PROGRAMMER TO PROGRAMMER™

Zostań specjalistą języka Visual Basic 2010!

- Szczegółowy opis tworzenia aplikacji
- Programowanie obiektowe z pomocą języka Visual Basic 2010
- Debugowanie oraz obsługa błędów
- Współpraca z bazami danych

Od podstaw

Microsoft® Visual Basic® 2010

Thearon Willis, Bryan Newsome

helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

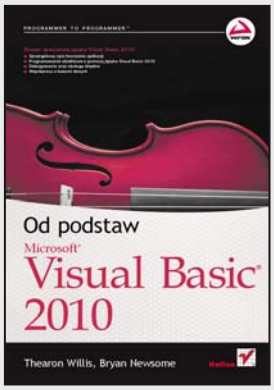
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Visual Basic 2010. Od podstaw



Autorzy: Thearon Willis, Bryan Newsome
Tłumaczenie: Tomasz Walczak
ISBN: 978-83-246-2827-8
Tytuł oryginału: Beginning Visual Basic 2010
Format: 172×245, stron: 736

Zostań specjalistą języka Visual Basic 2010!

- Szczegółowy opis tworzenia aplikacji
- Programowanie obiektowe z pomocą języka Visual Basic 2010
- Debugowanie oraz obsługa błędów
- Współpraca z bazami danych

Visual Basic 2010 to najnowsza wersja języka programowania Visual Basic .NET, jednego z języków współpracujących ze środowiskiem Visual Studio 2010. Jego największe zalety to łatwość stosowania i szybkość tworzenia szerokiego wachlarza aplikacji, od tych przeznaczonych dla systemu Windows, poprzez aplikacje internetowe, aż po te zaprojektowane dla urządzeń mobilnych. Platforma .NET daje programistom języka Visual Basic 2010 możliwość tworzenia w pełni obiektowych programów, napisanych za pomocą klas bazowych wspólnych dla wszystkich języków obsługiwanych przez Visual Studio 2010.

Dzięki tej książce poznasz zagadnienia potrzebne do tworzenia własnych programów w języku Visual Basic 2010. Naucz się podstaw budowania aplikacji Windows Forms, obsługi błędów oraz debugowania własnego kodu. Poznaj sposoby programowania obiektowego i zastosuj je w swoich aplikacjach. Dowiedz się, jak współpracować z zewnętrznymi bazami danych, tworzyć aplikacje sieciowe oraz używać języka XML w swoich programach. Zdobądź wiedzę niezbędną do tego, aby budować profesjonalne aplikacje, które wykorzystują wszystkie możliwości języka Visual Basic 2010.

- Aplikacje Windows Forms
- Podstawowe komendy
- Programowanie obiektowe
- Platforma .NET
- Technologia ASP.NET
- Struktury danych
- Wykorzystanie języka XML
- Wdrażanie aplikacji
- Współpraca z bazami danych
- Obsługa błędów

Dołącz do grona najlepszych programistów języka Visual Basic 2010!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991–2010

Spis treści

Podziękowania **14**
O autorach **15**
O redaktorze merytorycznym **17**
Wprowadzenie **19**

Rozdział 1. Wprowadzenie do języka Visual Basic 2010 ..... **23**
Programowanie oparte na zdarzeniach 24
Instalacja Visual Basic 2010 25
Środowisko programistyczne Visual Basic 2010 28
Ustawianie profilu 29
Menu 29
Paski narzędzi 30
Tworzenie prostej aplikacji 31
Okna środowiska Visual Studio 2010 32
Okno narzędzi 35
Zmodyfikowana notacja węgierska 39
Edytor kodu 40
Używanie systemu pomocy 43
Podsumowanie 43

Rozdział 2. Platforma .NET **47**
Zależność Microsoftu od systemu Windows 47
MSN 1.0 48
Wizja .NET 49
Czy nie przypomina to Javy? 50
Co dalej? 51
Pisanie oprogramowania dla systemu Windows 51
Klasy platformy .NET 52
Wykonywanie kodu 53
Wspólne środowisko uruchomieniowe 54
Ładowanie i wykonywanie kodu 55
Izolacja aplikacji 55
Bezpieczeństwo 55
Współdziałanie 56
Obsługa wyjątków 56
Wspólny system typów i specyfikacja wspólnego języka 57
Podsumowanie 57

Rozdział 3. Pisanie programów ... 61
Informacje i dane ... 61
Algorytmy ... 62
Czym jest język programowania? ... 63
Zmienne ... 63
Komentarze i odstępy ... 66
Komentarze ... 66
Odstępy ... 68
Typy danych ... 68
Używanie liczb ... 68
Podstawowe operacje matematyczne na liczbach całkowitych ... 69
Skrócone operatory matematyczne ... 71
Arytmetyka na liczbach zmiennoprzecinkowych ... 73
Używanie ciągów znaków ... 76
Używanie dat ... 84
Zmienne logiczne ... 90
Przechowywanie zmiennych ... 90
System dwójkowy ... 91
Bity i bajty ... 91
Reprezentowanie wartości ... 92
Przekształcanie wartości ... 93
Metody ... 95
Dlaczego warto używać metod? ... 95
Metody z tego rozdziału ... 96
Tworzenie metod ... 99
Nazwy metod ... 102
Zasięg ... 103
Podsumowanie ... 105

Rozdział 4. Sterowanie przebiegiem programu ... 109
Podejmowanie decyzji ... 109
Instrukcja If ... 110
Instrukcja Else ... 112
Obsługa wielu alternatyw za pomocą instrukcji ElseIf ... 113
Zagnieżdżone instrukcje If ... 114
Jednowierszowe instrukcje If ... 114
Operatory porównania ... 114
Porównywanie ciągów znaków ... 124
Wyrażenie Select Case ... 125
Używanie wyrażenia Select Case bez uwzględniania wielkości liter ... 129
Warunki z wieloma wartościami ... 132
Wyrażenie Case Else ... 133
Używanie różnych typów danych w wyrażeniach Select Case ... 134
Pętle ... 134
Pętle For … Next ... 134
Pętle Do … Loop ... 140

Pętle zagnieżdżone ... 145
Wczesne wychodzenie z pętli ... 147
Pętle nieskończone ... 149
Podsumowanie ... 150

Rozdział 5. Struktury danych ... 153
Wprowadzenie do korzystania z tablic ... 153
Definiowanie i używanie tablic ... 153
Używanie pętli For Each … Next ... 156
Przekazywanie tablic jako parametrów ... 159
Sortowanie tablic ... 161
Przechodzenie w odwrotnym kierunku ... 162
Inicjowanie tablicy ... 163
Wyliczenia ... 165
Używanie wyliczeń ... 165
Określanie stanu ... 168
Ustawianie niepoprawnych wartości ... 170
Stałe ... 171
Używanie stałych ... 171
Stałe różnych typów ... 173
Struktury ... 174
Tworzenie struktur ... 174
Dodawanie właściwości do struktur ... 177
Tablice ArrayList ... 178
Używanie klasy ArrayList ... 178
Usuwanie elementów z listy ArrayList ... 182
Wyświetlanie elementów tablic ArrayList ... 185
Używanie kolekcji ... 186
Tworzenie kolekcji CustomerCollection ... 187
Dodawanie właściwości Item ... 188
Wyszukiwanie elementów za pomocą kolekcji Hashtable ... 190
Używanie kolekcji Hashtable ... 190
Usuwanie elementów — metody Remove, RemoveAt i Clear ... 193
Wrażliwość na wielkość znaków ... 195
Zaawansowane techniki manipulacji tablicami ... 197
Tablice dynamiczne ... 197
Słowo kluczowe Preserve ... 199
Podsumowanie ... 200

Rozdział 6. Język XAML ... 203
Czym jest XAML? ... 203
Składnia języka XAML ... 205
Technologia WPF ... 208
Tworzenie bogatych interfejsów użytkownika w aplikacjach WPF ... 208
Używanie standardowych kontrolek technologii WPF ... 214
Podłączanie zdarzeń ... 218
Podsumowanie ... 222

Rozdział 7. Tworzenie aplikacji dla systemu Windows 225
Reagowanie na zdarzenia 225
Ustawianie zdarzeń przycisku 226
Tworzenie prostych aplikacji 233
Tworzenie formularza 233
Zliczanie liter 236
Zliczanie słów 239
Bardziej złożone aplikacje 244
Aplikacja do edycji tekstu 245
Tworzenie paska narzędzi 246
Tworzenie paska stanu 250
Tworzenie pola edycji 252
Usuwanie zawartości pola edycji 254
Obsługa działania przycisków paska narzędzi 256
Używanie wielu formularzy 262
Okno z informacjami o programie 262
Podsumowanie 265

Rozdział 8. Okna dialogowe 269
Okno komunikatu 269
Ikony okna komunikatu 270
Przyciski okna komunikatu 271
Ustawianie przycisku domyślnego 271
Inne opcje 271
Składnia metody Show 271
Przykładowe okna komunikatu 273
Kontrolka OpenFileDialog 276
Kontrolka OpenFileDialog 277
Właściwości kontrolki OpenFileDialog 277
Metody kontrolki OpenFileDialog 279
Używanie kontrolki OpenFileDialog 279
Kontrolka SaveFileDialog 283
Właściwości kontrolki SaveFileDialog 284
Metody kontrolki SaveFileDialog 285
Używanie kontrolki SaveFileDialog 285
Kontrolka FontDialog 288
Właściwości kontrolki FontDialog 288
Metody kontrolki FontDialog 288
Używanie kontrolki FontDialog 288
Kontrolka ColorDialog 291
Właściwości kontrolki ColorDialog 292
Używanie kontrolki ColorDialog 293
Kontrolka PrintDialog 294
Właściwości kontrolki PrintDialog 295
Używanie kontrolki PrintDialog 296

Klasa PrintDocument 296
Drukowanie dokumentu 296
Kontrolka FolderBrowserDialog 303
Właściwości kontrolki FolderBrowserDialog 303
Używanie kontrolki FolderBrowserDialog 304
Podsumowanie 306

Rozdział 9. Tworzenie menu 309
Właściwości menu 309
Rysunki 310
Klawisze dostępu 310
Klawisze skrótu 310
Znacznik wyboru 310
Okno właściwości 311
Tworzenie menu 312
Projektowanie menu 312
Dodawanie pasków narzędzi i kontrolek 314
Kod obsługujący menu 316
Dodawanie kodu obsługującego menu Widok oraz paski narzędzi 320
Testowanie kodu 321
Menu kontekstowe 324
Tworzenie menu kontekstowego 324
Włączanie i wyłączanie opcji menu oraz przycisków paska narzędzi 327
Podsumowanie 331

Rozdział 10. Debugowanie i obsługa błędów 333
Główne rodzaje błędów 334
Błędy składni 334
Błędy wykonania 337
Błędy logiczne 338
Debugowanie 339
Tworzenie przykładowego programu 339
Ustawianie punktów przerwania 355
Debugowanie za pomocą okien Watch i QuickWatch 362
Używanie okna Autos 364
Używanie okna Locals 364
Obsługa błędów 366
Używanie ustrukturalizowanej obsługi błędów 367
Podsumowanie 369

Rozdział 11. Tworzenie obiektów 373
Wprowadzenie do podejścia obiektowego 373
Hermetyzacja 375
Metody i właściwości 375
Zdarzenia 375
Widoczność 376
Czym jest klasa? 377

Tworzenie klas 377
Powtórne wykorzystanie kodu 378
Projektowanie klasy 379
Stan 380
Działanie 380
Zapisywanie stanu 381
Prawdziwe właściwości 383
Właściwości do odczytu i zapisu 386
Metoda IsMoving 389
Konstruktory 391
Dziedziczenie 393
Dodawanie nowych metod i właściwości 394
Dodawanie metody GetPowerToWeightRatio 397
Zmiana ustawień domyślnych 398
Polimorfizm — trudne słowo, łatwe pojęcie 400
Przesłanianie innych metod 401
Dziedziczenie po klasie Object 403
Obiekty i struktury 403
Klasy platformy .NET 404
Przestrzenie nazw 404
Instrukcja Imports 406
Tworzenie własnych przestrzeni nazw 407
Dziedziczenie na platformie .NET 410
Podsumowanie 411

Rozdział 12. Zaawansowane techniki programowania obiektowego 413
Tworzenie przeglądarki ulubionych stron internetowych 413
Skróty internetowe i adresy ulubionych stron 414
Używanie klas 416
Przeglądanie skrótów do ulubionych stron 422
Otwieranie stron 429
Inna wersja przeglądarki ulubionych 431
Dostęp do ulubionych stron za pomocą zasobnika 431
Wyświetlanie listy ulubionych stron 433
Używanie współdzielonych właściwości i metod 436
Używanie procedur współdzielonych 437
Używanie metod współdzielonych 441
Programowanie obiektowe i zarządzanie pamięcią 442
Przywracanie pamięci 444
Zwalnianie zasobów 444
Defragmentacja i kompaktowanie 445
Podsumowanie 446

Rozdział 13. Tworzenie bibliotek klas 449
Biblioteki klas 450
Tworzenie biblioteki klas 450
Tworzenie biblioteki klas dla projektu Favorites Viewer 452

Aplikacje wielowarstwowe 455
Używanie silnych nazw 456
Podpisywanie podzespołów 457
Wersje podzespołu 459
Rejestrowanie podzespołów 460
Narzędzie Gacutil 460
Dlaczego utworzonego podzespołu nie widać w oknie dialogowym References? 461
Projektowanie bibliotek klas 461
Używanie gotowych bibliotek klas 462
Podglądanie klas za pomocą przeglądarki obiektów 463
Podsumowanie 464

Rozdział 14. Tworzenie własnych kontrolek formularzy Windows 467
Kontrolki formularzy Windows 468
Tworzenie i testowanie kontrolek użytkownika 468
Udostępnianie właściwości kontrolek użytkownika 472
Dodawanie właściwości 472
Udostępnianie metod kontrolki użytkownika 474
Udostępnianie zdarzeń kontrolki użytkownika 475
Etap projektowania a czas wykonywania programu 479
Tworzenie kontrolki CommandLink 481
Tworzenie kontrolki CommandLink 482
Używanie kontrolki CommandLink 490
Podsumowanie 493

Rozdział 15. Dostęp do baz danych 495
Czym są bazy danych? 496
Obiekty bazodanowe Microsoft Access 496
Tabele 496
Kwerendy 497
Instrukcja SELECT języka SQL 497
Kwerendy w bazie danych Access 499
Komponenty dostępu do danych 503
DataSet 503
DataGridView 504
BindingSource 504
BindingNavigator 504
TableAdapter 504
Wiązanie danych 505
Podsumowanie 511

Rozdział 16. Programowanie baz danych przy użyciu SQL Server i ADO.NET 515
ADO.NET 517
Przestrzenie nazw ADO.NET 517
Klasa SqlConnection 518
Klasa SqlCommand 520

Klasa SqlDataAdapter 522
Klasa DataSet 526
Klasa DataView 527
Klasy ADO.NET w praktyce 529
Przykład zastosowania obiektu DataSet 530
Wiązanie danych 538
Obiekty BindingContext i CurrencyManager 538
Wiązanie kontrolek 539
Podsumowanie 567

Rozdział 17. Witryny w technologii Dynamic Data 571
Tworzenie witryny typu Dynamic Data Linq to SQL 571
Zmienianie projektu witryn typu Dynamic Data 577
Podsumowanie 583

Rozdział 18. ASP.NET 585
Architektura typu uproszczony klient 586
Formularze WWW a formularze Windows 587
Zalety formularzy Windows 587
Zalety formularzy WWW 587
Aplikacje sieciowe — podstawowe elementy 588
Serwery WWW 588
Przeglądarki 588
Hipertekstowy język znaczników 588
Język JavaScript 589
Kaskadowe arkusze stylów (CSS) 589
Technologia Active Server Pages 589
Zalety 590
Specjalne pliki witryn internetowych 590
Tworzenie aplikacji 590
Kontrolki — okno narzędzi 591
Tworzenie witryn 591
Tworzenie formularzy WWW oraz przetwarzanie po stronie klienta i po stronie serwera 591
Lokalizacje witryn internetowych w środowisku Visual Studio 2010 597
Przekazywanie danych i sprawdzanie ich poprawności 599
Projektowanie wyglądu i stylu witryny 604
Używanie kontrolki GridView do tworzenia formularzy WWW sterowanych danymi 609
Podsumowanie 614

Rozdział 19. Visual Basic 2010 i XML 619
Wprowadzenie do XML 619
Jak wygląda język XML? 620
XML dla osób poznających Visual Basic 622
Reguły 623

Książka adresowa 623
Tworzenie projektu 623
Klasa SerializableData 624
Wczytywanie plików XML 630
Modyfikowanie danych 633
Wysyłanie poczty elektronicznej 634
Tworzenie listy adresów 635
Pomijanie wybranych składowych 639
Wczytywanie danych adresowych 641
Dodawanie nowych adresów 642
Poruszanie się po danych 644
Usuwanie adresów 646
Integracja z książką adresową 648
Zasady integracji 648
Wczytywanie książki adresowej w innej aplikacji 650
Podsumowanie 655

Rozdział 20. Wdrażanie aplikacji 657
Czym jest wdrażanie? 657
Wdrażanie typu ClickOnce 658
Wdrażanie typu XCOPY 663
Tworzenie aplikacji instalacyjnych przy użyciu Visual Studio 2010 663
Edytor interfejsu użytkownika 667
Wdrażanie innych rozwiązań 670
Podzespoły prywatne 671
Podzespoły współdzielone 671
Wdrażanie aplikacji dla komputerów stacjonarnych 672
Wdrażanie aplikacji sieciowych 672
Wdrażanie usług WWW 673
Przydatne narzędzia 673

Podsumowanie 674

Dodatek A Rozwiązania ćwiczeń 677

Dodatek B Co dalej? 691

Skorowidz 697

3

Pisanie programów

CZEGO NAUCZYSZ SIĘ W TYM ROZDZIALE?

- Działania algorytmów.
- Stosowania zmiennych.
- Działania różnych typów danych, w tym liczb całkowitych, liczb zmiennoprzecinkowych, ciągów znaków i dat.
- Określania zasięgu kodu.
- Usuwania błędów z aplikacji.
- Przechowywania danych w pamięci komputera.

Po zainstalowaniu i uruchomieniu środowiska Visual Basic 2010, a nawet po napisaniu prostego, ale działającego programu, pora przyjrzeć się podstawowym informacjom dotyczącym pisania programów. Dzięki temu będziesz mógł samodzielnie zacząć tworzyć bardziej rozbudowane aplikacje.

