Gaurav Aroraa

Język C# w 7 dni

Solidne podstawy programowania obiektowego

Packt>

Tytuł oryginału: Learn C# in 7 days

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-4356-6

Copyright © Packt Publishing 2017. First published in the English language
under the title 'Learn C# in 7 days – (9781787287044)'

Polish edition copyright © 2018 by Grupa Helion
All rights reserved.

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

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

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

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

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/jezc7d.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jezc7d
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorach 9

O recenzencie 10

Wstęp 13

Rozdział 1. Dzień 1. — Przegląd platformy .NET 15

- **Czym jest programowanie?** 15
- **Czym jest .NET?** 17
- **Czym jest .NET Core?** 18
  - Cechy .NET Core 18
  - Z czego składa się .NET Core? 19
  - Czym jest .NET Standard? 19
  - Dostępne środowiska programistyczne i edytory do pisania kodu w C# 20
  - Przygotowywanie środowiska 21
- **Ćwiczenia praktyczne** 23
- **Podsumowanie dnia 1.** 23

Rozdział 2. Dzień 2. — Zaczynamy poznawać C# 25

- **Przedstawienie języka C#** 26
- **Historia języka C#** 27
- **Przedstawienie typowego programu C#** 27
  - 1 (System) 29
  - 2 (Dzien02) 30
  - 3 (Program) 31
  - 4 (Main) 31
  - 5 (Dzien02) 31
  - 6 (Dzien02) 32
  - 7 (Zależności) 33
  - 8 (Program.cs) 34
  - Dokładniejsze poznanie aplikacji z użyciem Visual Studio 34
  - Analiza kodu 36

Kup ksi k Pole ksi k

Przegląd słów kluczowych, typów i operatorów języka C# 38
Identyfikatory 39
Kontekstowe słowa kluczowe 51
Typy 53
Operatory 57
Przegląd informacji o konwersji typów 64
Konwersja niejawna 65
Konwersja jawna 65
Prezentacja instrukcji 66
Instrukcja deklaracji 67
Instrukcja wyrażenia 68
Instrukcje wyboru 68
Instrukcja obsługi wyjątków 76
Operacje na tablicach i łańcuchach znaków 76
Tablice 76
Typy tablic 78
Łańcuchy znaków 81
Struktury a klasy 82
Ćwiczenia praktyczne 85
Podsumowanie dnia 2. 87

Rozdział 3. Dzień 3. — Co nowego w C#? 89

Krotki i dekonstrukcja 89
Krotki 90
Dekonstrukcja 94
Krotki — ważne zagadnienia do zapamiętania 95
Dopasowywanie wzorców 96
Wyrażenie is 97
Instrukcja switch 99
Funkcje lokalne 102
Usprawnienia literałów 103
Literały dwójkowe 103
Separatory cyfr 104
Asynchroniczna metoda Main 105
Ograniczenia związane z nowymi sygnaturami 106
Wyrażenia domyślne 106
Zmienne składowe 107
Wnioskowanie nazw krotek 108
Inne planowane możliwości 110
Dopasowywanie wzorców z typami ogólnymi 110
Zestawy referencyjne 111
Ćwiczenia praktyczne 111
Podsumowanie dnia 3. 112

Kup ksi k Pole ksi k

Rozdział 4. Dzień 4. — Prezentacja składowych klas w języku C# 113

Modyfikatory 115
Modyfikatory oraz poziomy dostępu 115
Reguły związane ze stosowaniem modyfikatorów dostępu 122
Modyfikator abstract 123
Modyfikator async 127
Modyfikator const 127
Modyfikator event 128
Modyfikator extern 128
new 128
Modyfikator override 129
Modyfikator partial 129
Modyfikator readonly 130
Modyfikator sealed 131
Modyfikator static 133
Modyfikator unsafe 134
Modyfikator virtual 134
Metody 135
Jak należy używać metod? 135
Właściwości 138
Rodzaje właściwości 139
Indeksery 143
Operacje wejścia-wyjścia na plikach 144
Klasa FileStream 144
Obsługa wyjątków 145
Blok try 147
Blok catch 147
Blok finally 147
Wyjątki definiowane przez użytkownika 149
Wyrażenia regularne oraz ich znaczenie 151
Znaczenie wyrażeń regularnych 151
Ćwiczenia praktyczne 154
Podsumowanie dnia 4. 155

Rozdział 5. Dzień 5. — Przegląd informacji o odzwierciedlaniu i kolekcjach 157

Czym jest odzwierciedlanie? 157
Praktyczne stosowanie odzwierciedlania 161
Przegląd informacji o delegacjach i zdarzeniach 166
Delegacje 166
Zdarzenia 169
Kolekcje i typy nieogólne 170
ArrayList 171
Hashtable 173
SortedList 176

Kup ksi k Pole ksi k

Stack 179
Queue 181
BitArray 185
Ćwiczenia praktyczne 185
Podsumowanie dnia 5. 186

Rozdział 6. Dzień 6. — Dokładniejszy przegląd zagadnień zaawansowanych 187

Stosowanie kolekcji i typów ogólnych 188
Prezentacja klas kolekcji oraz sposobów ich stosowania 188
Wydajność — BitArray a tablica typu bool 189
Prezentacja typów ogólnych oraz ich zastosowania 191
Upiększanie kodu przy użyciu atrybutów 203
Typy atrybutów 203
Tworzenie i stosowanie niestandardowych atrybutów 207
Stosowanie dyrektyw preprocesora 209
Ważne zagadnienia 209
Prezentacja LINQ 214
Pisanie niebezpiecznego kodu 215
Pisanie asynchronicznego kodu 217
Ćwiczenia praktyczne 218
Podsumowanie dnia 6. 219

Rozdział 7. Dzień 7. — Podstawy programowania obiektowego w C# 221

Wprowadzenie do programowania obiektowego 222
Związki pomiędzy obiektami 223
Dziedziczenie 224
Wyjaśnienie dziedziczenia 224
Stosowanie dziedziczenia 233
Abstrakcja 238
Stosowanie abstrakcji 238
Hermetyzacja 242
Czym są modyfikatory dostępu w języku C#? 242
Stosowanie hermetyzacji 243
Polimorfizm 244
Typy polimorfizmu 245
Stosowanie polimorfizmu 249
Ćwiczenia praktyczne 251
Podsumowanie dnia 7. 252
Co dalej? 252

Skorowidz 255

Kup ksi k Pole ksi k

Dzień 6. — Dokładniejszy przegląd zagadnień zaawansowanych

Zaczynamy właśnie szósty dzień naszego siedmiodniowego kursu języka C#. W poprzednim rozdziale zaprezentowano ważne zagadnienia języka C# — odzwierciedlanie, kolekcje, delegacje oraz zdarzenia. Zostały one opisane i przedstawione na przykładach. W tym rozdziale skoncentrujemy się na kolejnych rodzajach kolekcji, zaimplementowanych jako typy ogólne; oprócz nich zaprezentujemy także dyrektywy preprocesora oraz atrybuty.

W tym rozdziale zostaną opisane następujące zagadnienia:

- stosowanie kolekcji i typów ogólnych;
- upiększanie kodu przy użyciu atrybutów;
- stosowanie dyrektyw preprocesora;
- prezentacja LINQ;
- pisanie niebezpiecznego kodu;
- pisanie kodu asynchronicznego.

Kup ksi k Pole ksi k

Stosowanie kolekcji i typów ogólnych

Kolekcje nie są dla nas czymś nowym, gdyż w poprzednim rozdziale poznaliśmy już kolekcje zaimplementowane jako typy nieogólne. Jednak oprócz nich istnieją także kolekcje będące typami ogólnymi. Właśnie nimi zajmiemy się w tym podrozdziale, gdzie zostaną dokładniej opisane i zaprezentowane na przykładach.

Prezentacja klas kolekcji oraz sposobów ich stosowania

Zgodnie z informacjami podanymi w piątym dniu kursu, kolekcje to wyspecjalizowane klasy służące do operacji na danych (ich przechowywania oraz pobierania). W poprzednim rozdziale przedstawionych już zostało kilka rodzajów kolekcji, a konkretnie: stos, kolejka, lista, tablica mieszająca, wchodzących w skład przestrzeni nazw `System.Collections.NonGeneric`. Zamieszczona poniżej tabela 6.1 zawiera zestawienie kilku nieogólnych klas kolekcji wraz z opisem ich stosowania oraz przeznaczenia.

Tabela 6.1. Klasy kolekcji

Klasa	Opis	Zastosowanie
`ArrayList`	Ta kolekcja zawiera uporządkowaną listę obiektów, do których można się odwoływać przy użyciu indeksów. Kolekcję tego typu można zadeklarować w następujący sposób: `ArrayList arrayList = new ArrayList();`	W drugim dniu kursu zostały przedstawione tablice oraz sposoby odwoływania się do ich poszczególnych elementów. W przypadku kolekcji `ArrayList` można skorzystać z różnych metod dodawania do kolekcji i usuwania z niej elementów, które zostały dokładniej opisane w piątym dniu kursu.
`Hashtable`	`Hashtable` to reprezentacja kolekcji zawierającej pary klucz – wartość, zorganizowane na podstawie klucza pełniącego rolę kodu mieszającego. Stosowanie tej kolekcji zaleca się w przypadkach, kiedy konieczne jest odwoływanie się do danych na podstawie klucza. Poniżej przedstawiony został sposób deklarowania kolekcji tego typu: `Hashtable hashtable = new Hashtable();`	Kolekcja typu `Hashtable` jest najbardziej użyteczna w sytuacjach, kiedy trzeba odwoływać się do danych przy użyciu klucza. W takich przypadkach dysponujemy kluczem i musimy stosować go do pobierania elementów kolekcji.
`SortedList`	`SortedList` to reprezentacja kolekcji par klucz – wartość, zorganizowanej na podstawie kluczy, której zawartość jest posortowana według tych kluczy. Kolekcja ta jest połączeniem kolekcji `ArrayList` oraz `Hashtable`; oznacza to, że do elementów tej kolekcji można się odwoływać przy użyciu kluczy lub indeksów.	Jak już podano, kolekcja `SortedList` jest połączeniem tablicy oraz tablicy mieszającej. Do jej elementów można się odwoływać przy użyciu kluczy lub indeksów. W przypadku odwoływania się przy użyciu indeksów działa ona jak `ArrayList`, natomiast w razie stosowania kluczy — jak `Hashtable`.

