Wstęp

C# jest językiem przeznaczonym dla programistów wykorzystujących platformę .NET firmy Microsoft. Microsoft przestawia C# jako nowoczesny i innowacyjny język służący do tworzenia oprogramowania z wykorzystaniem platformy .NET. C# 6.0 kontynuuje tę tradycję dostarczając nowe funkcje ułatwiające programowanie dynamiczne i równoległe. Ponadto pozwala na wykonanie zadań za pomocą mniejszej ilości kodu. C# nadal umożliwia zarówno programowanie deklaratywne, jak i funkcyjne, a zarazem zapewnia świetne wsparcie dla programowania obiektowego. Podsumowując, C# pozwala na pisanie kodu w stylu, który najlepiej pasuje do potrzeb programisty.

Zaczęliśmy pisać tę książkę razem, na podstawie problemów, z jakimi zetknęliśmy się podczas nauki C#. Książka rozrosła się wraz z pojawianiem się nowych wyzwań i możliwości języka. W tym wydaniu zmieniliśmy podejście do wielu rozwiązań, uwzględniając najnowsze udoskonalenia C#, takie jak nowy poziom wyrażeń (`nameof`, interpolacja łańcuchów, operator warunkowy `null`, inicjalizatory indeksów), deklaracje elementów członkowskich (inicjalizatory auto-właściwości, auto-właściwości służące tylko do pobierania wartości, elementy członkowskie funkcji wewnątrz wyrażeń) oraz funkcje na poziomie instrukcji (filtry wyjątków). Uwzględniliśmy także nowe elementy programowania dynamicznego (C# 4.0) i programowania asynchronicznego (C# 5.0), zarówno w istniejących, jak i w nowych przepisach. Chcieliśmy w ten sposób ułatwić Czytelnikom zrozumienie ich użycia.

Mamy nadzieję, że te dodatki pomogą Czytelnikom uporać się z pewnymi powszechnymi (a także z nieco rzadszymi) problemami oraz wątpliwościami, które pojawiają się u każdego, kto rozpoczyna naukę C#, zgłębia tajniki nowych możliwości języka lub pracuje nad zagadnieniami, które wymagają zboczenia z utartych ścieżek podczas pracy. W niektórych przepisach rozwiązujemy problemy, które według nas zostały pominięte w bibliotece .NET Framework Class Library (FCL), chociaż Microsoft udostępnił tysiące funkcji, dzięki którym nie musimy na nowo odkrywać Ameryki. Niektóre z tych rozwiązań mogą się nam przydać od razu, a z niektórymi problemami możemy się nigdy nie spotkać, lecz mamy nadzieję, że ta książka pomoże Czytelnikom wykorzystać pełnię możliwości języka C# i platformy .NET.

Książka została opracowana z myślą o tych typach problemów, które Czytelnik będzie rozwiązywać w ciągu swojej kariery programisty C#. Te rozwiązania są zwane *przepisami*; każdy przepis dotyczy jednego problemu, prezentuje jego rozwiązanie, zawiera

jego omówienie wraz z powiązanymi informacjami. Na końcu przepisu znajduje się lista zasobów, dotyczących na przykład lokalizacji w FCL, w której można znaleźć więcej informacji na temat wykorzystywanych klas. Wymienione są też inne książki, dotyczące tego problemu, powiązane artykuły i inne przepisy. Format pytań i odpowiedzi dostarcza kompletnych rozwiązań problemów, zwiększając czytelność i ułatwiając korzystanie z książki. Niemal każdy przepis zawiera kompletny, udokumentowany kod przykładowy, demonstrujący sposób rozwiązania problemu, omówienie działania technologii, a także listę technologii alternatywnych, ograniczeń i innych ważnych czynników.

Dla kogo jest ta książka

Z tej książki mogą korzystać nie tylko doświadczeni programiści C# lub .NET – jej przeznaczeniem jest służyć użytkownikom na wszystkich poziomach zaawansowania. Ta książka zawiera rozwiązania problemów, z którymi programiści stykają się na co dzień, a także niektórych rzadziej spotykanych problemów. Przepisy są przeznaczone dla programistów, którzy muszą rozwiązać problemy już teraz, a nie dla tych, którzy muszą się najpierw uczyć teorii. Chociaż podręczniki i przewodniki mogą nas nauczyć ogólnych koncepcji, zwykle nie pomogą w rozwiązaniu prawdziwych problemów. My wybraliśmy metodę nauczania na podstawie przykładów, która wykorzystuje naturalny sposób nauki większości osób.

Z większością problemów, opisanych w tej książce, programiści C# spotykają się dość często, lecz niektóre bardziej zaawansowane problemy wymagają bardziej skomplikowanych rozwiązań, które wymagają wykorzystania wielu technik. Każdy przepis został opracowany tak, aby umożliwić szybkie zrozumienie problemu, zaprezentować jego rozwiązanie oraz wskazać potencjalne konsekwencje. Dzięki temu Czytelnicy mogą szybko i efektywnie rozwiązać swoje problemy.

Aby zaoszczędzić Czytelnikom wysiłku związanego z przepisywaniem rozwiązania, na stronie wydawnictwa, prezentującej tę książkę, udostępniamy przykładowy kod zawarty w tej książce. Proponowany przez nas tryb kopiuj-wklej ułatwi mniej doświadczonym programistom sprawdzenie dobrych praktyk programistycznych w akcji. Pakiet kodu przykładowego zawiera przypadek testowy, który wykorzystuje każde rozwiązanie, lecz w samej książce można znaleźć wystarczająco dużo kodu, by zaimplementować rozwiązania bez kodu przykładowego. Kod przykładowy jest dostępny na stronie tej książki (https://github.com/oreillymedia/c_sharp_6_cookbook).

Co będzie potrzebne

Aby uruchomić przykłady z tej książki, potrzebny jest komputer z systemem operacyjnym Windows w wersji 7 lub późniejszej. Kilka rozwiązań związanych z zagadnieniami sieciowymi i XML wymaga działającego serwera Microsoft Internet Information Server

Kup ksi k

(IIS) w wersji co najmniej 7.5, a przykłady dotyczące FTP w rozdziale 9 wymagają lokalnie skonfigurowanego serwera FTP.

Aby otworzyć i skompilować przykłady z tej książki, potrzebny jest program Visual Studio 2015. Jeśli Czytelnik jest zaznajomiony z narzędziem Framework SDK oraz z jego kompilatorem wiersza poleceń, nie powinien mieć żadnych trudności ze śledzeniem tekstu tej książki oraz uruchomieniem przykładowego kodu.

Uwagi dotyczące plaformy

Rozwiązania przedstawione w tej książce zostały utworzone z wykorzystaniem programu Visual Studio 2015. Różnice między C# 6.0 i C# 3.0 są znaczące, co zostało odzwierciedlone w uaktualnionym kodzie przykładowym.

Warto wspomnieć, że chociaż obecna jest już wersja 6.0 języka, to najnowsze wydanie platformy .NET ma numer wersji 4.6. Język C# nadal podlega udoskonaleniu wraz z każdą nową wersją platformy .NET, a wersja C# 6.0 oferuje wiele możliwości, które pozwalają na tworzenie programów w dowolnym stylu, który najlepiej odpowiada bieżącemu zadaniu.

Organizacja książki

Ta książka zawiera 13 rozdziałów. Każdy z nich skupia się na konkretnym zagadnieniu, dla którego przedstawione są rozwiązania w języku C#. Poniżej przedstawiamy podsumowanie każdego rozdziału, aby Czytelnik poznał ogólny zarys treści tej książki:

Rozdział 1, Klasy i typy generyczne

: Ten obszerny rozdział zawiera przepisy dotyczące klas i struktur danych, a także typów generycznych, które pozwalają na przetwarzanie wartości różnych typów w jednakowy sposób. Przepisy znajdujące się w tym rozdziale dotyczą szerokiej tematyki, począwszy od typów zamkniętych przez przekształcanie klas do pełnego systemu przetwarzania argumentów z wiersza poleceń, aż po zagadnienia projektowania klas. Znajdziemy tu przepisy, które w ogólny sposób ułatwiają zrozumienie typów generycznych, a także przepisy prezentujące sytuacje, w których te typy sprawdzają się najlepiej, prezentujące wsparcie, jakie oferuje dla nich platforma .NET, a także pokazujące, jak można utworzyć własne implementacje kolekcji.

Rozdział 2, Kolekcje, enumeratory i iteratory

: Ten rozdział zawiera przepisy wykorzystujące kolekcje, enumeratory i iteratory. Przepisy dotyczące kolekcji wykorzystują – a także rozszerzają ich funkcje – tablice (jedno i wielowymiarowe, jak również tablice nieregularne), klasę `List<T>` i wiele innych klas kolekcji. Omówione są także kolekcje generyczne oraz różne sposoby tworzenia własnych, silnie typowanych kolekcji. Opisujemy tworzenie własnych enumeratorów, pokazujemy, jak można zaimplementować iteratory typów generycznych

Kup ksi k

i niegenerycznych, a także jak używać iteratorów w implementacji pętli `foreach`. Opisujemy też niestandardowe implementacje iteratorów.

Rozdział 3, Typy danych
: Ten rozdział dotyczy łańcuchów, liczb i typów wyliczeniowych. Zebrane w nim przepisy prezentują kodowanie i dekodowanie łańcuchów, przekształcenia liczbowe, a także sprawdzanie, czy łańcuchy zawierają wartość liczbową. W rozdziale tym zajmujemy się też wyświetlaniem, przekształcaniem i testowaniem typów wyliczeniowych oraz wykorzystaniem wyliczeń zawierających flagi bitowe.