Informacje i dane

Informacje opisują fakty i można je przedstawiać oraz wyszukiwać w dowolnym formacie, niezależnie od tego, czy dany format bardziej nadaje się dla ludzi, czy dla komputerów. Na przykład, jeśli cztery osoby będą miały za zadanie mierzyć natężenie ruchu na czterech różnych skrzyżowaniach, po zakończeniu pracy przedstawią cztery ręcznie zapisane listy z liczbą samochodów, które przejechały w danym okresie (może to być jedna lista na każdą godzinę).

Pojęcie *dane* służy do opisu informacji, które zostały zestawione, uporządkowane i sformatowane w taki sposób, aby możliwe było ich bezpośrednie wykorzystanie przez program komputerowy. Komputer nie może bezpośrednio użyć informacji zebranych przez osoby mierzące natężenie ruchu w postaci zestawu kartek pełnych odręcznych zapisków. Ktoś musi najpierw przekształcić te zapiski na dane. Na przykład można przepisać liczby do arkusza programu Excel, który można następnie bezpośrednio wykorzystać w programie zaprojektowanym do analizy wyników.

Algorytmy

Zmiany w przemyśle komputerowym zachodzą z niesamowitą szybkością. Większość profesjonalnych programistów musi cały czas uczyć się czegoś nowego, aby ich umiejętności były aktualne. Jednak niektóre aspekty tworzenia programów nie zmieniły się od czasu ich wymyślenia i prawdopodobnie nie zmienią się w najbliższym czasie. Dobrym przykładem aspektu technologii komputerowej, którego istota nie zmieniła się od początku, jest proces i dyscyplina tworzenia oprogramowania.

Aby program działał, musi mieć dane, na których może pracować. Następnie program bierze takie dane i przekształca je na inną postać. Na przykład aplikacja może pobierać bazę z danymi klientów zapisaną w pamięci komputera jako zestaw jedynek i zer, a następnie przekształcać ją na postać możliwą do odczytania na ekranie monitora. Komputer pokładowy w samochodzie nieustannie analizuje informacje dotyczące środowiska oraz stanu pojazdu i przystosowuje mieszankę paliwa tak, aby zapewnić jak najwydajniejszą pracę silnika. Dostawca usług telekomunikacyjnych zapisuje wykonane telefony i na podstawie tych informacji generuje rachunki.

Wspólną podstawą tych programów są *algorytmy*. Przed napisaniem programu rozwiązującego dany problem, trzeba rozbić go na pojedyncze kroki opisujące rozwiązanie problemu. Algorytm jest niezależny od języka programowania, dlatego możesz zapisać go zarówno za pomocą języka naturalnego, jak i diagramów lub w inny sposób ułatwiający wizualizację problemu. Wyobraź sobie, że pracujesz dla firmy telekomunikacyjnej i masz za zadanie wygenerować rachunki na podstawie telefonów wykonanych przez klientów. Poniższy algorytm opisuje jedno z możliwych rozwiązań:

1. Pierwszego dnia miesiąca musisz utworzyć rachunki dla wszystkich klientów.
2. Z każdym klientem powiązana jest lista połączeń wykonanych w ubiegłym miesiącu.
3. Znasz długość każdej rozmowy, a także porę jej przeprowadzenia. Na podstawie tych informacji możesz obliczyć koszt każdego połączenia.
4. Każdy rachunek to łączna suma wszystkich połączeń.
5. Jeśli klient rozmawiał dłużej, niż wynosi ustalony limit, musi zapłacić określoną kwotę za każdą dodatkową minutę.
6. Do każdego rachunku dodajesz podatek.
7. Po przygotowaniu rachunku trzeba go wydrukować i wysłać pocztą.

Tych siedem punktów opisuje — dość wyczerpująco — algorytm programu generującego rachunki za połączenia wychodzące w systemie operatora telefonii komórkowej. Nie jest istotne, czy gotowe rozwiązanie napisane będzie w języku C++, Visual Basic 2010, C#, Java, czy w innym — podstawowy algorytm programu nie zmieni się. Warto jednak pamiętać, że poszczególne punkty powyższego algorytmu można rozbić na mniejsze, bardziej szczegółowe algorytmy.

Dobrą wiadomością dla osób uczących się programować jest to, że zwykle utworzenie takiego algorytmu jest dość proste. Bardzo łatwo zrozumieć działanie powyższego algorytmu. Algorytmy zawsze tworzy się, wykorzystując zdrowy rozsądek, choć może się okazać, że trzeba napisać kod dla algorytmu zawierającego skomplikowane rozumowanie matematyczne lub naukowe. Możliwe, że nie uznasz tego za algorytm zdroworozsądkowy, jednak będzie on taki dla jego autora. Zła wiadomość jest taka, że proces przekształcania algorytmu na kod może być trudny. Dla programisty nauczenie się tworzenia algorytmów to kluczowa umiejętność.

Wszyscy dobrzy programiści uwzględniają to, że wybór języka programowania nie jest zbyt istotny. Różne języki są dobre do wykonywania odmiennych zadań. Język C++ daje programistom dużą kontrolę nad działaniem programów, jednak pisanie aplikacji w tym języku jest trudniejsze niż na przykład w Visual Basic 2010, podobnie jak tworzenie interfejsu użytkownika. Niektóre z problemów związanych z językiem C++ rozwiązuje jego zarządzana wersja udostępniana przez platformę .NET,

dlatego powyższe stwierdzenia są mniej prawdziwe dziś niż jeszcze kilka lat temu. Jako programista musisz nauczyć się wykorzystywać różne języki do rozwiązywania różnych problemów w jak najwydajniejszy sposób. Choć na początku większość programistów koncentruje się na jednym języku, warto pamiętać, że różne języki pozwalają tworzyć najlepsze rozwiązania dla odmiennych problemów. W pewnym momencie do napisania programu w nowym języku może okazać się konieczne użycie podstawowych umiejętności związanych z projektowaniem algorytmów i kodowaniem.

Czym jest język programowania?

W pewnym sensie możesz postrzegać jako język programowania cokolwiek, co potrafi podejmować decyzje w środowisku komputera. Komputery potrafią doskonale podejmować decyzje, te jednak muszą być stosunkowo proste, na przykład „czy dana liczba jest większa od trzech?" lub „czy ten samochód jest niebieski?".

Jeśli chcesz podjąć skomplikowaną decyzję, proces jej podejmowania należy rozbić na mniejsze elementy zrozumiałe dla komputera. Do rozbijania złożonych decyzji na proste służą algorytmy.

Dobrym przykładem problemu, z którym komputery radzą sobie niezbyt dobrze, jest rozpoznawanie ludzkich twarzy. Nie można po prostu zadać komputerowi pytania „czy to zdjęcie Kasi?". W zamian trzeba rozbić pytanie na serię prostszych pytań zrozumiałych dla komputera.

Decyzja podejmowana przez komputer to jedna z dwóch możliwych odpowiedzi: tak lub nie. Te możliwości określa się także jako prawdę i fałsz lub 1 i 0. Używając pojęć obecnych w programie, nie można kazać komputerowi podjąć decyzji na podstawie pytania „o ile większa jest liczba 10 w porównaniu z liczbą 4?". Poprawnie zadane pytanie brzmi: „czy liczba 10 jest większa od 4?". Różnica nie jest duża, ale istotna — pierwsze pytanie nie pozwala udzielić odpowiedzi tak lub nie, podczas gdy drugie to umożliwia. Komputer oczywiście potrafi udzielić odpowiedzi na pierwsze z tych pytań, ale wymaga to wykonania pewnych operacji. Mówiąc inaczej, aby odpowiedzieć na pierwsze pytanie, należy odjąć 4 od 10 i użyć wyniku tego działania w odpowiednim miejscu algorytmu.

Wymóg stawiania pytań pozwalających na udzielenie odpowiedzi tak lub nie może Ci się wydawać pewnym ograniczeniem, jednak nie należy tak go traktować. Decyzje podejmowane w życiu codziennym wyglądają tak samo. Kiedy podejmujesz jakąś decyzję, możesz coś zaakceptować (tak, prawda, 1) lub odrzucić (nie, fałsz, 0).

Ta książka opisuje język Visual Basic 2010, ale istotne aspekty programowania są w dużym stopniu niezależne od języka. Kluczowe jest zrozumienie, że każdy program, niezależnie od naszpikowania nowinkami i języka, w którym został napisany, składa się z *metod* (funkcji i procedur, które są wierszami kodu służącymi do implementacji algorytmu) oraz *zmiennych* (miejsc przechowywania danych, którymi manipulują metody).

Zmienne

Zmienna to coś, w czym przechowywana jest wartość używana w algorytmie. Na podstawie tych wartości można podejmować decyzje (na przykład „czy dana zmienna równa się 7?" lub „czy dana zmienna jest większa od 4?"), można też wykonywać na nich operacje i przekształcać je na inne wartości (na przykład „dodaj 2 do tej zmiennej" lub „pomnóż daną zmienną przez 6" i tak dalej).

Przed napisaniem kodu przyjrzyj się kolejnemu algorytmowi:

1. Utwórz zmienną o nazwie `intNumber` i przypisz do niej liczbę `27`.
2. Dodaj `1` do wartości zmiennej o nazwie `intNumber` i zapisz nową wartość w tej samej zmiennej.
3. Wyświetl zmienną `intNumber` użytkownikowi aplikacji.

W tym algorytmie tworzysz zmienną o nazwie `intNumber` i przypisujesz do niej liczbę `27`. Oznacza to, że program używa fragmentu pamięci komputera do przechowywania wartości `27`. Ten fragment pamięci przechowuje tę wartość do czasu jej zmiany lub poinformowania programu, że wartość nie jest już potrzebna.

Drugi krok polega na wykonaniu dodawania. Do wartości zmiennej `intNumber` zostaje dodana liczba `1`. Po wykonaniu tej operacji fragment pamięci przechowujący zmienną `intNumber` zawiera wartość `28`.

Na koniec program ma wyświetlić użytkownikowi wartość zmiennej `intNumber`. Należy wczytać tę wartość z pamięci i wyświetlić ją na ekranie.

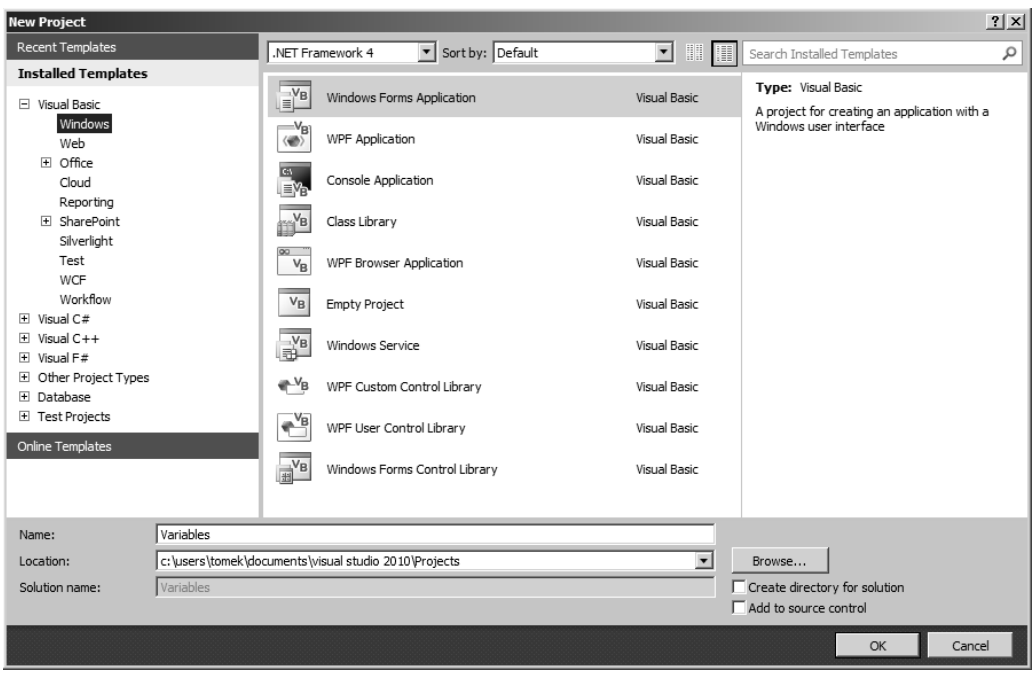
Także ten algorytm jest prosty i zrozumiały. Opiera się jedynie na zdrowym rozsądku. Jednak kod tego algorytmu w języku Visual Basic 2010 jest nieco bardziej tajemniczy.

SPRÓBUJ SAM Używanie zmiennych

Plik z kodem projektu Variables można pobrać z witryny helion.pl.

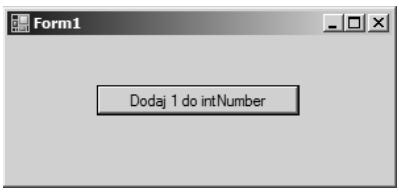
W tym ćwiczeniu „Spróbuj sam" nauczysz się, jak używać zmiennych.

1. Utwórz nowy projekt w środowisku Visual Studio 2010, wybierając z menu opcję *File/New/Project*. W oknie dialogowym *New Project* wybierz aplikację typu *Windows Forms Application* z panelu znajdującego się po prawej stronie, wpisz nazwę projektu, **`Variables`**, a następnie kliknij przycisk *OK* (rysunek 3.1).



Rysunek 3.1. Tworzenie projektu Variables

2. Zmniejsz nieco formularz `Form1` i dodaj do niego przycisk z okna narzędzi. Ustaw właściwość `Text` przycisku na **`Dodaj 1 do intNumber`**, a właściwość `Name` — na **`btnAdd`**. Formularz powinien wyglądać tak jak na rysunku 3.2.

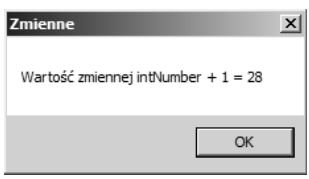


Rysunek 3.2. Główny formularz projektu Variables

3. Kliknij dwukrotnie przycisk, aby otworzyć metodę obsługi zdarzenia `btnAdd_Click`. Dodaj do tej metody kod wyróżniony pogrubieniem:

```
Private Sub btnAdd_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnAdd.Click
   Dim intNumber As Integer
   intNumber = 27
   intNumber = intNumber + 1
   MessageBox.Show("Wartość zmiennej intNumber + 1 = " & intNumber.ToString, _
      "Zmienne")
End Sub
```

4. Kliknij przycisk *Save All* na pasku narzędzi, aby zapisać projekt.
5. Uruchom projekt i kliknij przycisk *Dodaj 1 do intNumber*. Pojawi się okno komunikatu przedstawione na rysunku 3.3.



Rysunek 3.3. Wynik dodawania

Jak to działa?

Po kliknięciu przycisku przez użytkownika program wywołuje metodę obsługi zdarzenia `btnAdd_Click`, rozpoczyna działanie od góry kodu i wykonuje kolejne wiersze. Pierwszy wiersz definiuje nową zmienną o nazwie `intNumber`:

```
Dim intNumber As Integer
```

`Dim` to słowo kluczowe. Jak opisano to w rozdziale 1., słowa kluczowe mają w języku Visual Basic 2010 specjalne znaczenie i służą między innymi do wydawania poleceń. Słowo `Dim` informuje język, że dalej znajduje się definicja zmiennej.

> *Ta dziwna nazwa pochodzi z pierwszych wersji języka BASIC. Język BASIC musiał mieć informacje opisujące ilość miejsca rezerwowanego dla tablic (opisanych w rozdziale 5.), dlatego znajdowało się w nim polecenie informujące o „wymiarach" (ang. dimensions) tablicy, które w skrócie nazwano Dim. W języku Visual Basic rozszerzono to polecenie na inne rodzaje zmiennych i obecnie oznacza ono mniej więcej „przygotuj miejsce na…".*

Następnie pojawia się nazwa zmiennej — `intNumber`. Zauważ, że zmienna nazwana jest zgodnie ze zmodyfikowaną notacją węgierską opisaną w rozdziale 1. W tym przypadku przedrostek `int` to skrót od `Integer`. `Integer` to typ danych reprezentujący zmienną `intNumber`, co opisuje następny akapit. Po przedrostku znajduje się nazwa zmiennej — w tym przypadku jest to `Number`. Jeśli w kodzie natrafisz na taką zmienną, wiesz, że jest to zmienna reprezentująca liczbę typu `Integer`.

Nazwa `Integer` informuje język Visual Basic 2010 o rodzaju wartości przechowywanej przez zmienną. Rodzaj wartości to *typ danych*. Na razie wystarczy zapamiętać, że ta instrukcja informuje język, iż zmienna ma przechowywać wartości typu `Integer` (liczby całkowite).

Kolejny wiersz przypisuje wartość do zmiennej `intNumber`:

```
intNumber = 27
```

Oznacza to, że powyższa instrukcja zapisuje w zmiennej `intNumber` liczbę `27`.

Kolejna instrukcja dodaje do zmiennej `intNumber` liczbę `1`:

```
intNumber = intNumber + 1
```

Powyższy wiersz oznacza: „pobierz aktualną wartość zmiennej `intNumber` i dodaj do niej `1`".

Ostatni wiersz powoduje wyświetlenie okna dialogowego z tekstem `Wartość zmiennej intNumber + 1 =` oraz aktualną wartością tej zmiennej. Ta sama instrukcja ustawia także nagłówek okna dialogowego na `Zmienne`, aby odzwierciedlał przeznaczenie projektu. Przy korzystaniu z wartości liczbowych w tekście warto zastosować metodę `ToString` do zrzutowania liczby na ciąg znaków. Ułatwia to czytanie i rozumienie kodu, ponieważ wiadomo, że użyto w nim ciągu znaków:

```
MessageBox.Show("Wartość zmiennej intNumber + 1 = " & intNumber.ToString, _
                "Zmienne")
```

Komentarze i odstępy

Kiedy piszesz kod programu, zawsze pamiętaj, że w przyszłości ktoś inny może musieć wprowadzać w nim zmiany. Dlatego powinieneś starać się jak najbardziej ułatwić innym programistom odczytanie kodu. Komentarze i odstępy to dwa podstawowe środki do poprawiania czytelności kodu.

Komentarze

Komentarze to elementy programu ignorowane przez kompilator języka Visual Basic 2010, co oznacza, że możesz w nich zapisać dowolne informacje w dowolnym języku — po polsku, w C#, w Perl, w FORTRAN czy po chińsku. Te komentarze mają pomóc programistom czytającym kod w zrozumieniu działania danego fragmentu.

Wszystkie języki programowania umożliwiają dodawanie komentarzy. Nie jest to cecha dostępna wyłącznie w języku Visual Basic 2010. Jeśli na przykład przyjrzysz się kodowi w języku C#, zauważysz, że komentarze rozpoczynają się w nim od podwójnego ukośnika (`//`).