Kup ksi k   Pole ksi k

Tabela 6.1. Klasy kolekcji (ciąg dalszy)

Klasa	Opis	Zastosowanie
cd.	Oto sposób deklarowania kolekcji tego typu: `SortedList sortedList = new SortedList();`	Jej najważniejszą cechą jest to, że zawartość takiej kolekcji pozostaje posortowana na podstawie kluczy.
`Stack`	Stos to kolekcja obiektów działająca zgodnie z zasadą **ostatni dodany, pierwszy obsłużony** (ang. *LIFO*). Stosy obsługują dwie podstawowe operacje: umieszczenie na stosie (`Push`) oraz zdjęcie ze stosu (`Pop`). Pierwsza z tych operacji reprezentuje dodanie elementu na wierzchołku stosu, natomiast druga — usunięcie elementu z wierzchołka stosu. Oprócz tego można także odczytać element z wierzchołka stosu bez jego usuwania (`Peek`). Stos można zadeklarować w następujący sposób: `Stack stackList = new Stack();`	Kolekcje tego typu należy stosować w sytuacjach, kiedy ostatni element dodany do kolekcji ma być z niej usunięty jako pierwszy.
`Queue`	Kolejka reprezentuje kolekcję obiektów działającą zgodnie z zasadą **pierwszy dodany, pierwszy obsłużony** (ang. *FIFO*). Kolejki obsługują dwie podstawowe operacje: dodanie do kolejki (`Enqueue`) oraz usunięcie z niej (`Dequeue`). Poniżej przedstawiono sposób deklarowania kolekcji tego typu: `Queue queue = new Queue();`	Kolekcje tego typu należy stosować w sytuacjach, kiedy element dodany do kolekcji jako pierwszy ma być z niej usunięty jako pierwszy.
`BitArray`	`BitArray` to kolekcja reprezentująca tablicę zarządzającą bitami. Poszczególne bity są reprezentowane jako wartości logiczne. Logiczna prawda oznacza, że bit ma wartość `1`, a logiczny fałsz — że bit ma wartość `0`. Oto sposób deklarowania kolekcji tego typu: `BitArray bitArray = new BitArray(8);`	Ta kolekcja przydaje się w sytuacjach, kiedy konieczne jest przechowywanie bitów.

W tabeli 6.1 przedstawione zostały wyłącznie nieogólne klasy kolekcji. Dostępne są także klasy kolekcji zaimplementowane jako klasy ogólne; zostały one umieszczone w przestrzeni nazw `System.Collections`. Kolekcje te zostaną dokładniej przedstawione w dalszej części rozdziału.

Wydajność — BitArray a tablica typu bool

W zamieszczonej wcześniej tabeli 6.1 opisana została klasa `BitArray` stanowiąca tablicę służącą do zarządzania wartościami logicznymi (`true`, `false`) reprezentującymi bity (wartości `0` lub `1`). Jednak wewnętrzny sposób działania tej klasy opiera się na wykorzystaniu bajtów — grup ośmiu bitów — co wymaga wykonania różnych operacji logicznych i zużycia większej liczby cykli procesora. Z drugiej strony zwyczajna tablica wartości logicznych (`bool[]`) przechowuje

Kup ksi k Pole ksi k

elementy jako bajty, dzięki czemu zajmuje więcej pamięci, lecz jednocześnie zużywa mniej cykli procesora. Należy zatem uznać, że w stosunku do tablicy wartości logicznych (`bool[]`) klasa `BitArray` jest zoptymalizowana pod kątem zużycia pamięci.

Przeanalizujmy poniższy test, który pozwoli nam określić wydajność działania kolekcji `BitArray`:

```
private static long BitArrayTest(int max)
{
    Stopwatch stopwatch = Stopwatch.StartNew();
    var bitarray = new BitArray(max);
    for (int index = 0; index < bitarray.Length; index++)
    {
        bitarray[index] = !bitarray[index];
        WriteLine($"'bitarray[{index}]' = {bitarray[index]}");
    }
    stopwatch.Stop();
    return stopwatch.ElapsedMilliseconds;
}
```

Powyższy kod stanowi bardzo prosty test wydajności działania klasy `BitArray`, w którym w pętli `for` ustawiamy liczbę bitów odpowiadających zakresowi typu `int` — `int.MaxValue`.

Kolejny fragment kodu przedstawia analogiczny test sprawdzający wydajność działania tablicy `bool[]`; także w tym przypadku tekst operuje na tablicy wielkości `int.MaxValue`.

```
private static long BoolArrayTest(int max)
{
    Stopwatch stopwatch = Stopwatch.StartNew();
    var boolArray = new bool[max];
    for (int index = 0; index < boolArray.Length; index++)
    {
        boolArray[index] = !boolArray[index];
        WriteLine($"'boolArray[{index}]' = {boolArray[index]}");
    }
    stopwatch.Stop();
    return stopwatch.ElapsedMilliseconds;
}
```

Kolejny fragment kodu pokazuje, w jaki sposób można wykonać oba powyższe testy:

```
private static void BitArrayBoolArrayPerformance()
{
    // to prosty test;
    // nie sprawdza wydajności działania operacji takich jak przesunięcia bitowe
    WriteLine("Porównanie wydajności BitArray oraz bool[].\n");
    WriteLine($"Całkowita liczba elementów: {int.MaxValue}");
    PressAnyKey();
    WriteLine("Testowanie kolekcji BitArray:");
    var bitArrayTestResult = BitArrayTest(int.MaxValue);
    WriteLine("Test zakończony.");
```


Kup ksi k Pole ksi k

```
        WriteLine("Czas: {0:hh\\:mm\\:ss} ", TimeSpan.FromMilliseconds
        ↪(bitArrayTestResult));

        WriteLine("\nTestowanie tablicy bool[]:");
        WriteLine($"Całkowita liczba elementów: {int.MaxValue}");
        PressAnyKey();
        var boolArrayTestResult = BoolArrayTest(int.MaxValue);
        WriteLine("Test zakończony.");
a        WriteLine("Czas: {0:hh\\:mm\\:ss} ", TimeSpan.FromMilliseconds
a        ↪(boolArrayTestResult));
    }
```

Na moim komputerze wykonanie metody `BitArrayTest` zajęło 6 sekund, a wykonanie metody `BoolArrayTest` — 15 sekund.

Uzyskane wyniki wskazują, że klasa `BitArray` nie tylko zajmuje mniej miejsca w pamięci niż tablice `bool[]`, lecz także zapewnia większą wydajność działania.

Prezentacja typów ogólnych oraz ich zastosowania

Najprościej rzecz ujmując, przy korzystaniu z typów ogólnych można napisać kod operujący na innym typie danych niż ten, z myślą o którym był implementowany. Powiedzmy, że jeśli klasa ogólna była pisana z myślą o tym, że będzie operować na strukturze, to równie dobrze będzie ona operować na liczbach typu `int`, łańcuchach znaków czy też dowolnych strukturach. Takie klasy są także nazywane klasami ogólnymi. Ich działanie staje się jeszcze bardziej magiczne, kiedy podczas tworzenia instancji klasy ogólnej można określić typ danych, na jakich ma ona operować. Przeanalizujmy poniższy przykład, przedstawiający definicję zmiennej typu ogólnego wraz z określonym typem, na którym instancja ta ma działać:

```
IList<Person> persons = new List<Person>();
```

W powyższym przykładzie zadeklarowaliśmy zmienną `persons`, która jest kolekcją ogólnego typu `List`. Kolekcja ta podlega mechanizmom ścisłej kontroli typów i ma operować na obiektach klasy `Person`. Poniższy fragment kodu pokazuje, w jaki sposób można określić zawartość kolekcji `persons`:

```
private static IEnumerable<Person> CreatePersonList()
{
    IList<Person> persons = new List<Person>
    {
        new Person
        {
            Id = 1,
            FirstName = "Jan",
            LastName = "Kowalski",
            Age = 31
        },
```


Kup ksi k   Pole ksi k

```
            new Person
            {
                FirstName = "Joanna",
                LastName = "Tkaczyk",
                Age = 25
            },
            new Person
            {
                Id = 2,
                FirstName = "Szymon",
                LastName = "Wolański",
                Age = 40
            },
            new Person
            {
                Id = 3,
                FirstName = "Roman",
                LastName = "Piotrowski",
                Age = 43
            }
        };

        return persons;
    }
```

Powyższy fragment pokazuje sposób inicjalizacji kolekcji typu `Person`. Kolejny przykład pokazuje zapisanie elementów w kolekcji oraz wyświetlenie jej zawartości w pętli:

```
    private static void PersonList()
    {
        WriteLine("Lista osób:");
        foreach (var person in Person.GetPersonList())
        {
            WriteLine($"Personalia:{person.FirstName} {person.LastName}");
            WriteLine($"Wiek:{person.Age}");
        }
        ReadLine();
    }
```

Wyniki wykonania tej metody zostały przedstawione na rysunku 6.1.

Można także stworzyć listę ogólną korzystającą z mechanizmu ścisłej kontroli typów, która będzie operować na obiektach innych niż `Person`. Poniżej pokazano, jak to zrobić:

```
    private IEnumerable<T> CreateGenericList<T>()
    {
        IList<T> persons = new List<T>();
        // inny kod

        return persons;
    }
```


Kup ksi k Pole ksi k

```
C:\WINDOWS\system32\cmd.exe
Lista osób:
Personalia:Jan Kowalski
Wiek:31
Personalia:Joanna Tkaczyk
Wiek:25
Personalia:Szymon Wolański
Wiek:40
Personalia:Roman Piotrowski
Wiek:43
```

Rysunek 6.1. Operacje na kolekcji typu ogólnego

W tym przykładzie `T` może być klasą `Person` lub dowolnym innym typem.

Kolekcje i typy ogólne

W drugim dniu kursu wspominaliśmy o tablicach — zbiorach danych, które mają ściśle określoną pojemność. Można ich używać do przechowywania obiektów konkretnego typu i podlegających mechanizmom ścisłej kontroli typów. Co jednak zrobić w sytuacjach, kiedy chcielibyśmy zorganizować dane przy wykorzystaniu innej struktury danych, takiej jak: kolejka, lista, stos itd.? W takim przypadku wystarczy zastosować odpowiednią kolekcję (z przestrzeni nazw `System.Collections`).

`System.Collections` (*https://www.nuget.org/packages/System.Collections/*) to pakiet NuGet zawierający typy ogólne, takie jak te przedstawione w tabeli 6.2.

Tabela 6.2. Wybrane typy ogólne z przestrzeni nazw System.Collections

Ogólny typ kolekcji	Opis
`System.Collections.Generic.List<T>`	Ogólna lista podlegająca ścisłej kontroli typów.
`System.Collections.Generic.Dictionary<TKey, TValue>`	Ogólny słownik par klucz – wartość podlegający ścisłej kontroli typów.
`System.Collections.Generic.Queue<T>`	Ogólna klasa implementująca kolejkę.
`System.Collections.Generic.Stack<T>`	Ogólna klasa implementująca stos.
`System.Collections.Generic.HashSet<T>`	Ogólna klasa `HashSet`.
`System.Collections.Generic.LinkedList<T>`	Ogólna klasa reprezentująca listę połączoną.
`System.Collections.Generic.SortedDictionary<TKey, TValue>`	Ogólna klasa słownika przechowująca pary klucz – wartość, sortowane na podstawie klucza.

Powyższa tabela 6.2 zawiera jedynie niewielki fragment wszystkich klas dostępnych w przestrzeni nazw `System.Collections.Generic`. W dalszej części rozdziału klasy te zostaną dokładniej opisane i przedstawione na przykładach.