Rozdział 4, Zapytania LINQ i wyrażenia lambda
: Ten rozdział dotyczy wykorzystania technologii Language Integrated Query (LINQ), włącznie z przykładem równoległej wersji LINQ (PLINQ). Rozdział ten zawiera przepisy wykorzystujące wiele standardowych operatorów zapytań oraz pokazujące sposób użycia niektórych operatorów zapytań, które choć nie są słowami kluczowymi języka, to jednak są dość potężne. Omówione są też wyrażenia lambda, a prezentujące je przepisy wykorzystują je zamiast starego typu delegatów.

Rozdział 5, Debugowanie i obsługa wyjątków
: Ten rozdział dotyczy debugowania i obsługi wyjątków. Prezentujemy przepisy, które wykorzystują typy danych, dostępne w przestrzeni nazw `System.Diagnostics`, czyli logi zdarzeń, liczniki wydajności i niestandardowe widoki debugera dla naszych typów. Skupiamy się też na najlepszych sposobach implementacji obsługi wyjątków w aplikacji. Znajdziemy tu także przepisy pokazujące, jak zapobiegać wystąpieniu nieobsłużonych wyjątków. Wreszcie, przygotowaliśmy też przepisy prezentujące sposoby radzenia sobie z różnymi trudnymi sytuacjami, takimi jak wyjątki z metod z późnym wiązaniem oraz asynchroniczna obsługa wyjątków.

Rozdział 6, Odbicie i programowanie dynamiczne
: Ten rozdział prezentuje użycie wbudowanego systemu inspekcji zestawów, dostępnego w platformie.NET, w celu określenia typów, interfejsów i metod zaimplementowanych w zestawie. Pokazujemy też, jak uzyskać do nich dostęp z wykorzystaniem mechanizmu późnego wiązania. Prezentujemy również dynamiczny styl programowania z użyciem `dynamic`, `ExpandoObject` i `DynamicObject`.

Rozdział 7, Wyrażenia regularne
: Ten rozdział opisuje przydatny zestaw klas, służących do sprawdzania dopasowania łańcuchów do wyrażeń regularnych. W przepisach iterujemy po wynikach dopasowania wyrażeń regularnych, dzielimy łańcuchy na tokeny, wyszukujemy i zastępujemy znaki, a także weryfikujemy składnię wyrażeń regularnych. Uwzględniliśmy też przepis zawierający najpopularniejsze wzorce wyrażeń regularnych.

Rozdział 8, Operacje wejścia-wyjścia w systemie plików
: Ten rozdział opisuje interakcje z systemem plików na trzy różne sposoby: pierwszy dotyczy typowych interakcji z plikiem; drugi dotyczy interakcji opartych

Kup ksi k

na katalogach czy folderach; a trzeci dotyczy zaawansowanych zagadnień operacji wejścia-wyjścia w systemie plików.

Rozdział 9, Zagadnienia sieciowe
: Ten rozdział omawia oferowane przez platformę .NET opcje łączności oraz dostępu do zasobów sieciowych i treści WWW z poziomu kodu. Przepisy dotyczą bezpośredniego wykorzystania protokołu TCP/IP, wykorzystywania potoków nazwanych na potrzeby komunikacji, tworzenia własnego skanera portów, określania konfiguracji strony WWW z poziomu kodu i wielu innych zagadnień.

Rozdział 10, XML
: Jeśli korzystamy z platformy .NET, prawdopodobnie kiedyś zetkniemy się z formatem XML. W tym rozdziale omawiamy pewne przypadki użycia XML, a także pokazujemy, jak obsługiwać ten format programowo za pomocą techniki LINQ to XML, obiektu `XmlReader`/`XmlWriter` i `XmlDocument`. Rozdział ten zawiera przykłady wykorzystujące zarówno XPath, jak i XSLT, a także obejmuje zagadnienia weryfikacji XML i przekształceń formatu XML w HTML.

Rozdział 11, Bezpieczeństwo
: Istnieje wiele metod pisania niebezpiecznego kodu, a tylko kilka sposobów tworzenia bezpiecznego kodu. W tym rozdziale omawiamy zagadnienia, takie jak kontrolowanie dostępu do typów, szyfrowanie i odszyfrowywanie, bezpieczne przechowywanie danych, a także stosowanie zabezpieczeń programistycznych i deklaratywnych.

Rozdział 12, Wątki, synchronizacja i współbieżność
: Ten rozdział dotyczy zagadnienia wykorzystania wielu wątków w programie .NET, a także implementowania wątków w aplikacji, ochrony zasobów oraz zapewnienia bezpiecznego dostępu współbieżnego, przechowywania danych wątków, uruchamianie zadań w kolejności oraz używania podstawowych elementów synchronizacji platformy .NET w celu pisania kodu bezpiecznego wątkowo.

Rozdział 13, Przybornik
: W tym rozdziale zawarte są przepisy dotyczące różnorodnych operacji, często wykonywanych przez programistów, takich jak określanie lokalizacji zasobów systemowych, wysyłania wiadomości e-mail oraz pracy z usługami. W rozdziale tym uwzględniono też rzadziej używane, lecz przydatne elementy aplikacji, takie jak kolejki komunikatów, uruchamianie kodu w osobnej domenie AppDomain i określanie wersji podzespołów w Global Assembly Cache (GAC).

Niektóre przepisy są powiązane; w tej sytuacji w puncie *Zobacz także*, a czasem także w tekście punktu *Omówienie* znajdziemy odpowiednie odniesienie do innego przepisu.

Kup ksi k

Co zostało pominięte

Ta książka nie jest podręcznikiem ani elementarzem języka C#. Do godnych polecenia przewodników po języku oraz podręczników, które zostały wydane przez O'Reilly, należą: *C# 6.0 in a Nutshell* autorstwa Josepha Albahari i Bena Albahari; *C# 6.0 Pocket Reference*, także autorstwa Josepha Albahari i Bena Albahari; oraz *Concurrency in C# Cookbook* autorstwa Stephena Cleary. Nieoceniona jest także biblioteka MSDN. Znajdziemy ją w pakiecie Visual Studio 2015, a także online pod adresem *http://msdn.microsoft.com*.

Konwencje stosowane w tej książce

Ta książka wykorzystuje następujące konwencje typograficzne:

Kursywa
: Stosowana w adresach URL, nazwach katalogów i plików, w nazwach opcji, a także czasami w celu wyróżnienia.

`Krój o stałej szerokości`
: Stosowany w listingach oraz do wyróżnienia elementów kodu, takich jak polecenia, opcje, przełączników, zmiennych, atrybutów, kluczy, funkcji, typów, klas, przestrzeni nazw, metod, właściwości, parametrów, wartości, obiektów, zdarzeń, procedur obsługi zdarzeń, znaczników XML, znaczników HTML, makr, treści plików i wyniku poleceń.

`Pogrubiony krój o stałej szerokości`
: Wykorzystywany w listingach w celu podkreślenia ważnego fragmentu kodu.

`Krój o stałej szerokości z kursywą`
: Wykorzystywany do oznaczenia części kodu, które należy zastąpić.

`//...`
: Wielokropki w kodzie C# reprezentują tekst pominięty w celu większej przejrzystości.

`<!--...-->`
: Wielokropki w schematach XML i kodzie dokumentów oznaczają tekst pominięty w celu większej przejrzystości.

Ta ikona oznacza wskazówkę, sugestię lub ogólną uwagę.

Kup ksi k

Ta ikona oznacza ostrzeżenie lub przestrogę.

Kilka słów na temat kodu

Prawie każdy przepis znajdujący się w tej książce zawiera jeden lub kilka przykładów kodu. Te przykłady są zawarte w poszczególnych rozwiązaniach i mają postać fragmentów kodu oraz całych projektów, które można natychmiast wykorzystać we własnej aplikacji. Większość przykładowego kodu została umieszczona w klasie lub strukturze, co ułatwia jego wykorzystanie we własnej aplikacji. Ponadto, uwzględnione zostały także dyrektywy `using` dla każdego przepisu, dzięki czemu Czytelnik nie musi szukać modułów, które należy dołączyć do kodu.

Pełna obsługa błędów została uwzględniona tylko w krytycznych miejscach, takich jak parametry wejściowe. Dzięki temu możemy z łatwością rozpoznać poprawne i niepoprawne dane wejściowe. W wielu przepisach pominięto obsługę błędów. To ułatwia zrozumienie rozwiązania, ponieważ Czytelnik może się skupić na kluczowych koncepcjach.

Korzystanie z przykładowego kodu

Przykładowy kod z tej książki można znaleźć pod adresem:

https://github.com/oreillymedia/c_sharp_6_cookbook

Celem tej książki jest pomoc w wykonaniu różnych zadań. Ogólnie rzecz biorąc, kod zawarty w tej książce można wykorzystać w swoich programach i dokumentacji. Czytelnik nie musi się z nami kontaktować w celu uzyskania zezwolenia, chyba że zamierza odtworzyć znaczną część kodu. Na przykład, napisanie programu, który wykorzystuje kilka fragmentów kodu z tej książki, nie wymaga zezwolenia. Sprzedaż lub dystrybucja płyty CD-ROM zawierającej przykłady z książek wydawnictwa O'Reilly wymaga zezwolenia. Odpowiedź na pytanie, zawierająca cytat z tej książki wraz z kodem przykładowym nie wymaga zezwolenia. Włączenie znacznej części przykładowego kodu z tej książki do dokumentacji produktu nie wymaga zezwolenia.

Doceniamy, lecz nie wymagamy not wskazujących pochodzenie kodu. Mamy tu na myśli tytuł, autora, wydawcę i numer ISBN książki. Na przykład: *C# 6.0 Księga przepisów*, wydanie 4., autorstwa Jay'a Hilyarda i Stephena Teilheta. Copyright 2015 Jay Hilyard i Stephen Teilhet, 978-83-7541-160-7.