Kiedy wiadomo, że warto dodać komentarz? Zależy to od sytuacji, ale dobrą praktyczną wskazówką jest zastanowienie się nad algorytmem. Program z poprzedniego ćwiczenia „Spróbuj sam" działa według następującego algorytmu:

1. Zdefiniuj wartość zmiennej `intNumber`.
2. Dodaj `1` do wartości zmiennej `intNumber`.
3. Wyświetl nową wartość zmiennej `intNumber` użytkownikowi.

Możesz dodać do przykładowego kodu komentarze opisujące wszystkie kroki algorytmu:

```
' Definicja zmiennej intNumber.
Dim intNumber As Integer

' Ustawianie początkowej wartości.
intNumber = 27

' Dodanie 1 do wartości zmiennej intNumber.
intNumber = intNumber + 1

' Wyświetlenie nowej wartości zmiennej intNumber.
MessageBox.Show("Wartość zmiennej intNumber + 1 = " & intNumber.ToString, _
                "Zmienne")
```

W języku Visual Basic 2010 komentarze rozpoczynają się od apostrofu (`'`). Tekst znajdujący się w wierszu poprzedzonym apostrofem to komentarz. Można także dodawać komentarze w tym samym wierszu, w którym znajduje się kod, na przykład:

```
intNumber = intNumber + 1 ' Dodanie 1 do wartości zmiennej intNumber.
```

Takie rozwiązanie jest poprawne, ponieważ po apostrofie znajduje się jedynie komentarz, a nie kod. Zauważ, że komentarze w powyższym kodzie mniej więcej opisują działanie algorytmu. Dobrą praktyką związaną z komentowaniem kodu jest dodawanie krótkiego opisu tego etapu algorytmu, który jest wykonywany przez dany fragment kodu.

Środowisko Visual Studio 2010 udostępnia także komentarze XML, które pozwalają utworzyć bloki komentarzy stanowiące dokumentację metod. Aby użyć tej właściwości, umieść kursor w pustym wierszu powyżej definicji metody i wpisz trzy następujące po sobie apostrofy. Środowisko automatycznie doda wtedy blok komentarza widoczny w poniższym fragmencie kodu:

```
''' <summary>
'''
''' </summary>
''' <param name="sender"></param>
''' <param name="e"></param>
''' <remarks></remarks>
Private Sub btnAdd_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnAdd.Click
```

Najciekawszą właściwością komentarzy XML jest to, że środowisko Visual Studio 2010 automatycznie uzupełnia nazwy parametrów w bloku komentarzy na podstawie parametrów zdefiniowanych w metodzie. Jeśli dana metoda nie przyjmuje żadnych parametrów, środowisko nie dodaje elementu `<param>` do bloku komentarzy.

Po wstawieniu bloku komentarzy możesz dodać krótki opis działania metody oraz dodatkowe uwagi, które należy uwzględnić, wywołując tę metodę, a także specjalne wymagania stawiane przez metodę. Jeśli metoda zwraca wartość, środowisko dodaje do bloku komentarzy element `<returns>`, a programista może podać zwracaną wartość oraz jej opis.

Komentarze mają przede wszystkim ułatwić zrozumienie kodu, zarówno programistom, którzy stykają się z nim po raz pierwszy, jak i autorowi, który wraca do niego po dłuższej przerwie. Komentarze mają wskazywać coś, co może nie być oczywiste na pierwszy rzut oka, lub stanowić krótki opis działania fragmentu kodu, dzięki czemu programista nie musi analizować działania każdego wiersza, aby zrozumieć jego zastosowanie.

Programiści mają pewne zasady pisania komentarzy. Jeśli pracujesz dla dużej firmy programistycznej lub Twój przełożony albo nauczyciel zwraca dużą uwagę na standardy kodowania, na pewno dowiesz się, w jakiej postaci powinieneś dodawać komentarze, a także kiedy powinieneś je umieszczać.

Odstępy

Kolejny istotny aspekt pisania czytelnego kodu to dodawanie wielu odstępów. *Odstępy*, czyli miejsca na ekranie lub na stronie pozbawione znaków, poprawiają czytelność kodu, podobnie jak odstępy w zwykłym tekście napisanym po polsku. W ostatnim przykładzie przed każdym komentarzem występuje pusty wiersz. Dzięki temu osoba czytająca kod wie, że każdy wyróżniony w ten sposób blok kodu stanowi całość.

Więcej o odstępach dowiesz się w następnym rozdziale, gdzie opisane jest sterowanie przepływem programu za pomocą specjalnych bloków kodu. Jednak w praktyce zauważysz, że różni programiści używają odstępów w odmienny sposób. Na razie zapamiętaj, że nie należy bać się dodawać odstępy do kodu, ponieważ zwiększa to czytelność programów, szczególnie jeśli aplikacja składa się z długich fragmentów kodu.

Kompilator ignoruje odstępy i komentarze, dlatego kod z dużą liczbą komentarzy i odstępów jest równie wydajny jak kod całkowicie pozbawiony tych elementów.

Typy danych

Kiedy używasz zmiennych, powinieneś z góry wiedzieć, jakie wartości zamierzasz w nich przechowywać. Na razie spotkałeś się ze zmienną służącą do przechowywania liczb całkowitych.

Kiedy definiujesz zmienną, musisz poinformować język Visual Basic 2010 o typie danych przechowywanych w tej zmiennej. Jak się już może domyślasz, jest to *typ danych*, a wszystkie znaczące języki programowania udostępniają wiele różnych typów danych. Typ danych zmiennej ma istotny wpływ na uruchamianie kodu przez komputer. W tym punkcie poznasz działanie zmiennych oraz wpływ użytego typu na wydajność programu.

Używanie liczb

Używając liczb w języku Visual Basic 2010, możesz korzystać z dwóch ich rodzajów: z liczb całkowitych oraz zmiennoprzecinkowych. Oba te rodzaje mają specyficzne zastosowania. *Liczby całkowite* są mało użyteczne do obliczania ilości, na przykład ilości pieniędzy na rachunku czy czasu zapełniania basenu wodą. Do operacji tego typu lepiej jest używać liczb *zmiennoprzecinkowych*, ponieważ można za ich pomocą przedstawić części ułamkowe, co nie jest możliwe w przypadku liczb całkowitych.

Z drugiej strony w przypadku większości codziennych czynności bardziej przydatne są liczby całkowite. Większość programów używa liczb raczej do kontroli swojego działania za pomocą zliczania jednostek, niż do obliczania ilości.

Wyobraź sobie, że masz napisać program wyświetlający na ekranie szczegóły dotyczące klienta. Baza danych zawiera 100 klientów. Kiedy program rozpoczyna działanie, wyświetla na ekranie dane pierwszego klienta. Program musi kontrolować, który klient jest aktualnie wyświetlany, aby móc wyświetlić następnego klienta, gdy otrzyma takie polecenie od użytkownika.

Ponieważ komputer doskonale radzi sobie z liczbami, zwykle każdy klient ma niepowtarzalny numer. Ten numer to w prawie wszystkich przypadkach liczba całkowita. W tym przypadku oznacza to, że każdy klient ma przypisaną niepowtarzalną liczbę całkowitą z przedziału od 1 do 100. W programie można także umieścić zmienną przechowującą identyfikator aktualnie wyświetlanego klienta. Kiedy użytkownik poprosi o wyświetlenie danych kolejnego klienta, program musi tylko dodać 1 do tego identyfikatora (*inkrementacja o jeden*) i może wyświetlić następnego klienta.

Działanie mechanizmów tego typu poznasz przy okazji analizy bardziej zaawansowanych zagadnień. Na razie zapamiętaj, że liczb całkowitych używa się dużo częściej niż liczb zmiennoprzecinkowych. Przyjrzyj się teraz często wykonywanym operacjom.

Podstawowe operacje matematyczne na liczbach całkowitych

W tym punkcie utworzysz nową aplikację wykonującą operacje matematyczne. W poniższym ćwiczeniu „Spróbuj sam" zobaczysz, jak dodawać, odejmować, mnożyć i dzielić liczby całkowite.

SPRÓBUJ SAM Podstawowe operacje matematyczne

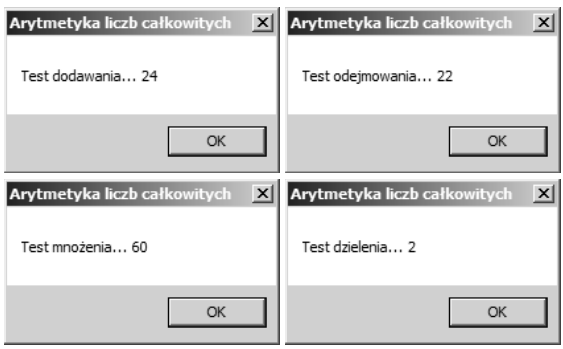
Plik z kodem projektu Integer Math można pobrać z witryny helion.pl.

1. Utwórz nowy projekt w środowisku Visual Studio 2010, wybierając z menu opcję *File/New/Project*. W oknie dialogowym *New Project* wybierz aplikację typu *Windows Forms Application* z prawego panelu (zobacz rysunek 3.1), wpisz nazwę projektu, **`Integer Math`**, a następnie kliknij przycisk *OK*.
2. Dodaj do formularza nowy przycisk z okna narzędzi. Ustaw jego właściwość `Name` na **`btnIntMath`**, a właściwość `Text` — na **`Test`**. Kliknij dwukrotnie ten przycisk i dodaj wyróżniony pogrubieniem kod do utworzonej przez środowisko metody obsługi zdarzenia `Click`:

```
Private Sub btnIntMath_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnIntMath.Click
   ' Deklaracja zmiennej.
   Dim intNumber As Integer

   ' Przypisanie liczby, dodanie do niej wartości i wyświetlenie wyniku.
   intNumber = 16
   intNumber = intNumber + 8
   MessageBox.Show("Test dodawania... " & intNumber.ToString, _
      "Arytmetyka liczb całkowitych")

   ' Przypisanie liczby, odjęcie od niej wartości i wyświetlenie wyniku.
   intNumber = 24
   intNumber = intNumber - 2
   MessageBox.Show("Test odejmowania... "  & intNumber.ToString, _
      "Arytmetyka liczb całkowitych")

   ' Przypisanie liczby, pomnożenie jej i wyświetlenie wyniku.
   intNumber = 6
   intNumber = intNumber * 10
   MessageBox.Show("Test mnożenia... " & intNumber.ToString, _
      "Arytmetyka liczb całkowitych")

   ' Przypisanie liczby, podzielenie jej i wyświetlenie wyniku.
   intNumber = 12
```

```
        intNumber = Ctype(intNumber / 6, Integer)
        MessageBox.Show("Test dzielenia... " & intNumber.ToString, _
           "Arytmetyka liczb całkowitych")
    End Sub
```

3. Zapisz projekt przez kliknięcie przycisku *Save All* na pasku narzędzi.
4. Uruchom projekt i kliknij przycisk *Test*. Pojawią się kolejno cztery okna dialogowe widoczne na rysunku 3.4.



Rysunek 3.4. Operacje arytmetyczne

Jak to działa?

Na szczęście żaden z przedstawionych fragmentów kodu nie powinien być niezrozumiały. Z operatorem dodawania spotkałeś się już w jednym z wcześniejszych przykładów. Tu pojawia się on ponownie:

```
' Przypisanie liczby, dodanie do niej wartości i wyświetlenie wyniku.
intNumber = 16
intNumber = intNumber + 8
MessageBox.Show("Test dodawania... " & intNumber.ToString, _
                "Arytmetyka liczb całkowitych")
```

Program przypisuje do zmiennej `intNumber` liczbę `16`.

Następnie program przypisuje do zmiennej `intNumber` sumę jej aktualnej wartości (`16`) oraz liczby `8`.

Jak widać w oknie dialogowym przedstawionym na rysunku 3.4, operacja dodawania daje `24`, co jest poprawnym wynikiem.

Operator odejmowania to znak minus (`-`). Oto, jak działa on w praktyce:

```
' Przypisanie liczby, odjęcie od niej wartości i wyświetlenie wyniku.
intNumber = 24
intNumber = intNumber - 2
MessageBox.Show("Test odejmowania... "  & intNumber.ToString, _
                "Arytmetyka liczb całkowitych")
```

Działanie tego fragmentu kodu jest podobne:

Program przypisuje do zmiennej `intNumber` liczbę `24`.

Następnie program przypisuje do zmiennej `intNumber` różnicę jej aktualnej wartości (`24`) oraz liczby `2`.

Operator mnożenia to gwiazdka (`*`). Należy go stosować w następujący sposób:

```
' Przypisanie liczby, pomnożenie jej i wyświetlenie wyniku.
intNumber = 6
intNumber = intNumber * 10
MessageBox.Show("Test mnożenia... " & intNumber.ToString, _
                "Arytmetyka liczb całkowitych")
```

Ten fragment kodu działa w następujący sposób:

Przypisuje do zmiennej `intNumber` liczbę `6`.

Następnie przypisuje do zmiennej `intNumber` iloczyn jej aktualnej wartości (`6`) oraz liczby `10`.

Ostatni z przedstawionych operatorów, operator dzielenia, to ukośnik (`/`). Działa on w następujący sposób:

```
' Przypisanie liczby, podzielenie jej i wyświetlenie wyniku.
intNumber = 12
intNumber = Ctype(intNumber / 6, Integer)
MessageBox.Show("Test dzielenia... " & intNumber.ToString, _
                "Arytmetyka liczb całkowitych")
```

Ten fragment kodu wykonuje następujące operacje:

Przypisuje do zmiennej `intNumber` liczbę `12`.

Przypisuje do zmiennej `intNumber` iloraz jej aktualnej wartości (`12`) oraz liczby `6`.

Operacja dzielenia wartości zmiennej `intNumber` przez liczbę `6` znajduje się w funkcji `CType`. Funkcja ta zwraca wynik po przekształceniu wyrażenia na określony typ danych. Tu jest to liczba całkowita, o czym informuje nazwa typu — `Integer`. Ponieważ dzielenie dwóch wartości może prowadzić do uzyskania liczby zmiennoprzecinkowej, należy użyć funkcji `CType` do utworzenia wyniku jako liczby całkowitej.

Ta bezpośrednia konwersja nie jest potrzebna, jeśli ustawienie *Option Strict* ma wartość *Off*. Z techniki tej trzeba jednak korzystać, jeżeli opcja ta ma wartość *On*. Ustawienie to pozwala włączyć lub wyłączyć powiadomienia kompilatora dotyczące konwersji na mniej pojemny typ w operacjach na liczbach, co pozwala uniknąć takich konwersji i zapobiec błędom w czasie wykonywania programu.

Aby zmienić wartość ustawienia *Option Strict*, wybierz opcję *Tools/Options* w środowisku Visual Studio 2010. W oknie dialogowym *Options* rozwiń węzeł *Projects and Solutions*, a następnie kliknij opcję *VB Defaults*. W tym miejscu możesz włączyć lub wyłączyć opcję *Option Strict*.

Skrócone operatory matematyczne

W następnym ćwiczeniu „Spróbuj sam” wykonasz te same operacje za pomocą mniejszej ilości kodu. Służą do tego *operatory skrócone*. Choć na początku nie są tak oczywiste jak ich bardziej rozwlekłe odpowiedniki, szybko je polubisz.

SPRÓBUJ SAM Używanie operatorów skróconych

W tym ćwiczeniu „Spróbuj sam” zmodyfikujesz kod z poprzedniego ćwiczenia i wykorzystasz operatory skrócone do dodawania, odejmowania i mnożenia liczb całkowitych.

1. Ponownie otwórz w edytorze kodu środowiska Visual Studio 2010 plik *Form1.vb* i wprowadź w nim wyróżnione pogrubieniem zmiany:

```
Private Sub btnIntMath_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnIntMath.Click
   ' Deklaracja zmiennej.
   Dim intNumber As Integer
   ' Przypisanie liczby, dodanie do niej wartości i wyświetlenie wyniku.
   intNumber = 16
   intNumber += 8
   MessageBox.Show("Test dodawania. " & intNumber.ToString, _
                   "Arytmetyka liczb całkowitych")
   ' Przypisanie liczby, odjęcie od niej wartości i wyświetlenie wyniku.
   intNumber = 24
   intNumber -= 2
   MessageBox.Show("Test odejmowania. "  & intNumber.ToString, _
                   "Arytmetyka liczb całkowitych")
   ' Przypisanie liczby, pomnożenie jej i wyświetlenie wyniku.
   intNumber = 6
   intNumber *= 10
   MessageBox.Show("Test mnożenia. " & intNumber.ToString, _
                   "Arytmetyka liczb całkowitych")
   ' Przypisanie liczby, podzielenie jej i wyświetlenie wyniku.
   intNumber = 12
   intNumber = CType(intNumber / 6, Integer)
   MessageBox.Show("Test dzielenia. " & intNumber.ToString, _
                   "Arytmetyka liczb całkowitych")
End Sub
```

2. Uruchom projekt i kliknij przycisk *Test*. Pojawią się te same okna dialogowe co w poprzednim ćwiczeniu.

Jak to działa?

Używając skróconej wersji operatorów, należy pominąć drugie wystąpienie zmiennej `intNumber` i umieścić operator po lewej stronie znaku równości. Pierwotna wersja wygląda tak:

```
intNumber = intNumber + 8
```

Wersja skrócona wygląda tak:

```
intNumber += 8
```

Operatory skrócone działają dobrze przy dodawaniu, odejmowaniu i mnożeniu liczb całkowitych. Nie można ich jednak używać przy dzieleniu, ponieważ wynikiem może być liczba z częścią ułamkową.

Problemy z arytmetyką liczb całkowitych

Główny problem z arytmetyką liczb całkowitych polega na tym, że nie można wykonywać żadnych operacji na liczbach zawierających części ułamkowe. Na przykład poniższa operacja jest niepoprawna:

```
' Próba pomnożenia liczb.
intNumber = 6
intNumber = intNumber * 10.23
```

Wprawdzie powyższy kod można uruchomić, ale wyniki będą inne od oczekiwanych. Ponieważ zmienna `intNumber` może zawierać jedynie liczby całkowite, wynik jest zaokrąglany w górę lub w dół do najbliższej liczby całkowitej. W tym przypadku dokładny wynik mnożenia to `61,38`, ale wartość zmiennej `intNumber` to `61`. Gdyby wynik wynosił `61,73`, program przypisałby do zmiennej wartość `62`.

UWAGA

Przy opcji *Option Strict* ustawionej na *On* powyższy kod spowoduje błąd w środowisku IDE, a program się nie skompiluje. Przy ustawieniu *Off* można skompilować ten fragment.

