań, podaliśmy jedną z odpowiedzi. Czy jesteś w stanie znaleźć pozostałe?

(Na niektóre z pytań jest więcej niż jedna prawidłowa odpowiedź).

Poniżej zamieściliśmy te, które nam udało się znaleźć; Ty mogłeś znaleźć zupełnie inne!

Zastosowanie łańcucha znaków jako klucza podczas pobierania obiektu z kolekcji Dictionary.

..... W drugim wierszu metody `AddEnemy()` używany jest łańcuch znaków „`EnemyTemplate`”, przy użyciu którego ze słownika `Resources` pobierany jest obiekt `ControlTemplate`.

Zastosowanie inicjalizatora.

..... W metodzie `AnimateEnemy()` podczas inicjalizacji obiektu `DoubleAnimation` określone są wartości trzech właściwości: `From`, `To` oraz `Duration`.

Dodanie do tej samej kolekcji obiektów dwóch różnych typów.

..... W metodzie `StartGame()` do kolekcji `playArea.Children` dodawany jest obiekt `StackPanel` (człowiek) oraz obiekt `Rectangle` (docelowy portal).

Wywołanie metody statycznej.

..... W procedurze obsługi zdarzeń `target_PointerEntered()` wywoływane są dwie statyczne metody klasy `Canvas`: `SetLeft()` oraz `SetTop()`.

✓ Zastosowanie metody procedury obsługi zdarzeń.

..... W oknie *Properties* została określona procedura obsługi
..... zdarzeń *PointerPressed* obiektu *StackPanel* reprezentującego
..... człowieka.

✓ Użycie słowa kluczowego *as* w celu rzutowania obiektu w dół.

..... Słownik *Resources* zwraca obiekt typu *<object, object>*,
..... a zatem wartość odwrotania *Resources["EnemyTemplate"]*
..... jest rzutowana w dół na konkretny typ *ControlTemplate*.

✓ Przekazanie do metody referencji do obiektu.

..... Referencja do obiektu *ContentControl* jest przekazywana
..... jako pierwszy parametr metody *AnimateEnemy()*.

✓ Utworzenie instancji obiektu i wywołanie jednej z jego metod.

..... W metodzie *AnimateEnemy()* tworzony jest obiekt klasy
..... *Storyboard*, a następnie wywoływana jest jego metoda
..... *Begin()*.

✓ Użycie typu wyliczeniowego w celu przypisania wartości.

..... W metodzie *EndTheGame()* jest używany typ wyliczeniowy
..... *Visibility*, a konkretnie właściwość *startButton.Visibility*
..... jest przypisywana wartość *Visibility.Visible*.



Technologia Windows Forms korzysta z grafu obiektów stworzonego przez IDE

Podczas tworzenia klasycznej aplikacji przeznaczonej dla systemu Windows IDE przygotowuje formularz i generuje jego kod, który umieszcza w pliku *Form1.Designer.cs*. Ale co właściwie jest umieszczane w tym pliku? Nie jest to bynajmniej nic tajemniczego. Już wiesz, że wszystkie kontrolki umieszczane na formularzu są obiektami, wiesz także, że referencje do obiektów można zapisywać w polach. A zatem w jakimś miejscu wygenerowanego kodu musi się znaleźć deklaracja pola reprezentującego każdy z obiektów formularza, kod służący do utworzenia tego obiektu oraz kod do wyświetlenia go na formularzu. Spróbujmy odszukać te wiersze kodu, byśmy mogli dokładnie sprawdzić, co się dzieje w formularzu.

- 1 Otwórz projekt *Prosty edytor tekstów*, który napisałeś w rozdziale 9., i otwórz plik *Form1.Designer.cs*. Przewiń jego zawartość w dół i odzyskaj deklaracje pól. Powinieneś ujrzeć po jednej deklaracji pola dla każdej kontrolki umieszczonej na formularzu.

```
Windows Form Designer generated code

private System.Windows.Forms.OpenFileDialog openFileDialog1;
private System.Windows.Forms.SaveFileDialog saveFileDialog1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.Button open;
private System.Windows.Forms.Button save;
private System.Windows.Forms.TableLayoutPanel tableLayoutPanel1;
private System.Windows.Forms.FlowLayoutPanel flowLayoutPanel1;
```

- 2 Rozwiń sekcję kodu wygenerowanego przez projektanta formularzy i odzyskaj kod tworzący kontrolki:

IDE mogło wygenerować te wiersze kodu w nieco innej kolejności, niemniej jednak na pewno znajdzie się tu po jednym wierszu kodu dla każdej kontrolki umieszczonej na formularzu.

```
private void InitializeComponent()
{
    this.openFileDialog1 = new System.Windows.Forms.OpenFileDialog();
    this.saveFileDialog1 = new System.Windows.Forms.SaveFileDialog();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.open = new System.Windows.Forms.Button();
    this.save = new System.Windows.Forms.Button();
    this.tableLayoutPanel1 = new System.Windows.Forms.TableLayoutPanel();
    this.flowLayoutPanel1 = new System.Windows.Forms.FlowLayoutPanel();
}
```

- 3 Kontrolki takie jak `TableLayoutPanel` oraz `FlowLayoutPanel`, mogące zawierać inne kontrolki, dysponują właściwością o nazwie **Controls**. Jest to obiekt typu `ControlCollection`, który pod bardzo wieloma względami przypomina obiekt typu `List<Control>`. Każda kontrolka umieszczana na formularzu jest obiektem klasy pochodnej klasy `Control`, a dodanie jej do kolekcji `Controls` panelu spowoduje, że zostanie ona wyświetlona wewnątrz niego. Przewiń zawartość pliku w dół, do miejsca, w którym do panelu `FlowLayoutPanel` dodawane są przyciski *Zapisz* i *Otwórz*:

```
this.flowLayoutPanel1.Controls.Add(this.save);
this.flowLayoutPanel1.Controls.Add(this.open);
```

Panel `FlowLayoutPanel` został z kolei umieszczony w komórce panelu `TableLayoutPanel`. Odszukaj miejsce kodu, w którym on oraz pole tekstowe są dodawane do panelu zewnętrznego:

```
this.tableLayoutPanel1.Controls.Add(this.textBox1, 0, 0);
this.tableLayoutPanel1.Controls.Add(this.flowLayoutPanel1, 0, 1)
```

Także sam obiekt `Form` jest pojemnikiem, do którego zostaje dodany obiekt panelu `TableLayoutPanel`:

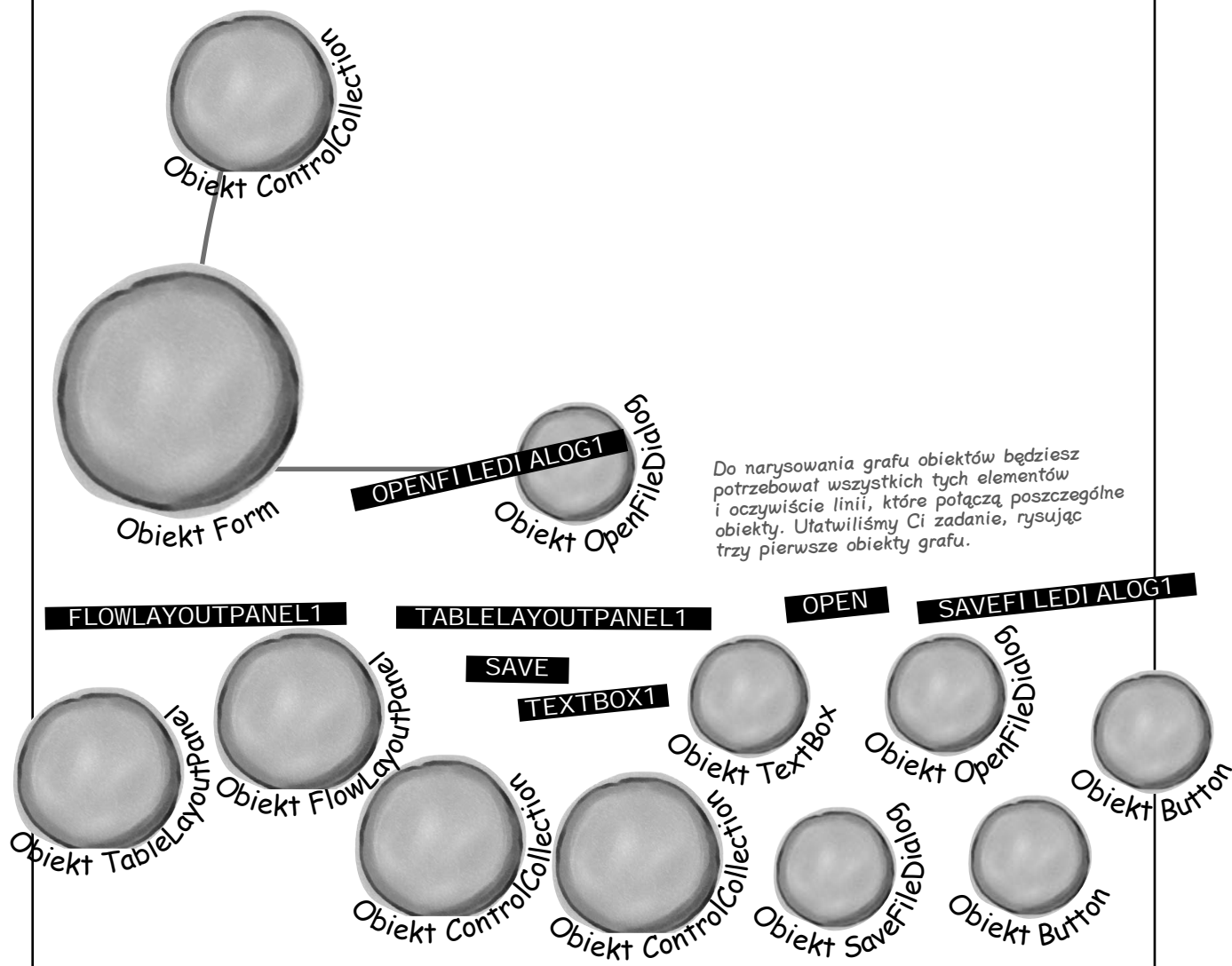
```
this.Controls.Add(this.tableLayoutPanel1);
```


Będziesz musiał otworzyć plik Form1.Designer.cs z projektu Prostego edytora tekstów, który napiszesz w rozdziale 9.

Zaostrz ołówek



Przyjrzyj się instrukcjom new oraz wywołaniu metody Controls.Add() wygenerowanym przez IDE w formularzu Prostego edytora tekstu i narysuj graf obiektów tworzonych podczas wykonywania tego kodu.



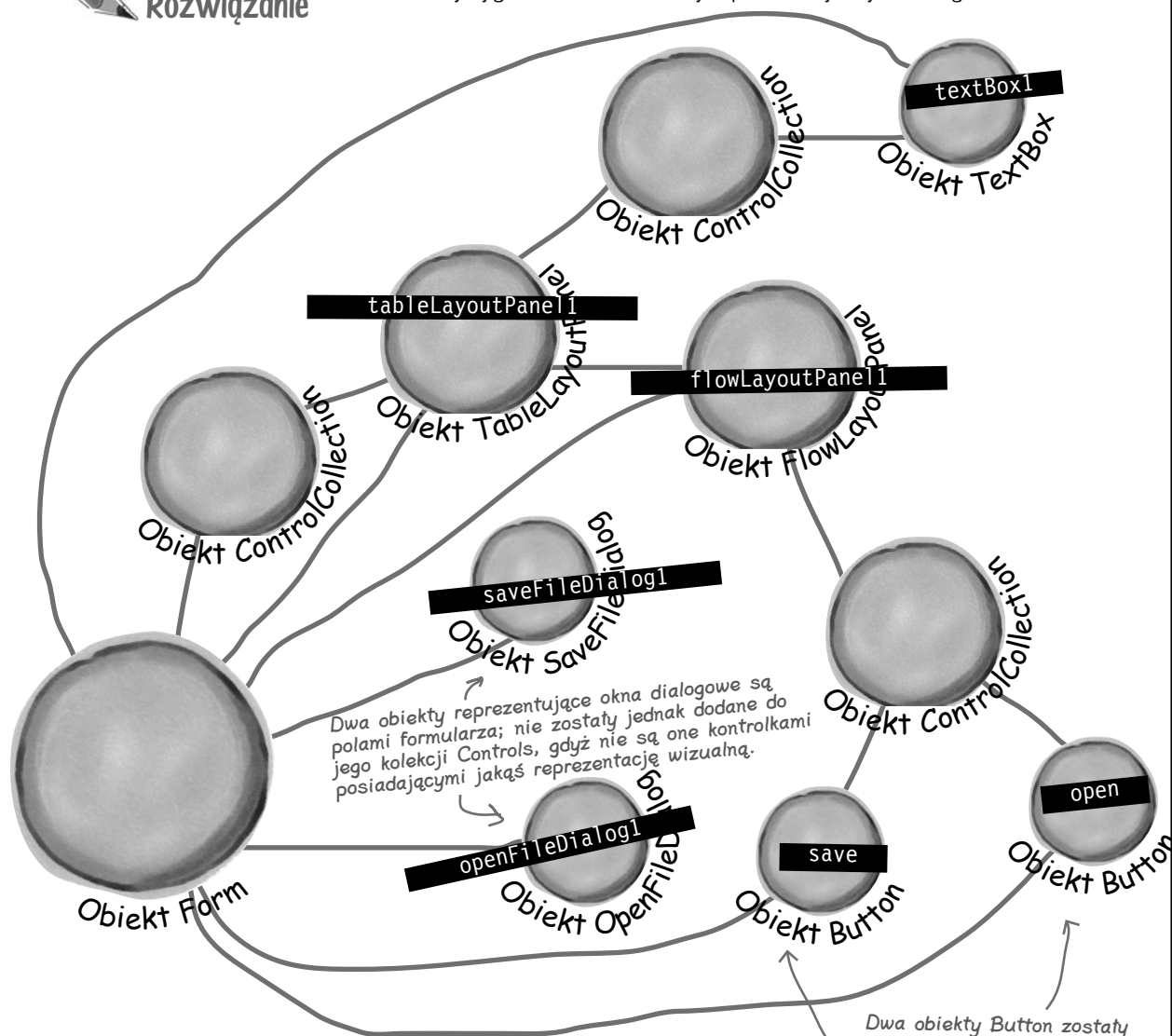
Formularze WinForms są świetnym narzędziem dydaktycznym

Zaostrz ołówek



Rozwiązanie

Przyjrzyj się instrukcjom new oraz wywołaniom metody `Controls.Add()` wygenerowanym przez IDE w formularzu Prostego edytora tekstu i narysuj graf obiektów tworzonych podczas wykonywania tego kodu.



Dwa obiekty reprezentujące okna dialogowe są polami formularza; nie zostały jednak dodane do jego kolekcji Controls, gdyż nie są one kontrolkami posiadającymi jakąś reprezentację wizualną.

Dwa obiekty Button zostały dodane do kolekcji Controls panelu FlowLayoutPanel, dlatego też zawiera on referencje do każdego z nich. Referencje do nich są także zapisane w polach `open` i `save` formularza.

To Ci się naprawdę przyda, kiedy będziesz wykonywał ćwiczenia. Przyjrzyj się także drugiej metodzie dostępnej w klasie `Debug`.

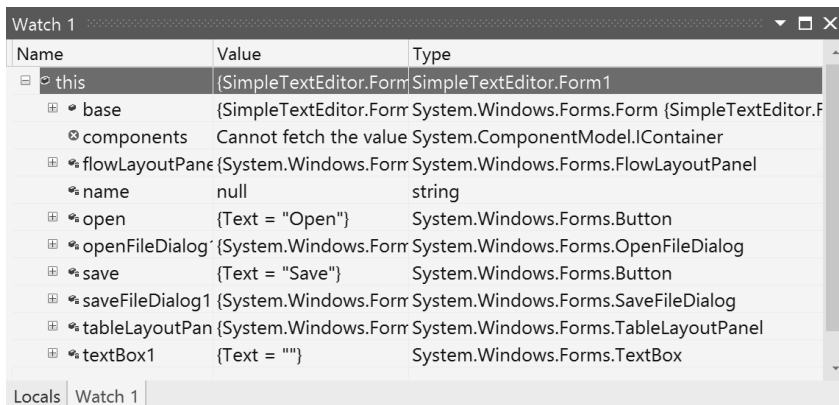


Podpowiedź do debugowania

Metoda `System.Diagnostics.Debug.WriteLine()` pozwala wyświetlać tekst w oknie wyników w trakcie sesji debugowania. Można jej używać tak samo jak metody `Console.WriteLine()` w aplikacjach Windows Forms.

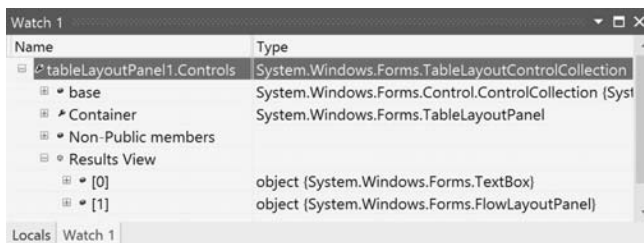
Użyj IDE do przejrzania grafu obiektów

Wróć do projektu Prostejgo edytora tekstu i umieść punkt przerwania na wywołaniu metody `InitializeComponent()`, w konstruktorze formularza. Następnie uruchom debugowanie programu. Gdy zostanie ono przerwane w punkcie przerwania, naciśnij klawisz `F10`, by wejść do metody, po czym **przejdź do okna Watch i wpisz this**, by wyświetlić i przejrzeć graf obiektów.



Kliknij przycisk „+” widoczny z lewej strony „this”, by rozwinąć zawartość obiektu i wyświetlić wszystkie jego pola, wygenerowane przez IDE w celu przechowywania referencji do kontrolki umieszczonej na formularzu.

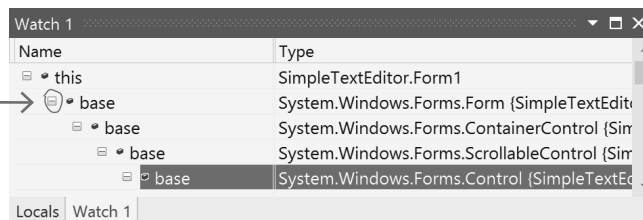
Dodaj do okna *Watch* pole `tableLayoutPanel1.Controls` i rozwiń pozycję *Results View*, by zobaczyć obiekty dodane do panelu:



Kolekcja *Controls* kontrolki *TableLayoutPanel* zawiera dwie kontrolki: *TextBox* oraz *FlowLayoutPanel* zawierającą przyciski *Zapisz* i *Otwórz*.

Okazuje się, że klasa `System.Windows.Form` nie posiada właściwości `Controls`. Właściwość ta jest dziedziczona po jej klasie bazowej, `ContainerControl`, która z kolei dziedziczy ją po swojej klasie bazowej, `ScrollableControl`, a ta od swojej klasy bazowej `Control`. Rozwijaj pozycję `base` w oknie *Watch*, aby przejść hierarchię dziedziczenia, docierając aż do klasy `System.Windows.Forms.Control`. (To właśnie po niej klasa `Form` dziedziczy kolekcję `Controls`). Następnie rozwiń pozycję *Results View* kolekcji `Controls`. Znajdziesz w niej po jednym obiekcie dla każdej kontrolki umieszczonej na formularzu!

Rozwiń pozycję „base”, by wyświetlić właściwości, które obiekt odziedziczył po swojej klasie bazowej:



To już jest Twoje ostatnie spotkanie z aplikacjami WinForms na kilka najbliższych rozdziałów! Wrócimy do nich jeszcze kilka razy, gdyż są one naprawdę doskonałymi narzędziami do nauki i poznawania języka C#.

Aplikacje dla Sklepu Windows używają XAML do tworzenia obiektów interfejsu użytkownika



Używając kodu XAML do tworzenia interfejsu użytkownika aplikacji przeznaczonych dla Sklepu Windows, w rzeczywistości tworzymy graf obiektów. Podobnie jak w aplikacjach WinForms, także i w tym przypadku można skorzystać z IDE oraz jego okna *Watch* do przeglądnięcia tych obiektów. Otwórz program, którego w **rozdziale 2. używaliśmy do „zabaw z instrukcją if-else”**. Następnie otwórz plik *MainPage.xaml.cs*, umieść punkt przerwania w konstruktorze, w wierszu zawierającym wywołanie metody `InitializeComponent()`, a następnie skorzystaj z możliwości IDE, by zbadać utworzone w aplikacji obiekty interfejsu użytkownika.

1 Uruchom debugowanie, a następnie naciśnij klawisz *F10*, aby wejść do metody `InitializeComponent()`. Visual Studio 2012 for Windows 8 posiada nieco inny układ okien od Visual Studio przeznaczonego do tworzenia tradycyjnych aplikacji, gdyż dysponuje większymi możliwościami; na przykład pozwala na otwieranie wielu okien *Watch* (co przydaje się, kiedy chcemy śledzić większą liczbę danych). Wyświetl to okno, wybierając z menu opcję *DEBUG/Windows/Watch/Watch 1*, a następnie wpisz w nim `this`:

Name	Type
• this	Chapter2Program2.MainPage
• base	Windows.UI.Xaml.Controls.Page {Chapter2Program2.MainPage}
• _contentLoaded	bool
• changeText	Windows.UI.Xaml.Controls.Button
• enableCheckbox	Windows.UI.Xaml.Controls.CheckBox
• labelToChange	Windows.UI.Xaml.Controls.TextBlock

2 A teraz przyjrzyj się kodowi XAML definiującemu postać strony:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>

  <Button x:Name="changeText" Content="Kliknięcie zmienia etykietę"
    HorizontalAlignment="Center" Click="changeText_Click"/>

  <CheckBox x:Name="enableCheckbox" Content="Włącza zmianę etykiety"
    HorizontalAlignment="Center" IsChecked="true" Grid.Column="1"/>

  <TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
    Text="Naciśnij przycisk, aby zmienić tekst"
    HorizontalAlignment="Center" VerticalAlignment="Center"
    Grid.ColumnSpan="2"/>
</Grid>
```

Kod XAML definiujący kontrolki umieszczone na stronie zostaje przekształcony w obiekt `Page`, którego pola i właściwości zawierają referencje do kontrolki interfejsu użytkownika.

3 Dodaj do okna *Watch* kilka właściwości kontrolki `labelToChange`:

Name	Value	Type
<code>labelToChange.Text</code>	"Naciśnij przycisk, aby zmienić tekst"	string
<code>labelToChange.HorizontalAlignment</code>	Center	Windows.UI.Xaml.HorizontalAlignment
<code>labelToChange.VerticalAlignment</code>	Center	Windows.UI.Xaml.VerticalAlignment
<code>labelToChange.TextWrapping</code>	Wrap	Windows.UI.Xaml.TextWrapping

Aplikacja automatycznie określi wartości właściwości na podstawie kodu XAML:

```
<TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
    Text="Naciśnij przycisk, aby zmienić tekst"
    HorizontalAlignment="Center" VerticalAlignment="Center"
    Grid.ColumnSpan="2"/>
```

Spróbuj jednak dodać do okna *Watch* właściwość `labelToChange.Grid` lub `labelToChange.ColumnSpan`. Kontrolka `labelToChange` jest typu `Windows.UI.Controls.TextBlock`, a klasa ta nie deklaruje żadnej z tych właściwości. Czy jesteś w stanie odgadnąć, co się dzieje w tym kodzie XAML?

4 Zatrzymaj program, otwórz plik *MainPage.xaml.cs* i odszukaj w nim deklarację klasy `MainPage`. Przyjrzyj się jej deklaracji — jak widać, dziedziczy ona po klasie `Page`. Umieść wskaźnik myszy nad słowem `Page`, tak by IDE wyświetliło pełną nazwę klasy:

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
    }
}
```

Przesuń wskaźnik myszy nad słowo `Page`, by wyświetlić nazwę klasy.

```
class Windows.UI.Xaml.Controls.Page
Encapsulates a page of content that can be navigated to.
```

Teraz ponownie uruchom program i naciśnij klawisz *F10*, by wejść do metody `InitializeComponent()`. Przejdź do okna *Watch* i rozwiń elementy `this`, następnie `base` i jeszcze raz `base`, przechodząc tym samym nieco w górę hierarchii dziedziczenia.