Jeśli Czytelnik uważa, że wykorzystanie przez niego przykładowego kodu wykracza poza opisane tu przypadki dopuszczalnego użycia, zapraszamy do kontaktu pod adresem *permissions@oreilly.com*.

Kup ksi k

Podziękowania

Praca nad tą książką zaczęła się dla nas, gdy zaczęliśmy się zapoznawać z językiem C#. Kontynuowaliśmy ją przez lata, wykorzystując język na wiele nowych i ekscytujących sposobów. Ponieważ od ostatniego wydania książki pojawiły się nie tylko nowe funkcje z wersji C# 6.0, ale także z wersji C# 4.0 i C# 5.0, zdecydowaliśmy się, że nadszedł czas, aby przejrzeć trzy pierwsze wydania i sprawdzić, jak można udoskonalić istniejące przepisy oraz poznać lepsze sposoby wykonania zadań programistycznych w C#. Wraz ze zdobywaniem coraz obszerniejszej wiedzy na temat języka C# i platformy .NET, pracowaliśmy ciężko nad tym wydaniem, aby przybliżyć Czytelnikom rozwój języka C# i pokazać, jak lepiej wykonywać swoją pracę.

Nie napisalibyśmy tej książki bez wsparcia wymienionych dalej osób, dlatego chcielibyśmy podziękować im za ich wysiłki.

Nasze uznanie należy się Brianowi MacDonaldowi (naszemu wydawcy), Heather Scherer, Rachel Monaghan, Nickowi Adamsowi i Sarze Peyton, którzy pilnowali terminów i wykonali kawał dobrej roboty, aby ta książka została ukończona i znalazła się na półkach księgarskich. Dziękujemy za Wasze wysiłki.

Chcemy też wyrazić wdzięczność zespołowi naszych recenzentów technicznych, składającemu się ze Stevena Munyana, Lee Cowarda i Nicka Pinkhama. Doceniamy cały czas spędzony na pomaganiu nam w ulepszaniu tej książki, a także dogłębne komentarze. Ta książka nie powstałaby bez Waszych cennych uwag i oboje Wam za to dziękujemy.

Od Jay'a Hilyarda

Dziękuję Steve'owi Teilhetowi za jego pomysły, poczucie humoru i chęć ponownego przyłączenia się do tej przygody. Praca z Tobą zawsze sprawiała mi radość, nawet jeśli zwykle odbywała się nocami i podczas weekendów, i niemal zawsze wirtualnie.

Chciałbym podziękować mojej żonie Brooke. Chociaż wiedziałaś, że będę musiał poświęcić czas przeznaczony dla rodziny, byłaś dla mnie wielkim wsparciem, służyłaś zachętą i pomocą. Jak zwykle, nie byłoby to możliwe bez Twojej pomocy. Dziękuję i kocham Cię!

Dziękuję moim synom, Owenowi i Drewowi. Ciągle zaskakuje mnie to, że potrafią sprawić, bym spojrzał na problem pod innym kątem. Czuję się dumny z Waszych osiągnięć. To, że oboje interesujecie się tą samą dziedziną, którą się zajmuję, bardzo mnie cieszy. Nie mógłbym sobie wymarzyć lepszych synów.

Dziękuję Philowi i Gailowi za ich wsparcie i zrozumienie, które mi okazywali, gdy musiałem pracować podczas wakacji, a także za obecność i gotowość do pomocy, jakiej mogą udzielić tylko dziadkowie. Dziękuję też mojej Mamie za miesięczną dawkę rozsądku.

Dziękuję „grupie" dobrych przyjaciół: Sethowi i Katie Fiermonti, Tomowi Bebbingtonowi i Jennie Roberts. Dzięki przyjaciołom wszystko staje się lepsze, szczególnie jeśli towarzyszy temu dobre piwo.

Kup ksi k

Dziękuję Scottowi Cronshawowi, Billowi Bolevicowi, Melissie Jurkoic, Mike'owi Kennie'owi, Alexowi Shore'owi, Dave'owi Flandersowi, Aaronowi Reddishowi, Rakshitowi Jain, Jasonowi Phelpsowi, Joshowi Clairmontowi, Bobowi Blaisowi, Kim Serpie, Stu Savage'owi, Gaurangowi Patelowi, Jesse'owi Petersowi, Kenowi Jonesowi, Maheshowi Unnikrishnanowi, T Antoniowi, Mary Ellen Sawyer, Jonowi Godboutowi, Atulowi Kaulowi, Markowi Millerowi, Richowi Labenskiemu, Lance'owi Simpsonowi, Timowi Beaulieu i Lee Horganowi za stworzenie wspaniałego zespołu osób, z którymi mogłem współpracować. Każdy z Was pracuje niesłychanie ciężko i doceniam wszystko, co robicie.

Wreszcie, chciałbym ponownie podziękować mojej rodzinie i przyjaciołom za pytania o książkę, której nie rozumieli i za ich ekscytację tym przedsięwzięciem.

Od Steve'a Teilheta

Jestem dumny z tego, że mogę liczyć na Jay'a Hilyarda, jak na dobrego przyjaciela, wspaniałego współpracownika i ciężko pracującego współautora. Nie codziennie spotyka się osobę, która będzie nie tylko zaufanym przyjacielem, ale z którą będzie się tak świetnie pracować. Dziękuję za kolejną udaną książkę.

Moja żona Kandis Teilhet towarzyszyła mi na każdym kroku tej drogi, dając mi siłę do wytrwania i ukończenia tej pracy. Mojej miłości do Ciebie nie zdołają wyrazić żadne słowa.

Moi dwaj synowie Patrick i Nicholas Teilhet wygładzali moje ścieżki. Nie mógłbym sobie wymarzyć lepszych synów. Teraz, gdy wkraczacie w kolejny etap życia, jestem podekscytowany tym, co zdołacie osiągnąć; może też napiszecie książkę.

Dziękuję mojej mamie, tacie i bratu, którzy byli zawsze gotowi, aby mnie wysłuchać i wesprzeć.

Wreszcie, chciałbym podziękować zespołowi IBM, w którego skład wchodzili Larry Rose, Babita Sharma, Jessica Berliner, Jeff Turnham, John Peyton, Kris Duer, Robert Stanzel, Shu Wang, Bingzhou Zheng, Dave Steinberg, Dave Stewart, Jason Todd, Alexei Pivkine, Joshua Clark, William Frontiero, Matthew Murphy, Omer Trip, Marco Pistoia, Enrique Varillas, Guillermo Hurtado, Bao Lu, Mary Santo, Diane Redfearn, Urmi Chatterjee, Joshua Ho, Kenneth Cheung, Andrew Mak, Daniel Nguyen, Jennifer Calder, Tahseen Shabab, Srinivas Sripada, David Marshak, Larry Gerard, Douglas Wilson, Steve Hikida i wielu innych. Wasza ciężka praca i zdolności są inspirujące.

Kup ksi k

Kup ksi k

Rozdział 1
Klasy i typy generyczne

1.0 Wprowadzenie

Zawarte w niniejszym rozdziale przepisy dotyczą podstaw języka C#. Tematy obejmują klasy i struktury, sposób ich wykorzystywania i różnice między nimi, a także powody stosowania jednego typu danych zamiast drugiego. Bazując na tej wiedzy będziemy konstruować klasy z funkcjonalnością odziedziczoną, czyli przykładowo takie, które mogą być sortowane, przeszukiwane, usuwane i klonowane. Dodatkowo zagłębimy się w tematyce unii, inicjalizacji pól, wyrażeń lambda, metod częściowych, delegatów pojedynczych i zbiorowych, domknięć, funktorów oraz kilku innych rzeczy. Rozdział ten zawiera również przepis dotyczący procesu parsowania parametrów wiersza poleceń, co jest jednym z najbardziej lubianych tematów.

Zanim jednak zagłębimy się w konkretne przepisy, powtórzymy sobie pewne kluczowe informacje w zakresie możliwości klas, struktur i typów generycznych, jakie wynikają z paradygmatu programowania zorientowanego obiektowo. Klasy są znacznie bardziej elastyczne od struktur. Struktury również mogą implementować interfejsy, ale w przeciwieństwie do klas nie mogą dziedziczyć po innych klasach lub strukturach. Ograniczenie to uniemożliwia nam tworzenie hierarchii struktur, co z powodzeniem możemy osiągnąć przy użyciu klas. Implementowany w ramach abstrakcyjnej klasy bazowej polimorfizm w przypadku struktur również jest niedopuszczalny, jako że struktura nie może dziedziczyć po innej klasie, z wyjątkiem konwersji boxing do typu `Object`, `ValueType` lub `Enum`.

Struktury, podobnie jak każdy inny typ wartości, dziedziczą niejawnie z klasy `System.ValueType`. Na pierwszy rzut oka struktura podobna jest do klasy, lecz w rzeczywistości znacznie się od niej różni. Wiedza na temat tego, kiedy należy używać struktury, a kiedy klasy w dużym stopniu ułatwi nam projektowanie aplikacji. Niepoprawne korzystanie ze struktur może skutkować niewydajnym, a przy tym trudnym do zmodyfikowania kodem.

