

>>> VISUAL C#

DLA ZUPEŁNIE POCZĄTKUJĄCYCH

WYDANIE IV



TONY GADDIS

Tytuł oryginału: Starting Out with Visual C#, 4th Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-4684-0

Authorized translation from the English language edition, entitled: STARTING OUT WITH VISUAL C#, Fourth Edition; ISBN 0134382609; by Tony Gaddis; published by Pearson Education, Inc. Copyright © 2017, 2014, 2012 by Pearson Education, Inc. or its affiliates

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

Polish language edition published by HELION S.A. Copyright © 2019.

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

Helion SA

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/viczp2.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/viczp2>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

Wstęp 11

Uwaga, czytelnicy 19

Rozdział 1. Wstępne informacje na temat komputerów i programowania 21

1.1.	Wstęp	21
1.2.	Sprzęt i oprogramowanie	22
1.3.	W jaki sposób komputer przechowuje dane	28
1.4.	W jaki sposób działa program	33
1.5.	Graficzny interfejs użytkownika	42
1.6.	Obiekty	45
1.7.	Proces tworzenia programu	48
1.8.	Rozpoczęcie pracy ze środowiskiem Visual Studio	53
PRZYKŁAD 1.1.	Uruchomienie Visual Studio i konfiguracja środowiska	54
PRZYKŁAD 1.2.	Tworzenie nowego projektu Visual C#	57
PRZYKŁAD 1.3.	Zapisanie i zamknięcie projektu	59
PRZYKŁAD 1.4.	Otwarcie istniejącego projektu	68
PRZYKŁAD 1.5.	Poznanie środowiska Visual Studio	70
<i>Ważne pojęcia 71 • Pytania kontrolne 72 • Ćwiczenia 79</i>		

Rozdział 2. Wprowadzenie do Visual C# 81

2.1.	Rozpoczęcie pracy z formularzami i kontrolkami	81
2.2.	Utworzenie graficznego interfejsu użytkownika w pierwszej aplikacji Visual C# — Hello World	93
PRZYKŁAD 2.1.	Utworzenie graficznego interfejsu użytkownika aplikacji typu Witaj, świecie!	93

2.3.	Wprowadzenie do kodu w języku C#	97
2.4.	Utworzenie kodu aplikacji Hello World	110
PRZYKŁAD 2.2.	Utworzenie kodu aplikacji Hello World	110
2.5.	Kontrolka Label	113
PRZYKŁAD 2.3.	Utworzenie aplikacji Language Translator	123
2.6.	Poznanie listy IntelliSense	126
2.7.	Kontrolka PictureBox	127
PRZYKŁAD 2.4.	Utworzenie aplikacji Flags	132
PRZYKŁAD 2.5.	Utworzenie aplikacji Card Flip	137
2.8.	Komentarze, puste linie i wcięcia	141
2.9.	Utworzenie kodu odpowiedzialnego za zamknięcie formularza aplikacji	144
2.10.	Usuwanie błędów składni	145
	<i>Ważne pojęcia 146 • Pytania kontrolne 147 • Ćwiczenia programistyczne 153</i>	

Rozdział 3. **Przetwarzanie danych 157**

3.1.	Odczyt danych wejściowych za pomocą kontrolki TextBox	157
3.2.	Pierwszy kontakt ze zmiennymi	160
PRZYKŁAD 3.1.	Aplikacja Birth Date String	168
3.3.	Zmienne i liczbowe typy danych	174
3.4.	Przeprowadzanie obliczeń	180
3.5.	Wprowadzanie i generowanie danych liczbowych	186
PRZYKŁAD 3.2.	Obliczenie zużycia paliwa	192
3.6.	Formatowanie liczb za pomocą metody ToString()	196
PRZYKŁAD 3.3.	Utworzenie aplikacji Sale Price Calculator wykorzystującej formatowanie wartości walutowych	199
3.7.	Prosta obsługa wyjątków	204
PRZYKŁAD 3.4.	Utworzenie aplikacji Test Average wraz z obsługą wyjątków	209
3.8.	Używanie stałych nazwanych	214
3.9.	Deklarowanie zmiennych jako pól	215
PRZYKŁAD 3.5.	Utworzenie aplikacji Change Counter	220
3.10.	Używanie klasy Math	225
3.11.	Więcej informacji na temat graficznego interfejsu użytkownika	227
3.12.	Używanie debuggera do wyszukiwania błędów logicznych	238
PRZYKŁAD 3.6.	Pojedyncze wykonywanie poleceń w kodzie aplikacji	239
	<i>Ważne pojęcia 244 • Pytania kontrolne 244 • Ćwiczenia programistyczne 249</i>	

Rozdział 4. **Podejmowanie decyzji** 255

4.1.	Konstrukcje warunkowe i polecenie if	255
PRZYKŁAD 4.1.	Dokończenie aplikacji Test Average	261
4.2.	Konstrukcja if-else	266
PRZYKŁAD 4.2.	Dokończenie aplikacji Payroll with Overtime	268
4.3.	Zagnieżdżone konstrukcje warunkowe	273
PRZYKŁAD 4.3.	Dokończenie aplikacji Loan Qualifier	276
4.4.	Operatory logiczne	286
4.5.	Zmienne boolowskie i flagi	292
4.6.	Porównywanie ciągów tekstowych	293
4.7.	Używanie metod TryParse() do unikania wyjątków podczas konwersji danych	298
PRZYKŁAD 4.4.	Obliczenie zużycia paliwa	303
4.8.	Weryfikacja danych wejściowych	307
4.9.	Przyciski opcji i pola wyboru	309
PRZYKŁAD 4.5.	Utworzenie aplikacji Color Theme	315
4.10.	Konstrukcja switch	318
4.11.	Wprowadzenie do kontrolki ListBox	321
PRZYKŁAD 4.6.	Utworzenie aplikacji Time Zone	324
<i>Ważne pojęcia 327 • Pytania kontrolne 328 • Ćwiczenia programistyczne 333</i>		

Rozdział 5. **Pętle, pliki i liczby losowe** 339

5.1.	Więcej na temat kontrolki ListBox	339
5.2.	Pętla while	342
PRZYKŁAD 5.1.	Użycie pętli do obliczenia wysokości salda	346
PRZYKŁAD 5.2.	Usprawnienie aplikacji Ending Balance	350
5.3.	Operatory ++ i --	354
5.4.	Pętla for	356
PRZYKŁAD 5.3.	Użycie pętli for	362
5.5.	Pętla do-while	366
5.6.	Użycie plików do przechowywania danych	367
PRZYKŁAD 5.4.	Zapis danych w pliku tekstowym	375
PRZYKŁAD 5.5.	Dołączenie danych do pliku Friend.txt	381
PRZYKŁAD 5.6.	Użycie pętli do odczytania całej zawartości pliku	390
PRZYKŁAD 5.7.	Obliczenie sumy bieżącej	395
5.7.	Kontrolki OpenFileDialog i SaveFileDialog	399
5.8.	Liczby losowe	405
PRZYKŁAD 5.8.	Symulacja rzutu monetą	407

5.9.	Zdarzenie Load	412
PRZYKŁAD 5.9.	Utworzenie procedury obsługi zdarzeń Load	413
<i>Ważne pojęcia 416 • Pytania kontrolne 417 • Ćwiczenia programistyczne 420</i>		

Rozdział 6. **Modularyzacja kodu za pomocą metod** 425

6.1.	Wprowadzenie do metod	425
6.2.	Metoda typu void	427
PRZYKŁAD 6.1.	Tworzenie i wywoływanie metod	432
6.3.	Przekazywanie argumentów metodzie	437
PRZYKŁAD 6.2.	Przekazanie argumentu metodzie	440
6.4.	Przekazywanie argumentów przez referencję	449
PRZYKŁAD 6.3.	Użycie parametru danych wyjściowych	453
6.5.	Metody zwracające wartość	458
PRZYKŁAD 6.4.	Utworzenie metody zwracającej wartość	463
PRZYKŁAD 6.5.	Modularyzacja weryfikacji danych wejściowych za pomocą metody boolowskiej	468
6.6.	Debugowanie metod	473
PRZYKŁAD 6.6.	Praca z poleceniem Step Into	474
PRZYKŁAD 6.7.	Praca z poleceniem Step Over	475
PRZYKŁAD 6.8.	Praca z poleceniem Step Out	477
<i>Ważne pojęcia 478 • Pytania kontrolne 478 • Ćwiczenia programistyczne 482</i>		

Rozdział 7. **Tablice i listy** 487

7.1.	Typy przekazywane przez wartość i referencję	487
7.2.	Ogólne informacje o tablicy	491
PRZYKŁAD 7.1.	Użycie tablicy do przechowywania liczb losowych	499
7.3.	Praca z plikami i tablicami	505
7.4.	Przekazywanie tablicy jako argumentu metody	509
7.5.	Wybrane użyteczne algorytmy tablic	516
PRZYKŁAD 7.2.	Przetwarzanie tablicy	528
7.6.	Zaawansowane algorytmy sortowania i przeszukiwania tablic	534
7.7.	Tablica dwuwymiarowa	543
PRZYKŁAD 7.3.	Dokończenie aplikacji Seating Chart	547
7.8.	Tablica tablic	553
7.9.	Kolekcja List	555
PRZYKŁAD 7.4.	Dokończenie aplikacji Test Score List	562
<i>Ważne pojęcia 568 • Pytania kontrolne 568 • Ćwiczenia programistyczne 572</i>		

Rozdział 8. Więcej informacji o przetwarzaniu danych 577

8.1.	Wprowadzenie	577
8.2.	Przetwarzanie znaków i ciągów tekstowych	577
PRZYKŁAD 8.1.	Dokończenie aplikacji Password Validation	584
PRZYKŁAD 8.2.	Dokończenie aplikacji Telephone Format	599
PRZYKŁAD 8.3.	Dokończenie aplikacji Telephone Unformat	604
PRZYKŁAD 8.4.	Dokończenie aplikacji CSV Reader	612
8.3.	Struktury	618
PRZYKŁAD 8.5.	Dokończenie aplikacji Phonebook	629
8.4.	Typy wyliczeniowe	636
PRZYKŁAD 8.6.	Dokończenie aplikacji Color Spectrum	639
8.5.	Kontrolka ImageList	645
PRZYKŁAD 8.7.	Dokończenie aplikacji Random Card	647
<i>Ważne pojęcia 650 • Pytania kontrolne 650 • Ćwiczenia programistyczne 654</i>		

Rozdział 9. Klasy i projekty złożone z wielu formularzy 659

9.1.	Wprowadzenie do klas	659
PRZYKŁAD 9.1.	Utworzenie i użycie klasy Coin	667
9.2.	Właściwości	673
PRZYKŁAD 9.2.	Utworzenie i użycie klasy CellPhone	677
9.3.	Parametryzowane konstruktory i przeciążanie	685
PRZYKŁAD 9.3.	Utworzenie i użycie klasy BankAccount	685
9.4.	Przechowywanie w tablicy i w kontenerze List obiektu typu klasy	693
PRZYKŁAD 9.4.	Dokończenie aplikacji Cell Phone Inventory	695
9.5.	Wyszukiwanie klas i ich zadania w problemie	699
9.6.	Tworzenie wielu formularzy w projekcie	709
PRZYKŁAD 9.5.	Utworzenie aplikacji zawierającej dwa formularze	715
PRZYKŁAD 9.6.	Uzyskanie dostępu do kontrolki w innym formularzu	721
<i>Ważne pojęcia 726 • Pytania kontrolne 726 • Ćwiczenia programistyczne 730</i>		

Rozdział 10. Dziedziczenie i polimorfizm 735

10.1.	Dziedziczenie	735
PRZYKŁAD 10.1.	Utworzenie i przetestowanie klas SavingsAccount i CDAccount	746
10.2.	Polimorfizm	755
PRZYKŁAD 10.2.	Dokończenie aplikacji Polymorphism	761

10.3. Klasa abstrakcyjna767

PRZYKŁAD 10.3. Dokończenie aplikacji Computer Science Student 769

Ważne pojęcia 774 • Pytania kontrolne 774 • Ćwiczenia programistyczne 778

Rozdział 11. **Bazy danych 781**

11.1. Wprowadzenie do systemu zarządzania bazą danych781

11.2. Tabele, rekordy i kolumny783

11.3. Utworzenie bazy danych w Visual Studio787

PRZYKŁAD 11.1. Rozpoczęcie pracy nad aplikacją Phone Book
i utworzenie bazy danych Phonelist.mdf 788

11.4. Kontrolka DataGridView798

PRZYKŁAD 11.2. Dokończenie aplikacji Phone Book 798

11.5. Nawiązanie połączenia z istniejącą bazą danych
i użycie kontrolki widoku szczegółowego806

PRZYKŁAD 11.3. Utworzenie aplikacji Products wraz z widokiem
szczegółowym 807

11.6. Więcej informacji na temat kontrolki dołączania danych816

PRZYKŁAD 11.4. Utworzenie aplikacji Product Lookup 820

PRZYKŁAD 11.5. Utworzenie aplikacji Multiform Products 824

11.7. Pobieranie danych za pomocą polecenia SQL Select830

PRZYKŁAD 11.6. Utworzenie aplikacji Product Queries 838

PRZYKŁAD 11.7. Dokończenie aplikacji Product Queries 848

PRZYKŁAD 11.8. Utworzenie aplikacji Product Search 852

Ważne pojęcia 858 • Pytania kontrolne 858 • Ćwiczenia programistyczne 863

Dodatek A **Podstawowe typy danych C#**867

Dodatek B **Dodatkowe kontrolki interfejsu użytkownika**869

Dodatek C **Tablica znaków ASCII**891

Dodatek D **Odpowiedzi do pytań z punktów kontrolnych**893

Skorowidz915

TEMATYKA

- | | |
|--|--|
| 2.1. Rozpoczęcie pracy z formularzami i kontrolkami | 2.5. Kontrolka Label |
| 2.2. Utworzenie graficznego interfejsu użytkownika w pierwszej aplikacji Visual C# — Hello World | 2.6. Poznanie listy IntelliSense |
| 2.3. Wprowadzenie do kodu w języku C# | 2.7. Kontrolka PictureBox |
| 2.4. Utworzenie kodu aplikacji Hello World | 2.8. Komentarze, puste linie i wcięcia |
| | 2.9. Utworzenie kodu odpowiedzialnego za zamknięcie formularza aplikacji |
| | 2.10. Usuwanie błędów składni |

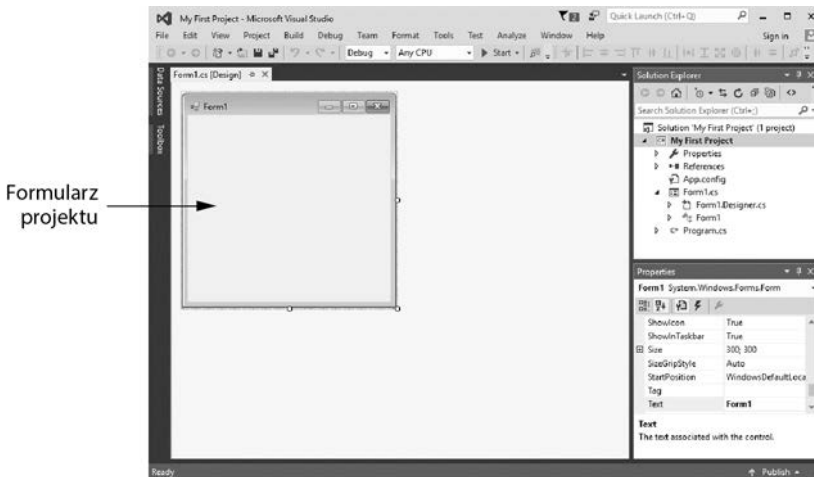
2.1. Rozpoczęcie pracy z formularzami i kontrolkami

WYJAŚNIENIE. Pierwszym krokiem podczas tworzenia aplikacji Visual C# jest przygotowanie graficznego interfejsu użytkownika. Oferowane przez Visual Studio okna *Designer*, *Toolbox* i *Properties* można wykorzystać do przygotowania formularza aplikacji wraz z żądanymi kontrolkami i zdefiniowanymi dla nich odpowiednimi właściwościami.

W tym rozdziale utworzysz pierwszą aplikację Visual C#. Jednak zanim przystąpisz do pracy, musisz poznać pewne podstawowe koncepcje dotyczące przygotowania graficznego interfejsu użytkownika w Visual Studio. W tym podrozdziale przedstawię podstawy z zakresu edycji formularzy i tworzenia kontroltek.

Formularz aplikacji

Gdy utworzysz nowy projekt Visual C#, Visual Studio automatycznie dodaje pusty formularz i wyświetla go w oknie *Designer*. Przykład pokazałem na rysunku 2.1. Ten pusty formularz możesz potraktować jako płótno, na którym zostanie umieszczony interfejs użytkownika aplikacji. Do formularza dodajesz kontrolki, zmieniasz jego wielkość i wiele innych parametrów. Po uruchomieniu aplikacji formularz zostanie wyświetlony na ekranie.



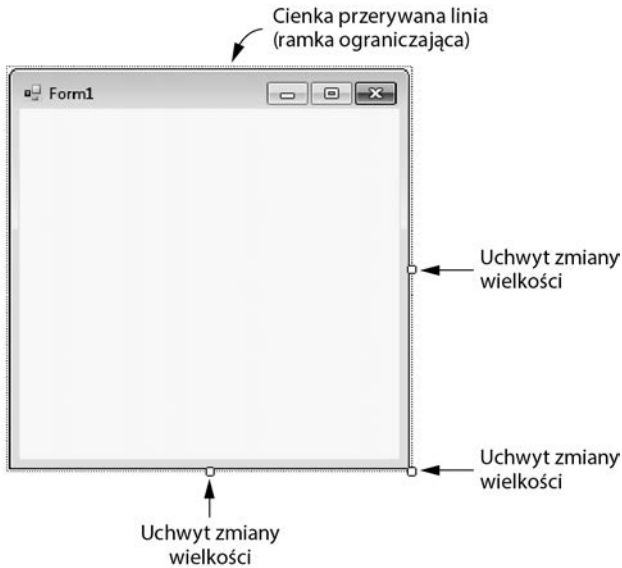
Rysunek 2.1. Nowy projekt wraz z pustym formularzem wyświetlonym w oknie Designer

Jeżeli przyjrzyj się bliżej formularzowi, zauważysz, że jest otoczony cienką przerywaną linią nazywaną **ramką ograniczającą** (ang. *bounding box*). Jak pokazałem na rysunku 2.2, ramka ograniczająca ma małe **uchwyty zmiany wielkości** wyświetlane po prawej stronie i na dole formularza oraz w jego prawym dolnym rogu. Gdy ramka ograniczająca jest wyświetlana wokół obiektu w oknie *Designer*, oznacza to, że dany obiekt został zaznaczony i jest gotowy do edycji.

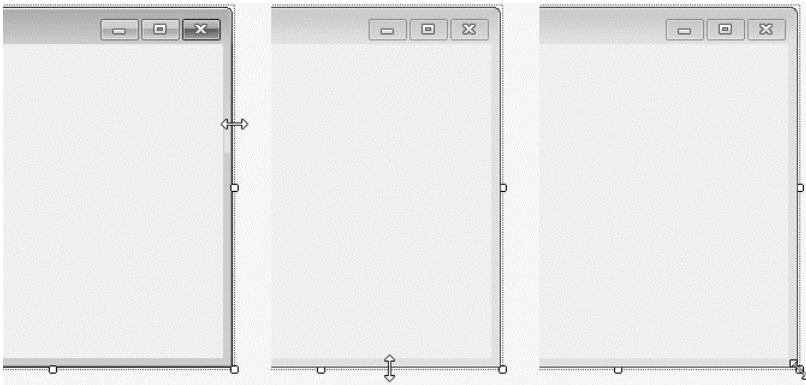
Początkowo wielkość formularza wynosi 300×300 pikseli i możesz ją łatwo zmienić za pomocą myszy. Po umieszczeniu kursora myszy nad dowolną krawędzią lub rogiem zawierającym uchwyt zmiany wielkości kursor zmieni kształt na dwie połączone ze sobą strzałki ($\leftrightarrow$). Przykłady pokazałem na rysunku 2.3. Po zmianie kursora myszy możesz kliknąć i przeciągnąć myszą, co spowoduje zmianę wielkości formularza.

Identyfikowanie formularzy i kontroltek za pomocą ich nazw

Graficzny interfejs użytkownika aplikacji składa się z formularzy i różnych kontroltek. Każdy formularz i kontrolka w graficznym interfejsie użytkownika aplikacji musi mieć identyfikującą ją nazwę. Pusty formularz początkowo utworzony przez Visual Studio w nowym projekcie nosi nazwę `Form1`.



Rysunek 2.2. Ramka ograniczająca i uchwyty zmiany wielkości



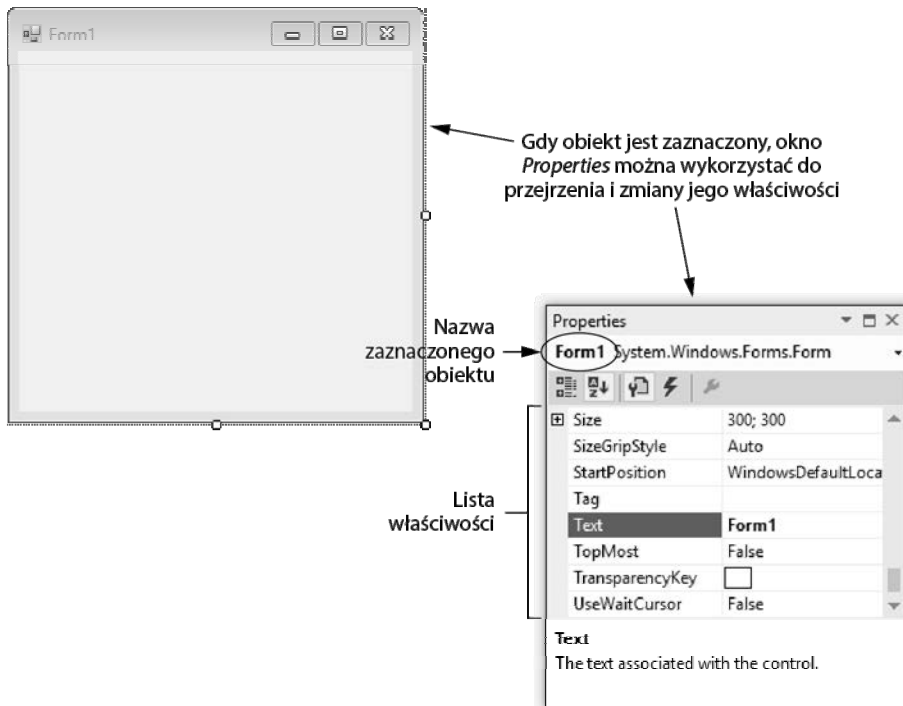
Rysunek 2.3. Użycie myszy do zmiany wielkości formularza



UWAGA. Z dalszej części książki dowiesz się, jak można zmienić nazwę formularza. W omawianym tutaj przykładzie pozostawimy nazwę domyślną, czyli Form1.

Okno Properties

Wygląd i inne cechy charakterystyczne obiektu graficznego interfejsu użytkownika są określane przez właściwości danego obiektu. Gdy zaznaczysz obiekt w oknie *Designer*, właściwości tego obiektu zostaną wyświetlone w oknie *Properties*. Przykładowo: po zaznaczeniu formularza Form1 jego właściwości będą wyświetlone w oknie *Properties*, jak pokazałem na rysunku 2.4.



Rysunek 2.4. Okno *Properties* wyświetlające właściwości zaznaczonego obiektu

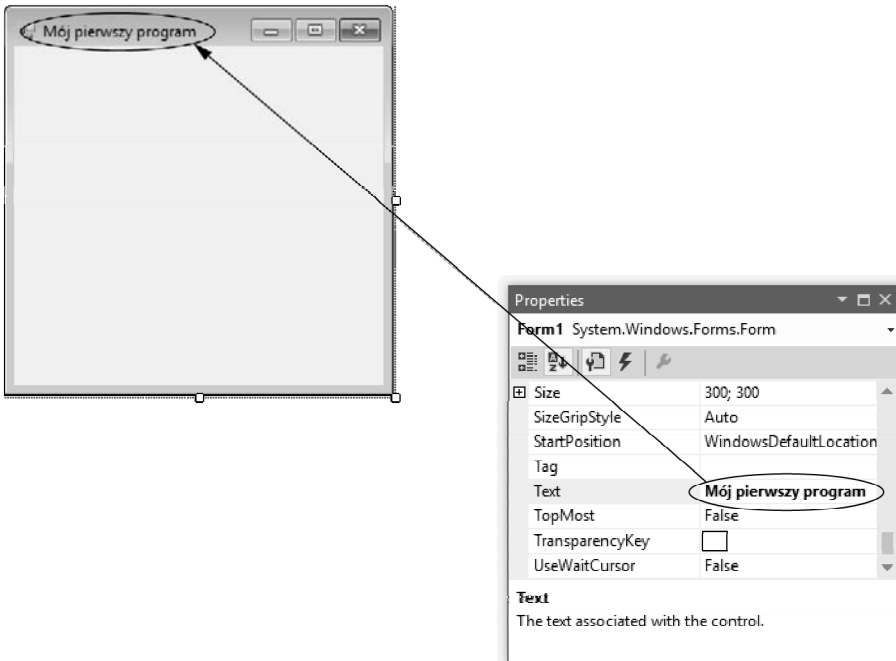


WSKAZÓWKA. Przypomnij sobie z rozdziału 1., że jeśli okno *Properties* ma włączoną funkcję automatycznego ukrywania, można kliknąć jego kartę, aby je wyświetlić. Jeżeli nie widzisz na ekranie okna *Properties*, przejdź do menu *View*, a następnie wybierz z niego opcję *Properties Window*.

Na górze okna *Properties* jest wyświetlana nazwa aktualnie zaznaczonego obiektu. Jak możesz zobaczyć na rysunku 2.4, w omawianym przykładzie tym obiektem jest *Form1*. Poniżej znajduje się przewijana lista właściwości. Ta lista składa się z dwóch kolumn: lewej zawierającej nazwę właściwości i prawej, na której widać jej wartość. Przykładowo: spójrz na właściwość *Size* widoczną na rysunku 2.4 — jej wartością jest 300; 300. Oznacza to, że wielkość formularza wynosi 300 × 300 pikseli. Następnie spójrz na właściwość *Text* określającą tekst wyświetlany w pasku tytułowym formularza (ten pasek znajduje się na górze formularza). Aktualną wartością tej właściwości jest *Form1*, więc pasek tytułowy formularza wyświetla po prostu *Form1*.

Po utworzeniu formularza jego właściwość *Text* ma początkowo taką samą wartość jak nazwa formularza. Gdy utworzysz nowy projekt, pusty formularz umieszczany w oknie *Designer* zawsze będzie miał nazwę *Form1*, stąd tekst *Form1* wyświetlany w pasku tytułu formularza. W większości przypadków będziesz zmieniał wartość właściwości *Text* formularza na bardziej znaczącą. Przyjmuję założenie, że aktualnie zaznaczyłeś formularz *Form1*. Aby zmienić wartość jego właściwości *Text* na *Mój pierwszy program*, należy wykonać przedstawione tutaj kroki.

- Krok 1.** W oknie *Properties* odszukaj właściwość *Text*.
- Krok 2.** Dwukrotnie kliknij słowo *Form1* aktualnie będące wartością właściwości *Text*, a następnie usuń je, naciskając klawisz *Delete*.
- Krok 3.** Wpisz *Mój pierwszy program* i naciśnij klawisz *Enter*. Wpisany tekst zostanie wyświetlony w pasku tytułu formularza, jak pokazałem na rysunku 2.5.



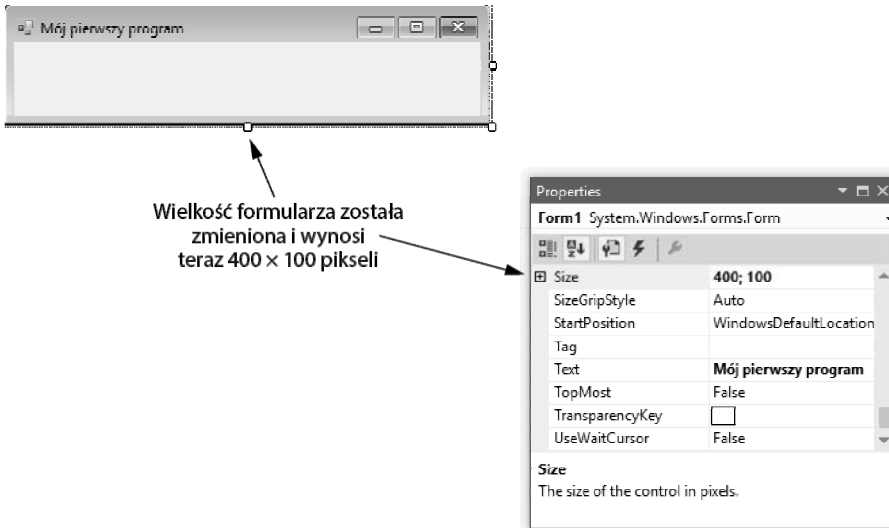
Rysunek 2.5. Wartość właściwości *Text* formularza wyświetlona w pasku tytułowym tego formularza



UWAGA. Zmiana właściwości *Text* obiektu nie powoduje zmiany jego nazwy. Dlatego też po zmianie wartości właściwości *Text* formularza *Form1* na *Mój pierwszy program* nazwą formularza wciąż będzie *Form1*. Zmienił się jedynie tekst wyświetlany w pasku tytułu formularza.

Wcześniej z tego rozdziału dowiedziałeś się, jak użyć myszy do zmiany wielkości formularza w oknie *Designer*. Alternatywna metoda polega na wykorzystaniu właściwości *Size* w oknie *Properties*. Przykładowo: przyjmując założenie, że nadal jest zaznaczony formularz *Form1*. Zmiana jego wielkości na 400×100 pikseli wymaga wykonania wymienionych niżej kroków.

- Krok 1.** W oknie *Properties* odszukaj właściwość *Size*.
- Krok 2.** Kliknij obszar zawierający wartość właściwości *Size* i usuń ją.
- Krok 3.** Wpisz *400; 100* i naciśnij klawisz *Enter*. Wielkość formularza zmieni się, jak pokazałem na rysunku 2.6.



Rysunek 2.6. Wielkość formularza została zmieniona na 400 × 100 pikseli



UWAGA. Zwróć uwagę, że na rysunku 2.6 został zaznaczony przycisk **Alphabetical** (🔍) na górze okna *Properties*. To powoduje wyświetlenie właściwości w kolejności alfabetycznej. Ewentualnie można zaznaczyć przycisk **Categorized** (📁) powodujący grupowanie właściwości. Domyślnie zaznaczony jest przycisk *Alphabetical* i w większości przypadków znacznie ułatwia to odszukanie żądanej właściwości.