Name	Type
• this	Chapter2Program2.MainPage
• base	Windows.UI.Xaml.Controls.Page {Chapter2Program2.MainPage}
• base	Windows.UI.Xaml.Controls.UserControl {Chapter2Program2.MainPage}
• base	Windows.UI.Xaml.Controls.Control {Chapter2Program2.MainPage}
• Content	Windows.UI.Xaml.UIElement {Windows.UI.Xaml.Controls.Grid}
• Static members	

Rozwiń te elementy, by wyświetlić klasy bazowe.

Rozwiń element `Content` i sprawdź jego węzeł (`Windows.UI.Xaml.Controls.Grid`).

Poświęć chwilę, by dokładnie zbadać obiekty wygenerowane na podstawie kodu XAML. Przyjrzymy się im nieco dokładniej w dalszej części książki. Na razie po prostu je przejrzyj, byś uświadomił sobie, jak wiele obiektów tworzy Twoją aplikację.

Przeprojektuj formularz Idź na ryby!, zmieniając go w aplikację dla Sklepu Windows

Gra Idź na ryby!, którą napisałeś w rozdziale 8., byłaby fantastyczną aplikacją dla Sklepu Windows. Uruchom zatem Visual Studio 2012 for Windows 8 i utwórz w nim nowy projekt **Windows Store** (taki sam jak w przypadku aplikacji *Ratuj ludzi*). Na kilku następujących stronach przeprojektujesz ten formularz w formie strony XAML, która będzie potrafiła dostosować się do urządzeń z ekranami o różnych rozmiarach. Zamiast korzystać z kontrolki Windows Forms umieszczanych na formularzu, skorzystasz z kontrolki charakterystycznych dla aplikacji przeznaczonych dla Sklepu Windows, umieszczanych na stronie.

Zrób to!

To zostaje przekształcone w element `<TextBox/>`

To zostaje przekształcone w element `<Button/>`

Użyjemy panelu `StackPanel` o układzie poziomym, aby zgrupować kontrolki `TextBox` oraz `Button`, tak by można je było wyświetlić w jednej komórce układu.

To zostaje przekształcone w element `<ListView/>`

To zostaje przekształcone w element `<ScrollView/>`

To zostaje przekształcone w element `<ScrollView/>`

To jest kolejna kontrolka dostępna w przyborniku. Prezentuje ona sekwencję tańców znaków, dodając do nich pionowe i poziome paski przewijania, jeśli tekst będzie większy od rozmiarów kontrolki.

To zostaje przekształcone w element `<Button/>`

Tak te kontrolki będą wyglądać na głównej stronie aplikacji:



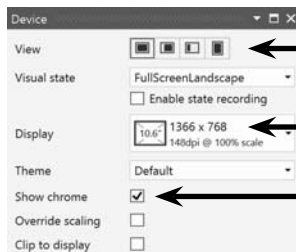
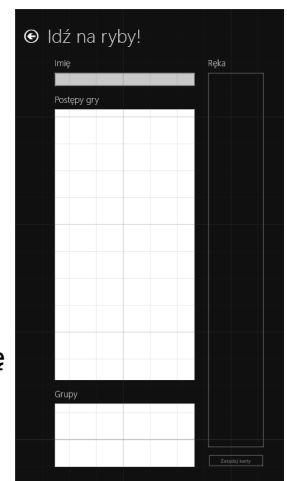
Większość kodu obsługującego przebieg gry pozostanie w niezmienionej postaci, zmieni się natomiast kod obsługujący interfejs aplikacji.



Kontrolki te zostaną umieszczone w siatce, której wiersze i kolumny będą się zmniejszać i powiększać w zależności od wielkości ekranu. Dzięki temu gra będzie mogła się zmniejszać i powiększać, dostosowując się do ekranu. Do określania różnych konfiguracji ekranu możesz skorzystać z okna *Device* dostępnego w IDE.



Grą będzie się można bawić niezależnie od rozmiarów strony.



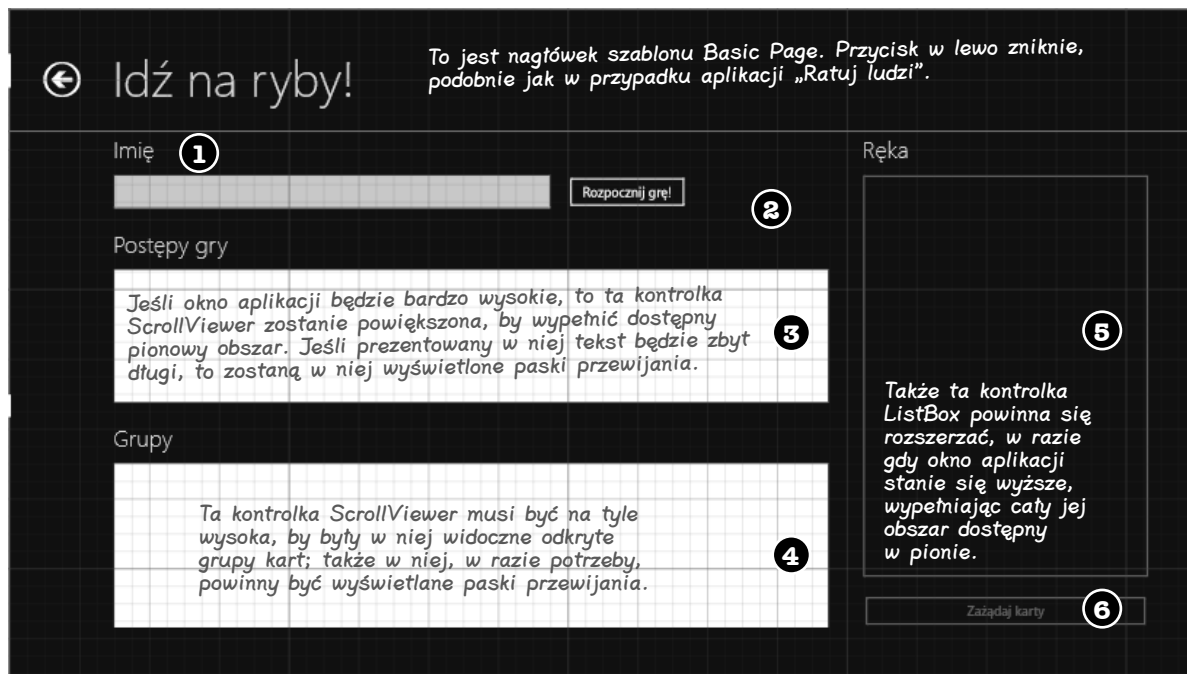
Przyciski View pozwalają wyświetlić stronę w układzie poziomym, pionowym oraz w układzie podzielonego ekranu.

Różne opcje dostępne na liście Display pozwalają wyświetlić stronę przy wykorzystaniu różnych rozdzielczości oraz proporcji ekranu.

Usuń zaznaczenie pola „Show chrome”, by ukryć obrazek reprezentujący oprawę ekranu urządzenia.

Określanie postaci strony rozpoczyna się od dodania kontrolek ✨

Technologie XAML oraz WinForms mają jedną wspólną cechę: w obu do określania postaci interfejsu użytkownika używane są kontrolki. Strona gry *Idź na ryby!* posiada dwa przyciski, kontrolkę `ListBox` służącą do wyświetlania kart w ręce, pole `TextBox`, w którym użytkownik może wpisać swoje imię, oraz cztery etykiety `TextBlock`. Interfejs użytkownika dopełniają dwie kontrolki `ScrollViewer` o białym tle, służące do prezentowania postępów gry oraz odłożonych grup.



Szablon *Basic Page* zawiera siatkę składającą się z dwóch wierszy. Jej górny wiersz zawiera nagłówek z nazwą aplikacji. Z kolei w drugim wierszu umieszczony jest obszar treści, zdefiniowany przez poniższą siatkę. Cała ta siatka jest umieszczona w wierszu 1. układu (oraz w jego jedynej kolumnie o numerze 0).

```
<Grid Grid.Row="1" Margin="120,0,60,60">
```

```
<TextBlock Text="Imię" Margin="0,0,0,20"
```

```
1 <Style="{StaticResource SubheaderTextStyle}"/>
```

Marginesy określają wcięcie siatki, dzięki czemu jest odsunięta od krawędzi strony. Lewy margines zawsze wynosi 120 pikseli.

Oto znacznik otwierający, rozpoczynający siatkę.

Zastosujemy kontrolkę typu `StackPanel`, aby w jednej komórce siatki umieścić pole `TextBox` do podania nazwy gracza oraz przycisk rozpoczynający grę:

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
```

```
<TextBox x:Name="playerName" FontSize="24"
```

```
Width="500" MinWidth="300" />
```

```
2 <Button x:Name="startButton" Margin="20,0" Content="Rozpocznij grę!"/>
```

```
</StackPanel>
```

Ta właściwość dodaje wokół kontrolki `TextBox` i `Button` margines o szerokości 20 pikseli. Kiedy właściwość `Margin` zawiera dwie liczby, określają one, odpowiednio: marginesy w poziomie (lewy i prawy) oraz w pionie (górny i dolny).



Każda etykieta umieszczona na stronie („Imię”, „Postępy gry” itd.) jest kontrolką `TextBlock` posiadającą niewielki margines u góry oraz u dołu, jak również określoną wartość właściwości `SubheaderTextStyle`:

```
<TextBlock Text="Postępy gry"
  Style="{StaticResource SubheaderTextStyle}"
  Margin="0,20,0,20" Grid.Row="2"/>
```

Kontrolka `ScrollViewer` wyświetla postępy gry i dysponuje paskami przewijania, które zostaną wyświetlone, kiedy tekst stanie się zbyt długi i przestanie się mieścić w obszarze kontrolki:

```
3 <ScrollViewer Grid.Row="3" FontSize="24"
  Background="White" Foreground="Black" />
```

Oto kolejne kontrolki, `TextBlock` oraz `ScrollViewer`, służące do wyświetlania grup kart. Domyślną wartością wyrównania kontrolki `ScrollViewer` w pionie oraz w poziomie jest `Stretch` i okaże się, że jest ona naprawdę przydatna. Skonfigurujemy wiersze i kolumny siatki w taki sposób, by kontrolki `ScrollViewer` rozszerzały się, dostosowując się do ekranów o różnych wielkościach.

```
<TextBlock Text="Grupy" Style="{StaticResource SubheaderTextStyle}"
  Margin="0,20,0,20" Grid.Row="4"/>
```

```
4 <ScrollViewer FontSize="24" Background="White" Foreground="Black"
  Grid.Row="5" Grid.RowSpan="2" />
```

Pośrodku siatki dodaliśmy kolumnę o szerokości 40 pikseli, by zapewnić odstęp pomiędzy kontrolkami w obu kolumnach. Oznacza to także, że pozostałe kontrolki — `ListBox` oraz `Button` — muszą zostać umieszczone w trzeciej kolumnie. Kontrolka `ListBox` zajmuje wiersze do 2. do 6., a zatem zastosowaliśmy w niej właściwości `Grid.Row="1"` oraz `Grid.RowSpan="5"`. Dzięki temu rozwiązaniu kontrolka ta będzie się powiększała i zmniejszała, dostosowując do wielkości strony.

```
5 <TextBlock Text="Ręka" Style="{StaticResource SubheaderTextStyle}"
  Grid.Row="0" Grid.Column="2" Margin="0,0,0,20"/>
<ListBox x:Name="cards" Background="White" FontSize="24" Height="Auto"
  Margin="0,0,0,20" Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"/>
```

Pamiętaj, że numeracja wierszy i kolumn rozpoczyna się od zera, a zatem w przypadku kontrolki umieszczonej w trzeciej kolumnie należy użyć właściwości `Grid.Column="2"`.

W przycisku *Zażądaj karty*, we właściwościach określających wyrównanie w pionie i w poziomie, zastosowaliśmy wartość `Stretch`, dzięki czemu wypełni on cały obszar komórki. Dodany do listy margines dolny o wysokości 20 pikseli zapewni niewielki odstęp pomiędzy listą i przyciskiem.

```
6 <Button x:Name="askForACard" Content="Zażądaj karty"
  HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
  Grid.Row="6" Grid.Column="2"/>
```

Wiersze i kolumny mogą zmieniać wielkość, dostosowując się do rozmiarów strony

Siatki są niezwykle efektywnym narzędziem do określania układu stron, gdyż ułatwiają projektowanie stron, które mogą być wyświetlane na wielu różnych urządzeniach. Wysokości oraz szerokości zakończone znakiem * są **automatycznie dostosowywane** do ekranów o różnej geometrii. Strona gry Idź na ryby! składa się z trzech kolumn. Pierwsza ma szerokość 5*, a trzecia 2*, a zatem będą się **proporcjonalnie** kurczyły i rozszerzały, zachowując zawsze proporcje 5:2. Druga kolumna ma ustaloną szerokość wynoszącą 40 pikseli i służy do wizualnego oddzielenia dwóch pozostałych kolumn. Poniżej zamieściliśmy schemat przedstawiający układ wierszy i kolumn strony (oraz kontrolki umieszczone w komórkach siatki):

	<ColumnDefinition Width="5*" />	<ColumnDefinition Width="40" />	<ColumnDefinition Width="2*" />
<RowDefinition Height="Auto" />	<TextBlock /> Row="1" oznacza drugi wiersz, gdyż ich numeracja zaczyna się od 0. ↓		<TextBlock Grid.Column="2" />
<RowDefinition Height="Auto" />	<StackPanel Grid.Row="1"> <TextBlock /> <Button /> </StackPanel >		<ListBox Grid.Column="2" Grid.RowSpan="5" /> ↑ Ta kontrolka ListBox zajmuje pięć kolejnych wierszy, w tym także wiersz czwarty, który będzie się powiększał, zajmując cały wolny obszar strony. Dzięki temu lista będzie się powiększać, zajmując całą prawą kolumnę strony.
<RowDefinition Height="Auto" />	<TextBlock Grid.Row="2" />		
<RowDefinition />	<ScrollView Grid.Row="3" /> Ten wiersz ma domyślną wysokość 1*, a umieszczona w tej komórce kontrolka ScrollView ma domyślne wartości wyrównania w poziomie i w pionie, czyli Stretch. Oznacza to, że będzie się powiększać i zmniejszać, dostosowując się do wielkości strony.		
<RowDefinition Height="Auto" />	<TextBlock Grid.Row="4" />		
<RowDefinition Height="Auto" MinHeight="150" />	<ScrollView Grid.Row="5" Grid.RowSpan="2" /> ↑ W tej kontrolce ScrollView użyliśmy właściwości Grid.RowSpan o wartości "2", by zajmowała ona dwa sąsiadujące wiersze. Umieściliśmy ją w szóstym wierszu (czyli właściwość Grid.Row ma wartość "6", gdyż numeracja wierszy w języku XAML zaczyna się od 0), którego minimalna wysokość wynosi 150; dzięki czemu mamy pewność, że kontrolka ScrollView nie będzie niższa od tej wartości.		
<RowDefinition Height="Auto" />			<Button Grid.Row="6" Grid.Column="2" /> ↑

Numeracja wierszy i kolumn w języku XAML rozpoczyna się od 0, dlatego też w tej kontrolce Button wiersz został określony jako 6, a kolumna jako 2 (aby pominąć kolumnę środkową). Właściwości określające wyrównanie kontrolki w pionie i poziomie mają wartość Stretch, dzięki czemu przycisk zajmie cały obszar komórki. Wysokość wiersza została określona jako Auto, dzięki czemu zostanie ona dostosowana do wymiarów zawartości (czyli przycisku oraz jego marginesów).

Oto jak wygląda definicja wierszy i kolumn dla takiego układu strony:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="5*" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="2*" />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" MinHeight="150" />
    <RowDefinition Height="Auto" />
</Grid.RowDefinitions>
```

Pierwsza kolumna zawsze będzie 2,5 razy szersza od trzeciej (proporcja 5:2), a obie będą od siebie oddzielone drugą kolumną o stałej szerokości 40 pikseli. W kontrolkach `ScrollViewer` oraz `ListBox`, używanych do prezentacji danych, właściwość `HorizontalAlignment` przypisano wartość "Stretch", dzięki czemu wypełnią one całą szerokość kolumn.

Czwarty wiersz ma domyślną wysokość 1*, dzięki czemu będzie się powiększał i zmniejszał, zajmując cały dostępny obszar, który nie został zajęty przez pozostałe wiersze. Kontrolki `ListBox` oraz `ScrollViewer` są wyświetlone także w tym wierszu, dzięki czemu także one będą się powiększać i zmniejszać.

Wysokość niemal wszystkich wierszy ma wartość `Auto`. Tylko jeden wiersz siatki będzie się powiększał i zmniejszał i wszystkie kontrolki wyświetlone w tym wierszu będą się zachowywały podobnie.

Definicje wierszy i kolumn można dodawać wewnątrz siatki, powyżej lub poniżej tego kodu. Tym razem dodaliśmy je poniżej.

```
</Grid>
```

Oto znacznik zamykający siatki. Cały kod XAML strony połączysz w całość pod koniec rozdziału, kiedy zakończysz przerabianie gry na aplikację dla Sklepu Windows.

Skorzystaj z systemu siatki, by określić układ stron aplikacji

Czy kiedykolwiek zauważyłeś, że różne aplikacje przeznaczone dla Sklepu Windows mają podobny wygląd? Dzieje się tak dlatego, że korzystają z **systemu siatki**, by nadać aplikacjom coś, co projektanci z firmy Microsoft określają mianem „spójnego zarysu”. Siatka składa się z kwadratów, nazywanych *jednostkami* (ang. *units*) oraz *podjednostkami* (ang. *subunits*) — miałeś już okazję się z nimi spotkać, gdyż są one wykorzystywane w Visual Studio IDE.

Jeśli już w rozdziale 1. nie użyłeś tych przycisków, umieszczonych u dołu okna Designer i służących, odpowiednio, do wyświetlania linii siatki, przyciągania oraz przyciągania do siatki, to skorzystaj z nich teraz.

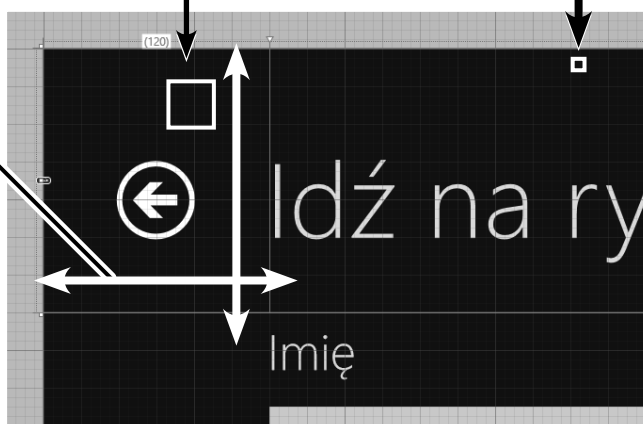


Nagłówek strony powinien mieć wysokość 7 jednostek, tekst powinien się zaczynać 6 jednostek od lewej krawędzi strony, a jego dolna krawędź powinna się znajdować na wysokości 5 jednostek poniżej górnej krawędzi.

Wiersz nagłówka jest automatycznie dodawany do szablonu Basic Page, który zastosujesz w dalszej części rozdziału, kiedy gra Idź na ryby! będzie już działać.

Siatka składa się z kwadratów o wielkości 20x20 pikseli, nazywanych jednostkami.

Każda jednostka jest podzielona na podjednostki o wielkości 5x5 pikseli.



W aplikacji Idź na ryby! w kontrolce <Grid> zawierającej wszystkie pozostałe kontrolki aplikacji skorzystamy z właściwości `Margin`, by stworzyć odstęp od krawędzi strony. Wartością tej właściwości może być: jedna liczba (określająca wielkość wszystkich czterech marginesów: lewego, górnego, prawego i dolnego), dwie liczby (pierwsza z nich określa wielkość marginesu lewego i prawego, a druga: górnego i dolnego) bądź też cztery liczby określające, odpowiednio, wielkość marginesu lewego, górnego, prawego i dolnego.

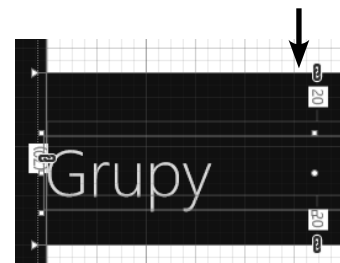
Twoja aplikacja Idź na ryby! ma lewy margines o wielkości 120 pikseli (czyli 6 jednostek), nie ma górnego marginesu, natomiast prawy i dolny margines mają wielkość 60 pikseli (3 jednostek):

```
<Grid Grid.Row="1" Margin="120,0,60,60">
```

Oprócz tego pomiędzy etykietami oraz innymi elementami strony także zostały dodane marginesy o wielkości jednej jednostki:

```
<TextBlock Text="Books"
  Style="{StaticResource SubheaderTextStyle}"
  Margin="0,20,0,20" Grid.Row="4"/>
```

Strona używa marginesów, by dodać odstęp o wielkości jednej jednostki pomiędzy etykietami i pozostałymi elementami.



Nie istnieją
grupy pytań

P: Jaki jest efekt przypisania wysokości wiersza lub szerokości kolumny wartości „Auto”?