Struktury mają nad typami referencyjnymi pewną przewagę wydajnościową. Jeśli struktura została przydzielona na stosie (a więc nie jest przechowywana w obrębie typu referencyjnego), dostęp do struktury i jej danych będzie nieco szybszy niż w przypadku

Kup ksi k

typu referencyjnego na stercie. By dostać się do własnych danych, obiekty typu referencyjnego muszą sięgać na stertę z użyciem referencji. Druga i jednocześnie dużo większa zaleta struktur odnosi się do czyszczenia pamięci. Proces czyszczenia przydzielonej na stosie pamięci struktury sprowadza się jedynie do prostej zmiany adresu wskazywanego przez wskaźnik stosu, co odbywa się przy powrocie z wywołania metody. Wywołanie to jest dużo szybsze w porównaniu z mechanizmem odzyskiwania pamięci, który automatycznie sprząta za nas typy referencyjne z zarządzanej sterty. Jako że proces sprzątania jest nieco odroczony w czasie, związany z nim koszt nie jest od razu zauważalny.

Gdy zachodzi potrzeba przekazania struktur do metod przez wartość, ich wydajność w porównaniu z wydajnością klas wypada zwyczajnie słabo. Ponieważ struktury przechowywane są na stosie, w momencie przekazania ich do metody przez wartość struktury wraz z zawartymi w nich danymi muszą zostać skopiowane do nowej zmiennej lokalnej (do parametru metody, który posłuży do odebrania struktury). Kopiowanie struktury trwa znacznie dłużej niż przekazanie do metody pojedynczej referencji do obiektu, chyba że rozmiar struktury jest taki sam lub mniejszy niż rozmiar wskaźnika na danej maszynie. Tym samym koszt skopiowania na maszynie 32-bitowej struktury o rozmiarze 32 bitów będzie tak samo niski, jak przekazanie jej przez referencję (która ma rozmiar wskaźnika). Pamiętajmy o tym, gdy następnym razem będziemy wybierać pomiędzy klasą a strukturą. O ile tworzenie, uzyskiwanie dostępu lub usuwanie obiektów klas może trwać dłużej, o tyle możemy dużo stracić na wydajności, gdy zachodzi potrzeba wielokrotnego przekazania struktury przez wartość do jednej lub wielu metod. Utrzymywanie rozmiarów struktur na niskim poziomie minimalizuje straty wydajności, jakie ponosimy przekazując je przez wartość.

Klasy używamy, gdy:

- Jej tożsamość jest istotna. Struktury kopiowane są niejawnie w momencie przekazania ich do metody przez wartość.
- Będzie ona zajmować dużą ilość pamięci.
- Jej pola wymagają inicjalizatorów.
- Musimy odziedziczyć coś z klasy bazowej.
- Potrzebujemy polimorficznego zachowania, tj. musimy zaimplementować abstrakcyjną klasę bazową, z której następnie utworzymy kilka podobnych i dziedziczących po niej klas (zwróćmy uwagę, że polimorfizm można również zaimplementować przy użyciu interfejsów, ale nie zaleca się stosowania interfejsu w przypadku typu wartości, bo gdy struktura zostanie skonwertowana do typu interfejsu, wówczas konwersja boxing spowoduje spadek wydajności).

Struktury używamy, gdy:

- Będzie się ona zachowywać niczym typ prymitywny (`int`, `long`, `byte`, itd.).
- Musi zajmować niewielką ilość pamięci.

Kup ksi k

- Wywołujemy metodę `P/Invoke`, która wymaga przekazania struktury przez wartość. *Platform Invoke*, lub w skrócie P/Invoke, umożliwia odwołanie się z poziomu kodu zarządzanego do metody niezarządzanej, wyeksponowanej w ramach biblioteki DLL. W wielu przypadkach niezarządzana metoda biblioteki DLL wymaga przekazania do niej struktury przez wartość. Skorzystanie ze struktury będzie w tym przypadku nie tylko bardziej wydajne, ale będzie w takim wypadku jedynym możliwym sposobem dokonania tego.
- Chcemy zredukować wpływ modułu odzyskiwania pamięci na wydajność aplikacji.
- Jej pola będą inicjowane wyłącznie do ich wartości domyślnych. Wartością domyślną dla typów numerycznych będzie `0`, dla typów Boolean `false`, a dla typów referencyjnych `null`. W wersji 6.0 języka C# struktury mogą mieć domyślny konstruktor, który może posłużyć do inicjalizacji pól struktury do wartości niestandardowych.
- Nie musimy niczego dziedziczyć po klasie bazowej (poza klasą `ValueType`, po której dziedziczą wszystkie struktury).
- Nie potrzebujemy polimorficznego zachowania.

W przypadku przekazywania struktur do metod oczekujących obiektów, co ma miejsce chociażby w przypadku niegenerycznych kolekcji w bibliotece Framework Class Library (FCL), struktury mogą obniżyć wydajność aplikacji. Przekazanie struktury (lub też jakiegokolwiek innego typu prostego) do metody wymagającej obiektu skutkować będzie konwersją boxing tej struktury. *Boxing* odnosi się do procesu opakowywania typu wartości w określony obiekt. Operacja ta jest czasochłonna i może negatywnie wpływać na wydajność.

Dodajmy jeszcze, że typy generyczne pozwalają nam pisać bezpieczny i wydajny kod oparty na kolekcjach i wzorcach. Typy generyczne dają nam sporo dodatkowych możliwości, ale wraz z nimi pojawia się konieczność ich prawidłowego stosowania. Jeśli rozważamy zamianę naszych obiektów `ArrayList`, `Queue`, `Stack` i `Hashtable` na ich generyczne odpowiedniki, warto zapoznać się z przepisami 1.9 i 1.10. Dowiemy się z nich przykładowo, że konwersja taka nie zawsze jest łatwa oraz że istnieją określone powody, dla których nie powinniśmy jej wcale dokonywać.

1.1 Tworzenie unii

Problem

Musimy utworzyć typ danych, który zachowywał się będzie podobnie jak unia w języku C++. Unia jest typem danych, który przydaje się głównie w scenariuszach interoperacyjnych, gdzie kod niezarządzany przyjmuje i/lub zwraca unię. Nie zaleca się jej wykorzystywania w innych sytuacjach.

Kup ksi k

Rozwiązanie

Skorzystamy ze struktury oznaczonej atrybutem `StructLayout` (z podanym w konstruktorze rodzajem układu `LayoutKind.Explicit`). Każde pole w tej strukturze oznaczymy atrybutem `FieldOffset`. Poniższa struktura definiuje unię, w której będziemy mogli przechować pojedynczą wartość numeryczną ze znakiem:

```
using System.Runtime.InteropServices;
[StructLayoutAttribute(LayoutKind.Explicit)]
struct SignedNumber
{
   [FieldOffsetAttribute(0)]
   public sbyte Num1;
   [FieldOffsetAttribute(0)]
   public short Num2;
   [FieldOffsetAttribute(0)]
   public int Num3;
   [FieldOffsetAttribute(0)]
   public long Num4;
   [FieldOffsetAttribute(0)]
   public float Num5;
   [FieldOffsetAttribute(0)]
   public double Num6;
}
```

Następna struktura podobna jest do struktury `SignedNumber`, z wyjątkiem tego, że prócz wartości numerycznej ze znakiem może ona dodatkowo przechowywać typ `String`:

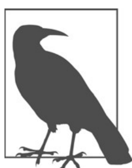
```
[StructLayoutAttribute(LayoutKind.Explicit)]
struct SignedNumberWithText
{
   [FieldOffsetAttribute(0)]
   public sbyte Num1;
   [FieldOffsetAttribute(0)]
   public short Num2;
   [FieldOffsetAttribute(0)]
   public int Num3;
   [FieldOffsetAttribute(0)]
   public long Num4;
   [FieldOffsetAttribute(0)]
   public float Num5;
   [FieldOffsetAttribute(0)]
   public double Num6;
   [FieldOffsetAttribute(16)]
```


Kup ksi k

```
    public string Text1;
}
```

Omówienie

Unie stosuje się zazwyczaj w kodzie C++, jednak z powodzeniem możemy je implementować również w języku C#, posługując się dostępnym w nim typem struktury. U*nia* jest strukturą danych, która w wyznaczonym dla niej obszarze pamięci przechowuje więcej niż jeden typ danych. Przykładowo struktura `SignedNumber` jest unią zbudowaną przy wykorzystaniu struktury C#. Struktura ta przyjmuje dowolną wartość numeryczną ze znakiem dowolnego typu (`sbyte`, `int`, `long`, itd.), przy czym przyjmuje je tylko w jednej lokalizacji w ramach struktury.



Jako że atrybut `StructLayoutAttribute` ma zastosowanie zarówno do struktur, jak i do klas, do utworzenia unii możemy również wykorzystać klasę.

Zwróćmy uwagę, że do konstruktora atrybutu `FieldOffsetAttribute` przekazywana jest wartość `0`. Oznacza to, że dane pole zostanie przesunięte względem początku struktury o zero bajtów. Atrybut ten wykorzystywany jest w połączeniu z atrybutem `StructLayoutAttribute` w celu ręcznego określenia początku poszczególnych pól struktury (inaczej mówiąc, przesunięcia względem początku struktury w pamięci, będącego początkiem każdego z jej pól). Atrybut `FieldOffsetAttribute` możemy stosować wyłącznie z atrybutem `StructLayoutAttribute` ustawionym na `LayoutKind.Explicit`. Nie możemy go jednak wykorzystywać dla statycznych członków wewnątrz struktury.

Unie mogą być niekiedy problematyczne, jako że w praktyce kilka typów danych jest na siebie nałożonych jeden na drugi. Największym problemem jest tu wyodrębnianie poprawnego typu danych ze struktury. Zastanówmy się, co się stanie, jeśli w strukturze `SignedNumber` zdecydujemy się przechować wartość numeryczną typu `long` o nazwie `long.MaxValue`. Później możemy przez przypadek spróbować wyodrębnić tę wartość w formie typu danych `byte`. Jeśli to zrobimy, zostanie nam zwrócony jedynie pierwszy bajt wartości typu long.