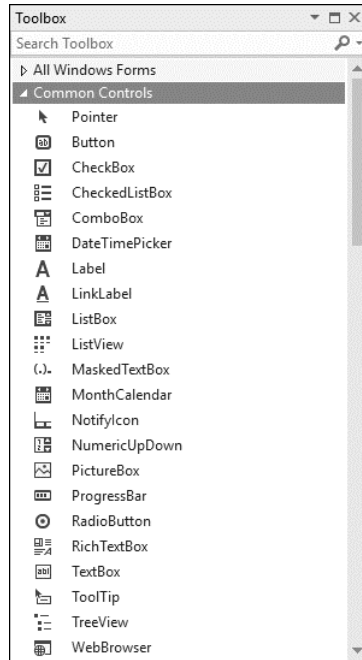
Dodawanie kontroltek do formularza

Gdy jesteś gotowy do utworzenia kontroltek w formularzu aplikacji, możesz skorzystać z okna *Toolbox*. Przypomnij sobie z rozdziału 1., że to okno jest zwykle wyświetlone po lewej stronie ekranu w środowisku Visual Studio. Jeżeli to okno ma włączoną funkcję automatycznego ukrywania, kliknij jego kartę, aby je wyświetlić. Na rysunku 2.7 pokazałem okno *Toolbox* w postaci standardowej po jego wyświetleniu.

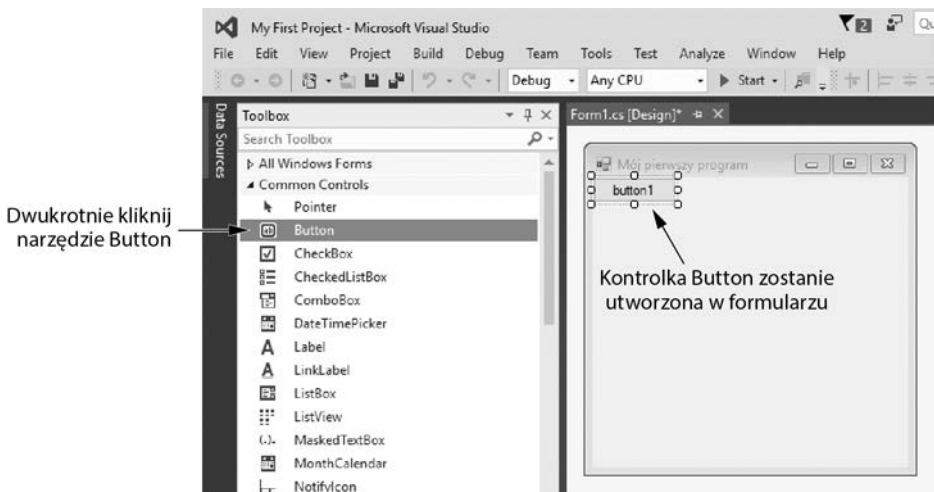


WSKAZÓWKA. Przypomnij sobie z rozdziału 1., że jeśli nie widzisz okna *Toolbox* na ekranie, należy z menu *View* wybrać opcję *Toolbox*.

Okno *Toolbox* zawiera przewijaną listę kontroltek, które można dodać do formularza. Aby dodać kontrolkę do formularza, po prostu odszukaj ją w oknie *Toolbox*, a następnie dwukrotnie kliknij myszą. Wybrana kontrolka zostanie utworzona w formularzu. Przykładowo: przyjmując założenie, że chcesz utworzyć kontrolkę *Button*. Odszukaj ją w oknie *Toolbox*, jak pokazałem na rysunku 2.8, dwukrotnie kliknij myszą, a pojawi się ona w formularzu.



Rysunek 2.7. Okno Toolbox



Rysunek 2.8. Utworzenie kontrolki Button



WSKAZÓWKA. Istnieje również możliwość kliknięcia i przeciągnięcia kontrolki z okna *Toolbox* na formularz.

Zmiana wielkości i przeniesienie kontrolki

Przyjrzyj się dobrze kontrolce Button pokazanej na rysunku 2.8. Zwróć uwagę na umieszczenie jej w ramce ograniczającej wraz z uchwytyami zmiany wielkości. To oznacza, że kontrolka jest zaznaczona. Gdy kontrolka jest zaznaczona, jej wielkość można zmienić myszą podobnie jak wcześniej mogłeś zmienić wielkość formularza. Za pomocą myszy kontrolkę można również przenieść do nowego położenia. Umieść kursor myszy w kontrolce, a gdy przyjmie on postać poczwórnej strzałki (↔↕), możesz kliknąć i przeciągnąć kontrolkę do nowego położenia. Na rysunku 2.9 pokazałem formularz, w którym kontrolka Button została powiększona i przeniesiona do innego położenia.



Rysunek 2.9. Zmiana wielkości i położenia kontrolki Button

Usunięcie kontrolki

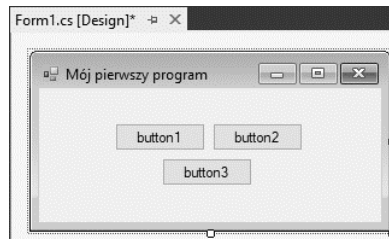
Usunięcie kontrolki jest proste — wystarczy ją zaznaczyć, a następnie nacisnąć klawisz *Delete*.

Więcej o kontrolce Button

Wcześniej z tego rozdziału dowiedziałeś się, że każdy formularz i kontrolka w graficznym interfejsie użytkownika aplikacji muszą mieć identyfikującą je nazwę. Po utworzeniu kontrolki Button automatycznie otrzymują one nazwy domyślne: `button1`, `button2` itd.

Kontrolka Button ma właściwość `Text` zawierającą tekst wyświetlany przez przycisk. Po utworzeniu kontrolki początkowa wartość właściwości `Text` jest taka sama jak nazwa kontrolki. Dlatego też ta nazwa będzie wyświetlona na przycisku. Przykładowo: formularz pokazany na rysunku 2.10 zawiera trzy kontrolki Button o nazwach `button1`, `button2` i `button3`.

Po utworzeniu kontrolki Button zawsze powinieneś zmienić wartość jej właściwości `Text`. Tekst wyświetlany przez przycisk powinien wskazywać jego przeznaczenie. Dlatego

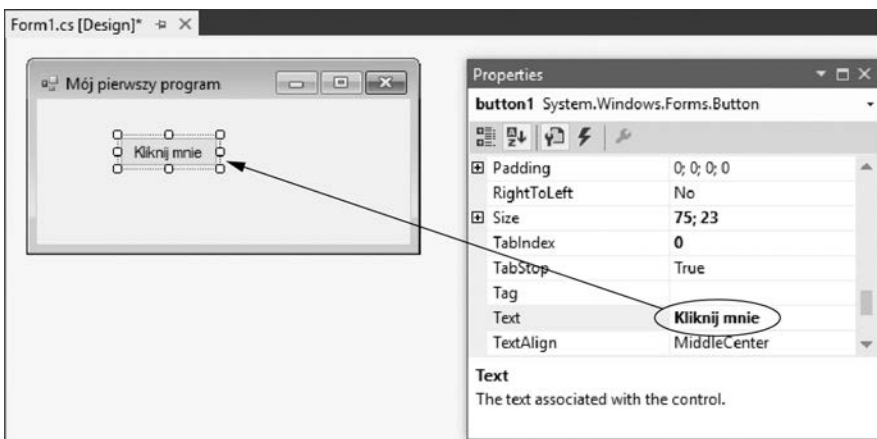


Rysunek 2.10. Formularz wraz z trzema kontrolkami Button

też przycisk, który po kliknięciu powoduje obliczenie wartości średniej, może być zatytułowany *Oblicz średnią*, natomiast wyświetlający raport może mieć tytuł *Wyświetl raport*. Oto kroki, jakie należy wykonać, aby zmienić wartość właściwości Text kontrolki Button.

- Krok 1.** Upewnij się, że kontrolka Button została zaznaczona. (Jeżeli nie widzisz ramki ograniczającej i uchwytów zmiany wielkości wokół kontrolki, po prostu ją kliknij, co spowoduje jej zaznaczenie).
- Krok 2.** W oknie *Properties* odszukaj właściwość Text.
- Krok 3.** Kliknij obszar zawierający wartość właściwości Text i usuń aktualną zawartość. Następnie wpisz nowy tekst i naciśnij klawisz *Enter*. Nowy tekst zostanie wyświetlony na przycisku.

Na rysunku 2.11 zaprezentowałem przykład pokazujący jak zmiana właściwości Text kontrolki Button powoduje zmianę tekstu wyświetlanego przez przycisk.



Rysunek 2.11. Zmiana właściwości Text kontrolki Button

Zmiana nazwy kontrolki

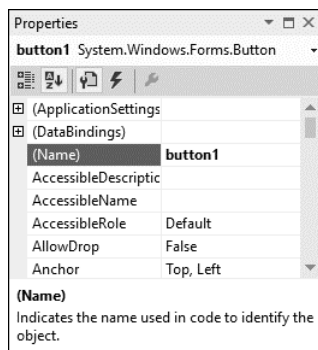
Nazwa kontrolki identyfikuje ją w kodzie aplikacji i w środowisku Visual Studio. Gdy utworzysz kontrolkę w formularzu aplikacji, zawsze powinieneś zmienić jej nazwę na znacznie lepiej (niż domyślnie nadana przez Visual Studio) wskazującą przeznaczenie

danej kontrolki. Warto w tym miejscu dodać, że nazwa kontrolki powinna wyraźnie odzwierciedlać jej przeznaczenie.

Przykładowo: przyjmuję założenie o utworzeniu kontrolki `Button`, której kliknięcie powoduje obliczenie wysokości podatku. Nazwa domyślna w postaci `button1` w żaden sposób nie określa sposobu działania tego przycisku. W omawianym przykładzie znacznie lepszą nazwą będzie `calculateTaxButton`. Gdy pracujesz z kodem aplikacji i zobaczysz nazwę `calculateTaxButton`, będziesz wiedział, do którego przycisku odwołuje się dany fragment kodu.

Zmiana nazwa kontrolki odbywa się za pomocą jej właściwości `Name`. Oto kroki konieczne do wykonania.

- Krok 1.** Upewnij się o zaznaczeniu kontrolki. (Jeżeli nie widzisz ramki ograniczającej i uchwytów zmiany wielkości wokół kontrolki, po prostu ją kliknij, co spowoduje jej zaznaczenie).
- Krok 2.** W oknie *Properties* przewiń do początku listę właściwości. Jak pokazałem na rysunku 2.12, powinieneś zobaczyć właściwość o nazwie `Name`. (Nazwa tej właściwości została ujęta w nawias, aby znajdowała się gdzieś na początku listy właściwości. Dzięki temu znacznie łatwiej można ją odszukać).
- Krok 3.** Kliknij obszar zawierający wartość właściwości `Name` i usuń aktualną zawartość. Następnie wpisz nową nazwę i naciśnij klawisz *Enter*. W ten sposób zmieniłeś nazwę zaznaczonej kontrolki.

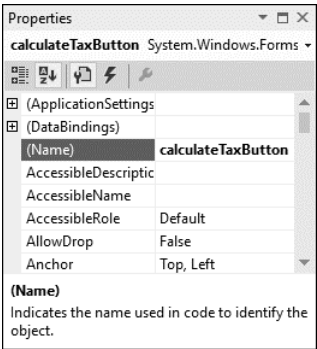


Rysunek 2.12. Właściwość `Name` kontrolki

Na rysunku 2.13 pokazałem okno *Properties* kontrolki `Button` po zmianie jej nazwy na `calculateTaxButton`.

Reguły dotyczące nadawania nazw kontrolkom

Nazwy kontrolki są określane również mianem **identyfikatorów**. Podczas nadawania nazwy kontrolce należy się stosować do wymienionych niżej reguł dotyczących identyfikatorów w języku C#:



Rysunek 2.13. Wartość właściwości Name została zmieniona na calculateTaxButton

- Pierwszym znakiem musi być mała lub duża litera bądź też znak podkreślenia (_).
- Drugim i kolejnym znakiem może być mała lub duża litera, cyfra bądź znak podkreślenia (_).
- Nazwa nie może zawierać spacji.

W tabeli 2.1 wymieniłem pewne identyfikatory, które mogą być użyte w nazwach kontrolerek Button, i jednocześnie wskazałem, czy są prawidłowe, czy też nie.

Tabela 2.1. Prawidłowe i nieprawidłowe identyfikatory w języku C#

Identyfikator	Prawidłowy czy nieprawidłowy?
showDayOfWeekButton	Prawidłowy.
3rdQuarterButton	Nieprawidłowy, ponieważ identyfikator nie może rozpoczynać się od cyfry.
change*color*Button	Nieprawidłowy, ponieważ identyfikator nie może zawierać gwiazdki.
displayTotalButton	Prawidłowy.
calculate Tax Button	Nieprawidłowy, ponieważ identyfikator nie może zawierać spacji.

Skoro nazwa kontrolki powinna odzwierciedlać jej przeznaczenie, programiści często tworzą nazwy składające się z kilku słów. Spójrz na przedstawione niżej przykłady nazw kontrolerek Button:

```
calculatetaxbutton
printreportbutton
displayanimationbutton
```

Niestety te nazwy nie są łatwe do odczytania przez człowieka ze względu na brak separatorów między słowami. Ponieważ w nazwie kontrolki nie można użyć spacji, konieczne okazało się znalezienie innego sposobu na rozdzielenie słów w takiej nazwie kontrolki, aby stała się czytelniejsza dla człowieka.

Większość programistów C# rozwiązało ten problem za pomocą konwencji nazw **camelCase**. W przypadku tej konwencji nazwy są zapisywane w następujący sposób:

- Nazwa rozpoczyna się małą literą.
- Pierwsza litera drugiego i kolejnego słowa jest duża.

Oto przykładowe nazwy kontroltek zapisane w konwencji camelCase:

```
calculateTaxButton
printReportButton
displayAnimationButton
```



UWAGA. Ta konwencja nazw nosi nazwę camelCase, ponieważ duże litery w nazwach czasami przypominają garby wielbłąda (ang. *camel*).



Punkt kontrolny

- 2.1. Co zostaje automatycznie utworzone i wyświetlone w oknie *Designer* po rozpoczęciu pracy nad nowym projektem Visual C#?
- 2.2. Po czym można poznać, że obiekt jest zaznaczony i gotowy do edycji w oknie *Designer*?
- 2.3. Jakie jest przeznaczenie uchwytów zmiany wielkości elementu?
- 2.4. Co muszą mieć każdy formularz i kontrolka w graficznym interfejsie użytkownika aplikacji, aby można było zidentyfikować dany element?
- 2.5. Jakie jest przeznaczenie okna *Properties*?
- 2.6. Na czym polega działanie przycisku *Alphabetical* po jego kliknięciu w oknie *Properties*?
- 2.7. Na czym polega działanie przycisku *Categorized* po jego kliknięciu w oknie *Properties*?
- 2.8. Co określa właściwość *Text* formularza?
- 2.9. Co określa właściwość *Size* formularza?
- 2.10. Co jest wyświetlane w oknie *Toolbox*?
- 2.11. Jak można dodać kontrolkę do formularza?
- 2.12. Co powinien wskazywać tekst wyświetlany na przycisku?
- 2.13. Jakie są reguły dotyczące nadawania nazw kontrolkom?
- 2.14. Jaką konwencję nazw stosuje większość programistów C#, aby rozdzielać słowa w identyfikatorach składających się z wielu słów?

2.2.

Utworzenie graficznego interfejsu użytkownika w pierwszej aplikacji Visual C# — Hello World

Gdy rozpoczyna się naukę nowego języka programowania, tradycyjnie pierwszy utworzony program jest typu **Witaj, świecie!** Jego działanie polega po prostu na wyświetleniu komunikatu *Witaj, świecie!*. W tym podrozdziale utworzysz pierwszą aplikację Visual C#, którą będzie sterowany zdarzeniami program typu *Witaj, świecie!*. Gdy opracowana tutaj aplikacja zostanie uruchomiona, wyświetli formularz pokazany po lewej stronie na rysunku 2.14.



Rysunek 2.14. Okna wyświetlane po ukończeniu pracy nad programem typu *Witaj, świecie!*

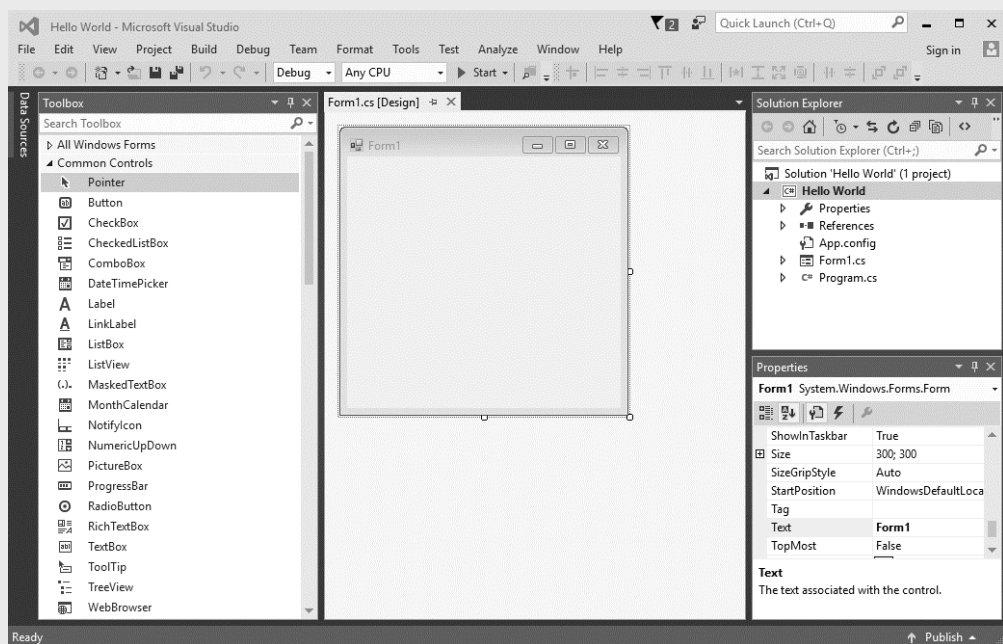
Proces utworzenia tej aplikacji został podzielony na dwa etapy. W pierwszym zajmiesz się przygotowaniem graficznego interfejsu użytkownika aplikacji, natomiast w drugim utworzysz kod powodujący wyświetlenie okna dialogowego wraz z komunikatem *Witaj, świecie!* po kliknięciu przycisku *Wyświetl komunikat*. W przykładzie 2.1 przedstawiłem kroki pokazujące proces utworzenia graficznego interfejsu użytkownika.

Przykład 2.1.

Utworzenie graficznego interfejsu użytkownika aplikacji typu *Witaj, świecie!*

- Krok 1.** Uruchom Visual Studio.
- Krok 2.** Utwórz nowy projekt, wybierając z menu *File* opcję *New project...*
- Krok 3.** Na ekranie powinno zostać wyświetlone okno *New Project*. Po lewej stronie okna, w sekcji *Installed/Templates* upewnij się o zaznaczeniu *Visual C#*. Następnie jako rodzaj aplikacji wybierz *Windows Forms Application*. W polu tekstowym *Name* (w dolnej części okna) zmień nazwę projektu na *Hello World* i kliknij przycisk *OK*.

Krok 4. Upewnij się, że na ekranie są wyświetlone okna *Toolbox*, *Solution Explorer* i *Properties* oraz mają wyłączoną funkcję automatycznego ukrywania. Środowisko Visual Studio powinno wyglądać, jak pokazałem na rysunku 2.15.



Rysunek 2.15. Środowisko Visual Studio

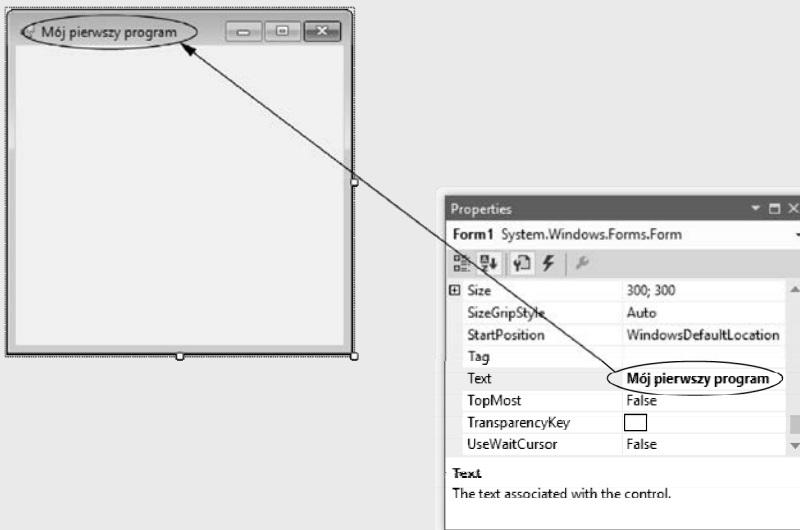
Krok 5. Wartość właściwości *Text* formularza *Form1* zmień na *Mój pierwszy program*, jak pokazałem na rysunku 2.16.

Krok 6. Domyślna wielkość formularza jest zbyt duża w przypadku tej aplikacji, dlatego należy go zmniejszyć. Wykorzystaj omówioną w poprzednim podrozdziale technikę i zmniejsz formularz za pomocą myszy. Po tej operacji formularz powinien być podobny do pokazanego na rysunku 2.17. (Nie przejmuj się konkretną wielkością formularza, powinien być jedynie podobny do pokazanego na rysunku 2.17).

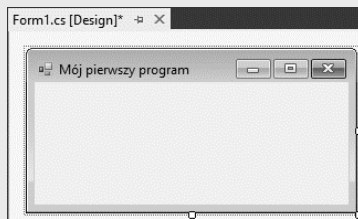
Krok 7. W tym momencie można już dodać kontrolkę *Button* do formularza. W oknie *Toolbox* odśledź narzędzie *Button* i dwukrotnie je kliknij. W formularzu powinna się pojawić kontrolka *Button*, jak pokazałem na rysunku 2.18. Przesuń ją tak, aby znalazła się mniej więcej w środku formularza (zobacz rysunek 2.19).

Krok 8. Wartość właściwości *Text* kontrolki *Button* zmień na *Wyświetl komunikat*. Po wprowadzeniu tej zmiany zauważysz, że zmienił się również tekst wyświetlany na przycisku, jak pokazałem na rysunku 2.20.

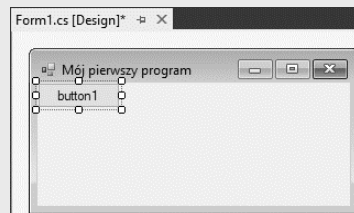
Krok 9. Kontrolka *Button* nie jest jeszcze na tyle duża, aby pomieścić cały tekst wpisany we właściwości *Text*. Dlatego też powiększ ją, jak pokazałem na rysunku 2.21.



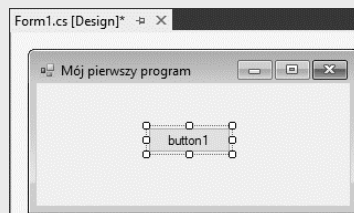
Rysunek 2.16. Wartość właściwości Text formularza została zmieniona na Mój pierwszy program



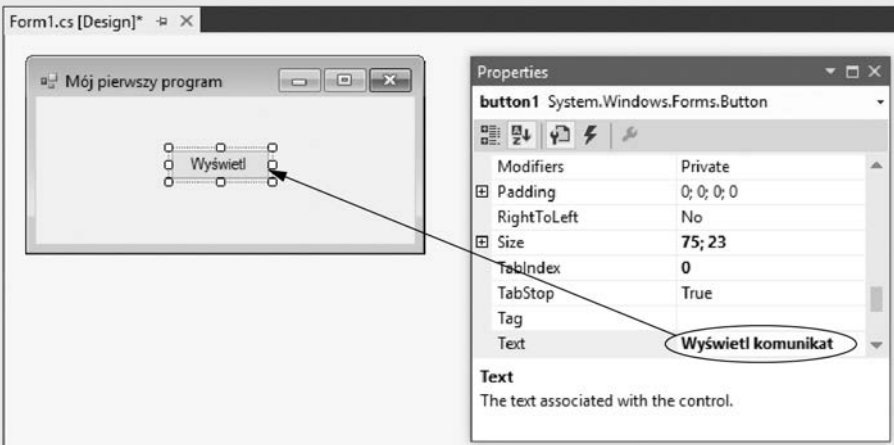
Rysunek 2.17. Formularz po zmianie jego wielkości



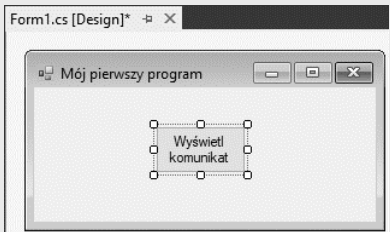
Rysunek 2.18. Kontrolka Button utworzona w formularzu



Rysunek 2.19. Kontrolka Button umieszczona na środku formularza

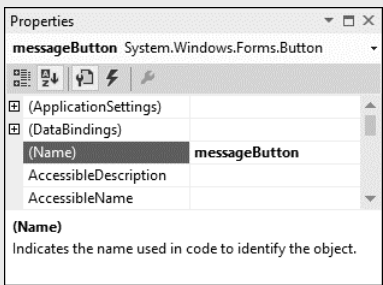


Rysunek 2.20. Zmiana wartości właściwości Text kontrolki Button



Rysunek 2.21. Powiększona kontrolka Button

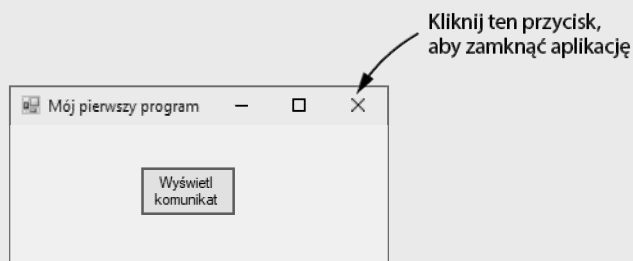
Krok 10. Jak wspomniałem w poprzednim podrozdziale, nazwa kontrolki powinna wskazywać na jej przeznaczenie. Utworzona w tym miejscu kontrolka przycisku ma po kliknięciu spowodować wyświetlenie komunikatu. Jej nazwa domyślna, `button1`, nie odzwierciedla przeznaczenia przycisku. Dlatego też wartość właściwości `Name` kontrolki `Button` zmień na `messageButton`. Po wprowadzeniu tej zmiany okno *Properties* powinno wyglądać, jak pokazałem na rysunku 2.22.



Rysunek 2.22. Wartość właściwości `Name` kontrolki `Button` została zmieniona na `messageButton`

Krok 11. Z menu *File* w Visual Studio wybierz opcję *Save All*, aby zapisać projekt.

Krok 12. Wprawdzie aplikacja została utworzona jedynie częściowo, ale można ją uruchomić i zobaczyć, jak prezentuje się przygotowany graficzny interfejs użytkownika. Aby uruchomić aplikację, naciśnij klawisz *F5* lub kliknij przycisk *Start Debugging* (▶) na pasku narzędzi. To spowoduje skompilowanie i uruchomienie aplikacji. Powinieneś zauważyć pewną zmianę wyglądu środowiska Visual Studio oraz wyświetlone na ekranie okno aplikacji, jak pokazałem na rysunku 2.23.



Rysunek 2.23. Okno uruchomionej aplikacji

Choć aplikacja została uruchomiona, to jej działanie sprowadza się jedynie do wyświetlenia formularza. Kliknięcie przycisku *Wyświetl komunikat* nie ma żadnego efektu. To wynika z tego, że nie utworzyłeś jeszcze żadnego kodu wykonywanego po kliknięciu przycisku. Przygotowaniem niezbędnego kodu źródłowego zajmiesz się w kolejnym przykładzie. Jeżeli chcesz zakończyć działanie aplikacji, kliknij standardowy w Windows przycisk zamknięcia aplikacji (X) znajdujący się w prawym górnym rogu okna.

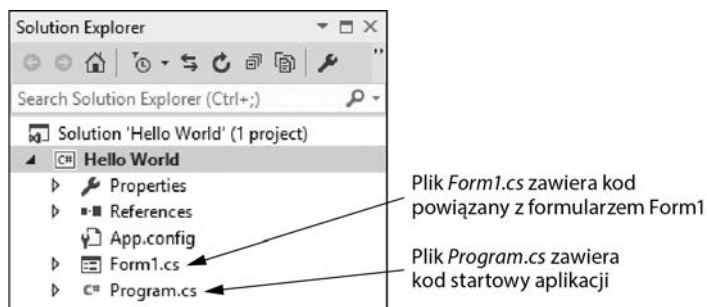
2.3. Wprowadzenie do kodu w języku C#

WYJAŚNIENIE. Edytora kodu Visual Studio będziesz używać do utworzenia kodu źródłowego aplikacji. Większość kodu tworzonego w aplikacji będą stanowiły procedury obsługi zdarzeń. Procedura obsługi zdarzeń reaguje na konkretne zdarzenie, które wystąpiło w trakcie działania aplikacji.

W poprzednim podrozdziale poznałeś podstawy dotyczące tworzenia graficznego interfejsu użytkownika aplikacji. Jednak aplikacja to znacznie więcej niż tylko interfejs użytkownika. Jeżeli chcesz, aby program wykonywał jakiegokolwiek użyteczne operacje, musisz utworzyć odpowiedni kod. W tym podrozdziale przedstawię wprowadzenie do kodu Visual C# i pokażę, jak przygotować aplikację, aby reagowała na kliknięcie przycisku.

Plik zawierający kod programu nosi nazwę pliku kodu źródłowego. Gdy rozpoczniesz pracę nad nowym projektem typu *Windows Forms Application* w Visual C#, Visual Studio automatycznie tworzy pewne pliki kodu źródłowego i umieszcza je w projekcie.

Jeżeli spojrzysz na pokazane na rysunku 2.24 okno *Solution Explorer*, zobaczysz nazwy dwóch takich plików: *Form1.cs* i *Program.cs*. (Pliki kodu źródłowego w C# zawsze mają rozszerzenie *.cs*).



Rysunek 2.24. Plik kodu źródłowego w oknie *Solution Explorer*

Oto krótkie omówienie tych dwóch plików:

- Plik *Program.cs* zawiera kod startowy aplikacji wykonywany po jej uruchomieniu. Kod zdefiniowany w tym pliku przeprowadza w tle inicjalizację zadań niezbędnych do prawidłowego uruchomienia aplikacji. Nie powinieneś modyfikować zawartości tego pliku, ponieważ możesz w ten sposób uniemożliwić uruchomienie aplikacji.
- Plik *Form1.cs* zawiera kod powiązany z formularzem *Form1*. Gdy tworzysz kod definiujący czynność powiązaną z formularzem *Form1*, np. reakcja na kliknięcie przycisku, niezbędne polecenia umieszczasz w tym właśnie pliku.

