

Dawid Farbaniec



Microsoft Visual Studio 2012

Programowanie w C#

Visual Studio i język C# — potężny duet w rękach programisty!

Opanuj Visual Studio 2012 i platformę .NET — narzędzia do tworzenia aplikacji
Odkryj niezwykle możliwości obiektowego języka programowania C#
Poznaj zaawansowane zagadnienia programowania obiektowego, podstawy obsługi
sieci oraz Asembler IL



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

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

Autor oraz 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.

Redaktor prowadzący: Michał Mrowiec

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Fotografia na okładce została wykorzystana za zgodą Shutterstock.com

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)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie?vsl2pc>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-246-6562-4

Copyright © Helion 2013

Printed in Poland.

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

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

Spis treści

Rozdział 1. Wstęp	9
1.1. O języku C# i platformie .NET	9
1.2. Podstawowe pojęcia	9
1.3. Potrzebne narzędzia	10
Rozdział 2. Microsoft Visual Studio 2012	11
2.1. Co nowego?	12
2.2. Instalacja	12
2.3. Konfiguracja	14
Pasek obsługi odpluskwiania	14
Numerowanie wierszy	14
2.4. Tworzenie nowego projektu	15
2.5. Kompilacja i uruchamianie	15
2.6. Odpluskwianie (ang. Debugging)	15
Błędy procesu kompilacji	15
Błędy pokompilacyjne	16
2.7. Okna, menu i paski narzędzi	16
Okna	16
Górne menu	16
Paski narzędzi	17
2.8. Składniki pakietu	17
2.9. Projektowanie diagramów UML	18
Rozdział 3. Język C#. Podstawy	19
3.1. Struktura kodu źródłowego	19
3.2. Komentarze	20
Komentarz blokowy	20
Komentarz liniowy	20
Komentarz XML	20
3.3. Program „Witaj, świecie!”	21
3.4. Typy danych	21
Typy proste	21
Typy referencyjne	22
Typ strukturalny	24
Typ wyliczeniowy	24
Rzutowanie i konwersja typów	24
3.5. Proste operacje wejścia/wyjścia	25

Wyświetlanie danych	25
Pobieranie danych	26
3.6. Preprocesor	26
Dyrektywa #if	26
Dyrektywa #else	27
Dyrektywa #elif	27
Dyrektywa #endif	27
Dyrektywa #define	27
Dyrektywa #undef	28
Dyrektywa #warning	28
Dyrektywa #error	28
Dyrektywa #line	28
Dyrektywa #region	29
Dyrektywa #endregion	29
Dyrektywa #pragma warning	29
3.7. Zmienne i stałe	30
3.8. Stos i sterta	31
Wydajność	31
3.9. Instrukcja warunkowa if	32
3.10. Instrukcja wyboru switch	34
3.11. Operatory	35
Podstawowe	36
Jednoargumentowe	38
Mnożenie, dzielenie i modulo	40
Przesunięcia	40
Relacje i sprawdzanie typów	41
Równość i różność	42
Koniunkcja logiczna	42
Alternatywa wykluczająca logiczna	42
Alternatywa logiczna	42
Koniunkcja warunkowa	43
Alternatywa warunkowa	43
Operator warunkowy	43
Przypisania	43
3.12. Pętle	45
Pętla do-while	45
Pętla for	45
Pętla foreach	48
Pętla while	49
Kontrola przepływu	49
3.13. Argumenty wiersza poleceń	52
3.14. Metody	53
Deklaracja metod	53
Przekazywanie przez referencję lub przez wartość	54
3.15. Tablice	55
Przekazywanie tablic jako argumentów metod	56
Klasa System.Array	57
3.16. Wskaźniki	60
Kod nienadzorowany (ang. unsafe code)	60
Typy wskaźnikowe	61

Rozdział 4. Język C#. Programowanie obiektowe	63
4.1. Klasy i obiekty	63
Słowo kluczowe this	65
4.2. Konstruktor i destruktor	66
4.3. Dziedziczenie	67
Klasy zagnieżdżone	68
4.4. Modyfikatory dostępu	69
Słowo kluczowe readonly	70
Pola powinny być prywatne	70
4.5. Wczesne i późne wiązanie	71
Wczesne wiązanie vs późne wiązanie	71
Opakowywanie zmiennych	72
4.6. Przeciążanie metod	72
4.7. Przeciążanie operatorów	73
Słowa kluczowe implicit i explicit	75
4.8. Statyczne metody i pola	76
4.9. Klasy abstrakcyjne i zabezpieczone	77
4.10. Serializacja	78
Użyteczność serializacji	79
Zapis obiektu do pliku XML	79
Odczyt obiektu z pliku XML	79
4.11. Przestrzenie nazw	80
4.12. Właściwości	82
4.13. Interfejsy	83
Płytki i głęboka kopia obiektu	84
4.14. Indeksery	86
4.15. Polimorfizm	88
Składowe wirtualne	91
Ukrywanie składowych klasy bazowej	92
Zapobieganie przesłanianiu wirtualnych składowych klasy pochodnej	92
Dostęp do wirtualnych składowych klasy bazowej z klas pochodnych	93
Przesłanianie metody ToString()	94
4.16. Delegaty	94
Metody anonimowe	95
Wyrażenia lambda	96
Delegat Func	97
4.17. Zdarzenia	98
4.18. Metody rozszerzające	98
4.19. Kolekcje	99
Wybieranie klasy kolekcji	100
Klasa Queue	101
Klasa Stack	102
Klasa ArrayList	103
Klasa StringCollection	103
Klasa Hashtable	104
Klasa SortedList	105
Klasa ListDictionary	105
Klasa StringDictionary	106
Klasa NameObjectCollectionBase	107
Klasa NameValueCollection	110
4.20. Typy generyczne	111
Klasa generyczna Queue	112
Klasa generyczna Stack	113
Klasa generyczna LinkedList	114

Klasa generyczna List	115
Klasa generyczna Dictionary	116
Klasa generyczna SortedDictionary	118
Klasa generyczna KeyedCollection	120
Klasa generyczna SortedList	123
4.21. Kontra i kowariancja	125
Rozdział 5. Język C#. Pozostałe zagadnienia	127
5.1. Wywoływanie funkcji przez PInvoke	127
5.2. Napisy (ang. Strings)	129
Deklaracja i inicjalizacja	129
Niezmienność obiektów String	130
Znaki specjalne	130
Formatowanie napisów	130
Napisy częściowe	131
Dostęp do pojedynczych znaków	132
Najważniejsze metody klasy String	132
5.3. Arytmetyka dużych liczb	132
5.4. Arytmetyka liczb zespolonych	134
5.5. System plików i rejestr	134
Pliki i katalogi	135
Strumienie	137
Czytelnicy i pisarze	138
Asynchroniczne operacje wejścia/wyjścia	139
Kompresja	139
Rejestr	140
5.6. Tworzenie bibliotek	141
5.7. Procesy i wątki	142
Procesy	142
Wątki	143
5.8. Obsługa błędów	146
Podsumowanie	147
Rozdział 6. Tworzenie interfejsu graficznego aplikacji	149
6.1. Projektowanie interfejsu graficznego	149
6.2. Wejście klawiatury	150
6.3. Wejście myszy	151
6.4. Symulowanie klawiatury i myszy	151
Symulowanie klawiatury	152
Symulowanie myszy	152
6.5. Przeciągnij i upuść	153
6.6. Przegląd wybranych kontrolki	153
6.7. Wstęp do Windows Presentation Foundation	155
Tworzenie projektu WPF	155
Przykład: „Witaj, świecie WPF!”	156
Rozdział 7. Podstawy programowania sieciowego	159
7.1. System DNS	159
7.2. Wysyłanie wiadomości e-mail	160
7.3. Protokół FTP	161
Przykład: Jak wysłać plik na serwer FTP?	161
7.4. Gniazda (ang. Sockets)	161