> Kompletną listę klas, struktur oraz interfejsów dostępnych w przestrzeni nazw `System.Collections.Generic` można znaleźć w jej oficjalnej dokumentacji, dostępnej na stronie *https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic?view=netcore-2.0*.

Dlaczego warto używać typów ogólnych?

Uniwersalnym typem elementów kolekcji nieogólnych jest `Object` (*https://docs.microsoft.com/en-us/dotnet/api/system.object?view=netframework-4.7.1*). Nie jest to rozwiązanie bezpieczne, gdyż nie pozwala na kontrolę typów podczas kompilacji kodu. Załóżmy, że używamy nieogólnej kolekcji typu `ArrayList`; takiej jak ta przedstawiona na poniższym przykładzie:

```
ArrayList authorArrayList = new ArrayList {"Jan Kowalski", "43"};
foreach (string author in authorArrayList)
{
    WriteLine($"Personalia:{author}");
}
```

Jak widać, mamy tu do czynienia z kolekcją `ArrayList` zawierającą łańcuchy znaków. Także wiek osoby jest zapisywany w formie łańcucha znaków, choć powinna to być wartość typu `int`. Zmodyfikujmy zatem naszą listę tak, by wiek faktycznie był przechowywany jako liczba całkowita:

```
ArrayList editorArrayList = new ArrayList { "Joanna Tkaczyk", 25 };
foreach (int editor in editorArrayList)
    WriteLine($"{editor}");
```

Taki kod można prawidłowo skompilować, jednak podczas próby jego wykonania zostanie zgłoszony wyjątek związany z błędem rzutowania (patrz rysunek 6.2). Oznacza to, że w przypadku stosowania klasy `ArrayList` nie jest przeprowadzana kontrola typów podczas kompilacji kodu:

```
private static void NonGenericListWithAuthorNameAndAge()
{
    // nie jest zgłaszany żaden wyjątek, ani podczas kompilacji, ani podczas wykonywania kodu
    ArrayList authorArrayList = new ArrayList { "Jan Kowalski", "43" };
    foreach (string author in authorArrayList)
        WriteLine($"{author}");

    // brak wyjątku podczas kompilacji - jest natomiast zgłaszany podczas wykonywania kodu
    ArrayList editorArrayList = new ArrayList { "Joanna Tkaczyk", 25 }; // zgłasza wyjątek podczas wykonywania kodu
    foreach (int editor in editorArrayList)
        WriteLine($"{editor}");
}
```

Nieobsługiwany wyjątek

System.InvalidCastException: „Unable to cast object of type 'System.String' to type 'System.Int32'."

Wyświetl szczegóły | Kopiuj szczegóły

Ustawienia wyjątków

Rysunek 6.2. Wyjątek zgłaszany podczas wykonywania ostatniego przykładu

Kup ksi k Pole ksi k

Po przeanalizowaniu powyższego fragmentu kodu bez trudu można stwierdzić, dlaczego podczas jego kompilacji nie został zgłoszony żaden wyjątek: klasa `ArrayList` akceptuje bowiem dowolny typ (zarówno wartościowy, jak i referencyjny), a następnie rzutuje go do uniwersalnego typu platformy .NET, czyli `Object`. Jednak podczas wykonywania kodu kolekcja wymaga zastosowania konkretnego typu, na przykład jeśli została zdefiniowana jako kolekcja łańcuchów znaków, to należy w niej zapisywać łańcuchy znaków, a nie inne obiekty. Właśnie dlatego podczas wykonywania tego kodu został zgłoszony wyjątek.

> Czynności związane z rzutowaniem, pakowaniem oraz rozpakowywaniem wykonywane przez kolekcję `ArrayList` mają niekorzystny wpływ na wydajność działania kodu — przy czym spadek efektywności jest zależny od wielkości kolekcji `ArrayList` oraz ilości przechowywanych w niej danych.

Na podstawie powyższego przykładu można wskazać podstawowe wady tej nieogólnej wersji klasy `ArrayList`:

1. Nie zapewnia kontroli typów podczas kompilacji kodu.
2. Ma negatywny wpływ na wydajność działania w przypadku operowania na dużych zbiorach danych.
3. Wszystkie zapisywane w niej dane są rzutowane na `Object`, zatem nie można uniemożliwić dodawania do kolekcji obiektów innego typu już w trakcie kompilacji kodu. Na przykład w poprzednim przykładzie dodawaliśmy do kolekcji łańcuchy znaków i liczby całkowite.

Powyższe problemy można rozwiązać, stosując kolekcje będące typami ogólnymi, które uniemożliwiają dodawanie obiektów innego typu niż ten podany podczas ich tworzenia. Przeanalizujmy poniższy przykład:

```
List<string> authorName = new List<string> {"Jan Kowalski"};
```

Powyższa lista została zdefiniowania w taki sposób, że można w niej zapisywać wyłącznie elementy będące łańcuchami znaków. A zatem można do niej dodawać wyłącznie łańcuchy znaków. Przeanalizujmy poniższy przykład:

```
List<string> authorName = new List<string>();
authorName.Add("Jan Kowalski");
authorName.Add(43);
```

W tym przykładzie próbujemy zapisać na liście element typu `int` (Czytelnik zapewne pamięta, że w przypadku użycia kolekcji `ArrayList` było to możliwe). Jednak w tym przypadku próba użycia takiego kodu spowoduje zgłoszenie błędu rzutowania — a zatem lista ogólna zdefiniowana jako lista łańcuchów znaków uniemożliwia dodawanie innych elementów niż łańcuchy. Szczegółową informację o błędzie można wyświetlić, umieszczając wskaźnik myszy na liczbie 43, jak pokazano na rysunku 6.3.

Kup ksi k Pole ksi k

```
private static void Main(string[] args)
{
    List<string> authorName = new List<string>();
    authorName.Add("Jan Kowalski");
    authorName.Add(43);
}
```

struct System.Int32
Represents a 32-bit signed integer.

Argument „1": nie można przekonwertować z „int" na „string"

Rysunek 6.3. Błąd konwersji podczas dodawania elementu do listy ogólnej

W powyższym przykładzie rozwiązaliśmy jeden z problemów — zadeklarowaliśmy listę łańcuchów znaków, dzięki czemu mogliśmy na niej zapisywać wyłącznie łańcuchy znaków. Jednak w kontekście przykładu o autorach oznacza to, że możemy na niej zapisywać wyłącznie imię i nazwisko, lecz już nie wiek autora. Można by się zastanawiać, po co używać kolekcji typu ogólnego, skoro w ich przypadku dodawanie nowego typu danych wymaga utworzenia nowej kolekcji? Obecnie potrzebujemy tylko dwóch informacji o autorze — personaliów oraz wieku; wystarczyłoby zatem utworzyć dwie listy: jedną typu `string` na personalia oraz drugą typu `int` na wiek. Gdybyśmy jednak potrzebowali przechowywać jeszcze inne informacje, musielibyśmy w tym celu stworzyć kolejną listę. Doprowadziłoby to do konieczności utworzenia wielu list różnych typów, takich jak: `string`, `int`, `decimal` itd. Jednak zamiast tego można utworzyć własny, niestandardowy typ. Przyjrzyjmy się poniższej deklaracji listy:

```
List<Person> persons = new List<Person>();
```

Jak widać, tworzymy tu listę obiektów typu `Person`. Ta ogólna lista pozwoli nam na zapisywanie na niej wyłącznie obiektów podanego typu. A oto przykładowa definicja klasy `Person`:

```
internal class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
}
```

Przedstawiona klasa `Person` definiuje trzy właściwości: dwie typu `string` oraz jedną typu `int`. W ten sposób znaleźliśmy rozwiązanie kolejnego z wcześniejszych problemów. Dzięki stworzeniu kolekcji `List` o zawartości typu `Person` możemy utworzyć listę i zapisywać w jej elementach informacje różnych typów. Poniższy przykład przedstawia praktyczne zastosowanie takiej listy:

```
private static void PersonList()
{
    List<Person> persons = new List<Person>
    {
        new Person
        {
```


Kup ksi k Pole ksi k

```
                FirstName = "Jan",
                LastName = "Kowalski",
                Age = 43
            }
        };
        WriteLine("Lista osób:");
        foreach (var person in persons)
        {
            WriteLine($"Personalia:{person.FirstName} {person.LastName}");
            WriteLine($"Wiek:{person.Age}");
        }
    }
```

Wyniki wykonania tego przykładu przedstawiono na rysunku 6.4.

```
C:\WINDOWS\system32\cmd.exe
Lista osób:
Personalia:Jan Kowalski
Wiek:43
```

Rysunek 6.4. Wyniki zastosowania obiektów niestandardowego typu na liście ogólnej

Taka lista obiektów typu `Person` będzie działać wydajniej niż kolekcja `ArrayList`, gdyż jest listą typu ogólnego, a zatem w jej przypadku nie będą wykonywane żadne operacje rzutowania do typu `Object` — jej zawartością będą obiekty wyłącznie określonego typu.

Stosowanie ograniczeń typów ogólnych

W poprzednim podpunkcie rozdziału przedstawiona została kolekcja `List` typu `Person`, akceptująca obiekty typu określonego w definicji. W przedstawionym kodzie obiekty te zawierały jedynie dane typu `string` i `int`, jednak ogólnie nic nie stoi na przeszkodzie, by używać także innych typów danych, takich jak wartości typów `float`, `double` itd. Z drugiej strony mogą się także pojawiać sytuacje, w których będziemy chcieli ograniczyć stosowane typy danych lub umożliwić stosowanie w typie ogólnym w ogóle tylko jednego, ściśle określonego typu. Właśnie z myślą o takich sytuacjach w języku C# wprowadzono ograniczenia typów ogólnych. Przeanalizujmy następujący przykład:

```
public class GenericConstraint<T> where T:class
{
    public T ImplementIt(T value)
    {
        return value;
    }
}
```


Kup ksi k

Pole ksi k

Klasa przedstawiona na tym przykładzie jest klasą ogólną. Zastosowany w niej ogólny typ `T` jest niczym innym jak dowolnym typem referencyjnym; oznacza to, że tworząc instancję klasy `GenericConstraint`, możemy zażądać, by operowała ona na dowolnym typie referencyjnym. Jednak w takiej klasie nie będzie można używać żadnych typów wartościowych. Nasza klasa definiuje metodę `ImplementIt`, która akceptuje parametr typu `T` i zwraca wartość typu `T`.

> Więcej informacji na temat wytycznych dotyczących określania parametrów typów można znaleźć na stronie *https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-type-parameters*.

