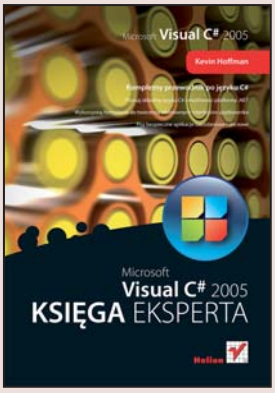
Microsoft Visual C# 2005. Księga eksperta



Autor: Kevin Hoffman
Tłumaczenie: Marcin Winiarski (wstęp, rozdz. 1-23, 30-35, 38-41), Tomasz Grochowski (rozdz. 24-27, 29, 36, 37), Dariusz Małyszko (rozdz. 28)
ISBN: 978-83-246-0664-1
Tytuł oryginału: Microsoft Visual C# 2005 Unleashed
Format: B5, stron: około 650

Kompletny przewodnik po języku C#

- Poznaj składnię języka C# i możliwości platformy .NET
- Wykorzystaj formularze do tworzenia efektownych interfejsów użytkownika
- Pisz bezpieczne aplikacje bazodanowe i sieciowe

Na platformie .NET można programować w wielu językach, jednak najważniejszym z nich jest C#, który został zaprojektowany specjalnie w tym celu; to język ułatwiający wygodne korzystanie z niesamowitych możliwości .NET Framework. Za pomocą C# możesz bez żmudnej nauki skomplikowanej składni błyskawicznie tworzyć efektowne programy dla systemu Windows, witryny internetowe, usługi sieciowe czy aplikacje bazodanowe. Szerokie zastosowania i duże możliwości połączone z łatwością nauki sprawiają, że język C# to doskonały wybór zarówno dla początkujących, jak i doświadczonych programistów.

Książka „Microsoft Visual C# 2005. Księga eksperta" to wyczerpujący przegląd najnowszej i najbardziej użytecznej wersji języka C#. Czytając ją, poznasz podstawowe elementy i składnię języka, a także używane w nim struktury danych i techniki programowania obiektowego. Dowiesz się, czym jest platforma .NET, jak działa, jakie ma możliwości oraz – co najważniejsze – jak wykorzystać ją do pisania własnych programów. Nauczysz się obsługiwać bazy danych przy użyciu technologii ADO.NET, tworzyć różnorodne aplikacje sieciowe, a także stosować formularze Windows do projektowania programów dla tego systemu. Zrozumiesz techniki tworzenia aplikacji rozproszonych za pomocą technologii remoting.

- Składnia języka C#
- Działanie platformy .NET
- Używanie formularzy do tworzenia efektownych interfejsów
- Praca z bazami danych w ADO.NET 2.0
- Tworzenie aplikacji sieciowych w ASP.NET 2.0
- Pisanie i konsumowanie usług sieciowych
- Zabezpieczanie programów i danych
- Aplikacje rozproszone i technologia remoting
- Instalowanie aplikacji za pomocą technologii ClickOnce
- Programowanie wielowątkowe

Jeśli chcesz nadążyć za najnowszą technologią – ta książka jest dla Ciebie

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 2.
Wyrażenia i struktury sterujące

U podstaw każdego języka programowania leży zdolność do spełniania dwóch różnych typów zadań: powtarzalności i logiki warunkowej. Powtarzalność pociąga za sobą zastosowanie konstrukcji językowych zwanych *pętlami* (ang. *loops*) w celu wielokrotnego wykonania tego samego zadania. Z kolei logika warunkowa wymaga sposobu zapisywania kodu, który będzie wykonany tylko wtedy, gdy zostaną spełnione określone warunki. Rozdział ten wprowadza podstawy umożliwiające nam realizację obu tych typów zadań z użyciem języka C#.