Rozdział 8. Asembler IL	165
8.1. Co to jest?	165
8.2. Program „Witaj, świecie!”	165
8.3. Kompilacja i uruchamianie	166
8.4. Zmienne lokalne	166
8.5. Metody	167
8.6. Rozgałęzienia	169
8.7. Pętle	170
8.8. Przegląd wybranych instrukcji	171
Instrukcje odkładające wartość na stos	171
Instrukcje zdejmujące wartość ze stosu	172
Instrukcje rozgałęzień	172
Instrukcje arytmetyczne	173
Pozostałe instrukcje	173
Rozdział 9. Podstawy tworzenia aplikacji w stylu Metro dla Windows 8	175
9.1. Co to są aplikacje Metro?	175
9.2. Potrzebne narzędzia	176
9.3. Uzyskiwanie licencji dewelopera	176
9.4. Program „Witaj, świecie Metro!”	177
Tworzenie nowego projektu	177
Zmodyfikuj stronę startową	177
Dodaj obsługę zdarzeń	178
Uruchom aplikację	178
9.5. Przegląd wybranych kontroltek	178
App bar	178
Button	178
Check box	179
Combo box	179
Grid view	179
Hyperlink	179
List box	180
List view	180
Password box	181
Progress bar	181
Progress ring	181
Radio button	181
Slider	182
Text block	182
Text box	182
Toggle switch	182
Tooltip	183
Dodatek A Słowa kluczowe języka C#	185
Dodatek B Zestaw instrukcji Asemblera IL	187
Operacje arytmetyczne	187
Dodawanie	187
Odejmowanie	187
Mnożenie	187
Dzielenie	188
Modulo	188
Wartość negatywna	188
Operacje bitowe	188
Koniunkcja	188

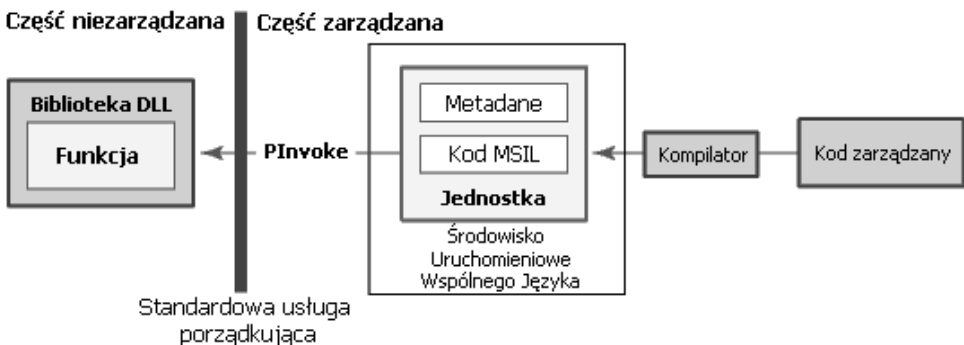
Alternatywa	188
Negacja	188
Alternatywa wykluczająca	188
Przesunięcie bitowe w prawo	188
Przesunięcie bitowe w lewo	188
Operacje odkładania na stos	189
Operacje zdejmowania ze stosu i zapisywania	190
Konwersje	191
Porównywanie	191
Skoki bezwarunkowe	192
Skoki warunkowe	192
Wywoływanie metod i powrót	192
Opakowywanie	192
Wyjątki	193
Bloki pamięci	193
Wskaźniki	193
Pozostałe	193
Skorowidz	195

Rozdział 5.

Język C#. Pozostałe zagadnienia

5.1. Wywoływanie funkcji przez PInvoke

PInvoke to skrót od *Platform Invocation Services*. Usługi te pozwalają z zarządzanego kodu wywoływać niezarządzane funkcje zaimplementowane w bibliotece DLL. PInvoke polega na metadanych, aby zlokalizować eksportowane funkcje i uporządkować ich argumenty w czasie działania programu. Proces ten został przedstawiony na rysunku 5.1.



Rysunek 5.1. Wywoływanie funkcji poprzez PInvoke

Proces ten można też scharakteryzować następującą listą kroków:

1. Zlokalizowanie biblioteki DLL, która zawiera funkcję do wywołania.
2. Załadowanie biblioteki DLL do pamięci.

3. Zlokalizowanie adresu funkcji w pamięci i odłożenie jej argumentów na stos, a także porządkowanie ich, gdy zachodzi taka potrzeba.
4. Przekazanie kontroli do funkcji niezarządzanej.



Wskazówka

Lokalizowanie i ładowanie biblioteki DLL do pamięci oraz lokalizowanie adresu funkcji następuje tylko przy jej pierwszym wywołaniu.

PInvoke rzuca wyjątki (spowodowane przez niezarządzane funkcje) do wywołującego zarządzanego.

Aby zadeklarować metodę, której implementację eksportuje biblioteka DLL, wykonaj następujące kroki:

- ◆ Zadeklaruj metodę ze słowami kluczowymi `static` i `extern`.
- ◆ Załącz atrybut `DllImport` do metody. Atrybut ten pozwala określić nazwę biblioteki DLL zawierającej metodę. Jest taki zwyczaj, że metodę w kodzie C# nazywa się tak samo jak metodę eksportowaną, można jednak dać jej również inną nazwę.
- ◆ Opcjonalnie określ informacje porządkujące dla parametrów metody i wartości zwracanej.

Poniższy przykład prezentuje, jak użyć atrybutu `DllImport` do wyświetlenia wiadomości poprzez wywołanie metody `puts` z biblioteki `msvcrt.dll`.

```
using System;
using System.Runtime.InteropServices;

class Program
{
    [DllImport("msvcrt.dll")] //zaimportowanie biblioteki DLL
    public static extern int puts(string c); //deklaracja funkcji z biblioteki
    [DllImport("msvcrt.dll")] //zaimportowanie biblioteki DLL
    internal static extern int _flushall(); //deklaracja funkcji z biblioteki
    static void Main()
    {
        puts("Witaj!"); //Wypisze na konsoli napis: Witaj!
        _flushall();
    }
}
```

Być może teraz w Twojej głowie pojawiło się pytanie, skąd brać sygnatury, które pojawiły się w powyższym kodzie? Polecam Ci serwis <http://www.pinvoke.net/>. Jest tam bardzo dużo materiałów o usługach PInvoke oraz sygnatury pogrupowane według nazw bibliotek.

5.2. Napisy (ang. Strings)

Napis jest obiektem typu `String`, którego wartością jest tekst. Patrząc od wewnątrz, moglibyśmy zobaczyć, że tekst w napisie jest przechowywany jako kolekcja obiektów typu `Char` (kolekcja tylko do odczytu). W języku C# napis nie kończy się znakiem zerowym. Właściwość `Length` napisu reprezentuje liczbę obiektów `Char`, które zawiera, a nie liczbę znaków Unicode.

Słowo kluczowe `string` jest aliasem dla klasy `System.String`. Wynika z tego, że `string` i `String` są swoimi odpowiednikami i możesz używać takiej nazwy, jaką preferujesz. Klasa `String` dostarcza wiele metod do bezpiecznego tworzenia, porównywania oraz do innych czynności, jakie można wykonać na napisie. Dodatkowo język C# przeciąża niektóre operatory, aby można było ich użyć do różnych operacji na napisach.