Kolejnym problemem może być rozpoczynanie pól struktury z właściwym przesunięciem. Unia `SignedNumberWithText` nakłada na siebie kilka wartości numerycznych ze znakiem, z przesunięciem równym zero. Ostatnie pole tej struktury rozpoczyna się z 16-bajtowym przesunięciem od początku pamięci struktury. Jeśli na którykolwiek z typów numerycznych ze znakiem nałożymy przypadkowo pole tekstowe `Text1`, wówczas w czasie działania programu zostanie rzucony wyjątek. Podstawowa zasada mówi, że możemy nakładać na siebie dowolne typy wartości, ale na typy wartości nie możemy nakładać typów referencyjnych. Jeśli pole `Text1` oznaczone jest poniższym atrybutem:

```
[FieldOffsetAttribute(14)]
```


Kup ksi k

wówczas w momencie uruchomienia programu zostanie rzucony poniższy wyjątek (zwróćmy uwagę, że kompilator nie wychwyci tego problemu):

```
System.TypeLoadException: Could not load type 'SignedNumberWithText' from
assembly 'CSharpRecipes, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=fe85c3941fbcc4c5' because it contains an object field at
offset 14 that is incorrectly aligned or overlapped by a non-object field.
```

W przypadku korzystania w języku C# ze złożonych unii należy zwrócić szczególną uwagę na przesunięcia pól.

Zobacz także

Temat „StructLayoutAttribute Class” w dokumentacji MSDN.

1.2 Umożliwianie sortowania typu

Problem

Mamy typ danych, który przechowywany będzie w postaci elementów na liście `List<T>` lub `SortedList<K,V>`. Korzystając z metody `List<T>.Sort` lub wewnętrznego mechanizmu sortowania klasy `SortedList<K,V>` chcemy umożliwić sobie niestandardowe sortowanie naszych danych w tablicy. Dodatkowo może zajść potrzeba wykorzystania tego typu danych w kolekcji `SortedList`.

Rozwiązanie

Przykład 1-1 demonstruje sposób implementacji interfejsu `IComparable<T>`. Widoczna w tym przykładzie klasa `Square` implementuje ten intefejs w taki sposób, że kolekcje `List<T>` i `SortedList<K,V>` mogą sortować i wyszukiwać obiekty `Square`.

Przykład 1-1 *Umożliwianie sortowania typu poprzez implementację interfejsu IComparable<T>*

```
public class Square : IComparable<Square>
{
    public Square(){}
    public Square(int height, int width)
    {
        this.Height = height;
        this.Width = width;
    }
    public int Height { get; set; }
```


Kup ksi k

```
    public int Width { get; set; }
    public int CompareTo(object obj)
    {
        Square square = obj as Square;
        if (square != null)
            return CompareTo(square);
        throw
          new ArgumentException(
              "Both objects being compared must be of type Square.");
    }
    public override string ToString()=>
                           ($"Height: {this.Height}     Width: {this.Width}");
    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;
        Square square = obj as Square;
        if(square != null)
           return this.Height == square.Height;
        return false;
    }
    public override int GetHashCode()
    {
        return this.Height.GetHashCode() | this.Width.GetHashCode();
    }
    public static bool operator ==(Square x, Square y) => x.Equals(y);
    public static bool operator !=(Square x, Square y) => !(x == y);
    public static bool operator <(Square x, Square y) => (x.CompareTo(y) < 0);
    public static bool operator >(Square x, Square y) => (x.CompareTo(y) > 0);
    public int CompareTo(Square other)
    {
        long area1 = this.Height * this.Width;
        long area2 = other.Height * other.Width;
        if (area1 == area2)
            return 0;
        else if (area1 > area2)
            return 1;
        else if (area1 < area2)
            return -1;
        else
            return -1;
    }
}
```


Kup ksi k

Omówienie

Dzięki implementacji w naszej klasie (lub strukturze) interfejsu `IComparable<T>` możemy skorzystać z dostępnych w klasach `List<T>` i `SortedList<K,V>` algorytmów sortowania. Wcześniej jednak musimy zdefiniować sposób sortowania naszej klasy. Dokonujemy tego w metodzie `IComparable<T>.CompareTo`.

Podczas sortowania listy obiektów `Square` w ramach wywołania metody `List<Square>.Sort` lista sortowana jest z wykorzystaniem interfejsu `IComparable<Square>`. Metoda `Add` klasy `SortedList<K,V>` wykorzystuje ten interfejs do sortowania obiektów w czasie dodawania ich do listy `SortedList<K,V>`.

Interfejs `IComparer<T>` umożliwia sortowanie obiektów w oparciu o różne kryteria, w różnym kontekście. Umożliwia on również sortowanie typów, które nie są naszego autorstwa. Jeśli chcielibyśmy posortować obiekty `Square` względem ich wysokości, moglibyśmy utworzyć nową klasę o nazwie `CompareHeight`, widoczną w przykładzie 1-2, która również będzie implementować interfejs `IComparer<Square>`.

Przykład 1-2 *Umożliwianie sortowania typu poprzez implementację interfejsu IComparer*

```
public class CompareHeight : IComparer<Square>
{
    public int Compare(object firstSquare, object secondSquare)
    {
        Square square1 = firstSquare as Square;
        Square square2 = secondSquare as Square;
        if (square1 == null || square2 == null)
          throw (new ArgumentException(
                       "Both parameters must be of type Square."));
        else
           return Compare(firstSquare,secondSquare);
    }
    #region IComparer<Square> Members
    public int Compare(Square x, Square y)
    {
        if (x.Height == y.Height)
            return 0;
        else if (x.Height > y.Height)
            return 1;
        else if (x.Height < y.Height)
            return -1;
        else
            return -1;
    }
```


Kup ksi k

```
        #endregion
    }
```

Klasa ta zostanie przekazana do parametru `IComparer` procedury `Sort`. Teraz możemy zdefiniować kilka sposobów sortowania naszych obiektów `Square`. Zaimplementowana metoda porównująca musi nie tylko być spójna, ale musi też stosować *porządek liniowy*. Da nam to pewność, że gdy funkcja porównująca zadeklaruje równość dwóch elementów, wówczas będą one całkowicie równe, a tym samym ich równość nie będzie przykładowo wynikiem tego, że jeden z nich jest jedynie nie większy lub nie mniejszy od drugiego.

W celu uzyskania jak największej wydajności utrzymuj metodę CompareTo w krótkiej i wydajnej postaci, ponieważ będzie ona wywoływana wielokrotnie przez metody Sort. W przypadku sortowania tablicy z czterema elementami metoda Compare wywoływana jest przykładowo 10 razy.

Metoda `TestSort` widoczna w przykładzie 1-3 demonstruje użycie klas `Square` i `CompareHeight` z wykorzystaniem instancji `List<Square>` i `SortedList<int,Square>`.

Przykład 1-3 *Metoda TestSort*

```
public static void TestSort()
{
    List<Square> listOfSquares = new List<Square>(){
                            new Square(1,3),
                            new Square(4,3),
                            new Square(2,1),
                            new Square(6,1)};
    // Przetestuj List<String>
    Console.WriteLine("List<String>");
    Console.WriteLine("Original list");
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }
    Console.WriteLine();
    IComparer<Square> heightCompare = new CompareHeight();
    listOfSquares.Sort(heightCompare);
    Console.WriteLine("Sorted list using IComparer<Square>=heightCompare");
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }
```


Kup ksi k

```
        Console.WriteLine();
        Console.WriteLine("Sorted list using IComparable<Square>");
        listOfSquares.Sort();
        foreach (Square square in listOfSquares)
        {
            Console.WriteLine(square.ToString());
        }

        // Przetestuj SortedList
        var sortedListOfSquares = new SortedList<int,Square>(){
                                { 0, new Square(1,3)},
                                { 2, new Square(3,3)},
                                { 1, new Square(2,1)},
                                { 3, new Square(6,1)}};
        Console.WriteLine();
        Console.WriteLine();
        Console.WriteLine("SortedList<Square>");
        foreach (KeyValuePair<int,Square> kvp in sortedListOfSquares)
        {
            Console.WriteLine ($"{kvp.Key} : {kvp.Value}");
        }
    }
```

Powyższy kod wyświetli następujący wynik:

```
List<String>
Original list
Height:1 Width:3
Height:4 Width:3
Height:2 Width:1
Height:6 Width:1
Sorted list using IComparer<Square>=heightCompare
Height:1 Width:3
Height:2 Width:1
Height:4 Width:3
Height:6 Width:1
Sorted list using IComparable<Square>
Height:2 Width:1
Height:1 Width:3
Height:6 Width:1
Height:4 Width:3
SortedList<Square>
0 : Height:1 Width:3
```


Kup książkę

```
1 : Height:2 Width:1
2 : Height:3 Width:3
3 : Height:6 Width:1
```

Zobacz także

Przepis 1.3 oraz temat „IComparable<T> Interface” w dokumentacji MSDN.

1.3 Umożliwianie przeszukiwania typu

Problem

Mamy typ danych, który przechowywany będzie w postaci elementów na liście `List<T>`. Chcemy skorzystać z metody `BinarySearch`, by umożliwić sobie niestandardowe wyszukiwanie naszych danych na liście.

Rozwiązanie

Skorzystamy z interfejsów `IComparable<T>` i `IComparer<T>`. Klasa `Square` z przepisu 1.2 implementuje interfejs `IComparable<T>` w taki sposób, że kolekcje `List<T>` i `SortedList<K,V>` mogą sortować i przeszukiwać tablicę lub kolekcję obiektów `Square`.