Poniższe deklaracje są prawidłowe, gdyż zastosowano w nich typy referencyjne:

```
GenericConstraint<string> genericConstraint = new GenericConstraint<string>();
Person person = genericPersonConstraint.ImplementIt(new Person());
```

Z kolei poniższa deklaracja nie jest prawidłowa, gdyż użyto w niej typu wartościowego, a definicja klasy `GenericConstraint` na to nie pozwala:

```
GenericConstraint<int> genericConstraint = new GenericConstraint<int>();
```

W drugim dniu kursu dowiedzieliśmy się, że `int` jest typem wartościowym, a nie referencyjnym. Dlatego też próba zastosowania takiej deklaracji spowoduje zgłoszenie błędu czasu kompilacji. Błąd ten jest wyświetlany w Visual Studio w sposób przedstawiony na rysunku 6.5.

```
private static void Main(string[] args)
{
    GenericConstraint<int> genericConstraint = new GenericConstraint<int>();
}
```

struct System.Int32
Represents a 32-bit signed integer.

Typ „int" musi być typem referencyjnym, aby można było używać go jako parametru „T" w typie ogólnym lub metodzie ogólnej „GenericConstraint<T>".

Rysunek 6.5. Błąd związany z naruszeniem ograniczenia typu ogólnego

A zatem dzięki zastosowaniu ograniczenia typu ogólnego uniemożliwiliśmy zastosowanie innego typu niż referencyjny.

> **Ograniczenia** można by w zasadzie uznać za zabezpieczenie naszej klasy ogólnej przed stosowaniem nieodpowiednich typów podczas tworzenia instancji naszego typu ogólnego. Jeśli klient spróbuje zastosować nieodpowiedni typ, spowoduje to zgłoszenie błędu czasu kompilacji. Do definiowania tych ograniczeń używane jest kontekstowe słowo kluczowe `where`.

W rzeczywistości możliwe jest definiowanie różnego rodzaju ograniczeń, dzięki którym można zabezpieczać klasy ogólne przed niepożądanymi sytuacjami. Przeanalizujmy kilka takich przypadków:

Kup ksi k Pole ksi k

Typy wartościowe

To pierwsze ograniczenie zostanie zdefiniowanie przy użyciu zapisu: `where T: struct`. W tym przypadku kod klienta, określając typ klasy ogólnej, powinien zastosować typ wartościowy — może to być dowolny typ wartościowy z wyjątkiem typów akceptujących wartość `null`.

Przykład

Oto przykład deklaracji klasy zawierającej ograniczenie typu wartościowego:

```
public class ValueTypeConstraint<T> where T : struct
{
    public T ImplementIt(T value)
    {
        return value;
    }
}
```

Zastosowanie

Poniższy fragment pokazuje przykład kodu klienta, w którym zastosowano klasę ogólną z ograniczeniem typu wartościowego:

```
private static void ImplementValueTypeGenericClass()
{
    const int age = 43;
    ValueTypeConstraint<int> valueTypeConstraint = new ValueTypeConstraint<int>();
    WriteLine($"Wiek:{valueTypeConstraint.ImplementIt(age)}");
}
```

Typy referencyjne

Takie ograniczenie można zdefiniować, używając zapisu o postaci: `where T:class`. W razie zastosowania takiego ograniczenia kod klienta, tworząc instancje typu ogólnego, może zastosować w niej dowolny typ referencyjny — może to być dowolna klasa, interfejs, delegacja czy też tablica.

Przykład

Poniższy przykład przedstawia deklarację klasy ogólnej z ograniczeniem typu referencyjnego:

```
public class ReferenceTypeConstraint<T> where T:class
{
    public T ImplementIt(T value)
    {
        return value;
    }
}
```


Kup ksi k Pole ksi k

Zastosowanie

Poniższy przykład przedstawia kod klienta, w którym zastosowano klasę ogólną z ograniczeniem typu referencyjnego:

```
private static void ImplementReferenceTypeGenericClass()
{
    const string thisIsAuthorName = "Jan Kowalski";
    ReferenceTypeConstraint<string> referenceTypeConstraint = new
    ↪ReferenceTypeConstraint<string>();

WriteLine($"Personalia:{referenceTypeConstraint.ImplementIt(thisIsAuthorName)}");

    ReferenceTypeConstraint<Person> referenceTypePersonConstraint = new
    ↪ReferenceTypeConstraint<Person>();

    Person person = referenceTypePersonConstraint.ImplementIt(new Person
    {
        FirstName = "Jan",
        LastName = "Kowalski",
        Age = 43
    });

    WriteLine($"Personalia:{person.FirstName}{person.LastName}");
    WriteLine($"Wiek:{person.Age}");
}
```

Domyślny konstruktor

To ograniczenie jest definiowane przy użyciu zapisu o postaci: `where T: new()`. Sprawia ono, że parametrem typu nie może być typ definiujący konstruktor domyślny. Jednocześnie narzuca ono wymóg, by typ `T` definiował publiczny konstruktor bezargumentowy. W razie stosowania z innymi ograniczeniami, ograniczenie `new()` należy podawać jako ostatnie.

Przykład

Poniższy przykład przedstawia deklarację klasy ogólnej z ograniczeniem konstruktora domyślnego:

```
public class DefaultConstructorConstraint<T> where T : new()
{
    public T ImplementIt(T value)
    {
        return value;
    }
}
```

Zastosowanie

Kolejny przykład przedstawia kod klienta, w którym zastosowano klasę ogólną z ograniczeniem konstruktora domyślnego:

Kup ksi k Pole ksi k

```
private static void ImplementDefaultConstructorGenericClass()
{
    DefaultConstructorConstraint<ClassWithDefaultConstructor>
    ↪constructorConstraint = new
DefaultConstructorConstraint<ClassWithDefaultConstructor>();
    var result = constructorConstraint.ImplementIt(new ClassWithDefaultConstructor
    ↪{ Name = "Jan Kowalski" });
    WriteLine($"Personalia:{result.Name}");
}
```

Ograniczenie klasy bazowej

Ograniczenie klasy bazowej można zdefiniować, używając następującego zapisu: `where T: <KlasaBazowa>`. To ograniczenie uniemożliwia zastosowanie jako parametru typu klasy innej niż dziedzicząca po określonej klasie bazowej.

Przykład

Oto przykład deklaracji klasy ogólnej wykorzystującej ograniczenie klasy bazowej:

```
public class BaseClassConstraint<T> where T:Person
{
    public T ImplementIt(T value)
    {
        return value;
    }
}
```

Zastosowanie

Poniższy fragment kodu przedstawia przykład zastosowania klasy ogólnej z ograniczeniem klasy bazowej:

```
private static void ImplementBaseClassConstraint()
{
    BaseClassConstraint<Author>baseClassConstraint = new
    ↪BaseClassConstraint<Author>();
    var result = baseClassConstraint.ImplementIt(new Author
    {
        FirstName = "Shivprasad",
        LastName = "Koirala",
        Age = 40
    });

    WriteLine($"Personalia:{result.FirstName} {result.LastName}");
    WriteLine($"Wiek:{result.Age}");
}
```


Kup ksi k

Pole ksi k

Ograniczenie interfejsu

Ten rodzaj ograniczenia można zdefiniować, używając następującego zapisu: `where T:<nazwa_interfejsu>`. W tym przypadku, tworząc instancję klasy ogólnej, kod klienta musi zastosować typ implementujący podany interfejs. Wskazany typ może implementować dowolnie wiele interfejsów.

Przykład

Oto przykład deklaracji klasy korzystającej z ograniczenia typu interfejsu:

```
public class InterfaceConstraint<T>:IDisposable where T : IDisposable
{
    public T ImplementIt(T value)
    {
        return value;
    }

    public void Dispose()
    {
        // czynności porządkowe
    }
}
```

Zastosowanie

Poniższy fragment kodu przedstawia kod klienta, w którym zastosowano klasę ogólną z ograniczeniem typu interfejsu:

```
private static void ImplementInterfaceConstraint()
{
    InterfaceConstraint<EntityClass> entityConstraint = new
    ↪InterfaceConstraint<EntityClass>();
    var result=entityConstraint.ImplementIt(new EntityClass
    ↪{Name = "Jan Kowalski"});
    WriteLine($"Personalia:{result.Name}");
}
```

W tym podrozdziale przedstawione zostały kolekcje oraz typy ogólne, jak również różne rodzaje ograniczeń typów ogólnych. Oprócz tego wyjaśniliśmy, dlaczego należy stosować typy ogólne.

> Więcej szczegółowych informacji na temat typów ogólnych można znaleźć w oficjalnej dokumentacji języka C# dostępnej na stronie *https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-type-parameters*.

Kup ksi k   Pole ksi k

Upiększanie kodu przy użyciu atrybutów

Atrybuty zapewniają możliwość kojarzenia informacji z kodem. Informacje te mogą mieć postać prostych komentarzy lub ostrzeżeń, lecz mogą także zawierać złożone operacje, a nawet kod. Atrybuty są definiowane przy użyciu znaczników. Rozwiązanie to pozwala także na upiększanie kodu poprzez dodawanie do niego niestandardowych atrybutów. Przeanalizujmy poniższy przykład:

```
private void PeerOperation()
{
    // inny kod
    WriteLine("Ukończono poziom 1.");
    // inny kod
}
```

W powyższej metodzie wyświetlamy komunikat tekstowy stanowiący informację dla użytkownika. Kolejny fragment kodu pokazuje, jak można dodać do metody atrybut. Oto przykład:

```
[PeerInformation("Ukończono poziom.")]
private void PeerOperation()
{
    // inny kod
}
```

W tym przykładzie dodaliśmy do metody jedynie atrybut z informacją tekstową.

Zgodnie z oficjalną dokumentacją (dostępną na stronie *https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/attributes*) atrybuty zapewniają sposób deklaratywnego kojarzenia informacji z kodem. Mogą także tworzyć elementy wielokrotnego użycia, które można dodawać do wielu różnych elementów docelowych.

Atrybutów można używać do:

- dodawania metadanych do kodu;
- dodawania do kodu komentarzy, opisów, instrukcji dla kompilatora i tak dalej.

W kolejnych punktach rozdziału zostaną szczegółowo opisane atrybuty oraz przedstawione przykłady ich zastosowania.

Typy atrybutów

W poprzednim punkcie rozdziału przedstawione zostały atrybuty, których można używać do upiększania oraz dodawania informacji do kodu. W tym punkcie zaprezentowane zostaną różne typy atrybutów.

Kup ksi k Pole ksi k

AttributeUsage

To predefiniowany atrybut frameworka .NET. Służy on do ograniczania zastosowania innych atrybutów. Oznacza to, że pozwala określić rodzaje elementów, do jakich można dodawać dany atrybut; takie elementy są także nazywane elementami docelowymi atrybutów. Poniżej przedstawiona została lista takich elementów docelowych:

- `Assembly`,
- `Class`,
- `Constructor`,
- `Delegate`,
- `Enum`,
- `Event`,
- `Field`,
- `GenericParameter`,
- `Interface`,
- `Method`,
- `Module`,
- `Parameter`,
- `Property`,
- `ReturnValue`,
- `Struct`.

Domyślnie typ elementu docelowego atrybutu może być dowolny, chyba że zostanie on jawnie określony.