Podobny problem występuje w przypadku dzielenia. Przyjrzyj się poniższemu fragmentowi kodu:

```
' Próba dzielenia liczb.
intNumber = 12
intNumber = intNumber / 7
```

Tym razem dokładna odpowiedź to `1,71`. Jednak ponieważ wynik musi zostać zaokrąglony w celu zapisania go w zmiennej `intNumber`, program przypisze do tej zmiennej liczbę `2`. Jak możesz sobie wyobrazić, gdybyś chciał napisać program obliczający dokładne wartości, miałbyś duży problem, ponieważ każdy krok obliczeń byłby obciążony błędem związanym z zaokrąglaniem.

Następny punkt opisuje, jak można wykonać tego typu obliczenia za pomocą liczb zmiennoprzecinkowych.

Arytmetyka na liczbach zmiennoprzecinkowych

Już wiesz, że liczby całkowite nie są odpowiednie do większości obliczeń matematycznych, ponieważ większość takich obliczeń obejmuje części ułamkowe. W dalszej części tego rozdziału nauczysz się wykorzystywać liczby zmiennoprzecinkowe na przykładzie obliczania powierzchni koła. Poniższe ćwiczenie „Spróbuj sam" stanowi proste wprowadzenie do używania liczb zmiennoprzecinkowych.

SPRÓBUJ SAM Arytmetyka zmiennoprzecinkowa

Plik z kodem projektu Floating Point Math można pobrać z witryny helion.pl.

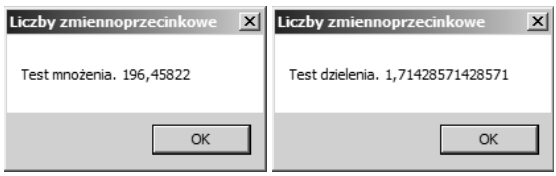
W tym ćwiczeniu „Spróbuj sam" utworzysz projekt, który mnoży i dzieli liczby zmiennoprzecinkowe.

1. Utwórz nowy projekt typu *Windows Forms Application* w środowisku Visual Studio 2010 i nazwij go **`Floating Point Math`**. Następnie umieść na formularzu przycisk, zmień jego nazwę na **`btnFloatMath`**, a właściwość `Text` — na **`Test`**.
2. Kliknij dwukrotnie przycisk `btnFloatMath` i dodaj wyróżniony pogrubieniem kod:

```
Private Sub btnFloatMath_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnFloatMath.Click

   ' Deklaracja zmiennej.
   Dim dblNumber As Double

   ' Przypisanie wartości, operacja mnożenia i wyświetlenie wyniku.
   dblNumber = 45.34
   dblNumber *= 4.333
   MessageBox.Show("Test mnożenia. " & dblNumber.ToString, _
                   "Liczby zmiennoprzecinkowe")

   ' Przypisanie wartości, operacja dzielenia i wyświetlenie wyniku.
   dblNumber = 12
   dblNumber /= 7
```

```
        MessageBox.Show("Test dzielenia. " & dblNumber.ToString, _
                        "Liczby zmiennoprzecinkowe")
    End Sub
```

3. Zapisz projekt przez kliknięcie przycisku *Save All* na pasku narzędzi.
4. Uruchom projekt. Powinieneś zobaczyć okna przedstawione na rysunku 3.5.



Rysunek 3.5. Operacje arytmetyczne na liczbach zmiennoprzecinkowych

Jak to działa?

Prawdopodobnie najważniejszą zmianą w porównaniu z wcześniejszym przykładem jest sposób deklaracji zmiennej:

```
' Deklaracja zmiennej.
Dim dblNumber As Double
```

Zamiast zwrotu `As Integer` powyższa deklaracja zawiera zwrot `As Double`. Informuje to język Visual Basic 2010, że zmienna ma przechowywać liczby zmiennoprzecinkowe o podwójnej precyzji, a nie liczby całkowite. Oznacza to, że wszystkie operacje wykonywane na zmiennej `dblNumber` to operacje zmiennoprzecinkowe, a nie całkowitoliczbowe. Zwróć także uwagę na inny przedrostek w zmodyfikowanej notacji węgierskiej, informujący, że zmienna przechowuje liczby typu `Double`.

Kod służący do wykonywania operacji arytmetycznych jest taki sam jak w przypadku liczb całkowitych. Poniższy kod przypisuje do zmiennej `dblNumber` liczbę ułamkową i mnoży ją przez inną liczbę tego typu:

```
' Przypisanie wartości, operacja mnożenia i wyświetlenie wyniku.
dblNumber = 45.34
dblNumber *= 4.333
MessageBox.Show("Test mnożenia. " & dblNumber.ToString, _
                "Liczby zmiennoprzecinkowe")
```

Wynik tej operacji to liczba `196,45822`, która zawiera szereg cyfr po przecinku. Następnie można użyć tej liczby do dalszych obliczeń.

Oczywiście przypisując wartość do liczby zmiennoprzecinkowej, nie trzeba jawnie podawać cyfr znajdujących się po przecinku:

```
' Przypisanie wartości, operacja dzielenia i wyświetlenie wyniku.
dblNumber = 12
dblNumber /= 7
MessageBox.Show("Test dzielenia. " & dblNumber.ToString, _
                "Liczby zmiennoprzecinkowe")
```

Wynik tego działania to także liczba zmiennoprzecinkowa, ponieważ zmienna `dblNumber` ma typ umożliwiający przechowywanie wyników tego typu. Świadczy o tym wyświetlany wynik — `1,71428571428571`. Tego samego wyniku mogłeś się spodziewać, wykonując tę samą operację na liczbach całkowitych.

Tu kod umożliwia zastosowanie operatora skróconego do dzielenia liczb, ponieważ zmienna na wynik może przechowywać wartości zmiennoprzecinkowe. Dlatego nie trzeba stosować funkcji `CType` do przekształcania wyników na liczbę całkowitą.

Liczby zmiennoprzecinkowe nazywają się tak ze względu na sposób zapisu, przypominający notację naukową. W notacji naukowej liczba składa się z potęgi liczby 10 pomnożonej przez liczbę z przedziału od 1 do 10. Wynik tego mnożenia to pierwotna liczba. Na przykład 10001 w notacji naukowej to $1{,}0001 * 10^4$, a 0,0010001 to $1{,}0001 * 10^{-3}$. Pozycja przecinka *zmienia się* tak, aby w obu przypadkach znajdowała się po pierwszej cyfrze. Zaleta notacji tego typu polega na tym, że duże i małe liczby można przedstawić z taką samą dokładnością (w tym przypadku z dokładnością do jednej dziesięciotysięcznej). Zmienne zmiennoprzecinkowe są przechowywane w pamięci komputera w ten sam sposób, ale ich podstawą jest liczba dwa, a nie 10. Więcej na ten temat dowiesz się w podrozdziale „Przechowywanie zmiennych" w dalszej części rozdziału.

Inne stany

Zmienne zmiennoprzecinkowe mogą reprezentować kilka innych wartości, nie tylko liczby ułamkowe. Są to między innymi stany:

- `NaN`, co jest skrótem od angielskiego *not a number*, czyli „to nie jest liczba".
- Dodatnią nieskończoność.
- Ujemną nieskończoność.

Ta książka nie przedstawia sposobów dochodzenia do takich wyników, ale matematycy na pewno stwierdzą, że platforma .NET spełnia ich zaawansowane potrzeby.

Liczby zmiennoprzecinkowe o pojedynczej precyzji

Nieco wcześniej pojawiło się pojęcie *liczba zmiennoprzecinkowa o podwójnej precyzji*. Platforma .NET udostępnia dwa sposoby reprezentowania liczb zmiennoprzecinkowych, których można używać w zależności od potrzeb. W niektórych przypadkach część ułamkowa liczby może ciągnąć się w nieskończoność (liczba *pi* jest szczególnie znanym przykładem), ale komputer nie ma nieskończonej ilości pamięci, w której mógłby przechowywać nieskończoną ilość cyfr, dlatego musi istnieć jakieś ograniczenie liczby cyfr po przecinku uwzględnianych przez komputer. To ograniczenie związane jest z rozmiarem zmiennej — to zagadnienie opisane jest dużo bardziej szczegółowo pod koniec rozdziału. Występują także ograniczenia wielkości składnika znajdującego się po lewej stronie od przecinka.

Liczby zmiennoprzecinkowe o podwójnej precyzji mogą przechowywać liczby od $-1{,}7*10^{308}$ do $1{,}7*10^{308}$ z wielką dokładnością (do jednego grosza na 45 miliardów złotych). Liczby zmiennoprzecinkowe o pojedynczej precyzji przechowują wartości od $-3{,}4*10^{38}$ do $3{,}4*10^{38}$. Także te liczby mogą być bardzo duże, ale przechowują mniej dokładne wartości (do jednego grosza na 330 tysięcy złotych). Liczby zmiennoprzecinkowe o pojedynczej precyzji potrzebują za to mniejszej ilości pamięci, dzięki czemu na niektórych komputerach obliczenia wykorzystujące liczby tego typu wykonywane są szybciej od obliczeń wykorzystujących liczby o podwójnej precyzji.

Powinieneś unikać używania liczb o podwójnej precyzji, dopóki naprawdę nie potrzebujesz dokładności większej niż zapewniana przez liczby o pojedynczej precyzji. Jest to szczególnie istotne w przypadku dużych systemów, gdzie użycie zmiennych o podwójnej precyzji zamiast liczby o pojedynczej precyzji może znacznie pogorszyć wydajność programu.

Wykonywane obliczenia zwykle dają wskazówkę co do typu liczb zmiennoprzecinkowych, którego należy użyć. Jeśli chcesz użyć liczby o pojedynczej precyzji, powinieneś zadeklarować zmienną typu `Single` zamiast `Double`:

```
Dim sngNumber As Single
```

Używanie ciągów znaków

Ciąg znaków (ang. *string*) to sekwencja znaków, którą umieszcza się w cudzysłowach. Widziałeś już, jak używać ciągów znaków do wyświetlania na ekranie wyników działania prostych programów. Ciągi znaków służą głównie do tego, informując użytkownika, co się stało, oraz o dalszym przebiegu programu. Innym częstym zastosowaniem ciągów znaków jest zapisywanie w nich fragmentów tekstu w celu późniejszego wykorzystania ich w algorytmie. W książce zobaczysz wiele ciągów znaków. Do tej pory używałeś ciągów znaków takiego typu:

```
MessageBox.Show("Test mnożenia. " & dblNumber.ToString, _
                "Liczby zmiennoprzecinkowe")
```

`"Test mnożenia. "` i `"Liczby zmiennoprzecinkowe"` to ciągi znaków. Informują o tym otaczające je cudzysłowy (`"`). Co jednak oznacza w tym kontekście `dblNumber`? Wartość przechowywana w zmiennej `dblNumber` jest przekształcana za pomocą metody `ToString` na ciąg znaków, który można wyświetlić na ekranie. `ToString` to metoda struktury `Double` definiującej typ zmiennej. Na przykład, aby wyświetlić zmienną `dblNumber` przechowującą wartość `27`, należy przekształcić tę wartość na ciąg o długości dwóch znaków. Właśnie w ten sposób działa metoda `ToString`.

SPRÓBUJ SAM Używanie ciągów znaków

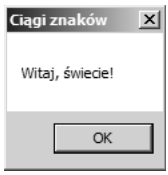
Plik z kodem projektu Strings można pobrać z witryny helion.pl.

To ćwiczenie „Spróbuj sam" ilustruje niektóre operacje, jakie można wykonać na ciągach znaków.

1. Utwórz nową aplikację typu *Windows Forms Application*, używając opcji menu *File/New/Project*. Nazwij ten projekt **`Strings`**.
2. Za pomocą okna narzędzi dodaj do formularza przycisk, nadaj mu nazwę **`btnStrings`** oraz zmień jego właściwość `Text` na **`Ciągi znaków`**. Kliknij dwukrotnie ten przycisk, a następnie dodaj wyróżniony pogrubieniem kod.

```
Private Sub btnStrings_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnStrings.Click
   ' Deklaracja zmiennej.
   Dim strResults As String

   ' Przypisywanie wartości do ciągu znaków.
   strResults = "Witaj, świecie!"

   ' Wyświetlanie tekstu.
   MessageBox.Show(strResults, "Ciągi znaków")
End Sub
```

3. Zapisz projekt przez kliknięcie przycisku *Save All* na pasku narzędzi.
4. Uruchom projekt i kliknij przycisk *Ciągi znaków*. Pojawi się okno komunikatu przedstawione na rysunku 3.6.



Rysunek 3.6. Wyświetlanie ciągu znaków

Jak to działa?

Do definiowania zmiennych przechowujących ciągi znaków używa się podobnego zapisu co w przypadku zmiennych liczbowych, jednak tym razem należy użyć instrukcji `As String`:

```
' Deklaracja zmiennej.
Dim strResults As String
```

Podobnie jak w przypadku zmiennych liczbowych do ciągów znaków można przypisać wartość:

```
' Przypisywanie wartości do ciągu znaków.
strResults = "Witaj, świecie!"
```

Trzeba użyć cudzysłowów wokół tekstu, aby *ograniczyć* ciąg znaków, czyli oznaczyć jego początek i koniec. Jest to istotne, ponieważ te cudzysłowy informują kompilator języka Visual Basic 2010, że nie powinien kompilować tekstu znajdującego się w ciągu znaków. Jeśli zapomnisz o cudzysłowach, język Visual Basic 2010 potraktuje wartość zmiennej jako część kodu programu i spróbuje ją skompilować. Ponieważ jest to niemożliwe, kompilacja programu zakończy się niepowodzeniem.

Po zapisaniu wartości `Witaj, świecie!` w zmiennej `strResults` można przekazać tę zmienną do okna komunikatu, które pobiera wartość zmiennej i wyświetla ją. Definiowanie i używanie zmiennych zawierających ciągi znaków nie różni się więc od definiowania i używania zmiennych liczbowych. Poniżej opisane są operacje, które można wykonywać na ciągach znaków.

Łączenie

Łączenie (ang. *concatenation*) oznacza wiązanie czegoś w łańcuch lub szereg. Jeśli używasz dwóch ciągów znaków, a następnie dołączasz jeden do drugiego, ciągi są połączone.

SPRÓBUJ SAM Łączenie ciągów znaków

Możesz traktować łączenie jak dodawanie ciągów znaków. W tym ćwiczeniu „Spróbuj sam” dowiesz się, jak przeprowadzać takie operacje.

1. Wykorzystaj utworzony wcześniej projekt `Strings`, otwórz okno projektowe z formularzem `Form1` i dodaj nowy przycisk. Nazwij go **`btnConcatenation`** i ustaw jego właściwość `Text` na **`Łączenie`**. Kliknij dwukrotnie ten przycisk i dodaj wyróżniony kod:

```
Private Sub btnConcatenation_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnConcatenation.Click
   ' Deklaracja zmiennych.
   Dim strResults As String
   Dim strOne As String
   Dim strTwo As String

   ' Przypisanie wartości do zmiennych.
   strOne = "Witaj,"
```

```
        strTwo = " świecie!"

        ' Łączenie ciągów znaków.
        strResults = strOne & strTwo

        ' Wyświetlanie wyniku.
        MessageBox.Show(strResults, "Ciągi znaków")
    End Sub
```

2. Uruchom projekt i kliknij przycisk *Łączenie*. Pojawi się okno znane już z rysunku 3.6.

Jak to działa?

To ćwiczenie „Spróbuj sam” rozpoczyna się od deklaracji trzech zmiennych przechowujących dane typu `String`:

```
' Deklaracja zmiennych.
Dim strOne As String
Dim strTwo As String
Dim strResults As String
```

Następnie program przypisuje wartości do dwóch pierwszych zmiennych:

```
' Przypisanie wartości do zmiennych.
strOne = "Witaj,"
strTwo = " świecie!"
```

Po przypisaniu wartości do dwóch pierwszych ciągów znaków program łączy je za pomocą operatora & i zapisuje wynik łączenia w trzeciej zmiennej o nazwie `strResults`:

```
' Łączenie ciągów znaków.
strResults = strOne & strTwo
```

Używanie operatora łączenia inline

Nie musisz definiować zmiennych, aby użyć operatora łączenia. Możesz użyć go „w locie”, jak ilustrują to projekty `Floating Point Math`, `Integer Math` i `Variables`. We wcześniejszych przykładach zobaczyłeś już, jak korzystać z operatora łączenia w ten sposób. Kod przekształcał w nich wartość zapisaną w zmiennej `dblNumber` na ciąg znaków, aby można go wyświetlić na ekranie. Przyjrzyj się następującemu fragmentowi kodu:

```
MessageBox.Show("Test dzielenia. " & dblNumber.ToString, _
                "Liczby zmiennoprzecinkowe")
```

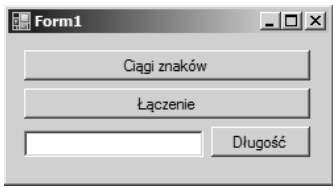
`"Test dzielenia. "` to ciąg znaków, jednak nie trzeba go definiować jako zmiennej tego typu. W słownictwie specyficznym dla języka Visual Basic 2010 jest to tak zwany *literał*, co oznacza, że ma on postać widoczną w kodzie i jest niezmienny. Przy stosowaniu operatora łączenia dla tego ciągu i wyrażenia `dblNumber.ToString` program przekształca wartość zmiennej `dblNumber` na ciąg znaków i umieszcza ją za ciągiem `"Test dzielenia. "`. Warto pamiętać, że metoda `ToString` przekształca wartość zmiennej na ciąg znaków. Wynik całej operacji to ciąg przekazywany do metody `MessageBox.Show`. Ciąg ten obejmuje zarówno podstawowy tekst, jak i obecną wartość zmiennej `dblNumber`.

Inne operacje na ciągach znaków

Na ciągach znaków można przeprowadzać także wiele innych operacji. Niektóre z nich opisuje następne ćwiczenie „Spróbuj sam”.

SPRÓBUJ SAM Zwracanie długości ciągu znaków

1. Najpierw nauczysz się sprawdzać długość ciągu znaków za pomocą służącej do tego właściwości.
2. W projekcie `Strings` otwórz formularz `Form1` w oknie projektowym, dodaj do niego pole tekstowe i nazwij je **`txtString`**. Następnie dodaj nowy przycisk, nazwij go **`btnLength`** i zmień jego właściwość `Text` na **`Długość`**. Uporządkuj kontrolki tak, aby wyglądały podobnie jak na rysunku 3.7.