Jeśli posiadamy pewną znajomość języków programowania, pojęcia omawiane w tym rozdziale powinny okazać się znajome i z łatwością podchwycimy składnię C# w tym zakresie. Jeśli korzystamy z C# już od jakiegoś czasu, możemy jedynie przejrzeć pokrótce ten rozdział w ramach przypomnienia, zanim przejdziemy do dalszej części książki.

Rozgałęzianie i logika warunkowa

Każdy nowoczesny język programowania ma wsparcie dla rozgałęziania i logiki warunkowej. Umożliwia to naszej aplikacji wykonywanie różnych rzeczy w oparciu o bieżący strumień wejściowy, zdarzenia, warunki występowania błędów lub dowolne inne warunki, jakie można logicznie wyrazić. W tym podrozdziale najpierw ujrzymy, jak C# umożliwia formułowanie wyrażeń logicznych, oraz jak tworzyć bloki kodu, które są wykonywane warunkowo w oparciu o te wyrażenia, a następnie zobaczymy kilka skrótów i dodatków, jakie zawiera C#, jeszcze bardziej ułatwiających stosowanie pewnych popularnych typów instrukcji warunkowych.

Wprowadzenie do wyrażeń logicznych

Wyrażenie logiczne to instrukcja kodu, która ostatecznie przyjmie wartość `true` lub `false` (prawda lub fałsz). Na najbardziej podstawowym poziomie wszystkie wyrażenia logiczne (czy też warunkowe), bez względu na to, jak długie i złożone, przyjmą wartość `true` lub `false`.

Najprostszym ze wszystkich wyrażeń logicznych jest *równość*. Wyrażenie to jest stosowane do sprawdzenia, czy jedna wartość jest równoważna innej, czy też nie. Może to być coś nawet tak prostego, jak:

```
2 == 4
```

Wyrażenie to przyjmie wartość `false` (fałsz), gdyż 2 jest różne od 4. Może to być również coś tak złożonego, jak:

```
MyObject.MyProperty == YourObject.YourProperty
```

To wyrażenie może przyjąć dowolną wartość i zostanie to określone w czasie wykonywania programu. Jeśli jesteśmy zaznajomieni z C, C++ lub nawet C#, to wiemy, że == (podwójny symbol równości) jest logicznym operatorem boolowskim, natomiast = (pojedynczy znak równości) jest operatorem przypisania używanym do ustawiania wartości. Często błędy występujące w czasie kompilacji i wykonywania mają związek ze stosowaniem tych operatorów w nieodpowiednich miejscach.

Tabela 2.1 przedstawia operatory logiczne dostępne w języku C# oraz krótki opis ich przeznaczenia.

Tabela 2.1. *Operatory logiczne w C#*

Operator	Opis
&&	Operator iloczynu logicznego (ang. *Boolean AND*). Wyrażenie, w skład którego wchodzi, przyjmie wartość `true` (prawdę), jeśli *oba* podwyrażenia (po prawej i lewej stronie operatora) będą prawdziwe (spełnione). W przeciwnym razie zawsze będzie to `false` (fałsz).
||	Operator sumy logicznej (ang. *Boolean OR*). Wyrażenie, w skład którego wchodzi, przyjmie wartość `true`, jeśli wyrażenie po lewej lub prawej stronie jest prawdziwe oraz wówczas gdy oba wyrażenia są prawdziwe.
!	Operator negacji (ang. *NOT*). Neguje wartość logiczną dowolnego wyrażenia, przy którym został zastosowany. Zatem jeśli `!` został użyty przy prawdziwym wyrażeniu, stanie się ono fałszywe. Analogicznie, jeśli `!` został użyty przy fałszywym wyrażeniu, stanie się ono prawdziwe.
^	Operator alternatywy wykluczającej (ang. *exclusive OR*, w wielu językach zapisywany jako *XOR*). Funkcjonuje on w taki sam sposób jak operator standardowej sumy logicznej (`||`), za wyjątkiem tego, że jeżeli *obie* strony wyrażenia są prawdziwe, operator ten zwróci fałsz.
~	Operator ten wykonuje dopełnienie bitowe na swoim operandzie (argumencie operacji). Oznacza to, że operator ten zamieni na odwrotny każdy bit tego operandu. Na wartości jednobitowej działa identycznie jak operator `!`, jednak na liczby takie, jak całkowite, dziesiętne itd. ma silniejszy wpływ.
&	Operator iloczynu bitowego (ang. *AND*). Zamiast iloczynu wykonywanego na dwóch wyrażeniach logicznych, operator ten działa na dwóch wartościach numerycznych. Liczby te są szeregowane, a mnożenie wykonuje się na każdej parze bitów sobie odpowiadających. Wynikiem tego działania jest zupełnie nowa liczba. Logika bitowa często ma zastosowanie, gdy przechowujemy długie listy wartości boolowskich jako liczby.
|	Operator sumy bitowej (ang. *OR*). Podobnie jak to było z iloczynem bitowym, wykonuje on operację sumowania na każdej parze odpowiadających sobie bitów obu argumentów danego wyrażenia. Wynik jest zwrócony jako wartość liczbowa.