Deklaracja i inicjalizacja

Zadeklarować i zainicjalizować napis można na różne sposoby:

```
//deklaracja bez inicjalizacji
string message1;

//inicjalizacja za pomocą wartości null
string message2 = null;

//inicjalizacja napisu jako pustego
string message3 = System.String.Empty;

//inicjalizacja napisu zwykłym tekstem
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

//inicjalizacja napisu stałą napisową (zauważ znak @ przed napisem)
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

//można użyć pełnej nazwy, czyli System.String zamiast String
System.String greeting = "Hello World!";

//zmienna lokalna (np. wewnątrz metody)
var temp = "Nadal jestem silnie typowanym napisem!";

//napis, w którym nie można przechować innego tekstu
const string message4 = "Nie można się mnie pozbyć!";

//Używaj konstruktora String tylko wtedy, gdy
//tworzysz napis z char*, char[] lub sbyte*.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Inicjalizując napisy wartością `Empty` zamiast `null`, zredukujesz szanse na otrzymanie wyjątku `NullReferenceException`. Używaj statycznej metody `IsNullOrEmpty(String)`, aby sprawdzić wartość napisu przed dostępem do niego.

Niezmienność obiektów String

Obiekty String są *niezmiennie*: nie mogą być zmienione po ich utworzeniu. Może się wydawać, że metody i operatory działające na napisach mogą zmieniać ich wartość. W każdym z tych przypadków napis nie jest modyfikowany, zatem jest tworzony nowy obiekt String. W przykładzie poniżej, gdy wartości s1 i s2 są łączone, powstaje nowy napis, a dwa oryginalne napisy pozostają niezmienione. Nowy obiekt jest przypisywany do zmiennej s1, a oryginalny obiekt, który był wcześniej do niej przypisany, zostaje zwolniony przez odśmiecaacz pamięci (ang. *Garbage Collector*), ponieważ żadna zmienna nie przechowuje do niego referencji.

```
string s1 = "Napis to więcej ";
string s2 = "niż znaki, jakie zawiera.";

//Połączenie s1 i s2 tworzy nowy obiekt
//i zachowuje go w s1, zwalniając
//referencję do oryginalnego obiektu
s1 += s2;
System.Console.WriteLine(s1); //Wypisze: Napis to więcej niż znaki, jakie zawiera
```

Ponieważ „modyfikacja” napisu polega na utworzeniu nowego napisu, musisz uważać, gdy stworzysz do nich referencje. Gdy stworzysz referencję do napisu, a następnie „zmodyfikujesz” oryginalny napis, to referencja nadal będzie wskazywać na oryginalny obiekt. Ilustruje to następujący przykład:

```
string s1 = "Witaj ";
string s2 = s1;
s1 += "świecie!";
System.Console.WriteLine(s2); //Wypisze: Witaj
```

Znaki specjalne

W napisach można używać znaków specjalnych, takich jak znak nowej linii czy tabulator. Znaki te przedstawia tabela 5.1.

Tabela 5.1. Znaki specjalne w języku C#

Znak specjalny	Nazwa	Kodowanie Unicode
\'	Apostrof	0x0027
\"	Cudzysłów	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	Nowa linia	0x000A
\r	Powrót karetki	0x000D

Tabela 5.1. Znaki specjalne w języku C# — ciąg dalszy

Znak specjalny	Nazwa	Kodowanie Unicode
\t	Tabulator poziomy	0x0009
\U	Sekwencja 8 cyfr heksadecymalnych oznaczających kod znaku Unicode	\Unnnnnnnn
\u	Sekwencja 4 cyfr heksadecymalnych oznaczających kod znaku Unicode	\u0041 = "A"
\v	Tabulator pionowy	0x000B
\x	Sekwencja od 1 do 4 cyfr heksadecymalnych oznaczających kod znaku Unicode	\x0041 = "A"

Formatowanie napisów

Napis formatowany to napis, którego zawartość określana jest dynamicznie (w czasie działania programu). Tworzy się go za pomocą statycznej metody `Format`.

Oto przykład:

```
//Pobierz dane od użytkownika
System.Console.WriteLine("Enter a number");
string input = System.Console.ReadLine();

//konwersja napisu (string) na liczbę całkowitą (int)
int j;
System.Int32.TryParse(input, out j);

//wypisanie różnych napisów podczas każdej iteracji pętli

string s;
for (int i = 0; i < 10; i++)
{
    s = System.String.Format("{0} * {1} = {2}", i, j, (i * j));
    System.Console.WriteLine(s);
}
```

Napisy częściowe

Napis częściowy (ang. *substring*) to sekwencja znaków określonej długości zawarta w napisie. Metoda `IndexOf` pozwala wyszukać części napisów. Metoda `Replace` zamienia wszystkie wystąpienia napisów częściowych w napisie.

Oto przykład:

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2)); //Wypisze: C#

System.Console.WriteLine(s3.Replace("C#", "Basic")); //Wypisze: Visual Basic Express

//Indeksy numerowane są od zera
int index = s3.IndexOf("C"); //indeks = 7
```

Dostęp do pojedynczych znaków

Do poszczególnych znaków napisu możesz uzyskać dostęp tak jak do tablicy, co prezentuje poniższy przykład. Oczywiście dostęp tylko do odczytu.

```
string s5 = "Piszę wstecz";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
//Wypisze: zcetsw ęzsiP
```

Najważniejsze metody klasy String

- ◆ `Concat(String, String)` — łączy dwa napisy w jeden.
- ◆ `Contains(String)` — sprawdza, czy podany tekst występuje w napisie.
- ◆ `Insert(Int32, String)` — wstawia tekst do napisu na podaną pozycję.
- ◆ `IsNullOrEmpty(String)` — sprawdza, czy napis jest pusty lub ma wartość `null`.
- ◆ `Replace(String, String)` — zastępuje wystąpienia napisu z drugiego parametru w napisie podanym jako pierwszy parametr (zwraca nowy napis).
- ◆ `ToLower()` — zamienia litery w napisie na małe (zwraca nowy napis).
- ◆ `ToUpper()` — zamienia litery w napisie na duże (zwraca nowy napis).

5.3. Arytmetyka dużych liczb

Na początek pragnę wspomnieć o strukturach `Int64` oraz `UInt64`. Pierwsza struktura reprezentuje 64-bitową liczbę całkowitą ze znakiem, a druga — bez znaku. Maksymalna wartość dla `Int64` wynosi 9 223 372 036 854 775 807, a dla `UInt64` jest to 18 446 744 073 709 551 615. Nie napisałem o tym w rozdziale 3., więc postanowiłem napisać tutaj. Jednak nie to jest głównym tematem tego rozdziału. Tym, czym się za chwilę zajmiemy, będą naprawdę duże liczby, a chodzi dokładnie o strukturę `BigInteger`.

W przestrzeni nazw `System.Numerics` znajduje się struktura `BigInteger`, która wspomaga operacje na dużych liczbach. Zawiera ona właściwości, metody i operatory do działania na takich liczbach.

Najpierw musimy dodać referencję do jednostki `System.Numerics.dll`. W okienku *Solution Explorer* kliknij prawym przyciskiem myszy *References* i wybierz *Add Reference...*, odnajdź na liście nazwę `System.Numerics`, zaznacz ją i kliknij przycisk *OK*.

Teraz na górze swojego programu dopisz linijkę:

```
using System.Numerics;
```

Od teraz możesz korzystać z całej przestrzeni nazw `System.Numerics`.

Nową zmienną typu `BigInteger` możemy utworzyć następująco:

```
//utworzenie zmiennej i zainicjalizowanie ją domyślną wartością (zero)
BigInteger a = new BigInteger();
```

```
//utworzenie zmiennej i zainicjalizowanie ją podaną wartością
BigInteger b = new BigInteger(18446744073709551615);
```

Struktura `BigInteger` zawiera przeciążone operatory, takie jak na przykład `+` (dodawanie), `-` (odejmowanie), `*` (mnożenie) i `/` (dzielenie), co ułatwia wykonywanie podstawowych operacji na dużych liczbach.