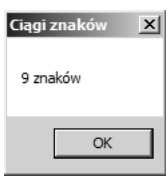


Rysunek 3.7. Formularz programu do wykonywania operacji na ciągach znaków

3. Kliknij dwukrotnie przycisk `Długość`, aby utworzyć metodę obsługi zdarzenia `Click`. Następnie dodaj do tej metody wyróżniony pogrubieniem kod:

```
Private Sub btnLength_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnLength.Click
   ' Deklaracja zmiennej.
   Dim strResults As String

   ' Pobranie tekstu z pola tekstowego.
   strResults = txtString.Text

   ' Wyświetlenie długości ciągu znaków.
   MessageBox.Show(strResults.Length.ToString & " znaków", _
                   "Ciągi znaków")
End Sub
```

4. Uruchom projekt i wpisz dowolny tekst w pole tekstowe.
5. Kliknij przycisk *Długość* — pojawi się okno podobne do tego na rysunku 3.8.



Rysunek 3.8. Wyświetlanie długości ciągu znaków

Jak to działa?

Na początku metoda deklaruje zmienną `na ciąg znaków`. Następnie pobiera tekst wpisany przez użytkownika w pole tekstowe i zapisuje go w zmiennej `strResults`.

```
' Deklaracja zmiennej.
Dim strResults As String

' Pobranie tekstu z pola tekstowego.
strResults = txtString.Text
```

Po pobraniu ciągu znaków można użyć właściwości `Length` do pobrania wartości określającej liczbę znaków w tym ciągu. Pamiętaj, że komputer liczy jako znaki także odstępy oraz znaki przestankowe. Ponieważ właściwość `Length` zwraca liczbę znaków jako wartość typu `Integer`, warto przekształcić ją na ciąg znaków za pomocą metody `ToString`:

```
' Wyświetlenie długości ciągu znaków.
MessageBox.Show(strResults.Length.ToString & " znaków", _
                "Ciągi znaków")
```

Podciągi

Popularne sposoby manipulowania ciągami znaków w programach obejmują używanie zbioru znaków pojawiających się na początku ciągu, zbioru znaków znajdujących się na końcu ciągu lub zbioru znaków pojawiających się w środku ciągu. Takie fragmenty to *podciągi*.

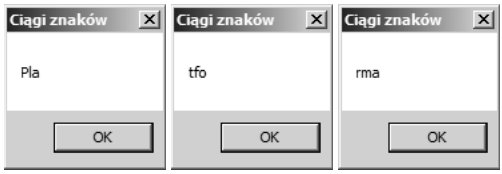
SPRÓBUJ SAM Używanie podciągów

To ćwiczenie „Spróbuj sam" opiera się na poprzedniej aplikacji i pokazuje, jak wyświetlić trzy pierwsze, trzy środkowe oraz trzy ostatnie znaki ciągu.

1. Otwórz formularz `Form1` w oknie projektowym w projekcie `Strings`. Dodaj kolejny przycisk do formularza `Form1`, nazwij go **`btnSplit`** i ustaw jego właściwość `Text` na **`Rozbij`**. Kliknij dwukrotnie ten przycisk i dodaj wyróżniony pogrubieniem kod:

```
Private Sub btnSplit_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnSplit.Click
   ' Deklaracja zmiennej.
   Dim strResults As String

   ' Pobieranie tekstu z pola tekstowego.
   strResults = txtString.Text

   ' Wyświetlanie trzech pierwszych znaków.
   MessageBox.Show(strResults.Substring(0, 3), "Ciągi znaków")

   ' Wyświetla trzy środkowe znaki.
   MessageBox.Show(strResults.Substring(3, 3), "Ciągi znaków")

   ' Wyświetla trzy ostatnie znaki.
   MessageBox.Show(strResults.Substring(strResults.Length - 3), "Ciągi znaków")
End Sub
```

2. Uruchom projekt. Wpisz w pole tekstowe słowo **`Platforma`**.
3. Kliknij przycisk *Rozbij* — pojawią się trzy okna komunikatu widoczne na rysunku 3.9.



Rysunek 3.9. Rozbijanie ciągu znaków na podciągi

Jak to działa?

Metoda `Substring` pozwala pobrać zbiór znaków znajdujących się w dowolnym miejscu ciągu. Tej metody można używać na dwa sposoby. Pierwszy sposób polega na podaniu początkowej pozycji oraz liczby pobieranych znaków. Pierwsza instrukcja informuje, że podciąg zaczyna się od pozycji zerowej, czyli od początku ciągu, oraz zawiera trzy znaki:

```
' Wyświetlanie trzech pierwszych znaków.
MessageBox.Show(strResults.Substring(0, 3), "Ciągi znaków")
```

Następna instrukcja pobiera podciąg zaczynający się od pozycji trzeciej i składający się z trzech znaków:

```
' Wyświetla trzy środkowe znaki.
MessageBox.Show(strResults.Substring(3, 3), "Ciągi znaków")
```

Ostatnia instrukcja przyjmuje tylko jeden parametr. W takiej sytuacji metoda `Substring` rozpoczyna pobieranie od podanej pozycji i pobiera wszystkie znaki do końca ciągu. W tym przypadku zostaje wykorzystana kombinacja metody `Substring` i właściwości `Length`, dlatego znaczenie tej instrukcji to: „pobierz wszystkie znaki ciągu od trzeciej pozycji od końca do końca”.

```
' Wyświetla trzy ostatnie znaki.
MessageBox.Show(strResults.Substring(strResults.Length - 3), "Ciągi znaków")
```

Formatowanie ciągów znaków

Liczby w ciągach znaków często wymagają zmiany sposobu ich wyświetlania. Rysunek 3.5 przedstawiał działanie operatora dzielenia. W tym przypadku nie jest potrzebne aż 14 miejsc po przecinku — dwa lub trzy wystarczą w zupełności. Można sformatować ciąg znaków tak, aby wyświetlał wszystkie znaki po lewej stronie przecinka, ale tylko trzy cyfry po jego prawej stronie. Następne ćwiczenie „Spróbuj sam” pokazuje, jak to zrobić.

SPRÓBUJ SAM Formatowanie ciągów znaków

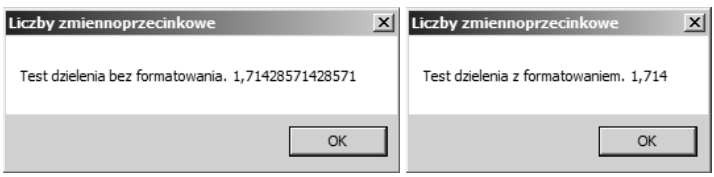
W tym ćwiczeniu „Spróbuj sam” zmodyfikujesz utworzony wcześniej projekt *Floating Point Math*, tak aby wyświetlał liczby w postaci ciągów o różnych formatach.

1. Otwórz projekt *Floating Point Math* utworzony na początku rozdziału.
2. Otwórz kod formularza `Form1` w edytorze kodu i wprowadź wyróżnione pogrubieniem zmiany w procedurze `btnFloatMath_Click`:

```
' Przypisuje wartość do zmiennej dblNumber i dzieli ją.
dblNumber = 12
dblNumber /= 7

' Wyświetla niesformatowany wynik.
```

```
        MessageBox.Show("Test dzielenia bez formatowania. " & _
                dblNumber.ToString, "Liczby zmiennoprzecinkowe")

        ' Wyświetla wynik po sformatowaniu.
        MessageBox.Show("Test dzielenia z formatowaniem. " & _
                String.Format("{0:n3}", dblNumber), "Liczby zmiennoprzecinkowe")
    End Sub
```

3. Uruchom projekt. Najpierw pojawi się okno z wynikiem testu mnożenia (z poprzedniego przykładu), a następnie dwa inne okna, co ilustruje rysunek 3.10.



Rysunek 3.10. Formatowanie ciągów znaków z liczbami

Jak to działa?

Sztuczka polega na wywołaniu metody `String.Format`. Ta użyteczna metoda pozwala formatować liczby. Podstawą działania tej metody jest jej pierwszy parametr, który definiuje format ciągu znaków:

```
MessageBox.Show("Test dzielenie z formatowaniem. " & _
   String.Format("{0:n3}", dblNumber), "Liczby zmiennoprzecinkowe")
```

Metoda `String.Format` w powyższym przykładzie przyjmuje dwa parametry. Pierwszy z nich, `"{0:n3}"`, określa pożądany format. Drugi parametr, `dblNumber`, to formatowana wartość. Warto zauważyć, że z uwagi na formatowanie liczby jako ciągu znaków nie trzeba stosować metody `ToString` po nazwie zmiennej `dblNumber`, co miało miejsce w poprzednim wywołaniu metody `Show` klasy `MessageBox`. Dzieje się tak, ponieważ metoda `String.Format` przyjmuje liczbę, a nie ciąg znaków.

`0` w parametrze określającym format informuje metodę `String.Format`, że formatowanie ma dotyczyć parametru danych o indeksie zero, czyli drugiego parametru metody, którym w tym przypadku jest zmienna `dblNumber`. Wartość znajdująca się po dwukropku określa sposób formatowania zmiennej `dblNumber`. `n3` oznacza, że formatowanie dotyczy liczby zmiennoprzecinkowej i ma uwzględniać trzy miejsca po przecinku. W celu wyświetlenia tylko dwóch liczb po przecinku należy użyć wartości `n2`.

Formatowanie według ustawień regionalnych

W czasie tworzenia aplikacji na platformę .NET trzeba pamiętać, że użytkownik może być przyzwyczajony do innych konwencji niż programista. Na przykład w Polsce do oddzielania całkowitej części liczby od części ułamkowej służy przecinek (`,`). Z kolei w Stanach Zjednoczonych używa się do tego kropki (`.`).

System Windows rozwiązuje takie problemy za programistów na podstawie ustawień regionalnych. Jeśli używasz platformy .NET w poprawny sposób, prawie nigdy nie musisz się przejmować takimi zagadnieniami.

Potwierdza to między innymi kolejny przykład. Użycie parametru `n3` do formatowania ciągu znaków informuje platformę .NET nie tylko o tym, że ma wyświetlać trzy cyfry po przecinku, ale także o konieczności używania separatorów co trzy cyfry po lewej stronie od przecinka (1 714,286).

UWAGA

Aby zobaczyć działanie separatora, w kodzie przykładu należy zmienić równanie z 12/7 na 12000/7.

Jeśli polecisz komputerowi użycie ustawień regionalnych Stanów Zjednoczonych i uruchomisz *ten sam kod* (nie wymaga to wprowadzania żadnych zmian w samej aplikacji), program wyświetli liczbę w innym formacie — `1,714.286`.

UWAGA

Możesz zmienić ustawienia regionalne, otwierając *Panel sterowania*, klikając ikonę *Opcje regionalne i językowe*, a następnie zmieniając język na *Angielski (Stany Zjednoczone)*.

W Stanach Zjednoczonych separator tysięcy to przecinek, a nie odstęp, a separator części ułamkowej to kropka, a nie przecinek. Używając metody `String.Format`, możesz pisać aplikacje działające poprawnie niezależnie od konfiguracji ustawień regionalnych na komputerze użytkownika.

Zamiana podciągów

Kolejną często stosowaną operacją na ciągach znaków jest zamiana wystąpień jednego ciągu na inny.

SPRÓBUJ SAM Zamiana podciągów

To ćwiczenie „Spróbuj sam" ilustruje zamianę podciągów. Dowiesz się tu, jak zmodyfikować aplikację *Strings* tak, aby zamieniała ciąg znaków `"Witaj"` na `"Żegnaj"`.

1. Otwórz program *Strings* utworzony w jednym z wcześniejszych ćwiczeń.
2. Otwórz w oknie projektowym formularz `Form1` i dodaj do niego kolejny przycisk. Zmień nazwę przycisku na **`btnReplace`** i ustaw właściwość `Text` na **`Zamień`**. Kliknij dwukrotnie ten przycisk i dodaj wyróżniony kod do metody obsługi zdarzenia `Click`:

```
Private Sub btnReplace_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnReplace.Click
   ' Deklaracja zmiennych.
   Dim strData As String
   Dim strResults As String

   ' Pobiera tekst z pola tekstowego.
   strData = txtString.Text

   ' Zamienia wystąpienia ciągu znaków Witaj.
   strResults = strData.Replace("Witaj", "Żegnaj")

   ' Wyświetla nowy ciąg znaków.
   MessageBox.Show(strResults, "Ciągi znaków")
End Sub
```

3. Uruchom projekt i wpisz ciąg **`Witaj, świecie!`** w polu tekstowym, zachowując wielkość liter.
4. Kliknij przycisk *Zamień*. Powinno pojawić się okno komunikatu z napisem `Żegnaj, świecie!`.

Jak to działa?

Metoda `Replace` wyszukuje w tekście ciąg znaków podany jako pierwszy parametr, a następnie zamienia jego wystąpienia na ciąg znaków podany jako drugi parametr. Po zamianie metoda zwraca nowy ciąg znaków, który można wyświetlić w zwykły sposób.

```
' Zamienia wystąpienia ciągu znaków Witaj.
strResults = strData.Replace("Witaj", "Żegnaj")
```

Nie musisz ograniczać się do pojedynczego wyszukiwania i zamiany. Jeśli wpiszesz dwa słowa **`Witaj`** w polu tekstowym i klikniesz przycisk *Zamień*, program wyświetli dwa słowa `Żegnaj`. Jednak musisz pamiętać o wielkości liter — ten program nie zamieni słowa **`witaj`**. Porównywanie ciągów znaków bez uwzględniania wielkości liter opisano w następnym rozdziale.

Używanie dat

Kolejny często używany typ danych to `Date`. Ten typ przechowuje wartości reprezentujące daty.

SPRÓBUJ SAM Wyświetlanie aktualnej daty

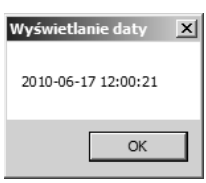
Plik z kodem projektu Date Demo można pobrać z witryny helion.pl.

To ćwiczenie „Spróbuj sam” pokazuje, jak wyświetlić aktualną datę.

1. Utwórz nową aplikację typu *Windows Forms Application* i nazwij ją **`Date Demo`**.
2. Za pomocą okna narzędzi dodaj do formularza nowy przycisk. Nazwij go **`btnShowDate`** i ustaw jego właściwość `Text` na **`Wyświetl datę`**.
3. Kliknij dwukrotnie przycisk, aby utworzyć metodę obsługi zdarzenia `Click`, a następnie dodaj wyróżniony pogrubieniem kod:

```
Private Sub btnShowDate_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnShowDate.Click

   ' Deklaracja zmiennej.
   Dim dteResults As Date

   ' Pobieranie aktualnej daty i czasu.
   dteResults = Now

   ' Wyświetlanie wyniku.
   MessageBox.Show(dteResults.ToString, "Wyświetlanie daty")
End Sub
```

4. Zapisz projekt przez kliknięcie przycisku *Save All* na pasku narzędzi.
5. Uruchom projekt i kliknij przycisk *Wyświetl datę*. Powinno pojawić się okno podobne do tego na rysunku 3.11. Dokładny format tekstu w oknie zależy od ustawień regionalnych.



Rysunek 3.11. Wyświetlanie aktualnej daty

Jak to działa?

Typ danych `Date` może służyć do przechowywania wartości reprezentujących datę i czas. Po utworzeniu zmiennej tego typu metoda inicjuje ją za pomocą aktualnej daty i czasu — służy do tego instrukcja `Now`. Następnie metoda wyświetla datę w oknie dialogowym. Warto zauważyć, że ponieważ program ma wyświetlać zmienną typu `Date` jako ciąg znaków, należy użyć metody `ToString` do przekształcenia wartości na taki ciąg:

```
' Deklaracja zmiennej.
Dim dteResults As Date

' Pobieranie aktualnej daty i czasu.
dteResults = Now

' Wyświetlanie wyniku.
MessageBox.Show(dteResults.ToString, "Wyświetlanie daty")
```

Typ danych `Date` nie różni się od innych typów danych, choć daje wiele możliwości. Kilka kolejnych punktów opisuje, jak manipulować datami i kontrolować sposób ich wyświetlania na ekranie.

Formatowanie dat

Widziałeś już jeden sposób formatowania daty. Domyślnie data po przekazaniu jej do metody `MessageBox.Show` zostanie wyświetlona w sposób przedstawiony na rysunku 3.11.

Ponieważ używana maszyna ma polskie ustawienia regionalne, data ma format rrrr-mm-dd, a czas jest wyświetlany w formacie 24-godzinnym. Jest to kolejny przykład wpływu ustawień regionalnych na formatowanie różnych typów danych. Jeśli zmienisz ustawienia regionalne komputera na Stany Zjednoczone, data będzie miała format m/d/rrrr, a czas będzie wyświetlany w formacie 12-godzinnym, na przykład 10/2/2006 10:48:06 AM.

Choć możesz kontrolować sposób wyświetlania daty i czasu, najlepiej polegać pod tym względem na platformie .NET. Dzięki temu dane są automatycznie wyświetlane w formacie dogodnym dla użytkownika.

SPRÓBUJ SAM Formatowanie dat

To ćwiczenie „Spróbuj sam” przedstawia cztery użyteczne metody, które umożliwiają formatowanie dat.

1. Otwórz kod formularza `Form1` w edytorze kodu, znajdź metodę obsługi zdarzenia `Click` przycisku, a następnie dodaj do niej wyróżniony pogrubieniem kod:

```
' Wyświetlanie wyniku.
MessageBox.Show(dteResults.ToString., "Wyświetlanie daty")

' Wyświetlanie dat.
MessageBox.Show(dteResults.ToLongDateString, "Wyświetlanie daty")
MessageBox.Show(dteResults.ToShortDateString, "Wyświetlanie daty")

' Wyświetlanie czasu.
MessageBox.Show(dteResults.ToLongTimeString, "Wyświetlanie daty")
MessageBox.Show(dteResults.ToShortTimeString, "Wyświetlanie daty")
```

2. Uruchom projekt. Pojawi się pięć kolejnych okien. Pierwsze okno dialogowe już widziałeś — wyświetla datę i czas według ustawień regionalnych komputera. Drugie okno wyświetla datę w długiej postaci, a trzecie — w skróconej. Czwarte okno wyświetla czas w długim formacie, a ostatnie — w skróconej.

Jak to działa?