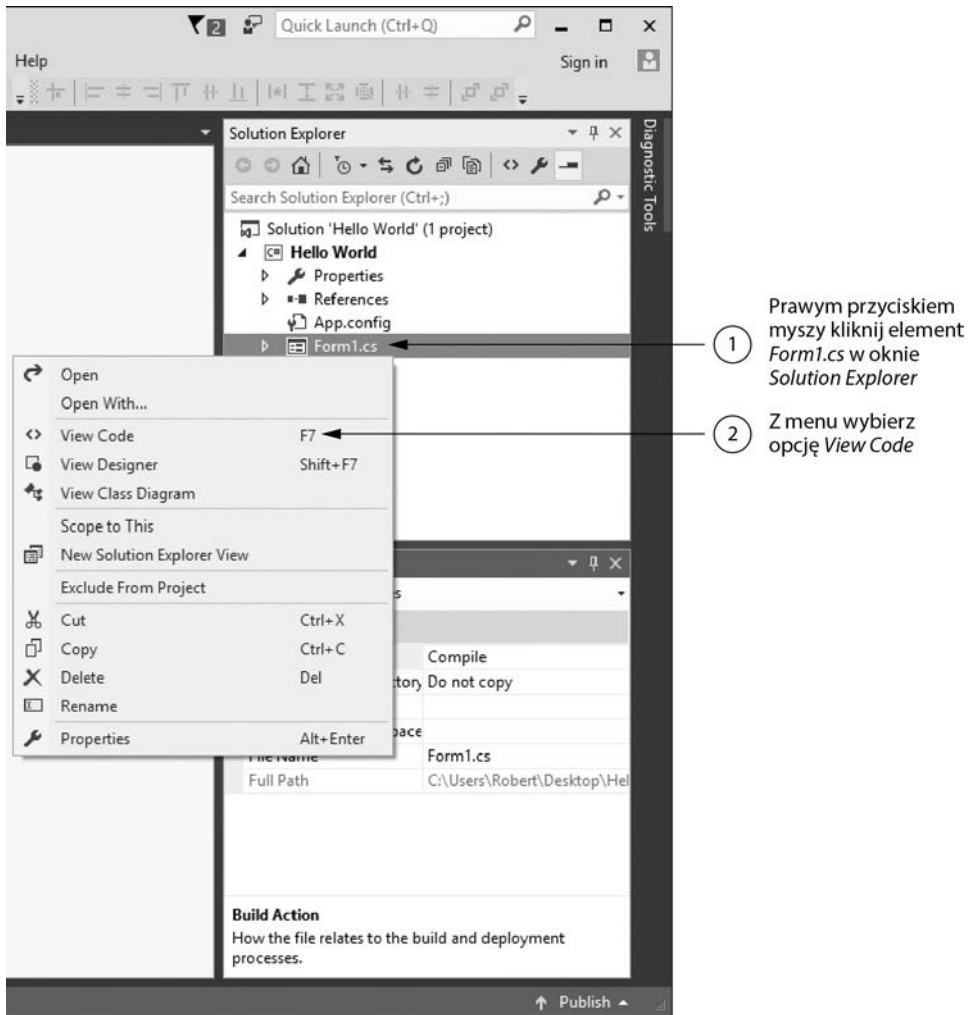


UWAGA. W oknie *Solution Explorer* być może zobaczysz jeszcze inne pliki kodu źródłowego niż pokazane wcześniej na rysunku 2.24.

Plik *Form1.cs* zawiera już kod wygenerowany przez Visual Studio podczas tworzenia projektu. Ten automatycznie wygenerowany kod możesz potraktować jako szkielet, do którego będziesz mógł dodawać własny kod w trakcie pracy nad aplikacją.

Spójrz na kod. Jeżeli nadal masz otwarty projekt *Hello World* z poprzedniego przykładu, prawym przyciskiem myszy kliknij element *Form1.cs* w oknie *Solution Explorer*. Na ekranie zostanie wyświetlone menu kontekstowe, takie jak pokazane na rysunku 2.25. Z menu kontekstowego wybierz opcję *View Code*. Zawartość pliku zostanie wyświetlona w edytorze kodu źródłowego Visual Studio, jak pokazałem na rysunku 2.26.

W tym momencie nie musisz rozumieć znaczenia poleceń znajdujących się w wyświetlonym kodzie źródłowym. O wiele ważniejsze jest teraz poznanie sposobu organizacji tego kodu, ponieważ później będziesz dodawać własny kod do pliku *Form1.cs*. Kod źródłowy w języku C# jest zorganizowany na trzy główne sposoby — przeszerzenie nazw, klasy i metody. Oto ich krótkie omówienie:



Rysunek 2.25. Otwarcie pliku Form1.cs w edytorze kodu źródłowego

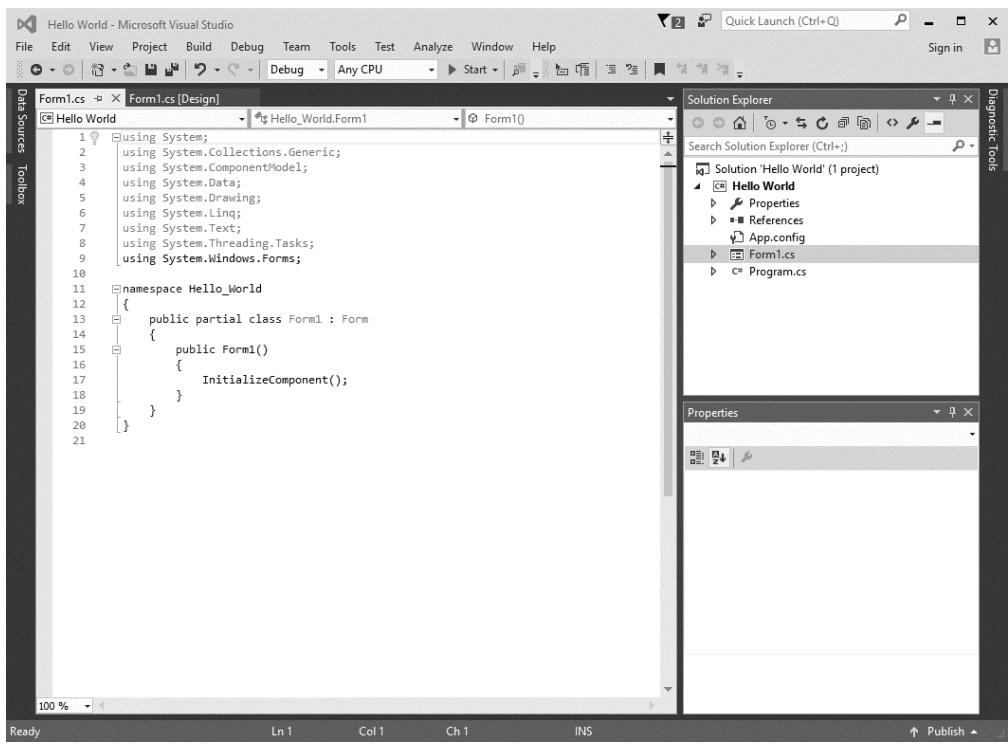
- **Przestrzeń nazw** to kontener przechowujący klasy.
- **Klasa** to kontener przechowujący metody.
- **Metoda** to co najmniej jedno polecenie przeprowadzające pewną operację.

Tak więc kod w C# jest zorganizowany w metody zdefiniowane w klasach znajdujących się w przestrzeniach nazw. Pamiętając o tej strukturze organizacyjnej, spójrz na listing 2.1.

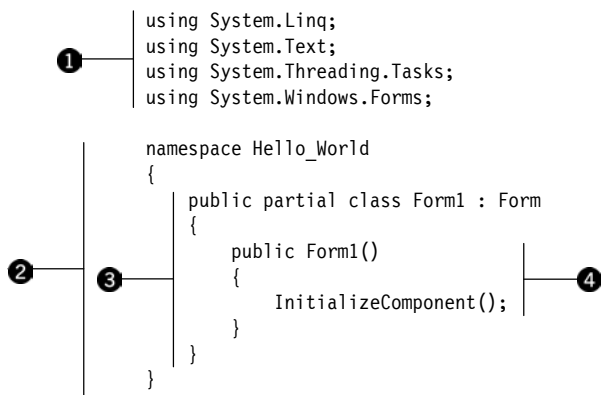
Listing 2.1. Sposób organizacji kodu w pliku Form1.cs

①

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
```



Rysunek 2.26. Zawartość pliku Form1.cs wyświetlona w edytorze kodu źródłowego Visual Studio



Na listingu 2.1 pokazałem cztery oddzielne sekcje kodu oznaczone kolejnymi cyframi. Przedstawię teraz krótkie omówienie poszczególnych sekcji:

- 1 Przypomnij sobie z rozdziału 1., że aplikacje C# opierają się na .NET Framework, czyli kolekcji klas i innego kodu. Kod .NET Framework został zorganizowany w postaci przestrzeni nazw. Seria poleceń `using` na początku kodu źródłowego C# wskazuje przestrzenie nazw .NET Framework, które będą używane przez dany program.

- ❷ Ta sekcja kodu powoduje utworzenie przestrzeni nazw do projektu. Pierwsze polecenie w tej sekcji, `namespace Hello_World`, oznacza początek przestrzeni nazw `Hello_World`. Zwróć uwagę na kolejny wiersz wraz z otwierającym nawiasem klamrowym (`{`) i ostatni wiersz wraz z zamykającym nawiasem klamrowym (`}`). Cały kod umieszczony między tymi nawiasami znajduje się w przestrzeni nazw `Hello_World`.
- ❸ Ta sekcja kodu jest deklaracją klasy. Pierwsze polecenie w tej sekcji, `public partial class ...`, oznacza początek klasy. Kolejny wiersz zawiera otwierający nawias klamrowy (`{`), a ostatni wiersz zawiera zamykający nawias klamrowy (`}`). Cały kod umieszczony między tymi nawiasami został zdefiniowany w klasie o podanej nazwie.
- ❹ Ta sekcja kodu jest deklaracją metody. Pierwsze polecenie w tej sekcji, `public Form1()`, oznacza początek metody. Kolejny wiersz zawiera otwierający nawias klamrowy (`{`), a ostatni wiersz zawiera zamykający nawias klamrowy (`}`). Cały kod umieszczony między tymi nawiasami został zdefiniowany w metodzie o podanej nazwie.

Trzeba koniecznie zwrócić uwagę na bardzo ważną kwestię — kontenery kodu, np. przestrzeń nazw, klasa i metoda, używają nawiasów klamrowych do wskazania zdefiniowanego w nich kodu. Każdemu otwierającemu nawiasowi klamrowemu musi odpowiadać zamykający nawias klamrowy. Na listingu 2.2 pokazałem nawiasy klamrowe użyte w pliku *Form1.cs*.

Listing 2.2. Odpowiadające sobie nawiasy klamrowe w pliku *Form1.cs*

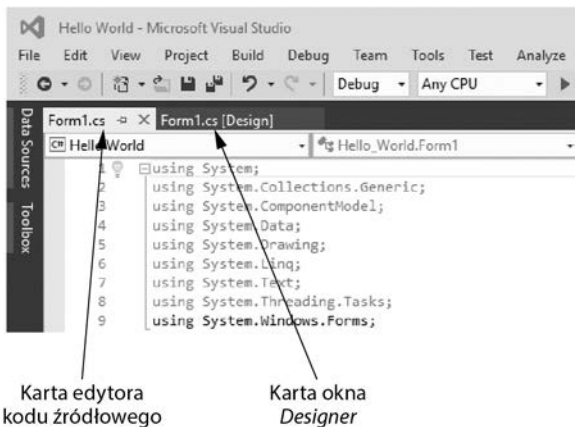
```

namespace Hello_World
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}

```

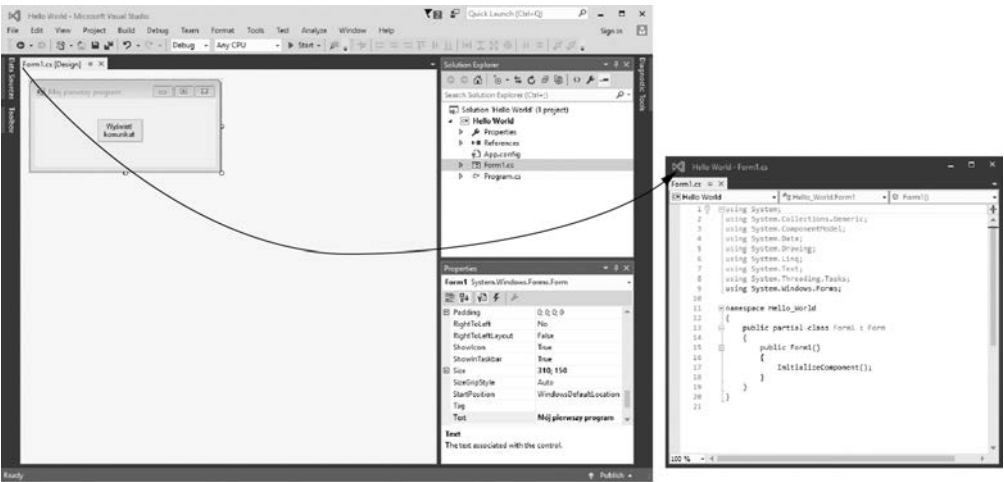
Przechodzenie między edytorem kodu źródłowego a oknem Designer

Gdy otworzysz edytor kodu źródłowego, zostanie wyświetlony w miejscu zajmowanym przez okno *Designer*. Podczas opracowywania aplikacji Visual C# bardzo często będziesz musiał przechodzić między edytorem kodu źródłowego a oknem *Designer*. Jednym ze sposobów na szybkie przechodzenie między nimi jest użycie kart pokazanych na rysunku 2.27. Patrząc na rysunek, zwróć uwagę na pierwszą kartę od lewej, *Form1.cs*, wskazującą edytor kodu źródłowego. Natomiast druga karta od lewej, *Form1.cs [Design]*, przedstawia okno *Designer*. (Karty nie zawsze będą wyświetlane w tej kolejności). Aby przejść między oknem *Designer* a edytorem kodu źródłowego, wystarczy po prostu kliknąć kartężądanego okna.

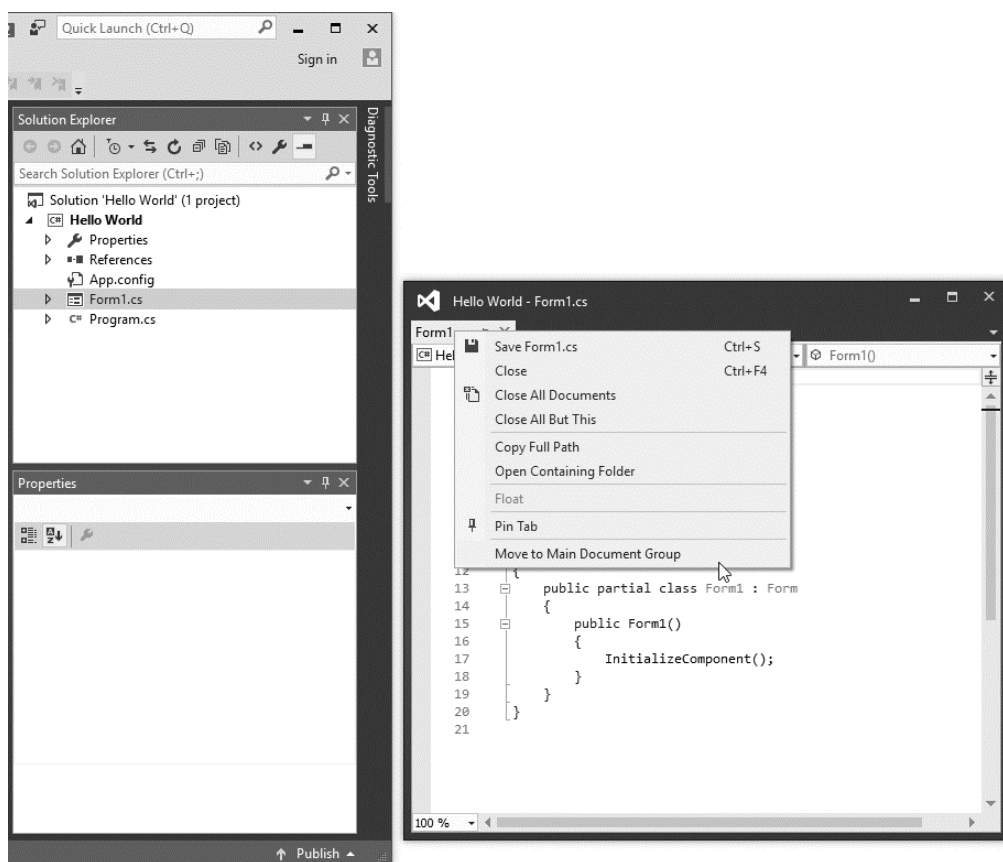


Rysunek 2.27. Karty edytora kodu źródłowego i okna Designer

Okno edytora kodu źródłowego możesz określić jako pływające i umieścić je w innym położeniu na ekranie. To pozwala na jednoczesne wyświetlanie obu wymienionych okien. Jak pokazałem na rysunku 2.28, wystarczy kliknąć okno edytora kodu źródłowego i przeciągnąć je w wybrane miejsce na ekranie. (Jeżeli do komputera masz podłączonych kilka monitorów, okno możesz przeciągnąć na dowolny z nich). Aby z powrotem przenieść okno edytora kodu źródłowego do środowiska IDE, prawym przyciskiem myszy kliknij kartę pliku kodu źródłowego w oknie kodu, a następnie z menu kontekstowego wybierz opcję *Move to Main Document Group* — pokazałem to na rysunku 2.29.



Rysunek 2.28. Zdefiniowanie okna edytora kodu źródłowego jako pływającego przez jego kliknięcie i przesunięcie

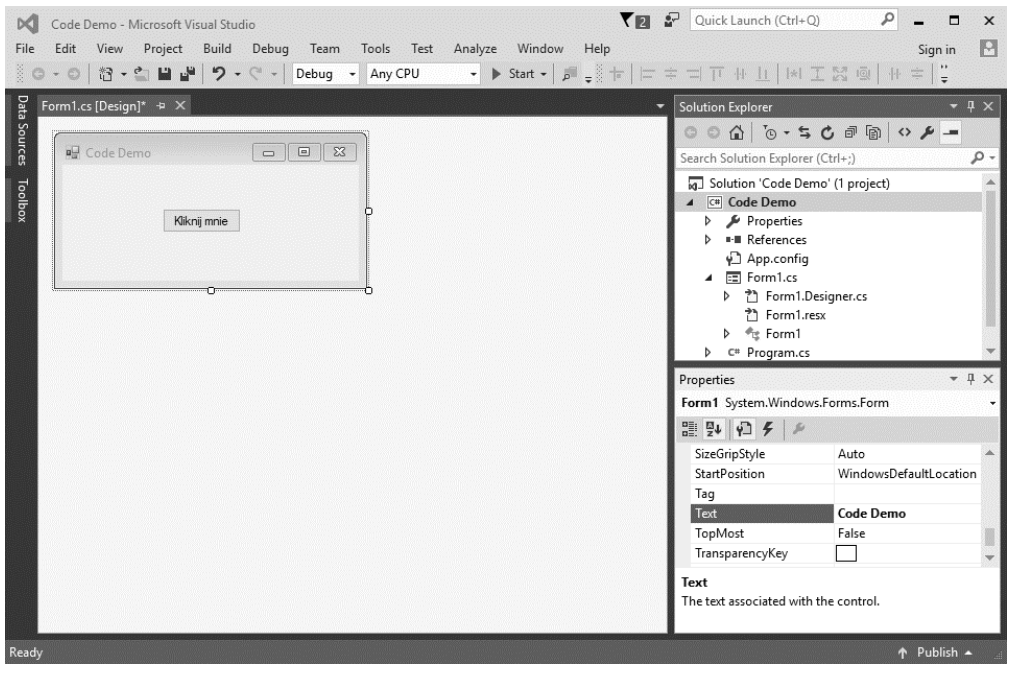


Rysunek 2.29. Przywrócenie okna edytora kodu źródłowego do środowiska IDE

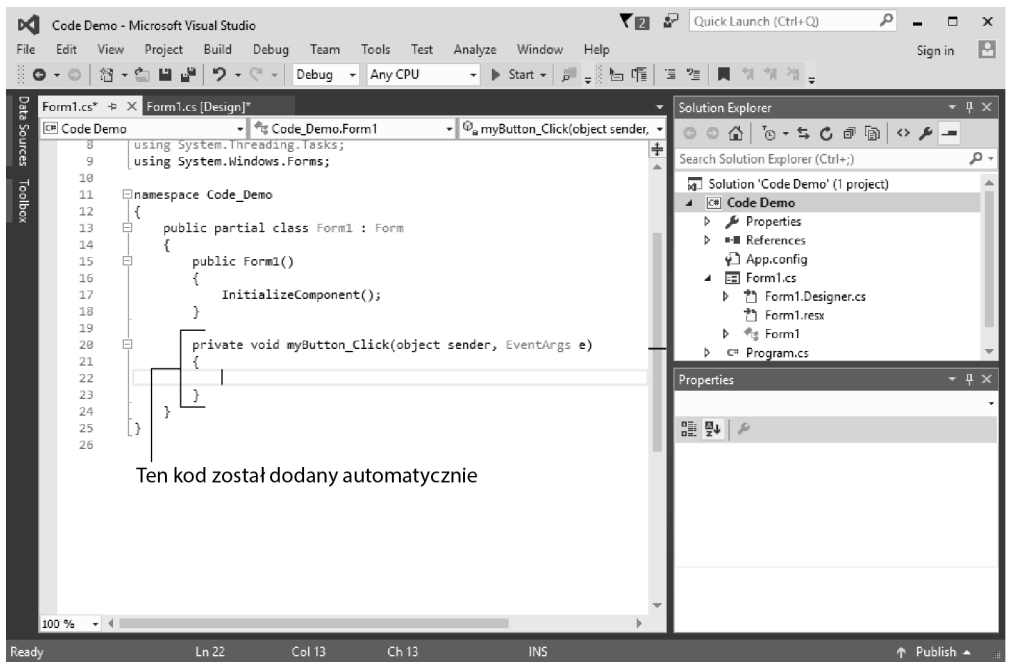
Dodawanie własnego kodu do projektu

Teraz możesz się dowiedzieć, jak dodać własny kod do projektu. Przyjmuję założenie, że utworzyłeś projekt o nazwie *Code Demo* i przygotowałeś formularz wraz z kontrolką Button, jak pokazałem na rysunku 2.30. Nazwa tej kontrolki Button to *myButton*, natomiast wartością jej właściwości Text jest *Kliknij mnie*.

Przyjmuję założenie, że po kliknięciu tego przycisku aplikacja ma wyświetlić komunikat *Dziękujemy za kliknięcie przycisku!*. Aby wykonać to zadanie, trzeba utworzyć specjalny typ metody nazywanej procedurą obsługi zdarzeń. **Procedura obsługi zdarzeń** to metoda wykonywana po wystąpieniu konkretnego zdarzenia w trakcie działania aplikacji. W omawianym przykładzie procedura obsługi zdarzeń ma zostać wywołana po kliknięciu kontrolki *myButton* przez użytkownika. Utworzenie procedury obsługi zdarzeń wymaga dwukrotnego kliknięcia kontrolki *myButton* w oknie *Designer*. To spowoduje otwarcie pliku *Form1.cs* w edytorze kodu źródłowego, jak pokazałem na rysunku 2.31, wraz z dodanym do niego nowym kodem.



Rysunek 2.30. Formularz wraz z kontrolką Button



Rysunek 2.31. Okno edytora kodu źródłowego zawierające wygenerowany kod procedury obsługi zdarzeń

Gdy aplikacja działa i użytkownik kliknie kontrolkę, wówczas mówimy o wystąpieniu w tej kontrolce **zdarzenia Click**. Kod dodany do pliku *Form1.cs* (widoczny na rysunku 2.31) to procedura obsługi zdarzeń, która zostanie wywołana po wystąpieniu zdarzenia Click w kontrolce `myButton`. W tym momencie nie musisz próbować zrozumieć wszystkich sekcji kodu procedury obsługi zdarzeń. Najważniejsze jest poznanie wymienionych tutaj koncepcji:

- Jak pokazałem na listingu 2.3, nazwą procedury obsługi zdarzeń jest `myButton_Click()`. Człon `myButton` nazwy wskazuje na powiązanie tej procedury obsługi zdarzeń z kontrolką `myButton`. Natomiast człon `Click` nazwy sygnalizuje, że dana procedura obsługi zdarzeń reaguje na zdarzenie Click. To jest typowa konwencja nazw używana przez Visual Studio podczas generowania kodu procedury obsługi zdarzeń. Gdy widzisz nazwę `myButton_Click()`, powinieneś od razu wiedzieć, że to jest procedura obsługi zdarzeń wywoływana po wystąpieniu zdarzenia Click w kontrolce `myButton`.

Listing 2.3. Szkielet kodu procedury obsługi zdarzeń

```

_____ Nazwa procedury obsługi zdarzeń.
private void myButton_Click(object sender, EventArgs e)
{
    | ←—— Między nawiasami klamrowymi znajduje się miejsce na Twój kod.
}

```

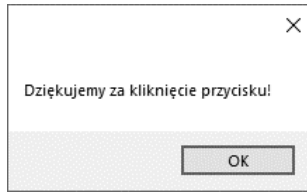
- Wygenerowana przez Visual Studio procedura obsługi zdarzeń nie wykonuje żadnego zadania. Możesz ją potraktować jako pusty kontener przeznaczony na Twój kod. Zwróć uwagę na drugi i ostatni wiersz tego fragmentu kodu — odpowiednio otwierający i zamykający nawias klamrowy. Każde polecenie, które ma zostać wykonane po kliknięciu kontrolki `myButton` przez użytkownika, musi się znaleźć między tymi nawiasami.

Teraz już wiesz, jak można utworzyć pustą procedurę obsługi zdarzeń Click do kontrolki `Button`. Mogłbyś w tym miejscu zapytać, jakie polecenia należy umieścić w tej procedurze. W omawianym przykładzie to będzie kod odpowiedzialny za wyświetlenie okna dialogowego wraz z komunikatem *Dziękujemy za kliknięcie przycisku!*.

Okno komunikatu

Okno komunikatu to małe okno czasami nazywane **oknem dialogowym**, które wyświetla pewien komunikat. Na rysunku 2.32 pokazałem przykład okna komunikatu wyświetlającego tekst *Dziękujemy za kliknięcie przycisku!*. Zwróć uwagę na to, że pokazane okno zawiera również przycisk OK. Jego kliknięcie spowoduje zamknięcie okna.

.NET Framework dostarcza metodę o nazwie `MessageBox.Show()`, której wykonanie w Visual C# spowoduje wyświetlenie okna komunikatu. Jeżeli chcesz wywołać tę metodę, musisz przygotować polecenie nazywane **wywołaniem metody**. (Akt



Rysunek 2.32. Przykładowe okno komunikatu

wykonania metody jest bardzo często przez programistów określany mianem *wywołania metody*). Kolejne polecenie pokazuje, jak można wywołać metodę `MessageBox.Show()` i wyświetlić okno komunikatu pokazane na rysunku 2.32:

```
MessageBox.Show("Dziękujemy za kliknięcie przycisku!");
```

W trakcie wywołania metody, w nawiasie okrągłym umieszczasz pewien ciąg tekstowy. (W programowaniu pojęcie **ciąg tekstowy** oznacza zbiór znaków). Ciąg tekstowy podany w nawiasie będzie tekstem wyświetlonym w oknie komunikatu. W omawianym przykładzie w nawiasie znalazł się ciąg tekstowy `Dziękujemy za kliknięcie przycisku!`.

Zwróć uwagę na ujęcie tego ciągu tekstowego w cudzysłów. Jednak w trakcie wyświetlania komunikatu (patrz rysunek 2.32) te znaki cudzysłów zostają pominięte. Cudzysłów jest potrzebny w kodzie, aby wskazać początek i koniec ciągu tekstowego.

Zwróć również uwagę na średnik umieszczony na końcu polecenia — jest on wymagany przez składnię C#. Podobnie jak kropka oznacza koniec zdania, tak samo **średnik** w kodzie źródłowym C# wskazuje na koniec polecenia.

Powracamy do przykładowego projektu *Code Demo*. Na listingu 2.4 pokazałem przykład wywołania metody `MessageBox.Show()` w procedurze obsługi zdarzeń `myButton_Click()`. Po wpisaniu polecenia w postaci pokazanej na listingu 2.4 naciśnij klawisz `F5` lub kliknij przycisk *Start Debugging* (🔍) na pasku narzędzi. To spowoduje skompilowanie i uruchomienie aplikacji. Uruchomiona aplikacja wyświetli formularz pokazany po lewej stronie na rysunku 2.33. Gdy klikniesz przycisk, wyświetlone zostanie okno komunikatu widoczne po prawej stronie na rysunku 2.33. Zamknięcie okna komunikatu następuje po kliknięciu przycisku `OK`.

Listing 2.4. Procedura obsługi zdarzeń wyświetlająca okno komunikatu

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

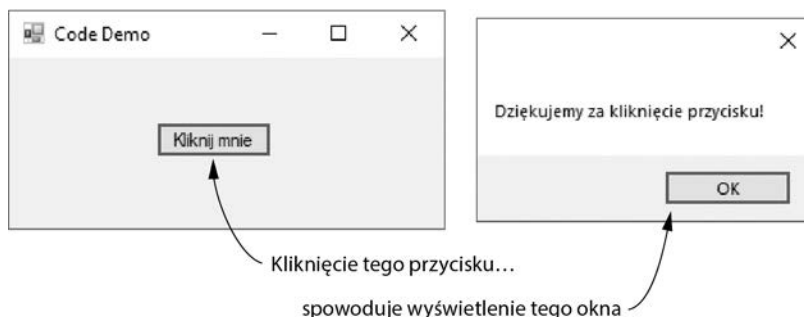
namespace Code_Demo
{
    public partial class Form1 : Form
    {
```

```

public Form1()
{
    InitializeComponent();
}

private void myButton_Click(object sender, EventArgs e)
{
    MessageBox.Show("Dziękujemy za kliknięcie przycisku!");
}
}

```



Rysunek 2.33. Uruchomiony projekt Code Demo



UWAGA. Podczas tworzenia procedury obsługi zdarzeń `Click` do kontrolki `Button` być może zastanawiałeś się, dlaczego konieczne jest najpierw dwukrotne kliknięcie tej kontrolki w oknie *Designer* i tym samym utworzenie szkieletu kodu procedury obsługi zdarzeń. Czy nie można pominąć tego kroku, otworzyć okno edytora kodu źródłowego, a następnie samodzielnie wpisać cały kod procedury obsługi zdarzeń? Odpowiedź brzmi: nie, nie można pominąć tego kroku. Dwukrotne kliknięcie kontrolki w oknie *Designer* powoduje utworzenie przez Visual Studio nie tylko szkieletu procedury obsługi zdarzeń, ale także wygenerowanie pewnego kodu, którego nie zobaczysz w innych miejscach projektu. Ten kod jest niezbędny do prawidłowego działania procedury obsługi zdarzeń.

