

# 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 niezwykłe możliwości obiektowego języka programowania C#  
Poznaj zaawansowane zagadnienia programowania obiektowego, podstawy obsługi  
sieci oraz Asembler IL

# Spis treści

<b>Rozdział 1.</b>	<b>Wstęp .....</b>	<b>9</b>
1.1.	O języku C# i platformie .NET .....	9
1.2.	Podstawowe pojęcia .....	9
1.3.	Potrzebne narzędzia .....	10
<b>Rozdział 2.</b>	<b>Microsoft Visual Studio 2012 .....</b>	<b>11</b>
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
<b>Rozdział 3.</b>	<b>Język C#. Podstawy .....</b>	<b>19</b>
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 stale .....	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
Koniuunkcja logiczna .....	42
Alternatywa wykluczająca logiczna .....	42
Alternatywa logiczna .....	42
Koniuunkcja warunkowa .....	43
Alternatywa warunkowa .....	43
Operator warunkowy .....	43
Przypisymania .....	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

<b>Rozdział 4. Język C#. Programowanie obiektowe .....</b>	<b>63</b>
4.1. Klasy i obiekty .....	63
Słowo kluczowe this .....	65
4.2. Konstruktor i destruktor .....	66
4.3. Dziedziczenie .....	67
Klasy zagnieździone .....	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 zapieczętowane .....	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
<b>Rozdział 5. Język C#. Pozostałe zagadnienia .....</b>	<b>127</b>
5.1. Wywoływanie funkcji przez PInvoke .....	127
5.2. Napisy (ang. Strings) .....	129
Deklaracja i inicjalizacja .....	129
Niezmiennosć 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
<b>Rozdział 6. Tworzenie interfejsu graficznego aplikacji .....</b>	<b>149</b>
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. Przeciagnij i upuść .....	153
6.6. Przegląd wybranych kontrolek .....	153
6.7. Wstęp do Windows Presentation Foundation .....	155
Tworzenie projektu WPF .....	155
Przykład: „Witaj, świecie WPF!” .....	156
<b>Rozdział 7. Podstawy programowania sieciowego .....</b>	<b>159</b>
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

<b>Rozdział 8. Asembler IL .....</b>	<b>165</b>
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
<b>Rozdział 9. Podstawy tworzenia aplikacji w stylu Metro dla Windows 8 .....</b>	<b>175</b>
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 kontrolek .....	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
<b>Dodatek A Słowa kluczowe języka C# .....</b>	<b>185</b>
<b>Dodatek B Zestaw instrukcji Assemblera IL .....</b>	<b>187</b>
Operacje arytmetyczne .....	187
Dodawanie .....	187
Odejmowanie .....	187
Mnożenie .....	187
Dzielenie .....	188
Modulo .....	188
Wartość negatywna .....	188
Operacje bitowe .....	188
Konjunkcja .....	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
Pozostale .....	193
<b>Skorowidz .....</b>	<b>195</b>

# Rozdział 1.

## Wstęp

Witaj w książce poświęconej Visual Studio 2012 — najnowszemu środowisku programistycznemu firmy Microsoft — oraz językowi C#. Książka ta przeznaczona jest dla osób chcących poznać możliwości środowiska Visual Studio w wersji 2012 oraz nauczyć się programować w bardzo popularnym i wydajnym języku, jakim jest C#. Jeżeli nigdy wcześniej nie programowałeś, to polecam Ci przeczytanie jakiegoś kursu języka C, gdyż jest on bardzo dobrą podstawą do nauki innych języków powstałych na jego bazie.

### 1.1. O języku C# i platformie .NET

Język C# (lub CSharp) to obiektowy język programowania zaprojektowany przez Andersa Hejlsberga dla firmy Microsoft. Jest on wieloplatformowy, czyli uruchamiany przez środowisko .NET lub jego odpowiedniki takie jak np. Mono dla systemu Linux. Aktualna najnowsza wersja tego języka to 5.0. Kod napisany w C# jest kompilowany do języka pośredniego o nazwie CIL (ang. *Common Intermediate Language*) i wykonywany przez środowisko uruchomieniowe. Najnowsza stabilna wersja środowiska .NET to 4.0, ale trwają już prace nad wersją 4.5 (jest już wersja beta), która będzie obowiązywała w Microsoft Visual Studio 2012.

### 1.2. Podstawowe pojęcia

**Środowisko programistyczne** — oprogramowanie do tworzenia aplikacji zawierające w sobie najczęściej edytor kodu źródłowego, odpluskwiacz oraz kompilator.

**Kompilator** — zamienia kod źródłowy programu (zrozumiały dla człowieka) na kod binarny (zrozumiały dla komputera).

**Odpluskwiacz (ang. *Debugger*)** — osobom zajmującym się inżynierią odwrotną pozwala zrozumieć działanie aplikacji, natomiast programistom służy do wykrywania błędów w programach.

**Kapsulkowanie (ang. *Encapsulating*)** — w programowaniu obiektowym jest to grupowanie ze sobą zmiennych, metod i innych składników.

## 1.3. Potrzebne narzędzia

Podczas pisania książki korzystałem z systemu operacyjnego Windows 7 Home Premium (64-bit) oraz środowiska programistycznego Microsoft Visual Studio 2012 Ultimate. Środowisko to możesz pobrać za darmo ze strony Microsoft w 90-dniowej wersji testowej.

www.ebook4all.pl

## Rozdział 2.

# **Microsoft Visual Studio 2012**

Wymagania do uruchomienia tego środowiska są następujące:

System operacyjny:

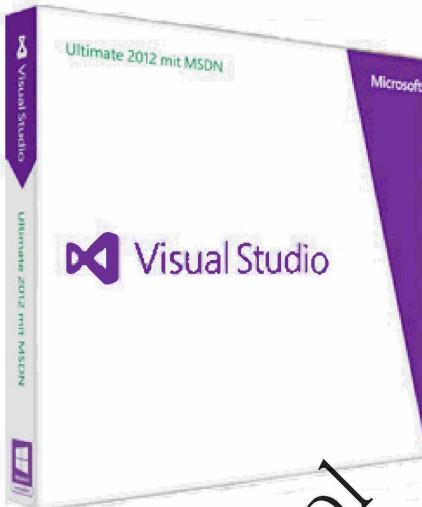
- ◆ Windows 7 (x86 lub x64)
- ◆ Windows 8 (x86 lub x64)
- ◆ Windows Server 2008 R2 SP1 (x64)
- ◆ Windows Server 2012 (x64)

Sprzęt:

- ◆ Procesor 1.6 GHz lub szybszy,
- ◆ 1 GB pamięci RAM (1,5 GB na maszynie wirtualnej),
- ◆ 10 GB (NTFS) miejsca na dysku,
- ◆ Dysk twardy 5400 RPM,
- ◆ Karta graficzna obsługująca DirectX 9 i rozdzielcość 1024×768 lub wyższą.

Na rysunku 2.1. widzimy wersję pudełkową tego środowiska w najbogatszym wydaniu (tj. Ultimate).

**Rysunek 2.1.**  
Microsoft Visual  
Studio 2012 Ultimate  
— wersja pudelkowa



## 2.1. Co nowego?

Nowości w Microsoft Visual Studio 2012:

- ◆ Projektowanie i budowanie aplikacji w stylu Metro (wymaga Windows 8).
- ◆ Odpuszkianie, optymalizacja i publikacja aplikacji Metro (wymaga Windows 8).
- ◆ Asynchroniczne ładowanie solucji (przyśpiesza pracę).
- ◆ Praca z projektami stworzonymi w Visual Studio 2010 (kompatybilność wsteczna).
- ◆ Dopasowanie schematu kolorów środowiska.
- ◆ Ulepszone kolorowanie kodu i podpowiedzi.
- ◆ Możliwość uzyskania ścieżki oraz numeru linii pliku z kodem wywołującym metodę (C#).
- ◆ Generowanie kodu C# z diagramu UML.

To by były te najważniejsze zmiany w środowisku.

## 2.2. Instalacja

Pobierz Microsoft Visual Studio 2012 z poniższego adresu:

<http://www.microsoft.com/visualstudio/11/en-us/downloads>

Kliknij dwukrotnie instalator, który pobrałeś (rysunek 2.2).

### Rysunek 2.2.

Instalator środowiska  
MS Visual Studio 2012 Ultimate



Po chwili powinno pojawić się okno z wyborem ścieżki instalacji oraz licencją, którą — aby zainstalować środowisko — musisz przeczytać i zaakceptować (rysunek 2.3).

### Rysunek 2.3.

Wybór ścieżki instalacji  
i licencja



Kolejne okno, które się pojawi, służy do wyboru składników, jakie zostaną zainstalowane (rysunek 2.4).

Teraz wystarczy kliknąć przycisk *INSTALL* i czekać, aż środowisko się zainstaluje.

**Rysunek 2.4.**  
Wybór składników do zainstalowania



## 2.3. Konfiguracja

Za chwilę przejdziemy do wstępnej konfiguracji środowiska (czyli przygotowania go do pracy). Uruchom środowisko, klikając *Start/Wszystkie programy/Microsoft Visual Studio 2012/Visual Studio 2012*.

### Pasek obsługi odpluskwiania

Z górnego menu wybierz *TOOLS/Customize...* i przejdź do zakładki *Toolbars*. Zaznacz *Debug* i kliknij *Close*. Teraz na górnym pasku z ikonami powinny się pojawić przyciski do zatrzymania odpluskwiania, zakończenia pracy aplikacji, restartu i odświeżenia.

### Numerowanie wierszy

Gdy program ma błędy, na dole pojawia się opis błędu i wiersz, w którym ten błąd występuje. Niestety, środowisko nie ma domyślnie włączonego numerowania wierszy i ciężko wtedy poprawiać błędy bez numerowania linii. Z górnego menu wybierz

TOOLS/Options.../Text Editor/All Languages/General/Display i z prawej strony zaznacz pole *Line Numbers*. Od tej pory w Twoich projektach wiersze kodu będą numerowane.

## 2.4. Tworzenie nowego projektu

Aby utworzyć nowy projekt w Microsoft Visual Studio 2012, należy z górnego menu kliknąć *FILE/New/Project...* — wtedy pojawi się okno tworzenia nowego projektu.

Do wyboru mamy (wymieniam te ważniejsze):

- ◆ Windows Forms Application — aplikacja z interfejsem WinForms,
- ◆ WPF Application — aplikacja z interfejsem WPF (*Windows Presentation Foundation*),
- ◆ Console Application — aplikacja konsolowa.

Po wybraniu rodzaju projektu na dole wpisujemy nazwę projektu, ścieżkę do katalogu, gdzie będą zapisane pliki projektu, oraz nazwę solucji. Gdy wypełnimy te pola, możemy kliknąć *OK*, aby utworzyć nowy projekt.

## 2.5. Kompilacja i uruchamianie

Zakładam, że utworzyłeś nowy projekt. Teraz opiszę, jak go skompilować i uruchomić. Aby skompilować projekt, należy w górnym menu kliknąć *Build Solution* (skrót klawiaturowy to *F6*) i chwilę poczekać. Natomiast uruchomić projekt można na dwa sposoby: albo z odpluskwianiem (*Start Debugging*, skrót klawiaturowy *F5*), albo bez odpluskwiania (*Start Without Debugging*, skrót klawiaturowy *Ctrl+F5*).

## 2.6. Odpluskwianie (ang. Debugging)

Odpluskwianie to usuwanie błędów z programu. Istnieją błędy komplikacji wykrywane przez środowisko oraz błędy pokompilacyjne. Oczywiście trudniejsze do wykrycia są błędy pokompilacyjne.

### Błędy procesu komplikacji

Są to błędy dość proste do usunięcia. Wystarczy dokładnie przeczytać treść błędu i wiersz, w którym występuje, a następnie zastosować poprawkę. Przykładowym błędem może być brak średnika na końcu instrukcji, wtedy środowisko wyświetli nam błąd o treści "; expected", co oznacza, że oczekuje w danej linii średnika.

## Błędy pokompilacyjne

Są trudniejsze do usunięcia od błędów procesu kompilacji. Program wtedy poprawnie się kompiluje, ale niepoprawnie uruchamia lub przerywa swoją pracę z powodu takiego błędu. Przykładowym błędem pokompilacyjnym może być przekroczenie zakresu tablicy. Wygląda to tak, że deklarujemy tablicę 10-elementową, a przypisujemy coś do jedenastego elementu tej tablicy, czyli poza jej zakres. Program się skompiluje poprawnie, ale jego działanie zostanie przerwane wyjątkiem o nazwie *IndexOutOfRangeException*, o czym poinformuje nas odpluskwiacz zawarty w środowisku (rysunek 2.5).

A screenshot of Microsoft Visual Studio showing a tooltip for an *IndexOutOfRangeException*. The tooltip contains the message: "Index was outside the bounds of the array." The code in the editor shows an attempt to assign a value to index 11 of an array of size 10.

```
int[] t = new int[10];
for(int i = 0; i <= 10; i++)
{
    t[i] = 0;
}
```

Rysunek 2.5. Wykrycie wyjątku wykroczenia poza zakres tablicy

Dodatkowo podczas procesu odpluskowania na dole w oknie o nazwie *Locals* wyświetlane są wartości zmiennych, co pomaga określić, czy nie przekroczyliśmy zakresu i czy wartości są takie, jakie powinny być.

## 2.7. Okna, menu i paski narzędzi

### Okna

*Solution Explorer* — wyświetla składowe naszego projektu w postaci drzewa.

*Properties* — wyświetla właściwości zaznaczonego elementu oraz zdarzenia (w przypadku formularzy i kontrolek).

*Error List* — wyświetla błędy w kodzie źródłowym (jeżeli jakieś występują), podając ścieżkę i numer błędnego wiersza.

*Toolbox* — zawiera dostępne kontrolki, które metodą *Przeciągnij i upuść* możemy wstawić do formularza aplikacji.

### Górne menu

*File* — zawiera między innymi takie opcje jak stworzenie nowego projektu (ang. *New Project*), otworzenie istniejącego projektu (ang. *Open Project*), zapisanie projektu (ang. *Save*) i inne.

*Edit* — zawiera opcje podstawowego edytora tekstowego.

*View* — określa, co ma być wyświetlane.

*Project* — pozwala dodać nowy składnik do projektu i zmienić ustawienia projektu (ang. *Properties*).

*Build* — zawiera opcje komplikacji (budowania) projektu.

*Debug* — zawiera opcje odpluskwiania i uruchamiania.

*Team* — pozwala połączyć się z serwerem obsługującym pracę w zespole.

*Tools* — zawiera podstawowe narzędzia oraz pozwala ustawić opcje środowiska.

*Test* — opcje testowania projektu.

*Architecture* — zawiera opcje dotyczące wizualnych diagramów klas, przypadków użycia (Use Case) i innych.

*Analyze* — zawiera narzędzia do dokonywania analizy.

*Window* — pozwala zmienić układ okien w środowisku.

*Help* — wszelka pomoc dotycząca pracy w środowisku.

## Paski narzędzi

Paski narzędzi pozwalają na szybki dostęp do najczęściej używanych funkcji środowiska. Klikając przycisk *Add or Remove Buttons*, możemy dowolnie ustawić ikony wyświetlane na paskach.

## 2.8. Składniki pakietu

Pakiet Microsoft Visual Studio 2012 Ultimate zawiera dodatkowe narzędzia, takie jak:

- ◆ Blend for Visual Studio 2012 — pomoże zaprojektować i utworzyć interfejs użytkownika dla aplikacji Metro (wymagany system Windows 8).
- ◆ Microsoft Help Viewer — pozwala przeglądać pomoc dotyczącą środowiska.
- ◆ Microsoft Test Manager — narzędzie do wykonywania testów.
- ◆ Developer Command Prompt — wiersz poleceń dla programisty.
- ◆ MFC-ATL Trace Tool — wyświetla wiadomości podczas odpluskwiania kodu MFC lub ATL.
- ◆ Microsoft Spy++ — prezentuje w formie graficznej procesy systemowe, wątki, okna i komunikaty.

## 2.9. Projektowanie diagramów UML

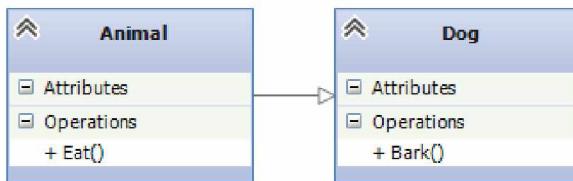
Najnowsze środowisko Visual Studio 2012 wspiera tworzenie diagramów UML. Udostępnia do tego wizualne narzędzie pozwalające bardzo szybko i łatwo zaprojektować określony diagram.

W górnym menu kliknij *ARCHITECTURE/New Diagram...*, a następnie z listy szablonów wybierz *UML Class Diagram*.

Teraz możesz projektować swój diagram. Z lewej strony z okienka *Toolbox* wybierz element i przeciągnij go na główne okno edytora. Przykładowy diagram prezentuje rysunek 2.6.

**Rysunek 2.6.**

Przykładowy diagram UML



## Rozdział 3.

# Język C#. Podstawy

## 3.1. Struktura kodu źródłowego

Struktura prostego programu w języku C# prezentuje się następująco:

```
using System; //używana przestrzeń nazw
public class Projekt1 //klasa główna
{
    public static int Main() //funkcja główna
    {
        //tutaj instrukcje
        return 0; //kod powrotu
    }
}
```

Najpierw po słowie kluczowym `using` mamy określenie przestrzeni nazw, z jakiej korzysta program (tych przestrzeni może być kilka). Dalej znajduje się główna klasa programu (rozbudowane programy mają przeważnie więcej niż jedną klasę). W klasie jest zdefiniowana funkcja główna o nazwie `Main`, w której umieszczamy instrukcje do wykonania przez program. Funkcja ta zwraca liczbę całkowitą. Aby zostało to wykonyane, należy po słowie kluczowym `return` podać wartość do zwrócenia. Warto zwrócić uwagę na formatowanie kodu źródłowego. Najczęściej spotyka się wcięcia o długości czterech spacji i takie wcięcia będę stosował w tej książce.

## 3.2. Komentarze

### Komentarz blokowy

Komentarz blokowy zawieramy pomiędzy znakami /\* oraz \*/. Tekst znajdujący się pomiędzy tymi znakami jest ignorowany podczas komilacji. Komentarzy blokowych nie można zagnieździć (czyli umieszczać jednego w drugim).

### Komentarz liniowy

Komentarz liniowy zaczyna się od znaków // i kończy wraz z końcem linii. Tekst od znaków // do końca linii jest pomijany przez komplator.

### Komentarz XML

Pozwala wygenerować dokumentację kodu na podstawie znaczników XML. Komentarz XML zaczynamy trzema ukośnikami i używamy w nim tagów XML.

Dostępne znaczniki XML przedstawia tabela 3.1.

**Tabela 3.1. Znaczniki XML**

Tag	Opis
<c>	Oznacza tekst jako kod
<code>	Oznacza kilkuliniowy tekst jako kod
<example>	Opis przykładowego kodu
<exception>	Pozwala określić rzucane wyjątki
<include>	Pozwala dodać plik z komentarzami
<list>	Tworzy listę wypunktowaną
<para>	Paragraf
<param>	Używane przy opisywaniu parametrów metody
<paramref>	Referencja do parametru
<permission>	Pozwala opisać dostęp do części np. klasy
<remarks>	Opisuje typ
<returns>	Opisuje wartość zwracaną
<see>	Tworzy link
<seealso>	Odwoluje się do sekcji <i>Zobacz także</i>
<summary>	Opisuje typ obiektu
<typeparam>	Opisuje typ parametru
<typeparamref>	Opisuje typ elementu
<value>	Opisuje właściwość

Przykładowy komentarz XML:

```
/// <summary>
/// Ta klasa wykonuje bardzo ważną funkcję
/// </summary>
public class MyClass{}
```

Jeżeli chcesz wygenerować dokumentację XML dla swojego programu, to do opcji kompilatora dodaj parametr /doc, np.

```
/doc:filename.xml
```

## 3.3. Program „Witaj, świecie!”

Oto kod programu wyświetlającego na konsoli napis „Witaj, świecie!”:

```
public class Hello1
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, World!");
    }
}
```

Każda metoda Main musi być zawarta w klasie. Klasa System.Console zawiera metodę WriteLine do wyświetlania tekstu na konsoli.

## 3.4. Typy danych

### Typy proste

#### Typy całkowitoliczbowe

Typy całkowitoliczbowe z ich zakresem zostały przedstawione w tabeli 3.2.

**Tabela 3.2.** Typy całkowite i ich zakres

Typ	Zakres	Rozmiar
sbyte	-128 do 127	8-bitowa liczba całkowita ze znakiem
byte	0 do 255	8-bitowa liczba całkowita bez znaku
char	U+0000 do U+ffff	16-bitowy znak Unicode
short	-32 768 do 32 767	16-bitowa liczba całkowita ze znakiem
ushort	0 do 65 535	16-bitowa liczba całkowita bez znaku
int	-2 147 483,648 do 2 147 483,647	32-bitowa liczba całkowita ze znakiem

**Tabela 3.2.** Typy całkowite i ich zakres — ciąg dalszy

Typ	Zakres	Rozmiar
uint	0 do 4 294 967,295	32-bitowa liczba całkowita bez znaku
long	-9 223 372 036 854 775 808 do 9 223 372 036 854 775 807	64-bitowa liczba całkowita ze znakiem
ulong	0 do 18 446 744 073 709 551 615	64-bitowa liczba całkowita bez znaku

## Typy zmiennoprzecinkowe

Typy zmiennoprzecinkowe i ich zakres zostały przedstawione w tabeli 3.3.

**Tabela 3.3.** Typy zmiennoprzecinkowe i ich zakres

Typ	Przybliżony zakres	Precyzja
float	$\pm 1,5 \times 10^{-45}$ do $\pm 3,4 \times 10^{38}$	7 cyfr
double	$\pm 5,0 \times 10^{-324}$ do $\pm 1,7 \times 10^{308}$	15 – 16 cyfr

## Typ decimal

Jest to typ 128-bitowy używany do operacji pieniężnych, gdyż ma większą precyzję i mniejszy zakres niż typy zmiennoprzecinkowe (tabela 3.4).