Zastosowanie podstawowych instrukcji warunkowych

W poprzedniej części tego podrozdziału przedstawione zostały narzędzia potrzebne do budowania wyrażeń logicznych. Pozwolą nam one stwierdzić, czy dane wyrażenie jest prawdziwe, czy też nie. Gdy zachodzi taka potrzeba, musimy jeszcze zastosować pewne instrukcje warunkowe w naszym kodzie, aby móc wykonać jakieś znaczące operacje w oparciu o wynik interesujących nas wyrażeń logicznych. Ta część bieżącego rozdziału przedstawi podstawowe instrukcje warunkowe będące nieodzownym elementem każdego sensownego kodu napisanego w C#.

Zastosowanie instrukcji if/else

Struktura instrukcji `if` jest następująca:

```
if ( wyrażenie )
   blok_kodu
else if ( wyrażenie_1 )
   blok_kodu
else if ( wyrażenie_2 )
   blok_kodu
else
   blok_kodu
```

Obie sekcje `else if`, a także `else`, są opcjonalne i są wymagane jedynie, gdy chcemy w naszym kodzie wykonać pewne alternatywne zadania, kiedy główne wyrażenie logiczne (w naszym przypadku wyrażenie warunkowe) przyjmie wartość `false`.

Ponieważ każdy blok kodu w powyższym przykładzie może również zawierać swoje własne instrukcje `if`, możemy zagnieżdżać nasz kod warunkowy niemal tak głęboko, jak tylko chcemy. Jednakże w dobrym stylu, jak i w zgodzie z zaleceniem etykiety programisty jest unikanie głębokiego zagnieżdżania, gdyż czyni to nasz kod niezwykle trudnym do odczytu, a tym bardziej analizy.

Poniższy przykład instrukcji `if` obrazuje zastosowanie jej w sposób prosty, zagnieżdżony oraz z użyciem instrukcji `else`:

```
if (strInput == "Witam")
   Console.WriteLine("Powiedziałeś 'Witam'");

if (strInput2 == "Żegnam")
   Console.WriteLine("Powiedziałeś 'Żegnam'");
else if (strInput2 == "Tymczasem")
   Console.WriteLine("Nie powiedziałeś 'Żegnam', lecz 'Tymczasem'.");
else
{
   if (strInput3 == "Hola")
      if (strInput4 == "Senor")
         Console.WriteLine("Witaj!");
}
```

Zastosowanie instrukcji switch

Jeśli chcemy zbadać pojedynczą zmienną pod kątem całej listy możliwych wartości, korzystając jedynie ze standardowych wyrażeń kluczowych `if/else`, posłużymy się kodem zbliżonym do następującego:

```
if (val == 1)
   ...
else if (val == 2)
   ...
else if (val == 3)
   ...
else if (val == 4)
   ...
else
   ...
```