Omówienie

Dzięki implementacji w naszej klasie (lub strukturze) interfejsu `IComparable<T>` możemy skorzystać z dostępnych w klasach `List<T>` i `SortedList<K,V>` algorytmów wyszukiwania. Wcześniej jednak musimy zdefiniować sposób przeszukiwania naszej klasy. Dokonamy tego w metodzie `IComparable<T>. CompareTo`.

Aby zaimplementować metodę `CompareTo`, zobacz przepis 1.2.

Klasa `List<T>` dostarcza metodę `BinarySearch`, która dokonuje wyszukiwania na zawartych w tej liście elementach. Elementy te porównywane są z obiektem przekazanym do metody `BinarySearch` w parametrze obiektu. Klasa `SortedList` nie zawiera metody `BinarySearch`; zamiast tego zawiera ona metodę `ContainsKey`, która dokonuje wyszukiwania binarnego na zawartym na liście kluczu. Metoda `ContainsValue` klasy `SortedList` do wyszukiwania wartości wykorzystuje wyszukiwanie liniowe. By wykonać swoją pracę, wyszukiwanie liniowe korzysta z metody `Equals` elementów w kolekcji `SortedList`. Metody `Compare` i `CompareTo` nie wywierają żadnego wpływu na operację wyszukiwania liniowego wykonywaną w klasie `SortedList`, jednak mają one wpływ na wyszukiwanie binarne.

Kup ksi k

Aby wykonać dokładne wyszukiwanie przy użyciu metody `BinarySearch` klasy `List<T>`, musisz najpierw posortować listę `List<T>` korzystając z jej metody `Sort`. Jeśli interfejs `IComparer<T>` przekażesz do metody `BinarySearch`, musisz również przekazać ten sam interfejs do metody `Sort`. W innym przypadku metoda `BinarySearch` może nie być w stanie znaleźć obiektu, którego szukasz.

Metoda `TestSort`, widoczna w przykładzie 1-4, demonstruje użycie klas `Square` i `CompareHeight` z instancjami kolekcji `List<Square>` i `SortedList<int,Square>`.

Przykład 1-4 *Umożliwianie przeszukiwania typu*

```
public static void TestSearch()
{
    List<Square> listOfSquares = new List<Square> {new Square(1,3),
                                                   new Square(4,3),
                                                   new Square(2,1),
                                                   new Square(6,1)};
    IComparer<Square> heightCompare = new CompareHeight();
    // Przetestuj kolekcję List<Square>
    Console.WriteLine("List<Square>");
    Console.WriteLine("Original list");
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }
    Console.WriteLine();
    Console.WriteLine("Sorted list using IComparer<Square>=heightCompare");
    listOfSquares.Sort(heightCompare);
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }
    Console.WriteLine();
    Console.WriteLine("Search using IComparer<Square>=heightCompare");
    int found = listOfSquares.BinarySearch(new Square(1,3), heightCompare);
    Console.WriteLine($"Found (1,3): {found}");
    Console.WriteLine();
    Console.WriteLine("Sorted list using IComparable<Square>");
    listOfSquares.Sort();
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
```


Kup ksi k

```
        }
        Console.WriteLine();
        Console.WriteLine("Search using IComparable<Square>");
        found = listOfSquares.BinarySearch(new Square(6,1)); // Use IComparable
        Console.WriteLine($"Found (6,1): {found}");
        // Przetestuj kolekcję SortedList<Square>
        var sortedListOfSquares = new SortedList<int,Square>(){
                                       {0, new Square(1,3)},
                                       {2, new Square(4,3)},
                                       {1, new Square(2,1)},
                                       {4, new Square(6,1)}};
       Console.WriteLine();
       Console.WriteLine("SortedList<Square>");
       foreach (KeyValuePair<int,Square> kvp in sortedListOfSquares)
       {
           Console.WriteLine ($"{kvp.Key} : {kvp.Value}");
       }
       Console.WriteLine();
       bool foundItem = sortedListOfSquares.ContainsKey(2);
       Console.WriteLine($"sortedListOfSquares.ContainsKey(2): {foundItem}");
       // Nie wykorzystuje interfejsów IComparer lub IComparable,
       // lecz korzysta z wyszukiwania liniowego z metodą Equals,
       // która nie została przeciążona
       Console.WriteLine();
       Square value = new Square(6,1);
       foundItem = sortedListOfSquares.ContainsValue(value);
       Console.WriteLine("sortedListOfSquares.ContainsValue " +
                         $"(new Square(6,1)): {foundItem}");
   }
```

Powyższy kod generuje następujący wynik:

```
List"Square>
Original list
Height:1 Width:3
Height:4 Width:3
Height:2 Width:1
Height:6 Width:1
Sorted list using IComparer"Square>=heightCompare
Height:1 Width:3
Height:2 Width:1
Height:4 Width:3
Height:6 Width:1
```


Kup ksi k

```
Search using IComparer"Square>=heightCompare
Found (1,3): 0
Sorted list using IComparable"Square>
Height:2 Width:1
Height:1 Width:3
Height:6 Width:1
Height:4 Width:3
Search using IComparable"Square>
Found (6,1): 2

SortedList"Square>
0 : Height:1 Width:3
1 : Height:2 Width:1
2 : Height:4 Width:3
4 : Height:6 Width:1
sortedListOfSquares.ContainsKey(2): True
sortedListOfSquares.ContainsValue(new Square(6,1)): True
```

Zobacz także

Przepis 1.2 oraz tematy „IComparable<T> Interface” i „IComparer<T> Interface” w dokumentacji MSDN.

1.4 Zwracanie z metody wielu elementów

Problem

W wielu przypadkach zwracanie przez metodę pojedynczej wartości może okazać się niewystarczające. Potrzebujemy sposobu, dzięki któremu z metody będziemy mogli zwrócić więcej niż jeden element.

Rozwiązanie

Parametrom metody, które posłużą nam do zwracania elementów, przypiszemy słowo kluczowe `out`. Poniższa metoda przyjmuje parametr `inputShape` i na podstawie podanej wartości oblicza wysokość (`height`), szerokość (`width`) oraz głębokość (`depth`):

```
public void ReturnDimensions(int inputShape,
                             out int height,
                             out int width,
                             out int depth)
```


Kup ksi k

```
{
    height = 0;
    width = 0;
    depth = 0;
    // Oblicz wysokość, szerokość i głębokość z podanej wartości inputShape.
}
```

Metoda ta może zostać wywołana w następujący sposób:

```
// Zadeklaruj parametry wyjściowe.
int height;
int width;
int depth;
// Wywołaj metodę i zwróć wysokość, szerokość i głębokość.
Obj.ReturnDimensions(1, out height, out width, out depth);
```

Kolejnym sposobem jest zwrócenie klasy lub struktury zawierającej wszystkie zwracane wartości. W poniższym przykładzie zmodyfikowaliśmy poprzednią metodę, tak by zamiast argumentów out zwracała tym razem strukturę:

```
public Dimensions ReturnDimensions(int inputShape)
{
    // Konstruktor domyślny automatycznie ustawia wartości struktury na 0.
    Dimensions objDim = new Dimensions();
    // Oblicz objDim.Height, objDim.Width, objDim.Depth
    // bazując na wartości inputShape...
    return objDim;
}
```

Sama struktura `Dimensions` zdefiniowana została poniżej:

```
public struct Dimensions
{
    public int Height;
    public int Width;
    public int Depth;
}
```

Wywołanie tej metody byłoby następujące:

```
// Wywołaj metodę i zwróć wysokość, szerokość i głębokość.
Dimensions objDim = obj.ReturnDimensions(1);
```

Zamiast zwracać zdefiniowaną przez użytkownika klasę lub strukturę, możemy skorzystać z obiektu `Tuple`, który zawierał będzie wszystkie zwracane wartości. Ponownie zmodyfikowaliśmy poprzednią metodę, która tym razem zwraca obiekt `Tuple`:

Kup ksi k

```
public Tuple<int, int, int> ReturnDimensionsAsTuple(int inputShape)
{
    // Oblicz objDim.Height, objDim.Width oraz objDim.Depth na podstawie
    // wartości inputShape, przykładowo {5, 10, 15}
    // Utwórz obiekt Tuple z wyliczonymi wartościami
    var objDim = Tuple.Create<int, int, int>(5, 10, 15);
    return (objDim);
}
```

Powyższą metodę możemy wywołać w następujący sposób:

```
// Wywołaj metodę i zwróć wysokość, szerokość i głębokość.
Tuple<int, int, int> objDim = obj.ReturnDimensions(1);
```

Omówienie

Oznaczanie parametru w sygnaturze metody słowem kluczowym `out` wskazuje, że parametr ten zostanie zainicjowany i zwrócony przez tę metodę. Sztuczka ta przydaje się, gdy od metody wymaga się zwrócenia więcej niż jednej wartości. Sama metoda jest w stanie zwrócić co najwyżej jedną wartość, ale dzięki słowu kluczowemu `out` możemy wyznaczyć kilka dodatkowych parametrów, które traktowane będą jak wartość zwracana.