**Tabela 3.4.** Typ decimal i jego zakres

Typ	Przybliżony zakres	Precyzja	Typ w .NET
decimal	$\pm 1,0 \times 10^{-28}$ do $\pm 7,9 \times 10^{28}$	28 – 29 cyfr mantysy	System.Decimal

## Typ logiczny (bool)

Jest to typ, który przechowuje dwa rodzaje wartości: prawda (ang. *True*) i falsz (ang. *False*). Typ ten jest aliasem dla `System.Boolean`.

## Typy referencyjne

### Klasa

Klasy deklarujemy, używając słowa kluczowego `class`. Przykład:

```
class TestClass
{
    // Metody, właściwości, pola, zdarzenia, delegaty
    // i klasy zagnieździone deklarujemy tutaj
}
```

Klasa może zawierać w sobie:

- ◆ Konstruktory
- ◆ Destruktory

- ◆ Stałe
- ◆ Pola
- ◆ Metody
- ◆ Właściwości
- ◆ Indeksery
- ◆ Operatory
- ◆ Zdarzenia
- ◆ Delegaty
- ◆ Klasy zagnieżdżone
- ◆ Interfejsy
- ◆ Struktury

Więcej o klasach będzie powiedziane w dalszej części książki, gdzie omówione zostanie programowanie zorientowane obiektowo.

## Interfejs

Interfejs zawiera w sobie sygnatury metod, delegat, właściwości, indekserów i zdarzeń. Może być zawarty w przestrzeni nazw lub klasie i dziedziczyć z jednego lub więcej interfejsów bazowych. Więcej o interfejsach w dalszej części książki.

## Delegaty

Deklaracja przykładowej delegaty prezentuje się następująco:

```
public delegate void TestDelegate(string message);
```

Są one używane do kapsulkowania metod nazwanych lub anonimowych. Są wprawdzie podobne do wskaźników funkcji w języku C, ale bezpieczniejsze.

## Objekt

Jest to alias dla `System.Object`. Wszystkie typy pośrednio lub bezpośrednio dziedziczą z obiektu.

## Napis

Jest to sekwencja znaków Unicode. Typ ten jest aliasem dla `System.String`. Chociaż jest to typ referencyjny, to napisy możemy porównywać, używając operatorów `==` (równe) oraz `!=` (różne).

## Typ strukturalny

Jest to typ, który służy do kapsulkowania powiązanych ze sobą zmiennych. Deklaração przykładowej struktury prezentuje się następująco:

```
public struct Book
{
    public decimal price;
    public string title;
    public string author;
}
```

## Typ wyliczeniowy

Służy do deklarowania grupy nazwanych stałych powiązanych ze sobą. Możemy dla przykładu stworzyć typ wyliczeniowy reprezentujący dni tygodnia:

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

Poszczególne stale są automatycznie numerowane liczbami całkowitymi, licząc od zera. Jednak można zmienić numerację, żeby była na przykład od jednego, i wtedy piszemy:

```
enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

## Rzutowanie i konwersja typów

Język C# bardzo dokładnie sprawdza zgodność typów zmiennych. Dlatego na przykład nie możemy zmiennej typu `int` przypisać wartości typu `double`, tylko musimy rzutować.

Rzutowanie ma taki schemat:

```
var1 = (type) var2;
```

Jeżeli chcemy rzutować `double` na `int`, piszemy:

```
double a = 2.99; //zmienna typu double o wartości 2.99
int b; //zmienna typu int
b = (int) a; //przypisanie z rzutowaniem na int
```

Takie rzutowanie jak powyżej powoduje oczywiście utracenie tego, co jest po kropce.

A co, jeśli chcemy zamienić liczbę na napis? Używamy wtedy metody `ToString()`:

```
int i=1;
string s = i.ToString();
```

## 3.5. Proste operacje wejścia/wyjścia

Dowiesz się za chwilę, jak wykonywać proste operacje wejścia/wyjścia. Korzystać będziemy z klasy System.Console.

### Wyświetlanie danych

Najlepiej jest się uczyć na przykładach, dlatego popatrz na poniższy kod:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.Write("napis1");
        Console.WriteLine("napis2");
        Console.Write("napis3\n");
    }
}
```

Użyte tutaj zostały dwie metody, pierwsza to `Write` — służy ona do wypisywania tekstu na standardowe wyjście, druga metoda to `WriteLine` — wypisuje tekst na standardowe wyjście i przechodzi do nowej linii. Warto też zwrócić uwagę na ostatnie wywołanie metody `Write`. W jej tekście musi być wypisany znak specjalny `\n`, który jest znakiem nowej linii. Tych znaków specjalnych jest więcej, dlatego przedstawiono je w tabeli 3.5.

Tabela 3.5. Znaki specjalne

Znak	Znaczenie	Unicode
\'	Pojedynczy cudzysłów	0x0027
\"	Podwójny cudzysłów	0x0022
\\"	Backslash	0x005C
\0	NULL	0x0000
\a	Alarm	0x0007
\b	Backspace	0x0008
\f	Form Feed	0x000C
\n	Nowa linia	0x000A
\r	Powrót	0x000D
\t	Tabulator poziomy	0x0009
\v	Tabulator pionowy	0x000B

## Pobieranie danych

Dane pobieramy równie łatwo, jak wyświetlamy. Służą do tego metody:

- ◆ `Console.Read();` — pobiera następny znak ze standardowego wejścia;
- ◆ `Console.ReadKey();` — pobiera następny znak lub klawisz wcisnięty przez użytkownika;
- ◆ `Console.ReadLine();` — pobiera następną linię ze standardowego wejścia.

A oto przykładowy program pytający użytkownika o imię i pozdrawiający go:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.Write("Podaj swoje imię: ");
        Console.WriteLine("Witaj {0}!", Console.ReadLine());
    }
}
```

## 3.6. Preprocesor

Preprocesor to program (często zintegrowany z kompilatorem), który przetwarza kod źródłowy za pomocą dyrektyw. Po takim przetworzeniu kod jest poddawany dalej analizie składniowej oraz komplikacji.

### Dyrektyna #if

Gdy kompilator spotka dyrektywę `#if`, a po niej dyrektywę `#endif`, skompiluje kod pomiędzy dyrektywami tylko wtedy, gdy określony symbol jest zdefiniowany. Sprawdzany przez dyrektywę `#if` symbol nie może być typu liczbowego tak jak w C i C++, może być tylko typu logicznego (*Boolean*).

Na przykład:

```
#define DEBUG
//...
#ifndef DEBUG
    Console.WriteLine("Wersja debug");
#endif
```

Do sprawdzania wartości logicznej symbolu możesz używać operatorów: `==` (równe) i `!=` (różne). Prawda (*True*) oznacza, że symbol jest zdefiniowany. Zapis `#if DEBUG` znaczy to samo co `#if (DEBUG == true)`. Możesz używać operatorów: `&&` (koniunkcja), `||` (alternatywa) i `!` (negacja) do sprawdzania wielu symboli.

## Dyrektywa #else

Pozwala na stworzenie złożonej dyrektywy warunkowej. Jeżeli symbole przy dyrektywie `#if` i opcjonalnie `#elif` nie są prawdą, skompilowany zostanie kod pomiędzy `#else` i `#endif`.

Na przykład:

```
#define DEBUG
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
        #if (DEBUG && !MYTEST)
            Console.WriteLine("DEBUG zdefiniowane");
        #elif (!DEBUG && MYTEST)
            Console.WriteLine("MYTEST zdefiniowane ");
        #elif (DEBUG && MYTEST)
            Console.WriteLine("DEBUG i MYTEST zdefiniowane ");
        #else
            Console.WriteLine("DEBUG i MYTEST niezdefiniowane ");
        #endif
    }
}
```

## Dyrektywa #elif

Pozwala na stworzenie złożonej dyrektywy warunkowej. Jeżeli symbole przy dyrektywie `#if` i wcześniejszych dyrektywach `#elif` nie są prawdą, skompilowany zostanie kod po `#elif`, przy którym symbol jest prawda.

## Dyrektywa #endif

Określa koniec dyrektywy warunkowej rozpoczętej przez `#if`.

## Dyrektywa #define

Dyrektywa ta pozwala zdefiniować symbol. Jeżeli użyjesz zdefiniowanego symbolu przy dyrektywie `#if`, wyrażenie będzie miało wartość prawda.

Na przykład:

```
#define DEBUG
```



Wskazówka

Dyrektywa `#define` nie może być użyta do definiowania stałych, tak jak się to przyjęło w językach C i C++. Stałe w C# najlepiej definiować jako statyczne składowe klasy lub struktury. Jeżeli masz więcej stałych, stwórz dla nich osobną klasę, np. o nazwie Constants.

## Dyrektyna `#undef`

Pozwala odwołać zdefiniowany wcześniej symbol. Gdy podamy ten symbol jako wyrażenie dyrektywie `#if`, wyrażenie będzie miało wartość falsz.

Na przykład:

```
#undef DEBUG
using System;
class MyClass
{
    static void Main()
    {
        #if DEBUG
            Console.WriteLine("DEBUG is defined");
        #else
            Console.WriteLine("DEBUG is not defined");
        #endif
    }
}
```

## Dyrektyna `#warning`

Pozwala wygenerować ostrzeżenie pierwszego stopnia z określonej lokalizacji w kodzie.

Na przykład:

```
#warning Kod użyty w tej metodzie jest zdeprecjonowany.
```

## Dyrektyna `#error`

Pozwala wygenerować błąd z określonej lokalizacji w kodzie.

Na przykład:

```
#error Kod użyty w tej metodzie jest zdeprecjonowany.
```

## Dyrektyna `#line`

Pozwala zmodyfikować numerowanie linii przez kompilator (gdy na przykład usuniemy jakiś kod, a chcemy, żeby numerowanie było takie, jakby ten kod nadal tam był). Użycie `#line default` przywraca domyślne numerowanie. Natomiast użycie `#line hidden` powoduje, że odpluskwiacz omija dany fragment kodu.

Na przykład:

```
class MainClass
{
    static void Main()
    {
#line 200 "Specjalne"
        int i: // CS0168 na linii 200
        int j: // CS0168 na linii 201
#line default
        char c: // CS0168 na linii 9
        float f: // CS0168 na linii 10
#line hidden // numerowanie wyłączone
        string s;
        double d;
    }
}
```

## Dyrektyna #region

Pozwala określić blok kodu, który można zwinąć lub rozwinąć, używając specjalnej właściwości, jaką posiada edytor kodu pakietu Visual Studio.

Na przykład:

```
#region Klasa MyClass
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

Dyrektyna ta musi się zawsze kończyć dyrektywą #endregion.

## Dyrektyna #endregion

Kończy blok kodu określony przez dyrektywę #region.

## Dyrektyna #pragma warning

Pozwala włączyć lub wyłączyć ostrzeżenia kompilatora. Jako argumenty dla dyrektywy podajemy numery ostrzeżeń z okna *Output*. Jeżeli nie podamy żadnych numerów, to dyrektywa włączy (lub wyłączy) wszystkie ostrzeżenia.

Na przykład:

```
#pragma warning disable lista-ostrzezen
#pragma warning restore lista-ostrzezen
```

## 3.7. Zmienne i state

Zmienne to konstrukcje programistyczne do przechowywania różnych danych potrzebnych podczas działania programu. Każda zmienna ma swój typ i nazwę. Teraz dla przykładu zadeklarujemy zmienną typu całkowitoliczbowego o nazwie Number:

```
int Number;
```

Zmiennej możemy także przypisać jakąś wartość za pomocą operatora przypisania (znak =):

```
Number = 100;
```

Można także od razu przy deklaracji przypisać zmiennej wartość, co nazywa się inicjalizacją zmiennej:

```
int Number = 100;
```

Jeżeli potrzebujemy kilku zmiennych tego samego typu, piszemy:

```
int Num1, Num2, Num3;
```

Część tych zmiennych (lub wszystkie) możemy zainicjalizować daną wartością, np.

```
int Num1 = 100, Num2, Num3 = 200;
```

A teraz pytanie: jak wyzerować kilka zmiennych jednocześnie? Używamy w tym celu wielokrotnego przypisania:

```
a = b = c = 0;
```

Bardzo ważne jest nazywanie zmiennych. Nazwa powinna odzwierciedlać znaczenie zmiennej, np. jeżeli mamy zmienną określającą liczbę studentów, to nazwijmy ją numberOfStudents lub po polsku liczbaStudentow. Przyjęło się także, aby nazwy zmiennych rozpoczynać od malej litery, a kolejne słowa w zmiennych od dużej. Nazwa zmiennej może zawierać małe i duże litery, znaki podkreślenia oraz cyfry, ale nie może się zaczynać od cyfry.

Istnieje też takie coś jak stale, czyli wartości, które nie mogą zostać zmienione. Deklarujemy je, używając słowa kluczowego const:

```
const int a = 100;  
a = 50; //Błąd! Nie można modyfikować stałej!
```

Stale mogą być tylko typy wbudowane. Natomiast metody, właściwości i zdarzenia nie mogą być stale.

## 3.8. Stos i sterta

Stos i sterta są limitowane przez pamięć wirtualną dostępną dla uruchomionego procesu. Za chwilę sprawdzimy zarządzanie pamięcią na stosie i na stercie. Gdy przestrzeń stosu zostanie wyczerpana, rzucany zostaje wyjątek `StackOverflowException`. Pamięć, która nie jest już dostępna dla procesu, określana jest jako pamięć „poza zakresem” i poddawana zostaje natychmiast recyklingowi. Wyłączając ramki rekurencyjne, przestrzeń stosu jest alokowana w czasie komplikacji.

Zarządzanie pamięcią na stercie jest trochę bardziej skomplikowane. Zmienne na stercie są zawsze alokowane za pomocą słowa kluczowego `new`:

```
Test t=new Test();
```

Oczywiście typy proste (zawierające bezpośrednio daną wartość), które są alokowane na stosie, również mogą być alokowane za pomocą słowa kluczowego `new`. Typy proste, które są składowymi typu referencyjnego, są alokowane na stercie jako część typu referencyjnego.

Plataforma .NET uruchamia osobny wątek, który monitoruje alokowanie pamięci na stercie. Gdy użycie sterty urośnie o kilka kilobajtów, odśmieczacz pamięci (ang. *Garbage Collector*) zatrzymuje proces i „czyści” obiekty, które nie są już używane.

## Wydajność

Popatrz na metodę, która dodaje dwie liczby całkowite:

```
int add(int x, int y) {  
    return x+y;  
}
```

Jeżeli wywołamy tę metodę w taki sposób:

```
int x = 3; //na stosie głównym, musi być skopiowane do wnętrza funkcji  
int y = 5; //na stosie głównym, musi być skopiowane do wnętrza funkcji  
int result = add(x,y); //Wynik jest kopiowany z funkcji do stosu głównego
```

wówczas liczby 3 i 5 są kopiowane do przestrzeni stosu metody `add()`. Wynik jest alokowany w przestrzeni stosu metody `add()`, a później kopiowany z powrotem do zmiennej `result`.

Alternatywna implementacja to:

```
int add()  
{  
    return this.x + this.y;  
}  
this.x = 3; //na stercie  
this.y = 5; //na stercie  
int result = add(); //wynik kopiowany z funkcji do stosu głównego
```

Mimo że metoda nie ma argumentów, to zanim pola `x` i `y` zostaną użyte w metodzie `add()`, są one kopowane ze sterty obiektu `Test` do stosu programu.

Szybki test:

```
Test mtest=new Test();
int result,result2;
int iterations=100000000;
DateTime dt=DateTime.Now;
for(int i=0;i<iterations;i++)
    result=mtest.Add(3,4);
DateTime dt2=DateTime.Now;
Console.WriteLine((dt2.Ticks-dt.Ticks)/100000.0);
for(int i=0;i<iterations;i++)
    result2=mtest.Add();
DateTime dt3=DateTime.Now;
Console.WriteLine((dt3.Ticks-dt2.Ticks)/100000.0);
```

pokazuje, że pierwsza metoda jest o 6% szybsza niż druga.



Wskazówka

Wydajniej jest kopiować liczby ze stosu głównego do ciała funkcji, niż ładować je ze sterty.

## 3.9. Instrukcja warunkowa if

Instrukcja `if` wybiera kod do wykonania zależnie od logicznej wartości podanego wyrażenia. Składnia instrukcji `if` prezentuje się następująco:

```
if (wyrażenie)
    kod_1
[else
    kod_2]
```

Jeżeli wyrażenie jest prawdziwe, wykona się `kod_1`, w przeciwnym razie wykona się `kod_2`. Jeżeli `kod_1` lub `kod_2` zawiera więcej niż jedną instrukcję, ujmujemy te instrukcje w blok `{ }` .

Spójrz na poniższy przykład:

```
if (x > 10)
    if (y > 20)
        Console.WriteLine("kod_1");
    else
        Console.WriteLine("kod_2");
```

W tym przykładzie na konsolę wypisany zostanie tekst `kod_2`, jeżeli warunek `y > 20` będzie miał wartość falsz. Jeżeli chcesz uzależnić `kod_2` od warunku `x > 10`, użyj klamer:

```
if (x > 10)
{
    if (y > 20)
```

```
        Console.WriteLine("kod_1");
    }
    else
        Console.WriteLine("kod_2");
```

W powyższym kodzie wyświetlony zostanie tekst kod\_2, gdy warunek  $x > 10$  będzie miał wartość falsz.

Jeżeli w instrukcji `if` sprawdzamy wartość wyrażenia typu logicznego, piszemy:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        bool b = true;

        if (b == true)
            Console.WriteLine("b jest prawdą");
        else
            Console.WriteLine("b jest fałszem");

    }
}
```

co jest równoważne skróconemu zapisowi:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        bool b = true;

        if (b)
            Console.WriteLine("b jest prawdą");
        else
            Console.WriteLine("b jest fałszem");

    }
}
```

Instrukcje warunkowe możemy dodatkowo zagnieździć w poniższy sposób:

```
if (wyrażenie_1)
    kod_1;
else if (wyrażenie_2)
    kod_2;
else if (wyrażenie_3)
    kod_3;
...
else
    kod_n;
```

Możemy dzięki temu sprawdzić kilka warunków i dla każdego z nich wykonać odpowiednie instrukcje programu.

Jest jeszcze jedna rzecz do opisania. Chodzi mianowicie o sprawdzenie kilku wyrażeń w jednej instrukcji `if`. Można wtedy użyć operatorów takich jak: `&&` (koniuunkcja), `||` (alternatywa) lub `!` (negacja). Zobacz poniższy przykład:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int i = 57;
        bool b = true;
        if (i > 0 && i < 100)
            Console.WriteLine("i jest z przedziału 0 do 100");
        i = -4;
        if (i == 0 || i < 0)
            Console.WriteLine("i jest równe 0 lub mniejsze od zera");
        if (!b == false)
            Console.WriteLine("b jest prawda");

    }
}
```

## 3.10. Instrukcja wyboru switch

Instrukcja ta służy do obsługi wielokrotnego wyboru. Przekazuje ona sterowanie do jednego z miejsc określonych przez `case`, jak w poniższym przykładzie:

```
string s = Console.ReadLine(); //Wczytaj linię tekstu z konsoli
int n = int.Parse(s); //Przekonwertuj tekst na liczbę
switch (n)
{
    case 1: //Jeżeli n == 1, wykonaj
        Console.WriteLine("Wybór 1");
        break;
    case 2: //Jeżeli n == 2, wykonaj
        Console.WriteLine("Wybór 2");
        break;
    case 3: //Jeżeli n == 3, wykonaj
        Console.WriteLine("Wybór 3");
        break;
    default:
        Console.WriteLine("Wybór domyślny");
        break;
}
```

Najpierw wczytujemy linię tekstu z konsoli do zmiennej `s` typu `string`. Następnie przekonwertowujemy zmienną `s` na liczbę całkowitą. Dalej następuje sprawdzanie wartości `n` i wykonywanie odpowiedniego kodu po słowie `case`. Słowo `break` przerywa

działanie instrukcji switch i wychodzi poza tę instrukcję. Zamiast break możemy też użyć instrukcji goto, która przekieruje kontrolę do określonego miejsca, jak w przykładzie poniżej:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Kawa: 1=Mała 2=Średnia 3=Duża");
        Console.Write("Twój wybór: ");
        string s = Console.ReadLine();
        int n = int.Parse(s);
        int cost = 0;
        switch (n)
        {
            case 1:
                cost += 25;
                break;
            case 2:
                cost += 25;
                goto case 1;
            case 3:
                cost += 50;
                goto case 1;
            default:
                Console.WriteLine("Błędny wybór. Wybierz 1, 2 lub 3.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine("Proszę wrzucić {0} złotych monet.", cost);
        }
        Console.WriteLine("Dziękuję.");
    }
}
```

## 3.11. Operatory

Język C# dostarcza zestaw operatorów, które są symbolami wykonującymi określone operacje w wyrażenях. Dodatkowo wiele operatorów może być przeciążonych dla typów zdefiniowanych przez programistę. Istnieje też coś takiego jak priorytet operatora, który określa, jaki operator ma pierwszeństwo przed innym. Niżej opisane operatory zostały ułożone od najwyższego priorytetu do najniższego.

## Podstawowe

### Operator x.y

Operator kropki jest używany do dostępu do składowych typu lub przestrzeni nazw. Na przykład często się go używa, aby mieć dostęp do metod pochodzących z bibliotek klas:

```
//Klasa Console jest w przestrzeni nazw System  
System.Console.WriteLine("witaj");
```

### Operator ()

Operator używany do określenia kolejności wykonywania działań w wyrażeniach, do wywoływanego funkcji, ale także do rzutowania typów, np.:

```
double x = 1234.7;  
int a:  
a = (int)x; //rzutowanie typu double na typ int
```

Operator ten nie może być przeciążony.