Literał ciągu tekstowego

Program prawie zawsze działa z pewnego rodzaju danymi. Przykładowo: kod przedstawiony na listingu 2.4 używa podczas wywołania metody `MessageBox.Show()` pokazanego tutaj ciągu tekstowego:

```
"Dziękujemy za kliknięcie przycisku!"
```

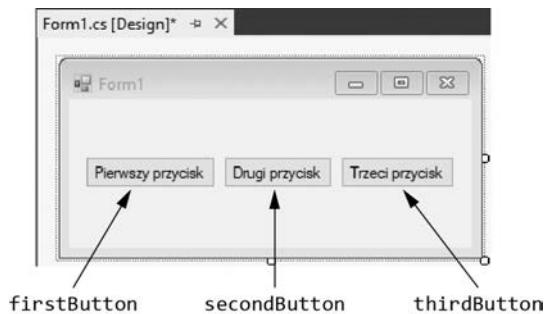
Ten ciąg tekstowy to dane wyświetlane przez program. Gdy pewien fragment danych zostaje umieszczony w kodzie źródłowym programu, wówczas nazywany go **literałem**, ponieważ jest dosłownie (ang. *literal*) zapisany w kodzie programu. Dlatego też ciąg tekstowy umieszczony w kodzie źródłowym programu jest nazywany *literałem ciągu tekstowego*. W języku C# wszystkie literały ciągu tekstowego muszą być ujęte w cudzysłów.



UWAGA. Programiści czasami mówią, że literały są **na stałe osadzone** w kodzie programu, ponieważ wartość literału nie może ulec zmianie w trakcie działania programu.

Wiele przycisków wraz z procedurami obsługi zdarzeń

Przedstawiony wcześniej w rozdziale projekt *Code Demo* zawierał tylko jeden przycisk wraz z procedurą obsługi zdarzeń `Click`. Wiele opracowywanych przez Ciebie aplikacji będzie zawierało większą liczbę przycisków, każdy z własną procedurą obsługi zdarzeń `Click`. Przykładowo: formularz pokazany na rysunku 2.34 ma trzy kontrolki `Button`. Jak pokazałem na rysunku, te kontrolki zostały nazwane `firstButton`, `secondButton` i `thirdButton`.



Rysunek 2.34. Formularz wraz z kilkoma kontrolkami `Button`

Aby utworzyć procedury obsługi zdarzeń do tych przycisków, należy po prostu w oknie *Designer* dwukrotnie kliknąć kontrolkę `Button` każdego z nich. To spowoduje wygenerowanie w pliku kodu źródłowego formularza szkieletu procedury obsługi zdarzeń. Visual Studio użyje następujących nazw tych procedur: `firstButton_Click()`, `secondButton_Click()` i `thirdButton_Click()`. Na listingu 2.5 pokazałem przykład kodu źródłowego formularza po wygenerowaniu wszystkich trzech procedur obsługi zdarzeń i po umieszczeniu w nich wywołania metody `MessageBox.Show()`.

Listing 2.5. Kod źródłowy pliku `Form1.cs` wraz z trzema procedurami obsługi zdarzeń

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Multiple_Buttons
{
    public partial class Form1 : Form
    {
```

```

public Form1()
{
    InitializeComponent();
}

private void firstButton_Click(object sender, EventArgs e)
{
}

Procedura obsługi zdarzeń Click dla kontrolki.

private void secondButton_Click(object sender, EventArgs e)
{
}

Procedura obsługi zdarzeń Click dla kontrolki.

private void thirdButton_Click(object sender, EventArgs e)
{
}

Procedura obsługi zdarzeń Click dla kontrolki.
}

```

Czas projektowania i czas działania

Gdy masz otwarty projekt w Visual Studio, czas potrzebny na opracowanie graficznego interfejsu użytkownika aplikacji i jej kodu źródłowego jest określany mianem **czasu projektowania**. W trakcie tego czasu korzystasz z okien *Designer* i *Toolbox* do umieszczania kontrolki w formularzu, z okna *Properties* do przypisywania wartości różnym właściwościom, z okna edytora kodu źródłowego do tworzenia kodu aplikacji itd. To jest faza, w trakcie której tworzysz lub modyfikujesz program.

Gdy jesteś gotów do uruchomienia projektu otwartego w Visual Studio, naciskasz klawisz *F5* lub klikasz przycisk *Start Debugging* (🔍) na pasku narzędzi. To spowoduje skompilowanie i uruchomienie aplikacji, o ile nie zawiera żadnych błędów. Czas, w trakcie którego aplikacja jest uruchomiona, nosi nazwę **czasu działania**. W trakcie tej fazy używasz aplikacji, choć nie możesz korzystać z okien *Designer*, *Toolbox* i *Properties*, a także z edytora kodu źródłowego i innych narzędzi Visual Studio do wprowadzania w niej zmian.



UWAGA. W informatyce i internecie można się również spotkać z innymi określeniami **czasu działania**, np. **runtime**. Wszystkie te terminy mają takie samo znaczenie.



Punkt kontrolny

- 2.15. Jak się nazywa plik zawierający kod programu?
- 2.16. Co musisz zrobić, jeśli chcesz, aby aplikacja wykonywała jakiejkolwiek użyteczne operacje?

- 2.17. Co zawiera plik *Program.cs*?
- 2.18. Co zawiera plik *Form1.cs*?
- 2.19. W jaki sposób jest zorganizowany kod w języku C#?
- 2.20. Co to jest przestrzeń nazw?
- 2.21. Jakie znaki są używane przez kontener kodu — np. przestrzeń nazw, klasę lub metodę — do ujęcia kodu?
- 2.22. Jak można przechodzić między oknami *Designer* i edytora kodu źródłowego?
- 2.23. Jak można utworzyć procedurę obsługi zdarzeń do przycisku?
- 2.24. Co to jest zdarzenie *Click*?
- 2.25. Jaka metoda w Visual C# służy do wyświetlenia okna komunikatu?
- 2.26. Co to jest literal?
- 2.27. W jaki sposób jest ujmowany literal ciągu tekstowego?
- 2.28. Jak można uruchomić projekt aktualnie otwarty w Visual Studio?

2.4.

Utworzenie kodu aplikacji Hello World

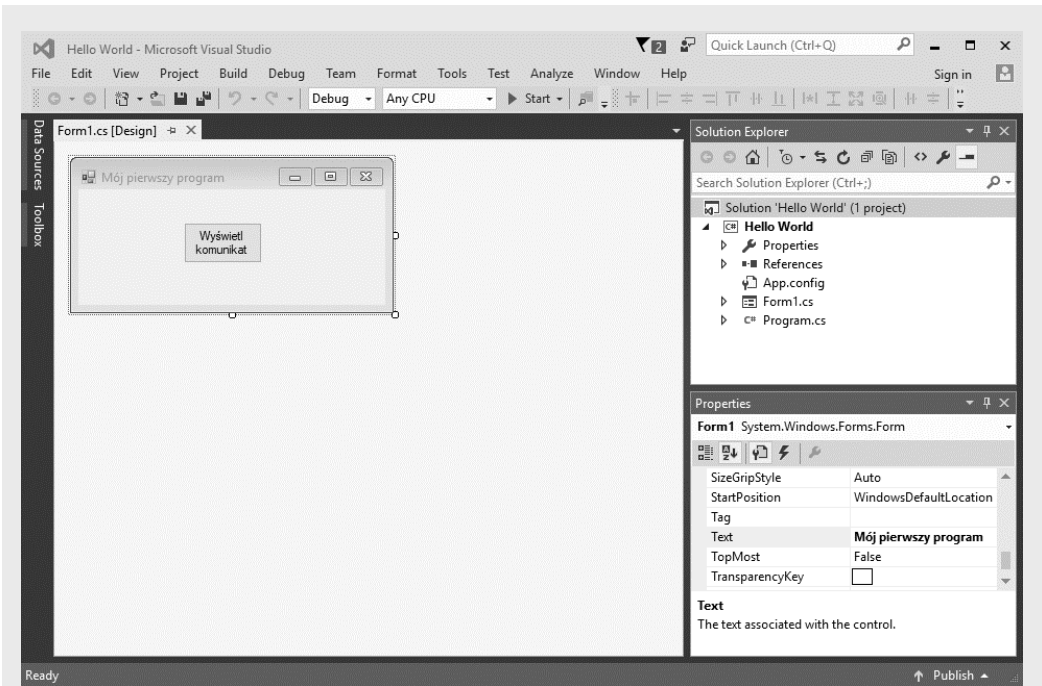
Masz już wszystko, co będzie niezbędne do ukończenia projektu *Hello World*. W przykładzie 2.2 otworzysz projekt i dodasz procedurę obsługi zdarzeń *Click* do kontrolki *messageButton*. Ta procedura wywoła metodę *MessageBox.Show()* odpowiedzialną za wyświetlenie okna komunikatu wraz z tekstem *Witaj, świecie!*.

Przykład 2.2.

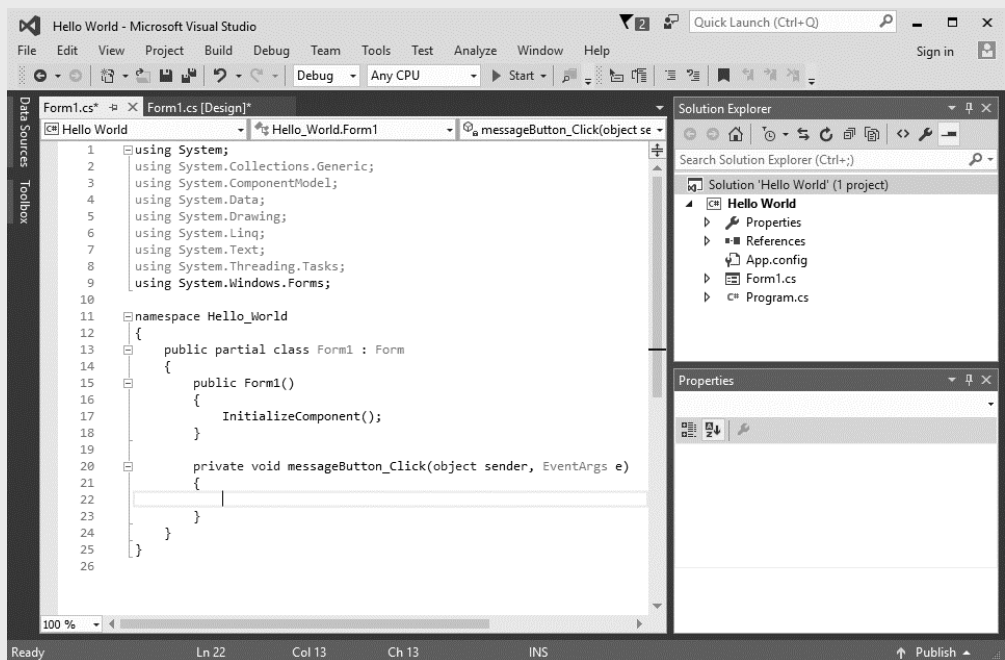
Utworzenie kodu aplikacji Hello World

- Krok 1.** Jeśli wcześniej tego nie zrobiłeś, uruchom Visual Studio, a następnie otwórz projekt *Hello World* utworzony w przykładzie 2.1 we wcześniejszej części rozdziału.
- Krok 2.** Upewnij się, że w oknie *Designer* jest wyświetlony formularz *Form1*, jak pokazałem na rysunku 2.35. Jeżeli nie, prawym przyciskiem myszy kliknij element *Form1.cs* w oknie *Solution Explorer*, a następnie z menu kontekstowego wybierz opcję *View Designer*.
- Krok 3.** W oknie *Designer* dwukrotnie kliknij kontrolkę *messageButton*. Na ekranie powinno się pojawić okno edytora kodu źródłowego, jak pokazałem na rysunku 2.36. Zauważ, że środowisko Visual Studio wygenerowało szkielet procedury obsługi zdarzeń o nazwie *messageButton_Click()*.
- Krok 4.** W wygenerowanej procedurze obsługi zdarzeń wpisz polecenie, tak jak pokazano tutaj:

```
MessageBox.Show("Witaj, świecie!");
```



Rysunek 2.35. Projekt Hello World wczytany w Visual Studio wraz z formularzem Form1 wyświetlonym w oknie Designer



Rysunek 2.36. Okno edytora kodu źródłowego wraz z pustą procedurą obsługi zdarzeń

Nie zapomnij o średniku na końcu polecenia. Po wprowadzeniu tej zmiany kod źródłowy procedury obsługi zdarzeń powinien wyglądać, jak pokazano na listingu 2.6.

Listing 2.6. Polecenie umieszczone w procedurze obsługi zdarzeń

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Hello_World
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            private void messageButton_Click(object sender, EventArgs e)
            {
                MessageBox.Show("Witaj, świecie!");
            }
        }
    }
}
```

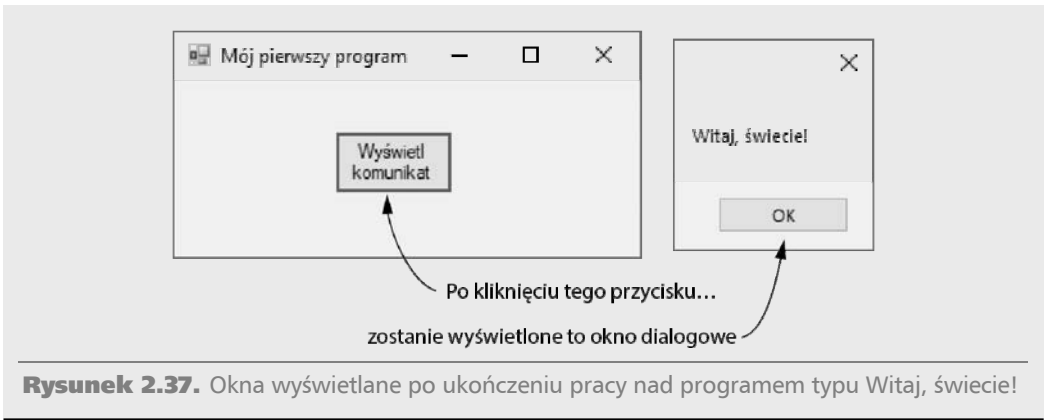
Krok 5. Zapisz projekt.

Krok 6. Naciśnij klawisz *F5* lub kliknij przycisk *Start Debugging* (🔍) na pasku narzędzi. To spowoduje skompilowanie i uruchomienie aplikacji.



UWAGA. Jeżeli w procedurze obsługi zdarzeń `messageButton_Click()` poprawnie wpisałeś polecenie (krok 4.), aplikacja powinna zostać uruchomiona. Natomiast w przypadku nieprawidłowego wpisania polecenia zostanie wyświetlone okno wraz z informacjami o błędach znalezionych podczas kompilacji programu. Jeśli tak się zdarzy, w wyświetlonym oknie kliknij przycisk *No*, a następnie popraw wpisane polecenie, aby wyglądało jak na listingu 2.7.

Po uruchomieniu aplikacji zostanie wyświetlony formularz pokazany po lewej stronie na rysunku 2.37. Kliknięcie przycisku *Wyświetl komunikat* spowoduje wyświetlenie okna komunikatu widocznego po prawej stronie na rysunku 2.37. Kliknięcie przycisku *OK* w oknie komunikatu powoduje jego zamknięcie.

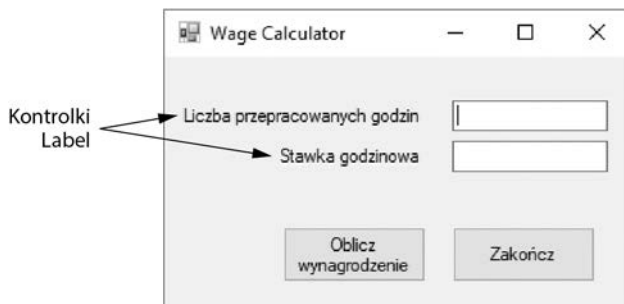


Rysunek 2.37. Okna wyświetlane po ukończeniu pracy nad programem typu Witaj, świecie!

2.5. Kontrolka Label

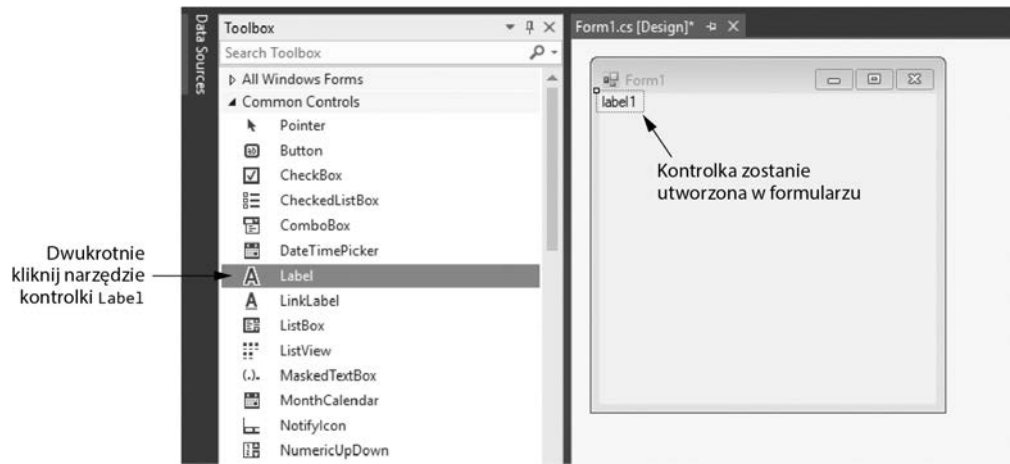
WYJAŚNIENIE. Kontrolka `Label` wyświetla tekst (etykietę) w formularzu. Ta kontrolka ma różne właściwości pozwalające na dostosowanie jej wyglądu do własnych potrzeb. Można ją wykorzystać do wyświetlania niezmiennego tekstu lub danych wyjściowych programu.

Gdy chcesz wyświetlić tekst w formularzu, użyj **kontrolki Label**. Na rysunku 2.38 pokazałem przykład formularza zawierającego dwie kontrolki `Label`. Po umieszczeniu kontrolki w formularzu, jej właściwość `Text` należy przypisać tekst przeznaczony do wyświetlenia. Przykładowo: na rysunku 2.38 wartością właściwości `Text` górnej kontrolki `Label` jest Liczba przepracowanych godzin, natomiast dolnej Stawka godzinowa.



Rysunek 2.38. Przykład formularza wraz z kontrolką `Label`

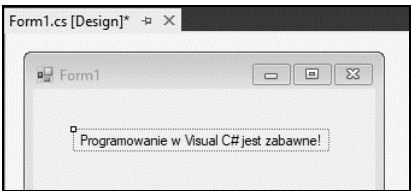
Narzędzie kontrolki `Label` znajdziesz w grupie *Common Controls* okna *Toolbox*, jak pokazałem na rysunku 2.39. Aby utworzyć tę kontrolkę w formularzu, dwukrotnie kliknij narzędzie `Label` w oknie *Toolbox*. (Alternatywne podejście polega na kliknięciu



Rysunek 2.39. Utworzenie kontrolki Label

i przeciągnięciu narzędzia kontrolki Label z okna *Toolbox* do formularza). Na rysunku 2.39 zwróć uwagę na ramkę ograniczającą wyświetloną wokół kontrolki i wskazującą na zaznaczenie tej kontrolki.

Po utworzeniu kontrolki Label automatycznie otrzymują one nazwy domyślne, takie jak `label1`, `label2` itd. Wartość właściwości `Text` kontrolki Label początkowo jest taka sama jak jej nazwa. Dlatego też po utworzeniu kontrolka Label będzie wyświetlała swoją nazwę, jak pokazałem w przykładzie na rysunku 2.39. Po jej zaznaczeniu w oknie *Designer* okno *Properties* można wykorzystać do zmiany wartości właściwości `Text`. Na rysunku 2.40 pokazałem kontrolkę Label po wprowadzeniu takiej zmiany — właściwości `Text` została przypisana wartość w postaci komunikatu Programowanie w Visual C# jest zabawne!.



Rysunek 2.40. Kontrolka Label wyświetlająca komunikat

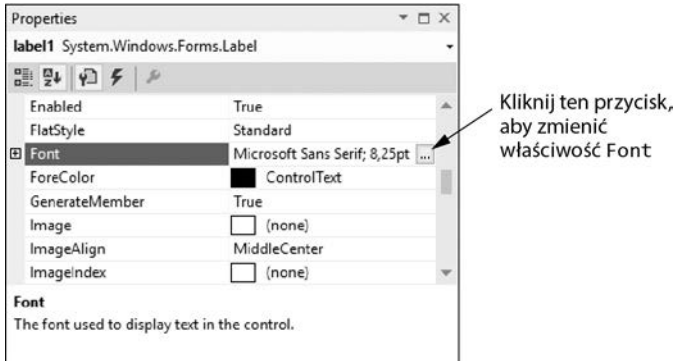
Istnieje również możliwość użycia okna *Properties* do zmiany nazwy kontrolki. Zawsze dobrym pomysłem jest zmiana nazwy kontrolki na znacznie bardziej opisową niż domyślnie wygenerowana przez Visual Studio.

Właściwość Font

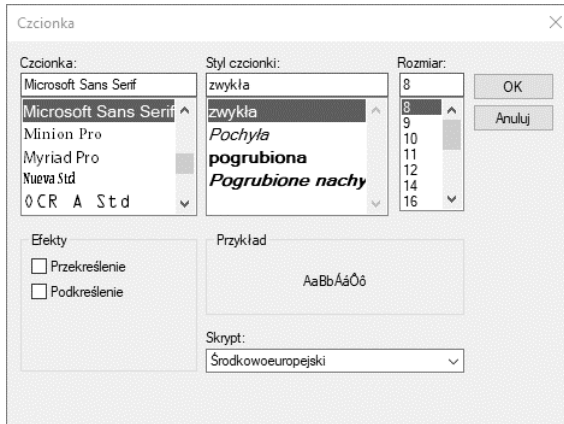
Jeżeli chcesz zmienić wygląd tekstu w kontrolce Label, możesz zmodyfikować jej właściwość **Font**. Pozwala to na zdefiniowanie czcionki, jej stylu i rozmiaru tekstu wyświetlanego przez tę kontrolkę. Po kliknięciu właściwości **Font** w oknie *Properties* zauwa-

żysz przycisk przedstawiający wielokropek (...) wyświetlony obok wartości właściwości, jak pokazałem na rysunku 2.41. Kliknięcie tego przycisku powoduje wyświetlenie okna dialogowego *Czcionka* pokazanego na rysunku 2.42. Wybierz czcionkę, styl i rozmiar, a następnie kliknij przycisk OK. Tekst wyświetlany przez kontrolkę zostanie uaktualniony z uwzględnieniem wybranych atrybutów. Przykładowo: na rysunku 2.43 pokazałem kontrolkę Label wraz z następującymi atrybutami czcionki:

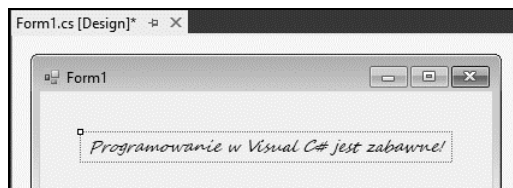
Czcionka: *Segoe Script*
 Styl czcionki: *pochyła*
 Rozmiar: *10 punktów*



Rysunek 2.41. Właściwość Font kontrolki Label



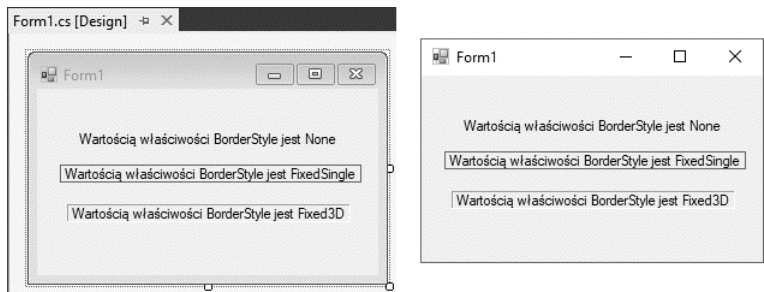
Rysunek 2.42. Okno dialogowe Czcionka



Rysunek 2.43. Wygląd etykiety zmieniony za pomocą atrybutów czcionki

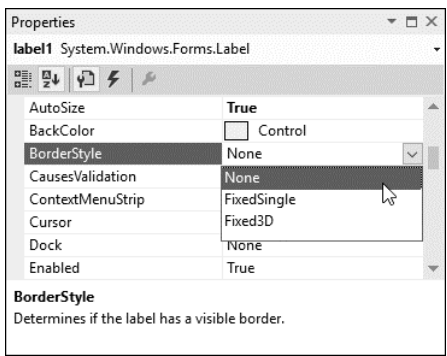
Właściwość **BorderStyle**

Kontrolka `Label` ma właściwość **BorderStyle** pozwalającą na wyświetlenie obramowania wokół tekstu czcionki. Tej właściwości można przypisać jedną z trzech wartości: `None`, `FixedSingle` i `Fixed3D`. Wartością domyślną tej właściwości jest `None`, co oznacza, że wokół tekstu kontrolki nie będzie wyświetlone żadne obramowanie. Przypisanie omawianej właściwości wartości `FixedSingle` powoduje wyświetlenie cienkiego obramowania wokół tekstu kontrolki. Natomiast przypisanie jej wartości `Fixed3D` powoduje nadanie kontrolce wyglądu 3D. Na rysunku 2.44 pokazałem przykłady kontrolki `Label` wraz z każdą wartością właściwości `BorderStyle`. Po lewej stronie rysunku widać wygląd formularza w oknie *Designer*, natomiast po prawej formularz w uruchomionej aplikacji.



Rysunek 2.44. Przykłady użycia poszczególnych wartości właściwości `BorderStyle`

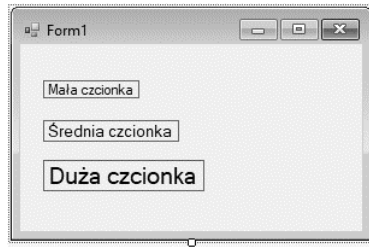
Aby zmienić właściwość `BorderStyle`, kliknij ją w oknie *Properties*, a następnie kliknij przycisk rozwijanej listy () znajdujący się obok wartości właściwości. Jak pokazałem na rysunku 2.45, rozwijana lista zawiera trzy możliwe wartości, które można przypisać omawianej właściwości. Po wybraniu żądanej wartości tekst kontrolki zostanie uaktualniony.



Rysunek 2.45. Rozwijana lista wartości właściwości `BorderStyle`

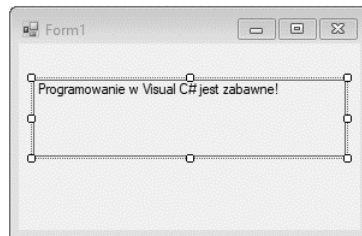
Właściwość AutoSize

Kontrolka Label ma właściwość **AutoSize** określającą sposób zmiany jej wielkości. To jest właściwość **boolowska**, co oznacza, że może mieć przypisaną jedną z dwóch wartości: **True** lub **False**. Domyślnie wartością właściwości **AutoSize** kontrolki Label jest **True**, więc kontrolka automatycznie zmienia wielkość, aby pomieścić tekst przeznaczony do wyświetlenia. Przykładowo: spójrz na trzy kontrolki Label pokazane na rysunku 2.46. Każda z nich wyświetla inną ilość tekstu o odmiennej wielkości. Ponieważ właściwość **BorderStyle** ma przypisaną wartość **FixedSingle**, możesz wyraźnie zobaczyć, że każda kontrolka jest na tyle duża, aby pomieścić tekst przeznaczony do wyświetlenia.



Rysunek 2.46. Kontrolki Label, w których właściwość **AutoSize** ma przypisaną wartość **True**

Gdy właściwość **AutoSize** kontrolki ma przypisaną wartość **True**, nie można ręcznie zmienić jej wielkości przez kliknięcie i przeciągnięcie ramki ograniczającej. Jeżeli chcesz zachować możliwość ręcznej zmiany wielkości kontrolki Label, jej właściwość **AutoSize** musi mieć wartość **False**. Wówczas wokół kontrolki będą wyświetlane uchwyty zmiany wielkości pozwalające na kliknięcie i przeciągnięcie ramki ograniczającej. Przykład pokazałem na rysunku 2.47. Tutaj kontrolka Label jest znacznie większa niż tekst przeznaczony do wyświetlenia.



Rysunek 2.47. Kontrolka Label wraz z właściwością **AutoSize** o wartości **False**



UWAGA. Gdy właściwość **AutoSize** kontrolki Label ma przypisaną wartość **True**, tekst będzie wyświetlany zawsze w jednym wierszu. Natomiast wartość **False** właściwości **AutoSize** oznacza możliwość umieszczenia tekstu w wielu wierszach, jeśli nie mieści się w jednym.

Właściwość TextAlign

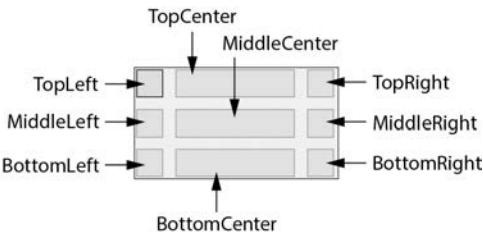
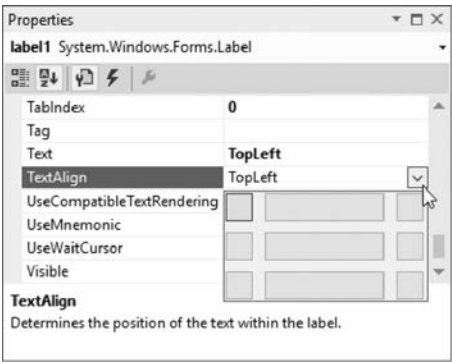
Po przypisaniu wartości `False` właściwości `AutoSize` kontrolki `Label`, a następnie ręcznej zmianie wielkości kontrolki czasami konieczna okazuje się zmiana sposobu wyrównania wyświetlanego tekstu. Domyślnie tekst wyświetlany przez kontrolkę `Label` jest wyrównany do górnej i lewej krawędzi ramki ograniczającej kontrolkę. Spójrz na kontrolki `Label` pokazane na rysunku 2.47. Zwróć uwagę na umieszczenie tekstu w lewym górnym rogu kontrolki.