Przykład

Poniższy atrybut może być dodawany wyłącznie do klas:

```
[AttributeUsage(AttributeTargets.Class)]
public class PeerInformationAttribute : Attribute
{
    public PeerInformationAttribute(string information)
    {
        WriteLine(information);
    }
}
```

W powyższym przykładzie zdefiniowaliśmy atrybut przeznaczony do stosowania w klasach. Jeśli spróbujemy dodać go do innego elementu docelowego niż klasa, spowoduje to wystąpienie błędu kompilacji, takiego jak ten przedstawiony na rysunku 6.6, informującego, że atrybut przeznaczony do użycia w deklaracjach klas został użyty w metodzie.

Kup ksi k Pole ksi k

```
[PeerInformation("Ukończono poziom.")]
private void Mc
    // jakiś kc
}
```

PeerInformationAttribute.PeerInformationAttribute(string information)

W tej deklaracji typu atrybut „PeerInformation" jest nieprawidłowy. Jest on prawidłowy tylko w deklaracjach „klasa".

Rysunek 6.6. Komunikat o dodaniu atrybutu do nieprawidłowego elementu docelowego

Obsolete

Czasami mogą się pojawić sytuacje, kiedy będziemy chcieli wygenerować ostrzeżenie w razie zastosowania określonego kodu. Właśnie do tego celu służy predefiniowany atrybut `Obsolete`.

Przykład

Przeanalizujmy poniższy kod, w którym deklaracja klasy została poprzedzona atrybutem `Obsolete`. Poniższy kod można z powodzeniem skompilować i uruchomić, nawet pomimo wyświetlenia ostrzeżenia, gdyż nie zażądaliśmy generowania błędu:

```
[Obsolete("Proszę nie używać tej klasy, zamiast niej należy użyć
↪klasy 'Person'.")]
public class Author:Person
{
    // inny kod
}
```

Rysunek 6.7 przedstawia komunikat informujący, że nie należy używać klasy `Author`, gdyż została ona uznana za przestarzałą. Pomimo komunikatu taki kod wciąż można skompilować i uruchomić (nie zażądaliśmy bowiem, by ostrzeżenie było traktowane jako błąd).

```
private static void ImplementBaseClassConstraint()
{
    BaseClassConstraint<Author> baseClassConstraint = new BaseClassConstraint<Author>();
    var result = baseClassConstraint.ImplementIt(new Author
    {
        FirstName = "Szymon",
        LastName = "Wolański",
        Age = 40
    });

    WriteLine($"Personalia: {result.FirstName} {result.LastName}");
    WriteLine($"Wiek: {result.Age}");
}
```

Author.Author()

Element „Author" jest przestarzały: „Proszę nie używać tej klasy, zamiast niej należy użyć klasy 'Person'."

Pokaż potencjalne rozwiązania (Alt+Enter or Ctrl+.)

Rysunek 6.7. Komunikat ostrzegawczy generowany dzięki zastosowaniu atrybutu Obsolete

Z kolei zastosowanie atrybutu o następującej postaci spowoduje nie tylko wyświetlenie ostrzeżenia, lecz także wygenerowanie błędu:

```
[Obsolete("Proszę nie używać tej klasy, zamiast niej należy użyć
↪klasy 'Person'.",true)]
```

```
public class Author:Person
{
    // inny kod
}
```

Poniższy rysunek pokazuje, że zastosowanie takiego atrybutu powoduje wygenerowanie wyjątku.

Rysunek 6.8. Zastosowanie atrybutu Obsolete do generowania błędu

Conditional

Kolejny z predefiniowanych atrybutów, `Conditional`, ogranicza wykonanie na podstawie warunku umieszczonego w kodzie.

Przykład

Przeanalizujmy poniższy przykład, który uzależnia możliwość wykonania metody od zastosowania dyrektywy `Debug` preprocesora (zagadnienia związane z preprocesorem oraz jego dyrektywami zostaną opisane w dalszej części rozdziału):

```
#define Debug
using System.Diagnostics;
using static System.Console;

namespace Dzien06
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            PersonList();
            ReadLine();
        }

        [Conditional("Debug")]
        private static void PersonList()
```

```
            {
                WriteLine("Lista osób:");
                foreach (var person in Person.GetPersonList())
                {
                    WriteLine($"Personalia:{person.FirstName} {person.LastName}");
                    WriteLine($"Wiek:{person.Age}");
                }
            }
        }
    }
```

Definiując dyrektywy preprocesora, trzeba pamiętać o jednej rzeczy: konieczne jest zapisywanie ich na samym początku pliku.

Tworzenie i stosowanie niestandardowych atrybutów

W poprzednim punkcie rozdziału zostały przedstawione predefiniowane atrybuty; wspomniano także, że ich możliwości są raczej ograniczone, a w praktycznych zastosowaniach konieczne będzie używanie znacznie bardziej złożonych atrybutów. W takich przypadkach może się przydać możliwość tworzenia własnych, niestandardowych atrybutów — przypominają one atrybuty predefiniowane, jednak to ich twórca określa sposób działania takich atrybutów oraz typy ich elementów docelowych. Wszystkie niestandardowe atrybuty należy definiować jako klasy dziedziczące po klasie `System.Attribute`.

W tym punkcie rozdziału zaimplementujemy niestandardowy atrybut spełniający następujące założenia:

- Atrybut będzie nosić nazwę `ErrorLogger`.
- Atrybut będzie działał we wszystkich dostępnych środowiskach: do debugowania, do tworzenia kodu, produkcyjnym itd.
- Zastosowanie atrybutu będzie się ograniczało jedynie do metod.
- Atrybut powinien powodować wyświetlanie niestandardowych lub podanych komunikatów o błędach.
- Domyślnie atrybut powinien działać w środowisku `Debug`.
- W razie zastosowania w środowisku do tworzenia kodu oraz środowisku `Debug` atrybut powinien wyświetlać komunikat i zgłaszać wyjątek.

Wymagania wstępne

Do tworzenia i stosowania niestandardowych atrybutów będziemy potrzebować:

- Visual Studio 2017 lub nowszego;
- .NET Core 1.1 lub nowszego.

Poniżej przedstawiony został kod niestandardowego atrybutu spełniającego opisane wcześniej założenia:

Kup ksi k   Pole ksi k

```
public class ErrorLogger : Attribute
{
    public ErrorLogger(string exception)
    {
        switch (Env)
        {
            case Env.Debug:
            case Env.Dev:
                WriteLine($"{exception}");
                throw new Exception(exception);
            case Env.Prod:
                WriteLine($"{exception}");
                break;
            default:
                WriteLine($"{exception}");
                throw new Exception(exception);
        }
    }


    public Env Env { get; set; }
}
```

Działanie powyższego atrybutu sprowadza się do wyświetlenia w oknie konsoli wyjątków podanych w kodzie, w którym atrybut został użyty. W przypadku środowisk `Debug` oraz `Dev` oprócz wyświetlenia komunikatu zostanie także zgłoszony wyjątek.

Kolejny fragment kodu przedstawia proste przykłady zastosowania tego atrybutu:

```
public class MathClass
{
    [ErrorLogger("Operacja Add jest w trakcie implementacji!", Env = Env.Debug)]
    public string Add(int num1, int num2)
    {
        return $"Suma liczb {num1} oraz {num2} wynosi: {num1 + num2}";
    }

    [ErrorLogger("Operacja Subtract jest w trakcie implementacji!", Env = Env.Dev)]
    public string Subtract(int num1, int num2)
    {
        return $"Różnica liczb {num1} i {num2} wynosi: {num1 - num2}";
    }

    [ErrorLogger("Operacja Multiply jest w trakcie implementacji!", Env = Env.Prod)]
    public string Multiply(int num1, int num2)
    {
        return $"Iloczyn liczb {num1} i {num2} wynosi: {num1 * num2}";
    }
}
```


Kup ksi k Pole ksi k

W powyższym przykładzie zostały zdefiniowane różne metody opatrzone atrybutami, które zostaną zastosowane w różnych środowiskach. Atrybuty te spowodują wyświetlenie komunikatu i zgłoszenie wyjątku podczas próby użycia poszczególnych metod.

Stosowanie dyrektyw preprocesora

Dyrektywy preprocesora, jak wskazuje ich nazwa, są przetwarzane przed rozpoczęciem procesu kompilacji kodu. Innymi słowy, preprocesor przekazuje kompilatorowi instrukcje dotyczące przetwarzania informacji, przy czym następuje to przed skompilowaniem kodu przez kompilator.

Ważne zagadnienia

Poniżej przedstawionych zostało kilka ważnych uwag związanych ze stosowaniem preprocesora:

- Dyrektywy preprocesora są w rzeczywistości warunkami uwzględnianymi przez kompilator.
- Dyrektywy preprocesora muszą zaczynać się od znaku `#`.
- W odróżnieniu od instrukcji dyrektyw preprocesora nie należy kończyć średnikiem (`;`).
- Preprocesor nie jest używany do tworzenia i stosowania makr.
- Poszczególne dyrektywy preprocesora należy deklarować w odrębnych wierszach.

Dyrektywy preprocesora w działaniu

Przeanalizujmy poniższy przykład:

```
#if ... #endif
```

Przedstawia on warunkową dyrektywę preprocesora; blok kodu umieszczony wewnątrz niej zostanie przetworzony, jeśli będzie spełniony warunek określony w dyrektywie. Powyższą parę dyrektyw można uzupełnić dyrektywami `#elseif` oraz `#else`. Ponieważ jest to dyrektywa warunkowa, a warunki logiczne w języku C# zwracają wynik typu `Boolean`, to w warunkach podawanych w tych dyrektywach można używać operatorów logicznych, takich jak: równy (`==`), różny (`!=`), koniunkcja logiczna (`&&`), alternatywa logiczna (`||`) oraz negacja (`!`).

> Symbole preprocesora należy definiować na samym początku pliku, w którym będą stosowane, używając do tego celu dyrektywy `#define`.

Przeanalizujmy poniższy fragment kodu, który informuje użytkownika o kompilacji warunkowej:

```
#define DEBUG
#define DEV
using static System.Console;
```


Kup ksi k Pole ksi k

```
namespace Dzien06
{
    public class PreprocessorDirective
    {
        public void ConditionalProcessor() =>
#if (DEBUG && !DEV)
            WriteLine("Zdefiniowano symbol DEBUG.");
#elif (!DEBUG && DEV)
            WriteLine("Zdefiniowano symbol DEV");
#else
            WriteLine("Zdefiniowano symbole DEBUG i DEV");
#endif
    }
}
```

W powyższym przykładzie zdefiniowaliśmy dwa różne symbole — `DEBUG` i `DEV` — dla dwóch różnych środowisk kompilacji. Ze względu na zastosowane dyrektywy preprocesora wykonanie powyższego przykładu spowoduje wyświetlenie wyników przedstawionych na rysunku 6.9.

```
C:\WINDOWS\system32\cmd.exe
Zdefiniowano symbole DEBUG i DEV
```

Rysunek 6.9. Efekty zastosowania dyrektyw preprocesora

Dyrektywy #define i #undef

Najprościej rzecz ujmując, dyrektywa `#define` definiuje symbol, który następnie może być używany w warunkowych dyrektywach preprocesora.