### Operator a[x]

Operator używany przy tablicach, indeksach i atrybutach. Może być także używany do wskaźników. Typ tablicy poprzedzamy znakami [ ]:

```
int[] t; //tablica liczb typu int  
t = new int[10]; //Utwórz tablicę 10-elementową
```

Jeżeli chcemy mieć dostęp do elementu tablicy, podajemy jego indeks w nawiasach kwadratowych:

```
t[3] = 7; //Element o indeksie 3 przyjmuje wartość 7
```

### Operator ++

Operator inkrementacji. Zwiększa wartość operandu o jeden. Może występować przed operandem (inkrementacja prefiksowa) lub po operandzie (inkrementacja postfiksowa).

Oto przykład:

```
using System;  
class MainClass  
{  
    static void Main()  
    {  
        double x;  
        x = 1.5;  
        Console.WriteLine(++x);  
        x = 1.5;  
        Console.WriteLine(x++);  
        Console.WriteLine(x);  
    }  
}
```

Wyjście przykładowego programu:

```
2.5  
1.5  
2.5
```

## Operator –

Operator dekrementacji. Zmniejsza wartość operandu o jeden. Podobnie jak operator inkrementacji, może być prefiksowy lub postfiksowy, np.:

```
using System;  
class MainClass  
{  
    static void Main()  
    {  
        double x;  
        x = 1.5;  
        Console.WriteLine(--x);  
        x = 1.5;  
        Console.WriteLine(x--);  
        Console.WriteLine(x);  
    }  
}
```

Wyjście:

```
0.5  
1.5  
0.5
```

## Operator new

Używa się go do tworzenia obiektów i wywoływania konstruktorów:

```
Class1 object1 = new Class1(); //Tworzy obiekt object1 klasy Class1
```

Operator ten jest także używany do wywoływania domyślnych konstruktorów typów:

```
int i = new int();
```

co jest równoważne zapisowi:

```
int i = 0;
```

## Operator typeof

Zwraca typ obiektu podanego jako parametr, np.:

```
System.Type type = typeof(int);
```

## Operator checked

Pozwala sprawdzić, czy nastąpiło przepelenienie przy operacjach arytmetycznych i konwersjach.

## Operator unchecked

Pozwala sprawdzić, czy nastąpiło przepelenie przy operacjach arytmetycznych i konwersjach. Ignoruje przepelenie.

## Operator ->

Operator ten używany jest do dereferencji i do dostępu do składowych. Operator ten może być użyty tylko w kodzie niezarządzanym. Przykład dla tego operatora:

```
using System;
struct Point
{
    public int x, y;
}

class MainClass
{
    unsafe static void Main()
    {
        Point pt = new Point();
        Point* pp = &pt;
        pp->x = 123;
        pp->y = 456;
        Console.WriteLine ( "{0} {1}", pt.x, pt.y );
    }
}
```



Więcej o wskaźnikach i kodzie nienadzorowanym w dalszej części książki.

## Jednoargumentowe

### Operator +

Operator ten jest używany do określenia znaku liczby. Jest też jednocześnie operatorem dodawania. Wartość operatora + i operandu jest po prostu wartością operandu. Operator ten jest też używany do łączenia dwóch napisów.

Przykład:

```
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine(+5);           //znak plus
        Console.WriteLine(5 + 5);        //dodawanie
        Console.WriteLine(5 + .5);       //dodawanie
        Console.WriteLine("5" + "5");    //łączenie napisów
        Console.WriteLine(5.0 + "5");    //łączenie napisów
        //Zauważ automatyczną konwersję z double na typ string
    }
}
```

## Operator -

Operator ten jest używany do określenia liczby ujemnej. Używa się go także do odejmowania oraz usuwania delegat. Można go także przeciążyć.

## Operator !

Operator negacji. Używany dla typów logicznych. Zwraca fałsz, jeżeli operand miał wartość prawda, i odwrotnie.

## Operator ~

Odwraca bity w podanym operandzie, np.:

```
using System;
class MainClass
{
    static void Main()
    {
        int[] values = { 0, 0x111, 0xfffff, 0x8888, 0x22000022 };
        foreach (int v in values)
        {
            Console.WriteLine("~0x{0:x8} = 0x{1:x8}", v, ~v);
        }
    }
}
```

Wyjście programu:

```
~0x00000000 = 0xffffffff
~0x00000111 = 0xffffffffee
~0x000fffff = 0xffff0000
~0x00008888 = 0xffff7777
~0x22000022 = 0xddffffdd
```

## Operator &

Zwraca adres operandu, jeżeli pracujemy ze wskaźnikami. Jest to też operator koniunkcji przy wyrażeniach logicznych. Przykład:

```
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine(true & false); // logical and
        Console.WriteLine(true & true); // logical and
        Console.WriteLine("0x{0:x}", 0xf8 & 0x3f); // bitwise and
    }
}
```

Wyjście programu:

```
False
True
0x38
```

## Operator sizeof

Zwraca rozmiar typu w bajtach. Na przykład pobranie rozmiaru typu int wygląda tak:

```
int intSize = sizeof(int);
```

## Mnożenie, dzielenie i modulo

### Operator \*

Operator mnożenia. Jest to także operator dereferencji, gdy pracujemy ze wskaźnikami.

### Operator /

Dzieli pierwszy operand przez drugi.

### Operator %

Zwraca resztę z dzielenia pierwszego operandu przez drugi.

## Przesunięcia

### Operator <<

Przesuwa bity pierwszego operandu w lewo o ilość podaną jako drugi operand. Drugi operand musi być typu int.

Przykład:

```
using System;
class MainClass
{
    static void Main()
    {
        int i = 1;
        long lg = 1;
        Console.WriteLine("0x{0:x}", i << 1);
        Console.WriteLine("0x{0:x}", i << 33);
        Console.WriteLine("0x{0:x}", lg << 33);
    }
}
```

### Operator >>

Przesuwa bity pierwszego operandu w prawo o ilość podaną jako drugi operand.

Przykład:

```
using System;
class MainClass
{
```

```
static void Main()
{
    int i = -1000;
    Console.WriteLine(i >> 3);
}
```

## Relacje i sprawdzanie typów

### Operator <

Operator porównuje dwa operandy i zwraca prawdę, jeżeli pierwszy operand jest mniejszy od drugiego — w przeciwnym wypadku zwraca fałsz.

### Operator >

Operator porównuje dwa operandy i zwraca prawdę, jeżeli pierwszy operand jest większy od drugiego — w przeciwnym wypadku zwraca fałsz.

### Operator <=

Operator porównuje dwa operandy i zwraca prawdę, jeżeli pierwszy operand jest mniejszy lub równy od drugiego — w przeciwnym wypadku zwraca fałsz.

### Operator >=

Operator porównuje dwa operandy i zwraca prawdę, jeżeli pierwszy operand jest większy od drugiego — w przeciwnym wypadku zwraca fałsz.

### Operator is

Sprawdza, czy obiekt jest kompatybilny z podanym typem. Na przykład jeżeli chcemy sprawdzić, czy obiekt jest kompatybilny z typem `string`, piszemy:

```
if (obj is string)
{
}
```

### Operator as

Używany do konwersji pomiędzy kompatybilnymi typami referencyjnymi, np.:

```
string s = someObject as string;
if (s != null)
{
    //someObject jest typu string.
}
```

## Równość i różność

### Operator ==

Operator zwraca prawdę, jeżeli podane operandy są równe — w przeciwnym wypadku zwraca fałsz.

### Operator !=

Operator zwraca prawdę, jeżeli podane operandy są różne — w przeciwnym wypadku zwraca fałsz.

## Koniunkcja logiczna

Zobacz „Operator &”.

## Alternatywa wykluczająca logiczna

Wykonuje alternatywę wykluczającą na podanych operandach. Np.:

```
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine(true ^ false); //alternatywa wykluczająca logiczna
        Console.WriteLine(false ^ false); //alternatywa wykluczająca logiczna
        //alternatywa wykluczająca bitowa:
        Console.WriteLine("0x{0:x}", 0xf8 ^ 0x3f);
    }
}
```

## Alternatywa logiczna

Wykonuje alternatywę na podanych operandach.

Przykład:

```
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine(true | false); //alternatywa logiczna
        Console.WriteLine(false | false); //alternatywa logiczna
        Console.WriteLine("0x{0:x}", 0xf8 | 0x3f); //alternatywa bitowa
    }
}
```

## Koniunkcja warunkowa

Wykonuje koniunkcję logiczną na podanych operandach.

Operacja:

$x \&& y$

jest równoważna:

$x \& y$

Wyjątek: gdy  $x$  jest falszem, wówczas wartość  $y$  nie jest sprawdzana, ponieważ niezależnie od tego, jaka by nie była ta wartość, to wartość wyrażenia i tak będzie falszem.

## Alternatywa warunkowa

Wykonuje alternatywę logiczną na podanych operandach.

Operacja:

$x \mid\mid y$

jest równoważna:

$x \mid y$

Wyjątek: gdy  $x$  jest prawdą, wówczas wartość  $y$  nie jest sprawdzana, ponieważ niezależnie od tego, jaka by nie była ta wartość, to wartość wyrażenia i tak będzie prawdą.

## Operator warunkowy

Sprawdza warunek i zwraca wartość pierwszą lub drugą. Składnia jest następująca:

warunek ? piersze\_wyrażenie : drugie\_wyrażenie;

Jeżeli warunek jest prawdziwy, zwracane jest pierwsze wyrażenie, gdy fałszywy — to drugie wyrażenie.

Na przykład:

```
string resultString = (myInteger < 10) ? "Mniejsze od 10"  
                                : "Większe lub równe 10";
```

## Przypisania

### Operator =

Operator ten przypisuje wartość z prawej strony do wyrażenia z lewej strony. Operandy muszą być typu kompatybilnego lub typu możliwego do przekonwertowania.

Przykład:

```
using System;
class MainClass
{
    static void Main()
    {
        double x;
        int i;
        i = 5; //przypisanie typu int do typu int
        x = i; //niejawnia konwersja int na double
        i = (int)x; //Wymaga rzutowania
    }
}
```

## Skrócone operatory przypisania

Język C# pozwala na skrócony zapis operatorów przypisania.

Na przykład zapis:

`x = x + 5;`

jest równoważny zapisowi:

`x += 5;`

Tak samo jest dla operatorów:

- ◆ `+=`
- ◆ `-=`
- ◆ `*=`
- ◆ `/=`
- ◆ `%=`
- ◆ `&=`
- ◆ `|=`
- ◆ `^=`
- ◆ `<<=`
- ◆ `>>=`

## Operator ??

Zwraca lewy operand, jeżeli wyrażenie jest różne od zera — w przeciwnym wypadku zwraca prawy operand.

Przykład:

```
int? x = null;
int y = x ?? -1; //y będzie miało wartość -1, bo x ma wartość null
```

## 3.12. Pętle

W programowaniu często zachodzi potrzeba wykonania czegoś więcej niż jeden raz. Pomogą nam w tym niżej opisane pętle.

### Pętla do-while

Pętla ta wykonuje instrukcję lub blok instrukcji, dopóki podane wyrażenie nie stanie się fałszywe. Ciało pętli powinno być ujęte w nawiasy klamrowe, chyba że zawiera jedną instrukcję, wtedy nawiasy są opcjonalne.

W poniższym przykładzie pętla wykonuje się, dopóki wartość zmiennej `x` jest mniejsza niż 5:

```
using System;

class Program
{
    static void Main()
    {
        int x = 0;
        do
        {
            Console.WriteLine(x);
            x++;
        } while (x < 5);

    }
}
```

Wyjście:

```
0
1
2
3
4
```

Pętla ta zawsze wykona się przynajmniej raz, gdyż jej warunek sprawdzany jest na końcu pętli. Możesz ją albo przerwać słowem `break`, albo przejść do sprawdzania wartości wyrażenia słowem `continue`, albo wyjść z pętli za pomocą jednego ze słów: `goto`, `return` lub `throw`.

### Pętla for

Dzięki tej pętli możesz wykonać daną instrukcję lub blok instrukcji, dopóki podane wyrażenie nie będzie fałszywe. Ten rodzaj pętli jest przydatny przy korzystaniu z tablic oraz wszędzie tam, gdzie wiemy dokładnie, ile razy pętla ma się wykonać.

W poniższym przykładzie zmienna *i* jest wypisywana na konsolę i zwiększana o 1 przy każdej iteracji:

```
using System;

class Program
{
    static void Main()
    {
        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine(i);
        }
    }
}
```

Wyjście:

```
1
2
3
4
5
```

Pętla **for** w poprzednim przykładzie wykonuje następujące akcje:

1. Najpierw jest ustawiana wartość inicjalizująca zmienną *i*. Dzieje się to tylko raz, na początku pętli, niezależnie od tego, ile razy pętla się wykona.
2. Sprawdzany jest warunek *i <= 5*.
3. Jeżeli wartość *i* jest mniejsza od 5 lub równa 5, wykonuje się następujące akcje:
  - ◆ Metoda `Console.WriteLine` wypisuje na ekran wartość zmiennej *i*.
  - ◆ Wartość zmiennej *i* jest zwiększana o 1 (inkrementacja).
  - ◆ Sterowanie jest przekazywane do punktu 2, czyli warunek sprawdzany jest ponownie.



Zauważ, że jeżeli w powyższym przykładzie wartość inicjalizująca zmienną licznikową byłaby większa od 5, to pętla nie wykonałaby się ani razu.

Każda pętlę **for** można podzielić na sekcje: inicjalizującą, warunkową i iteracyjną.

```
for (inicjalizacja; warunek; iterator)
{
    //wnętrze pętli
}
```

W skład sekcji inicjalizującej może wchodzić:

- ◆ Deklaracja i inicjalizacja lokalnej zmiennej licznika pętli. (Zmienna ta jest lokalna i nie ma do niej dostępu spoza pętli).

- ♦ Sekcja ta może też zawierać inne operacje, oddzielone przecinkiem, takie jak:
  - ♦ przypisanie,
  - ♦ wywołanie metody,
  - ♦ inkrementacja lub dekrementacja,
  - ♦ utworzenie obiektu słowem new,
  - ♦ wyrażenie await (usypia wykonywanie metody, dopóki nie wykonają się oczekujące zadania).

Sekcja warunkowa zawiera wyrażenie logiczne, od którego wartości zależy, czy pętla ma się wykonać jeszcze raz, czy zakończyć.

Sekcja iteracyjna decyduje o tym, co ma się wykonać po każdym obrocie pętli (iteracji), i może zawierać:

- ♦ przypisanie,
- ♦ wywołanie metody,
- ♦ inkrementację lub dekrementację,
- ♦ utworzenie obiektu słowem new,
- ♦ wyrażenie await (usypia wykonywanie metody, dopóki nie wykonają się oczekujące zadania).

Wnętrze pętli może zawierać instrukcję lub kilka instrukcji, które należy umieścić w nawiasach klamrowych.

Poprzedni przykład prezentował typowe zastosowanie pętli for. W kolejnym przykładzie pokazana zostanie pętla zawierająca:

- ♦ przypisanie zewnętrznej zmiennej wartości,
- ♦ wywołanie metody Console.WriteLine w sekcji inicjalizującej i iteracyjnej,
- ♦ zmienianie wartości dwóch zmiennych w sekcji iteracyjnej.

A oto przykład:

```
using System;

class Program
{
    static void Main()
    {
        int i;
        int j = 10;
        for (i = 0, Console.WriteLine("Start: {0}", i);
              i < j;
              i++, j--, Console.WriteLine("i={0}, j={1}", i, j))
        {
            //wnętrze pętli
        }
    }
}
```

Wyjście programu:

```
Start: 0
i=1, j=9
i=2, j=8
i=3, j=7
i=4, j=6
i=5, j=5
```

Wszystkie wyrażenia w pętli `for` są opcjonalne, dlatego nie musimy ich podawać. Pętla bez wyrażeń wygląda tak jak poniżej i jest pętlą nieskończoną:

```
for (: : )
{
    //...
}
```

## Pętla foreach

Pętla `foreach` powtarza grupę wbudowanych instrukcji dla każdego elementu tablicy lub kolekcji, która implementuje interfejs `System.Collections.IEnumerable` lub `System.Collections.Generic.IEnumerable<T>`. Może być ona użyta do iterowania po elementach kolekcji, ale nie do dodawania lub usuwania elementów — do tego użyj pętli `for`.

Pętlę `foreach` możesz przerwać słowem `break`, możesz przejść do następnej iteracji słowem `continue` albo wyjść z pętli za pomocą jednego ze słów: `goto`, `return` lub `throw`.

Oto prosty przykład iterowania po elementach tablicy i wyświetlania ich:

```
using System;

class Program
{
    static void Main()
    {
        int[] array1 = new int[] { 1, 2, 3, 4, 5 };
        foreach (int i in array1)
        {
            System.Console.WriteLine(i);
        }
    }
}
```

Wyjście:

```
1
2
3
4
5
```

Więcej o pracy z tablicami i kolekcjami w dalszej części książki.

## Pętla while

Wykonuje instrukcję lub blok instrukcji, dopóki określone wyrażenie nie będzie fałszywe.

Przykład:

```
using System;

class Program
{
    static void Main()
    {
        int n = 1;
        while (n < 6)
        {
            Console.WriteLine("Aktualna wartość n to {0}", n);
            n++;
        }
    }
}
```

Wyjście:

```
Aktualna wartość n to 1
Aktualna wartość n to 2
Aktualna wartość n to 3
Aktualna wartość n to 4
Aktualna wartość n to 5
```

Ponieważ sprawdzanie wyrażenia następuje przed instrukcjami, pętla wykona się zero lub więcej razy (w przeciwieństwie do pętli do-while, która wykona się jeden lub więcej razy).

Z pętli można wyjść za pomocą jednego ze słów: break, goto, return lub throw, a przejść do następnej iteracji — słowem continue.

## Kontrola przepływu

### break

Słowo to zamyka najbliższą pętlę lub instrukcję switch, w której się znajduje.

Przykład dla pętli for, która liczy od 1 do 100, ale przy 5 słowo break przerwuje pętlę:

```
using System;

class Program
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {

```

```
        if (i == 5) //Gdy i == 5, przerwij
        {
            break;
        }
        Console.WriteLine(i);
    }
}
```

Wyjście:

```
1
2
3
4
```

## continue

Przekazuje sterowanie do kolejnej iteracji w pętli, w której się znajduje.

Przykład:

```
using System;

class Program
{
    static void Main()
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i < 9)
            {
                continue;
            }
            Console.WriteLine(i);
        }
    }
}
```

Wyjście:

```
9
10
```

## goto

Przekazuje kontrolę do określonej etykiety. Bardzo przydatne, gdy chcemy wyjść z kilku zagnieżdżonych pętli.

Przykład:

```
using System;

class Program
{
    static void Main()
```

```
{  
    for (int i = 1; i <= 10; i++)  
    {  
        for (int j = 1; j <= 10; j++)  
            //Gdy i == 7 oraz j == 9,  
            //skocz do etykiety labell  
            if (i == 7 && j == 9)  
                goto labell;  
    }  
    labell:  
    Console.WriteLine("jestem poza pętlą");  
}
```

Wyjście:

```
jestem poza pętlą
```

## return

Kończy wykonywanie metody, w której występuje, i przekazuje kontrolę do metody wywołującej. Może być też użyte do wyjścia z pętli.

Przykład z metodą obliczającą pole koła o podanym promieniu:

```
using System;  
  
class Program  
{  
    static double CalculateArea(int r)  
    {  
        double area = r * r * Math.PI;  
        return area; //Zwróć wynik obliczeń  
    }  
  
    static void Main()  
    {  
        int radius = 5; //wartość promienia  
        double result = CalculateArea(radius);  
        Console.WriteLine("Pole wynosi {0:0.00}", result);  
    }  
}
```

Wyjście:

```
Pole wynosi 78.54
```



Więcej o metodach i ich argumentach w dalszej części książki.

## throw

Służy do rzucania wyjątków, czyli obsługi sytuacji anormalnych. Jaki to ma związek z pętlami? Otóż wyjątek przerwie wykonywanie pętli.

Prosty przykład rzucania wyjątku wyjścia poza zakres tablicy:

```
using System;

class Program
{
    static int GetNumber(int index)
    {
        int[] nums = { 300, 600, 900 };
        if (index > nums.Length)
        {
            throw new IndexOutOfRangeException();
        }
        return nums[index];
    }

    static void Main()
    {
        //Pobieramy liczbę o indeksie 3, ale tablica
        //o nazwie nums ma tylko indeksy 0, 1 oraz 2
        //Spowoduje to rzucenie wyjątku
        int result = GetNumber(3);
    }
}
```

Więcej o wyjątkach i obsłudze błędów w dalszej części książki.

## 3.13. Argumenty wiersza poleceń

Wywołując program konsolowy z wiersza poleceń, możemy podać mu argumenty, np. nazwę pliku do przetworzenia itp.

Argumenty znajdują się w tablicy napisów w funkcji `Main`. Pobrać je możemy np. pętlą `for` jak w poniższym przykładzie:

```
using System;
public class CommandLine
{
    public static void Main(string[] args)
    {
        //Właściwość Length jest użyta do określenia długości tablicy
        //Zauważ, że ta właściwość jest tylko do odczytu
        Console.WriteLine("Ilosc argumentow wiersza polecenia = {0}",
            args.Length);
        for(int i = 0; i < args.Length; i++)
        {
            Console.WriteLine("Arg[{0}] = [{1}]".i, args[i]);
        }
    }
}
```

Możemy to też zrobić, używając pętli `foreach`, tak jak poniżej:

```
using System;
public class CommandLine2
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Ilosc argumentow wiersza polecenia = {0}",
            args.Length);
        foreach(string s in args)
        {
            Console.WriteLine(s);
        }
    }
}
```

Zmienna `args` jest tablicą napisów — jak już wcześniej wspominałem — i do jej elementów można się dostać, używając operatora `[]`. Oczywiście należy pamiętać, że indeksy tablicy numerowane są od zera.