U: Przypisanie wartości `Auto` właściwości `Height` wiersza lub właściwości `Width` kolumny powoduje, że wielkość tego wiersza lub kolumny będzie się zmieniać i dostosowywać do wymiarów jego zawartości. Samemu możesz spróbować, jak to działa. Utwórz nowy projekt `Blank App` i zmodyfikuj siatkę na stronie `MainPage.xaml`, dodając do niej kilka wierszy i kolumn, których wysokości i szerokości zostały określone jako `Auto`. W oknie `Designer` nie zobaczysz niczego, gdyż wiersze i kolumny są puste, więc zostały zmniejszone do wysokości i szerokości 0 pikseli. Kiedy jednak dodasz jakieś kontrolki do różnych wierszy i kolumn, przekonasz się, że powiększyły się one, dostosowując się do wymiarów tych kontrolki.

P: A zatem czym to się różni od określenia wysokości wiersza lub szerokości kolumny jako `1*`, `2*` lub `5*`?

U: Zastosowanie `*` w określeniu wysokości wiersza lub szerokości kolumny sprawia, że wiersze lub kolumny będą powiększane proporcjonalnie, wypełniając cały obszar siatki. Jeśli siatka zawiera trzy kolumny o szerokościach `3*`, `3*` i `4*`, to każda z dwóch pierwszych kolumn będzie zajmowała 30% szerokości całej siatki pomniejszonej o szerokości kolumn o ustalonej lub automatycznie wyznaczonej (`Auto`) szerokości; trzecia kolumna (`4*`) będzie zajmowała 40% szerokości siatki.

Jest pewien powód, dla którego domyślna szerokość i wysokość określona jako `1*`, ma sens. Jeśli wszystkie wiersze i kolumny będą miały te domyślne wartości, to niezależnie od wielkości siatki wielkości poszczególnych wierszy i kolumn zawsze będą równe.

P: „Piksele”. Ciągłe używanie tego słowa. Nie sądzę, że oznacza ono to, co wy uważacie, że ono oznacza.

U: Wiele osób korzystających z języka XAML używa terminu *piksel*, ale masz rację — z technicznego punktu widzenia nie używasz tych samych pikseli, które widzisz na ekranie. Technicznym określeniem liczb podawanych

jako wartości właściwości `Margin`, `Height`, `Width` oraz innych jest: **jednostki niezależne od urządzenia**. Aplikacje przeznaczone dla Sklepu Windows muszą działać na ekranach o różnych wielkościach i kształtach, a zatem niezależnie od tego, jak duży lub mały jest ekran urządzenia, na którym została uruchomiona aplikacja, jedna jednostka niezależna od urządzenia zawsze będzie miała wielkość 1/96 cala. Kwadrat pięć na pięć takich jednostek tworzy jedną podjednostkę układu strony, a kwadrat cztery na cztery podjednostki tworzą jedną jednostkę układu strony. Te wszystkie jednostki układu oraz jednostki niezależne od urządzenia są nieco mylące, dlatego też te drugie będziemy nazywali *pikselami*.

Wszystkie wysokości oraz szerokości w języku XAML można także wyrazić w innych jednostkach, dodając do liczb odpowiednie określenia: `in` (cale), `cm` (centymetry) lub `pt` (punkty, chodzi tu o punkty typograficzne o wielkości 1/72 cala). Spróbuj określić układ strony, używając cali lub centymetrów. A następnie weź linijkę i sprawdź, czy system Windows nadał kontrolkom prawidłowe wymiary.

P: Czy jest jakiś prosty sposób, by mieć pewność, że moja aplikacja będzie dobrze wyglądać na różnych monitorach?

U: Tak, IDE udostępnia przydatne narzędzie, które zapewnia takie możliwości. Okno `Designer` pozwala sprawdzać, jak projektowane strony XAML będą wyglądały na różnych urządzeniach, i to na kilka sposobów. Można skorzystać z okna `Device`, by wyświetlić stronę w różnych rozdzielczościach oraz trybach podziału ekranu. W dalszej części książki pokazemy, jak można uruchamiać aplikację w symulatorze, zapewniającym możliwość prowadzenia z nią interakcji w symulowanych urządzeniach wyposażonych w ekrany o różnej wielkości.

Kiedy szerokość wiersza lub wysokość kolumny ma wartość `Auto`, oznacza to, że ich wymiary będą dostosowywane do zawartości.

Więcej informacji na temat określania układów stron można znaleźć w witrynie MSDN, na stronie <http://msdn.microsoft.com/pl-pl/library/windows/apps/hh872191.aspx>.



Ćwiczenia

Użyj języka XAML, by przeprojektować każdą z klasycznych aplikacji Windows na aplikację przeznaczoną dla Sklepu Windows. Dla każdej z nich utwórz nowy projekt Blank App i zmień stronę główną na stronę utworzoną przy użyciu szablonu Basic Page (dokładnie tak samo jak w aplikacji *Ratuj ludzi*). Następnie zmodyfikuj strony, wprowadzając odpowiednie zmiany w siatce i dodając do niej kontrolki. Te nowe aplikacje nie muszą działać. Wystarczy, że utworzysz odpowiedni kod XAML, tak by pasowały one wyglądem do poniższych formularzy.

Odszukaj odpowiednie miejsce w kodzie XAML strony Basic Page i dodaj tam nową kontrolkę `Grid` lub `StackPanel`, w której umieścisz pozostałe kontrolki strony. Tę nową siatkę powinieneś umieścić w drugim wierszu (`Grid.Row="1"`) nowo utworzonej strony Basic Page.

```
<Grid Style="{StaticResource LayoutRootStyle}">
  <Grid.RowDefinitions>
    <RowDefinition Height="140"/>
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid Grid.Row="1" Margin="120,0"...>
    <!-- Back button and page title -->
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition width="Auto"/>
      </Grid.ColumnDefinitions>
    </Grid>
  </Grid>
```

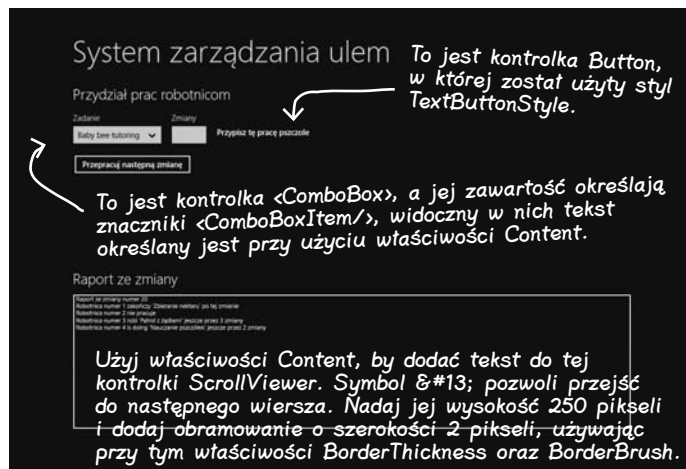
Oto kontrolka `<Grid>` umieszczona na tej stronie. Zwiniliśmy ją w edytorze XAML.

Odszukaj ten komentarz, a następnie, powyżej niego, dodaj nową kontrolkę `<Grid>`.

XAML dość elastycznie podchodzi do kolejności znaczników.

Poprosiliśmy Cię, byś dodawał kod XAML nowego układu strony poniżej definicji wierszy, gdyż dzięki temu łatwiej go będzie odnaleźć w pliku. Niektórzy programiści lubią zapisywać kod XAML w takiej samej kolejności, w jakiej poszczególne kontrolki są umieszczane na stronie. Mogą je umieszczać za zamykającym znacznikiem `</Grid>`, kończącym siatkę zawierającą przycisk ze strzałką w lewo oraz tytuł aplikacji. Radzimy, żebyś poeksperymentował, zapisując kod w różnych miejscach, gdyż warto, byś samemu określił, które miejsce będzie Ci się wydawało najbardziej intuicyjne.

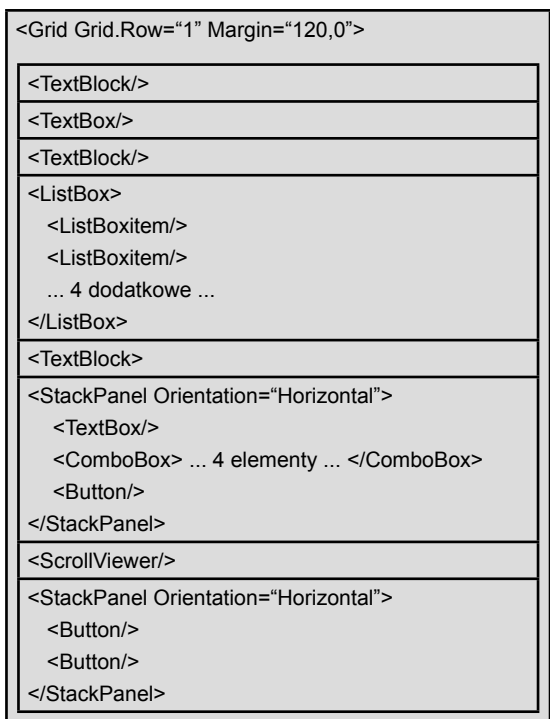
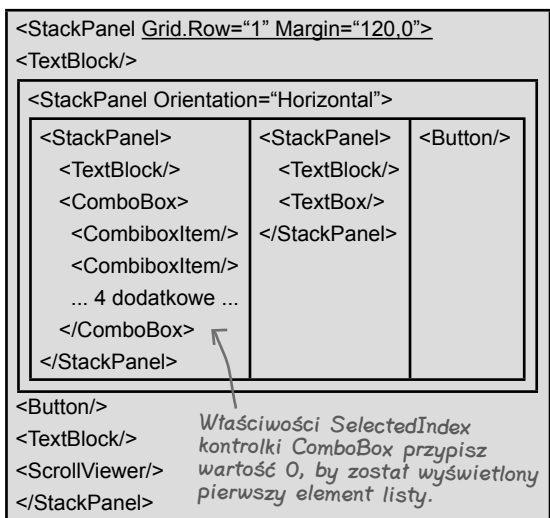
Zastosuj kontrolki `StackPanel`, by zaprojektować układ tego formularza. Składa się on z dwóch grup kontroltek. W nagłówkach tych grup został zastosowany styl `GroupHeaderTextStyle`, same grupy oddziela od siebie margines o wysokości 40 pikseli, a poniżej nagłówków został dodany margines o wysokości 20 pikseli. W etykietach umieszczonych nad kontrolkami został zastosowany styl `BodyTextStyle`, a nad kontrolkami dodano margines o wysokości 10 pikseli. Pomiędzy kontrolkami dodano poziomy margines o szerokości 20 pikseli.



Do zaprojektowania tego formularza użyj kontrolki `Grid`. Ma mieć osiem wierszy, a we wszystkich właściwość `Height` ma mieć wartość `Auto`, żeby dostosowywały się do zawartości. Zastosuj kontrolki `StackPanel`, żeby umieścić kilka kontroltek w jednym wierszu.



Zadbaj o to, by strony wyglądały tak samo jak na tych zrzutach, dodając do kontroltek **przykładowe dane**, które w normalnej aplikacji zostałyby określone przy użyciu metod i właściwości klas.



Może się zdarzyć, że kiedy spróbujesz wybrać opcję *New Item...*, by dodać do projektu nową stronę Basic Page, IDE wyświetli w oknie *Designer* komunikat o błędzie — Visual Studio będzie bowiem oczekiwać, że projekt zostanie zbudowany. W takim przypadku wybierz opcję *Rebuild Solution*, a błąd zniknie.



Rozwiązania ćwiczeń

Użyj języka XAML by przeprojektować każdą z klasycznych aplikacji Windows na aplikację przeznaczoną dla Sklepu Windows. Dla każdej z nich utwórz nowy projekt Blank App i zmień stronę główną na stronę utworzoną przy użyciu szablonu Basic Page (dokładnie tak samo jak w aplikacji *Ratuj ludzi*). Następnie zmodyfikuj strony wprowadzając odpowiednie zmiany w siatce i dodając do niej kontrolki. Te nowe aplikacje nie muszą działać. Wystarczy, że utworzysz odpowiedni kod XAML, tak pasowały one wyglądem do poniższych formularzy.

```
<StackPanel Grid.Row="1" Margin="120,0">
  <TextBlock Text="Przydział prac robotnicom"
    Style="{StaticResource GroupHeaderTextStyle}"/>
  <StackPanel Orientation="Horizontal" Margin="0,20,0,0">
    <StackPanel Margin="0,0,20,0">
      <TextBlock Text="Zadanie robotnicy" Margin="0,0,0,10"
        Style="{StaticResource BodyTextStyle}"/>
      <ComboBox SelectedIndex="0">
        <ComboBoxItem Content="Nauczanie pszczółek"/>
        <ComboBoxItem Content="Pielęgnacja jaj"/>
        <ComboBoxItem Content="Utrzymywanie ula"/>
        <ComboBoxItem Content="Wytwarzanie miodu"/>
        <ComboBoxItem Content="Zbieranie nektaru"/>
        <ComboBoxItem Content="Patrol z żądłami"/>
      </ComboBox>
    </StackPanel>
  </StackPanel>
  <StackPanel>
    <TextBlock Text="Shifts" Margin="0,0,0,10"
      Style="{StaticResource BodyTextStyle}"/>
    <TextBox/>
  </StackPanel>
  <Button Content="Przypisz to zadanie robotnicy" Margin="20,20,0,0"
    Style="{StaticResource TextButtonStyle}" />
</StackPanel>
<Button Content="Przepracuj następną zmianę" Margin="0,20,0,0" />

<TextBlock Text="Raport ze zmiany" Margin="0,40,0,20"
  Style="{StaticResource GroupHeaderTextStyle}"/>
<ScrollView BorderThickness="2" BorderBrush="White" Height="250"
  Content="
Raport zmiany numer 20&#13;
Robotnica numer 1 robi 'Zbieranie nektaru' jeszcze przez 2 zmiany&#13;
Robotnica numer 2 zakończyła swoje zadanie&#13;
Robotnica numer 2 nie pracuje&#13;
Robotnica numer 3 robi 'Patrol z żądłami' jeszcze przez 3 zmiany&#13;
Robotnica numer 4 robi 'Nauczanie pszczółek' jeszcze przez 4 zmiany
"/>
</StackPanel>
```

To jest margines, który miałeś zastosować. Opuszczenie wartości prawego i dolnego marginesu sprawi, że będą one miały takie same wielkości jak marginesy lewy i górny (120 i 0).

Czy Twój kod XAML wygląda inaczej niż ten? XAML pozwala na tworzenie stron o bardzo podobnym (lub nawet identycznym) wyglądzie na wiele różnych sposobów.

To jest nagłówek drugiej grupy, powyżej niego jest margines o wysokości 40 pikseli, a poniżej margines o wysokości 20 pikseli.

Oto przykładowy tekst, który umieściliśmy w polu raportu. Właściwość Content ignoruje znaki nowych wierszy dodawane w kodzie XAML — dodaliśmy je tutaj, by poprawić czytelność rozwiązania.


```

<Grid Grid.Row="1" Margin="120,0">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/><RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>

  <TextBlock Text="Imię drwała" Margin="0,0,0,10"
    Style="{StaticResource BodyTextStyle}"/>
  <TextBox Grid.Row="1"/>

  <TextBlock Grid.Row="2" Text="Kolejka do śniadania" Margin="0,20,0,10"
    Style="{StaticResource BodyTextStyle}"/>
  <ListBox Grid.Row="3">
    <ListBoxItem Content="1. Edek"/>
    <ListBoxItem Content="2. Zyga"/>
    <ListBoxItem Content="3. Bolek"/>
    <ListBoxItem Content="4. Ferdek"/>
    <ListBoxItem Content="5. Stachu"/>
    <ListBoxItem Content="6. Robert"/>
  </ListBox>

  <TextBlock Grid.Row="4" Text="Nakarm drwała" Margin="0,20,0,10"
    Style="{StaticResource BodyTextStyle}"/>
  <StackPanel Grid.Row="5" Orientation="Horizontal">
    <TextBox Text="2" Margin="0,0,20,0"/>
    <ComboBox SelectedIndex="0" Margin="0,0,20,0">
      <ComboBoxItem Content="chrupkiego"/>
      <ComboBoxItem Content="wilgotnego"/>
      <ComboBoxItem Content="rumianego"/>
      <ComboBoxItem Content="bananowego"/>
    </ComboBox>
    <Button Content="Dodaj naleśniki" Style="{StaticResource TextButtonStyle}"/>
  </StackPanel>

  <ScrollViewer Grid.Row="6" Margin="0,20,0,0" Content="Edek ma 7 naleśników"
    BorderThickness="2" BorderBrush="White"/>

  <StackPanel Grid.Row="7" Orientation="Horizontal" Margin="0,40,0,0">
    <Button Content="Dodaj drwała" Margin="0,0,20,0" />
    <Button Content="Następny drwał" />
  </StackPanel>
</Grid>

```

Z tych definicji usunęliśmy znaki nowych wierszy, by cały kod rozwiązania zmieścił się na jednej stronie.

Żeby wszystko było jasne: te przykładowe elementy listy miałeś dodać w ramach ćwiczenia, żeby strona przypominała wygląd używanej aplikacji. W dalszej części książki dowiesz się, jak powiązać kontrolki takie jak `ListBox` z właściwościami swoich klas.

↙ Kolejna przykładowa zawartość...

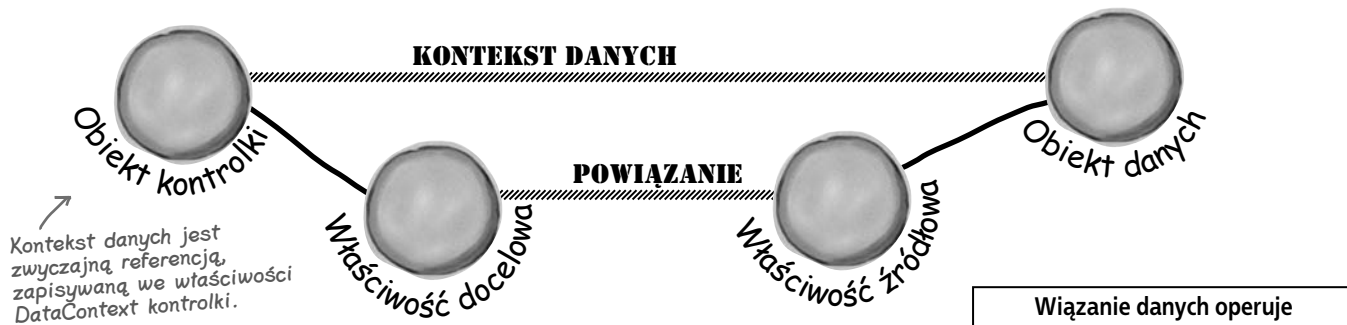


WYSIL
SZARE KOMÓRKI

Co sądzisz o układzie tej strony? Czy nie byłoby lepiej przenieść przyciski *Dodaj drwała* i *Następny drwał* na standardowy pasek aplikacji dostępny w systemie Windows 8?

Wiązanie danych kojarzy strony XAML z klasami

Kontrolki TextBox, ScrollViewer, TextBox oraz wiele innych zostały stworzone po to, by wyświetlać dane. W aplikacjach WinForms wyświetlanie danych w polach oraz dodawanie elementów do list wymagało korzystania z właściwości. W podobny sposób można także postępować w aplikacjach korzystających z języka XAML, choć istnieje także inne rozwiązanie: można skorzystać z **wiązania danych** (ang. *data binding*), by automatycznie zapisywać dane w kontrolkach umieszczonych na stronie. Co więcej, można nawet skorzystać z wiązania danych, by zapisywać w klasach dane wpisywane w kontrolkach.



Kontekst, ścieżka i wiązanie

Wiązanie danych w języku XAML jest relacją pomiędzy *właściwością źródłową* obiektu dostarczającego danych dla kontrolki oraz *właściwością docelową* kontrolki prezentującej te dane. Aby określić takie wiązanie, w **kontekście danych** kontrolki należy zapisać referencję do obiektu z danymi. Z kolei **wiązanie** musi zawierać **ścieżkę wiązania**, czyli właściwość obiektu zawierającą dane. Kiedy wszystkie te informacje zostaną określone, kontrolka będzie automatycznie odczytywać i wyświetlać wartość właściwości źródłowej.

Wiązanie danych operuje wyłącznie na właściwościach. Jeśli spróbujemy wykorzystać w tym celu publiczne pole, w kontrolce nie pojawią się żadne dane — co więcej, nie zostanie także wyświetlony żaden błąd!

Ścieżką wiązania zastosowaną w tej kontrolce TextBox jest właściwość Cash. Kontrolka będzie wyświetlać wartość tej właściwości, odczytywaną z dowolnego obiektu, z którym zostanie powiązana.

Aby określić wiązanie danych w kodzie XAML, należy określić właściwość źródłową w formie **{Binding Ścieżka}**:

```
<TextBlock x:Name="walletTextBlock" Text="{Binding Cash}"/>
```

Oprócz tego potrzebny jest obiekt, z którym kontrolka zostanie powiązana — w tym przypadku jest to obiekt Guy zapisany w zmiennej joe, którego właściwość Cash ma wartość 325.50. Kontekst jest określany przez zapisanie referencji do obiektu Guy we właściwości DataContext kontrolki.

```
Guy joe = new Guy("Józek", 47, 325.50M);  
walletTextBlock.DataContext = joe;
```

Kontekstem danych dla tej kontrolki TextBox jest referencja do obiektu Guy. Kontrolka ta odczyta z tego obiektu wszystkie powiązane właściwości.

Teraz powiązanie zostało już określone! Podałeś kontekst danych będący instancją klasy Guy i określiłeś ścieżkę powiązania odwołującą się do właściwości Cash. Skoro zastosowane powiązanie ma postać Cash, zatem kontrolka TextBox przejrzy obiekt danych, **poszukując w nim właściwości o nazwie Cash**.

Można także pominąć określanie ścieżki powiązania i zapisać we właściwości kontrolki jedynie wyrażenie {Binding}. W takim przypadku wyświetlony zostanie wynik zwrócony przez wywołanie metody ToString() klasy Guy.