Oto prosty kalkulator z użyciem `BigInteger`:

```
using System;
using System.Numerics;

class Program
{
    static void Main()
    {
        Console.WriteLine("Podaj pierwszą liczbę:");
        string strA = Console.ReadLine(); //Wczytaj linię tekstu z konsoli
        Console.WriteLine("Podaj drugą liczbę:");
        string strB = Console.ReadLine(); //Wczytaj linię tekstu z konsoli

        BigInteger a = BigInteger.Parse(strA); //Przekonwertuj napis na BigInteger
        BigInteger b = BigInteger.Parse(strB); //Przekonwertuj napis na BigInteger

        Console.WriteLine("Wyniki:");
        Console.WriteLine("{0} + {1} = {2}", a, b, a + b);
        Console.WriteLine("{0} - {1} = {2}", a, b, a - b);
        Console.WriteLine("{0} * {1} = {2}", a, b, a * b);
        Console.WriteLine("{0} / {1} = {2}", a, b, a / b);
    }
}
```

Przykładowe wyjście programu:

```
Podaj pierwszą liczbę:
18446744073709551615
Podaj drugą liczbę:
64
Wyniki:
18446744073709551615 + 64 = 18446744073709551679
18446744073709551615 - 64 = 18446744073709551551
18446744073709551615 * 64 = 1180591620717411303360
18446744073709551615 / 64 = 288230376151711743
```

5.4. Arytmetyka liczb zespolonych

W znanej nam już przestrzeni nazw `System.Numerics` znajduje się struktura `Complex`. Reprezentuje ona liczbę zespoloną. Liczba zespolona składa się z części rzeczywistej i części urojonej.

Nową instancję struktury `Complex` tworzymy w następujący sposób:

```
Complex complex1 = new Complex(17.34, 12.87);
Complex complex2 = new Complex(8.76, 5.19);
```

gdzie pierwsza wartość to część rzeczywista, a druga to część urojona.

Na liczbach zespolonych możemy wykonywać różne operacje. Struktura `Complex` z przestrzeni nazw `System.Numerics` udostępnia naprawdę wiele przydatnych metod, właściwości i operatorów.

Oto przykładowy kalkulator działający na liczbach zespolonych:

```
using System;
using System.Numerics;

class Program
{
    static void Main()
    {
        Complex complex1 = new Complex(17.34, 12.87);
        Complex complex2 = new Complex(8.76, 5.19);

        Console.WriteLine("{0} + {1} = {2}", complex1, complex2,
            complex1 + complex2);
        Console.WriteLine("{0} - {1} = {2}", complex1, complex2,
            complex1 - complex2);
        Console.WriteLine("{0} * {1} = {2}", complex1, complex2,
            complex1 * complex2);
        Console.WriteLine("{0} / {1} = {2}", complex1, complex2,
            complex1 / complex2);
    }
}
```

Wyjście programu:

```
(17.34, 12.87) + (8.76, 5.19) = (26.1, 18.06)
(17.34, 12.87) - (8.76, 5.19) = (8.58, 7.68)
(17.34, 12.87) * (8.76, 5.19) = (85.1031, 202.7358)
(17.34, 12.87) / (8.76, 5.19) = (2.10944241403558, 0.219405693054265)
```

5.5. System plików i rejestr

Operacje wejścia/wyjścia na plikach i strumieniach odnoszą się do transferu danych do i z nośnika danych. Przestrzenie nazw `System.IO` zawierają typy, które pozwalają czytać i pisać do strumieni danych i plików. Te przestrzenie nazw zawierają również

typy pozwalające na kompresję i dekompresję plików oraz typy pozwalające na komunikację poprzez łącza i porty.

Plik jest uporządkowaną i nazwaną kolekcją bajtów. Gdy pracujesz z plikami, pracujesz również ze ścieżkami dostępu, pamięcią oraz nazwami plików i katalogów.

Pliki i katalogi

Możesz używać typów z przestrzeni System.IO do interakcji z plikami i katalogami. Bazując na przykład na określonych kryteriach wyszukiwania, możesz pobrać i ustawić właściwości plików i katalogów, a także pobrać kolekcję plików i katalogów.

Poniżej lista często używanych klas do pracy z plikami i katalogami:

- ♦ `File` — dostarcza statyczne metody do tworzenia, kopiowania, usuwania, przenoszenia i otwierania plików.
- ♦ `FileInfo` — dostarcza metody instancji do tworzenia, kopiowania, usuwania, przenoszenia i otwierania plików.
- ♦ `Directory` — dostarcza statyczne metody do tworzenia, przenoszenia oraz wyliczania katalogów i podkatalogów.
- ♦ `DirectoryInfo` — dostarcza metody instancji do tworzenia, przenoszenia i wyliczania katalogów i podkatalogów.
- ♦ `Path` — dostarcza metody i właściwości do przetwarzania ścieżek dostępu do plików i katalogów.

Przykład: Jak kopiować katalogi?

```
using System;
using System.IO;

class DirectoryCopyExample
{
    static void Main()
    {
        //Kopiuje katalog cat1 do katalogu cat2 wraz z podkatalogami i plikami
        DirectoryCopy(@"C:\cat1", @"C:\cat2", true);
    }

    private static void DirectoryCopy(string sourceDirName, string destDirName,
        ↪bool copySubDirs)
    {
        DirectoryInfo dir = new DirectoryInfo(sourceDirName);
        DirectoryInfo[] dirs = dir.GetDirectories();

        if (!dir.Exists)
        {
            throw new DirectoryNotFoundException(
                "Katalog źródłowy nie istnieje lub nie może zostać odnaleziony: "
                + sourceDirName);
        }
    }
}
```

```

        if (!Directory.Exists(destDirName))
        {
            Directory.CreateDirectory(destDirName);
        }

        FileInfo[] files = dir.GetFiles();
        foreach (FileInfo file in files)
        {
            string tempPath = Path.Combine(destDirName, file.Name);
            file.CopyTo(tempPath, false);
        }

        if (copySubDirs)
        {
            foreach (DirectoryInfo subdir in dirs)
            {
                string tempPath = Path.Combine(destDirName, subdir.Name);
                DirectoryCopy(subdir.FullName, tempPath, copySubDirs);
            }
        }
    }
}

```

Przykład: Jak wylistować pliki w katalogu?

```

using System;
using System.IO;

public class DirectoryLister
{
    public static void Main(String[] args)
    {
        string path = Environment.CurrentDirectory;
        if (args.Length > 0)
        {
            if (Directory.Exists(args[0]))
            {
                path = args[0];
            }
            else
            {
                Console.WriteLine("{0} nie znaleziono; używając katalogu bieżącego:",
                    args[0]);
            }
        }
        DirectoryInfo dir = new DirectoryInfo(path);
        Console.WriteLine("Rozmiar \tData utworzenia \tNazwa");
        foreach (FileInfo f in dir.GetFiles("*.exe"))
        {
            string name = f.Name;
            long size = f.Length;
            DateTime creationTime = f.CreationTime;
            Console.WriteLine("{0,-12:N0} \t{1,-20:g} \t{2}", size,
                creationTime, name);
        }
    }
}

```

Przykład: Jak wylistować katalogi w podanej ścieżce?

```
using System;
using System.Collections.Generic;
using System.IO;

class Program
{
    private static void Main(string[] args)
    {
        try
        {
            string dirPath = @"C:\"; //katalog do przeszukania

            List<string> dirs = new List<string>(Directory.
                ↪EnumerateDirectories(dirPath));

            foreach (var dir in dirs)
            {
                Console.WriteLine("{0}", dir.Substring(dir.LastIndexOf("\\") + 1));
            }
            Console.WriteLine("W sumie znaleziono {0} katalogów.", dirs.Count);
        }
        catch (UnauthorizedAccessException UAEx)
        {
            Console.WriteLine(UAEx.Message);
        }
        catch (PathTooLongException PathEx)
        {
            Console.WriteLine(PathEx.Message);
        }
    }
}
```

Strumienie

Abstrakcyjna klasa bazowa `Stream` wspiera odczyt i zapis bajtów. Wszystkie klasy reprezentujące strumienie dziedziczą z klasy `Stream`.