## 3.14. Metody

Metoda to blok kodu zawierający ciąg instrukcji. Program wykonuje instrukcje poprzez wywołanie metody i określenie wymaganych parametrów (chyba że metoda ich nie wymaga). W języku C# każda instrukcja jest wykonywana w kontekście metody. Metoda o nazwie `Main` jest punktem wejścia każdej aplikacji napisanej w C# i jest wywoływaną przez CLR (*Common Language Runtime*).

### Deklaracja metod

Metody deklarujemy według następującego szablonu:

```
typ_wyniku nazwa_metody(typ_parametru_1 nazwa_parametru_1, ..., typ_parametru_n
    ↗nazwa_parametru_n)
{
    //Instrukcje
    return wartosc_zwracana;
}
```

Najpierw piszemy typ wyniku, jaki zwraca dana metoda. Jeżeli ma nie zwracać wyniku, piszemy tam typ `void`. Dalej występuje nazwa metody, a po niej w nawiasach typy i nazwy poszczególnych parametrów oddzielone przecinkami (chyba że metoda nie przyjmuje argumentów). Nazwę metody i listę parametrów nazywamy sygnaturą. Już tutaj wprowadzam to pojęcie, aby w późniejszych rozdziałach było wiadomo, o czym jest mowa. Na końcu w klamrach podajemy instrukcje, a po nich słowo `return` i wynik, jaki metoda ma zwrócić (chyba że wartość zwracana jest typu `void`, to nie wymaga się wtedy słowa `return`).

Metodę wywołujemy, podając jej nazwę i nawiasy, w których znajdują się argumenty oddzielone przecinkiem. Wywołanie kończymy średnikiem. Jeżeli chcemy przypisać wynik metody do zmiennej, piszemy:

```
var1 = Method1(arg1, arg2); //przypisanie wyniku metody Method1 zmiennej var1
```

Przykładowy program z metodą o nazwie `Square`, która liczbę przyjętą za argument podnosi do potęgi drugiej:

```
using System;

class Program
{
    //Metoda podnosi liczbę do potęgi drugiej
    static int Square(int a)
    {
        return a * a;
    }

    static void Main()
    {
        Console.WriteLine("{0}*{0} = {1}", 5, Square(5));
    }
}
```

Wyjście:

```
5*5 = 25
```

Pewnie zauważyłeś przed metodą słowo `static`. Więcej na ten temat będzie dalej (w rozdziale o programowaniu obiektowym), ale w prostych słowach można powiedzieć, że metoda, która nie jest wywoływana na rzecz obiektu, musi być statyczna.

## Przekazywanie przez referencję lub przez wartość

W języku C# argumenty mogą być przekazywane do parametrów metody przez referencję lub przez wartość. Przekazywanie przez referencję pozwala zmodyfikować wartość parametru, co nastąpi w środowisku wywołania. Jeżeli chcemy przekazać parametr przez referencję, należy użyć słowa kluczowego `ref` lub `out` (różnica między tymi dwoma slowami jest taka, że `ref` wymaga inicjalizacji zmiennej przed jej przekazaniem).

Oto przykład pokazujący przekazywanie przez referencję oraz przez wartość:

```
using System;

class Program
{
    static void Main()
    {
        int arg;

        //przekazanie przez wartość
        //Wartość arg w Main jest niezmieniona
        arg = 4;
        squareVal(arg);
    }
}

int squareVal(int arg)
{
    arg *= arg;
    return arg;
}
```

```
Console.WriteLine(arg);
//wyjście: 4

//przekazanie przez referencję
//wartość arg w Main jest zmieniona
arg = 4;
squareRef(ref arg);
Console.WriteLine(arg);
//wyjście: 16
}

static void squareVal(int valParameter)
{
    valParameter *= valParameter;
}

//przekazanie przez referencję
static void squareRef(ref int refParameter)
{
    refParameter *= refParameter;
}
}
```

## 3.15. Tablice

Tablica to struktura danych zawierająca kilka zmiennych tego samego typu. Deklarujemy ją następująco:

```
type[] arrayName;
```

Poniższy przykład prezentuje deklarację tablic różnych rodzajów:

```
using System;

class Program
{
    static void Main()
    {
        //deklaracja tablicy jednowymiarowej
        int[] array1 = new int[5];

        //deklaracja tablicy jednowymiarowej z podanymi wartościami
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        //alternatywna składnia do powyższej deklaracji
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        //deklaracja tablicy dwuwymiarowej
        int[,] multiDimensionalArray1 = new int[2, 3];

        //deklaracja tablicy dwuwymiarowej z podanymi wartościami
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        //deklaracja tablicy tablic
        int[][] jaggedArray = new int[6][];
    }
}
```

```
//ustawienie wartości pierwszej tablicy w tablicy tablic  
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };  
}  
}
```

Tablica ma następujące właściwości:

- ◆ Może być jednowymiarowa, wielowymiarowa lub być tablicą tablic.
- ◆ Domyślne wartości elementów tablicy są ustawiane na zero, a elementów referencyjnych na null.
- ◆ Tablica tablic ma elementy typu referencyjnego, które są inicjalizowane wartością null.
- ◆ Tablice są indeksowane od zera: tablica n elementów jest indeksowana od 0 do n-1.
- ◆ Elementy tablicy mogą być różnego typu, nawet typu Array.
- ◆ Typy tablic są typami referencyjnymi pochodnymi od abstrakcyjnego typu Array. Ponieważ ten typ implementuje IEnumarable oraz IEnumarable<T>, możesz używać pętli foreach na każdej tablicy w języku C#.

## Przekazywanie tablic jako argumentów metod

Tablice mogą być przekazane jako argumenty do parametrów metody.

Przykład programu wyświetlającego tablicę jednowymiarową na ekranie:

```
using System;  
  
class Program  
{  
    static void PrintArray(string[] arr)  
    {  
        for (int i = 0; i < arr.Length; i++)  
        {  
            System.Console.Write(arr[i] + "{0}", i < arr.Length - 1 ? " " : "");  
        }  
        System.Console.WriteLine();  
    }  
  
    static void Main()  
    {  
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };  
        PrintArray(weekDays);  
    }  
}
```

Przykład programu wyświetlającego tablicę dwuwymiarową na ekranie:

```
using System;  
  
class Program
```

```
{  
    static void Print2DArray(int[,] arr)  
    {  
        //Wypisz elementy tablicy  
        for (int i = 0; i < arr.GetLength(0); i++)  
        {  
            for (int j = 0; j < arr.GetLength(1); j++)  
            {  
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);  
            }  
        }  
    }  
  
    static void Main()  
    {  
        int [,] array1 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };  
        Print2DArray(array1);  
    }  
}
```



Podobnie jak pojedyncze zmienne, tablice mogą być również przekazywane przez wartość lub przez referencję.

## Klasa System.Array

Wszystkie tablice w C# dziedziczą po tej klasie, dlatego możemy łatwo operować na tablicach, używając metod i właściwości udostępnianych przez tę klasę. Poniżej opis najważniejszych właściwości i metod. Więcej informacji znajdziesz na stronach MSDN.

### Właściwość Length

Zwraca długość tablicy (ilość elementów).

Przykład użycia:

```
string[] days = new string[] { "Mo", "Tu", "We", "Th", "Fr" };  
Console.WriteLine(days.Length); //Wypisze 5
```

### Właściwość Rank

Zwraca ilość wymiarów tablicy.

Przykład użycia:

```
int[,] array1 = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };  
Console.WriteLine(array1.Rank); //Wypisze 2
```

### Metoda BinarySearch()

Wyszukiwanie binarne w tablicy. Pierwszy parametr to tablica, a drugi to szukana wartość. Funkcja zwraca numer indeksu, na którym występuje szukana wartość. Gdy nie znajdzie wartości, zwraca liczbę ujemną.

Przykład użycia:

```
string[] days = new string[] { "Mo", "Tu", "We", "Th", "Fr" };
Console.WriteLine(Array.BinarySearch(days, "We")); //Wypisze 2
```

## Metoda Clear()

Ustawia określoną ilość elementów w tablicy na wartość zero, false lub null, zależnie od typu elementu. Pierwszy parametr to tablica, drugi indeks początkowy, a trzeci indeks końcowy.

Przykład użycia:

```
string[] days = new string[] { "Mo", "Tu", "We", "Th", "Fr" };
Array.Clear(days, 0, days.Length);
```

## Metoda Clone()

Tworzy kopię tablicy, z której została wywołana.

Przykład użycia:

```
string[] days = new string[] { "Mo", "Tu", "We", "Th", "Fr" };
string[] days2 = (string[]) days.Clone();
```

## Metoda Copy()

Kopiuje określoną ilość elementów z jednej tablicy do drugiej. Pierwszy parametr to tablica źródłowa, drugi to tablica docelowa, a trzeci to ilość elementów.

Przykład użycia:

```
string[] days = new string[] { "Mo", "Tu", "We", "Th", "Fr" };
string[] days2 = new string[days.Length];
Array.Copy(days, days2, days.Length);
```

## Metoda Find()

Przeszukuje tablicę w celu znalezienia pierwszego elementu spełniającego podane kryteria. Kryteria określenia elementu jako pasującego zależą od napisanej przez nas metody sprawdzającej.

Przykład użycia (wyszukuje pierwszą liczbę parzystą w tablicy):

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = new int[] { 3, 8, 7, 5 };
        int first = Array.Find(numbers, FindNum);
        Console.WriteLine(first); //Wypisze 8
    }
}
```

```
static bool FindNum(int a)
{
    if (a % 2 == 0)
        return true;
    else
        return false;
}
```

## Metoda **FindAll()**

Przeszukuje tablicę w celu znalezienia wszystkich elementów spełniających podane kryteria.

Przykład użycia (wyszukuje w tablicy wszystkie liczby parzyste):

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = new int[] { 3, 8, 7, 5, 2, 4 };
        int[] even = Array.FindAll(numbers, FindNum);
        for (int i = 0; i < even.Length; i++)
            Console.WriteLine(even[i]);
    }

    static bool FindNum(int a)
    {
        if (a % 2 == 0)
            return true;
        else
            return false;
    }
}
```

## Metoda **Initialize()**

Inicjalizuje tablicę wartościami zero, `false` lub `null`, zależnie od typu elementu.

Przykład użycia:

```
array1.Initialize();
```

## Metoda **IndexOf()**

Zwraca indeks określonego elementu.

Przykład użycia:

```
string[] days = new string[] { "Mo", "Tu", "We", "Th", "Fr" };
Console.WriteLine(Array.IndexOf(days, "Th")); //Wypisze 3
```

## Metoda Resize()

Zmienia rozmiar tablicy. Pierwszy parametr to nazwa tablicy poprzedzona słowem `ref`, a drugi to nowy rozmiar dla tablicy.

Przykład użycia:

```
Array.Resize(ref array1, array1.Length + 5);
```

## Metoda Reverse()

Odwracia kolejność elementów tablicy.

Przykład użycia:

```
int[] array1 = new int[] { 1, 2, 3, 4, 5 };
foreach (int i in array1)
    Console.Write(i + " ");
Console.WriteLine();
Array.Reverse(array1); //Odwróć tablicę
foreach (int i in array1)
    Console.Write(i + " ");
```

## Metoda Sort()

Sortuje elementy tablicy jednowymiarowej.

Przykład użycia:

```
int[] array1 = new int[] { 7, 2, 4, 1, 5 };
Array.Sort(array1); //Sortuj tablicę
foreach (int i in array1)
    Console.Write(i + " ");
```

# 3.16. Wskaźniki

## Kod nienadzorowany (ang. unsafe code)

Język C# domyślnie nie wspiera arytmetyki wskaźników. Jednak używając słowa kluczowego `unsafe`, można zdefiniować kontekst, w którym będą używane wskaźniki.

Kod ten ma następujące właściwości:

- ◆ Metody, typy i bloki kodu mogą być zdefiniowane jako `unsafe`.
- ◆ W niektórych przypadkach kod ten może zwiększyć wydajność poprzez brak sprawdzania rozmiarów tablic.
- ◆ Kod `unsafe` jest wymagany, gdy wywołujesz natywne funkcje, które wymagają wskaźników.

- ◆ Kod ten może wywołać zagrożenie bezpieczeństwa i stabilności działania aplikacji.
- ◆ Aby skompilować taki kod, należy użyć polecenia dla kompilatora: /unsafe.

## Typy wskaźnikowe

Typ wskaźnikowy deklarujemy w następujący sposób:

```
type* identifier;  
void* identifier; //dostępne, ale niepolecane
```

Typem wskaźnikowym może być: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, typ wyliczeniowy `enum` lub struktura składająca się z niezarządzanych typów.

Typy wskaźnikowe nie dziedziczą z typu obiektu i nie ma możliwości konwersji pomiędzy typem wskaźnikowym a obiektem.

Deklaracja kilku wskaźników:

```
int* p1, p2, p3; //dobrze  
int *p1, *p2, *p3; //Nie zadziała w C#
```

Przykładowe deklaracje wskaźników:

```
int* p; //wskaźnik na typ int  
int** p; //wskaźnik na wskaźnik na typ int  
int*[] p; //jednowymiarowa tablica wskaźników na int  
char* p; //wskaźnik na typ char  
void* p; //wskaźnik na typ nieokreślony
```

Oto prosty przykład, który pozwoli zrozumieć podstawowe operacje na wskaźnikach:

```
using System;  
  
class Program  
{  
    static unsafe void Main()  
    {  
        int x = 100; //zmienna x o wartości 100  
        int* ptr = &x; //Wskaźnikowi na int przypisujemy adres zmiennej x  
        Console.WriteLine((int)&ptr); //Wyświetlamy adres (musimy rzutować na int)  
        Console.WriteLine(*ptr); //Wyświetlamy wartość  
    }  
}
```

## Operator \*

Operator dereferencji, czyli pobrania wartości z określonego adresu.

**Operator ->**

Operator dostępu do pola struktury przez wskaźnik.

**Operator []**

Operator indeksowania.

**Operator &**

Operator określający adres zmiennej.

**Operator ++ i -**

Inkrementacja i dekrementacja wskaźnika.

**Operator + i -**

Arytmetyka wskaźników.

**Operator ==, !=, <, >, <=, >=**

Porównywanie wskaźników.

## Rozdział 4.

# Język C#. Programowanie obiektowe

## 4.1. Klasy i obiekty

Klasa jest konstrukcją, która pozwala tworzyć własne typy poprzez grupowanie ze sobą zmiennych różnego typu, metod i innych składowych. Jeżeli klasa nie jest stała, to w kodzie klienta możesz tworzyć obiekty tej klasy. Aby zilustrować, czym jest klasa, a czym obiekt, spróbuj sobie wyobrazić następującą rzecz:

Klasą jest samochód i ma on takie cechy jak np. model czy moc silnika. Obiektem klasy jest wtedy konkretny samochód o określonych cechach.

Klasę deklarujemy, używając słowa kluczowego `class`, tak jak w poniższym przykładzie:

```
public class Car
{
}
```

Słowo kluczowe `class` jest poprzedzone słowem `public`, jest to modyfikator dostępu. Modyfikatory dostępu zostaną szerzej opisane w dalszej części. Na razie wystarczy wiedzieć, że słowo `public` przed klasą pozwala każdemu tworzyć obiekty tej klasy. W nawiasach klamrowych będą składowe klasy, takie jak np. pola czy metody.

Zadeklarujmy sobie teraz jakieś pola i metody dla klasy `Car` (samochód):

```
public class Car
{
    public string Model;
    public int Power;
```

```
public void StartEngine()
{
    Console.WriteLine("Silnik uruchomiony.");
}
public void Drive(int speed)
{
    Console.WriteLine("Poruszam się z prędkością {0} km/h.", speed);
}
public void Stop()
{
    Console.WriteLine("Samochód zatrzymany. Silnik wyłączony.");
}
}
```

W powyższym przykładzie zadeklarowaliśmy dwa pola i cztery metody. Każdy samochód ma jakiś model, moc, można uruchomić jego silnik, jechać nim i zatrzymać się. Teraz stwórzmy sobie obiekt, czyli konkretny samochód o określonych cechach:

```
Car mySubaru = new Car();
```

Ustawmy model i moc silnika naszego samochodu:

```
mySubaru.Model = "Subaru Impreza";
mySubaru.Power = 300;
```

Jak widać, nasze auto to Subaru Impreza, a moc jego silnika to 300 koni mechanicznych.

Myślę, że czas wypróbować naszą maszynę, zapalmy więc silnik i przejedźmy się:

```
mySubaru.StartEngine();
mySubaru.Drive(120);
mySubaru.Stop();
```

Na konsoli programu powinieneś zobaczyć:

```
Silnik uruchomiony.
Poruszam się z prędkością 120 km/h.
Samochód zatrzymany. Silnik wyłączony.
```

Oto cały kod przykładu:

```
using System;

public class Car
{
    public string Model;
    public int Power;
    public void StartEngine()
    {
        Console.WriteLine("Silnik uruchomiony.");
    }
    public void Drive(int speed)
    {
        Console.WriteLine("Poruszam się z prędkością {0} km/h.", speed);
    }
    public void Stop()
    {
        Console.WriteLine("Samochód zatrzymany. Silnik wyłączony.");
    }
}
```

```
        }

    class Program
    {
        static void Main()
        {
            Car mySubaru = new Car();
            mySubaru.Model = "Subaru Impreza";
            mySubaru.Power = 300;
            mySubaru.StartEngine();
            mySubaru.Drive(120);
            mySubaru.Stop();
        }
    }
```

## Słowo kluczowe this

Pozwala odwołać się do bieżącej instancji klasy. Jeżeli mamy pola w klasie o takiej samej nazwie jak parametry metody ustalającej wartości tym polem, to wtedy dobrze jest użyć `this`:

```
using System;

public class Class1
{
    private int A;
    private int B;
    public void SetAB(int A, int B)
    {
        this.A = A;
        this.B = B;
    }
    public void GetAB()
    {
        Console.WriteLine("A = {0}, B = {1}", A, B);
    }
}

class Program
{
    static void Main()
    {
        Class1 class1 = new Class1();
        class1.SetAB(2, 4);
        class1.GetAB();
    }
}
```

## 4.2. Konstruktor i destruktor

Kiedy tworzona jest klasa lub struktura, wywoływany jest konstruktor. Klasa lub struktura może mieć kilka konstruktorów, które przyjmują różne argumenty. Podstawowym zadaniem konstruktora jest ustawienie wartości domyślnych dla pól.

Dla klasy z poprzedniego podrozdziału utworzymy sobie konstruktor, który jako argument przyjmie model samochodu:

```
public Car(string _Model)
{
    Model = _Model;
}
```

Konstruktor to tak jakby metoda o nazwie takiej samej jak klasa.

Teraz, gdy mamy konstruktor, tworzenie obiektu wygląda następująco:

```
Car mySubaru = new Car("Subaru Impreza");
```

Zatem teraz linia:

```
mySubaru.Model = "Subaru Impreza";
```

nie jest potrzebna, gdyż konstruktor przy tworzeniu obiektu ustawi nam model naszego samochodu.

Jeżeli przed wykonaniem operacji chcemy wywołać inny konstruktor tej samej klasy, to piszemy:

```
using System;

public class A
{
    public A(int i)
    {
        System.Console.WriteLine("i = " + i + "\n");
    }
    public A(char c) : this(5) //Najpierw wywoła konstruktor A(5)
    {
        System.Console.WriteLine("c = " + c + "\n");
    }
}

class Program
{
    static void Main()
    {
        A a = new A('a'); //Utwórz obiekt
    }
}
```

Powyższy kod przed wywołaniem konstruktora `A('a')` wywoła konstruktor `A(5)`. Na konsoli wyświetli się zatem:

```
i = 5  
c = a
```

Teraz o destruktorze. Otóż jest on wywoływany przy niszczeniu obiektu. W przeciwieństwie do konstruktora — destruktor może być tylko jeden.

Deklaracja destrukторa wygląda następująco:

```
~Car() //destruktor  
{  
    //jakieś instrukcje sprzątające  
}
```

Destruktory występują tylko w klasach, nie można ich dodać do struktur. Nie mogą być one przeciążane ani dziedziczone. Destruktor nie może być także wywołany, jest uruchamiany automatycznie.

## 4.3. Dziedziczenie

Dziedziczenie jest to przejmowanie pól, metod i innych składowych klasy bazowej oraz dodanie własnych składowych.

Dobrym przykładem może być zwierzę, które będzie prezentowało klasę bazową `Animal`. Jednym z rodzajów zwierząt jest pies, czyli klasa `Dog` dziedziczy po klasie `Animal`.

Taka relacja miałaby następujący zapis:

```
public class Dog : Animal  
{  
    //Składowe klasy Animal są dziedziczone  
    //Tutaj będą nowe składowe klasy Dog  
}
```

Stwórzmy klasę `Animal` z metodą `Eat` (jeść):

```
public class Animal  
{  
    public void Eat(string food)  
    {  
        Console.WriteLine("mniam.");  
    }  
}
```

Teraz stwórzmy klasę `Dog` dziedziczącą po klasie `Animal` z metodą `Bark` (szczekać) oraz polem `Breed` (rasa):

```
public class Dog : Animal  
{  
    public string Breed; //rasa psa  
    public void Bark()  
    {  
        Console.WriteLine("hau! hau!");  
    }  
}
```

Każde zwierzę, aby żyć, musi jeść, dlatego metoda Eat jest w klasie Animal. Jednak nie każde zwierzę szczenka, dlatego metoda Bark musi być w klasie Dog.