Dyrektywy `#define` nie można używać do deklarowania stałych.

Przy stosowaniu tej dyrektywy należy pamiętać o następujących zagadnieniach:

- Dyrektywy `#define` nie można używać do deklarowania stałych.
- Dyrektywa ta pozwala zdefiniować symbol, lecz nie pozwala przypisywać mu wartości.
- Symbol preprocesora może być używany wyłącznie po jego uprzednim zdefiniowaniu; oznacza to, że dyrektywę `#define` należy umieszczać na samym początku pliku.
- Zasięg symbolu obejmuje cały plik źródłowy, w którym dany symbol został zdefiniowany.

Wróćmy teraz do przykładu prezentującego zastosowanie warunkowej dyrektywy `#if`, na którego początku zostały zdefiniowane dwa symbole preprocesora. Zdefiniować taki symbol można bardzo łatwo, na przykład: `#define DEBUG`.

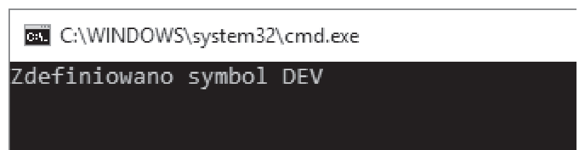
Kup ksi k Pole ksi k

Istnieje także możliwość usunięcia zdefiniowanego wcześniej symbolu; służy do tego dyrektywa preprocesora #undef. Przeanalizujmy kolejny przykład:

```
#define DEBUG
#define DEV
#undef DEBUG
using static System.Console;

namespace Dzien06
{
    public class PreprocessorDirective
    {
        public void ConditionalProcessor() =>
#if (DEBUG && !DEV)
            WriteLine("Zdefiniowano symbol DEBUG.");
#elif (!DEBUG && DEV)
            WriteLine("Zdefiniowano symbol DEV");
#else
            WriteLine("Zdefiniowano symbole DEBUG i DEV");
#endif
    }
}
```

W tym przykładzie najpierw definiujemy, a następnie usuwamy symbol DEBUG; oznacza to, że wykonanie powyższego przykładu wygeneruje wyniki przedstawione na rysunku 6.10.



Rysunek 6.10. Efekty zastosowania dyrektywy preprocesora #undef

Dyrektywy #region oraz #endregion

Dyrektywy #region oraz #endregion są niezwykle przydatne podczas pracy nad długimi plikami źródłowymi. Czasami podczas prac nad dużymi aplikacjami, na przykład korporacyjnymi, pliki źródłowe mogą przekraczać nawet 1000 wierszy kodu, obejmującego przeróżne funkcje, metody czy też logikę biznesową. Dla poprawy przejrzystości można podzielić taki kod na regiony. Dla każdego regionu można określić krótki opis wyjaśniający przeznaczenie danego fragmentu kodu. Przeanalizujmy przykład przedstawiony na rysunku 6.11.

Kup ksi k Pole ksi k

```
#define DEBUG
#define DEV
//#undef DEBUG
using static System.Console;

namespace Dzien06
{
    public class PreprocessorDirective
    {
        #region  Warunkowe dyrektywy preprocesora

        public void ConditionalProcessor() =>
#if (DEBUG && !DEV)
            WriteLine("Zdefiniowano symbol DEBUG.");
#elif (!DEBUG && DEV)
            WriteLine("Zdefiniowano symbol DEV");
#else
            WriteLine("Zdefiniowano symbole DEBUG i DEV");
#endif

        #endregion
    }
}
```

```
#define DEBUG
#define DEV
//#undef DEBUG
using static System.Console;

namespace Dzien06
{
    public class PreprocessorDirective
    {
        Warunkowe dyrektywy preprocesora
    }
}
```

Rysunek 6.11. Regiony kodu źródłowego tworzone przy użyciu dyrektyw #region oraz #endregion

Okno edytora widoczne z lewej strony rysunku 6.11 przedstawia rozwinięty region utworzony przy użyciu dyrektyw `#region … #endregion`, pokazujący, jak można je stosować. Z kolei okno edytora zamieszczone z prawej strony pokazuje, jak wygląda zwinięty region. Po umieszczeniu wskaźnika myszy na tekście opisu regionu poniżej zostanie wyświetlony prostokątny obszar prezentujący jego zawartość. A zatem wcale nie trzeba rozwijać regionu, by przejrzeć umieszczony w nim kod źródłowy.

Dyrektywa #line

Dyrektywa `#line` pozwala zmienić numer wiersza, który będzie wyświetlany w komunikatach kompilatora. Dodatkowo można w niej także określić nazwę pliku, która będzie wyświetlana w tych komunikatach, choć ta możliwość jest opcjonalna. Ta dyrektywa może być przydatna w zautomatyzowanych etapach procesu budowania kodu. W przypadkach, gdy z pierwotnego kodu źródłowego zostały usunięte numery wierszy, wyniki będą musiały być generowane na podstawie numeracji wierszy pierwotnego pliku źródłowego.

Z kolei dyrektywa o postaci `#line default` przywraca normalny sposób numerowania wierszy kodu.

Zastosowanie dyrektywy o postaci `#line hidden` sprawia, że ani nazwa pliku, ani numeracja wierszy kodu w komunikatach generowanych przez kompilator nie będzie modyfikowana.

I w końcu dyrektywa o postaci `#line` *`nazwa_pliku`* daje możliwość określenia nazwy pliku, która ma być wyświetlana w komunikatach generowanych przez kompilator. Nazwa pliku musi być zapisana w cudzysłowach i umieszczona przed ewentualnym numerem wiersza.

Przeanalizujmy poniższy fragment kodu:

```
        public void LinePreprocessor()
        {
#line 85 "LineprocessorIsTheFileName"
            WriteLine("Ta instrukcja jest w wierszu nr 85, a nie nr 25.");
```


Kup ksi k   Pole ksi k

```
#line default
            WriteLine("Ta instrukcja jest w wierszu nr 29, a nie nr 28.");
#line hidden
            WriteLine("Ta instrukcja jest w wierszu nr 30.");
        }
```

W powyższym przykładzie wiersz, który normalnie ma numer 25, będzie traktowany jak wiersz 85.

Dyrektywa #warning

Dyrektywa `#warning` zapewnia możliwość wygenerowania ostrzeżenia w dowolnym miejscu kodu i zazwyczaj jest stosowana razem z dyrektywami warunkowymi. Przeanalizujmy poniższy fragment kodu:

```
        public void WarningPreProcessor()
        {
#if DEBUG
#warning "Kompilacja w środowisku DEBUG."
            WriteLine("Aktualnie używane jest środowisko DEBUG.");
#endif
        }
    }
```

Zastosowanie powyższego kodu sprawi, że podczas kompilacji zostanie wyświetlone ostrzeżenie, którego tekst został podany w dyrektywie `#warning` (patrz rysunek 6.12).

```
        public void WarningPreProcessor()
        {
#if DEBUG
#warning "Kompilacja w środowisku DEBUG."
            WriteLine("Ak   ni
#endif
        }
```

#warning: „"Kompilacja w środowisku DEBUG.""
Pokaż potencjalne rozwiązania (Alt+Enter or Ctrl+.)

Rysunek 6.12. Ostrzeżenie wyświetlane przy użyciu dyrektywy #warning

Dyrektywa #error

Dyrektywa `#error` zapewnia możliwość wygenerowania błędu w dowolnym miejscu kodu. Przeanalizujmy poniższy fragment kodu:

```
        public void ErrorPreProcessor()
        {
#if DEV
#error "Kompilacja w środowisku DEV."
                WriteLine("Aktualnie używane jest środowisko DEV.");
#endif
        }
```

Zastosowanie dyrektywy `#error` spowoduje wygenerowanie błędu, który z kolei sprawi, że pliku nie uda się prawidłowo skompilować; wyświetlony komunikat o błędzie będzie odpowiadał tekstowi podanemu w dyrektywie `#error` (jak pokazano na rysunku 6.13).

```
        public void ErrorPreProcessor()
        {
#if DEV
#error "Kompilacja w środowisku DEV."
                    WriteLin     #błąd: „"Kompilacja w środowisku DEV.""   to DEV.");
#endif
        }

    }
}
```

Rysunek 6.13. Efekty zastosowania dyrektywy #error

W tym podrozdziale zostały przedstawione różne dyrektywy preprocesora oraz przykłady ich zastosowania.

> Pełna lista dyrektyw preprocesora języka C# jest dostępna w jego oficjalnej dokumentacji, na stronie *https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives/*.

Prezentacja LINQ

LINQ to skrót nazwy *Language Integrated Query* — mechanizmu języka C# pozwalającego na generowanie zapytań do danych. Dzięki zastosowaniu specjalnej składni, przypominającej nieco sposób zapisu pytania o dane spełniające określone kryteria, LINQ daje możliwość łatwego pisania. Innymi słowy, możemy powiedzieć, że LINQ to składnia służąca do przeszukiwania i pobierania danych.

W tym podrozdziale przedstawię bardzo prosty przykład zapytania LINQ. Zakładamy w nim, że dysponujemy listą obiektów klasy `Person`. Przedstawione zapytanie pozwala wyszukać obiekt, w którym identyfikator spełnia podany warunek:

```
private static void TestLinq()
{
    var person = from p in Person.GetPersonList()
                 where p.Id == 1
                 select p;
    foreach (var per in person)
    {
```


Kup ksi k   Pole ksi k

```
                WriteLine($"Id osoby:{per.Id}");
                WriteLine($"Personalia:{per.FirstName} {per.LastName}");
                WriteLine($"Wiek:{per.Age}");
            }
        }
```

W powyższym przykładzie przeglądamy listę osób w poszukiwaniu obiektu, w którym identyfikator (Id) osoby ma wartość 1. To zapytanie LINQ zwraca wyniki typu IEnumerable<Person>, które z łatwością można przejrzeć, używając pętli foreach. Wyniki wykonania powyższego przykładu zostały przedstawione na rysunku 6.14.

```
C:\WINDOWS\system32\cmd.exe
Id osoby:1
Personalia:Jan Kowalski
Wiek:31
```

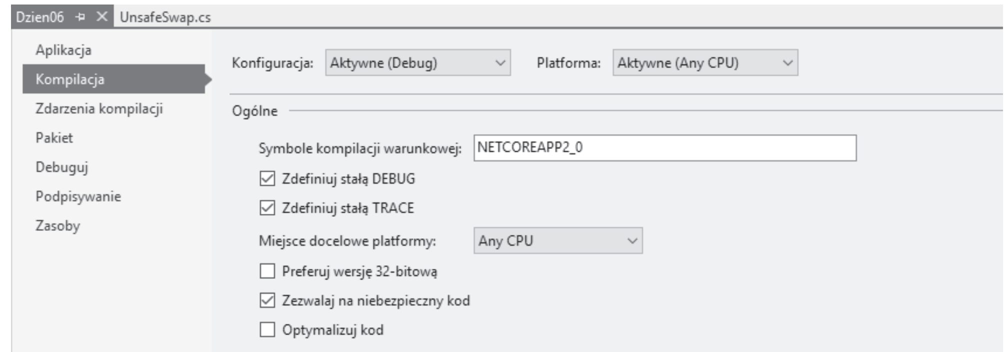
Rysunek 6.14. Dane pobrane przy użyciu zapytania LINQ

> Wyczerpująca prezentacja LINQ wykracza poza ramy niniejszej książki. Przykłady prezentujące pełne możliwości tej technologii są dostępne na stronie *https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b*.