Na strumieniach możemy wykonywać trzy fundamentalne operacje:

- ♦ **Czytanie** — transfer danych ze strumienia do struktury danych, takiej jak np. tablica bajtów.
- ♦ **Pisanie** — transfer danych do strumienia z określonego źródła.
- ♦ **Szukanie** — pobieranie i modyfikowanie bieżącej pozycji w strumieniu.

Poniżej lista często używanych klas do pracy ze strumieniami:

- ♦ `FileStream` — do czytania z pliku i pisania do pliku.
- ♦ `MemoryStream` — do czytania z pamięci i pisania do pamięci.
- ♦ `BufferedStream` — do zwiększenia wydajności operacji odczytu i zapisu.

- ◆ `NetworkStream` — do czytania i pisania poprzez gniazda sieciowe.
- ◆ `PipeStream` — do czytania i pisania poprzez łącza nienazwane i nazwane.
- ◆ `CryptoStream` — do łączenia strumieni danych z transformacjami kryptograficznymi.

Czytelnicy i pisarze

Przestrzeń nazw `System.IO` dostarcza typów do czytania znaków ze strumieni i zapisu ich do strumieni. Domyślnie strumienie stworzone są do pracy z bajtami. Typy czytelników i pisarzy obsługują konwersje znaków na bajty i bajtów na znaki.

Poniżej często używane klasy czytelników i pisarzy:

- ◆ `BinaryReader` i `BinaryWriter` — do odczytu i zapisu prymitywnych danych jako wartości binarnych.
- ◆ `StreamReader` i `StreamWriter` — do odczytu i zapisu znaków (z obsługą konwersji znaków na bajty i odwrotnie).
- ◆ `StringReader` i `StringWriter` — do odczytu i zapisu znaków do napisu (`String`) i z napisu (`String`).
- ◆ `TextReader` i `TextWriter` — klasy bazowe dla innych czytelników i pisarzy do zapisu oraz odczytu znaków i napisów, ale nie danych binarnych.

Przykład: Jak odczytać dane z pliku tekstowego?

```
using System;
using System.IO;

class Test
{
    public static void Main()
    {
        try
        {
            using (StreamReader sr = new StreamReader("TestFile.txt"))
            {
                String line = sr.ReadToEnd(); //Czytaj plik do końca
                Console.WriteLine(line); //Wyświetl zawartość na ekranie
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Nie można odczytać danych z pliku:");
            Console.WriteLine(e.Message);
        }
    }
}
```

Przykład: Jak zapisać dane do pliku tekstowego?

```
using System;
using System.IO;

class Program
{
    private static void Main(string[] args)
    {
        string str = String.Empty; //Utwórz pusty napis
        str = Console.ReadLine(); //Pobierz tekst z konsoli i zapisz do zmiennej
        //Utwórz Pisarza
        using (StreamWriter outfile = new StreamWriter(@"C:\file1.txt"))
        {
            outfile.Write(str); //Zapisz dane ze zmiennej str do pliku
        }
    }
}
```

Asynchroniczne operacje wejścia/wyjścia

Odczyt i zapis dużej ilości danych może powodować większe użycie zasobów. Powinieneś wykonywać takie czynności asynchronicznie, aby aplikacja mogła odpowiadać użytkownikowi. Przy synchronicznych operacjach wejścia/wyjścia wątek obsługi interfejsu użytkownika jest blokowany, dopóki nie skończą się te operacje.

Składowe asynchroniczne zawierają w swojej nazwie słowo Async, np. CopyToAsync, FlushAsync, ReadAsync czy WriteAsync. Używaj tych metod w połączeniu ze słowami kluczowymi async i await.

Kompresja

Kompresja polega na zmniejszaniu rozmiaru pliku. Dekompresja to proces wypakowywania zawartości skompresowanego pliku, aby można było użyć tej zawartości.

Przykład: Jak kompresować i wypakowywać dane w formacie ZIP?

```
using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string startPath = @"c:\example\start";
            string zipPath = @"c:\example\result.zip";
            string extractPath = @"c:\example\extract";

            ZipFile.CreateFromDirectory(startPath, zipPath);
        }
    }
}
```

```

        ZipFile.ExtractToDirectory(zipPath, extractPath);
    }
}

```



Aby użyć klasy ZipFile, musisz dodać referencję do jednostki System.IO.Compression.
 ↪ FileSystem w swoim projekcie.

Rejestr

W rejestrze możemy przechowywać dane dotyczące stworzonej przez nas aplikacji, takie jak na przykład jej informacje o konfiguracji.

Przykład: Jak stworzyć klucz w rejestrze?

```

using System;
using System.IO;
using Microsoft.Win32;

class Program
{
    static void Main(string[] args)
    {
        const string userRoot = "HKEY_CURRENT_USER";
        const string subkey = "Imiona";
        const string keyName = userRoot + "\\\" + subkey;

        Registry.SetValue(keyName, "Imię", "Izabela");
    }
}

```

Przykład: Jak odczytać wartość klucza z rejestru?

```

using System;
using System.IO;
using Microsoft.Win32;

class Program
{
    static void Main(string[] args)
    {
        const string userRoot = "HKEY_CURRENT_USER";
        const string subkey = "Imiona";
        const string keyName = userRoot + "\\\" + subkey;

        string str = (string) Registry.GetValue(keyName, "Imię", "brak");
        Console.WriteLine("{0}", str);
    }
}

```



Bezpieczniej jest zapisywać dane do klucza głównego HKEY_CURRENT_USER zamiast do klucza HKEY_LOCAL_MACHINE.

5.6. Tworzenie bibliotek

Bibliotekę stworzymy, wybierając typ nowego projektu o nazwie Class Library. Dla przykładu stworzymy bibliotekę DLL z metodami do prostych operacji arytmetycznych, takich jak dodawanie, odejmowanie, mnożenie i dzielenie dwóch podanych liczb.

Oto kod biblioteki:

```
using System;

public class Algebra
{
    public double Addition(double x = 0, double y = 0)
    {
        return x + y;
    }

    public double Subtraction(double x = 0, double y = 0)
    {
        return x - y;
    }

    public double Multiplication(double x = 0, double y = 0)
    {
        return x * y;
    }

    public double Division(double x = 0, double y = 1)
    {
        return x / y;
    }
}
```

Gdy skompilujesz ten kod, w folderze z projektem biblioteki pojawi się plik o rozszerzeniu *.dll, czyli gotowa biblioteka.

Teraz stwórz nowy projekt aplikacji konsolowej, a następnie w oknie *Solution Explorer* kliknij prawym przyciskiem *References* i wybierz *Add reference*. Przejdź do zakładki *Browse* i wybierz plik *Algebra.dll*.

W stworzonym projekcie wpisz poniższy kod:

```
using System;
using System.IO;

class Program
{
```

```

private static void Main(string[] args)
{
    Algebra alg1 = new Algebra(); //Utwórz obiekt klasy Algebra
    double a = 24.5;
    double b = 4.25;
    double c = alg1.Addition(a, b); //metoda Addition z klasy Algebra
    Console.WriteLine("{0} + {1} = {2}", a, b, c);
    //Wypisze: 24.5 + 4.25 = 28.75
}
}

```

Skompiluj powyższy projekt i uruchom. Jest to projekt z użyciem Twojej własnej biblioteki. To wszystko w tym rozdziale.

5.7. Procesy i wątki

Procesy

Klasa `Process` z przestrzeni nazw `System.Diagnostics` daje dostęp do lokalnych i zdalnych procesów oraz pozwala uruchamiać i zatrzymywać procesy na lokalnym systemie.