Co zrobić w sytuacji, gdy tekst ma zostać wyrównany inaczej? Przykładowo: co można zrobić, aby umieścić tekst na środku kontrolki lub w prawym dolnym rogu? Do zmiany sposobu wyrównania tekst wyświetlanego przez kontrolkę `Label` służy właściwość **`TextAlign`**. Tej właściwości można przypisać jedną z następujących wartości: `TopLeft`, `TopCenter`, `TopRight`, `MiddleLeft`, `MiddleCenter`, `MiddleRight`, `BottomLeft`, `BottomCenter` lub `BottomRight`. Na rysunku 2.48 pokazałem dziewięć kontroltek `Label`, każda z nich ma inną wartość `TextAlign`.



Rysunek 2.48. Wyrównanie tekstu w kontrolce `Label`

Aby zmienić wartość właściwości `TextAlign`, kliknij ją w oknie *Properties*, a następnie kliknij przycisk rozwijanej listy (▼) znajdujący się obok wartości właściwości. To spowoduje wyświetlenie na ekranie okna dialogowego wraz z dziewięcioma przyciskami, jak pokazałem po lewej stronie na rysunku 2.49. Po prawej stronie tego rysunku możesz zobaczyć dziewięć przycisków przedstawiających poprawne wartości właściwości `TextAlign`.



Rysunek 2.49. Przypisanie wartości właściwości `TextAlign`

Użycie kodu do wyświetlenia danych wyjściowych w kontrolce Label

Poza wyświetlaniem niezmiennego się tekstu w formularzu kontrolki Label okazują się użyteczne również do wyświetlania danych wyjściowych podczas działania aplikacji. Przykładowo: przyjmując założenie o utworzeniu aplikacji przeprowadzającej pewne obliczenia, których wynik ma zostać wyświetlony w konkretnym miejscu formularza. W takim przypadku idealnie sprawdza się zastosowanie kontrolki Label. Oto ogólne kroki, które należy wykonać.

- Krok 1.** Podczas tworzenia graficznego interfejsu użytkownika aplikacji należy umieścić kontrolkę Label w tym miejscu formularza, w którym ma być wyświetlony wynik. Następnie w oknie *Properties* trzeba usunąć zawartość właściwości Text kontrolki. Ponieważ właściwość Text jest pusta, więc po uruchomieniu aplikacji ta kontrolka Label niczego nie wyświetla.
- Krok 2.** W kodzie źródłowym aplikacji należy umieścić polecenia niezbędne do przeprowadzenia obliczeń i umieszczenia ich wyniku we właściwości Text kontrolki Label. Takie rozwiązanie spowoduje wyświetlenie wyniku w kontrolce Label znajdującej się w formularzu.



UWAGA. Obliczeniami będę się zajmował dopiero w rozdziale 3., więc teraz przedstawię jedynie te przykłady, które umieszczają w kontrolkach Label dane inne niż matematyczne.

Do umieszczenia wartości we właściwości kontrolki w kodzie można użyć **operatora przypisania**. Przykładowo: przyjmując założenie o utworzeniu kontrolki Label o nazwie outputLabel. Przedstawione tutaj polecenie spowoduje umieszczenie w jej właściwości Text wartości Dziękujemy bardzo!:

```
outputLabel.Text = "Dziękujemy bardzo!";
```

Znak równości jest nazywany **operatorem przypisania**. Powoduje przypisanie wartości znajdującej się po prawej stronie operatora elementowi umieszczonego po lewej stronie operatora. W omawianym przykładzie elementem po lewej stronie operatora przypisania jest wyrażenie outputLabel.Text. To jest właściwość Text kontrolki outputLabel. Z kolei wartością po prawej stronie operatora przypisania jest ciąg tekstowy Dziękujemy bardzo! Po wykonaniu omawianego polecenia ten ciąg tekstowy zostanie wyświetlony w kontrolce Label.



OSTRZEŻENIE! Podczas tworzenia poleceń przypisania pamiętaj, że element otrzymujący wartość musi się znajdować po lewej stronie operatora przypisania. Dlatego też następujące polecenie jest nieprawidłowe i spowoduje wygenerowanie błędu w trakcie kompilacji programu:

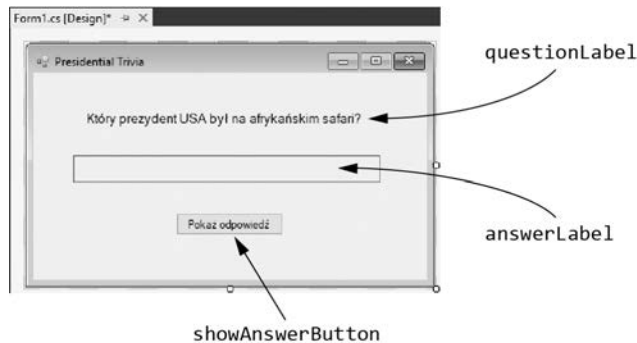
```
"Dziękujemy bardzo!" = outputLabel.Text; <- BŁĄD!
```



UWAGA. Standardowy zapis odwołania się do właściwości kontrolki w kodzie przedstawia się następująco:

NazwaKontrolki.NazwaWłaściwości

Przechodzę do przykładu aplikacji używającej kontrolki `Label` do wyświetlenia danych wyjściowych. Upewnij się, że pobrałeś z <ftp://ftp.helion.pl/przyklady/viczp2.zip> materiały przygotowane do książki. W katalogu `Rozdzial02` znajdziesz projekt o nazwie *Presidential Trivia*. Celem tego programu jest wyświetlenie łatwego pytania dotyczącego byłego prezydenta USA. Gdy użytkownik kliknie przycisk, odpowiedź na pytanie zostanie wyświetlona w formularzu. Formularz tego projektu pokazałem na rysunku 2.50.



Rysunek 2.50. Formularz programu *Presidential Trivia*

Jak pokazałem na rysunku, formularz zawiera trzy wymienione tutaj kontrolki:

- Kontrolkę `Label` o nazwie `questionLabel`. W tej kontrolce zostanie wyświetlone pytanie.
- Kontrolkę `Label` o nazwie `answerLabel`. Ta kontrolka na początku jest pusta, ale zostanie użyta do wyświetlenia odpowiedzi na pytanie.
- Kontrolkę `Button` o nazwie `showAnswerButton`. Po kliknięciu przycisku przez użytkownika w formularzu zostanie wyświetlona odpowiedź na pytanie.

W tabeli 2.2 wymieniałem ustawienia właściwości poszczególnych kontroltek, na które powinieneś zwrócić uwagę.

Jeżeli plik `Form1.cs` wyświetlisz w oknie edytora kodu źródłowego, zobaczysz kod pokazany na listingu 2.7. (Aby wyświetlić plik w oknie edytora kodu źródłowego, prawym przyciskiem myszy kliknij element `Form1.cs` w oknie *Solution Explorer*, a następnie z menu kontekstowego wybierz opcję *View Code*). Zwróć uwagę na metodę o nazwie `showAnswerButton_Click()`.

Listing 2.7. Kod w pliku `Form1.cs` aplikacji *Presidential Trivia*

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
```


Tabela 2.2. Ustawienia właściwości kontroltek

Nazwa kontrolki	Typ kontrolki	Ustawienia właściwości
questionLabel	Label	AutoSize: <i>False</i> BorderStyle: <i>None</i> Font: <i>Microsoft Sans Serif</i> (styl: zwykła, rozmiar: 10 punktów). Text: <i>Który prezydent USA był na afrykańskim safari?</i> TextAlign: <i>MiddleCenter</i>
answerLabel	Label	AutoSize: <i>False</i> BorderStyle: <i>FixedSingle</i> Font: <i>Microsoft Sans Serif</i> (styl: pogrubiona, rozmiar: 10 punktów). Text: (zawartość właściwości Text została usunięta). TextAlign: <i>MiddleCenter</i>
showAnswerButton	Button	Size: <i>110; 23</i> Text: <i>Pokaż odpowiedź</i>

```
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Presidential_Trivia
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

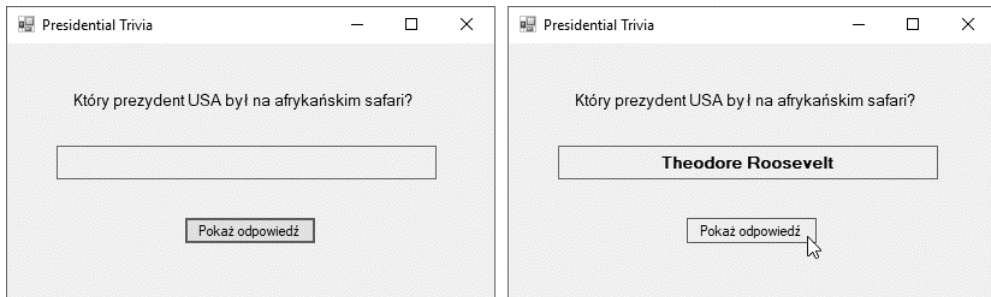
            private void showAnswerButton_Click(object sender, EventArgs e)
            {
                answerLabel.Text = "Theodore Roosevelt";
            }
        }
    }
}
```

To jest procedura obsługi zdarzeń kontrolki showAnswerButton i zawiera przedstawione tutaj polecenie:

```
answerLabel.Text = "Theodore Roosevelt";
```

Po wykonaniu tego polecenia ciąg tekstowy Theodore Roosevelt zostanie przypisany właściwości Text kontrolki answerLabel. W wyniku jego wykonania kontrolka Label wyświetli tekst Theodore Roosevelt.

Po uruchomieniu aplikacji zostanie wyświetlony formularz pokazany po lewej stronie na rysunku 2.51. Kliknij przycisk *Pokaż odpowiedź*, a wyświetli się odpowiedź na zadane pytanie, jak pokazałem po prawej stronie na rysunku 2.51.



Rysunek 2.51. Uruchomiona aplikacja Presidential Trivia

Właściwość Text akceptuje jedynie ciąg tekstowy

W tym miejscu trzeba koniecznie dodać, że właściwość `Text` kontrolki `Label` akceptuje jedynie wartość w postaci ciągu tekstowego. Dlatego też nie można jej przypisać liczby. Przykładowo: przyjmuję założenie, że aplikacja zawiera kontrolkę `Label` o nazwie `resultLabel`. Kolejne polecenie spowoduje wygenerowanie błędu, ponieważ tutaj mamy do czynienia z próbą przypisania liczby 5 właściwości `Text` kontrolki `resultLabel`:

```
resultLabel.Text = 5; <-- BŁĄD!
```

To oczywiście nie oznacza braku możliwości wyświetlania liczb w etykietach. Jeżeli liczba zostanie ujęta w cudzysłów, automatycznie staje się ciągiem tekstowym. Dlatego też kolejne polecenie działa zgodnie z oczekiwaniami:

```
resultLabel.Text = "5";
```

Usunięcie zawartości etykiety

Jeżeli w kodzie chcesz usunąć tekst wyświetlany przez kontrolkę `Label`, wystarczy właściwości `Text` tej kontrolki przypisać pusty ciąg tekstowy (`""`), jak pokazałem w następującym poleceniu:

```
resultLabel.Text = "";
```

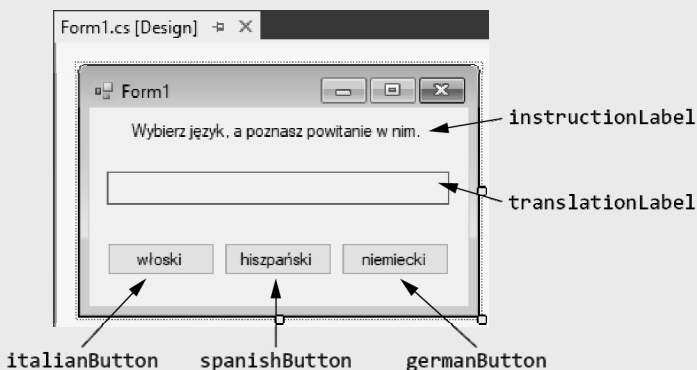
W przykładzie 2.3 widać sposób użycia wybranych właściwości kontrolki `Label`, które omówiłem w tym podrozdziale.

Przykład 2.3.

Utworzenie aplikacji Language Translator

W tym przykładzie utworzysz aplikację wyświetlającą w różnych językach powitanie Dzień dobry!. Formularz programu będzie zawierał trzy przyciski, po jednym dla języków włoskiego, hiszpańskiego i niemieckiego. Gdy użytkownik kliknie dowolny z tych przycisków, przetłumaczony komunikat powitania zostanie wyświetlony w kontrolce Label.

- Krok 1.** Uruchom Visual Studio i utwórz nowy projekt *Windows Forms Application* o nazwie *Language Translator*.
- Krok 2.** Przygotuj formularz aplikacji, jak pokazałem na rysunku 2.52. Zwróć uwagę na to, że wartością właściwości *Text* formularza jest *Language Translator*. Ten formularz zawiera dwie kontrolki *Label* i trzy *Button*. Nazwy kontrolki zostały pokazane na rysunku. Po umieszczeniu poszczególnych kontrolki w formularzu zajrzyj do tabeli 2.3, w której przedstawiłem ustawienia właściwości tych kontrolki.



Rysunek 2.52. Formularz aplikacji Language Translator

- Krok 3.** Po przygotowaniu formularza wraz z kontrolkami należy utworzyć procedury obsługi zdarzeń *Click* kontrolki *Button*. W oknie *Designer* dwukrotnie kliknij kontrolkę *italianButton*. To spowoduje wyświetlenie okna edytora kodu źródłowego, w którym zobaczysz szkielet procedury obsługi zdarzeń o nazwie *italianButton_Click()*. Umieść w niej następujące polecenie:

```
translationLabel.Text = "Buongiorno";
```

- Krok 4.** Powróć do okna *Designer* i dwukrotnie kliknij kontrolkę *spanishButton*. W oknie edytora kodu źródłowego zobaczysz szkielet procedury obsługi zdarzeń o nazwie *spanishButton_Click()*. Umieść w niej następujące polecenie:

```
translationLabel.Text = "Buenos dias";
```

Tabela 2.3. Ustawienia właściwości kontroltek

Nazwa kontrolki	Typ kontrolki	Ustawienia właściwości
instructionLabel	Label	Text: <i>Wybierz język, a poznasz powitanie w nim.</i>
translationLabel	Label	AutoSize: <i>False</i> BorderStyle: <i>FixedSingle</i> Font: <i>Microsoft Sans Serif</i> (styl: pogrubiona, rozmiar: 10 punktów). Text: (zawartość właściwości Text została usunięta). TextAlign: <i>MiddleCenter</i>
italianButton	Button	Text: <i>włoski</i>
spanishButton	Button	Text: <i>hiszpański</i>
germanButton	Button	Text: <i>niemiecki</i>

Krok 5. Powróć do okna *Designer* i dwukrotnie kliknij kontrolkę *germanButton*. W oknie edytora kodu źródłowego zobaczysz szkielet procedury obsługi zdarzeń o nazwie *germanButton_Click()*. Umieść w niej następujące polecenie:

```
translationLabel.Text = "Guten Morgen";
```

Krok 6. Obecną postać kodu źródłowego przedstawiłem na listingu 2.8. Numery wierszy nie stanowią fragmentu kodu źródłowego. Zdecydowałem się na ich użycie, ponieważ dzięki tym numerom mogę w tekście łatwiej odwoływać się do poszczególnych poleceń. Polecenia przedstawione pogrubioną czcionką to te, które zostały wpisane przez Ciebie. Upewnij się, że kod źródłowy ma postać taką jak na listingu 2.8. Nie zapomnij o średnikach na końcu poleceń!

Listing 2.8. Ukończony kod źródłowy aplikacji *Language Translator*

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9 using System.Windows.Forms;
10
11 namespace Language_Translator
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         private void italianButton_Click(object sender, EventArgs e)
21         {
```

```

22         translationLabel.Text = "Buongiorno";
23     }
24
25     private void spanishButton_Click(object sender, EventArgs e)
26     {
27         translationLabel.Text = "Buenos dias";
28     }
29
30     private void germanButton_Click(object sender, EventArgs e)
31     {
32         translationLabel.Text = "Guten Morgen";
33     }
34 }
35 }

```

Krok 7. Zapisz projekt, a następnie naciśnij klawisz *F5* lub kliknij przycisk *Start Debugging* (🔍) na pasku narzędzi. To spowoduje skompilowanie i uruchomienie aplikacji.



UWAGA. Jeżeli nie popełniłeś błędu w kodzie procedur obsługi zdarzeń, aplikacja powinna zostać uruchomiona. Natomiast w przypadku nieprawidłowych poleceń w procedurach obsługi zdarzeń zostanie wyświetlone okno wraz z informacjami o błędach w trakcie kompilacji. Jeśli tak się zdarzy, w wyświetlonym oknie kliknij przycisk *No*, a następnie popraw kod źródłowy, aby miał postać przedstawioną na listingu 2.9.

Na rysunku 2.53 pokazałem formularz aplikacji po jej uruchomieniu i kliknięciu każdej z kontrolek *Button*. Po przetestowaniu wszystkich przycisków możesz zamknąć formularz aplikacji.



Rysunek 2.53. Uruchomiona aplikacja Language Translator



Punkt kontrolny

- 2.29. W której grupie okna *Toolbox* znajduje się narzędzie kontrolki `Label`?
- 2.30. Która właściwość, po umieszczeniu kontrolki `Label` w formularzu, pozwala na zdefiniowanie wyświetlanego przez nią tekstu?
- 2.31. Która właściwość pozwala na zmianę wyglądu tekstu wyświetlanego przez kontrolkę `Label`?
- 2.32. Jaka jest wartość domyślna właściwości `BorderStyle` kontrolki `Label`?
- 2.33. Jak w oknie *Properties* można zmienić właściwość `BorderStyle` kontrolki?
- 2.34. Która właściwość określa, czy można zmienić wielkość etykiety?
- 2.35. Która właściwość określa sposób wyrównania tekstu w kontrolce `Label`?
- 2.36. Jak można użyć kontrolki `Label` do wyświetlania danych wyjściowych działającego programu?
- 2.37. Co się stanie, jeśli w kodzie źródłowym przypiszesz pusty ciąg tekstowy właściwości `Text` kontrolki?

2.6.

Poznanie listy IntelliSense

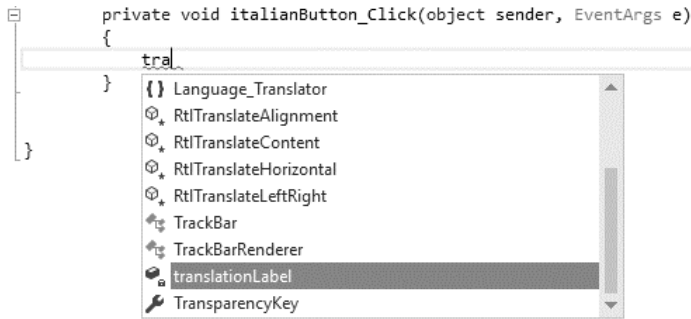
WYJAŚNIENIE. Podczas wpisywania kodu w edytorze Visual Studio wyświetlana jest lista IntelliSense mająca pomóc programiście w trakcie tworzenia kodu źródłowego. Tę listę możesz wykorzystać do automatycznego uzupełniania pewnych poleceń po wpisaniu zaledwie kilku pierwszych znaków.

Lista **IntelliSense** to funkcja Visual Studio oferująca automatyczne uzupełnianie kodu podczas pisania poleceń. Gdy tylko nauczysz się korzystać z tej listy, będziesz mógł znacznie szybciej tworzyć kod źródłowy. Jeżeli wykonałeś przykłady przedstawione wcześniej w rozdziale, miałeś okazję zobaczyć listę IntelliSense w działaniu. I tak, w kroku 3. przykładu 2.3 miałeś za zadanie wpisać następujące polecenie w procedurze obsługi zdarzeń `italianButton_Click()`:

```
translationLabel.Text = "Buongiorno";
```

Czy zauważyłeś, że po rozpoczęciu wpisywania tego polecenia na ekranie pojawiło się nowe okno? Jest ono nazywane listą IntelliSense. Zawartość tego okna zmienia się podczas wpisywania kolejnych znaków. Na rysunku 2.54 pokazałem okno listy IntelliSense po wpisaniu znaków `tra`.

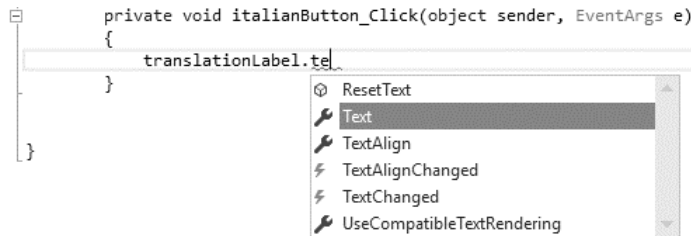
System IntelliSense przewiduje to, co chcesz wpisać, a po wpisaniu kolejnych znaków zawartość listy IntelliSense zmniejsza się i dopasowuje do już podanych znaków. Na rysunku 2.54 pokazałem listę IntelliSense zawierającą wszystkie elementy rozpoczynające się od znaków `tra` i będące opcjami dla wpisywanego polecenia. Zwróć uwagę



Rysunek 2.54. Wyświetlone okno listy IntelliSense

na zaznaczenie elementu `translationLabel` na liście. Po jego zaznaczeniu można naciśnąć klawisz `Tab`, a wpisane wcześniej znaki `tra` zostaną zastąpione przez `translationLabel`.

Teraz po wpisaniu kropki lista IntelliSense będzie zawierała nazwę wszystkich właściwości i metod obsługiwanych przez kontrolkę `translationLabel`. Wpisz `te`, a zostanie zaznaczona właściwość `Text`, jak pokazałem na rysunku 2.55. Naciśnięcie klawisza `Tab` po zaznaczeniu właściwości `Text` powoduje, że polecenie automatycznie przybiera postać `translationLabel.Text`. Teraz możesz kontynuować wpisywanie polecenia do końca.



Rysunek 2.55. Lista IntelliSense po wpisaniu `.te`

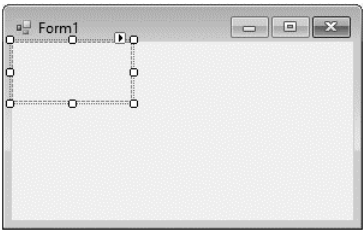
Skoro poznałeś sposób działania listy IntelliSense, zachęcam Cię do eksperymentowania z nią podczas tworzenia kodu w przyszłych projektach. Przy odrobinie praktyki praca z listą IntelliSense okazuje się bardzo intuicyjna.

2.7. Kontrolka PictureBox

WYJAŚNIENIE. Kontrolka `PictureBox` wyświetla obraz w formularzu. Ta kontrolka oferuje właściwości przeznaczone do sprawdzania sposobu, w jaki zostanie wyświetlony obraz. Omawiana kontrolka wywołuje procedurę obsługi zdarzeń `Click` wywoływaną po kliknięciu kontrolki w trakcie działania programu.

Kontrolkę PictureBox można wykorzystać do wyświetlenia obrazu w formularzu. Do obsługiwanych przez nią formatów zaliczamy bitmapę, GIF, JPEG, metafile i formaty ikon.

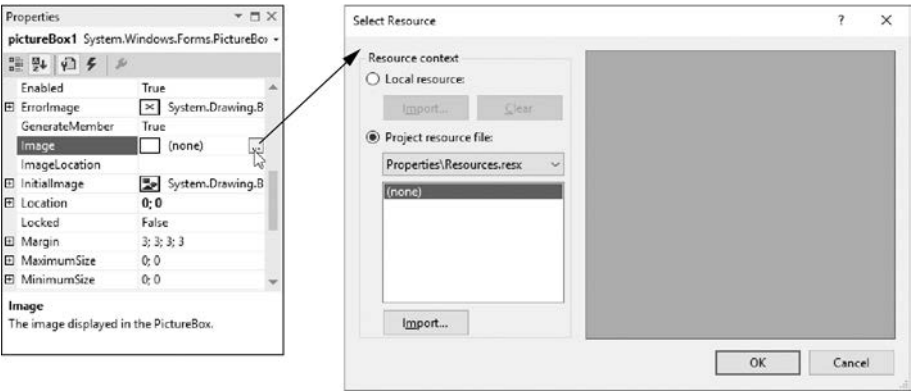
W oknie *Toolbox* narzędzie PictureBox znajduje się w grupie *Common Controls*. Dwukrotne kliknięcie tego narzędzia powoduje utworzenie w formularzu pustej kontrolki PictureBox, jak pokazałem na rysunku 2.56. Wprawdzie kontrolka jeszcze nie wyświetla obrazu, ale ma ramkę ograniczającą pokazującą jej wielkość i położenie. Ponadto ramka ograniczająca zawiera uchwyty pozwalające na zmianę wielkości kontrolki. Po utworzeniu kontrolki PictureBox automatycznie otrzymują one nazwy domyślne w postaci pictureBox1, pictureBox2 itd. Zawsze powinieneś zmienić domyślną nazwę kontrolki na lepiej przedstawiającą jej przeznaczenie.



Rysunek 2.56. Pusta kontrolka PictureBox w formularzu

Po utworzeniu kontrolki PictureBox **właściwość Image** wykorzystujesz do wskazania wyświetlanego przez nią obrazu. Wykonaj omówione niżej kroki.

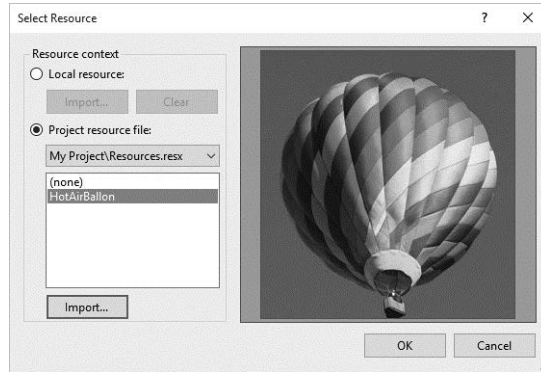
Krok 1. W oknie *Properties* kliknij właściwość Image. Pojawi się przycisk przedstawiający wielokropek (...), jak pokazałem po lewej stronie na rysunku 2.57.



Rysunek 2.57. Okno Select Resource właściwości Image

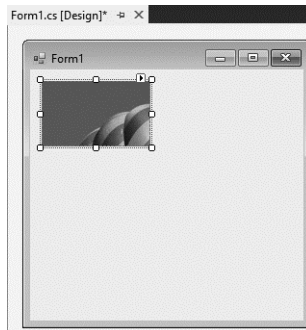
Krok 2. Kliknij przycisk przedstawiający wielokropek, co spowoduje wyświetlenie na ekranie okna *Select Resource*, jak pokazałem po prawej stronie na rysunku 2.57.

- Krok 3.** W oknie *Select Resource* kliknij przycisk *Image*. Na ekranie zostanie wyświetlone okno dialogowe *Open*. Wykorzystaj je do odszukania i wybrania pliku przeznaczonego do wyświetlenia.
- Krok 4.** Po zaznaczeniu obrazu zostanie on wyświetlony w oknie *Select Resources*. To wskazuje na zaimportowanie go do projektu. Na rysunku 2.58 pokazałem przykład tego okna po wybraniu i zaimportowaniu obrazu.



Rysunek 2.58. Wybrany obraz został zaimportowany

- Krok 5.** Kliknij przycisk *OK* w oknie *Select Resource*, a wybrany obraz pojawi się w kontrolce *PictureBox*. Przykład pokazałem na rysunku 2.59. W zależności od wielkości obrazu możesz zobaczyć wyświetloną tylko jego część. Tak się stało w omawianym przykładzie, ponieważ wybrany obraz jest większy niż kontrolka *PictureBox*. Kolejnym krokiem jest więc przypisanie wartości właściwości *SizeMode* i dopasowanie wielkości kontrolki.



Rysunek 2.59. Obraz wyświetlony przez kontrolkę *PictureBox*

Właściwość *SizeMode*

Właściwość *SizeMode* kontrolki *PictureBox* określa sposób wyświetlenia w niej obrazu. Omawianej właściwości można przypisać jedną z następujących wartości:

- **Normal**

To jest wartość domyślna. Obraz zostanie umieszczony w lewym górnym rogu kontrolki PictureBox. Jeżeli obraz jest zbyt duży, aby się zmieścił w kontrolce, wówczas będzie przycięty.

- **StretchImage**

Ta właściwość zmienia wielkość obrazu w poziomie i pionie, aby zmieścił się w kontrolce PictureBox. Jeżeli obraz w jednym kierunku zmieni wielkość bardziej niż w drugim, wówczas będzie wydawał się rozciągnięty.

- **AutoSize**

Przypisanie tej wartości kontrolce PictureBox powoduje automatyczną zmianę wielkości kontrolki, aby pomieściła obraz.

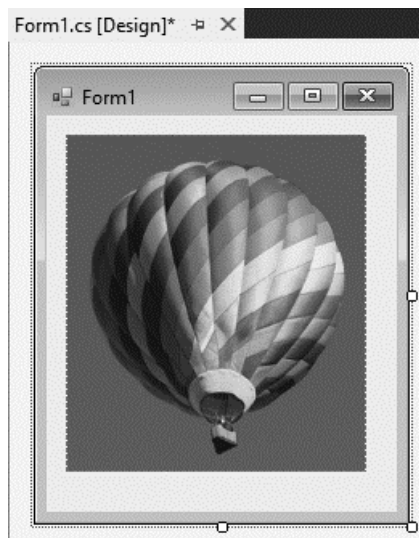
- **CenterImage**

Ta wartość powoduje wyśrodkowanie obrazu w kontrolce PictureBox bez zmiany jego wielkości.

- **Zoom**

Ta wartość powoduje zmianę wielkości obrazu jednakowo w obu kierunkach w taki sposób, aby zmieścił się w kontrolce PictureBox i jednocześnie nie zmienił pierwotnych proporcji. (**Proporcje** to współczynnik obliczony jako iloraz długości obrazu przez jego wysokość). To powoduje zmianę wielkości obrazu, który po tej operacji nie wygląda na rozciągnięty.

Na rysunku 2.60 pokazałem przykład obrazu wyświetlonego przez kontrolkę PictureBox. Właściwość `SizeMode` tej kontrolki ma przypisaną wartość `Zoom`, więc obraz zmienił wielkość i nie wygląda przy tym na rozciągnięty.



Rysunek 2.60. Obraz po zmianie jego wielkości za pomocą wartości `Zoom` przypisanej właściwości `SizeMode`



UWAGA. Kontrolka PictureBox również ma właściwość `BorderStyle` działającą w taki sam sposób jak w kontrolce `Label`.