Teraz utworzymy obiekt klasy Dog, określmy rasę psa i wywołajmy metodę Eat i Bark:

```
Dog dog1 = new Dog();
dog1.Breed = "Pitbull";
dog1.Eat("kość"); //To jest metoda z klasy bazowej (Animal)!
```

```
dog1.Bark();
```

Po uruchomieniu programu na konsoli powinno się pojawić:

```
mniam.
hau! hau!
```

Kod całego przykładu:

```
using System;

public class Animal
{
    public void Eat(string food)
    {
        Console.WriteLine("mniam.");
    }
}

public class Dog : Animal
{
    public string Breed;
    public void Bark()
    {
        Console.WriteLine("hau! hau!");
    }
}

class Program
{
    static void Main()
    {
        Dog dog1 = new Dog();
        dog1.Breed = "Pitbull";
        dog1.Eat("kość"); //To jest metoda z klasy bazowej (Animal)!
```

```
        dog1.Bark();
    }
}
```

## Klasy zagnieżdżone

Klasy mogą dziedziczyć to, co zostało wcześniej opisane, ale mogą też być zagnieżdżone. Jak należy rozumieć zagnieżdżanie i czym ono się różni od dziedziczenia? Zagnieżdżanie oznacza, że jedna klasa jest zawarta w drugiej. Na przykład: Silnik jest częścią samochodu.

Oto prosty przykład:

```
using System;

public class Car
{
    public void TurnOnLights()
    {
        Console.WriteLine("Światła włączone.");
    }
    public class Engine //klasa zagnieżdzona
    {
        public void StartEngine()
        {
            Console.WriteLine("Silnik uruchomiony.");
        }
    }
}

class Program
{
    static void Main()
    {
        Car.Engine engine1 = new Car.Engine(); //obiekt silnika
        Car car1 = new Car(); //obiekt samochodu
        engine1.StartEngine();
        car1.TurnOnLights();

    }
}
```

W powyższym przykładzie mamy klasę samochodu o nazwie `Car` i klasę silnika o nazwie `Engine`. W klasie `Car` jest metoda włączająca światła (`TurnOnLights`), a w klasie `Engine` jest metoda włączająca silnik (`StartEngine`). Obiekt klasy zagnieżdzonej tworzymy, podając nazwę klasy bazowej i po kropce nazwę klasy zagnieżdzonej. Po uruchomieniu powyższego przykładu na ekranie powinieneś zobaczyć tekst:

```
Silnik uruchomiony.  
Światła włączone.
```

## 4.4. Modyfikatory dostępu

Wszystkie typy i ich składowe mają jakiś poziom dostępności określający, kiedy mogą być używane.

Modyfikatory dostępu są następujące:

- ◆ `public` — typ (lub składowa) jest dostępny przez każdy kod w jednostce lub innych jednostkach.
- ◆ `private` — typ (lub składowa) jest dostępny tylko dla kodu tej samej klasy lub struktury.

- ◆ **protected** — typ (lub składowa) jest dostępny dla kodu tej samej klasy lub struktury oraz dla klasy, która dziedziczy po niej.
- ◆ **internal** — typ (lub składowa) jest dostępny przez każdy kod w jednostce, ale nie w innych jednostkach.
- ◆ **protected internal** — typ (lub składowa) jest dostępny przez każdy kod w jednostce, dla klas potomnych, ale nie w innych jednostkach.



Jeżeli nie podamy modyfikatora dostępu — uznawane jest, jakby był tam modyfikator **private**.

## Słowo kluczowe **readonly**

Jest to modyfikator, którego można użyć dla pól. Gdy deklaracja pola zawiera to słowo kluczowe, wówczas przypisać wartość temu polu można tylko podczas deklaracji lub w konstruktorze:

```
public class Age
{
    readonly int _year;
    Age(int year)
    {
        _year = year; //OK
    }
    void ChangeYear()
    {
        //_year = 1988; //Błąd kompilacji
    }
}
```

## Pola powinny być prywatne

Jest taka zasada w programowaniu obiektowym, że pola klasy powinny być prywatne, a do przeprowadzenia ich zmian należy utworzyć odpowiednie metody. Takie działanie nazywamy hermetyzacją lub enkapsulacją.

Przykład:

```
using System;

public class Point
{
    private int X;
    private int Y;

    public void SetXY(int _X, int _Y)
    {
        X = _X;
        Y = _Y;
    }
}
```

```
public void GetXY()
{
    Console.WriteLine("X = {0}, Y = {1}", X, Y);
}

class Program
{
    static void Main()
    {
        Point point1 = new Point();
        point1.SetXY(2, 4);
        point1.GetXY();

    }
}
```

Mamy klasę reprezentującą punkt. Ma ona pola `X` i `Y`, które są prywatne, dlatego stworzyliśmy metodę `SetXY`, która obiektowi klasy `Point` ustawia podane wartości współrzędnych.

## 4.5. Wczesne i późne wiązanie

Wczesne wiązanie występuje w sytuacji, gdy kompilator wie, jakiego rodzaju jest obiekt oraz jakie metody i właściwości zawiera. Gdy zadeklarujesz obiekt i postawisz po nim kropkę, wówczas technologia *Intellisense* wyświetli Ci metody i właściwości tego obiektu.

Przykładem dla wczesnego wiązania może być:

```
ComboBox cboItems;
ListBox lstItems;
```

Późne wiązanie polega na tym, że najpierw deklarujesz obiekt, a dopiero później pobierasz typ obiektu lub metody, jakie on zawiera. Wszystko będzie wiadome w czasie wykonywania.

Przykład:

```
Object objItems;
objItems = CreateObject("nazwa biblioteki DLL lub jednostki assembly");
```

W powyższym przykładzie typ `objItems` nie jest znany w czasie komplikacji.

## Wczesne wiązanie vs późne wiązanie

- ♦ Aplikacja będzie działać szybciej we wczesnym wiązaniu, dopóki nie używamy opakowywania zmiennych.
- ♦ Łatwiej pisać kod we wczesnym wiązaniu, gdyż pomaga nam *Intellisense*.

- ◆ Dzięki użyciu wczesnego wiązania popelnia się mniej błędów.
- ◆ Późne wiązanie wspiera różne rodzaje wersji, gdyż w czasie wykonywania wszystko jest wiadome.
- ◆ Łatwiej rozszerzać i rozwijać kod, który używa późnego wiązania.

Zarówno wczesne wiązanie, jak i późne wiązanie mają swoje wady i zalety, dlatego to, co wybierzesz, zależy tylko od Ciebie, programisto.

## Opakowywanie zmiennych

Jest to konwersja typów prostych na typ object. Gdy typ prosty jest opakowywany i staje się obiektem, wówczas jego wartość jest przechowywana na zarządzanej stercie.

W poniższym przykładzie zmienna i jest opakowywana i przypisywana do obiektu o:

```
int i = 123;  
object o = i; //opakowywanie
```

Rozpakowywanie działa odwrotnie:

```
o = 123;  
i = (int) o; //rozpakowywanie
```

Po co to wszystko? Chodzi o to, aby sprawić, żeby każdy typ był obiektem w języku C#.

## 4.6. Przeciążanie metod

Metody przeciążane to takie, które mają taką samą nazwę, ale różnią się typem lub ilością parametrów. Wprowadzimy teraz pojęcie sygnatury. Sygnatura to nazwa metody i lista jej wszystkich parametrów. Metody przeciążane muszą zatem mieć różne sygnatury. Popatrz na poniższą klasę z metodami przeciążanymi:

```
using System;  
  
public class A  
{  
    public void Method1(int A)  
    {  
        Console.WriteLine("A = {0}", A);  
    }  
    public void Method1(int A, int B)  
    {  
        Console.WriteLine("A = {0}, B = {1}", A, B);  
    }  
    public void Method1(double A)  
    {  
        Console.WriteLine("A = {0}", A);  
    }  
}
```

```
class Program
{
    static void Main()
    {
        A a = new A();
        a.Method1(2);      //wywołanie metody Method1(int A)
        a.Method1(3, 4);  //wywołanie metody Method1(int A, int B)
        a.Method1(5.2);   //wywołanie metody Method1(double A)
    }
}
```

Wyjście programu:

```
A = 2
A = 3, B = 4
A = 5.2
```

Przeciążać możemy też konstruktory, robi się to analogicznie.

Poniżej lista przykładowych metod z ich sygnaturami w komentarzach obok:

```
void F():           //F()
void F(int x);    //F(int)
void F(ref int x); //F(ref int)
void F(int x, int y); //F(int, int)
int F(string s);  //F(string)
int F(int x);     //F(int) błęd!
void F(string[] a); //F(string[])
void F(params string[] a); //F(string[]) błęd!
```

## 4.7. Przeciążanie operatorów

Język C# pozwala typom zdefiniowanym przez programistę przeciążyć operatory, definiując statyczne funkcje za pomocą słowa kluczowego operator. Poniżej znajduje się przykład prostej klasy reprezentującej ułamki. W tej klasie będą przeciążone: operator dodawania, operator mnożenia oraz operator konwersji ułamka zwykłego na typ double.

```
using System;

class Fraction
{
    int num, den; //num — licznik, den — mianownik
    public Fraction(int num, int den)
    {
        this.num = num;
        this.den = den;
    }

    //przeciążony operator +
    public static Fraction operator +(Fraction a, Fraction b)
    {
        return new Fraction(a.num * b.den + b.num * a.den,
                            a.den * b.den);
    }
}
```

```

//przeciążony operator *
public static Fraction operator *(Fraction a, Fraction b)
{
    return new Fraction(a.num * b.num, a.den * b.den);
}

//zdefiniowana przez programistę konwersja
public static implicit operator double(Fraction f)
{
    return (double)f.num / f.den;
}

class Program
{
    static void Main()
    {
        Fraction a = new Fraction(1, 2);
        Fraction b = new Fraction(3, 7);
        Fraction c = new Fraction(2, 3);
        Console.WriteLine((double)a);
        Console.WriteLine((double)(a * b + c));
    }
}

```

Wyjście programu:

```

0.5
0.880952380952381

```

Nie wszystkie operatory mogą być przeciążone, niektóre mają ograniczenia, co zostało opisane w tabeli 4.1.

**Tabela 4.1.** Lista operatorów i informacja, czy mogą być one przeciążone

Operatory	Przeciążanie
+, -, !, ~, ++, --, true, false	Te jednoargumentowe operatory mogą być przeciążone.
, -, *, /, %, &,  , ^, <<, >>	Te operatory dwuargumentowe mogą być przeciążone.
==, !=, <, >, <=, >=	Te operatory porównania mogą być przeciążone, ale muszą być przeciążane parami, czyli jeżeli przeciążymy ==, to musimy też przeciążyć !=.
&&,	Te operatory nie mogą być przeciążone, ale są one wykonywane przez & i  , które mogą być przeciążone.
[]	Ten operator nie może być przeciążony.
(T)x	Ten operator konwersji nie może być przeciążony, ale możesz zdefiniować nowe operatory konwersji.
+=, -=, *=, /=, %=, &=,  =, ^=, <<=, >>=	Te operatory nie mogą być przeciążone, ale na przykład operator += jest wykonywany przez +, który może być przeciążony.
=, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof	Te operatory nie mogą być przeciążone.

## Słowa kluczowe **implicit** i **explicit**

Slowo kluczowe **implicit** pojawiło się w przykładzie o przeciążaniu operatorów. Używa się go przy przeciążaniu operatora konwersji z typu zdefiniowanego przez programistę na inny typ, jeżeli jest pewne, że żadne dane nie zostaną utracone przy konwersji.

Poniżej znajduje się przykładowa klasa reprezentująca liczbę i są w niej przeciążone dwa operatory konwersji.

```
using System;

class Digit
{
    public Digit(double d) { val = d; }
    public double val;

    //zdefiniowana przez programistę konwersja z Digit na double
    public static implicit operator double(Digit d)
    {
        return d.val;
    }
    //zdefiniowana przez programistę konwersja z double na Digit
    public static implicit operator Digit(double d)
    {
        return new Digit(d);
    }
}

class Program
{
    static void Main()
    {
        Digit dig = new Digit(7);
        //wywołanie operatora konwersji z Digit na double
        double num = dig;
        //wywołanie operatora konwersji z double na Digit
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
    }
}
```

Natomiast slowo kluczowe **explicit** jest używane przy konwersji z jednego typu zdefiniowanego przez programistę na inny typ zdefiniowany przez programistę. Konwersja taka musi być wywoływaną razem z rzutowaniem.

Oto przykład przeciążenia operatora konwersji miary w jednostkach Fahrenheit na jednostki Celsius:

```
public static explicit operator Celsius(Fahrenheit fahr)
{
    return new Celsius((5.0f / 9.0f) * (fahr.degrees - 32));
}
```

Przykładowe wywołanie:

```
Fahrenheit fahr = new Fahrenheit(100.0f);
Console.WriteLine("{0} Fahrenheit", fahr.Degrees);
Celsius c = (Celsius)fahr;
```

## 4.8. Statyczne metody i pola

Składowe statyczne to takie, które istnieją nawet wtedy, gdy nie istnieje żaden obiekt danej klasy. Wcześniej spotkaliśmy się ze statyczną metodą Main. Jest to główna metoda programu, która wykonuje się przed tworzeniem jakichkolwiek obiektów, dlatego musi być ona statyczna.

Metody statyczne deklarujemy według szablonu:

```
modyfikator_dostępu static typ_zwracany nazwa_metody(lista_parametrów)
{
    wewnętrzne metody
}
```

Natomiast pola statyczne:

```
modyfikator_dostępu static typ_pola nazwa_pola;
```

Zobacz poniższy przykład:

```
using System;

class A
{
    public static int val = 0; //statyczne pole
    public static void f()      //statyczna metoda
    {
        Console.WriteLine("Statyczna metoda f() z klasy A");
    }
}

class Program
{
    static void Main()
    {
        A.val = 64;
        Console.WriteLine("A.val = {0}", A.val);
        A.f();
    }
}
```

Widac w tym przykładzie, że do pól i metod statycznych odwołujemy się, podając nazwę klasy i po kropce nazwę pola lub metody. Nie tworzymy żadnego obiektu.

## 4.9. Klasy abstrakcyjne i zapieczętowane

Słowo kluczowe `abstract` pozwala tworzyć klasy i składowe, które są niekompletne i muszą być zaimplementowane w klasie pochodnej.

Klasa może być zadeklarowana jako abstrakcyjna poprzez umieszczenie słowa kluczowego `abstract` przed definicją. Na przykład:

```
public abstract class A
{
    //Tutaj są składowe klasy
}
```

Nie można tworzyć instancji klasy abstrakcyjnej. Celem klasy abstrakcyjnej jest dostarczenie definicji klasy bazowej, którą klasy potomne mogą dzielić między sobą. Na przykład biblioteka może definiować klasę abstrakcyjną używaną jako parametr różnych funkcji i wymagać, aby programista korzystający z biblioteki stworzył własną implementację klasy poprzez utworzenie klasy potomnej.

Klasy abstrakcyjne mogą definiować metody abstrakcyjne. Robi się to poprzez dodanie słowa kluczowego `abstract` przed typem zwracanym przez metodę. Na przykład:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Metody abstrakcyjne nie mają implementacji, dlatego definicja takiej metody kończy się średnikiem, zamiast blokiem instrukcji. Klasa potomna klasy abstrakcyjnej muszą implementować wszystkie metody abstrakcyjne. Gdy klasa abstrakcyjna dziedziczy metodę wirtualną z klasy bazowej, klasa abstrakcyjna musi przesłonić metodę wirtualną metodą abstrakcyjną. Na przykład:

```
public class D
{
    public virtual void DoWork(int i)
    {
        //oryginalna implementacja
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        //nowa implementacja
    }
}
```

Jeżeli metoda wirtualna jest zadeklarowana jako abstrakcyjna, jest nadal wirtualna dla każdej klasy dziedziczącej z klasy abstrakcyjnej. Klasa dziedzicząca metodę abstrakcyjną nie ma dostępu do oryginalnej implementacji metody — w poprzednim przykładzie metoda `Dowork` klasy `F` nie może wywołać metody `Dowork` klasy `D`. Podsumowując, klasa abstrakcyjna może wymusić na klasach potomnych dostarczenie nowej implementacji metod wirtualnych.

Słowo kluczowe `sealed` pozwala zabronić dziedziczenia klasy lub składowych, które zostały wcześniej oznaczone jako `virtual`.

Klasy mogą być zadeklarowane jako zapieczętowane poprzez umieszczenie słowa kluczowego `sealed` przed definicją klasy. Na przykład:

```
public sealed class D
{
    //Tutaj składowe klasy
}
```

Klasa zapieczętowana nie może być używana jako klasa bazowa. Z tego powodu nie może być ona także klasą abstrakcyjną, gdyż zabrania tworzenia klas potomnych. Ponieważ klasa taka nie może być klasą bazową, to niektóre optymalizacje czasu uruchomienia (*run-time*) mogą spowodować nieco szybsze wywoływanie składowych klasy zapieczętowanej.

Składowa klasy potomnej, która przesyła wirtualną składową klasy bazowej, może zadeklarować tę składową jako zapieczętowaną. Neguje to aspekt wirtualny dla kolejnych klas potomnych. Wykonuje się to poprzez umieszczenie słowa kluczowego `sealed` przed słowem `override` w deklaracji składowej.

Przykład:

```
public class D : C
{
    public sealed override void Dowork() { }
}
```

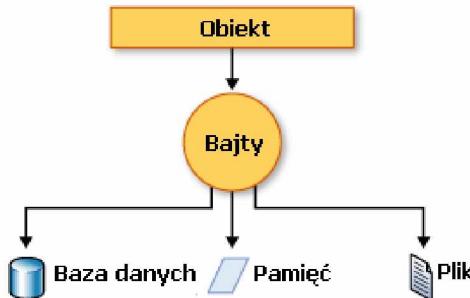
## 4.10. Serializacja

Serializacja to proces konwertowania obiektu na strumień bajtów w celu zachowania obiektu w bazie danych, w pamięci lub pliku. Głównym przeznaczeniem jest zapisanie stanu obiektu w celu późniejszego odtworzenia go w razie potrzeby. Proces odwrotny do serializacji nazywamy deserializacją.

Na rysunku 4.1. pokazany jest przebieg procesu serializacji.

Obiekt jest serializowany do strumienia, który zachowuje nie tylko dane, ale także wiele innych informacji o obiekcie, np. typ obiektu lub nazwa jednostki (ang. *assembly*).

Rysunek 4.1.  
Proces serializacji obiektu



## Użyteczność serializacji

Serializacja pozwala programistom zapisać stan obiektu i odtworzyć go, gdy zajdzie taka potrzeba. Obiekt po serializacji można przesłać przez sieć lub między różnymi aplikacjami.

## Zapis obiektu do pliku XML

Poniższy przykład definiuje klasę o nazwie Book, tworzy instancję tej klasy i używa serializacji XML w celu zapisania instancji obiektu do pliku XML.

```
public class Book
{
    public String title;
}

public void WriteXML()
{
    Book overview = new Book();
    overview.title = "Serialization Overview";
    System.Xml.Serialization.XmlSerializer writer =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));

    System.IO.StreamWriter file = new System.IO.StreamWriter(
        @"c:\temp\SerializationOverview.xml");
    writer.Serialize(file, overview);
    file.Close();
}
```

## Odczyt obiektu z pliku XML

Natomiast poniższy przykład odczytuje obiekt zapisany wcześniej do pliku XML za pomocą klasy XmlSerializer.

```
public class Book
{
    public String title;
}
```

```
public void ReadXML()
{
    System.Xml.Serialization.XmlSerializer reader =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));
    System.IO.StreamReader file = new System.IO.StreamReader(
        @"c:\temp\SerializationOverview.xml");
    Book overview = new Book();
    overview = (Book)reader.Deserialize(file);

    Console.WriteLine(overview.title);

}
```

## 4.11. Przestrzenie nazw

Przestrzeń nazw służy do łączenia różnych typów w logiczne grupy. Mogłeś o tym nie wiedzieć, ale w poprzednich rozdziałach tej książki korzystaliśmy już z przestrzeni nazw, między innymi z głównej przestrzeni nazw, jaką jest `System`.

Jeżeli chcesz odwołać się do typu w określonej przestrzeni nazw, wpisz przestrzeń nazw, następnie po kropce podrzędne przestrzenie nazw, a na końcu nazwę typu.

Wywołanie metody `WriteLine` z klasy `Console`, która znajduje się w przestrzeni nazw `System`, wygląda następująco:

```
class Program
{
    static void Main()
    {
        System.Console.WriteLine("Metoda WriteLine z klasy Console z przestrzeni
        nazw System");
    }
}
```

Aby skrócić zapis, można użyć słowa kluczowego `using`. Wtedy pomijamy przestrzeń nazw i od razu odwołujemy się do typu w tej przestrzeni:

```
using System; //Użyj przestrzeni nazw System

class Program
{
    static void Main()
    {
        Console.WriteLine("Metoda WriteLine z klasy Console z przestrzeni nazw System");
    }
}
```

Mozna także tworzyć własne przestrzenie nazw. Wykonuje się to według schematu:

```
namespace nazwa_przestrzeni
{
    //typy zawarte w przestrzeni nazw
}
```

Oto przykładowa przestrzeń nazw `Animals` z klasą `Dog` oraz `Wolf`:

```
using System;

namespace Animals //przestrzeń nazw Zwierzęta
{
    class Dog //klasa Pies
    {
        public void Bark() //metoda Szczekać
        {
            Console.WriteLine("Pies szczeka.");
        }
    }
    class Wolf //klasa Wilk
    {
        public void Howl() //metoda Wyć
        {
            Console.WriteLine("Wilk wyje.");
        }
    }
}

class Program
{
    static void Main()
    {
        Animals.Dog dog1 = new Animals.Dog();
        Animals.Wolf wolf1 = new Animals.Wolf();
        dog1.Bark(); //Pies szczeka
        wolf1.Howl(); //Wilk wyje
    }
}
```

Jak napisane jest wcześniej, możemy użyć słowa kluczowego `using` i wtedy pomijamy przestrzeń nazw w odwoływaniu się do typów w tej przestrzeni:

```
using System;
using Animals; //Użyj przestrzeni nazw Animals

namespace Animals //przestrzeń nazw Zwierzęta
{
    class Dog //klasa Pies
    {
        public void Bark() //metoda Szczekać
        {
            Console.WriteLine("Pies szczeka.");
        }
    }
    class Wolf //klasa Wilk
    {
        public void Howl() //metoda Wyć
        {
            Console.WriteLine("Wilk wyje.");
        }
    }
}
```

```
}
```

```
class Program
{
    static void Main()
    {
        Dog dog1 = new Dog(); //Nie musimy podawać przestrzeni nazw
        Wolf wolf1 = new Wolf(); //przy odwoływaniu się do jej typów
        dog1.Bark(); //Pies szczeka
        wolf1.Howl(); //Wilk wyje
    }
}
```

Jeżeli daną przestrzeń nazw chcemy nazwać tak, jak nam jest wygodnie, możemy stworzyć alias. Wykonuje się to według schematu:

```
using Sys = System; //utworzenie skrótu Sys do nazwy System
```

```
class Program
{
    static void Main()
    {
        Sys.Console.WriteLine("Witaj."); //Możemy napisać Sys zamiast System
    }
}
```

Wtedy możemy używać zarówno aliasu (Sys), jak i pełnej nazwy przestrzeni (System).