Powiązanie dwukierunkowe pozwala odczytywać i zapisywać wartość właściwości źródłowej

Wiązanie danych pozwala odczytywać wartość z obiektu danych. Jednak, korzystając z **powiązania dwukierunkowego**, można także modyfikować wartość właściwości źródłowej:

```
<TextBox x:Name="ageTextBox" Text="{Binding Age, Mode=TwoWay}"/>
```

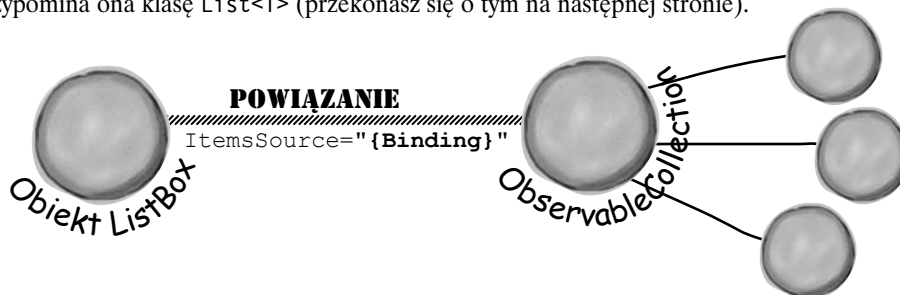
Ścieżka powiązania zastosowana w tej kontrolce TextBox odwołuje się do właściwości Age, natomiast samo powiązanie działa w trybie dwukierunkowym. W momencie wyświetlania strony kontrolka odczyta i wyświetli wartość właściwości Age dowolnego obiektu, z którym została związana. Jeśli jednak zmienimy wartość tej kontrolki, wywoła ona akcesor set właściwości Age i zaktualizuje jej wartość.



Mechanizm wiązania danych został opracowany w taki sposób, by przysparzać nam jak najmniej problemów. Jeśli ścieżka powiązania będzie się odwoływać do właściwości, która nie jest dostępna w kontekście danych, to kontrolka nie będzie wyświetlać ani aktualizować danych, a jednocześnie nie wystąpią żadne problemy w działaniu programu.

ObservableCollection pozwala tworzyć powiązania z kolekcjami

Niektóre kontrolki, takie jak TextBox lub TextBlock, wyświetlają łańcuchy znaków. Inne, takie jak ScrollViewer, wyświetlają zawartość obiektu. Niemniej jednak spotkałeś się już z kontrolkami wyświetlającymi kolekcje, takimi jak ListBox oraz ComboBox. To właśnie z tego względu .NET Framework udostępnia klasę ObservableCollection<T>, reprezentującą kolekcję opracowaną specjalnie pod kątem mechanizmów wiązania danych. Pod względem sposobu działania przypomina ona klasę List<T> (przekonasz się o tym na następnej stronie).



Utworzenie powiązania między właściwością ItemsSource kontrolki ListBox oraz kolekcją ObservableCollection sprawi, że kontrolka wyświetli wszystkie elementy kolekcji.

Utwórz powiązanie w kodzie (bez użycia kodu XAML!)

Jeśli spróbujemy sprawdzić kontrolkę, nie znajdziemy w niej właściwości o nazwie Binding. C# nie pozwala pobierać referencji do właściwości obiektu, a jedynie do całego obiektu. Okazuje się, że definiując powiązanie danych w kodzie XAML, określamy je, tworząc **obiekt klasy Binding**, zawierający nazwę właściwości źródłowej podaną w formie łańcucha znaków. Oto kod, który tworzy obiekt Guy i wiąże go z kontrolką TextBox o nazwie walletTextBlock, w taki sposób, że właściwość Text kontrolki będzie powiązana z właściwością Cash obiektu:

```
Guy joe = new Guy("Joe", 47, 325.50M);
Binding cashBinding = new Binding();
cashBinding.Path = new PropertyPath("Cash");
cashBinding.Source = joe;
walletTextBlock.SetBinding(TextBlock.TextProperty, cashBinding);
```

Istnieje klasa o nazwie DependencyProperty, a klasa TextBlock definiuje całą masę statycznych właściwości będących obiektami tej klasy. Jedną z nich nosi nazwę TextProperty.

Kontrolki XAML mogą zawierać tekst... i nie tylko

Porozmawiajmy trochę o **kodzie znacznikowym** XAML (w końcu to jego reprezentuje litera M w skrócie XAML, a odnosi się on do wszystkich znaczników definiujących stronę) oraz o **kodzie ukrytym** (ang. *code-behind*; czyli kodzie C# stanowiącym uzupełnienie kodu XAML i umieszczanym w plikach .cs).

Kiedy używasz kontrolki `Grid` lub `StackPanel`, inne, umieszczane wewnątrz nich kontrolki są zapisywane pomiędzy ich znacznikiem otwierającym i zamykającym. W taki sam sposób można także postępować, używając innych kontrolki. Na przykład wartość właściwości `Text` kontrolki `TextBox` lub `TextBlock` można określić, zapisując łańcuch znaków pomiędzy ich znacznikiem otwierającym i zamykającym:

```
<TextBlock>To jest prezentowany tekst</TextBlock>
```

Ten zapis jest odpowiednikiem określenia właściwości `Text`.

W takim przypadku nowe wiersze można tworzyć, używając znaczników `<LineBreak/>`, a nie symboli ``. W obu przypadkach sprowadza się to do zastosowania znaku Unicode o wartości `U+0013`, który jest interpretowany jako znak nowego wiersza. Można go także zapisać w formie szesnastkowej — ``; z kolei symbol `£` pozwala umieścić w łańcuchu znak £ (pamiętasz program *Tablica znaków*?).

```
<TextBlock>Pierwszy wiersz<LineBreak/>Drugi wiersz</TextBlock>
```

Spróbuj dodać tę kontrolkę `TextBlock` do kodu strony, a następnie wybierz opcję *Edit Text*, by zmienić jej zawartość i naciśnij kombinację klawiszy *Shift+Enter*, by dodać nowy wiersz. W efekcie IDE doda do strony następujący fragment kodu XAML:

```
<TextBlock>  
  <Run Text="Pierwszy wiersz"/>  
  <LineBreak/>  
  <Run Text="Drugi wiersz"/>  
</TextBlock>
```

Każde z tych trzech rozwiązań będzie wyglądało na ekranie tak samo, lecz jednocześnie spowoduje wygenerowanie innego grafu obiektów. Każdy znacznik `<Run>` jest przekształcany w odrębny obiekt łańcucha znaków, a każdy z tych obiektów może mieć swoją własną nazwę:

```
<Run Text="Pierwszy wiersz" x:Name="firstLine" />
```

Tej nazwy można następnie użyć w kodzie C# obsługującym stronę XAML do zmiany wyświetlanego łańcucha:

```
firstLine.Text = "To jest nowa zawartość pierwszego wiersza";
```

Kontrolki z zawartością, takie jak `ScrollView`, dysponują właściwością `Content` (a nie `Text`), która działa w inny sposób — może zawierać dowolne kontrolki. A takich kontrolki z treścią jest całkiem sporo. Jedną z bardziej przydatnych jest kontrolka `Border`, której można używać, by dodawać tło lub obramowanie do innych kontrolki, które normalnie ich nie mają, takich jak `TextBlock`:

```
<Border Background="Blue"  
  BorderBrush="Green" BorderThickness="3">  
</Border>
```

Kontrolka `ScrollView` dziedziczy po `ContentControl`, czyli tej samej kontrolce, której użyłeś, by stworzyć obcych w grze *Ratuj ludzi*. Utworzona przez Ciebie kontrolka `ContentControl` zawierała kontrolkę `Grid`, a w niej trzy kontrolki `Ellipse`.



ZACZYNAM
WYOBRAŻAĆ SOBIE, JAK
ZAPROJEKTUJĘ PROGRAM DO
ZARZĄDZANIA WYMÓWKAMI. ALE
W JAKI SPOSÓB KONTROLKI BĘDĄ
ODCZYTYWAĆ I MODYFIKOWAĆ
DANE PRZECHOWYWANE
W OBIEKTACH?

Nie istnieją grupy pytania

P: Moja strona zawiera siatkę, w której jest umieszczona inna siatka zawierająca StackPanel. Czy istnieje ograniczenie liczby kontroltek, które można umieszczać w innych kontrolkach?

U: Nie. Można umieszczać kontrolki wewnątrz innych kontroltek, a z kolei te w jeszcze innych. W dalszej części rozdziału nauczysz się tworzyć własne kontrolki, zaczynając od utworzenia kontenera i dodając do niego dalsze kontrolki. Na przykład siatkę można umieścić w dowolnej innej kontrolce z treścią — zrobiłeś już to w grze *Ratuj ludzi*, tworząc wroga z kontrolki Grid i trzech kontroltek Ellipse. To jedna z mocnych stron stosowania języka XAML do projektowania aplikacji — pozwala ona na tworzenie złożonych stron przy użyciu prostych kontroltek.

P: Gdybym mógł określić układ strony, używając kontrolki Grid lub StackPanel, to której z nich powinienem użyć?

U: To w dużym stopniu zależy od sytuacji. Na to pytanie nie ma jednej dobrej odpowiedzi: czasami lepszym rozwiązaniem będzie użycie kontrolki StackPanel, czasami Grid, a czasami połączenia ich obu. A nie są to bynajmniej nasze jedyne możliwości. Można skorzystać z kontrolki Canvas (której użyłeś w grze *Ratuj ludzi*), pozwalającej na wyświetlanie innych kontroltek w miejscach określonych przy użyciu właściwości Canvas.Left oraz Canvas.Right. Wszystkie te trzy kontrolki są klasami pochodnymi klasy Panel, a jedną z odziedziczonych po niej możliwości jest dodawanie i prezentowanie dowolnej liczby innych kontroltek.

P: Czy to oznacza, że są kontrolki, w których można umieszczać tylko jedną inną kontrolkę?

U: Owszem. Spróbuj umieścić na stronie kontrolkę ScrollView. Następnie spróbuj umieścić wewnątrz niej dwie inne kontrolki. Oto co zobaczysz:

```
<ScrollView>
  <TextBox/>
  <Button/>
</ScrollView>
```

The property 'Content' is set more than once.

Dzieje się tak dlatego, że w tym przypadku XAML określa wartość właściwości Content kontrolki ScrollView, a ta jest typu object. Spróbuj jednak zastąpić kontrolkę ScrollView kontrolką Grid:

```
<Grid>
  <TextBox/>
  <Button/>
</Grid>
```

Okaże się, że wszystko jest w porządku. W tym przypadku zagnieźdzone kontrolki są bowiem dodawane do kolekcji Children. (W grze *Ratuj ludzi* używałeś tej kolekcji od dodawania wrogów).

P: Dlaczego niektóre kontrolki, takie jak TextBox, mają właściwość Text, a niektóre właściwość Content?

U: Ponieważ mogą wyświetlać wyłącznie tekst; właśnie dlatego dysponują właściwością Text typu String, a nie właściwością Content typu object. Jest to tak zwana **domyślna właściwość** kontrolki. W przypadku kontroltek takich jak Grid oraz StackPanel tą domyślną właściwością jest kolekcja Children.

P: Czy powinienem wpisywać kod XAML ręcznie, czy raczej używać okna Designer IDE i przeciągać kontrolki z przybornika?

U: Warto spróbować obu tych sposobów i wybrać ten, który Ci bardziej odpowiada. Wielu programistów korzysta niemal wyłącznie z okna Designer, choć jest też sporo takich, którzy go niemal wcale nie używają, gdyż przekonali się, że wpisywanie kodu XAML jest szybsze. Dzięki technologii IntelliSense wpisywanie kodu XAML jest faktycznie wyjątkowo łatwe.

P: Przypomnijcie mi jeszcze raz, dlaczego miałem poznać technologię WinForms? Czemu nie zacząłem od razu uczyć się języka XAML i tworzenia aplikacji dla Sklepu Windows?

U: Ponieważ istnieje wiele pojęć, których znajomość znacznie ułatwia zrozumienie języka XAML. Na przykład przyjrzyjmy się kolekcji Children. Gdybyś nie rozumiał, czym są kolekcje, to czy odpowiedź na trzecie pytanie zamieszczone na tej stronie miałyby dla Ciebie jakikolwiek sens? Może. Jednak znacznie łatwiej ją zrozumieć, wiedząc czym są kolekcje. Z drugiej strony przeciąganie kontroltek z przybornika i umieszczanie ich na formularzu jest naprawdę łatwe. Korzystanie z technologii WinForms wymaga naprawdę znacznie mniej wiedzy niż projektowanie stron z użyciem języka XAML (co jest zrozumiałe, gdyż XAML jest znacznie nowszą i znacznie bardziej elastyczną technologią). Dzięki temu, że poświęciliśmy kilka rozdziałów, prezentując WinForms, nabrałeś nieco praktyki w projektowaniu aplikacji z graficznym interfejsem użytkownika i tworzeniu interesujących projektów. A to z kolei pozwoliło Ci przyswoić sobie wiele ważnych pojęć. Oprócz tego, poznanie dwóch sposobów utworzenia tych samych projektów jest bardzo wartościowe. To właśnie z tego powodu wróciliśmy do kilku projektów z poprzednich rozdziałów: poznając dwa sposoby napisania tej samej aplikacji, będziesz mógł lepiej zrozumieć zarówno technologię WinForms, jak i aplikację dla Sklepu Windows.

Technologia WinForms jest doskonałym narzędziem do nauki i poznawania języka C#, jednak język XAML jest znacznie lepszym narzędziem do tworzenia elastycznych i efektywnych aplikacji.

Użyj wiązania danych, by usprawnić aplikację Niechlujnego Janka

Pamiętasz program generujący menu dla Niechlujnego Janka, który napisałeś w rozdziale 4? Cóż, Janek zainstalował sobie teraz system Windows 8 i chciałby, żeby jego aplikacja do generowania menu została przystosowana dla Sklepu Windows. Zróbmy zatem to, o co prosi.

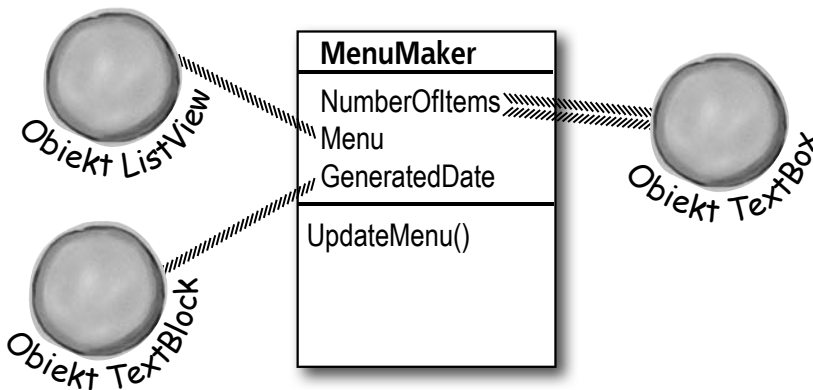
Oto strona, którą mamy zamiar stworzyć.

Będzie ona korzystać z jednokierunkowego wiązania danych, by wyświetlać informacje w kontrolce ListView oraz w jednym z obiektów Run umieszczonych wewnątrz kontrolki TextBlock oraz dwukierunkowego wiązania danych w kontrolce TextBox.

```
<StackPanel Grid.Row="1" Margin="120,0">  
  <StackPanel Orientation="Horizontal">  
    <TextBlock/>  
    <TextBox Text="{Binding NumberOfItems, Mode=TwoWay}">  
  </StackPanel>  
  <Button/>  
</StackPanel>  
<ListView ItemsSource="{Binding Menu}">  
<TextBlock>  
  <Run/>  
  <Run Text="{Binding GeneratedDate}">  
</TextBlock>  
</StackPanel>
```

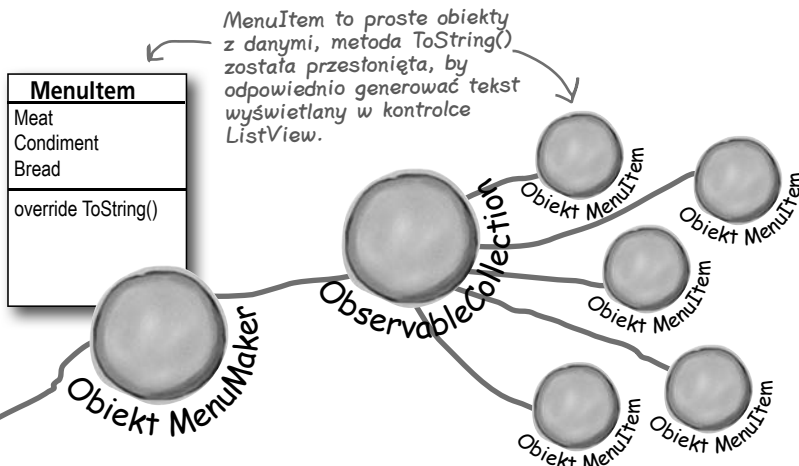
Będzie nam potrzebny obiekt, którego właściwości użyjemy do zdefiniowania powiązania.

Obiekt Page będzie zawierał instancję klasy MenuMaker, która z kolei dysponuje trzema właściwościami publicznymi: właściwością NumberOfItems typu int, kolekcją Menu typu ObservableCollection zawierającą menu oraz właściwością GeneratedDate typu DateTime.



Obiekt Page tworzy instancję klasy MenuMaker i używa jej jako kontekstu danych.

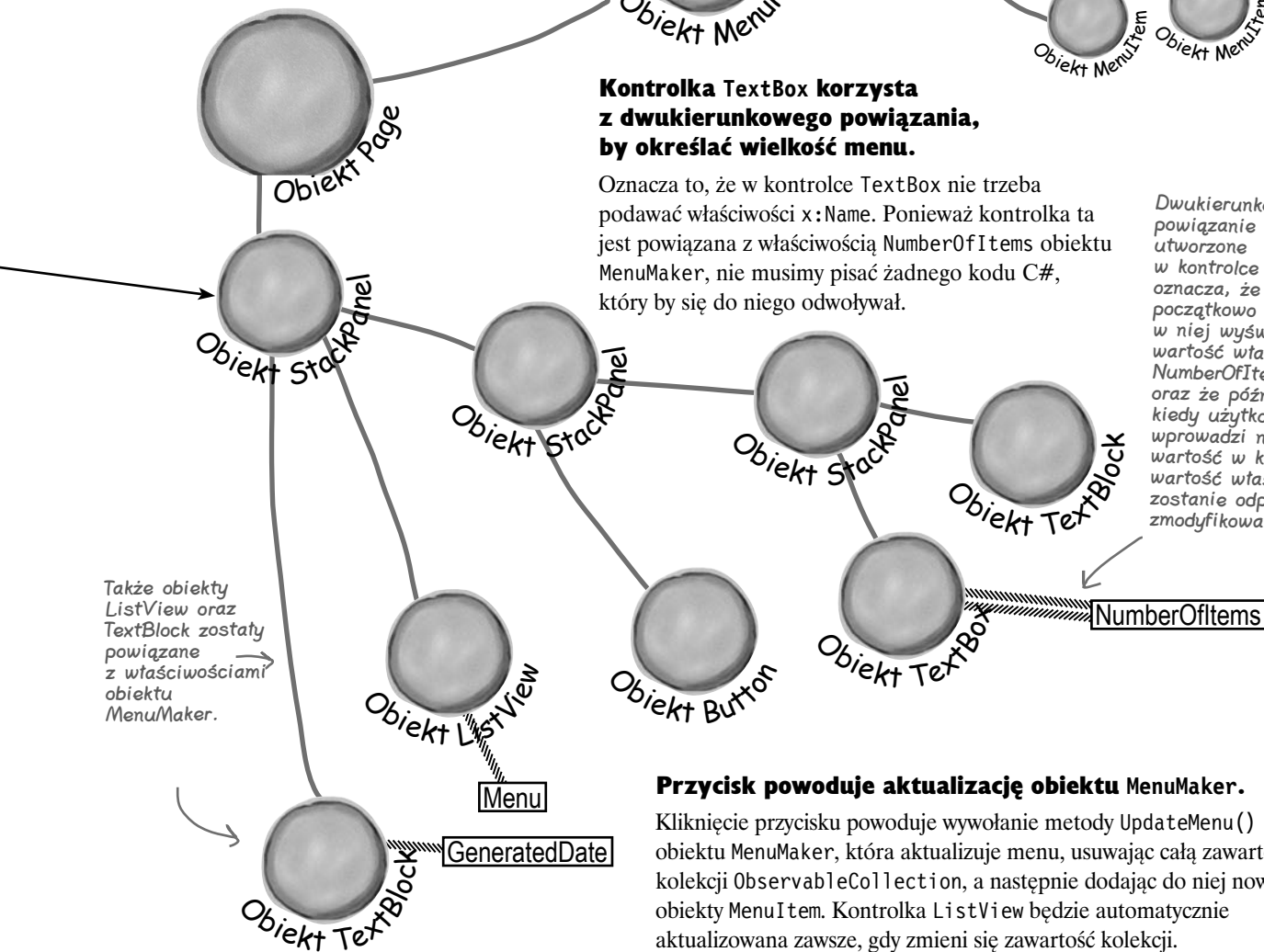
W konstruktorze klasy Page, we właściwości DataContext kontrolki StackPanel, zapiszemy referencję do obiektu MenuMaker. Samo powiązanie zostanie w całości zdefiniowane w kodzie XAML.



Kontrolka TextBox korzysta z dwukierunkowego powiązania, by określać wielkość menu.

Oznacza to, że w kontrolce TextBox nie trzeba podawać właściwości x:Name. Ponieważ kontrolka ta jest powiązana z właściwością NumberOfItems obiektu MenuMaker, nie musimy pisać żadnego kodu C#, który by się do niego odwoływał.

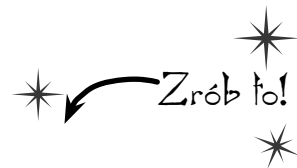
Dwukierunkowe powiązanie utworzone w kontrolce TextBox oznacza, że początkowo zostanie w niej wyświetlona wartość właściwości NumberOfItems oraz że później, kiedy użytkownik wprowadzi nową wartość w kontrolce, wartość właściwości zostanie odpowiednio zmodyfikowana.



Przycisk powoduje aktualizację obiektu MenuMaker.

Kliknięcie przycisku powoduje wywołanie metody UpdateMenu() obiektu MenuMaker, która aktualizuje menu, usuwając całą zawartość kolekcji ObservableCollection, a następnie dodając do niej nowe obiekty MenuItem. Kontrolka ListView będzie automatycznie aktualizowana zawsze, gdy zmieni się zawartość kolekcji.

Oto wyzwanie dla programistów. Bazując na podanych do tej pory informacjach, powiedz, jak wiele z nowej, poprawionej aplikacji dla Niechlujnego Janka jesteś w stanie napisać bez zaglądania na następną stronę?



1 Utwórz nowy projekt i zastąp stronę MainPage.xaml stroną utworzoną według szablonu Basic Page.

Utwórz nowy projekt aplikacji typu *Windows Store*. Następnie usuń z niego plik *MainPage.xaml*, **utwórz nową stronę na podstawie szablonu *Basic Page* i nadaj jej nazwę *MainPage.xaml***. Po wprowadzeniu tych zmian będziesz musiał ponownie zbudować projekt. To są dokładnie te same czynności, które wykonałeś, tworząc aplikację *Ratuj ludzi* (zajrzyj do rozdziału 1., jeśli musisz sobie odświeżyć pamięć).