Wskazanie parametru jako wartości zwracanej polega na oznaczeniu go w sygnaturze metody słowem kluczowym `out`, jak w poniższym przykładzie:

```
public void ReturnDimensions(int inputShape,
                             out int height,
                             out int width,
                             out int depth)
{
    ...
}
```

By móc prawidłowo wywołać tę metodę, w podobny sposób musimy również oznaczyć argumenty przy jej wywołaniu:

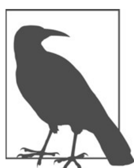
```
obj.ReturnDimensions(1, out height, out width, out depth);
```

W powyższym wywołaniu nie zachodzi potrzeba inicjalizacji argumentów `out`. Możemy je po prostu zadeklarować i przesłać do metody `ReturnDimensions`. Bez względu na to, czy zostały one zainicjowane przed wywołaniem, czy też nie, musimy je zainicjować przed użyciem wewnątrz metody `ReturnDimensions`. Nawet jeśli nie korzystamy z nich w każdym możliwym przepływie tej metody, ich wcześniejsza inicjalizacja jest wymagana. To właśnie dlatego metoda ta rozpoczyna się od trzech poniższych linii kodu:

Kup ksi k

```
height = 0;
width = 0;
depth = 0;
```

Nasuwa się tu pytanie, dlaczego zamiast parametru `out` nie skorzystaliśmy z parametru `ref`, który również pozwala metodzie na zmodyfikowanie wartości oznaczonych w ten sposób argumentów. Jest tak dlatego, że parametry `out` czynią kod niejako samodokumentującym się. Gdy napotykany jest parametr `out`, wówczas zachowuje się on jak wartość zwracana. Dodatkowo parameter `out`, w przeciwieństwie do parametru `ref`, nie wymaga żadnej dodatkowej pracy związanej z jego inicjalizacją przed przekazaniem do metody.



Parametr `out` nie musi być pakowany (boxing) przy wywoływaniu metody. Parametr ten pakowany jest raz, gdy metoda zwraca dane do wywołującego. Jakikolwiek inny rodzaj wywołania (przez wartość lub przez referencję z użyciem słowa kluczowego `ref`) wymaga dwustronnego pakowania wartości. Korzystanie ze słowa kluczowego `out` w scenariuszach pakowania podnosi wydajność zdalnej komunikacji.

Parametr `out` przydaje się, gdy zachodzi potrzeba zwrócenia niewielkiej liczby wartości. Jeśli jednak musimy zwrócić 4, 5, 6 lub więcej wartości, metoda ta staje się mało zaradna. Innym rozwiązaniem umożliwiającym zwracanie większej liczby wartości jest utworzenie i zwrócenie zdefiniowanej przez użytkownika klasy/struktury, bądź też skorzystanie z obiektu `Tuple`, który będzie w sobie zawierać wszystkie konieczne do zwrócenia wartości.

Zwracanie wartości z użyciem klasy lub struktury jest bardzo proste. Tworzymy określony typ (w naszym wypadku strukturę), jak w poniższym przykładzie:

```
public struct Dimensions
{
    public int Height;
    public int Width;
    public int Depth;
}
```

Wypełniamy strukturę konkretnymi danymi, a następnie zwracamy ją z wnętrza metody, jak zostało to przedstawione w Rozwiązaniu tego przepisu.

Drugi sposób, czyli wykorzystanie obiektu `Tuple`, jest jeszcze bardziej eleganckim rozwiązaniem. Obiekty `Tuple` mogą przechowywać dowolną liczbę wartości, które niekoniecznie muszą być tego samego typu. Co więcej, przechowywane przez nas dane w obiekcie `Tuple` są niezmienne. Po dodaniu danych do obiektu `Tuple` z użyciem jego konstruktora lub statycznej metody `Create` danych tych nie będziemy mogli zmienić.

Kup ksi k

Obiekty Tuple mogą przyjmować do ośmu oddzielnych wartości. Jeśli zachodzi potrzeba zwrócenia więcej niż ośmiu wartości, musimy się wówczas posłużyć specjalną klasą Tuple:

```
Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> Class
```

Chcąc utworzyć obiekt klasy Tuple z więcej niż ośmioma wartościami, nie możemy skorzystać z jej statycznej metody Create. Zamiast tego musimy posłużyć się konstruktorem tej klasy. W poniższym przykładzie tworzony jest obiekt Tuple przyjmujący 10 wartości typu całkowitego:

```
var values = new Tuple<int, int, int, int, int, int, int, Tuple<int, int, int>>
(
                    1, 2, 3, 4, 5, 6, 7, new Tuple<int, int, int> (8, 9, 10));
```

Oczywiście na końcach zagnieżdzonych obiektów Tuple również możemy wstawiać kolejne obiekty, tworząc w ten sposób jeden wielki obiekt Tuple, zdolny do przechowywania dowolnej liczby wartości.

Zobacz także

Tematy „Tuple Class” i „Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> Class” w dokumentacji MSDN.

1.5 Parsowanie parametrów wiersza poleceń

Problem

Chcemy, by nasza aplikacja przyjmowała jeden lub więcej parametrów wiersza poleceń w standardowym formacie (opisanym dalej w części Omówienie tego przepisu). Musimy uzyskać dostęp i sparsować wszystkie parametry przesłane do aplikacji z wiersza poleceń.

Rozwiązanie

Aby ułatwić sobie parsowanie parametrów wiersza poleceń, w przykładzie 1-5 posłużymy się klasami Argument, ArgumentDefinition i ArgumentSemanticAnalyzer. Ich kod znajduje się poniżej.

Przykład 1-5 *Klasa Argument*

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Collections.ObjectModel;
```


Kup ksi k

```
public sealed class Argument
{
    public string Original { get; }
    public string Switch { get; private set; }
    public ReadOnlyCollection<string> SubArguments { get; }
    private List<string> subArguments;
    public Argument(string original)
    {
        Original = original;
        Switch = string.Empty;
        subArguments = new List<string>();
        SubArguments = new ReadOnlyCollection<string>(subArguments);
        Parse();
    }
    private void Parse()
    {
        if (string.IsNullOrEmpty(Original))
        {
            return;
        }
        char[] switchChars = { '/', '-' };
        if (!switchChars.Contains(Original[0]))
        {
            return;
        }
        string switchString = Original.Substring(1);
        string subArgsString = string.Empty;
        int colon = switchString.IndexOf(':');
        if (colon >= 0)
        {
            subArgsString = switchString.Substring(colon + 1);
            switchString = switchString.Substring(0, colon);
        }
        Switch = switchString;
        if (!string.IsNullOrEmpty(subArgsString))
            subArguments.AddRange(subArgsString.Split(';'));
    }
    // Zestaw predykatów, który dostarcza cennych informacji na swój temat.
    // Zaimplementowano z użyciem operatorów lambda
    public bool IsSimple => SubArguments.Count == 0;
    public bool IsSimpleSwitch =>
                !string.IsNullOrEmpty(Switch) && SubArguments.Count == 0;
```


Kup książkę

```
    public bool IsCompoundSwitch =>
                !string.IsNullOrEmpty(Switch) && SubArguments.Count == 1;
    public bool IsComplexSwitch =>
                !string.IsNullOrEmpty(Switch) && SubArguments.Count > 0;
}
public sealed class ArgumentDefinition
{
     public string ArgumentSwitch { get; }
     public string Syntax { get; }
     public string Description { get; }
     public Func<Argument, bool> Verifier { get; }
     public ArgumentDefinition(string argumentSwitch,
                               string syntax,
                               string description,
                               Func<Argument, bool> verifier)
    {
        ArgumentSwitch = argumentSwitch.ToUpper();
        Syntax = syntax;
        Description = description;
        Verifier = verifier;
    }
    public bool Verify(Argument arg) => Verifier(arg);
}
public sealed class ArgumentSemanticAnalyzer
{
    private List<ArgumentDefinition> argumentDefinitions =
        new List<ArgumentDefinition>();
    private Dictionary<string, Action<Argument>> argumentActions =
        new Dictionary<string, Action<Argument>>();
    public ReadOnlyCollection<Argument> UnrecognizedArguments
                                            { get; private set; }
    public ReadOnlyCollection<Argument> MalformedArguments { get; private set; }
    public ReadOnlyCollection<Argument> RepeatedArguments { get; private set; }
    public ReadOnlyCollection<ArgumentDefinition> ArgumentDefinitions =>
           new ReadOnlyCollection<ArgumentDefinition>(argumentDefinitions);
    public IEnumerable<string> DefinedSwitches =>
                           from argumentDefinition in argumentDefinitions
                           select argumentDefinition.ArgumentSwitch;
    public void AddArgumentVerifier(ArgumentDefinition verifier) =>
        argumentDefinitions.Add(verifier);
    public void RemoveArgumentVerifier(ArgumentDefinition verifier)
    {
```


Kup ksi k

```
    var verifiersToRemove = from v in argumentDefinitions
                            where v.ArgumentSwitch ==
                               verifier.ArgumentSwitch
                            select v;
    foreach (var v in verifiersToRemove)
        argumentDefinitions.Remove(v);
}
public void AddArgumentAction(string argumentSwitch, Action<Argument>
                                action) =>
    argumentActions.Add(argumentSwitch, action);
public void RemoveArgumentAction(string argumentSwitch)
{
    if (argumentActions.Keys.Contains(argumentSwitch))
        argumentActions.Remove(argumentSwitch);
}
public bool VerifyArguments(IEnumerable<Argument> arguments)
{
   // Brak parametru do zweryfikowania, błąd.
   if (!argumentDefinitions.Any())
        return false;
    // Sprawdź, czy któryś z argumentów nie został zdefiniowany
    this.UnrecognizedArguments =
             ( from argument in arguments
               where !DefinedSwitches.Contains(argument.Switch.ToUpper())
               select argument).ToList().AsReadOnly();
    if (this.UnrecognizedArguments.Any())
        return false;
    // Sprawdź wszystkie argumenty, których przełącznik pasuje
    // do znanego przełącznika, przy uwzględnieniu, że nasz
    // predykat poprawności jest fałszywy.
    this.MalformedArguments = ( from argument in arguments
                                join argumentDefinition in
                                    argumentDefinitions
                                on argument.Switch.ToUpper() equals
                                    argumentDefinition.ArgumentSwitch
                                where !argumentDefinition.Verify(argument)
                                select argument).ToList().AsReadOnly();
    if (this.MalformedArguments.Any())
        return false;
    // Posortuj argumenty w "grupy" względem ich przełączników,
    // zlicz wszystkie grupy, a następnie wybierz te,
    // które zawierają więcej niż jeden element.
```


Kup ksi k