## 4.12. Właściwości

Właściwość to składowa, która dostarcza elastyczny mechanizm do odczytu, zapisu lub obliczenia wartości prywatnego pola. Z właściwościami wiąże się też pojęcie akcesorów, czyli specjalnych metod. Istnieją dwa akcesory: get i set. Akcesor get służy do kontroli zwracanej wartości. Natomiast akcesor set służy do kontroli przypisywanej wartości. Akcesor set zawiera także specjalne słowo o nazwie value, które definiuje wartość przypisywaną.

W poniższym przykładzie klasa `TimePeriod` przechowuje odcinek czasu wyrażony w sekundach, ale właściwość o nazwie `Hours` pozwala określić czas w godzinach. Akcesory właściwości `Hours` wykonują konwersję pomiędzy godzinami a sekundami.

```
class TimePeriod
{
    private double seconds; //prywatne pole seconds

    public double Hours //publiczna właściwość Hours
    {
        get { return seconds / 3600; }
        set { seconds = value * 3600; }
    }
}
```

```
class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();

        //Przypisanie do właściwości Hours powoduje wywołanie akcesora set
        t.Hours = 24;

        //Obliczenie właściwości Hours powoduje wywołanie akcesora get
        System.Console.WriteLine("Czas w godzinach: " + t.Hours);
    }
}
//Wyjście programu: Czas w godzinach: 24
```

## 4.13. Interfejsy

Interfejs opisuje grupę funkcjonalności, które mogą należeć do danej klasy lub struktury. Definiuje się go za pomocą słowa kluczowego `interface`, tak jak w przykładzie poniżej:

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

Interfejs może zawierać metody, właściwości, zdarzenia i indeksery. Nie może natomiast zawierać pól, operatorów, wartości stałych, konstruktorów, destruktatorów ani typów. Nie może także zawierać stałych składowych. Składowe interfejsów są automatycznie publiczne i nie mogą zawierać modyfikatorów dostępu.

Gdy klasa lub struktura implementuje interfejs, to implementuje wszystkie zdefiniowane w nim składowe.

Przykładowa implementacja wcześniej zdefiniowanego interfejsu `IEquatable<T>`:

```
public class Car : IEquatable<Car>
{
    public string Model { get; set; }
    public int Year { get; set; }

    //implementacja interfejsu IEquatable<T>
    public bool Equals(Car car)
    {
        if (this.Model == car.Model &&
            this.Year == car.Year)
        {
            return true;
        }
        else
            return false;
    }
}
```

Dzięki powyższej implementacji ktoś używający naszego kodu może za pomocą metody `Equals` porównać, czy obiekty klasy `Car` są takie same:

```
Car car1 = new Car();
Car car2 = new Car();

car1.Model = "Subaru Impreza";
car1.Year = 2005;

car2.Model = "Mitsubishi Lancer";
car2.Year = 1997;

if (car1.Equals(car2))
    Console.WriteLine("car1 == car2");
else
    Console.WriteLine("car1 != car2");
```

Oczywiście powyższe obiekty nie są takie same, czyli na konsoli zostanie wypisane:

```
car1 != car2
```

Podsumowując:

- ◆ Interfejs jest jak abstrakcyjna klasa bazowa: każdy nieabstrakcyjny typ implementujący interfejs musi zaimplementować wszystkie jego składowe.
- ◆ Interfejsy mogą zawierać zdarzenia, indeksery, metody i właściwości.
- ◆ Interfejsy nie zawierają implementacji metod.
- ◆ Klasa lub struktura może implementować więcej niż jeden interfejs.
- ◆ Interfejs może dziedziczyć z innych interfejsów.

## Płytką i głęboką kopią obiektu

Gdy chcemy skopiować wartość jednej zmiennej typu prostego, używamy operatora przypisania:

```
var1 = var2;
```

Jeżeli jednak zmienne są typu referencyjnego, to użycie operatora przypisania spowoduje skopiowanie referencji. Wtedy obie zmienne będą wskazywać na tę samą instancję obiektu w pamięci:

```
using System;

public class A
{
    public int a;
}

class Program
{
    static void Main()
    {
```

```
A a1 = new A();
a1.a = 16;
A a2 = new A();
a2 = a1; //skopiowanie referencji
Console.WriteLine("a2.a = {0}", a2.a); //Wypisze a2.a = 16
a1.a = 32;
Console.WriteLine("a2.a = {0}", a2.a); //Wypisze a2.a = 32
}
}
```

Jeżeli chcemy sprawdzić, czy referencje odwołują się do tego samego obiektu, używamy statycznej metody `Object.ReferenceEquals()`, która zwraca `true`, jeżeli tak jest, lub `false` w przeciwnym wypadku. Oczywiście w powyższym przykładzie metoda ta zwróciłaby wartość `true`.

## Płytką kopią

Jeżeli klasa obiektu, który chcemy skopiować, nie ma pól referencyjnych, możemy użyć płytkiej kopii. Wystarczy, żeby klasa dziedziczyła po interfejsie `ICloneable` i implementowała metodę `Clone()`, tak jak w przykładzie poniżej:

```
using System;

public class A : ICloneable
{
    public int a;
    public Object Clone()
    {
        return MemberwiseClone();
    }
}

class Program
{
    static void Main()
    {
        A a1 = new A();
        A a2 = new A();
        a1.a = 8;
        a2 = (A)a1.Clone();
        Console.WriteLine("a2.a = {0}", a2.a); //Wypisze a2.a = 8
        if (Object.ReferenceEquals(a1, a2))
            Console.WriteLine("Referencje a1 i a2 odwołują się do tego samego obiektu");
        else
            Console.WriteLine("Referencje a1 i a2 nie odwołują się do tego samego obiektu"); //To się wykona
    }
}
```

## Głęboka kopią

Głębokiej kopii używamy, gdy klasa ma pola referencyjne. Tworzymy wtedy specjalną metodę kopującą, tak jak w poniższym przykładzie:

```
using System;

class A
{
    public int a;
    public B objB;
    public A()
    {
        objB = new B();
    }
    public A DeepCopy()
    {
        A tempA = new A();
        tempA.a = this.a;
        tempA.objB.word = this.objB.word;
        return tempA;
    }
}

class B
{
    public string word;
}

class Program
{
    static void Main()
    {
        A a1 = new A();
        A a2 = new A();

        a1.a = 7;
        a1.objB.word = "przykład";

        a2 = a1.DeepCopy(); //głęboka kopia

        Console.WriteLine("a2.a = " + a2.a); //Wypisze a2.a = 7
        Console.WriteLine("a2.objB.word = " + a2.objB.word); //Wypisze a2.objB.word =
                                                               przykład

        if (Object.ReferenceEquals(a1.objB, a2.objB))
            Console.WriteLine("Referencje a1.objB i a2.objB odwołują się do tego
                           →samego obiektu");
        else
            Console.WriteLine("Referencje a1.objB i a2.objB nie odwołują się do
                           →tego samego obiektu"); //To się wykona
    }
}
```

## 4.14. Indeksery

Indeksery pozwalają instancjom klasy lub struktury być indeksowanym tak jak tablica. Są one podobne do właściwości — z tą jednak różnicą, że ich akcesory pobierają parametry.

Załóżmy, że potrzebujesz klasy TempRecord, która reprezentuje temperaturę w jednostkach Fahrenheit. Temperatura jest zapisywana 10 razy na dobę. Klasa zawiera tablicę temps liczb zmiennoprzecinkowych typu float. Implementując indekser w tej klasie, programista może mieć dostęp do temperatur przez float temp = tr[4] zamiast przez float temp = tr.temps[4].

Notacja ta nie tylko upraszcza składnię, ale powoduje także, że odwoływanie się jest bardziej intuicyjne.

Indekser deklarujemy według schematu:

```
public int this[int index] //deklaracja indeksera
{
    //akcesory get i set
}
```

Poniższy przykład pokazuje, jak zadeklarować prywatne pole tablicy temps oraz indekser. Indekser pozwala na bezpośredni dostęp do instancji poprzez zapis tempRecord[i]. Alternatywnym rozwiązaniem bez użycia indeksera jest zadeklarowanie tablicy jako publicznej.

Zauważ, że gdy następuje dostęp do indeksera, to wywoływany jest akcesor get. Jeżeli akcesor get nie będzie istniał, wówczas wystąpi błąd.

```
class TempRecord
{
    //tablica z wartościami temperatur
    private float[] temps = new float[10] { 56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
                                             61.3F, 65.9F, 62.1F, 59.2F, 57.5F };

    public int Length
    {
        get { return temps.Length; }
    }
    //deklaracja indeksera
    //Gdy indeks jest poza przedziałem, rzucony zostanie wyjątek
    public float this[int index]
    {
        get
        {
            return temps[index];
        }

        set
        {
            temps[index] = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        TempRecord tempRecord = new TempRecord();
```

```
//użycie akcesora set  
tempRecord[3] = 58.3F;  
tempRecord[5] = 60.1F;  
  
//użycie akcesora get  
for (int i = 0; i < 10; i++)  
{  
    System.Console.WriteLine("Element #{0} = {1}", i, tempRecord[i]);  
}  
}
```

Wyjście programu:

```
Element #0 = 56.2  
Element #1 = 56.7  
Element #2 = 56.5  
Element #3 = 58.3  
Element #4 = 58.8  
Element #5 = 60.1  
Element #6 = 65.9  
Element #7 = 62.1  
Element #8 = 59.2  
Element #9 = 57.5
```

Język C# nie ogranicza typu indeksu do liczby całkowitej. Może to być też napis, na przykład przy implementacji indeksera wyszukującego w kolekcji napisu.

Podsumowując:

- ◆ Indeksery pozwalają obiektom być indeksowanym podobnie do tablic.
- ◆ Akcesor `get` zwraca wartość. Akcesor `set` ją ustawia.
- ◆ Słowo kluczowe `this` jest używane do definiowania indekserów.
- ◆ Słowo kluczowe `value` jest używane do przypisywania wartości przez `set`.
- ◆ Indeksery nie muszą być indeksowane przez wartość typu `int`.
- ◆ Indeksery mogą być przeciążane.
- ◆ Mogą mieć więcej niż jeden formalny parametr, np. przy dostępie do tablicy dwuwymiarowej.

## 4.15. Polimorfizm

Polimorfizm jest często określany jako trzeci (po hermetyzacji i dziedziczeniu) filar programowania obiektowego. Polimorfizm to greckie słowo (oznacza „wielokształtny”). Ma on dwa odrębne aspekty:

- ◆ Podczas wykonywania programu obiekty klasy pochodnej mogą być traktowane jako obiekty klasy bazowej. W takim przypadku zadeklarowany typ obiektu nie jest identyczny z typem w *runtime* (podczas wykonywania programu).

- ◆ Klasy bazowe mogą definiować i implementować metody wirtualne, a klasy pochodne mogą je przesłaniać, co oznacza dostarczenie własnej definicji i implementacji. Podczas wykonywania programu kod klienta wywołuje metodę, a środowisko uruchomieniowe sprawdza typ obiektu czasu wykonania i wywołuje metodę, która jest przesłonięta. Oznacza to, że gdy ze swojego kodu źródłowego wywołujesz metodę klasy bazowej, powoduje to wywołanie metody w wersji klasy pochodnej.

Załóżmy, że potrzebujesz aplikacji rysującej różne kształty na ekranie. W czasie komplikacji nie wiesz, jakie kształty użytkownik chciałby narysować. Możesz to rozwiązać, korzystając z polimorfizmu w dwóch podstawowych krokach:

- ◆ Stwórz hierarchię klas, w której klasa każdego kształtu pochodzi z klasy bazowej.
- ◆ Użyj metod wirtualnych do wywołania określonej metody klasy pochodnej poprzez pojedyncze wywołanie metody klasy bazowej.

Najpierw stwórz klasę bazową o nazwie `Shape` (czyli kształt) oraz klasy pochodne `Rectangle`, `Circle` i `Triangle` (czyli prostokąt, koło i trójkąt). Daj klasie `Shape` metodę wirtualną `Draw` (czyli rysuj) i przesłoń ją w każdej klasie pochodnej. Stwórz obiekt `List<Shape>`, czyli listę (o typach generycznych będzie więcej w dalszej części książki), i dodaj do niej prostokąt (`Rectangle`), trójkąt (`Triangle`) i koło (`Circle`). W celu odświeżenia powierzchni do rysowania użyj pętli `foreach`, iterując po jej elementach i wywołując metodę `Draw` na każdym obiekcie typu `Shape`.

Oto przykładowy kod:

```
using System;

public class Shape
{
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    //metoda wirtualna
    public virtual void Draw()
    {
        Console.WriteLine("Podstawowe zadanie rysowania");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        //kod rysujący koło
        Console.WriteLine("Rysuję koło");
        base.Draw();
    }
}
class Rectangle : Shape
{
```

```
public override void Draw()
{
    //kod rysujący prostokąt
    Console.WriteLine("Rysuję prostokąt");
    base.Draw();
}
}
class Triangle : Shape
{
    public override void Draw()
    {
        //kod rysujący trójkąt
        Console.WriteLine("Rysuję trójkąt");
        base.Draw();
    }
}

class Program
{
    static void Main()
    {
        //Prostokąt, trójkąt i koło mogą być użyte wszędzie, gdzie wymagany
        //jest typ Shape. Nie jest tu wymagane żadne rzutowanie, ponieważ
        //istnieje domyślna konwersja z klasy pochodnej do bazowej
        System.Collections.Generic.List<Shape> shapes = new
        →System.Collections.Generic.List<Shape>();
        shapes.Add(new Rectangle());
        shapes.Add(new Triangle());
        shapes.Add(new Circle());

        //Metoda wirtualna Draw jest wywoływana na klasach pochodnych,
        //nie na klasie bazowej
        foreach (Shape s in shapes)
        {
            s.Draw();
        }
    }
}
```

### Wyjście programu:

Rysuję prostokąt  
Podstawowe zadanie rysowania  
Rysuję trójkąt  
Podstawowe zadanie rysowania  
Rysuję koło  
Podstawowe zadanie rysowania



W języku C# każdy typ jest polimorficzny, gdyż wszystkie typy, włączając w to typy zdefiniowane przez programistę, dziedziczą z typu Object.

## Składowe wirtualne

Gdy klasa pochodna dziedziczy z klasy pochodnej, zyskuje wszystkie metody, pola, właściwości i zdarzenia klasy bazowej. Projektujący klasę pochodną może zdecydować, czy:

- ◆ przesłonić wirtualne składowe klasy bazowej,
- ◆ dziedziczyć metody klasy bazowej bez ich przesłaniania,
- ◆ zdefiniować nowe niewirtualne implementacje tych składowych, które ukrywają implementację klasy bazowej.

Klasa pochodna może przesłonić składową klasy bazowej tylko wtedy, gdy ta składowa jest zadeklarowana w klasie bazowej jako wirtualna (słowo kluczowe `virtual`) lub abstrakcyjna (słowo kluczowe `abstract`). Pochodna składowa musi używać słowa kluczowego `override`, aby zaznaczyć, że będzie ona brała udział w wywołaniu wirtualnym.

Oto przykład:

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Pola nie mogą być wirtualne. Tylko metody, właściwości, zdarzenia i indeksery mogą być wirtualne. Gdy klasa pochodna przesyła wirtualną składową, jest ona wywoływana, kiedy następuje dostęp do instancji tej klasy jako do instancji klasy bazowej.

Oto przykład:

```
DerivedClass B = new DerivedClass();
B.DoWork(); //Wywołuje nową metodę

BaseClass A = (BaseClass)B;
A.DoWork(); //Również wywołuje nową metodę
```

Wirtualne metody i właściwości klasy pochodnej pozwalają rozszerzyć klasę bazową bez użycia implementacji metody z klasy bazowej.

## Ukrywanie składowych klasy bazowej

Jeżeli chcesz, aby składowa pochodna miała taką samą nazwę jak składowa klasy bazowej, ale nie chcesz, aby była wywoływana wirtualnie, możesz użyć słowa kluczowego `new`. Słowo kluczowe `new` umieszczamy przed typem zwracanym przez składową, która ma być zastąpiona.

Oto przykład:

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

Ukryte składowe klasy bazowej są nadal dostępne poprzez rzutowanie instancji klasy pochodnej nainstancję klasy bazowej, tak jak w przykładzie poniżej:

```
DerivedClass B = new DerivedClass();
B.DoWork(); //Wywołuje nową metodę
```

```
BaseClass A = (BaseClass)B;
A.DoWork(); //Wywołuje starszą metodę
```

## Zapobieganie przesłanianiu wirtualnych składowych klasy pochodnej

Wirtualne składowe pozostają wirtualne przez czas nieokreślony. Nieważne, ile klas zostało zadeklarowanych pomiędzy wirtualną składową a klasą, która oryginalnie tę składową deklaruje. Jeżeli klasa A deklaruje wirtualną składową, a klasa B pochodzi z klasy A i klasa C pochodzi z klasy B, to klasa C dziedziczy wirtualną składową i może ją przesłonić bez względu na to, że klasa B zadeklarowała przesłonięcie tej składowej.

Oto przykład:

```
public class A
{
    public virtual void DoWork() { }
```

```
}
```

```
public class B : A
```

```
{
```

```
    public override void DoWork() { }
```

```
}
```

Klasa pochodna może zatrzymać wirtualne dziedziczenie, deklarując przesłonięcie jako zapieczętowane (`sealed`). Wymaga to umieszczenia słowa kluczowego `sealed` przed słowem kluczowym `override` w deklaracji składowej klasy.

Oto przykład:

```
public class C : B
```

```
{
```

```
    public sealed override void DoWork() { }
```

```
}
```

W poprzednim przykładzie metoda `DoWork` nie jest już dłużej wirtualna dla żadnej klasy pochodnej z `C`. Jest jednak nadal wirtualna dla instancji klasy `C`, nawet gdy wykona się rzutowanie na typ `B` lub `A`. Zapieczętowane (`sealed`) metody mogą być zastąpione w klasie pochodnej za pomocą słowa kluczowego `new`.

Oto przykład:

```
public class D : C
```

```
{
```

```
    public new void DoWork() { }
```

```
}
```

W tym przypadku, jeżeli `DoWork` jest wywoływana z klasy `D` za pomocą zmiennej typu `D`, to wywoływana jest nowa wersja tej metody. Jeżeli zmienna typu `C`, `B` lub `A` jest używana przy dostępie do instancji `D`, to wywołanie metody `DoWork` odbędzie się według zasad wirtualnego dziedziczenia, czyli przekieruje te wywołania do implementacji metody `DoWork` z klasy `C`.

## Dostęp do wirtualnych składowych klasy bazowej z klas pochodnych

Klasa pochodna, która przesłoniła metodę lub właściwość, może mieć nadal dostęp do tej metody lub właściwości poprzez słowo kluczowe `base`.

Oto przykład:

```
public class Base
```

```
{
```

```
    public virtual void DoWork() { /*...*/ }
```

```
}
```

```
public class Derived : Base
```

```
{
```

```
    public override void DoWork()
```

```
    {
```

```
        //kod metody pochodnej
```

```
//...
//wywołanie metody DoWork klasy bazowej
base.DoWork();
}
}
```

## Przesłanianie metody ToString()

Każda klasa lub struktura w C# dziedziczy z klasy Object. Z tego powodu każdy obiekt dostaje metodę ToString, która zwraca reprezentację tego obiektu w postaci napisu. Na przykład każda zmienna typu int ma metodę ToString, która pozwala zwrócić zawartość zmiennej jako napis.

Oto przykład:

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx); //Wypisze: 42
```

Jeżeli tworysz własną klasę lub strukturę, powinieneś przesłonić metodę ToString, aby dostarczyć informacji o stworzonym przez Ciebie typie.

Najpierw należy zadeklarować metodę ToString z określonymi modyfikatorami dostępu i typie zwracanym:

```
public override string ToString(){ }
```

Dalej dobrze jest zaimplementować metodę, która zwraca napis. W poniższym przykładzie zwracana jest nazwa klasy oraz dane specyficzne dla danej instancji klasy:

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}
```

Teraz zostało wypróbować stworzoną metodę takim na przykład kodem:

```
Person person = new Person { Name = "Dawid", Age = 23 };
Console.WriteLine(person); //Wypisze: Person: Dawid 23
```

## 4.16. Delegaty

Delegat jest typem, który definiuje sygnaturę metody. Gdy tworysz instancję delegatu, możesz powiązać ją z metodą o kompatybilnej sygnaturze. Potem możesz wywołać tę metodę poprzez instancję delegatu. Delegaty używamy także do przekazania

metod jako argumentów innych metod (jest to delegat Func, o którym będzie mowa na końcu tego rozdziału). Metody obsługujące zdarzenia (będzie o nich mowa w dalszej części książki) są niczym innym jak metodami wywoływanymi przez delegaty. Delegaty języka C# można dla lepszego zobrazowania porównać do wskaźników na funkcje w języku C++.