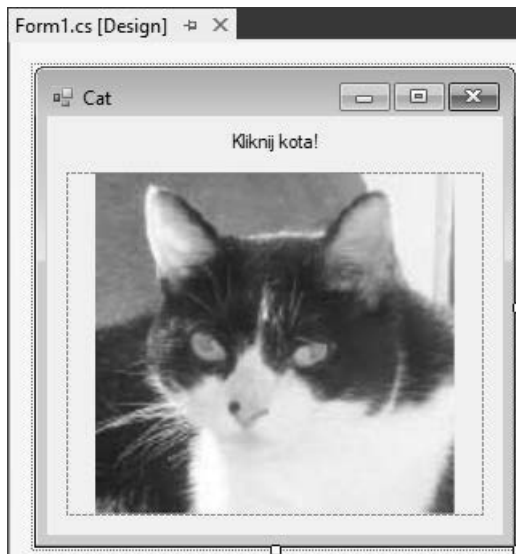
Utworzenie obrazu, który można kliknąć

Przyciski to nie jedyne kontrolki reagujące na zdarzenia `Click`. Kontrolka `PictureBox` również ma taką możliwość. To oznacza, że aplikacja może wyświetlić obraz i wykonać pewne zadania po jego kliknięciu przez użytkownika.

Aby obraz można było kliknąć, należy po prostu utworzyć procedurę obsługi zdarzeń `Click` kontrolki `PictureBox` wyświetlającej ten obraz. Ta procedura obsługi zdarzeń `Click` jest tworzona w taki sam sposób jak w przypadku kontrolki `Button`:

- Dwukrotnie kliknij kontrolkę `PictureBox` w oknie *Designer*. To spowoduje utworzenie szkieletu procedury obsługi zdarzeń `Click` w pliku kodu źródłowego formularza.
- W oknie edytora kodu umieść w procedurze obsługi zdarzeń te polecenia, które mają być wykonywane po kliknięciu obrazu.

Przykładowo: spójrz na projekt *Cat* umieszczony w materiałach przygotowanych do książki. Na rysunku 2.61 pokazałem formularz tej aplikacji. Nazwą kontrolki `PictureBox` jest `catPictureBox`. Wyświetla ona plik o nazwie *Cat.jpg*, który również znajduje się w materiałach przygotowanych do książki. Wartością właściwości `SizeMode` jest `Zoom`, natomiast wartością właściwości `BorderStyle` jest `FixedSingle`.



Rysunek 2.61. Formularz w projekcie *Cat*

W oknie edytora kodu otwórz plik *Form1.cs*, a zobaczysz już zdefiniowaną procedurę obsługi zdarzeń Click kontrolki PictureBox, jak pokazałem na listingu 2.9. Po uruchomieniu aplikacji i kliknięciu obrazu na ekranie zostanie wyświetlone okno dialogowe wraz z komunikatem *Miau*.

Listing 2.9. Kod pliku *Form1.cs* w projekcie *Cat*

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Cat
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void catPictureBox_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Miau");
        }
    }
}
```

Procedura obsługi zdarzeń Click dla kontrolki PictureBox.

Przykład 2.4 daje Ci możliwość przećwiczenia sposobu pracy z kontrolką PictureBox. W tym przykładzie utworzysz aplikację wraz z trzema możliwymi do kliknięcia kontrolkami PictureBox wyświetlającymi obrazy, które znajdziesz w materiałach przygotowanych do książki.

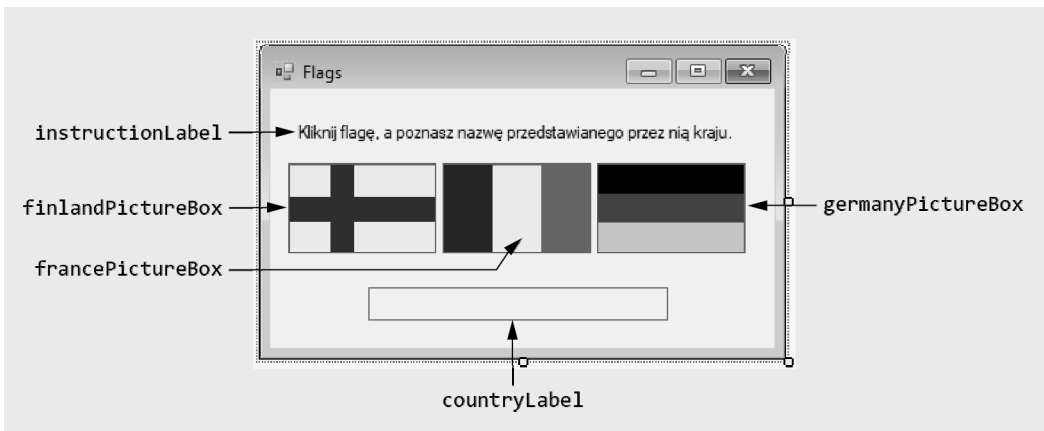


Przykład 2.4.

Utworzenie aplikacji Flags

W tym przykładzie utworzysz aplikację wyświetlającą w kontrolkach PictureBox flagi trzech państw: Finlandii, Francji i Niemiec. Po kliknięciu wybranej flagi etykieta wyświetli nazwę kraju wskazywanego przez tę flagę.

- Krok 1.** Uruchom Visual Studio i utwórz nowy projekt *Windows Forms Application* o nazwie *Flags*.
- Krok 2.** Przygotuj formularz aplikacji, jak pokazałem na rysunku 2.62. Zwróć uwagę na to, że wartością właściwości *Text* formularza jest *Flags*. Nazwy kontroltek zostały pokazane na rysunku. Po umieszczeniu poszczególnych kontroltek w formularzu zajrzyj do tabeli 2.4, w której przedstawiłem ustawienia właściwości tych kontroltek.



Rysunek 2.62. Formularz aplikacji Flags

Tabela 2.4. Ustawienia właściwości kontroltek

Nazwa kontrolki	Typ kontrolki	Ustawienia właściwości
instructionLabel	Label	Text: <i>Kliknij flagę, a poznasz nazwę przedstawianego przez nią kraju.</i>
finlandPictureBox	PictureBox	Image: wybierz i zaimportuj plik <i>Finland.bmp</i> z katalogu <i>Rozdział02</i> znajdującego się w materiałach przygotowanych do książki. BorderStyle: <i>FixedSingle</i> SizeMode: <i>AutoSize</i>
francePictureBox	PictureBox	Image: wybierz i zaimportuj plik <i>France.bmp</i> z katalogu <i>Rozdział02</i> znajdującego się w materiałach przygotowanych do książki. BorderStyle: <i>FixedSingle</i> SizeMode: <i>AutoSize</i>
germanyPictureBox	PictureBox	Image: wybierz i zaimportuj plik <i>Germany.bmp</i> z katalogu <i>Rozdział02</i> znajdującego się w materiałach przygotowanych do książki. BorderStyle: <i>FixedSingle</i> SizeMode: <i>AutoSize</i>
countryLabel	Label	AutoSize: <i>False</i> BorderStyle: <i>FixedSingle</i> Font: <i>Microsoft Sans Serif</i> (styl: pogrubiona, rozmiar: 10 punktów). Text: (zawartość właściwości Text została usunięta). TextAlign: <i>MiddleCenter</i>

Krok 3. Po przygotowaniu formularza wraz z kontrolkami należy utworzyć procedury obsługi zdarzeń Click kontrolki PictureBox. W oknie *Designer* dwukrotnie kliknij kontrolkę finlandPictureBox. To spowoduje wyświetlenie okna edytora kodu źródłowego, w którym zobaczysz szkielet procedury obsługi zdarzeń o nazwie finlandPictureBox_Click(). Umieść w niej następujące polecenie:

```
countryLabel.Text = "Finlandia";
```

Krok 4. Powrót do okna *Designer* i dwukrotnie kliknij kontrolkę francePictureBox. W oknie edytora kodu źródłowego zobaczysz szkielet procedury obsługi zdarzeń o nazwie francePictureBox_Click(). Umieść w niej następujące polecenie:

```
countryLabel.Text = "Francja";
```

Krok 5. Powrót do okna *Designer* i dwukrotnie kliknij kontrolkę germanyPictureBox. W oknie edytora kodu źródłowego zobaczysz szkielet procedury obsługi zdarzeń o nazwie germanyPictureBox_Click(). Umieść w niej następujące polecenie:

```
countryLabel.Text = "Niemcy";
```

Krok 6. Obecna postać kodu źródłowego przedstawiłem na listingu 2.10. Jak wcześniej wspomniałem, numery wierszy nie stanowią fragmentu kodu źródłowego. Zdecydowałem się na ich użycie, ponieważ dzięki tym numerom mogę w tekście łatwiej odwoływać się do poszczególnych poleceń. Polecenia przedstawione czcionką pogrubioną to te, które zostały wpisane przez Ciebie. Upewnij się, że kod źródłowy ma postać taką jak na listingu 2.10. Nie zapomnij o średnikach na końcu poleceń!

Listing 2.10. Ukończony kod źródłowy formularza Form1 aplikacji Flags

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9 using System.Windows.Forms;
10
11 namespace Flags
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         private void finlandPictureBox_Click(object sender, EventArgs e)
21         {
22             countryLabel.Text = "Finlandia";
23         }
24
25         private void francePictureBox_Click(object sender, EventArgs e)
26         {
27             countryLabel.Text = "Francja";
28         }
29     }
30 }
```

```

29
30     private void germanyPictureBox_Click(object sender, EventArgs e)
31     {
32         countryLabel1.Text = "Niemcy";
33     }
34 }
35 }

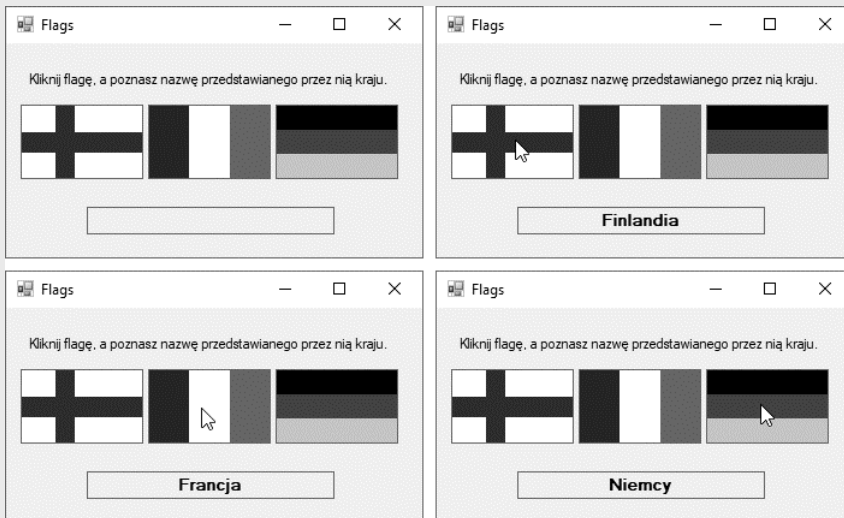
```

Krok 7. Zapisz projekt, a następnie naciśnij klawisz *F5* lub kliknij przycisk *Start Debugging* (🐞) na pasku narzędzi. To spowoduje skompilowanie i uruchomienie aplikacji.



UWAGA. Jeżeli nie popełniłeś błędu w kodzie procedur obsługi zdarzeń, aplikacja powinna zostać uruchomiona. Natomiast w przypadku nieprawidłowych poleceń w procedurach obsługi zdarzeń zostanie wyświetlone okno wraz z informacjami o błędach w trakcie kompilacji. Jeśli tak się zdarzy, w wyświetlonym oknie kliknij przycisk *No*, a następnie popraw kod źródłowy, aby miał postać pokazaną na listingu 2.11.

Na rysunku 2.63 przedstawiłem formularz aplikacji po jej uruchomieniu i kliknięciu każdej z kontroltek *PictureBox*. Po kliknięciu wszystkich flag, aby mieć pewność co do prawidłowego działania programu, możesz zamknąć formularz aplikacji.



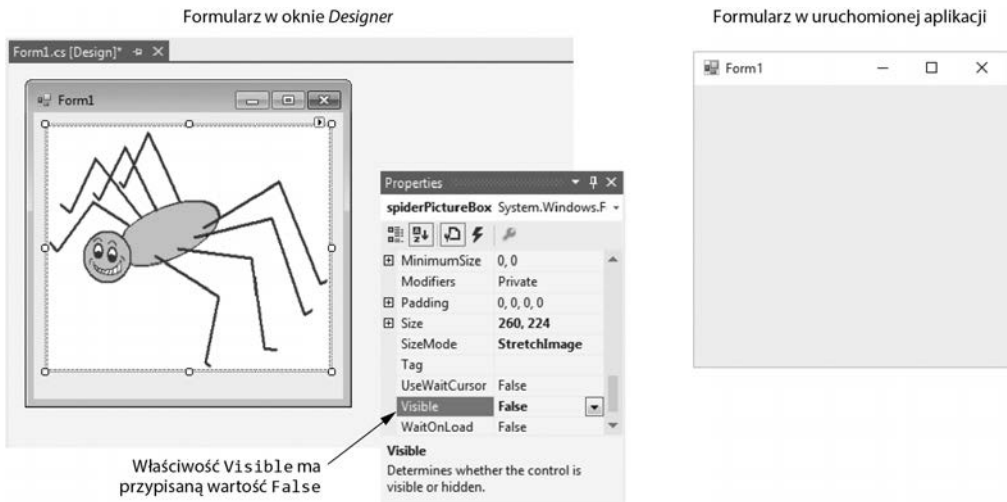
Rysunek 2.63. Uruchomiona aplikacja *Flags*

Właściwość *Visible*

Większość kontroltek ma właściwość **Visible** określającą, czy dana kontrolka będzie widoczna w formularzu podczas działania aplikacji. To jest właściwość typu boolowskiego, czyli można jej przypisać jedynie wartość *true* lub *false*. Przypisanie *True*

właściwości kontrolki oznacza, że będzie ona widoczna w trakcie działania aplikacji. Natomiast wartość `False` powoduje niewyświetlenie kontrolki przez uruchomioną aplikację. Wartością domyślną omawianej właściwości jest `True`.

Gdy używasz okna *Properties* do zmiany właściwości *Visible* kontrolki w trakcie jej projektowania, ona wciąż będzie widoczna w oknie *Designer*. Jednak po uruchomieniu aplikacji kontrolka pozostanie niewidoczna w formularzu. Przykładowo: po lewej stronie na rysunku 2.64 pokazałem formularz wyświetlony w oknie *Designer*. Właściwość *Visible* kontrolki *PictureBox* ma przypisaną wartość `False`, ale mimo to kontrolka nadal pozostaje widoczna w oknie *Designer*. Natomiast po prawej stronie na rysunku 2.64 pokazałem formularz w uruchomionej aplikacji. Jak możesz zobaczyć, kontrolka *PictureBox* jest niewidoczna.



Rysunek 2.64. Kontrolka *PictureBox*, której właściwość *Visible* ma przypisaną wartość `False`

Właściwość *Visible* kontrolki może zostać zmodyfikowana również w kodzie za pomocą operatora przypisania. To daje możliwość ukrycia lub wyświetlenia kontrolki w trakcie działania aplikacji. Przykładowo: kontrolka *PictureBox* na rysunku 2.64 nosi nazwę *spiderPictureBox*. Przypisanie jej właściwości *Visible* wartości `true` odbywa się za pomocą następującego polecenia:

```
spiderPictureBox.Visible = true;
```

Po wykonaniu tego polecenia kontrolka *spiderPictureBox* stanie się widoczna. Z kolei następne polecenie powoduje przypisanie właściwości *Visible* kontrolki wartości `false`:

```
spiderPictureBox.Visible = false;
```

Po wykonaniu tego polecenia kontrolka *spiderPictureBox* stanie się niewidoczna.



UWAGA. Podczas używania wartości `true` i `false` w kodzie źródłowym, jak w pokazanych przed chwilą poleceniach przypisania, te wartości muszą być zapisane małymi literami. W C# `true` i `false` to słowa kluczowe i jeśli nie zostaną zapisane małymi literami, nastąpi wygenerowanie błędu. Natomiast podczas użycia okna *Properties* do przypisania wartości właściwości boolowskiej, takiej jak `Visible`, wymienione wartości są zapisane w postaci `True` i `False`. Postaraj się, aby ta nieśpójność Cię nie zmyliła!

Przykład 2.5.

Utworzenie aplikacji Card Flip

W tym przykładzie utworzymy aplikację symulującą odwracanie kart. Po uruchomieniu aplikacja wyświetli formularz pokazany po lewej stronie na rysunku 2.65. Formularz początkowo wyświetla odwróconą kartę do gry w pokera. Gdy użytkownik kliknie przycisk *Pokaż przód karty*, wówczas karta zostanie odwrócona przodem, jak pokazałem po prawej stronie na rysunku 2.65. Natomiast kliknięcie przycisku *Pokaż tył karty* ponownie spowoduje odwrócenie karty.



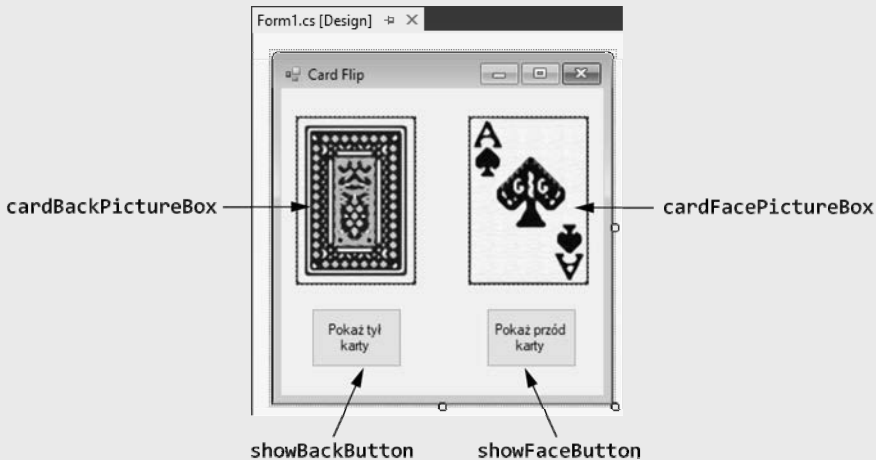
Rysunek 2.65. Aplikacja Card Flip

Do przeprowadzenia symulacji odwracania kart zostanie użyta następująca logika:

- Kliknięcie przycisku *Pokaż przód karty* powoduje, że kontrolka PictureBox wyświetlająca tył karty stanie się niewidoczna i zostanie pokazana kontrolka PictureBox wyświetlająca przód karty.
- Kliknięcie przycisku *Pokaż tył karty* powoduje, że kontrolka PictureBox wyświetlająca przód karty stanie się niewidoczna i zostanie pokazana kontrolka PictureBox wyświetlająca tył karty.

Krok 1. Uruchom Visual Studio i utwórz nowy projekt *Windows Forms Application* o nazwie *Card Flip*.

Krok 2. Przygotuj formularz aplikacji, jak pokazałem na rysunku 2.66. Zwróć uwagę na to, że wartością właściwości *Text* formularza jest *Card Flip*. Nazwy kontroltek zostały pokazane na rysunku. Wykorzystaj okno *Properties* do zdefiniowania właściwości tych kontroltek zgodnie z informacjami przedstawionymi w tabeli 2.5 (w szczególności zwróć uwagę na to, że właściwość *Visible* kontrolki *cardBackPictureBox* ma wartość *True*, natomiast właściwość *Visible* kontrolki *cardFacePictureBox* ma wartość *False*).



Rysunek 2.66. Formularz aplikacji *Card Flip*

Krok 3. Po przygotowaniu formularza wraz z kontrolkami należy utworzyć procedury obsługi zdarzeń *Click* kontroltek *Button*. W oknie *Designer* dwukrotnie kliknij kontrolkę *showBackButton*. To spowoduje wyświetlenie okna edytora kodu źródłowego, w którym zobaczysz szkielet procedury obsługi zdarzeń o nazwie *showBackButton_Click()*. Umieść w niej następujące polecenia:

```
cardBackPictureBox.Visible = true;
cardFacePictureBox.Visible = false;
```

Krok 4. Powrót do okna *Designer* i dwukrotnie kliknij kontrolkę *showFaceButton*. To spowoduje wyświetlenie okna edytora kodu źródłowego, w którym zobaczysz szkielet procedury obsługi zdarzeń o nazwie *showFaceButton_Click()*. Umieść w niej następujące polecenia:

```
cardBackPictureBox.Visible = false;
cardFacePictureBox.Visible = true;
```

Krok 7. Obecna postać kodu źródłowego przedstawiłem na listingu 2.11. Pamiętaj, że numery wierszy nie stanowią fragmentu kodu źródłowego. Polecenia przedstawione czcionką pogrubioną to te, które zostały wpisane przez Ciebie. Upewnij się, że kod źródłowy ma taką samą postać jak na listingu 2.11. Nie zapomnij o średnikach na końcu poleceń!

Tabela 2.5. Ustawienia właściwości kontroltek

Nazwa kontrolki	Typ kontrolki	Ustawienia właściwości
cardBackPictureBox	PictureBox	Image: wybierz i zaimportuj plik <i>Backface_Blue.jpg</i> z katalogu <i>Rozdział02</i> znajdującego się w materiałach przygotowanych do książki. Size: 100;140 SizeMode: Zoom Visible: True
cardFacePictureBox	PictureBox	Image: wybierz i zaimportuj plik <i>Ace_Spades.jpg</i> z katalogu <i>Rozdział02</i> znajdującego się w materiałach przygotowanych do książki. Size: 100;140 SizeMode: Zoom Visible: False
showBackButton	Button	Text: <i>Pokaż przód karty</i> (ręcznie zmień wielkość przycisku, aby zmieścił tekst, jak pokazałem na rysunku 2.66).
showFaceButton	Button	Text: <i>Pokaż tył karty</i> (ręcznie zmień wielkość przycisku, aby zmieścił tekst, jak pokazałem na rysunku 2.66).

Listing 2.11. Ukończony kod źródłowy formularza Form1 aplikacji Card Flip

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9 using System.Windows.Forms;
10
11 namespace Card_Flip
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         private void showBackButton_Click(object sender, EventArgs e)
21         {
22             cardBackPictureBox.Visible = true;
23             cardFacePictureBox.Visible = false;
24         }
25     }
26 }
```

```

25
26     private void showFaceButton_Click(object sender, EventArgs e)
27     {
28         cardBackPictureBox.Visible = false;
29         cardFacePictureBox.Visible = true;
30     }
31 }
32 }

```

Krok 6. Zapisz projekt, a następnie naciśnij klawisz *F5* lub kliknij przycisk *Start Debugging* (🔍) na pasku narzędzi. To spowoduje skompilowanie i uruchomienie aplikacji.

Przetestuj działanie aplikacji, klikając przyciski. Po kliknięciu przycisku *Pokaż przód karty* powinieneś zobaczyć przód karty, jednocześnie jej tył będzie ukryty. Z kolei kliknięcie przycisku *Pokaż tył karty* powinno spowodować ukrycie przodu karty i wyświetlenie jej tyłu. Po przetestowaniu aplikacji możesz ją zamknąć.



UWAGA. Jeżeli nie popełniłeś błędu w kodzie procedur obsługi zdarzeń, aplikacja powinna zostać uruchomiona. Natomiast w przypadku nieprawidłowych poleceń w procedurach obsługi zdarzeń zostanie wyświetlone okno wraz z informacjami o błędach w trakcie kompilacji. Jeśli tak się zdarzy, w wyświetlonym oknie kliknij przycisk *No*, a następnie popraw kod źródłowy, aby miał postać pokazaną na listingu 2.11.



UWAGA. Po PictureBox także wiele innych kontroltek ma właściwość *Visible*. Przykładowo: można ukryć lub wyświetlić kontrolkę *Label1*, zmieniając wartość jej właściwości *Visible*.

Sekwencyjne wykonywanie poleceń

W przykładzie 2.5 utworzone procedury obsługi zdarzeń zawierały więcej niż tylko jedno polecenie. Spójrz na przedstawioną tutaj metodę *showBackButton_Click()*:

```

private void showBackButton_Click(object sender, EventArgs e)
{
    cardBackPictureBox.Visible = true;
    cardFacePictureBox.Visible = false;
}

```

W tej metodzie znajdują się dwa polecenia przypisania. Po wywołaniu metody zostaną one wykonane w kolejności ich zdefiniowania, czyli od początku do końca metody. Dlatego też jako pierwsze będzie wykonane polecenie:

```
cardBackPictureBox.Visible = true;
```

Następnie zostanie wykonane polecenie:

```
cardFacePictureBox.Visible = false;
```

Jednak po uruchomieniu aplikacji tak naprawdę nie można wskazać kolejności poleceń na podstawie obserwowania czynności wykonywanych na ekranie. Po kliknięciu przycisku `showBackButton` zdarzenie `Click` zostaje wykonane tak szybko, że można odnieść wrażenie jednoczesnego wykonania obu poleceń. Trzeba koniecznie zapamiętać, że polecenia są wykonywane pojedynczo, w kolejności ich zdefiniowania w metodzie.

W tej konkretnej metodzie tak naprawdę nie ma żadnego znaczenia kolejność wykonania poleceń. Jeżeli zostałyby użyta odwrotna kolejność, i tak nie zobaczysz różnicy, ponieważ kod aplikacji jest wykonywany bardzo szybko. Jednak w większości aplikacji kolejność definiowania poleceń w procedurach obsługi zdarzeń będzie miała znaczenie krytyczne. W rozdziale 3. rozpoczniesz tworzenie procedur obsługi zdarzeń wykonujących wiele zadań, w większości sytuacji te kroki muszą być podejmowane w odpowiedniej kolejności. W przeciwnym przypadku program nie wygeneruje prawidłowych danych wyjściowych.



Punkt kontrolny

- 2.38. Do czego jest używana kontrolka `PictureBox`?
- 2.39. Gdzie w oknie *Toolbox* znajduje się narzędzie `PictureBox`?
- 2.40. Jak można wyświetlić obraz w kontrolce `PictureBox`?
- 2.41. Jaka jest wartość domyślna właściwości `SizeMode` kontrolki `PictureBox`?
- 2.42. Jaki wpływ na obraz wyświetlany w kontrolce `PictureBox` ma przypisanie wartości `Zoom` właściwości `SizeMode`?
- 2.43. Jak można utworzyć obraz, który da się kliknąć?
- 2.44. Czy wartość właściwości `Visible` kontrolki ma wpływ na wyświetlanie obrazu w trakcie projektowania i uruchomienia aplikacji?

2.8.

Komentarze, puste linie i wcięcia

WYJAŚNIENIE. Komentarz to krótka notatka umieszczona w kodzie źródłowym programu i wyjaśniająca sposób działania danego fragmentu aplikacji. Programiści zwykle używają pustych wierszy i wcięć kodu źródłowego programu, aby nadać mu wizualną strukturę i ułatwić jego odczytywanie.

Komentarze

Komentarz to krótka notatka umieszczona w kodzie źródłowym programu i wyjaśniająca sposób działania danego fragmentu aplikacji. Z założenia komentarze nie są przeznaczone dla kompilatora, a dla każdego człowieka, który będzie odczytywał kod źródłowy programu i spróbuje zrozumieć sposób jego działania.

W języku C# mamy trzy rodzaje komentarzy: jednowierszowy, blokowy i dokumentacji. **Komentarz jednowierszowy** znajduje się tylko w jednym wierszu programu. Taki komentarz rozpoczyna się od dwóch ukośników (//). Wszystko to, co znajduje się między tymi ukośnikami a znakiem końca wiersza, zostaje zignorowane przez kompilator. W przedstawionym tutaj fragmencie kodu pokazałem, jak komentarze jednowierszowe mogą być użyte w procedurze obsługi zdarzeń `showBackButton_Click()` z przykładu 2.5. Każdy komentarz wyjaśnia sposób działania kolejnego wiersza kodu:

```
private void showBackButton_Click(object sender, EventArgs e)
{
    // Wyświetlenie obrazu przedstawiającego tył karty.
    cardBackPictureBox.Visible = true;
    // Ukrycie obrazu przedstawiającego przód karty.
    cardFacePictureBox.Visible = false;
}
```

Komentarz jednowierszowy nie musi zajmować całego wiersza. Przypominam, że wszystko to, co znajduje się między ukośnikami // a znakiem końca wiersza zostaje zignorowane przez kompilator. Dlatego też komentarz można umieścić po poleceniu przeznaczonym do wykonania. Spójrz na przykład takiego rozwiązania:

```
private void showBackButton_Click(object sender, EventArgs e)
{
    cardBackPictureBox.Visible = true; // Wyświetlenie tyłu karty.
    cardFacePictureBox.Visible = false; // Ukrycie przodu karty.
}
```

Komentarz blokowy może zajmować kilka kolejnych wierszy programu. Taki komentarz rozpoczyna się od znaków /* (ukośnik i gwiazdka), a kończy znakami */ (gwiazdka i ukośnik). Wszystko to, co znajduje się między wymienionymi znacznikami, zostaje zignorowane. Kolejny fragment kodu pokazuje przykład komentarza blokowego:

```
/* Procedura obsługi zdarzenia Click kontrolki showBackButton.
   Ta metoda powoduje wyświetlenie obrazu przedstawiającego tył
   karty i jednocześnie ukrywa obraz przedstawiający przód karty.
*/
private void showBackButton_Click(object sender, EventArgs e)
{
    cardBackPictureBox.Visible = true; // Wyświetlenie tyłu karty.
    cardFacePictureBox.Visible = false; // Ukrycie przodu karty.
}
```

Pięć pierwszych wierszy w tym fragmencie kodu to przykład komentarza blokowego wyjaśniającego sposób działania metody `showBackButton_Click()`. Komentarz blokowy ułatwia tworzenie dłuższych wyjaśnień, ponieważ nie trzeba rozpoczynać każdego wiersza od znaku komentarza.

Podczas stosowania komentarzy blokowych pamiętaj o wymienionych tutaj kwestiach:

- Nie zamień miejscami znacznika rozpoczynającego (/*) i kończącego (*/) komentarz blokowy.
- Nie zapomnij o znaczniku kończącym komentarz.

Każdy z tych błędów może się okazać trudny do wychwycenia i jednocześnie uniemożliwi prawidłową kompilację programu.

Trzeci rodzaj komentarza jest znany jako **komentarz dokumentacji**. Taki komentarz jest przez profesjonalnych programistów używany do osadzania w kodzie źródłowym aplikacji wyczerpującej dokumentacji. Visual Studio potrafi wyodrębnić informacje z komentarzy dokumentacji i na ich podstawie wygenerować zewnętrzne pliki z dokumentacją. W takim przypadku komentarz jednowierszowy musi się rozpoczynać od trzech ukośników (///), natomiast blokowy od znaków `/**` i kończyć znakami `*/`. Wprawdzie komentarze dokumentacji są użyteczne dla profesjonalnych programistów, ale nie będziemy ich używać w tej książce.

Jako początkujący programista możesz opierać się idei umieszczania wielu komentarzy w programach. W końcu znacznie bardziej interesujące jest tworzenie kodu faktycznie wykonującego konkretne zadania. Jednak warto poświęcić nieco czasu na dodawanie komentarzy w tworzonym kodzie źródłowym. Niemal na pewno komentarze pozwolą Ci na zaoszczędzenie czasu w przyszłości, gdy będziesz próbować modyfikować lub debugować program. Nawet ogromne i skomplikowane aplikacje mogą się okazać łatwe w odczycie i do zrozumienia, o ile zawierają prawidłowe komentarze.

Używanie pustych wierszy i wcięć, aby ułatwić odczyt kodu

Programiści dość powszechnie używają pustych wierszy i wcięć w kodzie, aby w ten sposób zapewnić mu wizualną strukturę. To jest podobne do tego, jak autor rozmieszcza tekst na stronach książki. Zamiast napisać rozdział w postaci długiej sekwencji zdań, zwykle dzieli go na akapity rozdzielone na stronie. To nie zmienia informacji w książce, a jedynie ułatwia jej lekturę.

Przykładowo: w kolejnym fragmencie kodu umieściłem pusty wiersz w kodzie metody dzielącej jej kod na dwa zestawy poleceń. Wprawdzie ten pusty wiersz nie jest wymagany, ale powoduje, że kod staje się łatwiejszy w odczycie dla człowieka.