Przykład pokazujący różne sposoby uruchamiania procesów:

```

using System;
using System.Diagnostics;

namespace MyProcessSample
{
    class MyProcess
    {
        //otwieranie aplikacji
        void OpenApplication(string myFavoritesPath)
        {
            //Uruchom przeglądarkę Internet Explorer
            Process.Start("IExplore.exe");

            //Wyświetl zawartość folderu Ulubione
            Process.Start(myFavoritesPath);
        }

        //otwieranie stron i plików HTML przy użyciu przeglądarki Internet Explorer
        void OpenWithArguments()
        {
            //przekazanie adresu strony jako argumentu dla procesu
            Process.Start("IExplore.exe", "www.helion.pl");

            //otwieranie pliku HTML w przeglądarce Internet Explorer
            Process.Start("IExplore.exe", "C:\\index.html");
        }

        //użycie klasy ProcessStartInfo do uruchomienia procesu zminimalizowanego
        void OpenWithStartInfo()
    }
}

```



```
        {
            ProcessStartInfo startInfo = new ProcessStartInfo("IEExplore.exe");
            startInfo.WindowStyle = ProcessWindowStyle.Minimized;

            Process.Start(startInfo);

            startInfo.Arguments = "www.helion.pl";

            Process.Start(startInfo);
        }

    static void Main()
    {
        //Pobierz ścieżkę do folderu Ulubione
        string myFavoritesPath =
            Environment.GetFolderPath(Environment.SpecialFolder.Favorites);

        MyProcess myProcess = new MyProcess();

        myProcess.OpenApplication(myFavoritesPath);
        myProcess.OpenWithArguments();
        myProcess.OpenWithStartInfo();
    }
}
```

Z klasy `Process` dwie metody są najważniejsze do zapamiętania, pierwsza to metoda `Start`, która uruchamia proces, a druga to metoda `Kill`, która zabija określony proces. Więcej o procesach możesz znaleźć na stronach MSDN.

Wątki

Wątki pozwalają na wykonywanie kilku zadań „jednocześnie”. Dlaczego słowo „jednocześnie” jest w cudzysłowie? Chodzi o to, że te zadania wykonują się na przemian, a dzieje się to tak szybko, że mamy wrażenie, iż dzieje się to równocześnie. Najpierw wykona się część pierwszego zadania, potem następuje przełączenie na drugie zadanie, które też się wykona częściowo, następnie znów się wykonuje pierwsze zadanie — i tak w kółko.

Poniższy przykład tworzy klasę o nazwie `Worker` z metodą `DoWork`, która zostanie wywołana w osobnym wątku. Wątek rozpocznie wykonywanie przy wywołaniu metody i zakończy pracę automatycznie. Metoda `DoWork` wygląda tak:

```
public void DoWork()
{
    while (!_shouldStop)
    {
        Console.WriteLine("worker thread: working...");
    }
    Console.WriteLine("worker thread: terminating gracefully.");
}
```

Klasa `Worker` zawiera dodatkową metodę, która określa, kiedy metoda `DoWork` powinna się zakończyć. Metoda ta nazywa się `RequestStop` i wygląda następująco:

```
public void RequestStop()
{
    _shouldStop = true;
}
```

Metoda `RequestStop` przypisuje składowej `_shouldStop` wartość `true`. Ponieważ składowa ta jest sprawdzana przez metodę `DoWork`, przypisanie jej wartości `true` spowoduje zakończenie pracy wątku z metodą `DoWork`. Warto zwrócić uwagę, że metody `DoWork` i `RequestStop` będą uruchamiane przez osobne wątki, dlatego składowa `_shouldStop` powinna zostać zadeklarowana ze słowem kluczowym `volatile`.

```
private volatile bool _shouldStop;
```

Słowo kluczowe `volatile` informuje kompilator, że dwa lub więcej wątków będzie miało dostęp do tej składowej i kompilator nie powinien używać tutaj żadnej optymalizacji.

Użycie słowa kluczowego `volatile` przy składowej `_shouldStop` pozwala mieć do niej bezpieczny dostęp z kilku wątków bez użycia jakichkolwiek technik synchronizacji. Jest tak tylko dlatego, że typ tej składowej to `bool`. Oznacza to, że tylko pojedyncze operacje są używane do modyfikacji tej składowej.

Przed tworzeniem wątku funkcja `Main` tworzy obiekt typu `Worker` i instancję klasy `Thread`. Obiekt wątku jest konfigurowany do użycia metody `Worker.DoWork` jako punkt wejścia poprzez przekazanie referencji tej metody do konstruktora.

```
Worker workerObject = new Worker();
Thread workerThread = new Thread(workerObject.DoWork);
```

W tym momencie obiekt wątku istnieje i jest skonfigurowany. Jednak wątek jeszcze nie działa, zostanie on uruchomiony, gdy metoda `Main` wywoła metodę `Start` wątku:

```
workerThread.Start();
```

Po tym wątek już działa, ale jest to asynchroniczne względem wątku głównego. Oznacza to, że metoda `Main` wykonuje się dalej. Aby więc wątek nie został zakończony, musimy utworzyć pętlę, która będzie czekała na zakończenie wątku:

```
while (!workerThread.IsAlive);
```

Następnie wątek główny jest usypiany na określony czas poprzez wywołanie `Sleep`. Spowoduje to, że wątek roboczy wykona kilka iteracji pętli metody `DoWork`, zanim metoda `Main` wykona kolejne instrukcje.

```
Thread.Sleep(1);
```

Po upływie 1 milisekundy funkcja `Main` za pomocą metody `Worker.RequestStop` daje sygnał do wątku roboczego, że powinien się zakończyć.

```
workerObject.RequestStop();
```

Istnieje również możliwość zakończenia pracy wątku z innego wątku poprzez wywołanie `Abort`. Jednak użycie `Abort` powoduje, że wątek kończy się natychmiast, i to nieważne, czy ukończył swoje zadanie.

Ostatecznie funkcja `Main` wywołuje metodę `Join` na obiekcie wątku roboczego. Powoduje to oczekiwanie bieżącego wątku na zakończenie wątku roboczego. Metoda `Join` nie wróci, dopóki wątek roboczy sam się nie zakończy.

```
workerThread.Join();
```

W tym momencie główny wątek wywołujący funkcję `Main` istnieje. Wyświetla on komunikat i kończy działanie.

Cały przykład znajduje się poniżej:

```
using System;
using System.Threading;

public class Worker
{
    //Metoda ta zostanie wywołana, gdy wątek wystartuje
    public void DoWork()
    {
        while (!_shouldStop)
        {
            Console.WriteLine("wątek roboczy: pracuje...");
        }
        Console.WriteLine("wątek roboczy: zakończony.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    //Volatile informuje, że dostęp do tej składowej będzie z kilku wątków
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    static void Main()
    {
        //utworzenie obiektu wątku (wątek jeszcze nie wystartował)
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        //uruchomienie wątku roboczego
        workerThread.Start();
        Console.WriteLine("wątek główny: Uruchamiam wątek roboczy...");

        //pętla wstrzymująca, dopóki wątek roboczy jest aktywny
        while (!workerThread.IsAlive);

        //Uśpij wątek główny na 1ms, aby wątek roboczy miał czas
        //na wykonanie swoich operacji
        Thread.Sleep(1);
    }
}
```

```

//Wyślij żądanie do wątku roboczego, aby się zakończył
workerObject.RequestStop();

//zablokowanie wątku głównego, dopóki wątek roboczy nie zakończy swojej pracy
workerThread.Join();
Console.WriteLine("wątek główny: Wątek roboczy zakończył pracę.");
    }
}

```

5.8. Obsługa błędów

Obsługa błędów języka C# pozwala Ci kontrolować nieoczekiwane i wyjątkowe sytuacje, które mają miejsce podczas działania programu. Używane słowa kluczowe try, catch i finally pozwalają na przetestowanie sytuacji, które mogą zakończyć się niepowodzeniem. Wyjątki mogą być generowane przez Środowisko Uruchomieniowe Wspólnego Języka (ang. *Common Language Runtime*), .NET Framework oraz biblioteki. Wyjątki są tworzone za pomocą słowa kluczowego throw.