2 Dodaj nową, poprawioną klasę MenuMaker.

Przebyłeś już długą drogę od momentu, gdy zakończyłeś lekturę rozdziału 4. Napiszmy zatem prawidłowo hermetyzowaną klasę, która dzięki zastosowaniu właściwości pozwoli nam zapisywać i pobierać określone informacje. W jej konstruktorze utworzymy kolekcję *ObservableCollection* obiektów typu *MenuItem*, która będzie aktualizowana za każdym razem, gdy zostanie wywołana metoda *UpdateMenu()*. Metoda ta będzie także aktualizować właściwość *GeneratedDate* typu *DateTime*, określającą czas generacji bieżącego menu. Dodaj poniższą klasę *MenuMaker* do projektu:

Kliknij prawym przyciskiem myszy nazwę projektu wyświetloną w oknie *Solution Explorer* i dodaj nową klasę, tak samo jak robiłeś to w innych projektach.

Wartości tych właściwości będziesz wyświetlał na stronie, używając wiązania danych. W przypadku właściwości *NumberOfItems* użyjesz powiązania dwukierunkowego.

```
using System.Collections.ObjectModel;

class MenuMaker {
    private Random random = new Random();
    private List<String> meats = new List<String>()
    { "Pieczona wołowina", "Salami", "Indyk", "Szynka", "Karkówka" };
    private List<String> condiments = new List<String>() {"żółta musztarda",
    "brązowa musztarda", "musztarda miodowa", "majonez", "przyprawa", "sos francuski"};
    private List<String> breads = new List<String>() {"chleb ryżowy",
    "chleb biały", "chleb zbożowy", "pumperniel", "chleb włoski", "bułka"};

    public ObservableCollection<MenuItem> Menu { get; private set; }
    public DateTime GeneratedDate { get; private set; }
    public int NumberOfItems { get; set; }
    public MenuMaker() {
        Menu = new ObservableCollection<MenuItem>();
        NumberOfItems = 10;
        UpdateMenu();
    }
    private MenuItem CreateMenuItem() {
        string randomMeat = meats[random.Next(meats.Count)];
        string randomCondiment = condiments[random.Next(condiments.Count)];
        string randomBread = breads[random.Next(breads.Count)];
        return new MenuItem(randomMeat, randomCondiment, randomBread);
    }
    public void UpdateMenu() {
        Menu.Clear();
        for (int i = 0; i < NumberOfItems; i++) {
            Menu.Add(CreateMenuItem());
        }
        GeneratedDate = DateTime.Now;
    }
}
```

Będziesz potrzebował tej instrukcji *using*, gdyż klasa *ObservableCollection<T>* została zdefiniowana w tej przestrzeni nazw.

Nowa metoda *CreateMenuItem()* zwraca obiekt *MenuItem*, a nie łańcuch znaków. W ten sposób łatwiej nam będzie zmieniać wygląd elementów listy, gdyby pojawiła się taka konieczność.

Przyjrzyj się dokładnie, jak działa ta metoda. Okazuje się, że nie tworzy ona nowej kolekcji obiektów *MenuItem*, a jedynie modyfikuje istniejącą kolekcję, usuwając najpierw jej dotychczasową zawartość, a następnie dodając do niej nowe elementy.

Co się stanie, kiedy właściwość *NumberOfItems* zostanie przypisana wartość mniejsza od zera?

Używaj typu *DateTime*, by operować na datach
Spotkałeś się już z typem *DateTime* pozwalającym przechowywać daty. Można go także używać do tworzenia i modyfikowania dat i czasu. Definiuje on statyczną właściwość o nazwie *Now*, która zwraca bieżącą datę i godzinę. Udostępnia także metody, takie jak *AddSeconds()*, pozwalające na dodawanie i konwersję sekund, milisekund, dni itd. oraz właściwości takie jak *Hour* oraz *DayOfWeek* pozwalające na odczyt poszczególnych elementów dat i czasu. W sam raz na czasie!

3 Dodaj klasę MenuItem.

Przekonałeś się już, że przechowywanie danych w obiektach i klasach, a nie w łańcuchach znaków, pozwala tworzyć bardziej elastyczne programy. Oto prosta klasa służąca do przechowywania jednej kanapki dostępnej w menu. Dodaj ją do swojego projektu:

```
class MenuItem {
    public string Meat { get; private set; }
    public string Condiment { get; private set; }
    public string Bread { get; private set; }

    public MenuItem(string meat, string condiment, string bread) {
        Meat = meat;
        Condiment = condiment;
        Bread = bread;
    }

    public override string ToString() {
        return Meat + ", " + Condiment + ", " + Bread;
    }
}
```

Trzy łańcuchy znaków określające kanapkę są przekazywane do konstruktora i zapisywane w automatycznych właściwościach przeznaczonych tylko do odczytu.

Przeostań metodę ToString(), aby obiekty MenuItem wiedziały, w jaki sposób chcemy je wyświetlać.

4 Utwórz stronę XAML.

Oto zrzut ekranu aplikacji. Czy potrafiłbyś ją utworzyć, korzystając z kontrolki StackPanel? Kontrolka TextBox ma szerokość 100 pikseli. W dolnej kontrolce TextBox został zastosowany styl BodyTextStyle; zawiera ona dwa znaczniki <Run> (w drugim jest wyświetlana wyłącznie data i godzina).

Nie zapomnij zmienić nagłówka strony. W tym celu zmień wartość zasobu `AppName`, podaną w sekcji `<Page.Resources>` na górze strony.

To jest kontrolka `ListView`. Jest ona bardzo podobna do kontrolki `ListBox` — w rzeczywistości obie dziedziczą po tej samej klasie bazowej, dlatego dysponują tymi samymi możliwościami zaznaczania elementów list. Jednak aplikacje dla Sklepu Windows korzystają zazwyczaj z kontrolki `ListView`, a nie `ListBox`, gdyż sposób przewijania ich zawartości (z pewną inercją) oraz inne cechy ich interfejsu użytkownika bardziej przypominają „prawdziwe” aplikacje systemu Windows 8. Robiąc ten zrzut ekranu, zaznaczyliśmy pierwszy element listy i, jak widać, został on zaznaczony w charakterystyczny sposób.

Tym razem nie dodawaj żadnych danych przykładowych. Informacje do prezentacji zostaną dostarczone przez powiązanie danych.

Czy potrafisz samodzielnie stworzyć taką stronę, bazując wyłącznie na tym zrzucie ekranu?

5 Do kodu XAML dodaj nazwy obiektów i pozostałe parametry wiązania danych

Oto kod XAML, który należy dodać do strony *MainPage.xaml*. Upewnij się, że umieścisz go w **zewnętrznej siatce bezpośrednio przed komentarzem XAML** `<!-- Back button and page title -->`, dokładnie tak samo jak na stronie głównej aplikacji Ratuj ludzi. Przyciskowi nadaliśmy nazwę `newMenu`. Ponieważ w kontrolkach `ListView`, `TextBlock` oraz `TextBox` wykorzystaliśmy mechanizm wiązania danych, zatem nie musieliśmy określać ich nazw. *(Dodatkowe ułatwienie. Tak naprawdę nie musieliśmy nawet określać nazwy przycisku; zrobiliśmy to tylko po to, by IDE mogło automatycznie dodać do niego procedurę obsługi zdarzeń o nazwie `newMenu_Clicked`, w wyniku dwukrotnego kliknięcia tej kontrolki. Przekonaj się sam!).*

```
<StackPanel Grid.Row="1" Margin="120,0" x:Name="pageLayoutStackPanel">
  <StackPanel Orientation="Horizontal" Margin="0,0,0,20">
    <StackPanel Margin="0,0,20,0">
      <TextBlock Style="{StaticResource BodyTextStyle}"
        Text="Wielkość menu" Margin="0,0,0,10" />
      <TextBox Width="100" HorizontalAlignment="Left"
        Text="{Binding NumberOfItems, Mode=TwoWay}" />
    </StackPanel>
    <Button x:Name="newMenu" VerticalAlignment="Bottom" Click="newMenu_Click"
      Content="Wygeneruj nowe menu" Margin="0,0,20,0"/>
  </StackPanel>
  <ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0" />
  <TextBlock Style="{StaticResource CaptionTextStyle}">
    <Run Text="Data generacji menu: " />
    <Run Text="{Binding GeneratedDate}"/>
  </TextBlock>
</StackPanel>
```

Oto kontrolka `ListView`. Spróbuj zastąpić ją kontrolką `ListBox` i sprawdź, co się zmieni na stronie.

Do odczytu i ustawiania liczby kanapek w menu przy użyciu kontrolki `TextBox` konieczne było zastosowanie powiązania dwukierunkowego.

To właśnie w takich miejscach przydają się znaczniki `<Run>`. Dzięki nim wystarczyło użyć jednej kontrolki `TextBlock` i zdefiniować powiązanie jedynie z fragmentem tekstu.

Do pliku *MainPage.xaml.cs* dodaj kod obsługujący stronę.

W konstruktorze strony tworzona jest kolekcja zawierająca całe menu, obiekt `MenuMaker` oraz określany kontekst danych wykorzystujących mechanizm wiązania danych. Wymaga to dodatkowo zdefiniowania pola typu `MenuMaker` o nazwie `menuMaker`.

```
MenuMaker menuMaker = new MenuMaker();

public MainPage() {
  this.InitializeComponent();

  pageLayoutStackPanel.DataContext = menuMaker;
}
```

W pliku *MainPage.xaml.cs* pojawiło się pole typu `MenuMaker`, które posłuży jako kontekst danych dla kontrolki `StackPanel` zawierającej wszystkie powiązane kontrolki prezentujące dane na stronie.

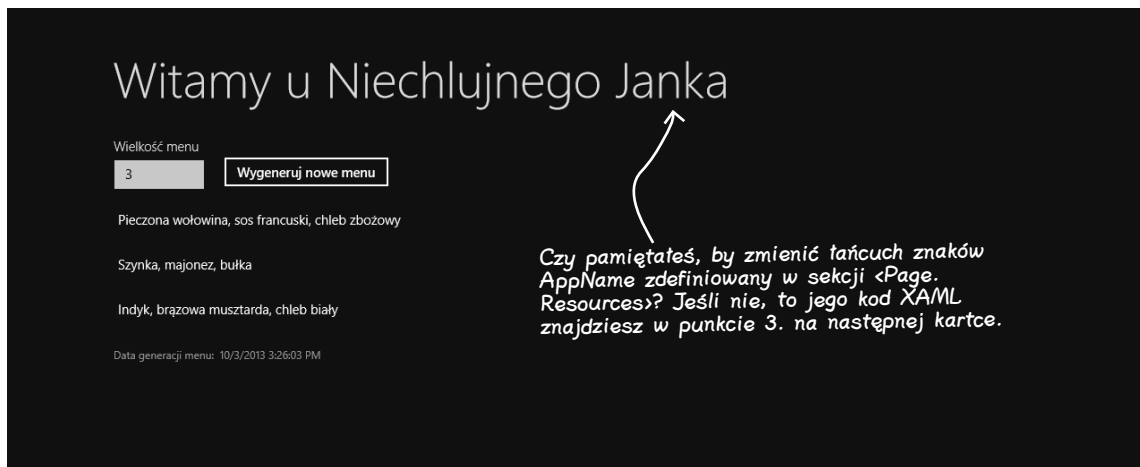
Musisz określić kontekst danych dla kontrolki `StackPanel`. Dzięki temu zostanie on przekazany do wszystkich kontrolki umieszczonych wewnątrz niej.

W końcu, dwukrotnie kliknij przycisk, aby wygenerować szkielec procedury obsługi zdarzeń `Click`. Oto cały kod obsługi tych zdarzeń — ogranicza się on do zaktualizowania menu:

```
private void newMenu_Click(object sender, RoutedEventArgs e) {
  menuMaker.UpdateMenu();
}
```

Jest pewien sposób pozwalający na łatwą zmianę nazwy procedury obsługi zdarzeń, tak by jednocześnie zaktualizować zarówno kod C# jak i XAML. Zajrzyj do punktu 8. dodatku „Pozostałości”, by dowiedzieć się więcej o narzędziach refaktoryzacji dostępnych w Visual Studio IDE.

A teraz uruchom program! Spróbuj zmienić liczbę w polu TextBox. Wpisz w nim wartość 3 i kliknij przycisk — program wygeneruje nowe menu zawierające trzy kanapki.



Teraz możesz już wypróbować powiązania danych, by przekonać się, jak bardzo są one elastyczne. Spróbuj wpisać w polu tekstowym „xyz” lub zostawić je puste. Nic się nie stanie! Wpisując cokolwiek w kontrolce TextBox, umieszczamy w niej łańcuch znaków. Kontrolka całkiem inteligentnie określa, co ma z tym łańcuchem zrobić. Wie, że ścieżka powiązania ma postać `NumberOfItems`, zatem przegląda kontekst danych, sprawdzając, czy jest w nim dostępna jakakolwiek właściwość o tej nazwie, a następnie stara się w możliwie najlepszy sposób skonwertować łańcuch znaków na typ tej właściwości.

Zwróć uwagę na wygenerowaną datę. Nie zmienia się ona, choć menu jest aktualizowane. Cóż, chyba wciąż pozostaje nam jeszcze coś do zrobienia.

MOJA WŁAŚCIWOŚĆ TEXT ZOSTAŁA POWIĄZANA Z WŁAŚCIWOŚCIĄ NUMBEROFITEMS. NO I PATRZ, W MOIM KONTEKŚCIE DANYCH JEST WŁAŚCIWOŚĆ NUMBEROFITEMS! CZY MOGĘ ZAPISAĆ W NIEJ ŁAŃCUCH ZNAKÓW „3”? WYGLĄDA NA TO, ŻE MOGĘ!

Obiekt TextBox



Obiekt TextBox

HMM... WŁAŚCIWOŚĆ NUMBEROFITEMS MOJEGO KONTEKSTU DANYCH JEST TYPU INT, ALE NIE WIEM, JAK SKONWERTOWAĆ ŁAŃCUCH „XYZ” NA LICZBĘ. W TAKIM RAZIE CHYBA LEPIEJ NIC NIE BĘDĘ ROBIĆ.

Korzystaj z zasobów statycznych, by deklarować obiekty w kodzie XAML

Tworząc stronę w języku XAML, w rzeczywistości tworzysz graf obiektów, posługując się przy tym takimi obiektami jak StackPanel, Grid, TextBlock czy też Button. Jak się już przekonałeś, nie ma w tym niczego magicznego ani tajemniczego — dodanie do kodu XAML znacznika <TextBox> sprawi, że w obiekcie strony pojawi się pole typu TextBox, w którym zostanie zapisana referencja do instancji klasy TextBox. Dodając do tego znacznika właściwość x:Name, sprawimy, że w kodzie ukrytym będziemy mogli używać tej nazwy, by odwoływać się do kontrolki.

Dokładnie w ten sam sposób można tworzyć instancje *niemal wszystkich* klas i zapisywać je w polach obiektu strony. Pozwalają na to tak zwane **zasoby statyczne** dodawane do kodu XAML. Co więcej, mechanizm wiązania danych bardzo dobrze współpracuje z takimi zasobami statycznymi, zwłaszcza jeśli wykorzystamy także okno Designer IDE. Wróć zatem do programu dla Niechlujnego Janka i przekształć obiekt MenuMaker w zasób statyczny.

1 Z PLIKU KODU UKRYTEGO USUŃ POLE MENU MAKER.

Masz zamiar utworzyć obiekt klasy MenuMaker oraz kontekst danych w kodzie XAML, a zatem usuń z kodu C# dwa wyróżnione wiersze:

```
MenuMaker menuMaker = new MenuMaker();  
  
public MainPage() {  
    this.InitializeComponent();  
  
pageLayoutStackPanel.DataContext = menuMaker;  
}
```

2 SPRAWDŹ, JAK WYGLĄDA PRZESTRZEŃ NAZW TWOJEJ APLIKACJI.

Spójrz na sam początek kodu XAML strony, a przekonasz się, że w jej znaczniku otwierającym jest umieszczonych kilka właściwości xmlns. Każda z nich definiuje przestrzeń nazw. Poszukaj takiej, która zaczyna się od xmlns:local i odpowiada przestrzeni nazw projektu. Powinna wyglądać mniej więcej tak:

To jest właściwość przestrzeni nazw XML. Zawiera ona tańcuch „xmlns:” zakończony identyfikatorem, którym w tym przypadku jest „local”.

Jeśli wartość przestrzeni nazw rozpoczyna się od „using:”, oznacza to, że odwołuje się ona do jednej z przestrzeni nazw dostępnych w projekcie. Może się także rozpoczynać od „http://”, co będzie oznaczać standardową przestrzeń nazw XAML.

→ **xmlns:local="using:NiechlujnyJanekRozdzial10"**

→

Będiesz używał tego identyfikatora podczas tworzenia obiektów w przestrzeni nazw projektu.

Ponieważ nasza aplikacja nosi nazwę NiechlujnyJanekRozdział10, zatem IDE utworzył tę przestrzeń nazw na nasze potrzeby. Odszukaj przestrzeń nazw używaną w Twojej aplikacji, gdyż to właśnie w niej będzie istniał obiekt MenuMaker.

Nie istnieją
grupy pytania

P: Ale zaraz! Ta aplikacja nie ma przycisku Zamknij! W jaki sposób mogę z niej wyjść?

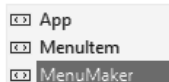
O: Aplikacje przeznaczone dla Sklepu Windows domyślnie nie mają żadnego przycisku do zamykania, gdyż zazwyczaj z większości spośród nich nigdy się nie wychodzi. Aplikacje tego typu działają zgodnie z **cyklem życia aplikacji**, który definiuje trzy stany: nie działa, działa, wstrzymana. Aplikacje mogą zostać wstrzymane, jeśli użytkownik z nich wyjdzie lub gdy system Windows wykryje, że w urządzeniu zaczyna brakować prądu. Oprócz tego, jeśli system będzie musiał odzyskać pamięć, to może zakończyć działanie aplikacji. W dalszej części książki nauczysz się tworzyć aplikacje działające zgodnie z tym cyklem.

3 DODAJ DO PLIKU XAML ZASÓB STATYCZNY I OKREŚL KONTEKST DANYCH.

Odszukaj w kodzie strony sekcję <Page.Resources> i wpisz <local: — spowoduje to wyświetlenie okienka *IntelliSense*:

```
<Page.Resources>
```

```
<local:
```



```
<!-- TODO: Delete this line if the key AppName is declared in App.xaml -->
```

```
<x:String x:Key="AppName">Witamy u Niechlujnego Janka</x:String>
```

```
</Page.Resources>
```

Zasoby statyczne można tworzyć wyłącznie w przypadku, gdy ich klasa definiuje konstruktor bezparametrowy. To ma sens! Gdyby konstruktor miał parametry, to skąd strona XAML miałaby wiedzieć, jakie argumenty przekazać w jego wywołaniu?

W okienku zostaną wyświetlone wszystkie klasy dostępne w przestrzeni nazw, których możesz używać. Wybierz MenuMaker i nadaj jej nazwę menuMaker:

```
<local:MenuMaker x:Name="menuMaker"/>
```

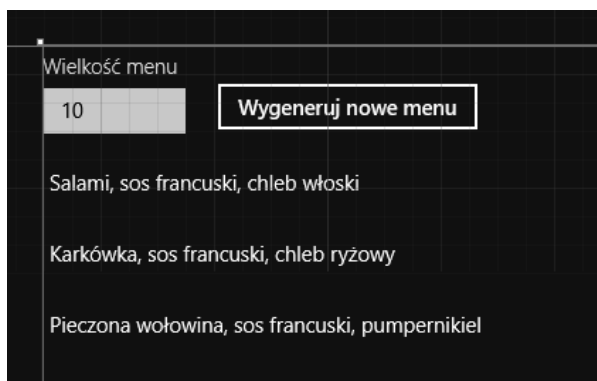
Teraz strona zawiera statyczny zasób typu MenuMaker o nazwie menuMaker.

4 OKREŚL KONTEKST DANYCH KONTROLKI STACKPANEL I WSZYSTKICH KONTROLEK WEWNĄTRZ NIEJ.

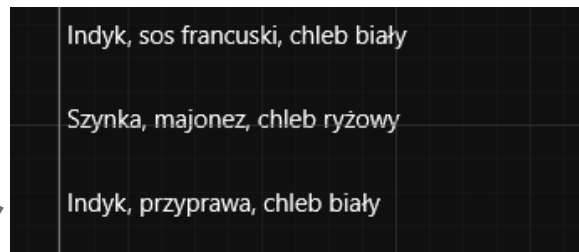
Odszukaj zewnętrzną kontrolkę StackPanel i określ jej właściwość DataContext w następujący sposób:

```
<StackPanel Grid.Row="1" Margin="120,0"
    DataContext="{StaticResource ResourceKey=menuMaker}">
```

Twój program będzie działał dokładnie tak samo jak wcześniej. Ale czy zauważyłeś, co się stało w IDE, kiedy do kodu XAML dodałeś kontekst danych? Gdy tylko to zrobiłeś, IDE utworzyło instancję klasy MenuMaker i użyło jej właściwości do określenia zawartości powiązanych kontroltek. W oknie *Designer* natychmiast pojawiło się wygenerowane menu — jeszcze zanim zdążyłeś uruchomić aplikację. Niezłe!



← W oknie *Designer* natychmiast zostanie wyświetlone wygenerowane menu, jeszcze zanim zdążyłeś uruchomić aplikację.



→ Hm... coś tu nie jest w porządku. Została wyświetlona liczba kanapek w menu, samo menu, ale nie data. O co chodzi?

Wyświetlaj obiekty, używając szablonów danych

Podczas wyświetlania elementów listy prezentowana jest zawartość kontrolki ListViewItem (używanych w kontrolkach ListView), ListBoxItem lub ComboBoxItem, powiązanych z obiektami zapisanymi w kolekcji ObservableCollection. Każda kontrolka ListViewItem w aplikacji Niechlujnego Janka jest powiązana z obiektem MenuItem przechowywanym w kolekcji Menu. Domyślnie obiekty ListViewItem wywołują metodę ToString() obiektów MenuItem, możesz jednak zastosować **szablon danych**, który skorzysta z mechanizmu wiązania danych, by wyświetlać informacje z właściwości powiązanych obiektów.

Zmodyfikuj znacznik <ListView>, dodając do niego prosty szablon danych. Używa on prostego wyrażenia {Binding}, by wywoływać metodę ToString() elementów listy.