Powyższe ćwiczenie pokazuje cztery sposoby, na które można wyświetlać datę i czas w aplikacjach dla systemu Windows. Są to: pełna data, skrócona data, pełny czas i skrócony czas. Nazwy formatów nie wymagają wyjaśnień:

```
' Wyświetlanie dat.
MessageBox.Show(dteResults.ToLongDateString, "Wyświetlanie daty")
MessageBox.Show(dteResults.ToShortDateString, "Wyświetlanie daty")

' Wyświetlanie czasu.
MessageBox.Show(dteResults.ToLongTimeString, "Wyświetlanie daty")
MessageBox.Show(dteResults.ToShortTimeString, "Wyświetlanie daty")
```

Pobieranie składników daty i czasu

Kiedy używasz zmiennej typu `Date`, możesz użyć kilku właściwości, które pozwalają pobrać składniki daty i czasu. Warto im się przyjrzeć.

SPRÓBUJ SAM Pobieranie składników daty i czasu

W tym ćwiczeniu „Spróbuj sam” zobaczysz, jak pobierać składniki daty i czasu zapisane w zmiennych typu `Date`.

1. Wróć do okna projektowego w projekcie *Date Demo* i dodaj do formularza kolejny przycisk. Nazwij go **`btnDateProperties`** i ustaw jego właściwość `Text` na **`Właściwości daty`**. Kliknij dwukrotnie ten przycisk i dodaj wyróżniony pogrubieniem kod do metody obsługi zdarzenia `Click`:

```
Private Sub btnDateProperties_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnDateProperties.Click
   ' Deklaracja zmiennej.
   Dim dteResults As Date

   ' Pobranie aktualnej daty i czasu.
   dteResults = Now

   ' Wyświetlanie różnych właściwości daty.
   MessageBox.Show("Miesiąc: " & dteResults.Month, "Wyświetlanie daty")
   MessageBox.Show("Dzień: " & dteResults.Day, "Wyświetlanie daty")
   MessageBox.Show("Rok: " & dteResults.Year, "Wyświetlanie daty")
   MessageBox.Show("Godzina: " & dteResults.Hour, "Wyświetlanie daty")
   MessageBox.Show("Minuta: " & dteResults.Minute, "Wyświetlanie daty")
   MessageBox.Show("Sekunda: " & dteResults.Second, "Wyświetlanie daty")
   MessageBox.Show("Dzień tygodnia: " & dteResults.DayOfWeek, "Wyświetlanie daty")
   MessageBox.Show("Dzień roku: " & dteResults.DayOfYear, "Wyświetlanie daty")
End Sub
```

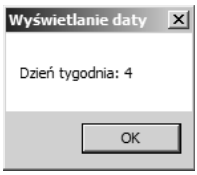
2. Uruchom projekt. Po kliknięciu przycisku zobaczysz zrozumiałe okna dialogowe.

Jak to działa?

Działanie tego kodu jest całkiem proste. Jeśli chcesz pobrać godzinę, musisz użyć właściwości `Hour`. Aby pobrać rok, używasz właściwości `Year` i tak dalej.

Stałe w datach

W poprzednim ćwiczeniu „Spróbuj sam" właściwość `DayOfWeek` zwracała liczbę całkowitą, co widać na rysunku 3.12.



Rysunek 3.12. Numer dnia tygodnia

Aktualna data to `17 czerwca 2010 roku`, jest `czwartek`, a program wyświetla liczbę `4`. Pierwszym dniem tygodnia w Polsce jest niedziela, a odliczanie rozpoczyna się od zera, dlatego czwartek ma wartość `4`. Możliwe jednak, że używasz komputera, na którym ustawienia regionalne określają poniedziałek jako pierwszy dzień tygodnia. W takiej sytuacji właściwość `DayOfWeek` zwróciłaby wartość `3`. Skomplikowane? Możliwe, jednak wystarczy zapamiętać, że dzień numer jeden to nie zawsze poniedziałek. Podobna sytuacja ma miejsce w języku naturalnym — polska środa to po angielsku Wednesday.

SPRÓBUJ SAM Pobieranie nazw dni i miesięcy

Jeśli chcesz pobrać nazwę dnia lub miesiąca, najlepiej jest użyć do tego platformy .NET, która zwróci odpowiednią nazwę na podstawie ustawień regionalnych komputera. Opisuje to poniższe ćwiczenie „Spróbuj sam".

1. Wróć do okna projektowego w projekcie *Date Demo* i dodaj do formularza nowy przycisk. Nazwij go **`btnDateNames`** i zmień jego właściwość `Text` na **`Nazwy dni i miesięcy`**. Kliknij dwukrotnie przycisk i dodaj wyróżniony pogrubieniem kod do metody obsługi zdarzenia `Click`:

```
Private Sub btnDateNames_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnDateNames.Click
   ' Deklaracja zmiennej.
   Dim dteResults As Date

   ' Pobranie aktualnej daty i czasu.
   dteResults = Now

   MessageBox.Show("Nazwa dnia tygodnia: " & dteData.ToString("dddd"), _
      "Wyświetlanie daty")
   MessageBox.Show("Nazwa miesiąca: " & dteData.ToString("MMMM"), _
      "Wyświetlanie daty")
End Sub
```

2. Uruchom projekt i kliknij przycisk. Zobaczysz dwa okna komunikatu. Jedno wyświetla nazwę dnia tygodnia (na przykład „Poniedziałek”), a drugie — nazwę miesiąca (na przykład „Wrzesień”).

Jak to działa?

Kiedy używasz metod z rodziny `ToLongDateString`, pozwalasz platformie .NET sprawdzić preferowany przez użytkownika format daty w ustawieniach regionalnych komputera. W ostatnim przykładzie użyłeś metody `ToString`, ale podałeś własny sposób formatowania danych.

```
MessageBox.Show("Nazwa dnia tygodnia: " & dteData.ToString("dddd"), _
   "Wyświetlanie daty")
MessageBox.Show("Nazwa miesiąca: " & dteData.ToString("MMMM"), _
   "Wyświetlanie daty")
```

Zwykle nie zaleca się używania metody `ToString` do formatowania dat, ponieważ lepiej polegać na wbudowanych formatach platformy .NET. Ciąg `"dddd"` pozwala wyświetlić nazwę dnia tygodnia, a `"MMMM"` reprezentuje nazwę miesiąca. Wielkość znaków jest istotna — ciąg znaków `"mmmm"` nie zadziała.

Aby zobaczyć działanie tej techniki, możesz zmienić ustawienia regionalne na angielski. Dowiesz się wtedy na przykład, że dzień tygodnia to `Friday`, a nazwa miesiąca to `February`.

Definiowanie literałów dat

Wiesz już, że literały ciągów znaków definiuje się w następujący sposób:

```
Dim strResults As String
strResults = "Hura"
```

Literały dat funkcjonują w podobny sposób, jednak do oznaczenia początku i końca daty służy znak kratki (#).

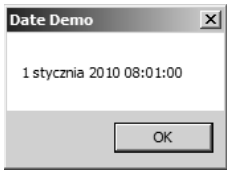
SPRÓBUJ SAM Definiowanie literałów dat

W tym ćwiczeniu „Spróbuj sam” dowiesz się, jak definiować literały dat.

1. Wróć do okna projektowego w projekcie *Date Demo* i dodaj do formularza następny przycisk. Zmień jego nazwę na **`btnDateLiterals`**, a właściwość `Text` — na **`Literały dat`**. Kliknij dwukrotnie przycisk i dodaj do metody obsługi zdarzenia `Click` wyróżniony pogrubieniem kod:

```
Private Sub btnDateLiterals_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnDateLiterals.Click
   ' Deklaracja zmiennych.
   Dim dteResults As Date

   ' Przypisywanie do zmiennej daty i czasu.
   dteResults = #1/1/2010 8:01:00 AM#

   ' Wyświetlanie daty i czasu.
   MessageBox.Show(dteResults.ToLongDateString & " " & _
      dteResults.ToLongTimeString, "Wyświetlanie daty")
End Sub
```

2. Uruchom projekt i kliknij przycisk. Powinien pojawić się komunikat widoczny na rysunku 3.13.



Rysunek 3.13. Wyświetlanie literału daty

Jak to działa?

Definiując literały dat, trzeba używać formatu mm/dd/rrrr, niezależnie od ustawień regionalnych komputera. Kompilator może nie zgłosić błędu, jeśli zdefiniujesz datę w formacie dd/mm/rrrr, ponieważ dana data może być poprawna w obu formatach (na przykład 06/07/2010). Ten wymóg pozwala uniknąć wieloznaczności — czy 6/7/2010 oznacza szósty lipca, czy siódmy czerwca?

UWAGA

W praktyce ta zasada dotyczy wszystkich aspektów programowania — nie ma czegoś takiego jak dialekty języków programowania. Zwykle najlepiej stosować się do standardów północnoamerykańskich. Jak dowiesz się z lektury dalszych rozdziałów tej książki, dotyczy to także nazw zmiennych i metod — na przykład używa się nazwy `GetColor` (amerykański angielski) zamiast `GetColour` (brytyjski angielski).

Warto także pamiętać, że nie trzeba wyświetlać jednocześnie daty i czasu. Można wyświetlić tylko jedną z tych informacji.

Manipulowanie datami

Jednym z zagadnień, które zawsze sprawiały pewne problemy programistom, jest manipulowanie datami. Pamiętasz przeddzień roku 2000, kiedy wszyscy obawiali się, czy komputery poradzą sobie z nadejściem nowego tysiąclecia? Kłopoty może sprawiać także obsługa lat przestępnych.

Najbliższy przełom stuleci, który przypada na rok przestępny, to przejście między rokiem 2399 a 2400. Poniższe ćwiczenie „Spróbuj sam” opisuje, jak użyć pewnych metod typu danych `Date` w celu przystosowania daty tak, aby płynnie przeszła w XXV wiek.

SPRÓBUJ SAM Manipulowanie datami

1. Wróć do okna projektowego w projekcie *Date Demo* i dodaj do formularza kolejny przycisk. Nazwij go **`btnDateManipulation`** i zmień jego właściwość `Text` na **`Manipulacja datami`**. Kliknij dwukrotnie ten przycisk i dodaj do metody obsługi jego zdarzenia `Click` wyróżniony pogrubieniem kod:

```
Private Sub btnDateManipulation_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnDateManipulation.Click
   ' Deklaracja zmiennych.
   Dim dteStartDate As Date
   Dim dteChangedDate As Date

   ' Wybrany dzień 2400 roku.
   dteStartDate = #2/28/2400#
```

```
    ' Dodanie dnia i wyświetlenie wyniku.
    dteChangedDate = dteStartDate.AddDays(1)
    MessageBox.Show(dteChangedDate.ToLongDateString, "Wyświetlanie daty")

    ' Dodanie kilku miesięcy i wyświetlenie wyniku.
    dteChangedDate = dteStartDate.AddMonths(6)
    MessageBox.Show(dteChangedDate.ToLongDateString, "Wyświetlanie daty")

    ' Odjęcie roku i wyświetlenie wyniku.
    dteChangedDate = dteStartDate.AddYears(-1)
    MessageBox.Show(dteChangedDate.ToLongDateString, "Wyświetlanie daty")
End Sub
```

2. Uruchom projekt i kliknij przycisk. Kolejno pojawią się trzy okna komunikatu. Pierwsze okno wyświetli długą wersję daty `2400-02-29`, drugie okno pokaże długą wersję daty `2400-08-28`, a w ostatnim oknie zobaczysz długą wersję daty `2399-02-28`.

Jak to działa?

Klasa `Date` udostępnia kilka metod służących do manipulowania datami. W poprzednim ćwiczeniu „Spróbuj sam” użyłeś trzech z nich:

```
' Dodanie dnia i wyświetlenie wyniku.
dteChangedDate = dteStartDate.AddDays(1)
MessageBox.Show(dteChangedDate.ToLongDateString, "Date Demo")

' Dodanie kilku miesięcy i wyświetlenie wyniku.
dteChangedDate = dteStartDate.AddMonths(6)
MessageBox.Show(dteChangedDate.ToLongDateString, "Date Demo")

' Odjęcie roku i wyświetlenie wyniku.
dteChangedDate = dteStartDate.AddYears(-1)
MessageBox.Show(dteChangedDate.ToLongDateString, "Date Demo")
```

Warto zauważyć, że przekazanie liczby ujemnej do metody `Add` zmiennej typu `Date` powoduje odjęcie odpowiedniej liczby jednostek czasu (w przykładzie rok zmienił się z `2400` na `2399`). Inne istotne metody z rodziny `Add` to `AddHours` (dodawanie godzin), `AddMinutes` (dodawanie minut), `AddSeconds` (dodawanie sekund) i `AddMilliseconds` (dodawanie milisekund).

Zmienne logiczne

Dotąd poznałeś typy danych `Integer`, `Double`, `Single`, `String` i `Date`. Inny ważny typ danych to `Boolean`. Gdy poznasz ten typ danych, będziesz umiał używać wszystkich podstawowych typów danych najczęściej stosowanych w programach.

Zmienne typu `Boolean` (zmienne logiczne) mają zawsze jedną z dwóch wartości: `True` lub `False`. Zmienne logiczne są bardzo istotne, kiedy programy mają podejmować decyzje, co bardziej szczegółowo opisuje rozdział 4.

Przechowywanie zmiennych

Najbardziej ograniczonym zasobem komputera jest zwykle pamięć. Powinieneś starać się jak najlepiej wykorzystać dostępną ilość pamięci. Kiedy tworzysz zmienną, zajmujesz fragment pamięci, dlatego musisz starać się używać jak najmniejszej liczby zmiennych i używać tych, których musisz, w jak najwydajniejszy sposób.

Obecnie nie musisz poznawać tajników maksymalnej optymalizacji wykorzystania zmiennych. Dzieje się tak z dwóch powodów. Po pierwsze, współczesne komputery mają zwykle dużo pamięci i minęły już czasy, kiedy programiści musieli męczyć się z obsługą systemów płatniczych przy użyciu 32 KB pamięci. Po drugie, nowoczesne kompilatory cechują się pewną „inteligencją", dzięki czemu potrafią utworzyć zoptymalizowany kod.

System dwójkowy

Komputery używają systemu dwójkowego (binarnego) do reprezentowania wszystkich informacji. Oznacza to, że wszystkie informacje, które zapisujesz w komputerze, są w formacie dwójkowym, czyli mają postać zer i jedynek. Poniższy przykład pokazuje zapis liczby 27. W kodzie dwójkowym ta liczba ma postać 11011, gdzie każda cyfra reprezentuje potęgę dwójki. Górny diagram na rysunku 3.14 przedstawia reprezentację liczby 27 w znanym wszystkim systemie dziesiętnym, a dolny — w systemie dwójkowym.

10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0
10 000 000	1 000 000	100 000	10 000	1 000	100	10	1
0	0	0	0	0	0	2	7

$2 \times 10 + 7 \times 1 = 27$

W systemie o podstawie 10 każda cyfra reprezentuje potęge liczby 10. Aby określić wartość reprezentowaną przez „wzorzec cyfr o podstawie 10", trzeba pomnożyć każdą cyfrę przez odpowiadającą jej pozycji potęgę liczby 10 i dodać wyniki

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	0	1	1	0	1	1

$1 \times 16 + 1 \times 8 + 1 \times 2 + 1 \times 1 = 27$

W systemie o podstawie 2, zwanym systemem dwójkowym, każda cyfra reprezentuje potęge liczby 2. Aby określić wartość reprezentowaną przez „wzorzec cyfr o podstawie 2", trzeba pomnożyć każdą cyfrę przez odpowiadającą jej pozycji potęgę dwójki i dodać wyniki

Rysunek 3.14. Obliczanie wartości liczb w systemach: dziesiętnym i dwójkowym

Choć w pierwszej chwili może się to wydawać nieco niezrozumiałe, spróbuj zastanowić się nad funkcjonowaniem systemu dziesiętnego. W systemie dziesiętnym (o podstawie 10) każda cyfra znajduje się na pewnej *pozycji*. Ta pozycja reprezentuje potęgę liczby 10. Pierwsza pozycja reprezentuje 10 do potęgi zerowej, następna 10 do potęgi pierwszej i tak dalej. Jeśli chcesz wiedzieć, jaką liczbę reprezentuje dany układ cyfr, musisz sprawdzić cyfry znajdujące się na kolejnych pozycjach, pomnożyć je przez potęgę liczby 10 przypisaną do danej pozycji i dodać wyniki.

Tak samo działa system dwójkowy. Trudniej go odczytać, ponieważ większość osób nie jest przyzwyczajona do liczb o podstawie dwa. Aby przekształcić liczbę z systemu dwójkowego na system dziesiętny, należy sprawdzić cyfry znajdujące się na kolejnych pozycjach (zera lub jedynki), pomnożyć je przez potęgę liczby *dwa* przypisaną do danej pozycji i dodać wyniki. Suma tych wyników to liczba w systemie dziesiętnym.

Bity i bajty

W komputerze każda pozycja przyjmująca wartość zero lub jeden to *bit*. Jest to najmniejsza jednostka informacji, stanowiąca odpowiedź na pytanie tak – nie i reprezentowana w elemencie układu komputera, przez który prąd płynie lub nie. Liczba przedstawiona na rysunku 3.14 zajmuje osiem pozycji (bitów), a osiem bitów składa się na jeden *bajt*. Bajt to jednostka miary służąca do opisu pamięci komputera.

Kilobajt (KB) to 1024 bajty. Dlaczego 1024, a nie 1000? Ponieważ komputery używają systemu dwójkowego, a 2^{10} to właśnie liczba 1024, która jest dla komputera „okrągłą liczbą". Komputery nie używają liczb o podstawie 10, dlatego bardziej naturalna jest dla nich liczba 1024 niż 1000.

Podobnie *megabajt* to 1024 kilobajty lub 1 048 576 bajtów. Jest to kolejna okrągła liczba, ponieważ jest to 2^{20}. *Gigabajt* to 1024 megabajty lub 1 073 741 824 bajtów. Ponownie jest to potęga liczby 2, tym razem 2^{30}. Wreszcie *terabajt* to 2^{40}, a *petabajt* to 2^{50}.

Po co te wszystkie informacje o liczbach? Znajomość sposobów przechowywania zmiennych w pamięci komputera pomaga projektować lepsze programy. Jeśli używasz komputera z 256 megabajtami pamięci, możesz wykorzystać w programie 262 144 KB (lub 268 435 456 bajtów, czyli 2 147 483 648 bitów). Kiedy piszesz aplikację, powinieneś starać się jak najlepiej wykorzystać dostępną pamięć.