```
    // Uzyskamy wówczas listę elementów tylko do odczytu.
    this.RepeatedArguments =
            (from argumentGroup in
                from argument in arguments
                where !argument.IsSimple
                group argument by argument.Switch.ToUpper()
            where argumentGroup.Count() > 1
            select argumentGroup).SelectMany(ag =>
                      ag).ToList().AsReadOnly();
    if (this.RepeatedArguments.Any())
    return false;
    return true;
}
public void EvaluateArguments(IEnumerable<Argument> arguments)
{
    // Teraz już tylko aplikujemy każdą z akcji:
    foreach (Argument argument in arguments)
        argumentActions[argument.Switch.ToUpper()](argument);
}
public string InvalidArgumentsDisplay()
{
    StringBuilder builder = new StringBuilder();
    builder.AppendFormat($"Invalid arguments: {Environment.NewLine}");
    // Dodaj nierozpoznane argumenty
    FormatInvalidArguments(builder, this.UnrecognizedArguments,
        "Unrecognized argument: {0}{1}");
    // Dodaj niepoprawne argumenty
    FormatInvalidArguments(builder, this.MalformedArguments,
        "Malformed argument: {0}{1}");
   // Powtórzone argumenty chcemy pogrupować w celu wyświetlenia,
   // więc pogrupujemy je według przełączników, a następnie dodamy je
   // do tworzonego łańcucha znaków.
   var argumentGroups = from argument in this.RepeatedArguments
                        group argument by argument.Switch.ToUpper() into ag
                        select new { Switch = ag.Key, Instances = ag};
   foreach (var argumentGroup in argumentGroups)
   {
       builder.AppendFormat($"Repeated argument:
                              {argumentGroup.Switch}{Environment.NewLine}");
       FormatInvalidArguments(builder, argumentGroup.Instances.ToList(),
           "\t{0}{1}");
   }
```


Kup książkę

```
        return builder.ToString();
    }
    private void FormatInvalidArguments(StringBuilder builder,
       IEnumerable<Argument> invalidArguments, string errorFormat)
    {
       if (invalidArguments != null)
       {
           foreach (Argument argument in invalidArguments)
           {
               builder.AppendFormat(errorFormat,
                   argument.Original, Environment.NewLine);
           }
       }
    }
}
```

Poniższy przykład pokazuje nam, w jaki sposób możemy korzystać z tych klas do przetwarzania wiersza poleceń:

```
public static void Main(string[] argumentStrings)
{
    var arguments = (from argument in argumentStrings
        select new Argument(argument)).ToArray();
    Console.Write("Command line: ");
    foreach (Argument a in arguments)
    {
        Console.Write($"{a.Original} ");
    }
    Console.WriteLine("");
    ArgumentSemanticAnalyzer analyzer = new ArgumentSemanticAnalyzer();
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("output",
            "/output:[path to output]",
            "Specifies the location of the output file.",
            x => x.IsCompoundSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("trialMode",
            "/trialmode",
            "If this is specified it places the product into trial mode",
            x => x.IsSimpleSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("DEBUGOUTPUT",
            "/debugoutput:[value1];[value2];[value3]",
```


Kup ksi k

```
            "A listing of the files the debug output " +
            "information will be written to",
            x => x.IsComplexSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("",
            "[literal value]",
            "A literal value",
            x => x.IsSimple));
    if (!analyzer.VerifyArguments(arguments))
    {
        string invalidArguments = analyzer.InvalidArgumentsDisplay();
        Console.WriteLine(invalidArguments);
        ShowUsage(analyzer);
        return;
    }
    // Ustaw obiekty przechowujące wyniki parsowania
    string output = string.Empty;
    bool trialmode = false;
    IEnumerable<string> debugOutput = null;
    List<string> literals = new List<string>();
    // Dla każdego parsowanego argumentu chcemy zastosować akcję,
    // więc dodamy je do analizatora.
    analyzer.AddArgumentAction("OUTPUT", x => { output = x.SubArguments[0]; });
    analyzer.AddArgumentAction("TRIALMODE", x => { trialmode = true; });
    analyzer.AddArgumentAction("DEBUGOUTPUT", x =>
                                          { debugOutput = x.SubArguments;
    });
    analyzer.AddArgumentAction("", x=>{literals.Add(x.Original);});
    // Sprawdź argumenty i wykonaj akcje
    analyzer.EvaluateArguments(arguments);
    // Wyświetl wyniki
    Console.WriteLine("");
    Console.WriteLine($"OUTPUT: {output}");
    Console.WriteLine($"TRIALMODE: {trialmode}");
    if (debugOutput != null)
    {
        foreach (string item in debugOutput)
        {
             Console.WriteLine($"DEBUGOUTPUT: {item}");
        }
    }
    foreach (string literal in literals)
```


Kup ksi k

```
        {
            Console.WriteLine($"LITERAL: {literal}");
        }
    }
    public static void ShowUsage(ArgumentSemanticAnalyzer analyzer)
    {
        Console.WriteLine("Program.exe allows the following arguments:");
        foreach (ArgumentDefinition definition in analyzer.ArgumentDefinitions)
        {
            Console.WriteLine($"\t{definition.ArgumentSwitch}:
                              ({definition.Description}){Environment.NewLine}
                              \tSyntax: {definition.Syntax}");
        }
    }
```

Omówienie

Zanim będziemy mogli sparsować parametry wiersza poleceń, musimy uzgodnić jakiś wspólny format. Zaproponowany w tym przepisie format bazuje na formacie wiersza poleceń dla kompilatora języka Visual C# .NET. Format ten zdefiniowany jest następująco:

- Wszystkie argumenty wiersza poleceń rozdzielone są co najmnej jednym białym znakiem.
- Każdy z argumentów może rozpoczynać się od znaku - lub /, lecz nie obu na raz. Jeśli argument nie spełnia tego formatu, wówczas uznaje się go za literał, będący przykładowo nazwą pliku.
- Każdy argument rozpoczynający się od znaku - lub / może zostać podzielony na przełącznik, po którym następuje dwukropek, a po nim jeden lub więcej argumentów rozdzielonych między sobą znakiem ; (średnik). Parametr wiersza pleceń `-sw:arg1;arg2;arg3` dzieli się na przełącznik (`sw`) oraz trzy argumenty (`arg1`, `arg2` oraz `arg3`). Zwróćmy uwagę, że w pełnym argumencie nie powinien występować żaden odstęp. W przeciwnym przypadku parser uzna go za dwa lub więcej argumentów.
- Łańcuchy znaków ograniczone cudzysłowem, np. "`c:\test\file.log`", zostaną cudzysłowu pozbawione. Procesu tego dokona funkcja systemu operacyjnego, która interpretuje argumenty przysłane do naszej aplikacji.
- Łańcuchy znaków ograniczone znakiem apostrofu nie podlegają temu zabiegowi.
- Aby zachować znaki codzysłowu, należy je poprzedzić znakiem sekwencji ucieczki \ (odwrotnym ukośnikiem).
- Znak \ obsługiwany jest jako znak sekwencji ucieczki tylko wtedy, gdy poprzedza znak cudzysłowu. W takim wypadku wyświetlany jest wyłącznie znak cudzysłowu.

Kup ksi k

- Znak `^` wykorzystywany jest przez parser wiersza poleceń *środowiska uruchomieniowego* jako znak specjalny.

Na szczęście większość z tych rzeczy obsługiwanych jest przez parser wiersza poleceń środowiska uruchomieniowego, zanim jeszcze nasza aplikacja otrzyma indywidualne i przeparsowane argumenty.

Parser wiersza poleceń środowiska uruchomieniowego przekazuje do punktu startowego naszej aplikacji kolekcję `string[]` zawierającą każdy z przeparsowanych argumentów. Punkt ten może przyjmować jedną z poniższych form:

```
public static void Main()
public static int Main()
public static void Main(string[] args)
public static int Main(string[] args)
```

Pierwsze dwie deklaracje nie przyjmują żadnych argumentów, natomiast dwie ostatnie przyjmują tablicę przeparsowanych argumentów wiersza poleceń. Zwróćmy uwagę, że właściwość statyczna `Environment.CommandLine` również zwraca łańcuch zawierający pełną listę argumentów wiersza poleceń, zaś metoda statyczna `Environment.GetCommandLineArgs` zwraca tablicę łańcuchów zawierających przeparsowane argumenty wiersza poleceń.

Trzy klasy zaprezentowane w Rozwiązaniu niniejszego przepisu stanowią poszczególne fazy pracy z argumentami wiersza poleceń:

`Argument`
: Enkapsuluje pojedynczy argument wiersza poleceń i odpowiada za jego przeparsowanie.

`ArgumentDefinition`
: Definiuje argument, który będzie prawidłowy dla bieżącego wiersza poleceń.

`ArgumentSemanticAnalyzer`
: Dokonuje weryfikacji i pozyskuje argumenty w oparciu o przygotowane definicje argumentów `ArgumentDefinition`.

Przekazanie do tej aplikacji poniższych argumentów wiersza poleceń:

```
MyApp c:\input\infile.txt -output:d:\outfile.txt -trialmode
```

wygeneruje w procesie parsowania następujące przełączniki i argumenty:

```
Command line: c:\input\infile.txt -output:d:\outfile.txt -trialmode
OUTPUT: d:\outfile.txt
TRIALMODE: True
LITERAL: c:\input\infile.txt
```


Kup ksi k