Choć spełni to swoje zadanie, nie będzie to jednak blok kodu ani elegancki, ani łatwo czytelny. W takiej sytuacji należałoby zastosować instrukcję C# o nazwie `switch`, która umożliwia przeprowadzenie kilku testów logicznych dla pojedynczego wyrażenia, co zostało pokazane za pomocą następującego przykładu:

```
switch (val)
{
   case 1:
      ...
      break;
   case 2:
      ...
      break;
   case 3:
      ...
      break;
   default:
      ...
}
```

Zastosowanie instrukcji goto

Instrukcja `goto` posiada dwa główne zastosowania. Pierwszym z nich jest przekazywanie kontroli z jednego bloku `case` do innego, w ramach instrukcji `switch`, natomiast drugim zastosowaniem jest przerywanie wykonywania pętli. O pętlach dowiemy się więcej w dalszej części tego rozdziału.

Poniżej znajduje się przykład instrukcji `switch`, w którym wykorzystano wyrażenie kluczowe `goto` do przekazania kontroli:

```
switch (val)
{
   case 1:
      ...
      break;
```

```
   case 2:
      ...
      goto case 1;
      break;
   case 3:
      ...
      goto case 1;
      break;
   default:
      ...
}
```

Zastosowanie zaawansowanych instrukcji warunkowych

Jednym z najczęstszych zastosowań instrukcji `if/else` jest warunkowe wypisanie jakichś informacji lub wyświetlenie czegoś w oknie Windows albo na formularzu Web Forms. Przykładowo załóżmy, że chcemy wyświetlić słowo „odpowiednia", jeśli marża wynosi więcej niż 20, lub „nieodpowiednia", jeśli wynosi co najwyżej 20. Można by było w tym celu użyć `if/else` w następujący sposób:

```
if (profitMargin > 20)
   Console.WriteLine("Marża jest odpowiednia!");
else
   Console.WriteLine("Marża jest nieodpowiednia!");
```

Jest to jak najbardziej poprawne, jednak C# zawiera trójskładnikowy operator zwany *operatorem warunkowym*, umożliwiający zawarcie istoty kombinacji `if/else` w pojedynczej linii. Możemy zapisać powyższy blok kodu w następującej postaci:

```
Console.WriteLine("Marża jest "+
   (profitMargin > 20 ? "odpowiednia!" : "nieodpowiednia!"));
```

Nasz trójskładnikowy operator warunkowy ma następujący format:

```
wyrażenie ? zwróć_gdy_prawdziwe : zwróć_gdy_fałszywe
```

Zapętlanie i powtarzalność

Zapętlanie jest, obok logiki warunkowej i rozgałęziania, jednym z najczęściej występujących zadań wykonywanych przez wszystkie aplikacje, bez względu na ich typ. Obecnie nie znajdziemy żadnej produktywnej aplikacji napisanej w .NET, która by się nie składała z choćby kilku pętli.

Pętla to powtórzenie wykonania danego bloku kodu pewną ilość razy. Ilość ta jest określona przez typ użytej pętli. Podrozdział ten zawiera podstawowe informacje dotyczące tworzenia i stosowania pętli w C#.

Zastosowanie pętli for

Każda pętla `for` składa się z trzech części:

1. Część inicjalizująca. Jest to blok kodu wykonywany raz, na początku pętli `for`. Zmienne zadeklarowane w tej części są dostępne w obrębie pętli w czasie jej wykonywania, ale nie będą dostępne po wyjściu z niej. Część ta jest zwykle używana do inicjalizacji zmiennej lub zmiennych zliczających.
2. Warunek zakończenia. Część ta definiuje wyrażenie logiczne używane do określenia tego, jak długo pętla `for` ma być wykonywana. Pętla ta będzie się wykonywała tak długo, jak długo owe wyrażenie będzie prawdziwe. Będzie ono sprawdzane na początku każdego cyklu pętli.
3. Kod iteracyjny. Jest to część opcjonalna i może zawierać dowolny kod, jaki chcemy wykonać na końcu każdego powtórzenia pętli. Zwykle jest to miejsce, gdzie zwiększana jest zmienna zliczająca pętle (licznik).