Reprezentowanie wartości

Większość współczesnych komputerów to komputery 32-bitowe, co oznacza, że są zoptymalizowane pod kątem liczb całkowitych o długości 32 bitów. Liczba przedstawiona w poprzednim przykładzie to liczba 8-bitowa. W przypadku liczb 8-bitowych największą możliwą wartością jest 255:

```
1x128 + 1x64 + 1x32 + 1x16 + 1x4 + 1x2 + 1x1 = 255
```

Liczby 32-bitowe mogą reprezentować dowolną wartość z przedziału od -2 147 483 648 do 2 147 483 647. Takie właśnie liczby mogą przechowywać zmienne typu `Integer`:

```
Dim intNumber As Integer
```

W wyniku takiej deklaracji platforma .NET przydziela 32-bitowy blok pamięci, który może zawierać dowolną wartość z przedziału od 0 do 2 147 483 647. Pamiętaj także, że dostępna ilość pamięci jest ograniczona. Na komputerze z 256 megabajtami pamięci możesz zapisać maksymalnie 67 108 864 liczby całkowite typu `Integer`. Wydaje się, że to dużo, ale pamiętaj, że pamięć jest używana przez wszystkie programy. Nie powinieneś pisać programów, które zajmują tyle pamięci, ile tylko mogą. Staraj się oszczędnie gospodarować pamięcią.

Możesz także definiować liczby zmiennoprzecinkowe o podwójnej precyzji, na przykład takie:

```
Dim dblNumber As Double
```

Do reprezentowania liczby tego typu potrzeba 64 bitów pamięci. Oznacza to, że na komputerze z 256 megabajtami pamięci można zapisać 33 554 432 liczby zmiennoprzecinkowe o podwójnej precyzji.

UWAGA

Liczby zmiennoprzecinkowe o pojedynczej precyzji zajmują 32 bity pamięci. Jest to połowa tego, co liczby zmiennoprzecinkowe o podwójnej precyzji, a także tyle samo, co liczby całkowite.

Jeśli zdefiniujesz zmienną typu `Integer`, zawsze zajmuje ona taką samą ilość pamięci (32 bity) niezależnie od tego, jaką liczbę w niej przechowujesz — 1, 3, 249 czy 2 147 483 647. Wielkość liczby nie ma żadnego wpływu na ilość pamięci potrzebnej do jej przechowywania. Może się to wydawać wielkim marnotrawstwem, ale komputery wymagają, aby liczby tego samego typu zajmowały taką samą ilość pamięci. W przeciwnym razie szybkość działania programów znacznie by się zmniejszyła.

Przyjrzyj się teraz przykładowej definicji ciągu znaków:

```
Dim strResults As String
strResults = "Witaj, świecie!"
```

W przeciwieństwie do typów `Integer` czy `Double` zmienne typu `String` nie mają stałej długości. Każdy znak w ciągu znaków zajmuje dwa bajty, czyli 16 bitów. Dlatego do reprezentowania ciągu znaków o długości 12 znaków potrzeba 24 bajtów, czyli 192 bitów. Oznacza to, że na komputerze z 256 megabajtami pamięci można zapisać niewiele ponad dwa miliony ciągów znaków o takiej długości. Oczywiście dwa razy dłuższy ciąg znaków wymaga dwukrotnie więcej pamięci i tak dalej.

Częstym błędem popełnianym przez początkujących programistów jest nieuwzględnianie wpływu używanego typu danych na rozmiar zajmowanej pamięci. Jeśli używasz zmiennej typu `String` do przechowywania wartości liczbowych, na przykład w taki sposób:

```
Dim strData As String
strData = "65536"
```

do jej zapisania potrzebujesz 10 bajtów (80 bitów). Jest to mniej wydajne rozwiązanie od przechowywania tej wartości w zmiennej typu `Integer`. Aby możliwe było zapisanie liczby w ciągu znaków, komputer musi przekształcić wszystkie cyfry na reprezentujące je znaki. Znaki te są zapisywane w standardzie *Unicode*, który definiuje sposób przechowywania znaków na komputerze. Każdemu znakowi odpowiada niepowtarzalna liczba z przedziału od 0 do 65 535 i to właśnie ta liczba jest przechowywana w każdym bajcie ciągu znaków.

Poniżej przedstawione są kody standardu Unicode reprezentujące cyfry wchodzące w skład przykładowego ciągu znaków:

- „6” — Unicode 54, w systemie dwójkowym 00000000 00110110
- „5” — Unicode 53, w systemie dwójkowym 00000000 00110101
- „5” — Unicode 53, w systemie dwójkowym 00000000 00110101
- „3” — Unicode 51, w systemie dwójkowym 00000000 00110011
- „6” — Unicode 54, w systemie dwójkowym 00000000 00110110

Każdy znak zajmuje 16 bitów, dlatego do zapisania 5-cyfrowej liczby potrzeba 80 bitów — pięciu 16-bitowych liczb. Lepszym rozwiązaniem jest zadeklarowanie tej liczby w następujący sposób:

```
Dim intNumber As Integer
intNumber = 65536
```

Powoduje to zapisanie wartości jako jednej liczby w systemie dwójkowym. Zmienne typu `Integer` zajmują 32 bity, dlatego reprezentacja powyższej liczby to 00000000 00000001 00000000 00000000, co zajmuje dużo mniej miejsca niż reprezentacja tej samej liczby w postaci ciągu znaków.

Przekształcanie wartości

Choć ciągi znaków są intuicyjne w użyciu dla ludzi, są nienaturalne dla komputerów. Naturalnym działaniem komputera jest pobranie dwóch liczb i wykonanie na nich pewnych podstawowych matematycznych operacji. Komputer potrafi w ciągu sekundy wykonać tak dużą liczbę takich prostych operacji, że użytkownik szybko otrzymuje żądane wyniki.

Wyobraź sobie, że komputer musi dodać liczbę 1 do liczby 27. Wiesz już, że liczba 27 w systemie dwójkowym to 11011. Rysunek 3.15 przedstawia operację dodawania.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	0	1	1	0	1	1

$1 \times 16 + 1 \times 8 + 1 \times 2 + 1 \times 1 = 27$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
128	64	32	16	8	4	2	1	
0	0	0	1	1	1	0	0	← Dodawanie 1

Przenoszenie 1 Przenoszenie 1

Podobnie jak w zwykłej arytmetyce, jeśli cyfra na danej pozycji dojdzie do górnej granicy podstawy systemu (w tym przypadku do 2), należy ustawić ją na zero i przenieść jedynkę

$1 \times 16 + 1 \times 8 + 1 \times 4 = 28$

Rysunek 3.15. Dodawanie liczb w systemie dwójkowym

Jak widać, arytmetyka na liczbach w systemie dwójkowym nie różni się od arytmetyki w systemie dziesiętnym. Nie można dodać jedności do ostatniej pozycji, ponieważ w systemie dwójkowym cyfry mogą przyjmować wartości zero lub jeden, dlatego trzeba zmienić cyfrę na ostatniej pozycji na zero i przenieść jedynkę na następną pozycję. Także tu nie można dodać jedności, więc ponownie należy zmienić bit na zero i przenieść jedynkę na następną pozycję. W tym momencie dodawanie się kończy, a wartość nowej liczby to zgodnie z oczekiwaniami 28.

Komputer przed wykonaniem operacji na wartości musi ją przekształcić na proste liczby. W celu poprawy wydajności aplikacji należy starać się minimalizować liczbę konwersji. Poniżej znajduje się prosty przykład:

```
Dim strResults As String
strResults = "27"
strResults = strResults + 1
MessageBox.Show(strResults)
```

Zastanów się, jak działa powyższy fragment kodu:

1. Program tworzy zmienną typu `String` o nazwie `strResults`.
2. Program przypisuje do tej zmiennej wartość `27`. Wymaga to czterech bajtów pamięci.
3. Aby dodać jeden do tej wartości, komputer musi przekształcić ciąg znaków `"27"` na wewnętrzną, ukrytą zmienną typu `Integer`, która zawiera wartość `27`. Powoduje to zajęcie dodatkowych czterech bajtów pamięci, co razem daje 8 bajtów. Ponadto, co istotniejsze, przekształcanie wymaga czasu.
4. Po przekształceniu ciągu znaków na liczbę komputer dodaje do niej jeden.
5. Następnie komputer musi przekształcić nową wartość z powrotem na ciąg znaków.
6. W końcu program wyświetla nową wartość na ekranie.

Wydajne programy powinny wymagać jak najmniejszej liczby konwersji zmiennych między różnymi typami danych. Przekształcenia powinny mieć miejsce tylko wtedy, kiedy jest to konieczne.

Poniższy kod działa tak samo jak wcześniejszy przykład:

```
Dim intNumber As Integer
intNumber = 27
intNumber = intNumber + 1
MessageBox.Show(intNumber.ToString)
```

1. Program tworzy zmienną `intNumber` typu `Integer`.
2. Program przypisuje do tej zmiennej `27`.
3. Program dodaje do tej zmiennej `1`.
4. Następuje przekształcenie zmiennej na ciąg znaków i wyświetlenie jej na ekranie.

W tym przypadku potrzebna jest tylko jedna konwersja, i to uzasadniona. Metoda `MessageBox.Show` używa ciągów znaków, dlatego trzeba przekształcić liczbę na odpowiedni typ.

Jedna prosta zmiana w kodzie pozwoliła zmniejszyć liczbę konwersji z dwóch (z ciągu znaków na liczbę i z powrotem na ciąg znaków) do jednej. Dzięki temu program będzie działał bardziej wydajniej i zużyje mniej pamięci. Poprawa wydajności w tym przypadku nie jest wielka, jednak wyobraź sobie, że taka operacja ma miejsce setki tysięcy razy na minutę. Zapewni to poprawę wydajności systemu jako całości.

UWAGA

Niezwykle istotne jest, abyś używał typów danych odpowiednich do wykonywanych zadań. W prostych programach, takich jak te przedstawione w tym rozdziale, utrata wydajności nie jest widoczna. Jednak kiedy zaczniesz pisać bardziej złożone, zaawansowane aplikacje, powinieneś zadbać o optymalizację kodu i używać odpowiednich typów danych.

Metody

Metoda to samodzielny blok kodu, który wykonuje jakieś operacje. Metody są kluczowe z dwóch powodów. Po pierwsze, pozwalają podzielić program na mniejsze i bardziej zrozumiałe fragmenty. Po drugie, ułatwiają *powtórne wykorzystanie* kodu (to zagadnienie wielokrotnie pojawia się w dalszych rozdziałach tej książki).

Jak już wiesz, kiedy zaczynasz pisać program, powinieneś zacząć od utworzenia ogólnego algorytmu, a następnie dopracowywać jego szczegóły do czasu powstania reprezentującego go kodu. Metoda opisuje instrukcję algorytmu, na przykład „otwórz plik", „wyświetl tekst na ekranie", „wydrukuj dokument" i tak dalej.

Umiejętność rozkładania programu na metody programista nabywa wraz z doświadczeniem. Co gorsza, dużo łatwiej zrozumieć, dlaczego należy używać metod, kiedy pisze się dużo bardziej skomplikowane programy niż te przedstawione do tej pory. Dalsze fragmenty tego podrozdziału opisują, jak i dlaczego należy używać metod.

Dlaczego warto używać metod?

W typowych programach musisz przekazać do metody informację, aby uzyskać potrzebny wynik. Informację może stanowić pojedyncza liczba całkowita, zbiór ciągów znaków, jak również inne dane. Są to tak zwane *dane wejściowe*. Jednak niektóre metody nie przyjmują danych wejściowych, dlatego nie jest to konieczna cecha metody. Metody używają danych wejściowych i informacji środowiskowych (na przykład danych o aktualnym stanie programu) do wykonywania potrzebnych operacji.

Umieszczanie w metodzie potrzebnych informacji to *przekazywanie* danych. Przekazywane dane to tak zwane *parametry* metody. Użycie metody określa się jako jej *wywołanie*.

UWAGA

Oto podsumowanie — wywołujesz metody, przekazując do nich dane za pomocą parametrów.

Celem stosowania metod jest umożliwienie wielokrotnego wykorzystania kodu. Zasady używania metod mają sens, kiedy spojrzysz na to z perspektywy całego programu. Kiedy przyjrzysz się wszystkim algorytmom składającym się na program, możesz znaleźć w nich wspólne elementy. Jeśli musisz wykonać daną operację więcej niż raz, warto umieścić ją w metodzie, którą można wielokrotnie wywoływać.

Wyobraź sobie program składający się z wielu algorytmów. Niektóre z nich wymagają obliczenia powierzchni koła. Ponieważ *pewne* algorytmy muszą zawierać operacje prowadzące do obliczenia tej powierzchni, można przekształcić je w metodę. Wystarczy napisać kod obliczający powierzchnię koła na podstawie jego promienia i poddać go *hermetyzacji* („opakować") w metodzie, którą można wykorzystać w innych algorytmach. Dzięki temu nie trzeba wielokrotnie pisać kodu wykonującego te same operacje — wystarczy zrobić to raz i wykorzystać ten sam kod w innych miejscach.

Możliwe, że jeden algorytm wymaga obliczenia powierzchni koła o promieniu 100, podczas gdy inny algorytm potrzebuje powierzchni koła o promieniu 200. Tworząc metodę, która przyjmuje promień jako parametr, możesz użyć jej do obliczenia powierzchni koła o dowolnym promieniu.

UWAGA

W języku Visual Basic 2010 metody definiuje się, używając słowa kluczowego `Sub` lub słowa kluczowego `Function`. `Sub` to skrót od angielskiego *subroutine* (procedura) — słowo to służy do tworzenia metod, które nie zwracają wartości, o czym dowiedziałeś się już w rozdziale 1. Słowo kluczowe `Function` pozwala definiować metody zwracające wartość, czyli funkcje.

Metody z tego rozdziału

Dobrą wiadomością jest to, że używałeś już metod. Przyjrzyj się fragmentowi kodu z początku rozdziału:

```
Private Sub btnAdd_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnAdd.Click
   ' Definicja zmiennej intNumber.
   Dim intNumber As Integer

   ' Przypisanie początkowej wartości.
   intNumber = 27

   ' Dodanie 1 do zmiennej intNumber.
   intNumber = intNumber + 1

   ' Wyświetlenie nowej wartości zmiennej intNumber.
   MessageBox.Show("Wartość zmiennej intNumber + 1 = " & intNumber.ToString, _
                   "Zmienne")
End Sub
```

Powyższy kod to metoda. Jest to samodzielny blok kodu wykonujący pewne operacje. Ta metoda dodaje `1` do wartości zmiennej `intNumber` i wyświetla wynik w oknie komunikatu.

Ta metoda nie zwraca wartości — jest to procedura, dlatego rozpoczyna się od słowa kluczowego `Sub`, a kończy wyrażeniem `End Sub`. Wszystko między tymi wyrażeniami to kod metody. Przyjrzyj się utworzonej automatycznie przez Visual Basic 2010 definicji metody:

```
Private Sub btnAdd_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnAdd.Click
```

1. Na początku pojawia się słowo `Private`. Znaczenie tego słowa wyjaśnione jest w jednym z dalszych rozdziałów. Na razie zapamiętaj, że takiej metody nie można wywołać poza daną klasą.
2. Następnie występuje słowo kluczowe `Sub`, które informuje język Visual Basic 2010, że jest to definicja procedury.
3. Kolejny element to wyrażenie `btnAdd_Click`. Jest to nazwa procedury.
4. W nawiasie znajduje się wyrażenie `ByVal sender As System.Object, ByVal e As System.EventArgs`. To wyrażenie informuje język Visual Basic 2010, że metoda przyjmuje dwa parametry: `sender` i `e`. Stosowanie parametrów omówiono w dalszej części tego rozdziału.
5. Na końcu znajduje się wyrażenie `Handles btnAdd.Click`. To wyrażenie informuje język Visual Basic 2010, że daną metodę należy wywoływać w wyniku kliknięcia przycisku `btnAdd`.

SPRÓBUJ SAM Używanie metod

Plik z kodem projektu Three Buttons można pobrać z witryny helion.pl.

To ćwiczenie „Spróbuj sam” opisuje, jak utworzyć metodę wyświetlającą okno komunikatu, a także jak wywołać tę metodę za pomocą trzech różnych przycisków.

1. Utwórz nową aplikację typu *Windows Forms Application* i nazwij ją **`Three Buttons`**.
2. Za pomocą okna narzędzi dodaj do formularza trzy przyciski.
3. Kliknij dwukrotnie pierwszy przycisk (*Button1*), aby utworzyć nową metodę obsługi zdarzenia `Click`. Dodaj do programu wyróżniony pogrubieniem kod:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Button1.Click
   ' Wywołanie metody.
   SayHello()
End Sub

Private Sub SayHello()
   ' Wyświetlanie okna komunikatu.
   MessageBox.Show("Witaj, świecie!", "Trzy przyciski")
End Sub
```

4. Zapisz projekt przez kliknięcie przycisku *Save All* na pasku narzędzi.
5. Uruchom projekt. Pojawi się formularz z trzema przyciskami. Kliknij przycisk znajdujący się najwyżej, a wtedy program wyświetli okno komunikatu z napisem `Witaj, świecie!`

Jak to działa?

Jak już wiesz, dwukrotne kliknięcie przycisku w oknie projektowym powoduje automatyczne utworzenie nowej metody:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Button1.Click

End Sub
```

Wyrażenie `Handles Button1.Click` na końcu definicji metody informuje język Visual Basic 2010, że metoda powinna być automatycznie wywoływana w wyniku wyzwolenia zdarzenia `Click` przycisku. Elementem definicji są dwa udostępniane przez język Visual Basic 2010 parametry, którymi na razie nie musisz się przejmować. Poza tą metodą znajduje się definicja nowej metody:

```
Private Sub SayHello()
   ' Wyświetlanie okna komunikatu.
   MessageBox.Show("Witaj, świecie!", "Trzy przyciski")
End Sub
```

Ta nowa metoda nosi nazwę `SayHello`. Kod znajdujący się pomiędzy pierwszym a ostatnim wierszem to kod metody wykonywany w wyniku jej wywołania. W tym przypadku kod metody wyświetla okno komunikatu.

W momencie kliknięcia przycisku program wywołuje metodę `Button1_Click`. Z kolei metoda `Button1_Click` wywołuje metodę `SayHello`. W sumie w wyniku kliknięcia przycisku program wyświetla okno komunikatu.

```
Private Sub Button1_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Button1.Click

   ' Wywołanie metody.
   SayHello()
End Sub
```

Powinno to wyjaśnić ogólne działanie metod, ale czy naprawdę warto rozbijać kod na odrębne metody w celu wyświetlenia okna komunikatu? Następne ćwiczenie „Spróbuj sam” stanowi odpowiedź na to pytanie.

SPRÓBUJ SAM Wielokrotne wykorzystanie metod

W tym ćwiczeniu zobaczysz, jak powtórnie wykorzystać metodę przez wywołanie jej w innych obszarach kodu.