```
private void showBackButton_Click(object sender, EventArgs e)
{
    // Wyświetlenie obrazu przedstawiającego tył karty.
    cardBackPictureBox.Visible = true;

    // Ukrycie obrazu przedstawiającego przód karty.
    cardFacePictureBox.Visible = false;
}
```

Do wizualnej organizacji kodu programiści używają także wcięć. Prawdopodobnie zauważyłeś w edytorze kodu źródłowego, że wszystkie polecenia umieszczone w nawiasie klamrowym (`{}`) zostały wcięte. Dlatego też wszystkie polecenia w bloku przestrzeni nazw (`namespace`) zostały wcięte, podobnie jak polecenia klasy i metod. W rzeczywistości Visual Studio jest domyślnie skonfigurowane do automatycznego stosowania wcięć w kodzie.

Wprawdzie wcięcia nie są wymagane, ale bardzo ułatwiają późniejsze odczytywanie kodu źródłowego. Dzięki zastosowaniu wcięć poleceń metody wizualnie odróżniają się od pozostałych. Dlatego też po jedynie krótkim spojrzeniu na kod źródłowy

programista jest w stanie określić, które polecenia zostały zdefiniowane w metodzie. To samo dotyczy klas i przestrzeni nazw. Praktyka polegająca na używaniu wcięć kodu źródłowego jest stosowana w zasadzie przez wszystkich programistów.



Punkt kontrolny

- 2.45. Jaki jest cel istnienia komentarzy?
- 2.46. Czym się różnią komentarze jednowierszowe i blokowe?
- 2.47. Dlaczego powinieneś zwracać baczną uwagę na to, aby pamiętać o znacznikach początkowym i końcowym komentarza blokowego?
- 2.48. Dlaczego programiści wstawiają puste wiersze w kodzie źródłowym i stosują w nim wcięcia?

2.9.

Utworzenie kodu odpowiedzialnego za zamknięcie formularza aplikacji

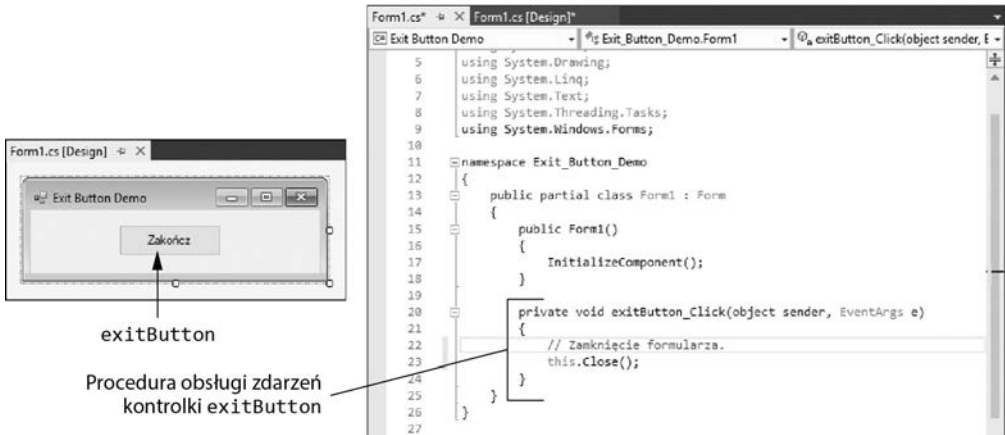
WYJAŚNIENIE. Aby z poziomu kodu źródłowego zakończyć działanie aplikacji, należy użyć polecenia `this.Close()`;

Wszystkie utworzone dotąd w rozdziale aplikacje wymagały od użytkownika kliknięcia standardowego przycisku zamknięcia programu w Windows (X), aby zakończyć działanie. Ten przycisk jest wyświetlany w prawym górnym rogu w zasadzie każdego okna Windows. Jednak w wielu aplikacjach chcesz umożliwić użytkownikowi alternatywny sposób zamknięcia programu. Przykładowo: chciałbyś utworzyć przycisk *Zakończ*, którego kliknięcie zakończy działanie aplikacji.

Zamknięcie formularza aplikacji wymaga wykonania następującego polecenia:

```
this.Close();
```

Spójrz na przykład użycia tego polecenia. Na rysunku 2.67 pokazałem formularz i kod aplikacji pochodzące z projektu o nazwie *Exit Button Demo*. Kontrolka Button widoczna w formularzu nosi nazwę `exitButton`. Jak możesz zobaczyć po prawej stronie rysunku 2.67, w kodzie aplikacji została utworzona procedura obsługi zdarzeń `Click` tego przycisku. Po jego kliknięciu zostanie zamknięty formularz, a tym samym cała aplikacja.



Rysunek 2.67. Formularz wraz z przyciskiem Zakończ

2.10. Usuwanie błędów składni

WYJAŚNIENIE. Edytor kodu źródłowego Visual Studio analizuje każde wpisane polecenie i informuje o wszelkich znalezionych błędach składni. To pozwala na ich szybkie usunięcie.

Tworzenie kodu wymaga dużej precyzji. Nawet niewielki błąd, np. użycie małej litery tam, gdzie powinna być duża, lub pominięcie średnika na końcu polecenia, może uniemożliwić kompilację kodu źródłowego i uruchomienie programu. Przypomnij sobie z rozdziału 1., że takie pomyłki są nazywane błędami składni.

Edytor kodu źródłowego w Visual Studio doskonale radzi sobie ze zgłaszaniem błędów składni tuż po ich popełnieniu przez programistę. Gdy wpisujesz polecenie w edytorze, Visual Studio je analizuje i jeśli znajdzie w nim błąd, podkreśla je postrzępioną linią. Przykład pokazałem na rysunku 2.68. Jeżeli umieścisz kursor myszy nad tym podkreślonym poleceniem, zostanie wyświetlone okno podpowiedzi zawierające opis błędu. Ten opis zwykle jest na tyle wyczerpujący, że pozwala na ustalenie przyczyny błędu i jego usunięcie.

```

private void messageButton_Click(object sender, EventArgs e)
{
    MessageBox.Sh<u>o</u>("Witaj, świecie!");
}