Poniższy przykład zawiera metodę, która dzieli dwie liczby. Gdy liczba, przez którą zostanie wykonane dzielenie, będzie zerem, złapany zostanie wyjątek. Bez obsługi błędów program ten zakończyłby się wyświetleniem komunikatu *DivideByZeroException was unhandled*.

```

using System;

class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new System.DivideByZeroException(); //Rzuc wyjątek DivideByZeroException
        return x / y;
    }
    static void Main()
    {
        double a = 98, b = 0;
        double result = 0;

        try //Spróbuj wykonać dzielenie
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} / {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException e) //Złap wyjątek, gdy wystąpił
        {
            Console.WriteLine("Próbujesz dzielić przez zero.");
        }
    }
}

```

Podsumowanie

- ♦ Wszystkie wyjątki pochodzą z `System.Exception`.
- ♦ Używaj bloku `try` w miejscach, w których mogą wystąpić wyjątki.
- ♦ Gdy w bloku `try` wystąpi wyjątek, kontrola jest natychmiast przekazywana do obsługi wyjątków. W języku C# słowo kluczowe `catch` jest używane do definiowania obsługi wyjątków.
- ♦ Gdy wystąpi wyjątek, który nie zostanie obsłużony, program zakończy swoje działanie, wyświetlając komunikat o błędzie.
- ♦ Gdy blok `catch` definiuje zmienną wyjątku, możesz jej użyć, aby uzyskać więcej informacji o wyjątku, który wystąpił.
- ♦ Kod w bloku `finally` jest wykonywany nawet wtedy, gdy wystąpił wyjątek. Użyj tego bloku do zwolnienia zasobów, na przykład zamknięcia strumieni lub plików, które zostały otwarte w bloku `try`.

Skorowidz

A

akcesor
 get, 88
 set, 88
alternatywa
 logiczna, 42
 warunkowa, 43
 wykluczająca logiczna, 42
aplikacje
 klient-serwer, 164
 konsolowe, 15
 Metro, 175
 z interfejsem WinForms, 15
 z interfejsem WPF, 15
argumenty wiersza poleceń, 52
assembler IL, 165

B

biblioteka mscorlib, 166
biblioteki DLL, 127, 141
błędy, 146
błędy procesu kompilacji, 15

C

cechy WPF, 155
CIL, Common Intermediate Language, 9, 165
CLR, Common Language Runtime, 146

D

deklaracja
 destruktora, 67
 klasy, 22, 63
 konstruktora, 66
 metody, 53

 metody statycznej, 76
 napisu, 129
 obiektu, 64
 pola statycznego, 76
 typu wskaźnikowego, 61
delegat Func, 97
delegaty, 23, 94
deserializacja, 78
destruktor, 66
diagram UML, 18
DNS, Domain Name System, 159
dostęp
 do wirtualnych składowych, 93
 do znaków, 132
duże liczby, 132
dyrektywa
 #define, 27
 #elif, 27
 #else, 27
 #endif, 27
 #endregion, 29
 #error, 28
 #if, 26
 #line, 28
 #pragma warning, 29
 #region, 29
 #undef, 28
 #warning, 28
 .assembly, 166
 .entrypoint, 166
 .maxstack, 167
dziedziczenie, 67

E

e-mail, 160

F

FIFO, first-in-first-out, 101
 format pliku, *Patrz* pliki
 formatowanie napisów, 130
 funkcja Main, 144

G

gniazda, Sockets, 161

I

indeksery, 86
 instrukcja
 call, 166
 if, 32
 ldstr, 166
 stloc, 172
 switch, 34
 instrukcje
 Asemblera IL, 187
 rozgałęzień, 169
 interfejs, 23, 83
 interfejs graficzny, 149

J

język pośredni CIL, 9, 165
 instrukcje arytmetyczne, 173
 instrukcje rozgałęzień, 172
 kompilacja programu, 166
 metody, 167
 odkładanie na stosie, 171
 pętle, 170
 rozgałęzienia, 169
 uruchamianie programu, 166
 zdejmowanie ze stosu, 172
 zmienne lokalne, 166

K

kafelki, 176
 kapsułkowanie, Encapsulating, 10
 klasa, 22
 ArrayList, 103
 DNS, 159
 generyczna
 Dictionary, 116
 KeyedCollection, 120
 LinkedList, 114
 List, 115
 Queue, 101, 112

 SortedDictionary, 118
 SortedList, 105, 123
 Stack, 102, 113
 Hashtable, 104
 ListDictionary, 105
 MailMessage, 160
 NameObjectCollectionBase, 107
 NameValueCollection, 110
 NetworkCredential, 160
 Object, 94
 Process, 142
 SntpClient, 160
 Stream, 137
 StringCollection, 103
 StringDictionary, 106
 System.Array, 57
 właściwość Length, 57
 właściwość Rank, 57
 System.Console, 25
 ZipFile, 140
 klasy
 abstrakcyjne, 77
 bazowe, 89
 czytelników i pisarzy, 138
 do pracy z plikami, 135
 do pracy ze strumieniami, 137
 kolekcji, 100
 pochodne, 88
 zagnieżdżone, 68
 zapieczętowane, 78
 klawiatura, 150
 klucz, 140
 kod nienadzorowany, unsafe code, 60
 kolekcje, 99
 komentarz
 blokowy, 20
 liniowy, 20
 XML, 20
 kompilator, 9
 kompilator ilasm.exe, 165
 kompresja, 139
 koniunkcja
 logiczna, 42
 warunkowa, 43
 konstruktor, 66
 kontrawariancja, 125
 kontrolka, 153
 App bar, 178
 Button, 178
 Check box, 179
 Combo box, 179
 Grid view, 179
 Hyperlink, 179
 List box, 180

- List view, 180
- Password box, 181
- Progress bar, 181
- Progress ring, 181
- Radio button, 181
- Slider, 182
- Text block, 182
- Text box, 182
- Toggle switch, 182
- Tooltip, 183

- konwersja typów, 24

- kopia obiektu
 - głęboka, 85
 - plytką, 85

- kopiowanie katalogów, 135

- kowariancja, 125

L

- licencja dewelopera, 176

- liczby zespolone, 134

- LIFO, last-in-first-out, 102

- listowanie

- katalogów, 137

- plików, 136

M

- metoda, 53

- add(), 31

- BinarySearch(), 57

- Clear(), 58

- Clone(), 58, 85

- Console.Read(), 26

- Console.ReadKey(), 26

- Console.ReadLine(), 26

- Copy(), 58

- Find(), 58

- FindAll(), 59

- IndexOf(), 59

- Initialize(), 59

- Kill, 143

- Resize(), 60

- Reverse(), 60

- Sort(), 60

- Start, 143

- ToString(), 94

- metody

- abstrakcyjne, 77

- anonimowe, 95

- klasy String, 132

- klasy System.Array, 57

- rozszerzające, 98

- statyczne, 76

- modyfikator

- internal, 70

- private, 69

- protected, 70

- protected internal, 70

- public, 69

- modyfikatory dostępu, 69

N

- napisy, Strings, 23, 129

- napisy częściowe, Substrings, 131

- narzędzia deweloperskie, 176

- niezmiennosc obiektów String, 130

O

- obiekt, 23

- obiekty String, 130

- obsługa

- błędów, 146

- zdarzeń, 178

- odczyt z pliku, 79

- odczyt z pliku tekstowego, 138

- odpluskwiacz, Debugger, 10

- odśmiecacz pamięci, Garbage Collector, 31

- okno Properties, 149

- opakowywanie zmiennych, 72

- operacje

- asynchroniczne, 139

- wejścia/wyjścia, 25

- operator

- , 39, 62

- , 37, 62

- !, 39

- !=, 42, 62

- %, 40

- &, 39, 62

- (), 36

- *, 40, 61

- /, 40

- ??, 44

- [], 62

- ~, 39

- +, 38, 62

- ++, 36, 62

- <, 41, 62

- <<, 40

- <=, 41, 62

- =, 43

- ==, 42, 62

- >, 41, 62

- >, 38, 62

- >=, 41, 62

operator
 >>, 40
 a[x], 36
 as, 41
 checked, 37
 is, 41
 new, 37
 sizeof, 40
 typeof, 37
 unchecked, 38
 warunkowy, 43
 x.y, 36
 operatory skrócone przypisania, 44