Pisanie niebezpiecznego kodu

W tym podrozdziale zostanie zamieszczone wprowadzenie do zagadnień pisania niebezpiecznego kodu w środowisku Visual Studio. Język C# pozwala na pisanie kodu, który jest kompilowany i tworzy obiekty, a nimi z kolei zarządza tak zwany mechanizm odzyskiwania pamięci (ang. *garbage collector*; więcej informacji na jego temat można znaleźć w pierwszym dniu kursu). Najprościej rzecz ujmując, w języku C#, w odróżnieniu od C++, nie występuje pojęcie wskaźników ani do funkcji, ani do danych. Jednak czasami nawet w języku C# występują sytuacje, w których zastosowanie takich wskaźników, analogicznych do wskaźników w językach C lub C++, może być niezbędne. Właśnie z myślą o takich sytuacjach w języku C# wprowadzono możliwość pisania tak zwanego **niebezpiecznego kodu**. Język ten udostępnia modyfikator unsafe, który informuje, że fragment kodu nie jest zarządzany przez mechanizm odzyskiwania pamięci oraz że mogą w nim być używane wskaźniki i inne niebezpieczne rozwiązania. Aby móc pisać taki niebezpieczny kod, w pierwszej kolejności trzeba włączyć w Visual Studio (w wersji 2017 lub nowszej) opcję niebezpiecznej kompilacji. W tym celu należy wyświetlić właściwości projektu, przejść na kartę *Kompilacja*, a następnie zaznaczyć pole wyboru *Zezwalaj na niebezpieczny kod*, jak pokazano na rysunku 6.15.

Kup ksi k Pole ksi k



Rysunek 6.15. Włączanie możliwości pisania niebezpiecznego kodu

Bez włączenia tej opcji tworzenie niebezpiecznego kodu nie będzie możliwe, co pokazano na rysunku 6.16.

```
1   namespace Dzien06
2   {
3       public  class UnsafeSwap
4       {
5
6           public unsafe void SwapNumbers(int*  num1, int* num2)
7           {
8
9           }
10      }
11  }
12
13
```

void UnsafeSwap.SwapNumbers(int* num1, int* num2)

Niebezpieczny kod może się pojawić tylko w przypadku kompilowania przy użyciu opcji /unsafe

Rysunek 6.16. Próba użycia niebezpiecznego kodu bez włączenia odpowiedniej opcji w ustawieniach projektu

Po włączeniu opcji zezwalającej na stosowanie niebezpiecznego kodu możemy spróbować napisać funkcję, która będzie zamieniać wartości dwóch zmiennych przekazanych jako wskaźniki. Przeanalizujmy poniższy fragment kodu:

```
public unsafe void SwapNumbers(int* num1, int* num2)
{
    int tempNum = *num1;
    *num1 = *num2;
    *num2 = tempNum;
}
```

Jak widać, jest to bardzo prosta funkcja, która zamienia wartości dwóch zmiennych, dysponując ich wskaźnikami. Poniżej przedstawiony został kod, który używa tej funkcji i wyświetla wyniki jej działania:

```
private static unsafe void TestUnsafeSwap()
{
    Write("Podaj pierwszą liczbę: ");
    var num1 = Convert.ToInt32(ReadLine());
    Write("Podaj drugą liczbę: ");
```


Kup ksi k Pole ksi k

```
        var num2 = Convert.ToInt32(ReadLine());
        WriteLine("Liczby przed wykonaniem zamiany:");
        WriteLine($"Liczba1:{num1}, Liczba2:{num2}");
        // wywołanie funkcji zamieniającej liczby
        new UnsafeSwap().SwapNumbers(&num1, &num2);
        WriteLine("Liczby po wykonaniu zamiany:");
        WriteLine($"Number1:{num1}, Number2:{num2}");
    }
```

W powyższym przykładzie najpierw pobieramy dwie liczby i zapisujemy w dwóch zmiennych, wyświetlamy je, a następnie zamieniamy i ponownie wyświetlamy obie zmienne. Wyniki wykonania powyższego przykładu przedstawiono na rysunku 6.17.

```
C:\WINDOWS\system32\cmd.exe
Podaj pierwszą liczbę: 35
Podaj drugą liczbę: 138
Liczby przed wykonaniem zamiany:
Liczba1:35, Liczba2:138
Liczby po wykonaniu zamiany:
Liczba1:138, Liczba2:35
```

Rysunek 6.17. Efekty działania niebezpiecznego kodu korzystającego ze wskaźników

W tym podrozdziale zostało zamieszczone wprowadzenie do zagadnień związanych z pisaniem niebezpiecznego kodu.

> Więcej informacji na temat pisania niebezpiecznego kodu można znaleźć w oficjalnej dokumentacji języka, na stronie *https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/unsafe-code*.

Pisanie asynchronicznego kodu

Zanim przedstawimy możliwości tworzenia kodu, który będzie wykonywany asynchronicznie, przedstawimy jego zwyczajną wersję — czyli kod wykonywany synchronicznie:

```
public class FilePolling
{
    public void PoleAFile(string fileName)
    {
        Console.Write($"To jest plik ankiety: {fileName}");
        // kod obsługi ankiety
    }
}
```


Kup ksi k

Pole ksi k

Powyższy przykład jest prosty i uroczy. Wyświetla on nazwę pliku ankiety. W tym przypadku system musi zaczekać na zakończenie obsługi ankiety, zanim przystąpi do wykonywania kolejnych zadań. Właśnie w taki sposób działa kod synchroniczny. A teraz wyobraźmy sobie scenariusz, w którym rozpoczęcie innych operacji lub zadań nie musi następować po zakończeniu czynności zaczętych wcześniej. Na taki sposób funkcjonowania pozwala kod asynchroniczny, który można pisać dzięki słowu kluczowemu `async`.

Przeanalizujmy poniższy fragment kodu:

```
public async void PoleAFileAsync(string fileName)
{
    Console.Write($"To jest plik ankiety: {fileName}");
    // asynchroniczny kod obsługi ankiety
}
```

Dzięki dodaniu jednego słowa kluczowego, `async`, kod może być wykonywany asynchronicznie.

Analizując powyższy przykład, można stwierdzić, że programowanie asynchroniczne nie zmusza kodu klienta do czekania z wykonaniem innej funkcji lub operacji na zakończenie innej operacji asynchronicznej. Albo mówiąc inaczej: kod asynchroniczny nie może uniemożliwić wywołania innej operacji, która ma być wykonana.

W tym podrozdziale przedstawione zostało wprowadzenie do programowania asynchronicznego. Wyczerpujący opis zagadnień związanych z programowaniem asynchronicznym wykracza poza ramy tematyczne tej książki. Więcej informacji o tym sposobie programowania można znaleźć w oficjalnej dokumentacji języka, na stronie *https://docs.microsoft.com/en-us/dotnet/csharp/async*.

Ćwiczenia praktyczne

1. Zdefiniuj ogólną klasę `StringCalculator` na podstawie kodu: *https://github.com/garora/TDD-Katas/tree/master/Src/CSharp/Net%20Core/StringCalculator*.
2. Napisz ogólną i nieogólną klasę kolekcji, a następnie porównaj wydajność ich działania.
3. W podpunkcie pt. „Dlaczego warto używać typów ogólnych?” przedstawiony został fragment kodu informujący o wyjątkach zgłaszanych podczas kompilacji. Wyjaśnij, dlaczego nie warto używać kodu takiego, jak ten przedstawiony poniżej:

```
internal class Program
{
    private static void Main(string[] args)
    {
        // żadne wyjątki nie są zgłaszane, ani w trakcie kompilacji,
        // ani w trakcie wykonywania kodu
```


Kup ksi k Pole ksi k

```
            ArrayList authorEditorArrayList = new ArrayList {
                "Jan Kowalski", 43, "Artur Nowak", 25 };
            foreach (var authorEditor in authorEditorArrayList)
            {
                WriteLine($"{authorEditor}");
            }
        }
    }
```

4. Jakie jest przeznaczenie słowa kluczowego `default` w kodzie ogólnym? Opisz je i przedstaw na przykładzie.
5. Napisz prosty przykład, w którym zostaną wykorzystane wszystkie trzy typy predefiniowanych atrybutów.
6. Jaki jest domyślny typ ograniczenia dla atrybutów? Napisz program prezentujący wszystkie typy ograniczeń.
7. Zaimplementuj niestandardowy atrybut o nazwie `LogFailuresAttribute`, którego działanie będzie polegać na rejestrowaniu wszystkich zgłaszanych wyjątków w pliku tekstowym.
8. Dlaczego dyrektywy preprocesora `#define` nie można używać do deklarowania stałych?
9. Napisz program tworzący listę obiektów `Author` i przeszukaj ją, korzystając z technologii LINQ.
10. Napisz program sortujący tablicę.
11. Napisz kompletny program, który będzie zapisywał zawartość w pliku, używając kodu synchronicznego i asynchronicznego.

Podsumowanie dnia 6.

W tym rozdziale zostały przedstawione zagadnienia zaawansowane, takiej jak: typy ogólne, atrybuty, dyrektywy preprocesora, LINQ, pisanie niebezpiecznego kodu oraz programowanie asynchroniczne.

Rozdział rozpoczęła prezentacja typów ogólnych, w ramach której, na odpowiednich przykładach, wyjaśniono, czym są klasy ogólne. Następnie zostały opisane atrybuty i przedstawione przykłady stosowania predefiniowanych atrybutów dostępnych w języku C#. Potem napisaliśmy własny, niestandardowy atrybut i zastosowali go w kodzie. Kolejnym opisanym zagadnieniem były dyrektywy preprocesora — w ramach ich prezentacji przedstawiono wybrane dyrektywy preprocesora oraz ich zastosowanie na przykładach. Na końcu rozdziału zostały pokrótce opisane: technologia LINQ, tworzenie niebezpiecznego kodu oraz programowanie asynchroniczne.

Kolejny rozdział będzie stanowić zakończenie tego siedmiodniowego kursu języka C#. Będzie on poświęcony zagadnieniom programowania obiektowego oraz rozwiązaniom obiektowym zastosowanym w języku C#.