Format pętli `for` jest następujący:

```
for ( część inicjalizująca; warunek zakończenia; kod iteracyjny )
{
   ...
}
```

Zatem aby wywołać pętlę `for`, która wykona blok kodu pięć razy, można napisać:

```
for ( int x = 0; x < 5; x ++ )
   Console.WriteLine("Wartość licznika to " + x.ToString() );
```

Spowoduje to wyświetlenie następującego wyniku:

```
Wartość licznika to 0
Wartość licznika to 1
Wartość licznika to 2
Wartość licznika to 3
Wartość licznika to 4
```

Zastosowanie pętli while

Podczas gdy pętla `for` zwykle używana jest po to, by zapewnić indeksowanie cykli (zmienna zliczająca), pętla `while` wykonuje w kółko te same operacje, dopóki podane wyrażenie logiczne będzie prawdziwe. Format tej pętli jest następujący:

```
while ( wyrażenie ) { ... }
```

Jeśli chcemy wykonywać pewne czynności, dopóki pewna flaga (znacznik stanu) nie wskaże, że nie możemy już więcej ich wykonywać, wówczas nasza pętla `while` mogłaby wyglądać podobnie do:

```
bool canContinue = true;
while (canContinue)
{
```

```
    ... wykonuj czynności ...
    canContinue = ...
}
```

Wyrażenie jest badane na początku każdego cyklu pętli i jeśli przyjmuje wartość `false`, wówczas kod w obrębie pętli nie zostanie wykonany. Często spotykanym błędem podczas używania pętli `while` jest zaniedbanie ustawienia warunku wyjścia z pętli. W naszym przykładzie jest nim zmienna `canContinue`. Jeśli nie ustawimy warunku wyjścia, pętla `while` będzie wykonywana w nieskończoność (lub dopóki nie wyłączymy naszej aplikacji „na siłę").

Zastosowanie pętli do

Pętla `do` jest bardzo podobna do pętli `while`, poza faktem, że wyrażenie warunkowe jest sprawdzane na końcu każdego cyklu pętli zamiast na jego początku. Format pętli `do` jest następujący:

```
do
{
   ...
} while ( wyrażenie );
```

Użycie pętli `do` zawsze gwarantuje, że kod w obrębie pętli zostanie wykonany przynajmniej raz. Wszystkie inne typy pętli mają sposobność niewykonania się ani razu wówczas, gdy warunek wejściowy nie zostanie spełniony. Oto przykład pętli `do`:

```
int x = 1;
do
{
   Console.WriteLine(x);
   x++;
} while ( x < 10 );
```

Podsumowanie

Podczas pracy nad bazami danych, grafiką 3D, aplikacjami konsolowymi, aplikacjami opartymi na oknach Windows czy aplikacjami sieciowymi opartymi na Web Forms zawsze występuje kilka typów zadań, które muszą zostać wykonane, a należą do nich: zapętlanie oraz rozgałęzianie.

Zapętlanie i rozgałęzianie są głównymi mechanizmami zdefiniowanymi w każdym języku programowania. Bieżący rozdział pokazał nam, jak użyć języka C# do tworzenia złożonych rozgałęzień z użyciem wyrażeń warunkowych, oraz wiele sposobów na wykonanie zadań programowania iteracyjnego z użyciem pętli.

Rozdział ten zawiera również kilka informacji, które same w sobie mogą się wydawać mało użyteczne. Gdyby się jednak przyjrzeć możliwie każdemu problemowi programistycznemu, z jakim się zetkniemy w przyszłości, będziemy mieli twardy orzech do zgryzienia, poszukując sposobu rozwiązania go i nie biorąc pod uwagę zastosowania pętli czy też rozgałęzień.