P

pakiet Microsoft Visual Studio 2012 Ultimate, 17
 pasek
 aplikacji, App bar, 175, 178
 funkcji, The charms, 175
 pętla
 do-while, 45
 for, 45
 foreach, 48
 while, 49
 PInvoke, 127
 plik ilasm.exe, 165
 pliki
 DLL, 141
 IL, 165
 CS, 156
 XAML, 156
 XML, 79
 ZIP, 139
 pobieranie danych, 26
 pole, 70
 pole statyczne, 76
 polecenie cd, 166
 polimorfizm, 88
 późne wiązanie, 71
 preprocesor, 26
 priorytet operatora, 35
 procesy, 142
 programowanie
 obiektowe, 63
 sieciowe, 159
 protokół FTP, 161
 przeciążanie
 metod, 72
 operatorów, 73
 przekazywanie
 argumentów
 przez referencję, 54
 przez wartość, 54
 tablic, 56

przesłanie metody ToString(), 94
 przestrzeń nazw, 80
 System.Diagnostics, 142
 System.Exception, 147
 System.IO, 134
 System.Net, 159
 System.Numerics, 133

R

rejestr, 140
 rodzaje projektu, 15
 rzutowanie, 24

S

serializacja, 78
 składniki pakietu, 17
 składowe wirtualne, 91
 słowa kluczowe języka C#, 185
 słowo kluczowe
 abstract, 77, 91
 break, 49
 catch, 146
 class, 22, 63
 continue, 50
 explicit, 75
 finally, 146
 goto, 50
 implicit, 75
 interface, 83
 new, 92
 operator, 73
 override, 91
 public, 63
 readonly, 70
 return, 51
 sealed, 78
 this, 65, 88
 throw, 51, 146
 try, 146
 unsafe, 60
 using, 80
 value, 88
 virtual, 91
 volatile, 144
 stała, 30
 sarta, 31
 stos, 31
 strona startowa, 177
 struktura
 BigInteger, 132
 Complex, 134
 Int64, 132
 UInt64, 132

struktura programu, 19
 strumienie, 137
 symulowanie
 klawiatury, 152
 myszy, 152
 system
 DNS, 159
 plików, 134

Ś

środowisko .NET 4.0, 9

T

tablice, 55
 technika przeciągnij i upuść, 153
 technologia Intellisense, 71
 tworzenie
 bibliotek, 141
 interfejsu graficznego, 149
 klucza, 140
 projektu, 15, 177
 projektu WPF, 155
 typ
 BigInteger, 133
 decimal, 22
 logiczny, 22
 polimorficzny, 90
 strukturalny, 24
 wskaźnikowy, 61
 wyliczeniowy, 24
 typy
 całkowite, 21
 generyczne, 111
 referencyjne, 22
 zmiennoprzecinkowe, 22

U

ukrywanie składowych, 92
 uruchamianie
 aplikacji Metro, 178
 procesów, 142

V

Visual Studio 2012
 diagramy UML, 18
 górne menu, 16
 instalacja, 12
 kompilacja, 15
 konfiguracja, 14

odpluskwanie, 15
 okna, 16
 paski narzędzi, 17
 tworzenie projektu, 15, 177
 uruchamianie, 15
 wymagania, 11

W

wartość klucza, 140
 wątki, 143
 wczesne wiązanie, 71
 wejście
 klawiatury, 150
 myszy, 151
 wiersz poleceń, 52
 Windows Presentation Foundation, 155
 Windows Forms, 153
 Windows Metro Style, 177
 wirtualne
 dziedziczenie, 93
 składowe, 92
 właściwość, 82
 właściwości tablicy, 56
 WPF, Windows Presentation Foundation, 15, 155
 wskaźniki, 60
 wydajność, 31
 wyjątek
 NullReferenceException, 129
 StackOverflowException, 31
 wyrażenia lambda, 96
 wysyłanie wiadomości
 do serwera, 161
 e-mail, 160
 wyświetlanie danych, 25
 wywoływanie funkcji poprzez PInvoke, 127

Z

zapis do pliku, 79
 zapis do pliku tekstowego, 139
 zdarzenia, 98
 dotyczące klawiatury, 151
 dotyczące myszy, 151
 zmienna, 30
 znaczniki XML, 20
 znak @, 185
 znaki specjalne, 25, 131

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**

Microsoft Visual Studio 2012

Programowanie w C#

O fantastycznych właściwościach Visual Studio 2012 z pewnością słyszał już każdy, kto choć odrobinę interesuje się tematem programowania. To środowisko programistyczne zapewnia użytkownikom komfort działania, jakiego próżno szukać gdziekolwiek indziej. Jego wygoda oraz łatwość obsługi pozwalają programiście skupić się na tym, co rzeczywiście chce osiągnąć, bez irytującego rozpraszania się na tysiące drobiazgów, o których kiedyś musiał pamiętać. Jeśli w dodatku ten programista poprawnie używa wydajnego, elastycznego języka C#, może stworzyć naprawdę świetną aplikację. Czy nie powinienes w końcu się tego nauczyć?

Ta książka pomoże Ci wejść w świat programowania w C# z użyciem najnowszej wersji Visual Studio. Znajdziesz tu dokładny opis działania środowiska oraz szczegółowe informacje na temat posługiwanie się językiem C# — od najprostszych operatorów i tablic, przez zachowania klas i obiektów, aż po tworzenie bibliotek i obsługę błędów. Dowiesz się, jak wykorzystać Asembler IL i na czym polega programowanie sieciowe. Odkryjesz, jak należy budować aplikację w stylu Metro dla systemu Windows 8, a na dokładkę dostaniesz listę słów kluczowych C# i zestaw instrukcji Asemblera IL. Co Ty na to?

Dzięki tej książce dowiesz się więcej o:

- języku C# i platformie .NET
- Microsoft Visual Studio 2012
- zaawansowanemu programowaniu obiektowym i innych zagadnieniach w C#
- tworzeniu interfejsu graficznego aplikacji
- podstawach programowania sieciowego
- obsłudze Asemblera IL
- podstawach tworzenia aplikacji w stylu Metro dla Windows 8
- słowach kluczowych języka C#

Programuj w C# — koniecznie z Visual Studio 2012!

helion.pl
księgarnia
internetowa

Nr katalogowy: 13118

Księgarnia internetowa:
<http://helion.pl>

Zamówienia telefoniczne:
0 801 339900
0 601 339900



Helion

Sprawdź najnowsze promocje:

• <http://helion.pl/promocje>

Książki najchętniej czytane:

• <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

• <http://helion.pl/nowości>

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po **WIECEJ**



KOD KORZYŚCI

ISBN 978-83-246-6562-4



9 788324 665624

Cena: 39,90 zł

Informatyka w najlepszym wydaniu