To naprawdę prosty szablon danych, co więcej, wygląda on bardzo podobnie do standardowego szablonu używanego do wyświetlania elementów ListViewItem.

```
<ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0">
  <ListView.ItemTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding}" />
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Nie zmieniaj nic wewnątrz znacznika ListView, tylko zastąp /> znakiem > i dodaj poniżej zamykający znacznik </ListView>. Następnie pomiędzy znacznikami ListView dodaj znacznik ListViewItem zawierający szablon danych.

Dodanie {Binding} bez żadnej ścieżki powoduje wywołanie metody ToString() powiązanego obiektu.

Zmień szablon danych, dodając do niego trochę kolorów.

Zmień znacznik <DataTemplate> wraz z zawartością, lecz resztę znacznika ListView pozostaw bez zmian.

```
<DataTemplate>
  <TextBlock>
    <Run Text="{Binding Meat}" Foreground="Blue" /><Run Text=", " />
    <Run Text="{Binding Bread}" FontWeight="Light" /><Run Text=", " />
    <Run Text="{Binding Condiment}" Foreground="Red" FontWeight="ExtraBold" />
  </TextBlock>
</DataTemplate>
```

Możesz powiązać poszczególne znaczniki Run. Możesz w nich zmieniać czcionkę, kolor oraz inne właściwości.

Szynka, chleb zbożowy, przyprawa

Salami, pumpnikiel, sos francuski

Zaszalej! Szablon danych może zawierać zupełnie dowolne kontrolki.

```
<DataTemplate>
  <StackPanel Orientation="Horizontal">
    <StackPanel>
      <TextBlock Text="{Binding Bread}" />
      <TextBlock Text="{Binding Bread}" />
      <TextBlock Text="{Binding Bread}" />
    </StackPanel>
    <Ellipse Fill="DarkSlateBlue" Height="Auto" Width="10" Margin="10,0" />
    <Button Content="{Binding Condiment}" FontFamily="Segoe Script" />
  </StackPanel>
</DataTemplate>
```

Właściwość Content obiektu DataTemplate może zawierać tylko jeden obiekt, a zatem, chcąc umieścić w szablonie danych więcej kontrolki, będziesz musiał użyć jakiegoś kontenera, takiego jak StackPanel.

pumpnikiel
pumpnikiel
pumpnikiel

sos francuski

Nie istnieją grupy pytania

P: A zatem do określania układu mogę używać kontrolek **StackPanel** lub **Grid**. Mogę tworzyć zasoby statyczne w kodzie XAML lub używać pól definiowanych w kodzie ukrytym. Mogę ustawiać właściwości kontrolek lub używać wiązania danych. Po co jest tyle sposobów na realizację tego samego celu?

U: Ponieważ C# oraz XAML są niezwykle elastycznymi narzędziami do tworzenia aplikacji. Ta elastyczność pozwala projektować bardzo szczegółowe strony działające na wielu różnych urządzeniach z różnymi ekranami. Dzięki nim dysponujesz obszernym zestawem narzędzi, których możesz używać do tworzenia odpowiednich stron. Nie postrzegaj zatem tej sytuacji jako źródła zamieszania, wyobraź sobie, że dysponujesz wieloma opcjami, spośród których możesz wybrać te najlepsze.

P: Wciąż nie do końca rozumiem, jak działają zasoby statyczne. Co się dzieje, kiedy umieszczę znacznik wewnątrz sekcji `<Page.Resources>`?

U: Dodając tam znacznik, aktualizujesz obiekt `Page`. Odszukaj zasób `AppName`, w którym podałeś nową wartość, by zmienić nagłówek strony:

```
<x:String x:Key="AppName">Witamy u Niechlujnego Janka</x:String>
```

A teraz przejrzyj kod, który IDE dodało podczas tworzenia szablonu `Basic Page`, i sprawdź, gdzie ten zasób jest używany:

```
<TextBlock x:Name="pageTitle" Grid.Column="1"
    Text="{StaticResource AppName}"
    Style="{StaticResource PageHeaderTextStyle}"/>
```

Strona używa `go`, by określić tekst. A zatem, co się tak naprawdę dzieje? Możesz się o tym przekonać, używając IDE. Umieść punkt przerwania w procedurze obsługi kliknięć przycisku, uruchom aplikację i kliknij przycisk. Do okna *Watch* dodaj wyrażenie `this.Resources["AppName"]` — zauważysz, że zawiera ono referencję do łańcucha znaków. W taki sposób działają wszystkie zasoby statyczne — dodanie takiego zasobu do kodu powoduje utworzenie obiektu i zapisanie referencji do niego w kolekcji o nazwie `Resources`.

P: Czy mogę używać zapisu `{StaticResource}` we własnym kodzie, czy tylko w szablonach, takich jak `Basic Page`?

U: Oczywiście — możesz tworzyć i używać zasobów w taki sposób. W szablonie `Blank Page` nie ma niczego szczególnego, podobnie jak w innych szablonach używanych w tej książce. Zawierają one zwyczajny kod C# i XAML i nie robią niczego, czego nie mógłbyś zrobić samemu.

P: Użyłem właściwości `x:Name`, by określić nazwę zasobu `MenuMaker`, natomiast zasób `AppName` używa właściwości `x:Key`. Czym one się różnią?

Nazwa „zasób statyczny” jest nieco myląca. Są one tworzone dla każdej instancji i w żadnym przypadku nie przypominają pól statycznych!

U: Zastosowanie właściwości `x:Key` powoduje, że zasób statyczny jest dodawany do kolekcji `Resources` i kojarzony z podanym kluczem, natomiast nie jest tworzone pole (a zatem nie możesz użyć identyfikatora `AppName` w kodzie C#, musisz odwoływać się do takiego zasobu, używając kolekcji `Resources`). W razie użycia właściwości `x:Name` zasób jest dodawany do kolekcji `Resources`, a oprócz tego jest dodawany jako pole do obiektu `Page`. To dzięki temu byłeś w stanie wywołać metodę `UpdateMenu()` na rzecz statycznego zasobu `MenuMaker`.

P: Czy ścieżka powiązania musi wskazywać na właściwość zawierającą łańcuch znaków?

U: Nie. Można wiązać właściwości dowolnych typów. O ile tylko można skonwertować typ właściwości źródłowej i docelowej, to powiązanie będzie działać. W przeciwnym razie dane zostaną zignorowane. Pamiętaj także, że nie wszystkie właściwości kontrolek są łańcuchami znaków. Załóżmy, że w kontekście danych jest dostępna właściwość logiczna o nazwie `EnableMyObject`. Możesz ją powiązać z dowolną właściwością typu logicznego, taką jak `IsEnabled`. W takim przypadku kontrolka będzie aktywowana i dezaktywowana zależnie od wartości właściwości `EnableMyObject`:

```
IsEnabled="{Binding EnableMyObject}"
```

Oczywiście, jeśli powiążesz ją z właściwością tekstową, to będzie wyświetlany jedynie tekst `True` lub `False` (co, jeśli się nad tym nieco zastanowić, będzie całkowicie zrozumiałe).

P: Dlaczego IDE wyświetla dane na formularzu, kiedy utworzę zasób statyczny i powiązanie w kodzie XAML, lecz nie kiedy zrobię to w kodzie C#?

U: Ponieważ IDE rozumie kod XAML, zawierający wszystkie informacje niezbędne do utworzenia obiektów koniecznych do wyświetlenia strony. Kiedy tylko utworzyłeś w kodzie XAML zasób `MenuMaker`, IDE utworzyło instancję tej klasy. Nie było w stanie zrobić tego na podstawie instrukcji `new` w konstruktorze, gdyż mógł on zawierać wiele innych instrukcji i także one musiałyby zostać wykonane. IDE wykonuje kod ukryty C# wyłącznie w wyniku uruchomienia aplikacji. Jeśli jednak dodasz do strony zasób statyczny, to IDE `go` utworzy, tak samo jak tworzy instancje kontrolek `TextBlock`, `StackPanel` i wszystkich innych. Ustawia także właściwości tych kontrolek, by je wyświetlić w oknie *Designer*, a zatem, kiedy określisz kontekst danych oraz ścieżki powiązań, także one zostaną użyte i w efekcie menu pojawi się w oknie IDE.

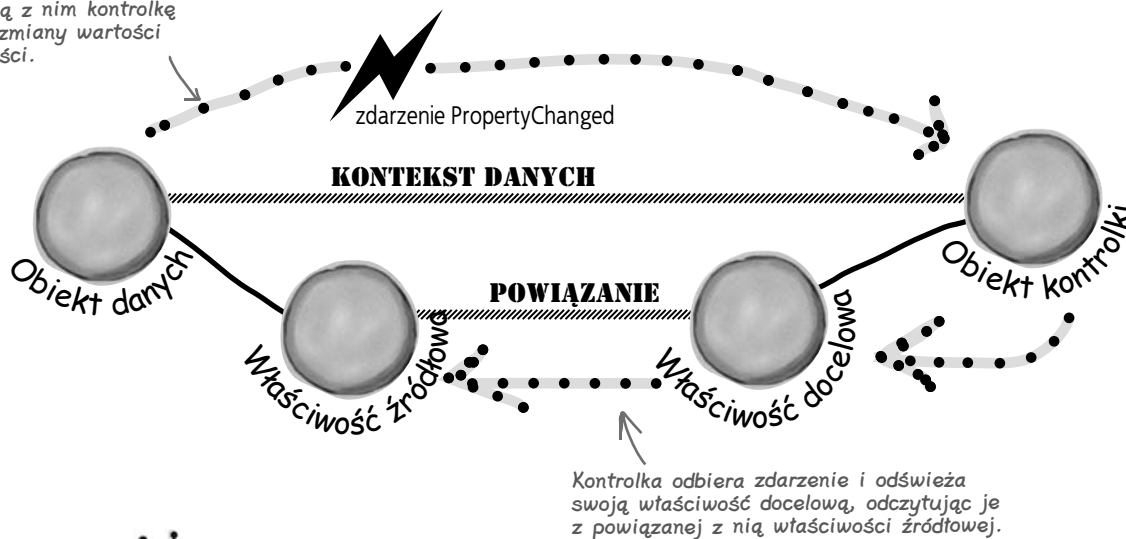
Zasoby statyczne dodawane do strony są tworzone w momencie wczytywania strony, a później mogą być używane przez dowolne obiekty aplikacji.

Interfejs `INotifyPropertyChanged` pozwala powiązanym obiektom przesyłać aktualizacje

Kiedy klasa `MenuMaker` aktualizuje menu kanapek, powiązana z nią kontrolka `ListViev` zostaje zaktualizowana. Jednak dokładnie w tym samym momencie klasa `MenuMaker` zmienia także wartość właściwości `GeneratedDate`. Dlaczego zatem nie jest aktualizowana powiązana z nią kontrolka `TextBlock`? Wynika to z faktu, że każda zmiana kolekcji typu `ObservableCollection` powoduje **zgłoszenie zdarzenia**, które informuje dowolną powiązaną kontrolkę o pojawieniu się zmian. W analogiczny sposób kliknięcie przycisku powoduje zgłoszenie zdarzenia `Click`, a upływanie czasu odmierzanego przez obiekt `Timer` powoduje zgłoszenie zdarzenia `Tick`. Zawsze gdy dodasz lub usuniesz jakies elementy kolekcji `ObservableCollection`, zgłosisz ona odpowiednie zdarzenie.

Możesz zadbać, by obiekty danych informowały o swoich zmianach właściwości docelowe i powiązane kontrolki. Należy w tym celu **zaimplementować interfejs `INotifyPropertyChanged`**, zawierający tylko jedno zdarzenie: `PropertyChanged`. Wystarczy zgłosić to zdarzenie po każdej zmianie wartości właściwości i obserwować, jak powiązana kontrolka zostanie automatycznie zaktualizowana.

Obiekt danych zgłasza zdarzenie `PropertyChanged`, by powiadomić dowolną powiązaną z nim kontrolkę o fakcie zmiany wartości właściwości.



Uwaga!

Kolekcje działają niemal w ten sam sposób co obiekty danych

W rzeczywistości klasa `ObservableCollection<T>` nie implementuje interfejsu `INotifyPropertyChanged`. Zamiast tego implementuje ściśle z nim związany interfejs `INotifyCollectionChanged`, który zamiast zdarzeń `PropertyChanged` generuje zdarzenia `CollectionChanged`. Kontrolka wie, że powinna oczekiwać na te zdarzenia, gdyż klasa `ObservableCollection` implementuje interfejs `INotifyCollectionChanged`. Zapisanie we właściwości `DataContext` obiektu implementującego ten interfejs sprawi, że kontrolka będzie reagować na zdarzenia `CollectionChanged`.

Zmodyfikuj klasę MenuMaker, by informowała Cię, gdy zmieni się właściwość GeneratedDate



To pierwszy raz, kiedy zgłaszasz zdarzenia.

Interfejs `INotifyPropertyChanged` został zdefiniowany w przestrzeni nazw `System.ComponentModel`, a zatem powinieneś zacząć od dodania na samym początku pliku klasy `MenuMaker` następującej instrukcji `using`:

```
using System.ComponentModel;
```

Zmodyfikuj klasę `MenuMaker`, dodając do jej deklaracji interfejsu `INotifyPropertyChanged`, a następnie skorzystaj z możliwości IDE, by go zaimplementować:



Procedury obsługi zdarzeń piszesz już od samego początku tej książki, jednak pierwszy raz masz okazję samemu zgłaszać zdarzenia. Wszystkiego na temat zgłaszania zdarzeń oraz tego, jak one działają, dowiesz się w rozdziale 15. Jak na razie wystarczy, żebyś wiedział, że interfejs może deklarować zdarzenia oraz że Twoja metoda `OnPropertyChanged()` jest zgodna ze standardowym wzorcem przekazywania zdarzeń do innych obiektów, stosowanym w C#.

Będzie to wyglądało nieco inaczej od przykładów, które widziałeś w rozdziałach 7. i 8. Nie będziesz musiał dodawać żadnych metod ani właściwości. Zamiast tego dodasz zdarzenie:

```
public event PropertyChangedEventHandler PropertyChanged;
```

A następnie dodasz poniższą metodę `OnPropertyChanged()`, której będziesz używał do zgłaszania zdarzenia `PropertyChanged`:

```
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEventArgs = PropertyChanged;
    if (propertyChangedEventArgs != null) {
        propertyChangedEventArgs(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

← To jest standardowy wzorec zgłaszania zdarzeń stosowany w .NET Framework.

Teraz jedyną rzeczą, którą musisz zrobić, by poinformować powiązaną kontrolkę o zmianie właściwości, jest wywołanie metody `OnPropertyChanged()` i przekazanie do niej nazwy właściwości, która uległa zmianie. Chcemy, by po każdej zmianie menu była aktualizowana kontrolka `TextBox` powiązana z właściwością `GeneratedDate`; w tym celu wystarczy, że dodamy do metody `UpdateMenu()` jeden wiersz kodu:

```
public void UpdateMenu() {
    Menu.Clear();
    for (int i = 0; i < NumberOfItems; i++) {
        Menu.Add(CreateMenuItem());
    }
    GeneratedDate = DateTime.Now;

    OnPropertyChanged("GeneratedDate");
}
```

Teraz po wygenerowaniu nowego menu powinna się także zmienić data.



Uwaga!

Nie zapomnij zaimplementować interfejsu `INotifyPropertyChanged`.

Mechanizm wiązania danych działa tylko wtedy, gdy kontrolki implementują ten interfejs. Jeśli do deklaracji klasy nie dodasz `: INotifyPropertyChanged`, to powiązane kontrolki nie będą aktualizowane — i to nawet jeśli obiekt danych będzie zgłaszał zdarzenia `PropertyChanged`.



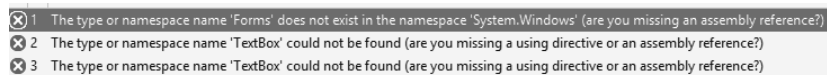
Ćwiczenia

Zakończ przerabianie gry *Idź na ryby!* do postaci aplikacji dla Sklepu Windows. Będziesz musiał zmodyfikować kod XAML przygotowany wcześniej w tym rozdziale, dodając do niego powiązania danych, skopiować wszystkie klasy i typy wyliczeniowe z aplikacji napisanej w rozdziale 8. (lub pobrać je z serwera FTP wydawnictwa Helion) i zaktualizować klasy `Player` oraz `Game`.

1 Dodaj istniejące pliki klas i dostosuj deklarowaną w nich przestrzeń nazw do bieżącej aplikacji.

Do swojego projektu dodaj następujące pliki, pochodzące z gry *Idź na ryby!*, którą napisałeś w rozdziale 8.: *Values.cs*, *Suits.cs*, *Card.cs*, *Deck.cs*, *CardComparer_bySuit.cs*, *CardComparer_byValue.cs*, *Game.cs* oraz *Player.cs*. Możesz skorzystać z opcji *Add Existing Item* dostępnej w oknie *Solution Explorer*, jednak w każdym z tych plików będziesz musiał **zmienić przestrzeń nazw**, dostosowując ją do przestrzeni używanej w projekcie (jak już robiłeś w przypadku kilku innych projektów w tej książce).

Spróbuj zbudować projekt. W plikach *Game.cs* oraz *Player.cs* powinny pojawić się błędy, przypominające te przedstawione poniżej:



2 Usuń odwołania do wszystkich klas i obiektów WinForms i dodaj do klasy Game odpowiednie instrukcje using.

Nie znajdujemy się już w świecie WinForms, dlatego też musisz usunąć z plików *Game.cs* oraz *Player.cs* instrukcje `using System.Windows.Forms`. Będziesz także musiał usunąć wszystkie odwołania do kontrolki `TextBox`. Oprócz tego konieczne będzie dodanie do klasy `Game` interfejsu `INotifyPropertyChanged` oraz kolekcji `ObservableCollection<T>`, a to oznacza, że na początku pliku *Game.cs* będziesz musiał dodać następujące instrukcje `using`:

```
using System.ComponentModel;
using System.Collections.ObjectModel;
```

3 Dodaj instancję klasy Game jako zasób statyczny i określ kontekst danych.

Zmodyfikuj kod XAML strony, dodając do niego instancję klasy `Game` jako zasób statyczny, i użyj jej jako kontekstu danych dla siatki zawierającej stronę *Idź na ryby!*, którą przygotowałeś we wcześniejszej części rozdziału. Oto kod XAML definiujący zasób statyczny: `<local:Game x:Name="game"/>`. Będziesz także potrzebował nowego konstruktora w klasie `Game`, gdyż zasoby mogą być tworzone tylko wtedy, gdy klasa definiuje konstruktor bezargumentowy:

```
public Game() {
    PlayerName = "Edek";
    Hand = new ObservableCollection<string>();
    ResetGame();
}
```

4 Do klasy Game dodaj publiczne właściwości, których użyjesz do zdefiniowania powiązań danych z kontrolkami.

Oto właściwości, które powiążesz z kontrolkami na stronie:

```
public bool GameInProgress { get; private set; }
public bool GameNotStarted { get { return !GameInProgress; } }
public string PlayerName { get; set; }
public ObservableCollection<string> Hand { get; private set; }
public string Books { get { return DescribeBooks(); } }
public string GameProgress { get; private set; }
```

5 Użyj mechanizmu wiązania danych, by aktywować i dezaktywować kontrolki TextBox, ListBox oraz Button.

Chcesz, by pole tekstowe *Imię* oraz przycisk *Rozpocznij grę* były aktywne wyłącznie wtedy, gdy gra jeszcze nie została rozpoczęta, oraz by kontrola *ListBox Ręka* i przycisk *Zażądaj karty* były aktywne tylko wtedy, gdy gra została już rozpoczęta. Dodasz do klasy *Game* kod określający wartość właściwości *GameInProgress*. Przyjrzyj się właściwości *GameNotStarted*. Przekonaj się, jak ona działa, a następnie dodaj następujące powiązania danych do kontrolki *TextBox*, *ListBox* oraz dwóch kontrolki *Button*.

Każdej z nich będziesz musiał użyć dwa razy.

```
{
  IsEnabled="{Binding GameInProgress}"      IsEnabled="{Binding GameNotStarted}"
  IsEnabled="{Binding GameInProgress}"      IsEnabled="{Binding GameNotStarted}"
}
```

6 Zmodyfikuj klasę Player, by zmuszała klasę Game do wyświetlania postępów w grze.

Wcześniejsza wersja klasy *Player*, używana w aplikacji *WinForms*, używała kontrolki *TextBox* przekazywanej w wywołaniu jej konstruktora. Zmień ją, by przekazywana była referencja typu *Game*, zapisywana następnie w polu prywatnym. (Spójrz na przedstawioną poniżej metodę *StartGame()*, by zobaczyć, jak ma być używany ten nowy konstruktor podczas dodawania graczy). Odszukaj wiersze zawierające odwołania do kontrolki *TextBox* i zastąp je wywołaniami metody *AddProgress()* obiektu *Game*.