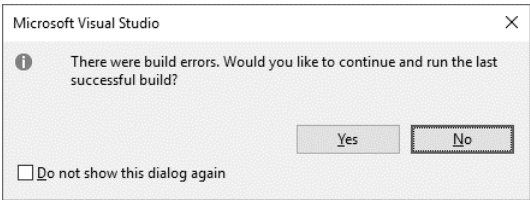
```

Ta linia wskazuje na błąd

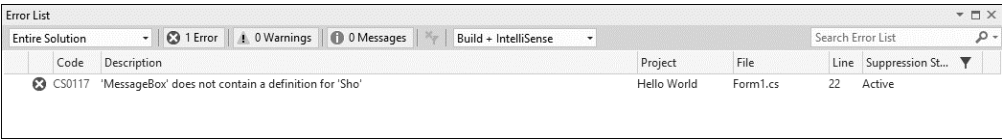
Rysunek 2.68. Podkreślony błąd w kodzie źródłowym

Jeżeli w kodzie projektu znajduje się błąd składni, a mimo to spróbujesz skompilować i uruchomić projekt — przez naciśnięcie klawisza *F5* lub kliknięcie przycisku *Start*

Debugging (🔍) na pasku narzędzi — zobaczysz pokazane na rysunku 2.69 okno informujące o powstaniu błędów podczas kompilacji. Kliknij przycisk *No*, aby zamknąć to okno. Visual Studio wyświetli kolejne okno *Error List*, które pokazałem na rysunku 2.70.



Rysunek 2.69. Okno informujące o powstaniu błędów podczas kompilacji



Rysunek 2.70. Okno *Error List*

Zwróć uwagę na to, że okno *Error List* zawiera opis błędu, nazwę projektu i pliku kodu źródłowego, a także numery wiersza i kolumny, w którym został znaleziony ten błąd. Jeżeli dwukrotnie klikniesz komunikat błędu wyświetlany w oknie *Error List*, edytor kodu źródłowego podkreśli polecenie, które spowodowało wystąpienie danego błędu.



Punkt kontrolny

- 2.49. Którego polecenia użyjesz, aby zamknąć formularz aplikacji z poziomu kodu źródłowego?
- 2.50. W jaki sposób Visual Studio informuje o znalezieniu błędu składni?
- 2.51. Co się stanie po umieszczeniu kursora myszy nad podkreślonym poleceniem?
- 2.52. Co się stanie, jeśli spróbujesz skompilować i uruchomić program zawierający błędy składni?

Ważne pojęcia

ciąg tekstowy
czas działania
czas projektowania
identyfikator
komentarz blokowy
komentarz jednowierszowy

komentarze dokumentacji
kontrolka `Label`
kontrolka `PictureBox`
lista `IntelliSense`
nawias
okno dialogowe

okno komunikatu
operator przypisania
plik *Form1.cs*
plik kodu źródłowego
plik *Program.cs*
polecenie przypisania
procedura obsługi zdarzeń
proporcje
przestrzeń nazw
Przycisk *Alphabetical*
przycisk *Categorized*
ramka ograniczająca
styl camelCase

średnik
uchwyty do zmiany wielkości
wartość boolowska
Witaj, świecie!
właściwość *AutoSize*
właściwość *BorderStyle*
właściwość *Font*
właściwość *Image*
właściwość *SizeMode*
właściwość *TextAlign*
właściwość *Visible*
wywołanie metody
zdarzenie *Click*

Pytania kontrolne

Test jednokrotnego wyboru

- _____ to cienka przerywana linia wokół obiektu w oknie *Designer*.
a. znacznik wyboru
b. znacznik kontrolny
c. ramka ograniczająca
d. kontener obiektu
- Małe kwadraty wyświetlane na prawej i dolnej krawędzi oraz w prawym dolnym rogu ramki ograniczającej formularza są nazywane _____.
a. zaczepami zmiany wielkości
b. krawędziami formularza
c. znacznikami granicznymi
d. uchwytami zmiany wielkości
- _____ to nazwa pustego formularza tworzonego przez Visual Studio początkowo w nowym projekcie.
a. *Form1*
b. *Main*
c. *New1*
d. *Blank*
- Właściwość _____ przechowuje tekst wyświetlany przez przycisk.
a. *Name*
b. *Text*
c. *Tag*
d. *Face*
- Plik zawierający kod programu jest nazywany _____.
a. docelowym plikiem kodu
b. plikiem wykonywalnym

- c. plikiem języka maszynowego
 - d. plikiem kodu źródłowego
6. Przestrzeń nazw to kontener przechowujący _____.
- a. metody
 - b. nazwy
 - c. spacje
 - d. klasy
7. _____ to metoda wykonywana po wystąpieniu określonego zdarzenia w trakcie działania aplikacji.
- a. proces akcji
 - b. procedura obsługi zdarzeń
 - c. procedura uruchomieniowa
 - d. metoda zdarzenia
8. Polecenie `MessageBox.Show("Witaj, świecie!");` to przykład _____.
- a. wywołania metody
 - b. przestrzeni nazw
 - c. zdarzenia `Click`
 - d. procedury obsługi zdarzeń
9. W programowaniu pojęcie *ciąg tekstowy* oznacza _____.
- a. wiele wierszy kodu
 - b. równoległe położenia w pamięci
 - c. zbiór znaków
 - d. praktycznie wszystko
10. _____ oznacza koniec polecenia w C#.
- a. średnik
 - b. kropka
 - c. myślnik
 - d. podkreślenie
11. Fragment danych umieszczonych w kodzie programu to _____.
- a. identyfikator
 - b. specyfikator
 - c. słowo kluczowe
 - d. literal
12. Czas, w trakcie którego tworzysz graficzny interfejs użytkownika i kod aplikacji, jest określany mianem _____.
- a. czasu działania
 - b. czasu projektowania
 - c. czasu programowania
 - d. czasu planowania
13. Czas, w trakcie którego działa aplikacja, jest określany mianem _____.
- a. czasu uruchomienia
 - b. czasu projektowania
 - c. czasu wykonania
 - d. czasu działania

14. Gdy chcesz wyświetlić tekst w formularzu, używasz kontrolki _____.
 - a. Button
 - b. PictureBox
 - c. Label
 - d. TextBox
15. Właściwość _____ pozwala na zdefiniowanie czcionki, stylu czcionki i wielkości tekstu kontrolki.
 - a. Style
 - b. AutoSize
 - c. Text
 - d. Font
16. Właściwość _____ może mieć przypisaną jedną z dwóch możliwych wartości: prawda lub fałsz.
 - a. boolowska
 - b. logiczna
 - c. binarna
 - d. podwójna
17. Kontrolka Label ma właściwość _____ określającą sposób, w jaki można zmienić jej wielkość.
 - a. Stretch
 - b. AutoSize
 - c. Dimension
 - d. Fixed
18. Właściwość _____ może zostać użyta do zmiany sposobu wyrównania tekstu w etykiecie.
 - a. TextPosition
 - b. AutoAlign
 - c. TextCenter
 - d. TextAlign
19. W kodzie można użyć _____ do przechowywania wartości we właściwości kontrolki.
 - a. zdarzenia Click
 - b. wywołania metody
 - c. polecenia przypisania
 - d. wartości boolowskiej
20. Znak równości jest nazywany _____.
 - a. symbolem równości
 - b. operatorem przypisania
 - c. operatorem równości
 - d. położeniem właściwości
21. Standardowa notacja odwołania się w kodzie do właściwości kontrolki to _____.
 - a. *NazwaKontrolki.NazwaWłaściwości*
 - b. *NazwaKontrolki=NazwaWłaściwości*

- c. *NazwaWłaściwości.NazwaKontrolki*
 - d. *NazwaWłaściwości=NazwaKontrolki*
22. _____ to funkcja Visual Studio zapewniająca automatyczne uzupełnianie kodu podczas tworzenia kodu źródłowego.
- a. AutoCode
 - b. AutoComplete
 - c. IntelliSense
 - d. IntelliCode
23. Kontrolki _____ możesz użyć do wyświetlenia obrazu w formularzu.
- a. Graphics
 - b. PictureBox
 - c. Drawing
 - d. ImageBox
24. Po utworzeniu kontrolki PictureBox jej właściwości _____ można użyć do wskazania obrazu, który ma zostać wyświetlony.
- a. Image
 - b. Source
 - c. DrawSource
 - d. ImageList
25. Właściwość _____ kontrolki PictureBox określa sposób wyświetlania wskazywanego przez nią obrazu.
- a. RenderMode
 - b. DrawMode
 - c. SizeMode
 - d. ImageMode
26. _____ to współczynnik długości obrazu do jego wysokości.
- a. proporcja
 - b. współczynnik wielkości
 - c. współczynnik projekcji
 - d. współczynnik obszaru
27. Większość kontrolki ma właściwość _____ określającą ich widoczność w trakcie działania programu.
- a. Render
 - b. Viewable
 - c. Visible
 - d. Draw
28. _____ znajduje się w jednym wierszu kodu źródłowego.
- a. komentarz osadzony
 - b. komentarz jednowierszowy
 - c. komentarz przedni
 - d. komentarz maszynowy
29. _____ może zajmować wiele kolejnych wierszy kodu źródłowego.
- a. komentarz blokowy
 - b. komentarz miejscowy

- c. komentarz wielowierszowy
 - d. komentarz maszynowy
30. Programiści bardzo często używają pustych wierszy i wcięć w kodzie źródłowym, aby zapewnić pewną _____.
- a. logikę
 - b. wizualną strukturę
 - c. dokumentację
 - d. przepływ działania programu
31. Aby z poziomu kodu źródłowego zamknąć formularz aplikacji, należy użyć polecenia _____.
- a. `Close()`;
 - b. `Close.This()`;
 - c. `Close()`
 - d. `this.Close()`;

Prawda czy fałsz?

1. Zmiana właściwości `Text` obiektu powoduje również zmianę jego nazwy.
2. Po utworzeniu formularza jego właściwość `Text` ma początkowo taką samą nazwę jak nazwa formularza.
3. Tytuł formularza jest wyświetlany na pasku tytułu widocznym na górze formularza.
4. Pliki kodu źródłowego `C#` zawsze mają rozszerzenie `.cs`.
5. Podczas tworzenia aplikacji możesz dodawać własny kod do pliku `Program.cs`.
6. Kod w języku `C#` jest zorganizowany w postaci metod zdefiniowanych w klasach, które znajdują się w przestrzeniach nazw.
7. W kodzie `C#` każdemu nawiasowi otwierającemu musi odpowiadać nawias zamykający.
8. Gdy dwukrotnie klikniesz kontrolkę w oknie *Designer*, Visual Studio nie tylko utworzy pustą procedurę obsługi zdarzeń, ale jednocześnie wygeneruje w projekcie pewien kod, którego nie widać, choć jest niezbędny do prawidłowego funkcjonowania procedury obsługi zdarzeń.
9. Właściwość `Text` kontrolki `Label` ma początkowo przypisaną taką samą wartość jak nazwa tej kontrolki.
10. Gdy właściwość `AutoSize` kontrolki `Label` ma przypisaną wartość `True`, nie można ręcznie zmienić wielkości przez kliknięcie i przeciągnięcie uchwytów ramki ograniczającej.
11. Domyślnie tekst etykiety jest wyrównany do dolnej i prawej krawędzi ramki ograniczającej kontrolkę `Label`.
12. Kontrolka `Label` okazuje się użyteczna podczas wyświetlania danych wyjściowych działającej aplikacji.

13. Operator przypisania powoduje przypisanie wartości po jego lewej stronie elementowi znajdującemu się po jego prawej stronie.
14. Kontrolka `PictureBox` ma właściwość `BorderStyle` działającą podobnie jak właściwość o tej samej nazwie w kontrolce `Label`.
15. `Button` to jedyna kontrolka, która może odpowiadać na zdarzenia `Click`.
16. Właściwość `Visible` jest binarna, co oznacza, że może przyjmować jedynie wartości 1 i 0.
17. Gdy w kodzie źródłowym zapisujesz wartości prawdy (`true`) i fałszu (`false`), muszą być podane małymi literami.
18. W języku C# mamy trzy rodzaje komentarzy: jednowierszowe, blokowe i dokumentacji.
19. Aby zamknąć formularz aplikacji z poziomu kodu źródłowego, należy użyć polecenia `Close.This();`.
20. Edytor kodu Visual Studio analizuje każde wpisane polecenie i informuje o wszelkich napotkanych błędach składni.

Krótką odpowiedź

1. Co w oknie *Designer* oznacza ramka ograniczająca wyświetlana wokół obiektu?
2. Co się stanie po umieszczeniu kursora myszy nad krawędzią lub rogiem ramki ograniczającej, która ma uchwyty zmiany wielkości?
3. Co ma wpływ na wygląd obiektu i inne jego cechy charakterystyczne?
4. Co jest wyświetlane przez każdą kolumnę okna *Properties*?
5. Jakie kroki muszą zostać podjęte, aby zmienić właściwość `Text` formularza?
6. Jakie kroki muszą zostać podjęte, aby zmienić właściwość `Size` formularza w oknie *Properties*?
7. Jak za pomocą myszy przenieść kontrolkę do nowego położenia w formularzu?
8. Jakie kroki muszą zostać podjęte, aby zmienić właściwość `Text` kontrolki `Button`?
9. Pokrótkie przedstaw zawartość pliku *Form1.cs*.
10. W jakie znaki jest ujęty literal w kodzie źródłowym?
11. Czy podczas tworzenia procedury obsługi zdarzeń do przycisku można pominąć krok pierwszy i otworzyć edytor kodu, aby samodzielnie wpisać cały kod procedury obsługi zdarzeń? Uzasadnij odpowiedź.
12. Pokrótkie przedstaw różnice między czasami projektowania i działania.
13. Opisz wygląd kontrolki `Label`, której właściwość `BorderStyle` ma przypisaną wartość `Fixed3D`.
14. Jaki efekt ma przypisanie wartości `True` właściwości `AutoSize` kontrolki `Label`?
15. Jakie wartości mogą zostać przypisane właściwości `TextAlign`?

16. Jak można z poziomu kodu usunąć tekst wyświetlany przez kontrolkę `Label`?
17. Jakie formaty obrazów są obsługiwane przez kontrolkę `PictureBox`?
18. Wymień wartości, które można przypisać właściwości `SizeMode` kontrolki `PictureBox`.
19. Jakie trzy rodzaje komentarzy można stosować w `Visual C#`?
20. W jaki sposób `Visual Studio` pomaga w szybkim usunięciu błędów składni?

Warsztat projektanta algorytmów

1. Za pomocą jakiego polecenia możesz wyświetlić komunikat *Dzień dobry* w oknie komunikatu?
2. Za pomocą jakiego polecenia możesz wyświetlić swoje imię w oknie komunikatu?
3. Przyjmując założenie, że graficzny interfejs użytkownika aplikacji ma kontrolkę `Label` o nazwie `dogLabel`. Utwórz polecenie wyświetlające słowo *Fido* w kontrolce `dogLabel`.
4. Przyjmując założenie, że graficzny interfejs użytkownika aplikacji ma kontrolkę `Label` o nazwie `outputLabel`. Utwórz polecenie usuwające wszelki tekst wyświetlany przez tę kontrolkę.
5. Przyjmując założenie, że graficzny interfejs użytkownika aplikacji ma kontrolkę `PictureBox` o nazwie `myPicture`. Utwórz polecenie ukrywające tę kontrolkę.

Ćwiczenia programistyczne

1. Tłumacz z łaciny

Spójrz na przedstawioną tutaj listę łacińskich słów i ich znaczeń w języku polskim:

łacina	polski
sinister	lewy
dexter	prawy
medium	środkowy

Opracuj program, który będzie tłumaczył słowa z języka łacińskiego na polski. Formularz powinien mieć trzy przyciski, po jednym dla każdego słowa z łaciny. Gdy użytkownik kliknie dany przycisk, program powinien wyświetlić w etykiecie tłumaczenie słowa na język polski.

2. Możliwe do kliknięcia obrazy z numerami

W katalogu *Rozdział02* w materiałach przygotowanych do książki znajdziesz pliki obrazów pokazane na rysunku 2.71. Utwórz aplikację wyświetlającą te obrazy w kontrolkach `PictureBox`. Ten program powinien wykonywać wymienione tutaj zadania:

- Gdy użytkownik kliknie obraz przedstawiający cyfrę 1, aplikacja powinna wyświetlić w oknie komunikatu słowo *Jeden*.

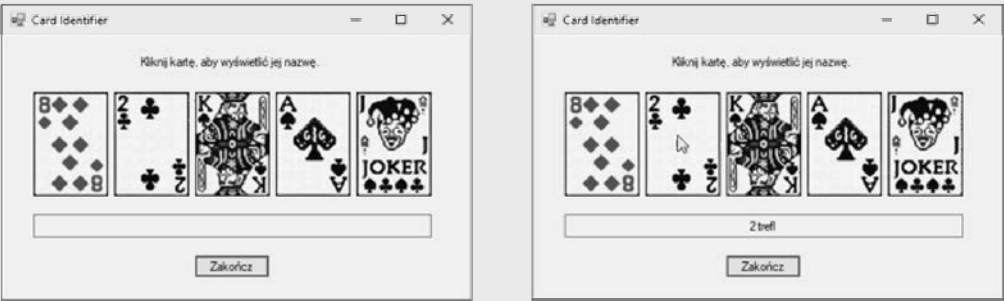
- Gdy użytkownik kliknie obraz przedstawiający cyfrę 2, aplikacja powinna wyświetlić w oknie komunikatu słowo *Dwa*.
- Gdy użytkownik kliknie obraz przedstawiający cyfrę 3, aplikacja powinna wyświetlić w oknie komunikatu słowo *Trzy*.
- Gdy użytkownik kliknie obraz przedstawiający cyfrę 4, aplikacja powinna wyświetlić w oknie komunikatu słowo *Cztery*.
- Gdy użytkownik kliknie obraz przedstawiający cyfrę 5, aplikacja powinna wyświetlić w oknie komunikatu słowo *Pięć*.



Rysunek 2.71. Pliki obrazów do ćwiczenia 2

3. Aplikacja Card Identifier

W materiałach przygotowanych do książki znajdziesz katalog o nazwie *Images\Cards\Poker Large*. W wymienionym katalogu umieściłem pliki w formacie JPEG przedstawiający talię kart. Utwórz aplikację wraz z pięcioma kontrolkami PictureBox. Każda z nich powinna wyświetlać inną kartę z dostępnego zbioru obrazów. Gdy użytkownik kliknie dowolną kontrolkę PictureBox, nazwa karty powinna zostać wyświetlona w kontrolce Label. Na rysunku 2.72 pokazałem uruchomioną aplikację. Po lewej stronie rysunku jest formularz aplikacji po uruchomieniu, natomiast po prawej stronie formularz po kliknięciu jednej z kart.



Rysunek 2.72. Aplikacja Card Identifier

4. Żart

Żart zwykle składa się z dwóch części, często pytania i odpowiedzi. Oto przykład pytania:

Ilu programistów potrzeba do zmiany żarówki?

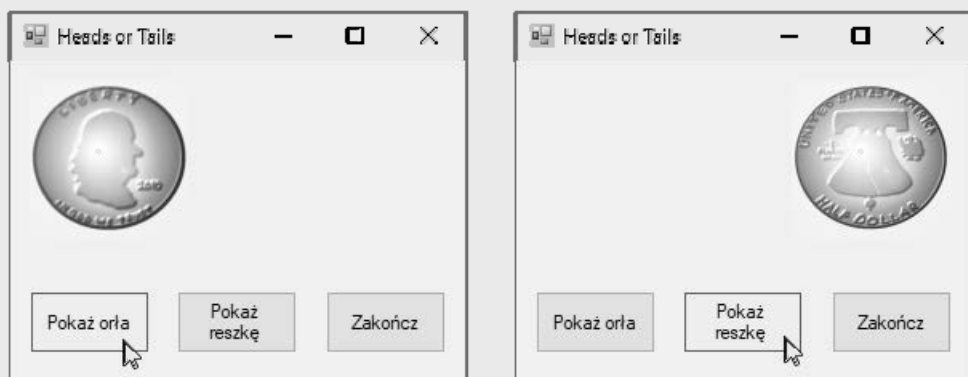
A to przykład odpowiedzi na nie:

Żadnego. To jest problem sprzętowy.

Przypomnij sobie swój ulubiony żart i zidentyfikuj jego oba fragmenty. Następnie utwórz aplikację zawierającą w formularzu etykietę i dwa przyciski. Pierwszy powinien nosić tytuł *Pytanie*, natomiast drugi *Odpowiedź*. Po kliknięciu pierwszego przycisku, w etykiecie powinno zostać wyświetlone pytanie żartu. Z kolei kliknięcie drugiego przycisku ma spowodować wyświetlenie w kontrolce *Label* odpowiedzi na to pytanie.

5. Orzeł czy reszka?

W materiałach przygotowanych do książki znajdziesz katalog o nazwie *Images\Coins*. W wymienionym katalogu umieściłem pliki przedstawiające obie strony monety. Utwórz aplikację wraz z przyciskami *Pokaż orła* i *Pokaż reszkę*. Gdy użytkownik kliknie przycisk *Pokaż orła*, powinien zostać wyświetlony obraz przedstawiający jedną stronę monety. Natomiast po kliknięciu przycisku *Pokaż reszkę* program ma wyświetlić drugą stronę monety. Na rysunku 2.73 pokazałem przykłady formularza omówionej aplikacji.

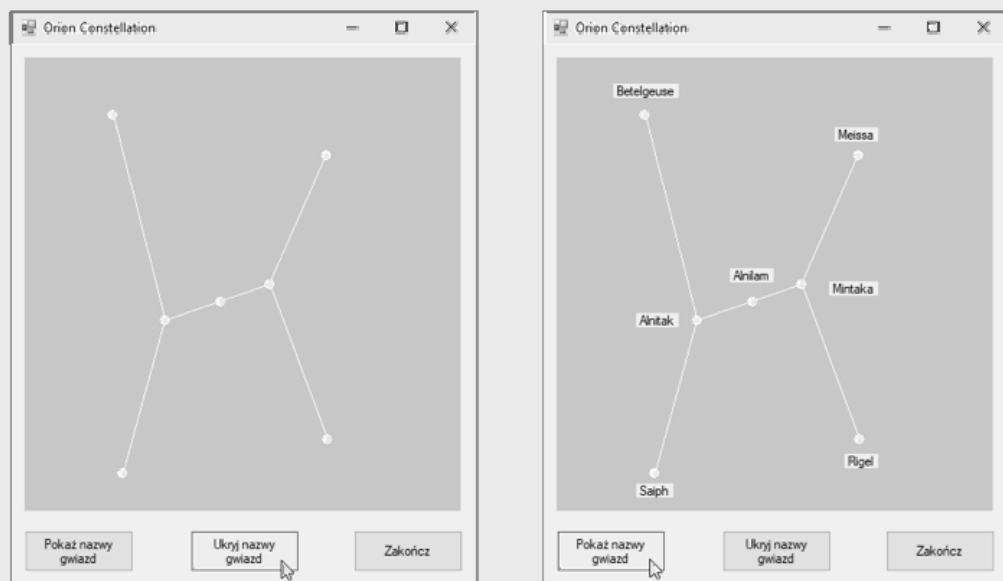


Rysunek 2.73. Aplikacja Heads or Tails

6. Gwiazdozbiór Oriona

Orion to jeden z najbardziej znanych gwiazdozbiorów. W katalogu *Rozdział02* w materiałach przygotowanych do książki znajdziesz plik o nazwie *Orion.bmp* przedstawiający gwiazdozbiór Oriona. Utwórz aplikację wyświetlającą ten obraz w kontrolce *PictureBox*, jak pokazałem po lewej stronie na rysunku 2.74. Aplikacja powinna mieć przycisk, którego kliknięcie spowoduje wyświetlenie wszystkich gwiazd tego gwiazdozbioru, jak pokazałem po prawej stronie na rysunku 2.74. Program powinien mieć również drugi przycisk, którego kliknięcie powoduje ukrycie nazw gwiazd. Oto nazwy gwiazd Oriona: Betelgeuse, Meissa, Alnitak, Alnilam, Mintaka, Saiph i Rigel.

Podpowiedź. W formularzu umieść kontrolkę *PictureBox* wraz z obrazem przedstawiającym gwiazdozbiór. Następnie dodaj kontrolki *Label* przedstawiające nazwy poszczególnych gwiazd. Wykorzystaj okno *Properties* do przypisania



Rysunek 2.74. Aplikacja Orion Constellation

wartości `False` właściwości `Visible` każdej kontrolki `Label`. To spowoduje ukrycie etykiet po uruchomieniu aplikacji. Przycisk *Pokaż nazwy gwiazd* powinien przypisać wartość `true` właściwości `Visible` każdej kontrolki `Label`. Z kolei przycisk *Ukryj nazwy gwiazd* powinien przypisać wartość `false` właściwości `Visible` każdej kontrolki `Label`.

Skorowidz

.NET Framework, 47

A

adapter tabeli, 788

akcesor, 673

get, 675

set, 675

aktywne kontrolki, 230

akumulator, 394

alfabet Morse'a, 657

algorytm, 51

przeszukiwania tablic, 534

sortowania, 534

wyszukiwania, 516

wyszukiwania binarnego, 540

alokacja pamięci, 488

aplikacja

Account Simulator, 686, 688

American Colonies, 517

Array Argument, 511

Birth Date String, 168

Car Demo, 740

Car List, 626

Car Truck SUV Demo, 744

Card Flip, 137

Card Identifier, 154

CD Account Test, 749, 753

Cell Phone Inventory, 695

Cell Phone Test, 678

Change Array, 512

Change Counter, 220

Coin Toss, 408, 671

Color Spectrum, 639

Color Theme, 315

Computer Science Student, 770–773

CSV Reader, 612, 613

Cups To Ounces, 464

Display Elements, 495

Ending Balance, 350

Flags, 132

Friend File, 375, 381

Fuel Economy, 192, 208

Heads or Tails, 155

HScrollBar Demo, 874

Language Translator, 123

Load Event, 413

Loan Qualifier, 276

Lottery Numbers, 499

Multiform Products, 824, 828

Name List, 340

North America, 454

Number List, 340

Password Validation, 584

Pay and Bonus, 468

Payroll with Overtime, 268

Phone Book, 788, 798, 804

Phonebook, 629

Polymorphism, 762, 765

Product Lookup, 820

Product Queries, 838, 848, 851

Product Search, 852, 854

Products, 807

Random Card, 647

Sale Price Calculator, 199

Seating Chart, 547

Secret Word, 294

Selection Sort, 538

South America, 390

Speed Converter, 363

Squares, 359

Sum, 461

Telephone Format, 599, 600

Telephone Unformat, 604

Test Average, 209, 213, 261, 529, 534

Test Score Average, 265

- aplikacja
 - Test Score List, 562
 - Time Zone, 324
 - Total Sales, 398
 - WebBrowser Demo, 877
- aplikacje .NET, 787
- argumenty, 187, 444
 - domyślne, 447
 - nazwane, 446
- arkusze kalkulacyjne, 368
- ASCII, 31, 891
- assembler, 36
- automatyczne
 - ukrywanie, 61
 - uzupełnianie kodu, 126

B

- bajt, 28
- baza danych
 - Phonelist.mdf, 788
 - ProductDB.mdf, 806
- bazy danych, 781
 - aplikacje .NET, 787
 - dodawanie tabel, 790
 - kolumna identyfikacyjna, 785
 - kolumny, 783
 - kopie, 815
 - nawiązanie połączenia, 806
 - położenie pliku, 797
 - rekordy, 783
 - system zarządzania, 781
 - tabele, 783
 - tworzenie, 787
 - typy danych, 785
 - uaktualnienie, 794, 795
- binarny system liczbowy, 29
- bit, 28
- blok
 - catch, 206
 - try, 206
- błąd
 - logiczny, 52, 238
 - przesunięcia o jeden, 501
 - składni, 41, 51, 145

C

- centralna jednostka obliczeniowa, 23
- ciąg tekstowy, 107
 - formatowania, 196
 - konkatenacja, 162
 - modyfikacja, 597, 599

- pobieranie poszczególnych znaków, 579
 - porównywanie, 293
 - tokenizacja, 608
 - wyszukiwanie podciągu, 588
 - CPU, central processing unit, 23
 - CSV, comma separated value, 613
 - cyfrowe dane, 32
 - cykl
 - rozkazowy, 35, 36
 - tworzenia programu, 48
 - czas
 - działania, 109
 - istnienia pola, 218
 - projektowania, 109

D

- dane, 28
 - wejściowe, 157, 307
- DBMS, database management system, 781
- debugger, 238
- debugowanie, 52
 - metod, 473
- definicja klasy, 768
- deklaracja
 - elementów składowych, 661
 - klasy, 661, 664
 - metody, 429
 - tablicy dwuwymiarowej, 544
 - zmiennej, 160
- dekrementacja, 354, 361
- dodawanie
 - kolumny do kontrolki, 820
 - kontrolkek, 86
 - własnego kodu, 103
- domena problemu, 700
- domyślny komunikat błędu, 208
- dostęp
 - do dokumentacji, 70
 - do elementu tablicy, 545
 - do elementu w obiekcie List, 557
 - do kontrolki w formularzu, 721
 - do pliku, 370
 - do pola struktury, 621
 - do przycisków, 231
 - sekwencyjny, 370
 - swobodny, 370
- dostosowanie widoku szczegółowego, 818
- dysk
 - CD, 26
 - SSD, 26
- działanie
 - pętli for, 358
 - programu, 33

dziedziczenie, 735
 dzielenie całkowite, 184

E

edytor
 kodu źródłowego, 99, 104
 typu WYSIWYG, 50
 edytory graficzne, 368
 egzemplarz klasy, 660
 etykieta, 46

F

flaga, 292
 format
 CSV, 613
 notacji naukowej, 197
 notacji wykładniczej, 197
 procentu, 198
 waluty, 197
 formatowanie liczb, 196
 formularz, 46, 82, 218
 DetailsForm, 827
 MainForm, 716, 721
 MessageForm, 716, 717
 NutritionForm, 722
 formularze
 dodawanie do projektu, 710
 dodawanie kontroltek, 86
 dostęp do kontrolki, 721
 kontrolka Button, 89, 104
 kontrolka DataGridView, 803
 kontrolki dołączania danych, 823
 modalne, 719
 niemodalne, 719
 organizacja, 875
 tło, 234
 tworzenie, 709
 usunięcie, 714
 właściwość Text, 85
 wyświetlenie, 714
 zmiana nazwy, 710
 zamknięcie, 144
 funkcje matematyczne SQL, 847

G

generowanie liczb, 186
 graficzny interfejs użytkownika, GUI, 42, 43,
 93, 227
 gry, 368

I

IDE, integrated development environment, 53
 identyfikator, 91
 indeks, 323
 tablicy, 493
 informacje o tablicy, 491
 inicjalizacja
 obiektu typu List, 556
 tablicy, 496
 tablicy dwuwymiarowej, 546
 zmiennej, 172
 inkrementacja, 354
 instrukcja, 34
 IntelliSense, 126
 interfejs
 konsoli, 42
 użytkownika, 42
 wiersza poleceń, 42
 interpreter, 40
 iteracja, 342
 przez tablicę, 497, 643

J

język
 asemblera, 36
 maszynowy, 34, 36
 SQL, 830
 języki
 programowania, 38
 wysokiego poziomu, 37

K

karta
 pamięci SD, 26
 Table Designer, 791, 793
 katalog
 projektu, 67
 rozwiązania, 67
 klasa, 47, 99, 659
 BankAccount, 685
 CDAccount, 746
 CellPhone, 677
 Coin, 667
 List, 555
 Math, 225
 SavingsAccount, 746
 StreamReader, 371
 StreamWriter, 371

- klasy
 - abstrakcyjne, 768
 - bazowe, 736, 761
 - deklaracja, 664
 - określanie zakresu obowiązków, 705
 - po pochodne, 736
 - pole publiczne, 676
 - tworzenie, 661
 - właściwość, 673, 676
 - automatyczna, 683
 - automatyczna tylko do odczytu, 684
 - tylko do odczytu, 682
- klauzula
 - catch, 206
 - Order By, 835
 - Where, 832
 - while, 342
- klucz podstawowy, 785
- kod, 97, 110
 - wygenerowany automatycznie, 804
 - źródłowy, 41
 - źródłowy formularza, 218
- kolejność
 - aktywowania kontrolek, 228
 - operacji, 181
- kolekcja List, 555
- kolumna, 783
 - identyfikacyjna, 785
- komentarz, 141
 - blokowy, 142
 - dokumentacji, 143
 - jednowierszowy, 142
- kompilator, 40
- komponent ErrorProvider, 878
- komunikat, 106
 - błędu wyjątku, 208
- konfiguracja środowiska Visual Studio, 54
- konkatenacja ciągu tekstowego, 162
- konstrukcja
 - if-else, 266, 267
 - if-else-if, 283
 - switch, 318
 - try-catch, 506
- konstrukcje
 - warunkowe, 255
 - warunkowe zagnieżdżone, 273
- konstruktor, 662
 - domyślny, 692
 - klasy bazowej, 756
 - klasy pochodnej, 756
 - pozbawiony parametrów, 692
 - parametryzowany, 685
- kontener List, 694
- kontrolka, 82
 - Button, 87, 88, 826, 846
 - CheckBox, 313
 - DataGridView, 798, 803, 817, 825, 839
 - Details, 814
 - GroupBox, 235
 - ImageList, 645
 - Label, 113, 114, 821
 - właściwość AutoSize, 117
 - właściwość BorderStyle, 116
 - właściwość Font, 114
 - właściwość TextAlign, 118
 - wyświetlenie danych, 119
 - ListBox, 321, 339, 821
 - dołączanie kolumny, 820
 - właściwość Items.Count, 341
 - właściwość SelectedItem, 322
 - MenuStrip, 884, 885
 - OpenFileDialog, 399
 - właściwość Filename, 401
 - właściwość InitialDirectory, 402
 - Panel, 238
 - PictureBox, 127
 - właściwość Image, 128
 - właściwość SizeMode, 129
 - właściwość Visible, 135
 - zdarzenie Click, 131
 - SaveFileDialog, 403
 - właściwość Filename, 404
 - właściwość InitialDirectory, 404
 - właściwość Title, 405
 - TabControl, 875
 - TextBox, 157, 880
 - konwersja wartości, 188
 - usunięcie zawartości, 160
 - weryfikacja danych, 301
 - ToolTip, 869
 - WebBrowser, 876
- kontrolki
 - aktywne, 228, 230
 - dołączania danych, 798, 816, 823
 - kolejność aktywowania, 228
 - pobieranie liczby, 187
 - przeniesienie, 88
 - właściwość Name, 90
 - zmiana nazwy, 89
 - zmiana wielkości, 88
- konwencja nazw camelCase, 91
- konwersja
 - danych, 298
 - niejawna, 191
 - wartości, 178
 - znaków, 580

kopia
 bazy danych, 815
 referencyjna, 519
 kopiowanie tablicy, 519
 kreator Data Source Configuration Wizard,
 799–801, 808–812

L

liczbowe typy danych, 174
 liczby
 losowe, 405
 pseudolosowe, 411
 licznik, 345, 360
 lista, 487
 inicjalizacyjna, 497
 IntelliSense, 126
 kontrolek, 86
 rozkazów procesora, 34
 rozwijana, 871
 rzeczowników, 701
 literal
 ciągu tekstowego, 107
 liczbowy, 175
 znakowy, 578
 logika programu, 50

Ł

łączenie typów danych, 183
 łącznik źródła, 788

M

mechanizm usuwania nieużytków, 504
 menu
 Designer, 883
 systemowe, 883
 metoda, 45, 99, 425
 Add(), 556
 AveragePrice(), 850
 ChangeArray(), 513
 char.IsDigit(), 580, 581
 char.IsLetter(), 581
 char.IsLetterOrDigit(), 581, 582
 char.IsLower(), 582
 char.IsPunctuation(), 582, 583
 char.IsUpper(), 583
 char.IsWhiteSpace(), 583, 584
 Clear(), 560
 decimal.TryParse(), 299
 double.TryParse(), 299
 File.CreateText(), 383

FillByPrice(), 843, 845
 FillByUnits(), 845
 Focus(), 230
 Insert(), 560, 597
 int.TryParse(), 299
 Items.Add(), 339
 Items.Clear(), 341
 MessageBox.Show(), 105, 107
 Next(), 406
 NextDouble(), 407
 Parse(), 187
 ReadLine(), 383, 386
 Remove(), 597
 RemoveAt(), 559
 SearchDesc(), 856
 SequentialSearch(), 518
 ShowDialog(), 715
 ToLower(), 597
 ToString(), 190, 196, 639
 ToUpper(), 597
 Trim(), 597
 TrimEnd(), 597
 TrimStart(), 597
 TryParse(), 298
 Write(), 373
 zmiennaTekstowa.Contains(), 589
 zmiennaTekstowa.EndsWith(), 590
 zmiennaTekstowa.IndexOf(), 590, 593
 zmiennaTekstowa.Insert(), 597
 zmiennaTekstowa.LastIndexOf(), 593, 595
 zmiennaTekstowa.Remove(), 597, 598
 zmiennaTekstowa.StartsWith(), 589
 zmiennaTekstowa.Substring(), 596
 zmiennaTekstowa.ToLower(), 598
 zmiennaTekstowa.ToUpper(), 598
 zmiennaTekstowa.Trim(), 598
 zmiennaTekstowa.TrimEnd(), 599
 zmiennaTekstowa.TrimStart(), 598

metody

abstrakcyjne, 769
 boolowskie, 466–468
 debugowanie, 473
 deklaracja, 429
 klasy Math, 226
 nagłówki, 428
 nazwa, 428
 przeciążone, 690
 przekazywanie argumentów, 437, 440
 przekazywanie obiektów, 672
 treść, 429
 tworzenie, 432
 typu void, 426
 wywoływanie, 429, 432
 zwracające wartość, 426, 458, 463

Microsoft SQL Server Express Edition, 783
 mikroprocesor, 24
 mnemonika, 36
 modularyzacja
 kodu, 425
 weryfikacji danych, 467
 modyfikacja ciągu tekstowego, 597, 599
 modyfikator dostępu, 216, 428

N

nadpisywanie właściwości, 760
 nagłówki
 klasy, 661
 metody, 428
 napęd USB, 26
 narzędzia programistyczne, 27
 nawias
 klamrowy, 101
 okrągły, 106
 nazwa
 formularza, 82
 kolumny, 792
 kontrolki, 82, 89
 metody, 428
 tabeli, 794
 zmiennej, 161, 166
 nieaktualne dane, 683
 niejawna konwersja, 191
 nośnik danych, 26
 notacja
 camelCase, 91
 dziedziczenia, 739
 naukowa, 197
 wykładnicza, 197

O

obiekt, 45, 660, 663
 List, 555, 694
 dodawanie elementów, 556
 dostęp do elementu, 557
 inicjalizacja, 556
 przekazanie metodzie, 558
 tworzenie, 555
 usuwanie elementów, 559
 właściwość Count, 557
 wstawienie elementu, 560
 wyszukiwanie elementu, 560
 sportsCar, 620
 StreamWriter, 371
 ToolStripMenuItem, 885, 889

obiekty
 tworzenie, 663
 typu klasy, 694
 ukryte, 46
 widoczne, 46
 obliczenie sumy bieżącej, 394, 395
 obsługa
 wyjątków, 204, 205, 209, 374
 zdarzeń, 103, 105, 108, 302, 313
 zdarzeń Load, 413
 odczyt
 danych liczbowych, 387
 z pliku, 369, 383, 506
 okna
 dialogowe, 43, 105
 komunikatu, 105, 106
 podpowiedzi, 64
 zakotwiczone i pływające, 65
 okno
 Add New Item, 666, 712, 789
 Czcionka, 115
 Data Source Configuration Wizard, 799–801,
 808–813
 Designer, 60, 68, 82, 101, 713
 String Collection Editor, 323
 Error List, 146
 Images Collection Editor, 646
 IntelliSense, 127
 Locals, 242
 New Project, 57
 Preview Database Updates, 794
 Properties, 60, 83
 Server Explorer, 790
 Solution Explorer, 60, 98, 713, 715, 790
 TableAdapter Configuration Wizard, 837
 TableAdapter Query Configuration Wizard,
 840–849, 855
 TabPage Collection Editor, 876
 Toolbox, 63, 64, 86
 T-SQL, 793
 okres trwałości zmiennej, 166
 opcja Show Table Data, 795
 operand, 180
 operator, 39
 !, 286, 288
 &&, 286
 | |, 286, 287
 --, 354
 !=, 260
 ++, 354
 =, 490
 ==, 259
 And, 834

- Like, 833
- new, 490, 621
- Or, 834
- operatory
 - logiczne, 286
 - matematyczne, 180
 - przypisania, 119
 - przypisania rozszerzone, 185
 - relacji, 258
 - relacyjne w SQL, 832
 - rzutowania, 178, 179
- oprogramowanie, 27
 - systemowe, 27
- otwarcie projektu, 67

P

- pamięć
 - operacyjna, 24
 - RAM, 25
- parametr, 444, 675
 - out, 512
 - ref, 512
- parametry
 - danych wyjściowych, 453
 - referencyjne, 450
 - zapytania, 852
- pasek
 - menu, 61
 - menu narzędzi, 61
 - przewijania, 873
- pętla
 - do-while, 366
 - for, 356–362
 - foreach, 502, 513
 - nieskończona, 353
 - while, 342, 344
- piksel, 32
- plik, 367, 505
 - Form1.cs, 98, 99
 - Program.cs, 98
- pliki
 - bazy danych, 797
 - binarne, 370
 - definicji schematu, 836
 - dołączenie danych, 379, 381
 - kodu źródłowego, 98
 - metody dostępu, 370
 - nazwa, 370
 - obiekt, 370
 - obliczenie sumy bieżącej, 394, 395
 - obsługa wyjątków, 374
 - odczyt danych, 383, 387, 506
 - odczyt w pętli, 389
 - rozszerzenia, 371
 - rozwiązania, 67
 - tekstowe, 370, 375, 378
 - użycie pętli, 390
 - wejściowe, 368
 - wewnętrzny wskaźnik, 385
 - wyjściowe, 368
 - wykrywanie końca, 389
 - zapis danych, 373, 375, 378
 - zapis tablicy, 505
- pobieranie
 - danych, 830
 - znaków ciągu, 579
- podjęcie decyzji, 255
- podklasa, 736
- pole, 45, 216
 - publiczne, 676
 - stałe, 219
 - tekstowe, 46
 - wspierające, 673
 - wyboru, 312, 816
- polecenia debugowania, 243
- polecenie, 40
 - if, 255
 - Select, 830, 835
 - Step Into, 473, 474
 - Step Out, 473, 477
 - Step Over, 473, 475
- polimorfizm, 756
- połączenie z bazą danych, 806
- porównywanie
 - ciągów tekstowych, 293
 - tablic, 520
 - wyliczeń, 643
- procedura
 - obsługi wyjątku, 205
 - obsługi zdarzeń, 103–108, 282, 302, 313, 413
- procesor, 23
 - tekstowy, 368
- programowanie zorientowane obiektowo, 45
- programy, 21
 - modularne, 426
 - narzędziowe, 27
 - sterowane zdarzeniami, 44
 - użytkowe, 27
 - z GUI, 44
- projekt, 65
- projektowanie
 - GUI, 49
 - od ogółu do szczegółu, 436
- przeciążanie metod, 685, 690
- przeglądarki WWW, 368

przekazywanie
 argumentu metodzie, 437, 440
 przez referencję, 449, 487
 przez wartość, 448, 487
 tablicy, 509
 wielu argumentów, 444
 obiektu metodzie, 672
 obiektu parametrowi klasy, 761
 właściwości jako argumentu, 677
 przestrzeń nazw, 99
 przeszukiwanie
 sekwencyjne, 516
 tablicy, 516
 przetwarzanie
 ciągów tekstowych, 577
 danych, 157, 577
 tablicy, 528
 znaków, 577
 przycisk, 46
 akceptacji, 232
 Alphabetical, 86
 anulowania, 232
 Categorized, 86
 opcji, 309
 pola wyboru, 309
 Start Debugging, 97
 przyciski paska narzędzi, 63
 pseudokod, 51
 punkt
 kontrolny, 239
 wykonywania, 240
 puste linie, 141

R

RAM, random-access memory, 25
 referencja, 449
 rekordy, 783
 relacja typu „jest”, 735, 761
 rzutowanie, 178

S

schemat blokowy, 51
 pętli while, 345
 struktury warunkowej, 319
 sekwencyjne wykonywanie poleceń, 140
 składnia, 39
 skróty klawiszowe, 887
 słowo kluczowe, 39
 abstract, 768
 base, 755
 new, 492
 out, 299

override, 758
 private, 428
 virtual, 757
 sortowanie przez wybieranie, 534
 sprawdzanie
 warunków, 281
 przedziału liczbowego, 289
 sprzęt, 22
 SQL, 830
 funkcje matematyczne, 847
 parametry zapytania, 852
 stałe nazwane, 214
 struktura, 618
 sekwencyjna, 255, 275
 warunkowa, 256, 260, 261
 podwójnego wyboru, 266
 pojedynczego wyboru, 256
 wielokrotnego wyboru, 318
 wyboru, 256
 struktury
 porównywanie obiektów, 623
 przechowywanie obiektu, 625
 przekazywanie obiektu, 622
 przypisanie, 622
 tablice obiektów, 624
 tworzenie, 621
 uzyskiwanie dostępu, 621
 suma bieżąca, 394
 superklasa, 736
 sygnatura metody, 691
 symulacja rzutu monetą, 407
 system
 operacyjny, 27
 zarządzania bazą danych, 781

Ś

średnik, 106
 środowisko Visual Studio, 53

T

tabela, 783
 prawdy, 287–289
 tablice, 487, 491
 częściowo zapełnione, 526
 dwuwymiarowe, 543
 deklarowanie, 544
 dostęp do elementu, 545
 inicjalizacja, 546
 określanie wielkości, 546
 sumowanie elementów, 551
 sumowanie kolumny, 552
 sumowanie wiersza, 551

- elementy, 493, 494
- indeksy, 493
- inicjalizacja, 496
- iteracja, 497, 643
- jako argument metody, 509
- kopiowanie, 519
- nieprawidłowy indeks, 498
- obiektów struktur, 624
- obiektów typu klasy, 693
- pętla foreach, 502
- porównywanie, 520
- przechowywanie liczb losowych, 499
- przetwarzanie, 528
- przypisanie zmiennej referencyjnej, 503
- sekwencyjne przeszukiwanie, 516
- sortowanie przez wybieranie, 534
- sumowanie wartości, 522
- tablic, 553
- uśrednianie wartości, 522
- właściwość Length, 498
- wyszukiwanie binarne, 541
- wyszukiwanie elementu, 523
- znaków ASCII, 891
- tło formularza, 234
- tokenizacja ciągu tekstowego, 608
- tryb przerywania, 204
- tworzenie
 - bazy danych, 787
 - egzemplarza struktury, 621
 - graficznego interfejsu użytkownika, 93
 - klas, 48, 661
 - kodu źródłowego, 51, 110
 - kontrolki, 87
 - obiektu, 663
 - programu, 48
 - określenie przeznaczenia, 49
 - projektowanie GUI, 49
 - projektowanie logiki programu, 50
 - testowanie, 52
 - usunięcie błędów logicznych, 52
 - usunięcie błędów składni, 51
 - wielu formularzy, 709
- typ danych, 32, 161, 867
 - char, 578
 - decimal, 175, 178
 - double, 175, 177
 - int, 175, 176
 - string, 162
- typ wartości zwrotnej, 428
- typy
 - danych kolumn, 784
 - plików, 370
 - przekazywane

- przez referencję, 449, 487
- przez wartość, 448, 487
- wyliczeniowe, 636

U

- uaktualnienie bazy danych, 795
- unikanie wyjątków, 298
- urządzenia
 - cyfrowe, 32
 - wejściowe, 26
 - wyjściowe, 26
- usuwanie
 - błędów składni, 145
 - formularza, 714
- użycie parametru referencyjnego, 450

V

- Visual Studio, 53
 - automatyczne ukrywanie okien, 61
 - dostęp do dokumentacji, 70
 - ekran początkowy, 54
 - konfiguracja, 54
 - nowy projekt, 57
 - okna, 60
 - okna zakotwiczone i pływające, 65
 - organizacja rozwiązań, 67
 - pasek menu, 61
 - pasek narzędzi, 61
 - uruchomienie środowiska, 54
 - ustawienia, 56
 - zamknięcie projektu, 59

W

- wartość Null, 786
- wcięcia, 141, 280
- weryfikacja danych, 301
 - wejściowych, 307, 467
- widok szczegółowy, 807, 818–822
- wielokrotne użycie kodu, 426
- wiersz poleceń, 42
- właściwości, 45, 673, 676
 - abstrakcyjne, 769
 - automatyczne, 683
 - automatyczne tylko do odczytu, 684
 - kolumny, 792
 - kontrolek, 121, 124, 133, 139, 169, 194, 202, 210, 221, 325, 409
 - nadpisywanie, 760
 - obiektu, 84
 - tylko do odczytu, 682

właściwość

- AutoSize, 117
- BackColor, 233
- BorderStyle, 116
- Checked, 313, 888
- Count, 557
- Filename, 401, 404
- Font, 114
- ForeColor, 233
- Identity Specification, 792
- Image, 128
- InitialDirectory, 402, 404
- Items.Count, 341
- Length, 498
- Modifiers, 720
- Name, 90
- SelectedIndex, 323
- SelectedItem, 322
- SelectionLength, 880
- SelectionStart, 880
- ShortcutKeys, 887
- ShowShortcut, 887
- SizeMode, 129
- TabIndex, 228
- Text, 85
- TextAlign, 118
- Title, 405
- Visible, 135

wskaźnik pliku, 385

wyjętek, 189, 204

wyliczenia, 639, 642

wyrażenie

- boolowskie, 257, 259, 286
- inicjalizujące, 357
- matematyczne, 180
- sprawdzające, 357
- uaktualniające, 357, 360

WYSIWYG, 50

wyszukiwanie

- binarne, 540
- klas, 699
- podciągu tekstowego, 588
- rzeczowników, 700

wyświetlanie

- wartości liczbowych, 190
- danych, 119
- formularza, 714

wywoływanie metod, 105, 429, 432

zapis do pliku, 369, 371, 505

zapisywanie

- liczb, 29, 32
- znaków, 31

zapytania, 831

- adaptera tabeli, 835, 838

zarządzanie bazą danych, 781

zasięg zmiennej, 165

- parametru, 444

zasobnik komponentów, 399

zaznaczenie tekstu, 880

zbiór danych, 788

zdarzenie, 103

- addButton_Click(), 302
- checkButton_Click(), 291
- CheckedChanged, 314
- Click, 105, 131, 889
- determineGradeButton_Click(), 282, 284
- Load, 412
- okButton_Click(), 313

zgodność typu danych, 444

zintegrowane środowisko programistyczne, IDE, 53

zmiana

- nazwy formularza, 710
- nazwy kontrolki, 89
- nazwy projektu, 58
- wielkości formularza, 83
- wielkości kontrolki, 88
- właściwości Text, 85, 89

zmienna, 160

- typu decimal, 178
- typu double, 177
- typu int, 176
- typu string, 162

zmienne

- boolowskie, 292
- inicjalizacja, 172
- jako pola, 215
- licznika, 345, 360
- lokalne, 164
- okres trwałości, 166
- powtarzanie nazw, 166
- referencyjne, 489, 503
- referencyjne klasy bazowej, 761
- typu wyliczeniowego, 639, 643
- zasięg, 165

znacznik inteligentny, 816

zwracanie ciągu tekstowego, 471

Z

zagnieżdżone konstrukcje warunkowe, 273

zamknięcie formularza aplikacji, 144

zaokrąglanie liczb, 198

Ż

źródło danych, 788

PROGRAM PARTNERSKI

— GRUPY 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
Helion 

>>> VISUAL C#. SOLIDNE PODSTAWY PROWADZĄ DO PERFEKCJI!

C# to nowoczesny i popularny wśród programistów język ogólnego zastosowania. Jego sztandarową zaletą jest wszechstronność i elastyczność: może posłużyć do tworzenia serwisów internetowych, aplikacji biznesowych oraz gier. Oprogramowanie zbudowane w C# będzie poprawnie działać na tradycyjnych komputerach, serwerach, urządzeniach mobilnych, a także na specjalnych urządzeniach do gier. Tworzenie kodu C# w środowisku Visual Studio jest bardzo efektywnym, przyjemnym i motywującym sposobem pracy. Aby jednak napisane w ten sposób aplikacje działały bezproblemowo przez długi czas, należy dobrze poznać reguły rządzące programowaniem.

Ta książka jest przystępnie napisanym podręcznikiem dla początkujących programistów. Dokładne omówienie koncepcji programistycznych umożliwi zrozumienie zasad pisania kodu C#, działania środowiska .NET Framework czy koncepcji relacyjnych baz danych. Dzięki jej lekturze można bardzo szybko zacząć tworzyć atrakcyjne, oparte na zdarzeniach aplikacje zawierające graficzny interfejs użytkownika. Znalazło się tu znakomite wprowadzenie do programowania obiektowego. Wyjaśniono, w jaki sposób należy korzystać z klas dostarczonych wraz z .NET Framework. W zrozumiały sposób pokazano podstawy operacji wejścia-wyjścia, struktur kontrolnych, tablic, list i operacji na plikach, nie zabrakło także omówienia zagadnień dziedziczenia i polimorfizmu. Każde z prezentowanych kwestii zostało zilustrowane zrozumiałymi i praktycznymi przykładami działającego kodu.

W tej książce:

- > solidne podstawy Visual C# i Visual Studio
- > przetwarzanie danych i sterowanie działaniem programu
- > tablice, listy i programowanie obiektowe
- > debugger w Visual Studio i analiza kodu aplikacji
- > bazy danych: podstawy i tworzenie baz w Visual Studio

Tony Gaddis

napisał wiele bardzo popularnych podręczników do nauki programowania. Od ponad 20 lat prowadzi kursy informacyjne, przede wszystkim w Haywood Community College w Karolinie Północnej. Laureat wielu nagród dla nauczycieli i trenerów, jest ceniony za umiejętność przystępnego wyjaśniania nawet bardzo złożonych zagadnień. Dotyczy to szczególnie nauczania języków: Java, Visual Basic i C#.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI
Ściągnij po więcej!



ISBN 978-83-283-4684-0



9 788328 346840

Cena: 149,00 zł

PEARSON
ALWAYS LEARNING

INFORMATYKA W NAJLEPSZYM WYDANIU