W języku C# 1.0 tworzyłeś instancję delegatu, inicjalizując ją metodą zdefiniowaną gdzieś w kodzie. Natomiast C# 2.0 pozwala już tworzyć anonimowe metody. Jest to sposób na napisanie nienazwanego bloku kodu, który jest wykonywany przy wywoływaniu delegatu. W C# 3.0 wprowadzono już wyrażenia lambda, które są podobne do metod anonimowych, ale bardziej wyraziste i zwięzłe. Te dwie cechy są znane jako funkcje anonimowe. Podsumowując, aplikacje, których celem jest Framework w wersji 3.5 i nowszej, powinny korzystać z wyrażeń lambda.

Oto przykład demonstrujący ewolucję delegatów od C# 1.0 do C# 3.0:

```
class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        //Oryginalna składnia delegatu wymaga
        //inicjalizacji metodą nazwaną
        TestDelegate testDelA = new TestDelegate(M);

        //C# 2.0: Delegat może być inicjalizowany
        //metodą anonimową. Ta metoda pobiera
        //napis jako parametr wejściowy.
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

        //C# 3.0. Delegat może być inicjalizowany
        //wyrażeniem lambda. Parametr wejściowy to x
        //Typ parametru jest określany przez kompilator
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        //wywołanie delegatów
        testDelA("Witam. Jestem M i piszę ten tekst.");
        testDelB("Jestem anonimowy.");
        testDelC("Jestem również znany autorem.");
    }
}
```

## Metody anonimowe

Używając metod anonimowych, zmniejszasz złożoność kodu, gdyż nie musisz tworzyć osobnej metody dla delegatu.

Poniższy przykład prezentuje dwa sposoby tworzenia delegatu:

- ◆ Skojarzenie delegatu z metodą anonimową.
- ◆ Skojarzenie delegatu z metodą nazwaną.

Oto przykład:

```
//deklaracja delegatu
delegate void Printer(string s);

class TestClass
{
    static void Main()
    {
        //delegat z metodą anonimową
        Printer p = delegate(string j)
        {
            System.Console.WriteLine(j);
        };

        p("Delegat z użyciem metody anonimowej został wywołany.");
    }

    //tworzenie instancji delegatu za pomocą metody nazwanej DoWork
    p = new Printer(TestClass.DoWork);

    p("Delegat z użyciem metody nazwanej został wywołany.");
}

//metoda skojarzona z nazwanym delegatem
static void DoWork(string k)
{
    System.Console.WriteLine(k);
}
```

## Wyrażenia lambda

Wyrażenie lambda jest anonimową funkcją, którą możesz użyć do stworzenia delegatu. Używając wyrażeń lambda, możesz pisać lokalne funkcje, które będą przesłane jako argumenty funkcji lub jako wartość zwracana.

Jeżeli chcesz stworzyć wyrażenie lambda, określasz parametry wejściowe (jeżeli są z lewej strony operatora => i piszesz wyrażenie lub blok kodu z prawej strony operatora. Na przykład wyrażenie  $x \Rightarrow x * x$  określa parametr nazwany  $x$  i zwraca wartość  $x*x$ .

Wyrażenie możesz przypisać do delegatu, tak jak w przykładzie poniżej:

```
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

## Delegat Func

Enkapsuluje metodę z jednym parametrem (lub bez parametrów) z wartością zwracaną określona w parametrze TResult.

Składnia:

```
public delegate TResult Func<in T, out TResult>(  
    T arg  
)
```

lub

```
public delegate TResult Func<out TResult>()
```

Oto przykład:

```
using System;  
using System.IO;  
  
public class TestDelegate  
{  
    public static void Main()  
    {  
        OutputTarget output = new OutputTarget();  
        Func<bool> methodCall = output.SendToFile;  
        if (methodCall())  
            Console.WriteLine("Sukces!");  
        else  
            Console.WriteLine("Operacja zapisu się nie powiodła.");  
    }  
}  
  
public class OutputTarget  
{  
    public bool SendToFile()  
    {  
        try  
        {  
            string fn = Path.GetTempFileName();  
            StreamWriter sw = new StreamWriter(fn);  
            sw.WriteLine("Witaj świecie!");  
            sw.Close();  
            return true;  
        }  
        catch  
        {  
            return false;  
        }  
    }  
}
```

## 4.17. Zdarzenia

Zdarzenia pozwalają klasie lub obiektowi poinformować inne klasy lub obiekty, że w programie zaszło coś wartego uwagi. Klasa, która wysyła zdarzenie, nazywana jest wydawcą, a klasy, które obsługują zdarzenia, są nazywane subskrybentami.

W typowej aplikacji *Windows Forms* (o czym będzie mowa w dalszej części książki) subskrybuje się zdarzenia wysypane przez kontrolki takie jak listy czy przyciski. Środowisko Visual Studio pozwala wybrać zdarzenia, które chcemy obsługiwać, i wygenerować dla nich odpowiedni kod.

Zdarzenia:

- ◆ Wydawca sprawdza, czy zdarzenie zostało wysłane. Subskrybenci decydują, jaką akcję wykonać w odpowiedzi na zdarzenie.
- ◆ Zdarzenie może mieć wielu subskrybentów. Subskrybent może obsługiwać wiele zdarzeń, od wielu wydawców.
- ◆ Zdarzenia, które nie mają subskrybentów, nigdy nie są wysypane.
- ◆ W bibliotece klas .NET Framework zdarzenia są bazowane na delegacie `EventHandler` i klasie bazowej `EventArgs`.

## 4.18. Metody rozszerzające

Za chwilę dowiesz się, jak zaimplementować własne metody rozszerzające dla każdego typu w bibliotece klas .NET Framework lub jakiegokolwiek innego typu .NET, który chcesz rozszerzyć.

Załóżmy, że do klasy `System.String` chcemy dodać metodę, która zwróci ilość słów w zmiennej tego typu.

Najpierw tworzymy przestrzeń nazw o przykładowej nazwie `ExtensionMethods`, a w niej statyczną oraz publiczną klasę `MyExtensions`. W tej klasie tworzymy statyczną oraz publiczną metodę `WordCount`, która zwraca ilość słów w napisie. A wszystko wygląda następująco:

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

Pierwszy parametr metody rozszerzającej określa, na jakim typie operuje metoda. Przed typem tego parametru należy dodać słowo kluczowe `this`.

Aby użyć metody z tej przestrzeni nazw, należy na górze kodu programu dodać linię:

```
using ExtensionMethods;
```

Przykładowe użycie metody zliczającej ilość słów wygląda tak:

```
string s = "Hello Extension Methods";
int i = s.WordCount();
Console.WriteLine("i == {0}", i); //Wypisze: i == 3
```

Na koniec jeszcze przedstawię cały przykładowy program, gotowy do kompilacji i przetestowania:

```
using System;
using ExtensionMethods;

namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' })
                .StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

class Program
{

    static void Main()
    {
        string s = "Hello Extension Methods";
        int i = s.WordCount();
        Console.WriteLine("i == {0}", i); //Wypisze: i == 3
    }
}
```

## 4.19. Kolekcje

Zdarza się w programowaniu, że musimy przechowywać dużą ilość obiektów. Początkujący programista pomyśli o tablicach, ale one są mało elastyczne, a potrzebujemy czegoś, co łatwiej pozwoli nam operować na przechowywanych obiektach. Tutaj z pomocą przychodzą nam kolekcje. Są one bardziej nowoczesne od tablic i w większości przypadków lepsze.

## Wybieranie klasy kolekcji

Musisz być pewny wyboru klasy z System.Collections. Używanie zlego typu może ograniczyć możliwości Twojej kolekcji.

- ◆ Czy potrzebujesz ciąglej listy, w której element jest wyrzucany po pobraniu jego wartości?
  - ◆ Jeśli tak, rozważ użycie klasy Queue, w której pierwszy element na wejściu jest pierwszym na wyjściu (*first-in-first-out, FIFO*). Jeśli potrzebujesz, aby element ostatni na wejściu był pierwszym na wyjściu (*last-in-first-out, LIFO*), użyj klasy Stack.
  - ◆ Jeżeli nie, rozważ użycie innych kolekcji.
- ◆ Czy potrzebujesz mieć dostęp do elementów w określonym porządku, takim jak LIFO czy FIFO?
  - ◆ Klasa Queue oferuje dostęp typu FIFO (pierwszy na wejściu to pierwszy na wyjściu).
  - ◆ Klasa Stack oferuje dostęp typu LIFO (ostatni na wejściu to pierwszy na wyjściu).
  - ◆ Klasa generyczna LinkedList oferuje dostęp do elementów od początku do końca lub od końca do początku.
  - ◆ Reszta kolekcji oferuje dostęp swobodny.
- ◆ Czy potrzebujesz mieć dostęp do określonego elementu przez indeks?
  - ◆ Klasy ArrayList i StringCollection oraz klasa generyczna List oferują dostęp do elementów poprzez indeks (od zera).
  - ◆ Klasy Hashtable, SortedList, ListDictionary i StringDictionary oraz klasy generyczne Dictionary i SortedDictionary oferują dostęp do elementów poprzez klucz.
  - ◆ Klasy NameObjectCollectionBase i NameValueCollection oraz klasy generyczne KeyedCollection i SortedList oferują dostęp do elementów poprzez indeks (od zera) lub poprzez klucz.
- ◆ Czy każdy element będzie zawierał wartość, kombinację jeden klucz i jedna wartość, czy kombinację jeden klucz i wiele wartości?
  - ◆ Jedna wartość: użyj jakiejkolwiek kolekcji, która bazuje na interfejsie IList.
  - ◆ Jeden klucz i jedna wartość: użyj jakiejkolwiek kolekcji, która bazuje na interfejsie IDictionary.
  - ◆ Jedna wartość z wbudowanym kluczem: użyj klasy generycznej KeyedCollection.
  - ◆ Jeden klucz i wiele wartości: użyj klasy NameValueCollection.

- ◆ Czy potrzebujesz sortować elementy w innej kolejności niż tej, w której zostały podane?
  - ◆ Klasa `Hashtable` sortuje elementy po ich kodach (ang. *hash codes*).
  - ◆ Klasa `SortedList` oraz klasy generyczne `SortedDictionary` i `SortedList` sortują swoje elementy według klucza, bazując na implementacji interfejsu `IComparer`.
  - ◆ `ArrayList` dostarcza metodę `Sort`, która pobiera implementację `IComparer` jako parametr.
- ◆ Czy potrzebujesz szybko wyszukiwać i pobierać informacje?
  - ◆ `ListDictionary` jest szybsza niż `Hashtable` dla małych kolekcji (10 i mniej). Klasa generyczna `SortedDictionary` oferuje szybszy dostęp niż klasa generyczna `Dictionary`.
- ◆ Czy potrzebujesz kolekcji, która przyjmuje tylko typ napisowy (`String`)?
  - ◆ `StringCollection` (bazowana na `IList`) i `StringDictionary` (bazowana na `IDictionary`) znajdują się w przestrzeni nazw `System.Collections.Specialized`.
  - ◆ Dodatkowo możesz użyć jakiejkolwiek generycznej kolekcji z przestrzeni nazw `System.Collections.Generic`, określając jej generyczne argumenty jako typ `String`.

## Klasa Queue

Reprezentuje kolekcję obiektów (kolejkę), w której pierwszy na wejściu jest pierwszym na wyjściu (*first-in-first-out, FIFO*).

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public class Queue : ICollection, IEnumerable, ICloneable
```

Przykładowy kod:

```
using System;
using System.Collections;

class Program
{
    static void Main()
    {
        Queue myQ = new Queue(); //utworzenie kolejki

        //dodanie elementów do kolejki
        myQ.Enqueue("Witaj");
        myQ.Enqueue("świecie");
        myQ.Enqueue("!");
    }
}
```

```
//wypisanie ilości elementów oraz ich wartości
Console.WriteLine("myQ");
Console.WriteLine("Ilość: {0}", myQ.Count);
Console.Write("Wartości:");
foreach (Object obj in myQ)
    Console.Write("{0}", obj);
Console.WriteLine();
}
}
```

## Klasa Stack

Reprezentuje prostą niegeneryczną kolekcję obiektów (stos), w której ostatni na wejściu jest pierwszym na wyjściu (*last-in-first-out, LIFO*).

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public class Stack : ICollection, IEnumerable, ICloneable
```

Przykładowy kod:

```
using System;
using System.Collections;

class Program
{
    static void Main()
    {
        Stack myStack = new Stack(); //utworzenie stosu

        //odłożenie elementów na stos
        myStack.Push("Witaj");
        myStack.Push("świecie");
        myStack.Push("!");

        //wyświetlenie ilości elementów
        Console.WriteLine("myStack");
        Console.WriteLine("Ilość: {0}", myStack.Count);
        Console.Write("Wartości:");

        //zdjęcie ze stosu elementu (znaku "!")
        myStack.Pop();

        //wyświetlenie wartości wszystkich elementów
        foreach (Object obj in myStack)
            Console.Write("{0}", obj);
    }
}
```

## Klasa ArrayList

Implementuje interfejs `IList` za pomocą tablicy, której rozmiar jest dynamicznie zwiększany w miarę potrzeby.

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public class ArrayList : IList, ICollection, IEnumerable,
    ICloneable
```

Przykładowy kod:

```
using System;
using System.Collections;

class Program
{

    static void Main()
    {
        //utworzenie ArrayList
        ArrayList myAL = new ArrayList();

        //dodanie elementów
        myAL.Add("Witaj");
        myAL.Add("świecie");
        myAL.Add("!");

        //wyświetlenie właściwości oraz wartości z myAL
        Console.WriteLine("myAL:");
        Console.WriteLine("Ilość elementów: {0}", myAL.Count);
        Console.WriteLine("Pojemność: {0}", myAL.Capacity);
        Console.Write("Wartości:");
        foreach (Object obj in myAL)
            Console.Write(" {0}", obj);
    }
}
```

## Klasa StringCollection

Reprezentuje kolekcję napisów.

Składnia:

```
[SerializableAttribute]
public class StringCollection : IList, ICollection, IEnumerable
```

Przykładowy kod:

```
using System;
using System.Collections;
using System.Collections.Specialized;
```

```
class Program
{
    static void Main()
    {
        //utworzenie kolekcji napisów
        StringCollection myCol = new StringCollection();

        //utworzenie tablicy napisów
        String[] myArr = new String[] { "czerwony", "zielony", "niebieski" };

        //dodanie napisów do kolekcji
        myCol.AddRange(myArr);

        Console.WriteLine("Wartości kolekcji napisów:");

        //wyświetlenie elementów kolekcji
        foreach (Object obj in myCol)
            Console.WriteLine("{0}", obj);
    }
}
```

## Klasa Hashtable

Reprezentuje kolekcję par klucz/wartość.

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public class Hashtable : IDictionary, ICollection, IEnumerable,
    ISerializable, IDeserializationCallback, ICloneable
```

Przykładowy kod:

```
using System;
using System.Collections;

class Program
{
    static void Main()
    {
        //utworzenie Hashtable
        Hashtable openWith = new Hashtable();

        //dodanie kluczy i wartości
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        //wyświetlenie wartości dla klucza "rtf"
        Console.WriteLine("dla klucza \"rtf\" wartość = {0}.", openWith["rtf"]);

        //wyświetlenie par klucz/wartość
        foreach (DictionaryEntry de in openWith)
```

```
        {
            Console.WriteLine("klucz = {0}, wartość = {1}", de.Key, de.Value);
        }
    }
}
```

## Klasa SortedList

Reprezentuje kolekcję par klucz/wartość posortowanych według klucza i dostępnych poprzez klucz lub indeks.

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public class SortedList : IDictionary, ICollection, IEnumerable,
    ICloneable
```

Przykładowy kod:

```
using System;
using System.Collections;

class Program
{

    static void Main()
    {
        //utworzenie i inicjalizacja nowej SortedList
        SortedList mySL = new SortedList();
        mySL.Add("Pierwszy", "Witaj");
        mySL.Add("Drugi", "świecie");
        mySL.Add("Trzeci", "!");

        //wyświetlenie właściwości i wartości
        Console.WriteLine("mySL");
        Console.WriteLine(" Ilość: {0}", mySL.Count);
        Console.WriteLine(" Pojemność: {0}", mySL.Capacity);
        Console.WriteLine(" Klucze i wartości:");

        Console.WriteLine("\t-Klucz-\t-Wartość-");
        for (int i = 0; i < mySL.Count; i++)
        {
            Console.WriteLine("\t{0}:\t{1}", mySL.GetKey(i), mySL.GetByIndex(i));
        }
    }
}
```

## Klasa ListDictionary

Implementuje interfejs `IDictionary` za pomocą listy. Klasa polecana dla kolekcji o ilości elementów równej 10 lub mniej.

Składnia:

```
[SerializableAttribute]
public class ListDictionary : IDictionary, ICollection, IEnumerable
```

Przykładowy kod:

```
using System;
using System.Collections;
using System.Collections.Specialized;

class Program
{
    static void Main()
    {
        //utworzenie ListDictionary
        ListDictionary myCol = new ListDictionary();

        //dodanie kluczy i wartości
        myCol.Add("Jabłka Braeburn", "1.49");
        myCol.Add("Jabłka Fuji", "1.29");
        myCol.Add("Jabłka Gala", "1.49");

        //wyświetlenie par klucz/wartość za pomocą pętli foreach
        Console.WriteLine("    Klucz           Wartość");
        foreach (DictionaryEntry de in myCol)
            Console.WriteLine("    {0,-25} {1}", de.Key, de.Value);
    }
}
```

## Klasa StringDictionary

Implementuje kolekcję z parami klucz/wartość silnie typowaną na typ napisowy.

Składnia:

```
[SerializableAttribute]
public class StringDictionary : IEnumerable
```

Przykładowy kod:

```
using System;
using System.Collections;
using System.Collections.Specialized;

class Program
{
    static void Main()
    {
        //utworzenie i inicjalizacja StringDictionary
        StringDictionary myCol = new StringDictionary();
        myCol.Add("czerwony", "red");
        myCol.Add("zielony", "green");
        myCol.Add("niebieski", "blue");
    }
}
```

```
//wyświetlenie kluczy i wartości
Console.WriteLine("  Klucz           Wartość");
foreach (DictionaryEntry de in myCol)
    Console.WriteLine(" {0,-25} {1}", de.Key, de.Value);
}
}
```

## Klasa NameObjectCollectionBase

Dostarcza abstrakcyjną klasę bazową dla kolekcji z kluczem typu napisowego, natomiast z wartościami typu obiektowego. Dostęp do elementów może odbywać się poprzez klucz lub indeks.

Składnia:

```
[SerializableAttribute]
public abstract class NameObjectCollectionBase : ICollection, IEnumerable,
➥ISerializable,
IDeserializationCallback
```

Przykładowy kod:

```
using System;
using System.Collections;
using System.Collections.Specialized;

public class MyCollection : NameObjectCollectionBase
{
    //tworzy pustą kolekcję
    public MyCollection()
    {
    }

    //dodaje elementy z IDictionary do nowej kolekcji
    public MyCollection(IDictionary d, Boolean bReadOnly)
    {
        foreach (DictionaryEntry de in d)
        {
            this.BaseAdd((String)de.Key, de.Value);
        }
        this.IsReadOnly = bReadOnly;
    }

    //pobieranie pary klucz/wartość za pomocą indeksu
    public DictionaryEntry this[int index]
    {
        get
        {
            return (new DictionaryEntry(
                this.BaseGetKey(index), this.BaseGet(index)));
        }
    }

    //pobranie/ustawienie wartości powiązanej z określonym kluczem
    public Object this[String key]
```

```
{  
    get  
    {  
        return (this.BaseGet(key));  
    }  
    set  
    {  
        this.BaseSet(key, value);  
    }  
}  
  
//pobranie tablicy napisów ze wszystkimi kluczami kolekcji  
public String[] AllKeys  
{  
    get  
    {  
        return (this.BaseGetAllKeys());  
    }  
}  
  
//pobranie tablicy obiektów ze wszystkimi wartościami kolekcji  
public Array AllValues  
{  
    get  
    {  
        return (this.BaseGetAllValues());  
    }  
}  
  
//pobranie tablicy napisów ze wszystkimi wartościami kolekcji  
public String[] AllStringValues  
{  
    get  
    {  
        return ((String[])this.BaseGetAllValues(typeof(string)));  
    }  
}  
  
//pobranie wartości określającej, czy klucze nie są zerem  
public Boolean HasKeys  
{  
    get  
    {  
        return (this.BaseHasKeys());  
    }  
}  
  
//dodanie wpisu do kolekcji  
public void Add(String key, Object value)  
{  
    this.BaseAdd(key, value);  
}  
  
//usunięcie wpisu z określonym kluczem z kolekcji  
public void Remove(String key)  
{  
    this.BaseRemove(key);  
}
```

```
//usunięcie wpisu z określonym indeksem z kolekcji
public void Remove(int index)
{
    this.BaseRemoveAt(index);
}

//usunięcie wszystkich elementów kolekcji
public void Clear()
{
    this.BaseClear();
}

}

class Program
{

    static void Main()
    {
        //utworzenie i inicjalizacja nowej kolekcji (tylko do odczytu)
        IDictionary d = new ListDictionary();
        d.Add("czerwony", "jabłko");
        d.Add("żółty", "banan");
        d.Add("zielony", "gruszka");
        MyCollection myROCol = new MyCollection(d, true);

        //próba dodania nowego elementu (wystąpi błąd, bo kolekcja jest do odczytu)
        try
        {
            myROCol.Add("niebieski", "niebo");
        }
        catch (NotSupportedException e)
        {
            Console.WriteLine(e.ToString());
        }

        //wypisanie kluczy i wartości
        Console.WriteLine("Kolekcja tylko do odczytu:");
        PrintKeysAndValues(myROCol);

        //utworzenie i inicjalizacja pustej kolekcji (do zapisu)
        MyCollection myRWCol = new MyCollection();

        //dodanie nowych elementów do kolekcji
        myRWCol.Add("fioletowy", "winogron");
        myRWCol.Add("pomarańczowy", "mandarynka");
        myRWCol.Add("czarny", "jagody");
        Console.WriteLine("Kolekcja do zapisu (po dodaniu wartości):");
        PrintKeysAndValues(myRWCol);

        //zmiana wartości elementu
        myRWCol["pomarańczowy"] = "grejpfrut";
        Console.WriteLine("Kolejka do zapisu (po zmianie jednej wartości):");
        PrintKeysAndValues(myRWCol);

        //usunięcie jednego wpisu z kolekcji
        myRWCol.Remove("czarny");
    }
}
```

```
Console.WriteLine("Kolejka do zapisu (po usunięciu jednej wartości):");
PrintKeysAndValues(myRwCol);

//usunięcie wszystkich elementów kolekcji
myRwCol.Clear();
Console.WriteLine("Kolejka do zapisu (po wyczyszczeniu kolekcji):");
PrintKeysAndValues(myRwCol);
}

//Wypisuje indeksy, klucze i wartości
public static void PrintKeysAndValues(MyCollection myCol)
{
    for (int i = 0; i < myCol.Count; i++)
    {
        Console.WriteLine("[{0}] : {1}, {2}", i, myCol[i].Key, myCol[i].Value);
    }
}

//Wypisuje klucze i wartości, używając AllKeys
public static void PrintKeysAndValues2(MyCollection myCol)
{
    foreach (String s in myCol.AllKeys)
    {
        Console.WriteLine("{0}, {1}", s, myCol[s]);
    }
}
```

## Klasa NameValueCollection

Reprezentuje kolekcję o wartościach i kluczach typu napisowego.