7 Zmodyfikuj klasę Game.

Zmień typ wynikowy metody *PlayOneRound()* z *bool* na *void*, a następnie zmodyfikuj jej kod tak, by postępy w grze były wyświetlane przez metodę *AddProgress()*, a nie poprzez odwoływanie się do kontrolki *TextBox*. Jeśli gracz wygrał, wyświetl postęp gry, przywróć ją do stanu początkowego i zakończ działanie metody. W przeciwnym razie odśwież kolekcję *Hand* i wyświetl informacje o kartach gracza. Będziesz także musiał dodać lub zaktualizować poniższe cztery metody i określić, do czego one służą i jak mają działać.

```
public void StartGame() {
    ClearProgress();
    GameInProgress = true;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    Random random = new Random();
    players = new List<Player>();
    players.Add(new Player(PlayerName, random, this));
    players.Add(new Player("Bartek", random, this));
    players.Add(new Player("Janek", random, this));
    Deal();
    players[0].SortHand();
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
    OnPropertyChanged("Books");
}

public void AddProgress(string progress)
{
    GameProgress = progress +
        Environment.NewLine +
        GameProgress;
    OnPropertyChanged("GameProgress");
}
```

```
public void ClearProgress() {
    GameProgress = String.Empty;
    OnPropertyChanged("GameProgress");
}
```

Będziesz także musiał zaimplementować interfejs *INotifyPropertyChanged* i dodać tę samą metodę *OnPropertyChanged()*, którą dodałeś do klasy *MenuMaker*. Używaj jej zaktualizowane metody i będzie jej także używać metoda *PullOutBooks()*.

```
public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}
```



Rozwiązania ćwiczeń

Poniżej przedstawiliśmy cały kod ukryty, który miałeś napisać:

```
private void startButton_Click(object sender, RoutedEventArgs e) {
    game.StartGame();
}
private void askForACard_Click(object sender, RoutedEventArgs e) {
    if (cards.SelectedIndex >= 0)
        game.PlayOneRound(cards.SelectedIndex);
}
private void cards_DoubleTapped(object sender, DoubleTappedRoutedEventArgs e) {
    if (cards.SelectedIndex >= 0)
        game.PlayOneRound(cards.SelectedIndex);
}
```

A oto zmiany, które miałeś wprowadzić w klasie Player:

```
class Player {
    private string name;
    public string Name { get { return name; } }
    private Random random;
    private Deck cards;
    private Game game;
    public Player(String name, Random random, Game game) {
        this.name = name;
        this.random = random;
        this.game = game;
        this.cards = new Deck(new Card[] { });
        game.AddProgress(name + " przyłączył się do gry");
    }
    public Deck DoYouHaveAny(Values value)
    {
        Deck cardsIHave = cards.PullOutValues(value);
        game.AddProgress(Name + " ma " + cardsIHave.Count + " " + Card.Plural(value, cardsIHave.Count));
        return cardsIHave;
    }
    public void AskForACard(List<Player> players, int myIndex, Deck stock, Values value) {
        game.AddProgress(Name + " pyta czy ktoś ma " + Card.Plural(value, 1));
        int totalCardsGiven = 0;
        for (int i = 0; i < players.Count; i++) {
            if (i != myIndex) {
                Player player = players[i];
                Deck CardsGiven = player.DoYouHaveAny(value);
                totalCardsGiven += CardsGiven.Count;
                while (CardsGiven.Count > 0)
                    cards.Add(CardsGiven.Deal());
            }
        }
        if (totalCardsGiven == 0) {
            game.AddProgress(Name + " pobrał kartę z kupki.");
            cards.Add(stock.Deal());
        }
    }
}

// ... reszta kodu klasy Player nie została zmieniona ...
```

Oto zmiany w kodzie XAML strony:

```
<Grid Grid.Row="1" Margin="120,0,60,60" DataContext="{StaticResource ResourceKey=game}" >
  <TextBlock Text="Imię" Margin="0,0,0,20"
    Style="{StaticResource SubheaderTextStyle}"/>
  <StackPanel Orientation="Horizontal" Grid.Row="1">
    <TextBox x:Name="playerName" FontSize="24" Width="500" MinWidth="300"
      Text="{Binding PlayerName, Mode=TwoWay}" IsEnabled="{Binding GameNotStarted}" />
    <Button x:Name="startButton" Margin="20,0" IsEnabled="{Binding GameNotStarted}"
      Content="Rozpocznij grę!" Click="startButton_Click" />
  </StackPanel>
  <TextBlock Text="Postępy gry"
    Style="{StaticResource SubheaderTextStyle}" Margin="0,20,0,20" Grid.Row="2" />
  <ScrollViewer Grid.Row="3" FontSize="24" Background="White" Foreground="Black"
    Content="{Binding GameProgress}" />
  <TextBlock Text="Grupy" Style="{StaticResource SubheaderTextStyle}"
    Margin="0,20,0,20" Grid.Row="4"/>
  <ScrollViewer FontSize="24" Background="White" Foreground="Black"
    Grid.Row="5" Grid.RowSpan="2" Content="{Binding Books}" />
  <TextBlock Text="Ręka" Style="{StaticResource SubheaderTextStyle}"
    Grid.Row="0" Grid.Column="2" Margin="0,0,0,20"/>
  <ListBox Background="White" FontSize="24" Height="Auto" Margin="0,0,0,20"
    x:Name="cards" Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"
    ItemsSource="{Binding Hand}" IsEnabled="{Binding GameInProgress}"
    DoubleTapped="cards_DoubleTapped" />
  <Button x:Name="askForACard" Content="Zażądaj karty" HorizontalAlignment="Stretch"
    VerticalAlignment="Stretch" Grid.Row="6" Grid.Column="2"
    Click="askForACard_Click" IsEnabled="{Binding GameInProgress}" />
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="5*" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="2*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" MinHeight="150" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
</Grid>
```

Ta kontrolka
TextBox używa
dwukierunkowego
powiązania
z właściwością
PlayerName.

Kontekstem danych dla siatki jest klasa
Game, gdyż wszystkie powiązania
odwołują się do właściwości tej klasy.

To procedura obsługi zdarzeń Click przycisku,
który rozpoczyna grę.

Kontrolki ScrollViewer
prezentujące postępy gry
oraz grupy są powiązane
z właściwościami
Progress oraz Books.

Właściwość IsEnabled aktywuje
i dezaktywuje kontrolkę. Jest to właściwość
typu Boolean, można ją zatem powiązać
z właściwością tego samego typu dostępną
w kontrolce, by włączać ją i wyłączać na
podstawie wartości właściwości.



Rozwiązania ćwiczeń

Poniżej przedstawiliśmy wszystkie zmiany, które należało wprowadzić w kodzie klasy Game, łącznie z kodem, który podaliśmy już w instrukcjach do ćwiczenia.

```
using System.ComponentModel;
using System.Collections.ObjectModel;

class Game : INotifyPropertyChanged {
    private List<Player> players;
    private Dictionary<Values, Player> books;
    private Deck stock;
    public bool GameInProgress { get; private set; }
    public bool GameNotStarted { get { return !GameInProgress; } }
    public string PlayerName { get; set; }
    public ObservableCollection<string> Hand { get; private set; }
    public string Books { get { return DescribeBooks(); } }
    public string GameProgress { get; private set; }

    public Game() {
        PlayerName = "Edek";
        Hand = new ObservableCollection<string>();
        ResetGame();
    }

    public void AddProgress(string progress) {
        GameProgress = progress + Environment.NewLine + GameProgress;
        OnPropertyChanged("GameProgress");
    }

    public void ClearProgress() {
        GameProgress = String.Empty;
        OnPropertyChanged("GameProgress");
    }

    public void StartGame() {
        ClearProgress();
        GameInProgress = true;
        OnPropertyChanged("GameInProgress");
        OnPropertyChanged("GameNotStarted");
        Random random = new Random();
        players = new List<Player>();
        players.Add(new Player(PlayerName, random, this));
        players.Add(new Player("Bartek", random, this));
        players.Add(new Player("Janek", random, this));
        Deal();
        players[0].SortHand();
        Hand.Clear();
        foreach (String cardName in GetPlayerCardNames())
            Hand.Add(cardName);
        if (!GameInProgress)
            AddProgress(DescribePlayerHands());
        OnPropertyChanged("Books");
    }
}
```

Te instrukcje są niezbędne, by móc korzystać z interfejsu `INotifyPropertyChanged` i klasy `ObservableCollection`.

Te właściwości są używane przez mechanizm wiązania danych.

To jest nowy konstruktor klasy `Game`. Tworzymy tylko jedną kolekcję i czyszcimy ją w momencie przywracania początkowego stanu gry. Gdybyśmy tworzyli nowy obiekt, to formularz straciłby odwołanie do niego, a aktualizacje zostałyby przerwane.

Te metody zapewniają działanie mechanizmu wiązania danych w grze. Nowe wiersze są dodawane na górze, dzięki czemu wcześniejsze zdarzenia są przesuwane ku dołowi kontrolki `ScrollView`.

Wszystkie programy, które napisałeś podczas lektury tej książki, można przerobić na aplikacje dla Sklepu Windows napisane przy użyciu języka XAML. Można je jednak napisać na bardzo wiele sposobów, co jest szczególnie prawdziwe w przypadku korzystania z języka XAML! To właśnie dlatego pokazaliśmy Ci tak dużo kodu w ramach opisu ćwiczenia.

Oto metoda `StartGame()`, którą pokazaliśmy już wcześniej. Czyści ona postępy gry, tworzy graczy, rozdaje karty, a następnie aktualizuje postępy gry i grupy.

← Ta metoda wcześniej zwracała wartość typu bool, dzięki której formularz mógł aktualizować przebieg gry. Teraz musi ona jedynie wywoływać metodę AddProgress(), a o resztę zatroszczy się mechanizm wiązania danych.

```
public void PlayOneRound(int selectedPlayerCard) {
    Values cardToAskFor = players[0].Peek(selectedPlayerCard).Value;
    for (int i = 0; i < players.Count; i++) {
        if (i == 0)
            players[0].AskForACard(players, 0, stock, cardToAskFor);
        else
            players[i].AskForACard(players, i, stock);
        if (PullOutBooks(players[i])) {
            AddProgress(players[i].Name + " ma nową grupę");
            int card = 1;
            while (card <= 5 && stock.Count > 0) {
                players[i].TakeCard(stock.Deal());
                card++;
            }
        }
        OnPropertyChanged("Books");
        players[0].SortHand();
        if (stock.Count == 0) {
            AddProgress("Na kupce nie ma już żadnych kart. Gra skończona!");
            AddProgress("Zwycięzcą jest... " + GetWinnerName());
            ResetGame();
            return;
        }
    }
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
}

```

← Grupy uległy zmianie, a formularz musi o tym wiedzieć, by mógł zaktualizować kontrolki ScrollView.

← Oto zmiany wprowadzone w metodzie PlayOneRound(), które aktualizują postęp gry, gdy została ona zakończona, bądź aktualizują rękę i grupy, jeśli gra jeszcze trwa.

```
public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}