1. Jeśli program wciąż działa, zamknij go.
2. Wróć do okna projektowego i dwukrotnie kliknij drugi przycisk. Dodaj do nowej metody wyróżniony pogrubieniem kod:

   ```
   Private Sub Button2_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles Button2.Click

      ' Wywołanie metody.
      SayHello()
   End Sub
   ```

3. Ponownie otwórz okno projektowe i kliknij dwukrotnie trzeci przycisk. Dodaj do niego wyróżniony pogrubieniem kod:

```
Private Sub Button3_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Button3.Click
   ' Wywołanie metody.
   SayHello()
End Sub
```

4. Uruchom projekt. Zobaczysz, że kliknięcie każdego przycisku powoduje wyświetlenie tego samego okna komunikatu.
5. Zatrzymaj program i przejdź do definicji metody `SayHello`. W kodzie tej metody zmień tekst wyświetlany w oknie, na przykład:

```
Private Sub SayHello()
   ' Wyświetlanie okna komunikatu.
   MessageBox.Show("Zmieniłem się!", "Trzy przyciski")
End Sub
```

6. Ponownie uruchom projekt. Zauważysz, że teraz kliknięcie każdego przycisku powoduje wyświetlenie okna z nowym tekstem.

Jak to działa?

Metody obsługi zdarzenia `Click` wszystkich przycisków wywołują tę samą metodę `SayHello()`:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Button1.Click
   ' Wywołanie metody.
   SayHello()
End Sub

Private Sub Button2_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Button2.Click
   ' Wywołanie metody.
   SayHello()
End Sub

Private Sub Button3_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Button3.Click
   ' Wywołanie metody.
   SayHello()
End Sub
```

Zauważ, że słowo kluczowe `Handles` wiąże każdą z tych metod z innym przyciskiem — `Button1`, `Button2` i `Button3`.

Bardzo istotne (i sprytne!) jest to, że efekty zmiany wprowadzonej w metodzie `SayHello` są takie same dla każdego z trzech przycisków. Jest to bardzo ważna technika programistyczna. Dzięki niej możesz zebrać kod w jednym miejscu, a wprowadzone w nim zmiany są odzwierciedlane w całej aplikacji. Pozwala to zaoszczędzić czas potrzebny na wielokrotne przepisywanie tego samego lub bardzo podobnego kodu.

Tworzenie metod

W poprzednim ćwiczeniu „Spróbuj sam” zbudowałeś metodę wyświetlającą statyczny tekst. Metody są najprzydatniejsze, jeśli przyjmują dane i wykonują na nich użyteczne operacje. Czasem metoda powinna zwracać wartość, co ilustruje następne ćwiczenie „Spróbuj sam”.

SPRÓBUJ SAM Tworzenie metody

W tym ćwiczeniu „Spróbuj sam” opisano tworzenie metody zwracającej wartość, a dokładniej — tworzenie metody zwracającej powierzchnię koła obliczoną na podstawie jego promienia. Służy do tego poniższy algorytm:

- Podnieść promień do kwadratu.
- Pomnożyć wynik przez liczbę *pi*.

W tym ćwiczeniu możesz wykorzystać utworzony wcześniej projekt *Three Buttons*.

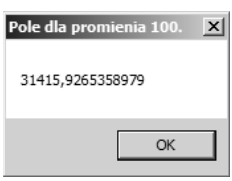
1. Dodaj poniższy kod, który definiuje nową metodę. Jest to funkcja, ponieważ zwraca wartość:

```
' Metoda CalculateAreaFromRadius — oblicza powierzchnię koła.
Private Function CalculateAreaFromRadius(ByVal radius As Double) As Double
   ' Deklaracja zmiennych.
   Dim dblRadiusSquared As Double
   Dim dblResult As Double

   ' Podnoszenie promienia do kwadratu.
   dblRadiusSquared = radius * radius

   ' Mnożenie wyniku przez liczbę pi.
   dblResult = dblRadiusSquared * Math.PI

   ' Zwracanie wyniku.
   Return dblResult
End Function
```

2. Teraz usuń starą wersję metody `Button1_Click` i wstaw zamiast niej wyróżniony pogrubieniem kod:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Button1.Click
   ' Deklaracja zmiennej.
   Dim dblArea As Double

   ' Obliczanie powierzchni koła o promieniu 100.
   dblArea = CalculateAreaFromRadius(100)

   ' Wyświetlanie wyniku.
   MessageBox.Show(dblArea.ToString, "Pole dla promienia 100.")
End Sub
```

3. Uruchom projekt i kliknij przycisk `Button1`. Pojawi się okno widoczne na rysunku 3.16.



Rysunek 3.16. Powierzchnia koła o promieniu 100

Jak to działa?

Ten program zawiera odrębną metodę o nazwie `CalculateAreaFromRadius`. Do deklarowania tej metody służy blok `Private Function … End Function`.

```
Private Function CalculateAreaFromRadius(ByVal radius As Double) As Double
…
End Function
```

Kod znajdujący się między wyrażeniami `Private Function` a `End Function` to *ciało* metody wykonywane jedynie w wyniku jej wywołania.

Wyrażenie `ByVal radius As Double` definiuje parametr tej metody. `ByVal` oznacza, że parametr jest przekazywany przez wartość. W przypadku parametrów przekazywanych przez wartość platforma .NET tworzy nową zmienną i zapisuje w niej informacje przekazane za pomocą parametru. Oryginalna wartość parametrów przekazywanych w ten sposób nie zmienia się. W tym przypadku definicja informuje platformę .NET, że do metody przekazywany jest parametr o nazwie `radius`. Powoduje to utworzenie nowej zmiennej o nazwie `radius` — takiej samej jak zmienna utworzona w wyniku wykonania poniższej instrukcji:

```
Dim radius As Double
```

Jest jednak pewna istotna różnica. Wartość zmiennej przekazywanej jako parametr jest automatycznie ustawiana na podaną wartość, dlatego jeśli jako wartość parametru przekażesz `200`, jest to równoważne następującej instrukcji:

```
Dim radius As Double = 200
```

Jeśli przekażesz jako parametr liczbę `999`, jest to równoznaczne z instrukcją:

```
Dim radius As Double = 999
```

UWAGA

Parametry można przekazywać także przez referencję, do czego służy słowo kluczowe `ByRef`. Kiedy parametr jest przekazywany przez referencję, jego nazwa używana w ciele metody staje się po prostu inną nazwą oryginalnej zmiennej, dlatego wszelkie zmiany wprowadzone w tej zmiennej w ciele metody są odzwierciedlane w oryginale.

Wyrażenie `As Double` na końcu deklaracji metody informuje język Visual Basic 2010, że metoda zwraca liczbę zmiennoprzecinkową o podwójnej precyzji:

```
Private Function CalculateAreaFromRadius(ByVal radius As Double) As Double
```

Teraz możesz dokładniej przyjrzeć się kodowi metody. Wiesz już, że do obliczenia powierzchni koła służy następujący algorytm:

1. Pobranie liczby reprezentującej promień koła.
2. Podniesienie tej liczby do kwadratu.
3. Pomnożenie tej liczby przez *pi* (π).

Dokładnie tak działa metoda `CalculateAreaFromRadius`:

```
' Deklaracja zmiennych.
Dim dblRadiusSquared As Double
Dim dblResult As Double
```

```
' Podnoszenie promienia do kwadratu.
dblRadiusSquared = radius * radius

' Mnożenie wyniku przez liczbę pi.
dblResult = dblRadiusSquared * Math.PI
```

`Math.PI` w powyższym kodzie to stała platformy .NET, która zawiera wartość liczby *pi*. W ostatnim wierszu metody należy zwrócić wynik do kodu wywołującego tę metodę. Służy do tego następująca instrukcja:

```
' Zwracanie wyniku.
Return dblResult
```

Nowy kod metody `Button1_Click` wywołuje metodę `CalculateAreaFromRadius` i wyświetla zwrócony przez nią wynik:

```
' Deklaracja zmiennej.
Dim dblArea As Double

' Obliczanie powierzchni koła o promieniu 100.
dblArea = CalculateAreaFromRadius(100)

' Wyświetlanie wyniku.
MessageBox.Show(dblArea.ToString, "Pole dla promienia 100")
```

Na początku znajduje się definicja zmiennej o nazwie `dblArea`, przeznaczonej do przechowywania powierzchni koła. Następnie metoda przypisuje do tej zmiennej wartość zwracaną przez metodę `CalculateAreaFromRadius`. Nawiasy po nazwie metody służą do przekazywania parametrów. W tym przypadku jest to jeden parametr — liczba `100`.

Po wywołaniu metody `CalculateAreaFromRadius` należy czekać na zwrócenie przez nią wyniku (wykonanie wiersza z instrukcją `Return`), a następnie zapisać go w zmiennej `dblArea`. Wtedy można wyświetlić wynik na ekranie w zwykły sposób.

Nazwy metod

Jest kilka standardów nazewnictwa zalecanych przy pisaniu programów na platformę .NET. Te standardy pomagają programistom łatwo przechodzić między językami, co opisuje rozdział 2. Do tworzenia nazw metod zaleca się używać *notacji paskalowej*. W tej notacji pierwsze litery wszystkich słów są duże, a pozostałe — małe. Jest to tylko sugestia związana z najlepszymi praktykami kodowania, a nie wymóg języka Visual Basic 2010. Przykłady takich nazw to:

- `CalculateAreaFromRadius`
- `OpenXmlFile`
- `GetEnvironmentValue`

Zwróć uwagę, że w tej notacji nawet skróty (w tym przypadku XML) *nie* są pisane dużymi literami. Pozwala to uniknąć dwuznaczności w sytuacjach, kiedy nie wiadomo, czy dane słowo jest normalnie pisane dużymi literami.

Z kolei notacją zalecaną do nazw parametrów jest tak zwana *notacja wielbłądzia*. Jeśli miałeś kiedyś styczność z kodem pisanym w języku Java, prawdopodobnie znasz już tę notację. Notacja wielbłądzia jest podobna do notacji paskalowej, ale pierwsze słowo nazwy jest pisane małą literą, na przykład:

- `myAccount`
- `customerDetails`
- `updateDnsRecord`

Także w tym przypadku skróty (DNS) traktuje się jak zwykłe słowa, dlatego — podobnie jak w notacji paskalowej — składają się z dużych i małych liter.

UWAGA

Notacja wielbłądzia zawdzięcza nazwę temu, że w środku nazw pojawiają się garby, na przykład `takiGarb`. Nazwa notacji paskalowej pochodzi od języka programowania Pascal, na którego potrzeby opracowano tę notację.

W rozdziale 2. dowiedziałeś się, że platforma .NET nie jest związana z jednym konkretnym językiem. Ponieważ niektóre języki są *wrażliwe na wielkość znaków*, a inne nie, ważne jest, aby używać standardów ułatwiających pracę innym programistom, którzy mogą mieć doświadczenie w pisaniu programów w innych językach.

Wrażliwość na wielkość znaków oznacza, że ważne jest to, czy nazwa zawiera duże, czy małe litery. W językach wrażliwych na wielkość znaków nazwy `MYACCOUNT` i `myAccount` to coś zupełnie innego. Język Visual Basic 2010 *nie jest* wrażliwy na wielkość znaków, dlatego można używać słów z literami dowolnej wielkości, a nazwy `MYACCOUNT` i `mYacCounT` oznaczają to samo.

UWAGA

Języki wrażliwe na wielkość znaków to między innymi Java, C# i C++.

Zasięg

W opisie pojęcia „metoda” pojawiło się stwierdzenie *samodzielny* blok kodu. Ta cecha ma duże znaczenie ze względu na sposób używania i definiowania zmiennych w metodach. Wyobraź sobie, że używasz dwóch metod, a w każdej z nich znajduje się zmienna o nazwie `strName`:

```
Private Sub DisplaySebastiansName()
   ' Deklaracja zmiennej i przypisanie do niej wartości.
   Dim strName As String
   strName = "Sebastian Blackwood"

   ' Wyświetlanie wartości zmiennej.
   MessageBox.Show(strName, "Zasięg")
End Sub

Private Sub DisplayBalthazarsName()
   ' Deklaracja zmiennej i przypisanie do niej wartości.
   Dim strName As String
   strName = "Balthazar Keech"

   ' Wyświetlanie wartości zmiennej.
   MessageBox.Show(strName, "Zasięg")
End Sub
```

Choć obie metody używają zmiennej o takiej samej nazwie (`strName`), samodzielność metod sprawia, że jest to dozwolone, a nazwy zmiennych w różnych metodach są niezależne od siebie. Możesz sam się o tym przekonać.

SPRÓBUJ SAM Zasięg

Plik z kodem projektu Scope Demo można pobrać z witryny helion.pl.

W tym ćwiczeniu zaczniesz poznawanie zasięgu zmiennych za pomocą używanej w dwóch różnych metodach zmiennej o tej samej nazwie.

1. Utwórz nowy projekt typu *Windows Forms Application* i nazwij go **`Scope Demo`**.
2. Dodaj przycisk do formularza, nazwij go **`btnScope`** i ustaw jego właściwość `Text` na **`Zasięg`**. Kliknij dwukrotnie ten przycisk i dodaj wyróżniony pogrubieniem kod do metody obsługi zdarzenia `Click`. Następnie dodaj dwie pozostałe przedstawione metody.

```
Private Sub btnScope_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnScope.Click
   ' Wywołanie metody.
   DisplayBalthazarsName()
End Sub

Private Sub DisplaySebastiansName()
   ' Deklaracja zmiennej i przypisanie do niej wartości.
   Dim strName As String
   strName = "Sebastian Blackwood"

   ' Wyświetlanie wartości zmiennej.
   MessageBox.Show(strName, "Zasięg")
End Sub

Private Sub DisplayBalthazarsName()
   ' Deklaracja zmiennej i przypisanie do niej wartości.
   Dim strName As String
   strName = "Balthazar Keech"

   ' Wyświetlanie wartości zmiennej.
   MessageBox.Show(strName, "Zasięg")
End Sub
```

3. Zapisz projekt przez kliknięcie przycisku *Save All* na pasku narzędzi.
4. Uruchom projekt. Po kliknięciu przycisku zobaczysz okno komunikatu wyświetlające imię i nazwisko `Balthazar Keech`.

Jak to działa?

To ćwiczenie pokazuje, że choć używasz tej samej nazwy zmiennej w dwóch różnych miejscach, program działa poprawnie:

```
Private Sub DisplaySebastiansName()
   ' Deklaracja zmiennej i przypisanie do niej wartości.
   Dim strName As String
   strName = "Sebastian Blackwood"

   ' Wyświetlanie wartości zmiennej.
```

```
    MessageBox.Show(strName, "Zasięg")
End Sub

Private Sub DisplayBalthazarsName()
    ' Deklaracja zmiennej i przypisanie do niej wartości.
    Dim strName As String
    strName = "Balthazar Keech"

    ' Wyświetlanie wartości zmiennej.
    MessageBox.Show(strName, "Zasięg")
End Sub
```

Po rozpoczęciu wykonywania metody zdefiniowane w niej (w obrębie wyrażeń `Sub` i `End Sub` lub `Function` i `End Function`) zmienne mają *zasięg lokalny*. *Zasięg* określa, w jakich miejscach programu dana zmienna jest widoczna. Zmienne o zasięgu *lokalnym* są widoczne *w obrębie metody*, w której zostały zdefiniowane.

Zmienna `strName` formalnie nie istnieje do momentu rozpoczęcia wykonywania metody. Na tym etapie platforma .NET i system Windows przydzielają pamięć dla tej zmiennej, dzięki czemu można użyć jej w kodzie. Na początku metoda przypisuje wartość do tej zmiennej, a następnie wyświetla okno komunikatu. Zmienna zdefiniowana w metodzie `DisplayBalthazarsName` powstaje w momencie wywołania tej metody, następnie uruchamiany jest kod zmieniający wartość tej zmiennej, a po zakończeniu działania metody środowisko usuwa zmienną.

UWAGA

W rozdziale 4. dowiesz się, że zasięg można ograniczyć jeszcze bardziej — do pętli działających w procedurach i funkcjach.

Podsumowanie

Ten rozdział opisuje podstawy pisania programów nie tylko w języku Visual Basic 2010, ale we wszystkich językach programowania. Na początku przedstawione jest pojęcie algorytmu, który stanowi podstawę każdego programu komputerowego. Następnie opisane są zmienne oraz najczęściej używane typy danych — `Integer`, `Double`, `String`, `Date` i `Boolean`. Zobaczyłeś, jak używać tych typów danych do wykonywania operacji matematycznych, łączenia ciągów znaków, zwracania długości ciągów znaków, dzielenia tekstu na podciągi, pobierania aktualnej daty i pobierania właściwości dat. Dowiedziałeś się także, jak zmienne są przechowywane w komputerze.

Następnie opisane zostały metody — czym są, do czego ich potrzebujesz i jak je tworzyć. Dowiedziałeś się także, że zmienne zadeklarowane w obrębie danej metody mają zasięg lokalny i nie są widoczne poza tą metodą. Ponadto poznałeś różnice między funkcjami a procedurami.

ĆWICZENIA

1. Utwórz aplikację typu *Windows Forms Application* i dodaj do niej dwa przyciski. W metodzie obsługi zdarzenia Click pierwszego przycisku zadeklaruj dwie zmienne typu Integer i przypisz do nich dowolne liczby. Wykonaj na tych zmiennych dowolną operację arytmetyczną i wyświetl wynik w oknie komunikatu.

W metodzie obsługi zdarzenia Click drugiego przycisku zadeklaruj dwie zmienne typu String i przypisz do nich dowolny tekst. Połącz obie zmienne i wyświetl wynik w oknie komunikatu.

2. Utwórz aplikację typu *Windows Forms Application* i dodaj do niej pole tekstowe oraz przycisk. W metodzie obsługi zdarzenia Click przycisku wyświetl trzy okna komunikatu. Pierwsze okno powinno wyświetlać długość ciągu znaków wpisanego w polu tekstowym, drugie — pierwszą część ciągu znaków, a trzecie — jego drugą część.

CZEGO NAUCZYŁEŚ SIĘ W TYM ROZDZIALE?

Obszar	Informacje
Algorytmy	Czym są algorytmy i jak stosować je do tworzenia programów.
Zmienne	Deklarowanie najczęściej stosowanych typów danych i korzystanie z nich.
Funkcje ciągów znaków	Stosowanie popularnych funkcji do zarządzania ciągami znaków przy korzystaniu z typu danych `String`.
Typ danych Date	Używanie typu danych `Date` oraz automatyczne wyświetlanie daty i czasu według ustawień regionalnych komputera.
Metody	Tworzenie i stosowanie prostych metod, które przyjmują parametry oraz zwracają wartość lub nie mają tych cech.