Kup książkę Poleć książkę

Kup ksi k

Pole ksi k

Skorowidz

.NET, 17
.NET Core, 18
.NET Standard, 19

A

abstrakcja, 238
agregacja, 223
analiza kodu, 36
aplikacja konsolowa, 27, 29
ArrayList, 171, 188
asynchroniczna metoda Main, 105
asynchroniczny kod, 217
atrybut, 203
  AttributeUsage, 204
  Conditional, 206
  Obsolete, 205
atrybuty niestandardowe, 207

B

biblioteka
  Corefx, 19
BitArray, 185, 189
blok
  catch, 147
  finally, 147
  try, 147

C

cechy
  interfejsów, 240
  klas abstrakcyjnych, 239
CLI, Command Line Interface, 19, 21
Cloud9, 20
Coreclr, 19
Corefx, 19
CUI, Console User Interface, 36

D

debugowanie, 29
deklarowanie
  kolekcji
    ArrayList, 171
    Hashtable, 173
    Stack, 179
    Queue, 182
  typu delegacji, 166
  zdarzeń, 169
dekonstrukcja, 94
delegacje, 57, 166
dopasowywanie wzorców, 96, 110, 151
dostęp do składowej, 115
dyrektywa preprocesora, 209
  #define, 210
  #endregion, 211
  #error, 213
  #line, 212
  #regi, 211
  #undef, 210
  #warning, 213
dziedziczenie, 224, 233
  hierarchiczne, 226
  hybrydowe, 227
  niejawne, 228
  pojedyncze, 225
  wielokrotne, 225, 235
  wielopoziomowe, 227
dźwięk, 37

E

edytor tekstu, 20

F

finalizatory, 229
funkcje lokalne, 102

H

Hashtable, 173, 188
hermetyzacja, 242, 243
historia języka, 27

I

IDE, Integrated Development Environment, 20
    Cloud9, 20
    JetBrain Rider, 20
    Visual Studio Code, 20
    Zeus IDE, 20
identyfikatory, 39
indeksery, 143
informacje
    o delegacjach, 166
    o konwersji typów, 64
    o metodach rozszerzeń, 164
    o typie, 161
    o zdarzeniach, 166
instalowanie Visual Studio Community 2017, 22
instancje delegacji, 167
instrukcja, 66
    break, 75
    continue, 75
    default, 76
    if, 68, 70
    if … else, 69
    if … else if … else, 69
    switch, 71, 99
    try … catch … finally, 146
instrukcje
    deklaracji, 67
    iteracji, 72
    obsługi wyjątków, 76
    skoku, 74
    wyboru, 68
    wyrażenia, 68
interfejsy, 56, 239

J

JetBrain Rider, 20

K

karta
    Kompilacja, 35
    Pakiet, 35
klasa, 55, 82
    ArrayList, 171, 188
    BitArray, 185, 189
    FileStream, 144
    Hashtable, 173, 188
    Queue, 181, 189
    SortedList, 176, 188
    Stack, 179, 189
    System.Reflection.TypeExtensions, 164
    System.Type, 163
klasy
    abstrakcyjne, 238
    bazowe, 201
    kolekcji, 188
klauzula when, 101
kod
    asynchroniczny, 217
    niebezpieczny, 215
    źródłowy, 34
kolejka, 189
kolekcje, 170, 188, 193
kolory, 37
konsolowy interfejs użytkownika, CUI, 36
konstruktory
    domyślne, 200
    statyczne, 228
kontekstowe słowa kluczowe, 39, 51
konwersja
    jawna, 65
    niejawna, 65
krotki, 89, 95, 108

L

liczba dopasowań, 152
LINQ, Language Integrated Query, 214
lista obiektów, 188
literały dwójkowe, 103

Kup ksi k Pole ksi k

Ł

łańcuch znaków, 76, 81

M

metoda
  Beep, 37
  Main, 31, 105
metody, 135
  klasy
    ArrayList, 172
    Hashtable, 174
    Queue, 183
    SortedList, 178
    Stack, 180
  rozszerzeń, 164
modyfikator, 38, 115
  abstract, 123
  async, 127
  const, 127
  event, 128
  extern, 128
  internal, 118
  internald, 231
  new, 128, 237
  override, 129
  partial, 129
  private, 121, 229
  protected, 117, 230
  public, 115, 232
  readonly, 130
  sealed, 131
  static, 133
  unsafe, 134
  virtual, 134
  złożony, 120
modyfikatory
  dostępu, 242
  reguły, 122

N

narzędzia wiersza poleceń, 19
nazwa
  klasy, 31
  rozwiązania, 31
  krotki, 108
niebezpieczny kod, 215
Notepad++, 20

O

obiekty, 223
  agregacja, 223
  powiązanie, 223
  złożenie, 223
obsługa wyjątków, 76, 145
odzwierciedlanie, 157, 161
ograniczenie, 106
  domyślnego dostępu, 122
  dziedziczenia, 125
  interfejsu, 202
  klasy bazowej, 201
  łączenia, 122
  przestrzeni nazw, 122
  typu najwyższego poziomu, 122
  typu zagnieżdżonego, 122
OOP, Object Oriented Programming, 221
operacje
  na tablicach, 76
  wejścia-wyjścia, 144
operator new, 128, 237
operatory, 38, 57
  priorytety, 61
  przeciążanie, 62

P

pakiet NuGet ValueTuple, 108
para klucz – wartość, 188
parametry metod, 39
PCL, Portable Class Libraries, 20
pętla
  do … while, 72
  for, 73
  foreach, 74
  while, 73
pierwszy program, 27
platforma .NET, 15
plik rozwiązania, 32
pliki projektu, 32
polimorfizm, 236, 244
  czasu kompilacji, 245
  czasu wykonywania, 247
powiązanie, 223

Kup ksi k   Pole ksi k

poziomy dostępu, 115, 123
prezentacja instrukcji, 66
priorytety operatorów, 61
programowanie, 15
  obiektowe, OOP, 221
  proceduralne, 222
  strukturalne, 222
projekt, 28
przeciążanie operatorów, 62
przestrzeń nazw, 30
  System.Collections, 193

Q

Queue, 181, 189

R

reguły dotyczące modyfikatora
  abstract, 124
  static, 133
relacja
  JEST, 233
  Ma/Może, 236
rodzaje właściwości, 139

S

separatory cyfr, 104
składowe, 228
  będące wyrażeniem lambda, 141
  nieabstrakcyjne, 125
  o ciele blokowym, 141
słowa kluczowe, 38–53
  dostępu, 39
  instrukcji, 39
  kontekstowe, 39, 51
  konwersji, 39
  operatorów, 39
  przestrzeni nazw, 39
  traktowane dosłownie, 39
  zapytań, 39
SortedList, 176, 188
Stack, 179, 189
stałe, 107
stos, 189
stosowanie
  abstrakcji, 238
  delegacji, 167
  dziedziczenia, 233
  dziedziczenia wielokrotnego, 235
  hermetyzacji, 243
  odzwierciedlania, 161
  polimorfizmu, 249
struktura, 82
  aplikacji, 29
  System.ValueTuple, 91
strumień
  wejściowy, 144
  wyjściowy, 144
sygnatura, 106
symbole kompilacji warunkowej, 35

Ś

środowisko
  programistyczne, 20
  wykonawcze, 19

T

tablica, 76, 189
  typu bool, 189
tablice
  jednowymiarowe, 78
  nieregularne, 79
  wielowymiarowe, 78
tryb debugowania, 29
tworzenie projektu, 27, 28
typ, 38, 53
  Null, 57
  String, 57
  wskaźnikowy, 57
typy
  atrybutów, 203
  danych, 54
  nieogólne, 170
  ogólne, 110, 191, 193
    ograniczenia, 197
  polimorfizmu, 245
  referencyjne, 55, 199
  tablic, 78
  wartościowe, 53, 199

U

użycie niebezpiecznego kodu, 216

Kup ksi k
Pole ksi k

V

Visual Studio Code, 20
Visual Studio Community 2017, 21

W

widoczność składowych, 228
wieloplatformowość, 18
wirtualny charakter, 126
właściwości, 138
  do odczytu i zapisu, 139
  klasy
    ArrayList, 171
    Hashtable, 174
    Queue, 183
    SortedList, 176
    Stack, 180
    System.Type, 163
  obliczane, 141
  projektu, 34
  tylko do odczytu, 140
  wykonujące weryfikację, 142
wybór instalowanych komponentów, 22
wydajność, 189
wyjątki, 145
  definiowane przez użytkownika, 149
  generowane przez kompilator, 148
wyrażenia
  domyślne, 106
  regularne, 151
wyrażenie
  case, 101
  is, 97
wzorce, 96, 151
  stałej, 100
  typu, 101

Z

zagnieżdżone instrukcje if, 70
zależności, 33
zapytania do danych, 214
zasada implementacji, 125
zdarzenia, 169
zestawy referencyjne, 111
Zeus IDE, 20
zintegrowane środowisko programistyczne, IDE, 20
złożenie, 223
zmienne składowe, 107
znak
  cyfry, 152
  kropki, 151
  minus, 152
  odstępu, 151
  słowa, 151
znaki specjalne, 151

Kup książkę
Poleć książkę

Notatki

PROGRAM
PARTNERSKI
GRUPY WYDAWNICZEJ HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW
w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA WYDAWNICZA
Helion SA

Podstawy języka C# w 7 dni!

Język C# to nowoczesne narzędzie programowania obiektowego stworzone przez Microsoft. Wyjątkowo przydatne, wielofunkcyjne i powszechnie używane. Także ze względu na wieloplatformowość i otwartą specyfikację. Jednak najważniejsze, że wystarczy gruntownie opanować podstawy języka C# i już można tworzyć solidne i wydajne aplikacje zarówno do zastosowań desktopowych, jak i na urządzenia mobilne! To świetna wiadomość i dla początkujących programistów, i dla tych, którzy postanowili poznać nowy język.

Ten podręcznik ułatwi Ci szybkie opanowanie podstaw języka C#. Autor, Gaurav Aroraa, doświadczony guru IT, proponuje 7-dniowy plan działania, którego realizacja da Ci pełne spektrum umiejętności programistycznych. Zacznij już teraz kurs języka C#! Językowa biegłość przyniesie Ci satysfakcję i pewność siebie, w efekcie poprawi Twoją pozycję w zespole czy na rynku pracy.

W 7 dni przejdziesz intensywny kurs:

- architektury i konfiguracji środowiska .NET
- pisania programów C# w zintegrowanym środowisku Visual Studio 2017
- definiowania zmiennych, składni, instrukcji, tablic oraz sterowania przepływem
- pojęć programowania obiektowego
- wiedzy o atrybutach, kolekcjach, typach ogólnych oraz LINQ
- tworzenia i stosowania składowych klas, takich jak modyfikatory, metody, właściwości, indeksery
- operacji wejścia-wyjścia na plikach, obsługi błędów oraz stosowania wyrażeń regularnych
- pisania praktycznych aplikacji w języku C#

Gaurav Aroraa w ciągu prawie 20-letniej kariery był mentorem tysięcy studentów informatyki i branżowych specjalistów. Posiada tytuł Microsoft MVP — jest trenerem Scrum, XEN w zastosowaniach ITIL-F. Ma certyfikaty PRINCE-P i PRINCE-F APMG. Tworzy treści w TechNet Wiki. Jest jednym z założycieli firmy Innatus Curo Software LLC.