```

← A to jest metoda ResetGame() z instrukcji do ćwiczenia. Czyści ona grupy, rękę i kupkę kart.

```
public event PropertyChangedEventHandler PropertyChanged;
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

← To jest standardowy wzorzec użycia właściwości PropertyChanged, przedstawiony już wcześniej w tym rozdziale.

// ... reszta kodu klasy Game nie została zmieniona ...

Skorowidz

A

abstrakcja, 366, *Patrz też:* klasa
abstrakcyjna, metoda abstrakcyjna
 get, 261, 264, 267, 612
 set, 261, 263, 267, 283
animacja, 27, 140, 816
 obiektów, 808
 wartości
 obiektów, 808
 zmiennoprzecinkowych, 807
aplikacja, 27, 45, 134
 architektura, 230
 cykl życia, 552
 zarządzanie, 748
 dla Sklepu Windows, 519
 konsolowa, 304, 305, 306, 410,
 684, 878
 nawigacja trójpoziomowa, 727
 o luźnych powiązaniach, 27
 publikowanie, 90
 pusta, 54
 Split App, *Patrz:* szablon Split App
 struktura, 52
 tworzenie, 96, 115, 129, 135, 720
 uruchamianie na innym
 komputerze, 91
 Windows Forms, *Patrz:* Windows
 Forms Application
 Windows Phone, *Patrz:* Windows
 Phone
 wykorzystująca strony, 686
 zawieszenie, 748, 749
assembly, *Patrz:* złożenie
attribut, 477
 DataContract, 577
 DataMember, 577
 DllImport, 646

B

biblioteka
 .NET, 76
 for Windows Desktop, 99
 for Windows Store, 99
 klas, 351, 882, 883

Blend for Visual Studio, 824
blok
 catch, 614, 617, 618, 620, 622, 625,
 626, 631, 633, 634
 finally, 620, 622, 631, 633
 try, 614, 617, 618, 620, 622, 631,
 633
 using, 632, 633, *Patrz też:* słowo
 kluczowe using
błąd, 76, 81, 539, 617
 inconsistent accessibility, 230
 kompilacji, 109, 230

C

CLR, 99, 210, 211, 646, 659, 885
code-behind, *Patrz:* kod ukryty
Common Language Runtime, *Patrz:*
 CLR

D

dane
 kontekst, *Patrz:* kontekst danych
 kontrakt, 567, 577
 serializacja, *Patrz:* serializacja
 kontraktu danych
 łączenie, 705
 modyfikacja, 681
 sortowanie, *Patrz:* lista sortowanie
 szablon, *Patrz:* szablon danych
 wiązanie, *Patrz:* wiązanie danych
data binding, *Patrz:* wiązanie danych
debugger, 92, 111, 112, 397, 405, 411,
 526, 528, 605, 609, 610, 617, 618,
 751, 761, 767
 ikony, 609
 krokowe wykonanie programu, 617
delegat, 737, 758, 759, 760, 764, 766,
 768, 770, 869, 898
deserializacja, 472, 473, 476, 614
diagram klas, 149, 164, 166, 320,
 341, 370
diament śmierci piekielny, 364
drzewo obiektów, 752, 753
dyrektywa, *Patrz:* słowo kluczowe

dziedziczenie, 16, 275, 285, 286, 287,
 294, 297, 303, 308, 327, 331, 366,
 608, 655, 671
 interfejsów, *Patrz:* interfejs
 dziedziczenie
 wielokrotne, 364
dźwięk, 857

E

ekran
 dotykowy, 848, 852, 860
 startowy, 89
 tytułowy, 89
element wyliczeniowy, 387
elipsa, 825
enumerator, 413, 681, 893, 894
etykieta, 131

F

factory method, *Patrz:* wzorzec metody
 wytwórczej
finalizator, 646, 647, 648, 649, 650,
 652, 653
format
 ASCII, 492
 big endian, 492
 binarny, 480, 481, *Patrz też:* system
 dwojkowy
 little endian, 492
 Unicode, 480, 481, 492
 UTF-8, 492
formularz, 172, 177, 209, 210, 265, 266,
 524, 745
 kolor tła, 140
 przycisk
 maksymalizacji, 148
 minimalizacji, 148
 z kontrolką, 219
 zakładka, 281
funkcja zwrotna, 764, 766, 767, 768

G

GAC, 883
garbage collection, *Patrz:* odśmiecianie

Skorowidz

GDI+, 896
geocaching, 256
Global Assembly Cache, *Patrz:* GAC
Go To Definition, 461
GPS, 256
gra Invaders, *Patrz:* Invaders
graf, 475, 476, 524, 527, 552, 576, 581, 755
Graphical User Interface, *Patrz:* GUI
GUI, 52, 152, 228

H

heisenbug, 612
hermetyzacja, 15, 235, 247, 249, 250, 252, 255, 256, 257, 259, 260, 261, 263, 267, 354, 366, 374, 587, 668, 786
 automatyczna, 263
hex dump, *Patrz:* widok szesnastkowy
hierarchia klas, *Patrz:* klasa hierarchia
Hyper-V, 861

I

indeksator, 894
instrukcja, 11, 123, 267, 631, *Patrz też:*
 słowo kluczowe
 blok, 104
 break, 469, 470
 if/else, 114, 118, 189, 210, 279, 468
 new, *Patrz:* słowo kluczowe new
 return, 103, 146, 147, 469
 skoku, 878
 switch, 469, 470
 szablon, *Patrz:* szablon instrukcji
 throw, 626, 631
 using, *Patrz:* słowo kluczowe using
 warunkowa, 880, 897
 yield return, 894
IntelliSense, *Patrz:* okno IntelliSense
interfejs, 17, 329, 332, 333, 334, 336, 348, 354, 356
 dziedziczenie, 341
 ICollection, 893
 IComparable, 405
 IComparer, 406, 407, 408, 409
 IDisposable, 461, 462, 632, 633, 648
 IEnumerable, 413, 414, 431, 435, 438, 681, 682, 694, 893
 IEquatable, 890
 implementacja, 334, 348, 401

INotifyPropertyChanged, 556, 557, 785
instancja, 338, 354
ISemanticZoomInformation, 713
IStorageFile, 578, 579
IStorageFolder, 578, 579, 580
IStorageItem, 579
IValueConverter, 798
nazwa, 333
publiczny, 333
referencja, 338, 339, 354
rozszerzający, 671
rzutowanie, 347
użytkownika, 45, 787, 790, 814, 886
graficzny, *Patrz:* GUI
wbudowany w .NET, 333
właściwość, 340
Invaders, 29, 836
iteracja, 119, 121, 212, 398, 682, 685, 695, 705, 708, 845, 878, 893

J

język
 XAML, *Patrz:* XAML
 XML, *Patrz:* XML

K

kalkulator, 183
kanał RSS, 901
karta, 100, 281
klasa, 11, 17, 102, 103, 107, 123, 135, 272, 549, 668
 abstrakcyjna, 356, 358
 atribut, *Patrz:* atrybut
 bazowa, 286, 289, 299, 310, 345, 655
 BinaryFormatter, 476, 567, 614
 BinaryReader, 484
 BinaryWriter, 483
 Binding, 543
 ContainerControl, 527
 Control, 524
 diagram, *Patrz:* diagram klas
 Directory, 456
 DoubleAnimation, 807
 EventArgs, 734
 Exception, 608, 631
 Excuse, 493, 589
 File, 456, 492, 570
 FileInfo, 456, 492
 FileIO, 570, 572
 FileOpenPicker, 572
 FileSavePicker, 572
 FileStream, 492
 hermetyczna, 258, 354, 493, 548
 hierarchia, 287, 292, 309, 345
 instancja, 152, 552, 671
 pole, *Patrz:* pole
 tworzenie, 154
 ItemsControl, 821
 KnownFolders, 580
 List, 394
 Math, 117
 MenuMaker, 546, 547, 553, 556
 MessageBox, 133, 136, 570
 modelu, 797, 897
 modelu widoku, 797
 nadpisywanie, 823
 nadrzędna, *Patrz:* klasa bazowa
 nazwa, 162
 Object, 288, 411
 ObservableCollection, 543, 820
 pochodna, *Patrz:* klasa potomna
 potomna, 286, 290, 293, 299, 351
 przesłanianie, 290, 298
 public, 107, *Patrz też:* słowo
 kluczowe public
 rozszerzanie, 293
 ScrollableControl, 527
 Selector, 821
 SerializationException, 618
 SettingsPage, 770
 składowe, 351
 zasięg, 352
 statyczna, 456, 670, 770
 Stream, 443
 StreamReader, 449, 457
 StreamWriter, 445, 446
 struktura, 164
 strumieni .NET, 19
 System.Windows.Form, 527
 Task, 587
 TextBlock, 528, 533
 tworzenie, 176
 typ, 687
 Type, 889
 UIElement, 823
 UserControl, 781, 786
 wbudowana, *Patrz:* zestaw

- widoku, 797, 820
 - Windows.Storage, 570
 - zagnieżdżanie, 402, 632, 889
 - klawiatura, 848, 852
 - klucz, 702, 707
 - Rejestru systemowego, 883
 - klucz-wartość, 413
 - kod
 - IL, 885
 - maszynowy, 885
 - ukryty, 543
 - znacznikowy, 543
 - źródłowy, 885
 - kolejka, 435, 436
 - kolekcja, 18, 543, 556, 680, 684, 694
 - Children, 545, 814, 817, 820, 821
 - generyczna, 398, 401, 435
 - inicjalizator, 402
 - ObservableCollections, 776, 787, 823
 - kolizja, 840, 845
 - kolor, 140
 - komentarz, 102, 111
 - kompilator, 98, 269, 682, 885
 - komponent, *Patrz:* kontrolka
 - konkatenacja, *Patrz:* łańcuch znaków
 - konkatenacja
 - konsola, 262
 - konstruktor, 265, 266, 310, 386, 619
 - bezargumentowy, 267, 553, 817
 - klasy
 - bazowej, 311
 - potomnej, 311
 - przeciążony, 471, 817
 - StreamWriter, 445
 - kontekst danych, 542, 547, 550, 551, 552, 553
 - kontrakt danych, *Patrz:* dane kontrakt
 - serializacja, *Patrz:* serializacja kontraktu danych
 - kontrolka, 52, 66, 96, 99, 115, 132, 219, 454, 530
 - AnimatedImage, 825, 849
 - AppBar, 572
 - BackgroundWorker, 886
 - Border, 544, 572, 753
 - Button, 532
 - Canvas, 63, 65, 66, 85, 545, 814, 818, 826
 - CheckBox, 125, 281
 - ComboBox, 321, 371, 375, 423, 543
 - ContentControl, 65
 - dodawanie, 62, 63
 - Grid, 52, 65, 528, 532, 535, 536, 538, 544, 545, 555, 753, 814
 - GridView, 713, 821
 - interfejsu użytkownika, 814
 - kontekst danych, 542
 - kontenera, 59
 - kopiowanie, 281
 - ListBox, 216, 543, 821
 - ListView, 692, 713, 777, 821
 - niewizualna, *Patrz:* komponent
 - NumericUpDown, 148
 - NumericUpDown, 281
 - OpenFileDialog, 454
 - PictureBox, 138, 228, 230, 232, 289, 498
 - Polygon, 825
 - Popup, 838, 850
 - ProgressBar, 52, 66, 692, 886, *Patrz też:* pasek postępu
 - ScrollView, 532, 533, 535, 542, 543, 544, 572
 - StackPanel, 52, 65, 539, 555, 753, 777, 814
 - StatusStrip, 216
 - szablon, *Patrz:* szablon kontrolek
 - TextBlock, 63, 65, 67, 117, 542, 543, 544, 545
 - TextBox, 148, 281, 459, 532, 542, 543, 547, 551
 - Timer, 215, 216, 231
 - ToggleSwitch, 753
 - tworzenie, 781
 - użytkownika, 781, 786, 817
 - z zawartością, 544
 - zagnieżdżanie, 545
 - zoom semantyczny, 712, 714
 - konwerter, 27, 773, 798, 800
 - BooleanNotConverter, 800
 - BooleanVisibilityConverter, 800
 - wartości, 798
 - kowariancja, 414
- L**
- library assembly, *Patrz:* złożenie biblioteczne
 - licznik czasu, 53, 80, 82
 - LINQ, 25, 680, 681, 685, 694, 695, 698, 701, 845, 898
 - LINQ to XML, 900, 901
 - LINQPad, 718
 - Linux, 885
 - lista, 393, 394, 395, 395, 396, 397, 398, 401, 422, 439
 - ArrayList, 401
 - nienegeneryczna, *Patrz:* lista ArrayList
 - sortowanie, 404, 408, 681
 - stała, 401
 - wyliczeniowa, 387
 - literał, 183, 189, 211, 240, 241, 242, 243, 889
- Ł**
- ładowanie bezpośrednio, 91
 - łańcuch
 - strumieni, 450
 - znaków, 110, 147, 205, 243, 386, 411, 457, 483, 549
 - dzielenie na fragmenty, 471
 - konkatenacja, 188, 411
 - konwersja na tablicę bajtów, 457
- M**
- Macintosh, 885
 - managed resources, *Patrz:* zasoby zarządzane
 - manifest
 - aplikacji, 53
 - pakietu, 580, 586
 - maszyna wirtualna, 99, 210
 - metoda, 11, 73, 102, 103, 211, 267, 351
 - abstrakcyjna, 356, 359
 - anonimowa, 768, 898, 899
 - AppendAllText, 456
 - AppendLines, 570
 - AppendLinesAsync, 570
 - Application.DoEvents, 140, 377, 382, 886,
 - argument, 188, *Patrz też:* metoda parametr
 - nazwany, 664
 - przekazywany przez referencję, 656, 657, 660, 661, 662, 663
 - przekazywany przez wartość, 656, 657, 660, 661, 662, 663

metoda

Array.Resize, 392
asynchroniczna, 22, 565, 578, 587, 592
Close, 445, 457, 492
Color.FromArgb, 140
Compare, 406
CompareTo, 405
Console.Error.WriteLine, 490
Console.ReadKey, 409, 410
Console.WriteLine, 262, 409, 411, 412, 431, 662
CreateDirectory, 456
definicja, 461
deklaracja, 147
Delete, 456
Deserialize, 476
Dispose, 461, 462, 633, 648, 650, 651, 652
Enum.Parse, 472
Equals, 890
Exists, 456
File.Create, 483, 485
File.OpenWrite, 483, 485
GC.Collect, 646, 653
GetEnumerator, 413, 681, 893
GetFiles, 456
GetHashCode, 890
GetLastAccessTime, 456
GetLastWriteTime, 456
GetLeft, 826
GetType, 889
InitializeComponent, 528
IsNullOrEmpty, 320, 351
List.Sort, 405, 407
Main, 135, 139, 304
MessageBox.Show, 137, 189, 431
MessageBox.Show, 137
MoveNext, 413, 894
Navigate, 687
nazwa, 81, 162, 269, 587, 897
Object.ReferenceEquals, 890, 892
obsługująca zdarzenie, 735
OnLaunched, 748, 751
OpenFile, 575
parametr, 103, 189, 207, 220, *Patrz też*: metoda argument logiczny, 241 opcjonalny, 664

wyjściowy, 662
pomocnicza, 822
prywatna, 251, 252, 844, 845
przełączona, 389
tworzenie, 415
przesłanie, 290, 298, 306, 309, 310, 331, 411, 892
publiczna, 252, 259
nazwa, 269
Read, 443
ReadAllBytes, 481, 492
ReadAllText, 459, 481, 492
ReadBlock, 489
ReadExcuseAsync, 595, 618
ReadText, 570
ReadTextAsync, 570
RemoveAt, 398
rozszerzająca, 670, 671, 672, 681
SaveCurrentExcuseAsync, 595
SaveFile, 575
Seek, 443
Serialize, 476
SetLeft, 826
SetTarget, 817
SetTargetProperty, 817
ShowAsync, 615
ShowDialog, 453, 454, 455
Sleep, 140, 142, 289, 377, 612
Sort, 405, 407, 408
Split, 471
statyczna, 157, 159, 249, 303, 670
Stream.Read, 490, 492
String.Format, 411
sygnatura, 267, 735
szkielet, 73, 74
ToArray, 401
ToList, 843
ToString, 188, 411, 412
TryParse, 663
ukrywanie, 306, 307, 309
wartość zwracana, 103, 146
wirtualna, 303
Write, 443, 445, 457
WriteAllBytes, 492
WriteAllText, 459, 492
WriteExcuseAsync, 595
WriteLine, 445, 457
wytwórcza, 822, 856

Microsoft Developer Network, *Patrz*: MSDN

model, 777, 786, 788, 792

widoku, 777, 786, 787, 788, 809, 846, 851

modyfikator, 662

async, 568

dostępu, 351, 352, 884

internal, 852, 884

out, 662

public, 884

ref, 663

sealed, 670

Mono, 885

MSDN, 90

MVC, 787

MVVM, 27, 773, 777, 787, 790, 797, 822, 838

N

notacja

Pascal, 260, 269

wielbłądzia, 260, 269

O

obiekt, 15, 26, 143, 150, 158, 195, 197, 198, 210, 549, 639, 655, 659

Button, 526

definicja, 461

Exception, 604, 608, 612, 625, 656

FileStream, 444, 445, 450, 456

Frame, 687

graf, *Patrz*: graf

inicjalizator, 175, 177, 264, 691

OpenFileDialog, 455, 457

Page, 528

porównywanie, 404, 408

SaveFileDialog, 455

stan, 474

Stream, *Patrz*: strumień

StreamReader, 489, 492

StreamWriter, 483, 492

tworzenie, 150, 151, 160, 175, 242

usuwanie, 461, 646, 647, 648

odświeżanie, 198, 201, 210, 211, 646, 647, 650, 653

okno

Designer, 54, 123, 536, 537, 552, 872

Device, 537

- dialogowe, 453, 454
- tworzenie, 457
- Document Outline, 66, 67, 84, 572
- Error List, 76, 82, 101, 104, 264
- IntelliSense, 74, 76, 86, 101, 255, 265, 389, 553
- MessageBox, 148, 160, 171, 187, 648
- MessageDialog, 768
- O aplikacji, 850
- Output, 262
- Properties, 64, 218, 572
- Reference Manager, 883
- Solution Explorer, 54, 56, 100, 324, 351, 778, 882, 884
- Solution Manager, 303
- Toolbox, 125, 767
- Watch, 112, 527, 528, 555, 611, 612, 617, 648
- \$exception, 604
- this, 527
- wiersza poleceń, 306, 885
- oop, *Patrz:* programowanie obiektowe
- opakowywanie, 667, 668
- operator, *Patrz też:* znak
 - , 110
 - !, 110
 - !=, 118, 892
 - &&, 118, 141
 - *=, 110
 - ||, 118
 - +, 110, 187, 188, 411, 457
 - ++, 110
 - +=, 110
 - =, 114, 123, 188
 - =, 194
 - ==, 114, 118, 123, 890, 892
 - await, 568, 587, 615
 - lambda, 899
 - logiczny, 85, 118, 880
 - new, 157
 - przeciążanie, 892
 - przypisania, *Patrz:* operator ==
 - warunkowy, 118
- P**
 - pamięci oczyszczanie, *Patrz:* odświeżanie
 - pasek
 - aplikacji, 591, 802
 - narzędzi
 - Debug, 609
 - Debug Location, 751
 - postępu, 66
 - przewijania, 692
 - pętla, 113
 - for, 121
 - foreach, 397, 398, 411, 435, 843, 893, 894
 - nieskończona, 121, 140, 141
 - while, 113, 119, 123, 141, 279
 - piksel, 537
 - platforma .NET, 11, 99
 - plik, 415, 441, 442, 443, 444, 445, 446, 449, 450, 456, 457, 459, 570, 579
 - .cs, 53, 544
 - .Designer.cs, 324
 - .DLL, 882, 884
 - .EXE, 53, 882, 884
 - .png, 53
 - .resx, 324
 - App.xaml.cs, 46
 - binarny, 480, 481, 482, 484, 485, 486, 487, 576
 - Form1.cs, 303
 - MainPage.xaml, 46, 54, 55, 56, 75, 96, 107, 687
 - MainPage.Xaml.cs, 46, 75, 96, 107, 529, 550
 - Package.manifest, 580
 - Program.cs, 134
 - Program.cs, 134, 303, 304
 - SplitPage.xaml, 721
 - StandardStyles.xaml, 573
 - system, *Patrz:* system plików
 - ścieżka dostępu, 570
 - tekstowy, 431
 - wykonywalny, 98
 - XML, 576, 577, 581
 - zamykanie, 444
 - podklasa, *Patrz:* klasa potomna
 - pole, 158, 211, 351, 354
 - nazwa, 897
 - prywatne, 249, 251, 255, 260, 779, 851
 - inicjalizacja, 265
 - nazwa, 269
 - przestanianie, 266, 273
 - publiczne, 264, 542
 - tekstowe, 172, 173
 - tworzenie, 159, 267
 - wewnętrzne, 261, 372
 - wyboru, 125, 131, 246, 247
 - polimorfizm, 366, 367, 393, 655, 669
 - powiązanie dwukierunkowe, 543, 547
 - procedura obsługi zdarzeń, 84, 132, 262, 589, 731, 735, 746, 750, 848
 - łańcuch, 737, 746
 - nazwa, 735
 - PointerEntered, 85, 86
 - PointerPressed, 85
 - standardowa, 737
 - tworzenie, 738
 - process assembly, *Patrz:* złożenie wykonywalne
 - program, 98
 - Blend for Visual Studio, *Patrz:* Blend for Visual Studio
 - debugowanie, *Patrz:* debugger
 - LINQPad, *Patrz:* LINQPad
 - odporny, 614, 666
 - zatrzymywanie, 77, 81
 - programowanie
 - obiektowe, 366
 - specyfikacja, 424
 - prostokąt, 825
 - przeciążanie, 389
 - przestrzeń nazw, 11, 102, 107, 123, 135, 324, 415, 552, 586, 672, 797, 882
 - System, 117, 123, 476
 - System.ComponentModel, 557
 - SYSTEM.IO, 566
 - System.Linq, 123
 - System.Runtime.Serialization, 618
 - using, 552
 - Windows.Foundation, 827
 - Windows.Storage, 579, 580
 - Windows.UI.ApplicationSettings, 770
 - Windows.UI.Xaml, 823, 829
 - Windows.UI.Xaml.Controls, 687
 - Windows.UI.Xaml.Data, 798
 - przetwarzanie skrócone, 879, 880
 - przycisk, 117, 125, 131, 172, 177, 777
 - maksymalizacji formularza, 148
 - minimalizacji formularza, 148
 - przycisk Undo, 123
 - pułapka, 610, 611, 612, 756

Skorowidz

punkt

przerwania, 111, 527
wejścia, 134, 135, 136, 139, 303

Q

Queue, *Patrz:* kolejka

R

refaktoryzacja, 896
referencja, 196, 197, 198, 204,
207, 209, 211, 347, 354, 371, 647,
655, 656, 669, *Patrz też:* zmienna
referencyjna
do klasy pochodnej, 302
do kontrolki, 219
do samego siebie, 209
null, 392, 665
obiektu wywołującego zdarzenie,
737
wielokrotna, 199
rekurencja, 661
relacja, 294
Remote Debugger, 91
rodzic, 288, *Patrz też:* klasa bazowa
routed events, *Patrz:* zdarzenie
trasowane
rzutowanie, 388, 476, 892
w dół, 346, 347, 348, 367, 371
w górę, 345, 347, 348, 367, 414, 734

S

sekwencja, 694
formatująca, 117
semantic zoom, *Patrz:* kontrolka zoom
semantyczny
separacja zagadnień, 272, 316, 589, 786
serializacja, 472, 473, 474, 475, 476,
485, 545, 567, 576, 581, 650, 651
klasy, 476, 577
kontraktu danych, 576, 578, 581,
582
wyjątek, 614
siatka, 55, 58, 534, 777
kolumna, 59, 60, 115, 534
534, 537
system, 536, 537
wiersz, 58, 115
wysokość, 60, 535, 537
wymiały, 60

sideloading, *Patrz:* ładowanie
bezpośrednie
Sklep Windows, 90, 96, 129, 517, 519,
528, 530, 536, 546, 552, 567, 568,
570, 578, 579, 580, 684, 686, 748,
770, 874
słownik, 421, 422, 423
długość, 413
element, 413
pobranie listy kluczy, 413
values, 685
wyszukiwanie na podstawie klucza,
413
słowo kluczowe, 190, 211, 221
abstract, 359
as, 343, 346, 347, 669
await, *Patrz:* operator await
base, 310
break, 878
by, 707
case, 469, 470
continue, 878
event, 734
EventHandler, 734, 737, 744
group, 707
interface, 333
internal, 351
is, 340, 343, 347, 668
join, 705, 707, 708
new, 150, 157, 307, 311, 338, 402,
691
null, 210, 392, 665
object, 660
on ... equals, 707
override, 298, 304, 307, 308, 309,
736, 752
partial, 107, 123
private, 258, 351, 353, 354, 768
protected, 351, 352, 353, 354
public, 228, 230, 258, 351, 354,
Patrz też: klasa public
readonly, 823
ref, 663
sealed, 351
select new, 707
static, 157, 159
struct, 655
this, 209, 211, 266, 269, 352, 527,
653, 670, 737

typeof, 687
using, 76, 102, 135, 415, 462, 631,
633, 648, 829
var, 211, 682, 708
virtual, 298, 304, 308, 309
splash screen, *Patrz:* ekran tytułowy
sprajt, 823
sprite, *Patrz:* sprajt
SQL, 685
Stack, *Patrz:* stos
stała, 240, 242
Environment.NewLine, 431
stan, 787, 790, 792
sterta, 160, 196, 659, 668
stos, 435, 437, 659, 661, 668
wywołań, 612
strona
Basic Page, 55, 818, 846
szablon, 55, 846
główna, *Patrz:* Basic Page
nagłówek, 65
nawigacja, 686, 687
układ, 71, 534
struktura, 655, 656, 657, 658, 668, 669,
890
Nullable, 665
Point, 827
Rect, 827
Size, 827
strumień, 442, 443, 450, 476, 632, 656
plików, 450, 456, 487
sieciowy, 443, 450
StreamReader, 449
StreamWriter, 446, 457
zamykanie, 462
system
dwójkowy, 19, 480, *Patrz też:*
format binary
Linux, 885
plików, 462, 578
siatki, *Patrz:* siatka system
szesnastkowy, 19, 480, 487, 488,
Patrz też: widok szesnastkowy
szablon
Basic Page, 55, 846
danych, 554
Grid App, 727
instrukcji, 113
Items Page, 727

kontrolek, 27, 67, 88, 773
 Split App, 720, 723
 Split Page, 727

T

tablica, 205, 207, 391, 395, 396, 401, 681
 args, 490
 bajtów, 482
 długość, 206, 392
 metod wirtualnych, 303
 zmiennych referencyjnych, 206, 318, 338
 znaków, 480
 typ, 211, 682, 889
 anonimowy, 691, 708, 898, 899
 bool, 182, 184, 656
 byte, 182, 211, 388
 char, 183, 184, 481
 DateTime, 546, 548, 657, 890
 decimal, 183, 184, 186, 243, 244, 656
 double, 182, 183, 187, 663, 890
 enum, *Patrz:* typ wyliczeniowy
 EventHandler, *Patrz:* słowo kluczowe EventHandler
 float, 183, 184, 187
 generyczny, 398
 int, 182, 186, 388, 411, 656, 890
 long, 182, 211, 388
 Nullable<DateTime>, 665
 object, 183
 referencyjny, 657, 660, 668
 rzutowanie, 186, 187, 188
 sbyte, 182
 short, 182
 string, 182, 184, 188, 481
 rozszerzanie, 672
 uint, 182
 ulong, 182
 ushort, 182
 void, 146, 737
 wartościowy, 211, 401, 411, 656, 657, 658, 660, 668
 ze znakiem ?, 665
 wyliczeniowy, 387, 388, 390, 401, 411, 431, 702
 wynikowy, 146, 146, 265, 266

U

user control, *Patrz:* kontrolka użytkownika

V

Visual Designer, 45
 Visual Studio Express, 38, 49
 Visual Studio for Windows Desktop, *Patrz:* Windows Desktop
 Visual Studio for Windows Phone, *Patrz:* Windows Phone
 Visual Studio IDE, 44, 46, 47, 48, 73, 90, 96, 97, 98, 100, 104, 175, 527, 687, 897
 Visual Studio Integrated Development Environment, *Patrz:* Visual Studio IDE
 Visual Studio Remote Debugger, *Patrz:* Remote Debugger
 vtable, *Patrz:* tablica metod wirtualnych

W

warstwa
 modelu, 790
 widoku, 868
 widoku, 790, 797
 wątek, 888
 wiązanie danych, 542, 543, 546, 551, 552, 554, 557, 574, 721, 776, 820
 dwukierunkowe, 543, 547
 widok, 777, 786, 809
 szesnastkowy, 487
 wielokąt, 825
 wiersz poleceń, 306, 885
 Windows 8 Camp Training Kit, 874
 Windows App Certification Kit, 90
 Windows Desktop, 129, 141, 882
 Windows Forms Application, 99, 130, 182, 262, 303, 304, 524, 545
 Windows Phone, 29, 859, 861
 Windows Presentation Foundation, *Patrz:* WPF
 Windows Runtime, 99
 WinForm, *Patrz:* Windows Forms Application
 właściwość, 64, 96, 261, 267, 333, 340, 351, 354
 automatyczna, 263, 372
 tylko do odczytu, 263, 264
 BackgroundImage, 498

Content, 544, 545
 Controls, 524
 docelowa, 542, 543
 dołączona, 820
 domyślna, 545
 e.Handled, 757
 FileName, 455
 Filter, 454, 455, 459
 FlowDirection, 459
 InitialDirectory, 459
 ItemsSource, 821
 nazwa, 897
 przesłanianie, 354
 publiczna, 269
 sygnatura, 267
 Text, 545
 Title, 459
 x:Key, 555
 x:Name, 79, 547, 552, 555
 xmlns:, 552
 local, 552
 źródłowa, 542, 543

WPF, 38, 53, 55, 902
 wyjątek, 23, 599, 604, 605, 607, 608, 616, 620, 622, 626, 634, 635, 653, 843
 FormatException, 663
 IndexOutOfRangeException, 604
 InvalidOperationException, 888
 nieobsłużony, 612, 622, 625
 NotImplementedException, 746, 798
 NullReferenceException, 767
 SerializationException, 614, 620
 wyrażenie lambda, 768, 898, 899
 wywołanie Application.DoEvents, 140
 wzorzec, 153, 154
 metody wytwórczej, 822
 Model-View-ViewModel, *Patrz:* MVVM
 model-widok-kontroler, *Patrz:* MVC
 obserwatora, 768
 projektowy, 768

X

XAML, 21, 45, 49, 53, 517, 528, 538
 kod znacznikowy, *Patrz:* kod znacznikowy
 siatka, *Patrz:* siatka
 XML, 49, 900

Z

- zablokowanie, 568
- zapytanie, 681, 682, 683, 684, 705, 900
 - edycja, 718
 - from, 685, 698
 - orderby, 685, 698
 - select, 698, 701
 - where, 685, 698
- zasoby
 - alokacja, 461
 - niezarządzane, 646
 - statyczne, 552, 553, 555, 573, 716, 803
 - zarządzane, 646
 - zwalnianie, 461
- zdarzenie, 218, 745, 768, 792
 - bez procedur obsługi, 736
 - Click, 746, 793
 - CollectionChanged, 797
 - Completed, 867
 - dotknięcia ekranu, 848
 - funkcja zwrotna, *Patrz:* funkcja zwrotna
 - generowanie, 731, 732
 - kiwnięcia, 848
 - klawiatury, 848
 - nadawca, 758
 - nasłuchiwanie, 731, 732
 - NavigatedFrom, 750
 - obsługa, *Patrz:* procedura obsługi zdarzeń
 - odbiorca, 758
 - parametr, *Patrz:* delegat
- propagacja, 752, 757
- PropertyChanged, 556, 797
- prywatne, 768
- publiczne, 734, 763
- SizeChanged, 824, 833
- subskrypcja, 731, 733, 737, 763
- sygnatura, 737
- Tick, 220
- trasowane, 752, 753, 755, 757
- typ, 744
- zarządzające cyklem życia procesu, 748
- zegara, 800, 843, 852, 854
- zgłoszenie, 556
- zestaw, *Patrz:* złożenie
- złożenie, 351, 882, 883
 - biblioteczne, 882
 - wykonywalne, 882
- zmienna, 108, 140, 172, 174, 184, 186, 195, 211
 - definicja, 461
 - logiczna, 110, 113
 - lokalna, 659
 - nazwa, 108, 113, 117, 269, 897
 - private, *Patrz:* słowo kluczowe private
 - protected, *Patrz:* słowo kluczowe protected
 - public, 251, 252, *Patrz też:* klasa public, słowo kluczowe public
 - referencyjna, 196, 205, 307, 338, 401, 474, *Patrz też:* referencja
 - grupa, 206
 - typ, 108, 117, 682
 - wartość, 109, 112
- znacznik, 49
 - kolejności bajtów, 492
 - kontenera, 49
 - otwierający, 49
 - VisualState, 806
 - zamykający, 49
- znak, *Patrz też:* operator
 - ‘, 183
 - *, 534, 537
 - */, 111
 - /, 535
 - /*, 111
 - //, 102
 - ;, 294, 334, 337, 348
 - ?, 665
 - ? :, 794
 - @, 446, 457
 - [], 104, 894
 - \\, 457
 - \n, 117, 183, 431, 457
 - \r\n, 431, 457
 - \t, 183, 457
 - _, 779
 - { }, 113, 123
 - + =, 735, 737, 738, 739, 766, 767
 - apostrof, *Patrz:* znak ‘
 - dwukropek, *Patrz:* znak :
 - Unicode, 480, 481, 492

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

- 
1. ZAREJESTRUJ SIĘ
 2. PREZENTUJ KSIĄŻKI
 3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

C#. Rusz głową!

C# to odpowiedź firmy Microsoft na odnoszący sukcesy język Java. Za pomocą C# możesz pisać przenośny kod, który Twoi klienci uruchomią w dowolnym systemie. Jest tylko jeden warunek — muszą mieć dostęp do środowiska uruchomieniowego .NET Framework, Mono lub DotGNU. Innymi słowy, C# spełnił marzenia programistów — raz stworzony kod można uruchomić bez dodatkowych nakładów na różnych platformach.

Najnowsze wydanie tej książki, należącej do cenionej serii „Rusz głową!”, zostało zaktualizowane, poprawione oraz rozszerzone o dodatkowy projekt na platformę Windows Phone. Gdy pochłoniesz te kilkaset stron, zaczniesz swobodnie korzystać z języka C#, jego konstrukcji i możliwości. Książka charakteryzuje się doskonałą przejrzystością oraz przystępnie przedstawioną wiedzą. Znajdziesz tu elementy programowania obiektowego, operacje na plikach, obsługę wyjątków oraz pracę z wieloma wątkami — to tylko niektóre z poruszanych zagadnień. Ponadto przekonasz się, jak sprawnie stworzyć atrakcyjny interfejs użytkownika, oraz zrozumiesz, do czego służy język LINQ. Jest to wyśmienita pozycja dla wszystkich czytelników chcących rozpocząć przygodę z językiem C# oraz platformą .NET.

Poznaj możliwości języka C#!

Dzięki tej książce:

- ▼ przygotujesz swoje środowisko pracy
- ▼ poznasz składnię oraz konstrukcję języka C#
- ▼ zaznajomisz się z elementami programowania obiektowego
- ▼ wykorzystasz możliwości języka LINQ
- ▼ stworzysz projekt dla Windows Phone
- ▼ wykorzystasz potencjał platformy .NET

helion.pl
księgarnia internetowa



Helion

Nr katalogowy: 20272



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900

Sprawdź najnowsze promocje:

👉 <http://helion.pl/promocje>

Książki najchętniej czytane:

👉 <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

👉 <http://helion.pl/novosci>

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel.: 32 230 98 63

e-mail: helion@helion.pl

<http://helion.pl>

sięgnij po WIECEJ



KOD KORZYŚCI

ISBN 978-83-246-8311-6



9 788324 683116

Cena 99,00 zł

Informatyka w najlepszym wydaniu