Składnia:

```
[SerializableAttribute]
public class NameValueCollection : NameObjectCollectionBase
```

Przykładowy kod:

```
using System;
using System.Collections;
using System.Collections.Specialized;

class Program
{

    static void Main()
    {
        //utworzenie i inicjalizacja nowej kolekcji
        NameValueCollection myCol = new NameValueCollection();
        myCol.Add("red", "czerwony");
        myCol.Add("green", "zielony");
        myCol.Add("blue", "niebieski");
        myCol.Add("red", "czerwien");

        //wyświetlenie wartości kolekcji na dwa różne sposoby
    }
}
```

```
Console.WriteLine("Wyświetlenie elementów za pomocą AllKeys i Item:");
PrintKeysAndValues(myCol);
Console.WriteLine("Wyświetlenie elementów za pomocą GetKey i Get:");
PrintKeysAndValues2(myCol);

//pobranie wartości przez indeks lub klucz
Console.WriteLine("Indeks 1 zawiera wartość {0}.", myCol[1]);
Console.WriteLine("Klucz \"red\" zawiera wartość {0}.", myCol["red"]);
Console.WriteLine();

//kopianie wartości do tablicy napisów
String[] myStrArr = new String[myCol.Count];
myCol.CopyTo(myStrArr, 0);
Console.WriteLine("Tablica napisów zawiera:");
foreach (String s in myStrArr)
    Console.WriteLine(" {0}", s);
Console.WriteLine();

//wyszukanie i usunięcie klucza
myCol.Remove("green");
Console.WriteLine("Kolekcja zawiera następujące elementy (po usunięciu
→klucza \"green\"):");
PrintKeysAndValues(myCol);

//wyczyszczenie całej kolekcji
myCol.Clear();
Console.WriteLine("Kolekcja zawiera następujące elementy po wyczyszczeniu:");
PrintKeysAndValues(myCol);
}

public static void PrintKeysAndValues(NameValueCollection myCol)
{
    IEnumerator myEnumerator = myCol.GetEnumerator();
    Console.WriteLine(" KLUCZ      WARTOŚĆ");
    foreach (String s in myCol.AllKeys)
        Console.WriteLine(" {0,-10} {1}", s, myCol[s]);
    Console.WriteLine();
}

public static void PrintKeysAndValues2(NameValueCollection myCol)
{
    Console.WriteLine(" [INDEKS] KLUCZ      WARTOŚĆ");
    for (int i = 0; i < myCol.Count; i++)
        Console.WriteLine(" [{0}]      {1,-10} {2}", i, myCol.GetKey(i),
→myCol.Get(i));
    Console.WriteLine();
}
```

## 4.20. Typy generyczne

Typy generyczne zostały dodane do języka C# w wersji 2.0. Są one podobne do szablonów z języka C++. Za pomocą typów generycznych często tworzy się kolekcje.

Typ generyczny tworzymy w następujący sposób:

```
class Generic<T>
{
    public void Show(T var1)
    {
        Console.WriteLine("{0}", var1);
    }
}
```

Teraz taki prosty typ generyczny możemy użyć w następujący sposób:

```
Generic<int> myG = new Generic<int>();
myG.Show(64); //Wypisze: 64
```

Mogemy także tworzyć metody generyczne, np.:

```
using System;

class Program
{

    static void Main()
    {
        Method1("Witaj!"); //Wypisze: Witaj!
        Method1(32); //Wypisze: 32
        Method1(2.5); //Wypisze: 2.5
    }

    static void Method1<T>(T var1)
    {
        Console.WriteLine("{0}", var1);
    }
}
```

## Klasa generyczna Queue

Reprezentuje kolekcję obiektów (kolejkę), w której pierwszy na wejściu jest pierwszym na wyjściu (*first-in, first-out, FIFO*).

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(false)]
public class Queue<T> : IEnumerable<T>, ICollection,
    IEnumerable
```

Przykładowy kod:

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

class Program
{
    static void Main()
    {
```

```
Queue<string> numbers = new Queue<string>();
numbers.Enqueue("jeden");
numbers.Enqueue("dwa");
numbers.Enqueue("trzy");
numbers.Enqueue("cztery");
numbers.Enqueue("pięć");

//wyświetlenie elementów kolejki
foreach (string number in numbers)
{
    Console.WriteLine(number);
}

Console.WriteLine("Następny element do wyjścia: {0}",
    numbers.Peek());
Console.WriteLine("Element '{0}' wyszedł z kolejki", numbers.Dequeue());

//usunięcie wszystkich elementów kolejki
numbers.Clear();
}
```

## Klasa generyczna Stack

Klasa generyczna reprezentująca stos.

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(false)]
public class Stack<T> : IEnumerable<T>, ICollection,
    IEnumerable
```

Przykładowy kod:

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

class Program
{
    static void Main()
    {
        Stack<string> numbers = new Stack<string>();
        numbers.Push("jeden");
        numbers.Push("dwa");
        numbers.Push("trzy");
        numbers.Push("cztery");
        numbers.Push("pięć");

//wyświetlenie elementów stosu
foreach (string number in numbers)
{
    Console.WriteLine(number);
}

//usunięcie wszystkich elementów
```

```
        numbers.Clear();
    }
}
```

## Klasa generyczna LinkedList

Lista podwójnie łączona.

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(false)]
public class LinkedList<T> : ICollection<T>, IEnumerable<T>,
    ICollection, IEnumerable, ISerializable, IDeserializationCallback
```

Przykładowy kod:

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

class Program
{
    static void Main()
    {
        //utworzenie listy
        string[] words = { "lis", "skoczył", "nad", "psem" };
        LinkedList<string> sentence = new LinkedList<string>(words);
        Display(sentence, "Wartości listy:");

        //Dodaj słowo „dzisiaj” na początek listy
        sentence.AddFirst("dzisiaj");
        Display(sentence, "Test 1:");

        //Przenies pierwszy element, tak aby był ostatnim
        LinkedListNode<string> mark1 = sentence.First;
        sentence.RemoveFirst();
        sentence.AddLast(mark1);
        Display(sentence, "Test 2:");
    }

    private static void Display(LinkedList<string> words, string test)
    {
        Console.WriteLine(test);
        foreach (string word in words)
        {
            Console.Write(word + " ");
        }
        Console.WriteLine();
        Console.WriteLine();
    }
}
```

## Klasa generyczna List

Silnie typowana lista obiektów z dostępem poprzez indeks.

Składnia:

```
[SerializableAttribute]
public class List<T> : IList<T>, ICollection<T>,
    IEnumerable<T>, IList, ICollection, IEnumerable
```

Przykładowy kod:

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

class Program
{
    static void Main()
    {
        //utworzenie listy
        List<string> dinosaurs = new List<string>();

        //wyświetlenie pojemności listy
        Console.WriteLine("\nPojemność: {0}", dinosaurs.Capacity);

        //dodanie nowych pozycji do listy
        dinosaurs.Add("Tyrannosaurus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Mamenchisaurus");
        dinosaurs.Add("Deinonychus");
        dinosaurs.Add("Compsognathus");

        Console.WriteLine();
        foreach (string dinosaur in dinosaurs)
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nPojemność: {0}", dinosaurs.Capacity);
        Console.WriteLine("Ilość: {0}", dinosaurs.Count);

        Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
            dinosaurs.Contains("Deinonychus")); //sprawdzenie, czy lista zawiera daną pozycję

        Console.WriteLine("\nInsert(2, \"Compsognathus\")");
        dinosaurs.Insert(2, "Compsognathus"); //wstawienie nowej pozycji pod indeks 2

        Console.WriteLine();
        foreach (string dinosaur in dinosaurs)
        {
            Console.WriteLine(dinosaur);
        }

        //dostęp poprzez indeks
        Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);
```

```
Console.WriteLine("\nRemove(\"Compsognathus\")");
dinosaurs.Remove("Compsognathus"); //usuńcie pozycji z listy

Console.WriteLine();
foreach (string dinosaur in dinosaurs)
{
    Console.WriteLine(dinosaur);
}

//Przytnij listę
dinosaurs.TrimExcess();
Console.WriteLine("\nTrimExcess()");
Console.WriteLine("Pojemność: {0}", dinosaurs.Capacity);
Console.WriteLine("Ilość: {0}", dinosaurs.Count);

//Wyczyść listę
dinosaurs.Clear();
Console.WriteLine("\nClear()");
Console.WriteLine("Pojemność: {0}", dinosaurs.Capacity);
Console.WriteLine("Ilość: {0}", dinosaurs.Count);
}

}
```

## Klasa generyczna Dictionary

Reprezentuje kolekcję kluczów i wartości.

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(false)]
public class Dictionary<TKey, TValue> : IDictionary<TKey, TValue>,
ICollection<KeyValuePair<TKey, TValue>>,
IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable,
ISerializable, IDeserializationCallback
```

Przykładowy kod:

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

class Program
{
    static void Main()
    {
        //utworzenie nowej kolekcji napisów z kluczami (również jako napisy)
        Dictionary<string, string> openWith =
            new Dictionary<string, string>();

        //dodanie kilku elementów. Nie ma tutaj duplikatów kluczy,
        //ale są duplikaty wartości
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
```

```
//próba dodania nowego klucza, który już jest w kolekcji
try
{
    openWith.Add("txt", "winword.exe");
}
catch (ArgumentException)
{
    Console.WriteLine("Element o kluczu = \"txt\" już istnieje.");
}

//dostęp do wartości poprzez podanie klucza
Console.WriteLine("Dla klucza = \"rtf\", wartość = {0}.", 
    openWith["rtf"]);

//zmiana wartości
openWith["rtf"] = "winword.exe";
Console.WriteLine("Dla klucza = \"rtf\", wartość = {0}.", 
    openWith["rtf"]);

//Jeżeli klucz nie istnieje, ustawienie indeksera doda nową parę klucz/wartość
openWith["doc"] = "winword.exe";

//Jeżeli klucza nie ma w słowniku, rzucony zostanie wyjątek
try
{
    Console.WriteLine("Dla klucza = \"tif\", wartość = {0}.", 
        openWith["tif"]);
}
catch (KeyNotFoundException)
{
    Console.WriteLine("Klucz = \"tif\" nie istnieje.");
}

//Metoda ContainsKey może być użyta do sprawdzenia, czy klucz istnieje
if (!openWith.ContainsKey("ht"))
{
    openWith.Add("ht", "hypertrm.exe");
    Console.WriteLine("Dodano wartość dla klucza = \"ht\": {0}." , 
        openWith["ht"]);
}

//Podczas dostępu do kolekcji poprzez pętlę foreach
//elementy są zwracane jako obiekty KeyValuePair
Console.WriteLine();
foreach (KeyValuePair<string, string> kvp in openWith)
{
    Console.WriteLine("Klucz = {0}, Wartość = {1}",
        kvp.Key, kvp.Value);
}

//Do pobrania samych wartości użyj właściwości Values
Dictionary<string, string>.ValueCollection valueColl =
    openWith.Values;

//ElementyValueCollection są silnie typowane
Console.WriteLine();
foreach (string s in valueColl)
{
    Console.WriteLine("Wartość = {0}", s);
```

```
}

//Do pobrania samych kluczy użyj właściwości Keys
Dictionary<string, string>.KeyCollection keyColl =
    openWith.Keys;

//Elementy KeyCollection są silnie typowane
Console.WriteLine();
foreach (string s in keyColl)
{
    Console.WriteLine("Klucz = {0}", s);
}

//Używaj metody Remove do usunięcia pary klucz/wartość
Console.WriteLine("\nRemove(\"doc\")");
openWith.Remove("doc");

if (!openWith.ContainsKey("doc"))
{
    Console.WriteLine("Klucz \"doc\" nie istnieje.");
}
}
```

## Klasa generyczna SortedDictionary

Reprezentuje kolekcję par klucz/wartość posortowaną według kluczy.

Składnia:

```
[SerializableAttribute]
public class SortedDictionary<TKey, TValue> : IDictionary<TKey, TValue>,
    ICollection<KeyValuePair<TKey, TValue>>,
    IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable
```

Przykładowy kod:

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

class Program
{
    static void Main()
    {
        //utworzenie nowej posortowanej kolekcji napisów z kluczami (również jako napisami)
        SortedDictionary<string, string> openWith =
            new SortedDictionary<string, string>();

        //dodanie kilku elementów do kolekcji. Wartości mogą się powtarzać
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        //Metoda Add rzuci wyjątek, jeżeli klucz już jest w kolekcji
        try
```

```
{  
    openWith.Add("txt", "winword.exe");  
}  
catch (ArgumentException)  
{  
    Console.WriteLine("Element o kluczu = \"txt\" już istnieje.");  
}  
  
//dostęp do wartości poprzez klucz  
Console.WriteLine("Dla klucza = \"rtf\", wartość = {0}.".openWith["rtf"]);  
  
//zmiana wartości  
openWith["rtf"] = "winword.exe";  
Console.WriteLine("Dla klucza = \"rtf\", wartość = {0}.".openWith["rtf"]);  
  
//Jeżeli klucz nie istnieje, zostanie dodany  
openWith["doc"] = "winword.exe";  
  
//Jeżeli klucz nie istnieje, rzucony zostanie wyjątek  
try  
{  
    Console.WriteLine("Dla klucza = \"tif\", wartość = {0}.".openWith["tif"]);  
}  
catch (KeyNotFoundException)  
{  
    Console.WriteLine("Klucz = \"tif\" nie istnieje.");  
}  
  
//sprawdzenie, czy da się pobrać wartość, a następnie pobranie jej  
string value = "";  
if (openWith.TryGetValue("tif", out value))  
{  
    Console.WriteLine("Dla klucza = \"tif\", wartość = {0}.".value);  
}  
else  
{  
    Console.WriteLine("Klucz = \"tif\" nie istnieje.");  
}  
  
//ContainsKey może być użyta do sprawdzenia kluczy przed ich dodaniem  
if (!openWith.ContainsKey("ht"))  
{  
    openWith.Add("ht", "hypertrm.exe");  
    Console.WriteLine("Dodano wartość dla klucza = \"ht\": {0}.".openWith["ht"]);  
}  
  
//Jeżeli używasz pętli foreach przy dostępie do elementów kolekcji.  
//są one zwracane jako obiekty KeyValuePair  
Console.WriteLine();  
foreach (KeyValuePair<string, string> kvp in openWith)  
{  
    Console.WriteLine("Klucz = {0}, Wartość = {1}.".kvp.Key, kvp.Value);  
}
```

```
}

//Do pobrania samych wartości użyj właściwości Values
SortedDictionary<string, string>.ValueCollection valueColl =
    openWith.Values;

//ElementyValueCollection są silnie typowane
Console.WriteLine();
foreach (string s in valueColl)
{
    Console.WriteLine("Wartość = {0}", s);
}

//Do pobrania samych kluczy użyj właściwości Keys
SortedDictionary<string, string>.KeyCollection keyColl =
    openWith.Keys;

//Elementy KeyCollection są silnie typowane
Console.WriteLine();
foreach (string s in keyColl)
{
    Console.WriteLine("Klucz = {0}", s);
}

//Użyj metody Remove do usunięcia pary klucz/wartość
Console.WriteLine("\nRemove(\"doc\")");
openWith.Remove("doc");

if (!openWith.ContainsKey("doc"))
{
    Console.WriteLine("Klucz \"doc\" nie istnieje.");
}
}
```

## Klasa generyczna KeyedCollection

Dostarcza abstrakcyjną klasę bazową, której klucze są wbudowane w wartości.

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(false)]
public abstract class KeyedCollection<TKey, TItem> : Collection<TItem>
```

Przykładowy kod:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;

public class SimpleOrder : KeyedCollection<int, OrderItem>
{
    //Konstruktor bezparametryowy klasy bazowej tworzy
    // kolekcję z wewnętrznym słownikiem. W tym przykładzie
    // nie zostały zaprezentowane inne konstruktory
```

```
public SimpleOrder() : base() {}

//To jest jedyna metoda, która musi zostać przesłonięta, ponieważ
//bez tego kolekcja nie mogłaby wydobyć kluczy
protected override int GetKeyForItem(OrderItem item)
{
    //W tym przykładzie kluczem jest numer partii produkcyjnej
    return item.PartNumber;
}

public class Demo
{
    public static void Main()
    {
        SimpleOrder weekly = new SimpleOrder();

        //Metoda Add dziedzicząca z Collection pobiera OrderItem
        weekly.Add(new OrderItem(110072674, "Widget", 400, 45.17));
        weekly.Add(new OrderItem(110072675, "Sprocket", 27, 5.3));
        weekly.Add(new OrderItem(101030411, "Motor", 10, 237.5));
        weekly.Add(new OrderItem(110072684, "Gear", 175, 5.17));

        Display(weekly);

        //Metoda Contains pobiera klucz, w tym przypadku jest to typ int
        Console.WriteLine("\nContains(101030411): {0}",
            weekly.Contains(101030411));

        //Domyślana właściwość Item pobiera klucz
        Console.WriteLine("\nweekly[101030411].Description: {0}",
            weekly[101030411].Description);

        //Metoda Remove pobiera klucz
        Console.WriteLine("\nRemove(101030411)");
        weekly.Remove(101030411);
        Display(weekly);

        //Metoda Insert dziedziczy z Collection i pobiera indeks oraz OrderItem
        Console.WriteLine("\nInsert(2, New OrderItem(...))");
        weekly.Insert(2, new OrderItem(111033401, "Nut", 10, .5));
        Display(weekly);

        Collection<OrderItem> coweekly = weekly;
        Console.WriteLine("\ncoweekly[2].Description: {0}",
            coweekly[2].Description);

        Console.WriteLine("\ncoweekly[2] = new OrderItem(...)");
        coweekly[2] = new OrderItem(127700026, "Crank", 27, 5.98);

        OrderItem temp = coweekly[2];

        //Metoda IndexOf dziedzicząca z Collection<OrderItem> pobiera OrderItem zamiast klucza
        Console.WriteLine("\nIndexOf(temp): {0}", weekly.IndexOf(temp));

        //Metoda Remove również pobiera OrderItem
        Console.WriteLine("\nRemove(temp)");
        weekly.Remove(temp);
```

```
        Display(weekly);

        Console.WriteLine("\nRemoveAt(0)");
        weekly.RemoveAt(0);
        Display(weekly);

    }

    private static void Display(SimpleOrder order)
    {
        Console.WriteLine();
        foreach( OrderItem item in order )
        {
            Console.WriteLine(item);
        }
    }
}

//klasa reprezentująca prostą listę zamówień
public class OrderItem
{
    public readonly int PartNumber;
    public readonly string Description;
    public readonly double UnitPrice;

    private int _quantity = 0;

    public OrderItem(int partNumber, string description,
                    int quantity, double unitPrice)
    {
        this.PartNumber = partNumber;
        this.Description = description;
        this.Quantity = quantity;
        this.UnitPrice = unitPrice;
    }

    public int Quantity
    {
        get { return _quantity; }
        set
        {
            if (value<0)
                throw new ArgumentException("Ilość nie może być ujemna.");
            _quantity = value;
        }
    }

    public override string ToString()
    {
        return String.Format(
            "{0.9} {1.6} {2,-12} at {3.8:#.###.00} = {4.10:###.##.00}",
            PartNumber, _quantity, Description, UnitPrice,
            UnitPrice * _quantity);
    }
}
```

## Klasa generyczna SortedList

Reprezentuje kolekcję par klucz/wartość posortowaną według klucza i bazującą na implementacji IComparer.

Składnia:

```
[SerializableAttribute]
[ComVisibleAttribute(false)]
public class SortedList<TKey, TValue> : IDictionary<TKey, TValue>,
ICollection<KeyValuePair<TKey, TValue>>,
IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable
```

Przykładowy kod:

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

class Program
{
    static void Main()
    {
        //utworzenie nowej posortowanej listy napisów z kluczami
        //((również jako napisami)
        SortedList<string, string> openWith =
            new SortedList<string, string>();

        //dodanie kilku elementów do listy
        //Dla jednego klucza może być kilka wartości
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        //Metoda Add rzuca wyjątek, jeżeli klucz już istnieje
        try
        {
            openWith.Add("txt", "winword.exe");
        }
        catch (ArgumentException)
        {
            Console.WriteLine("Element o kluczu = \"txt\" już istnieje.");
        }

        //dostęp do elementu poprzez indeks
        Console.WriteLine("Dla klucza = \"rtf\", wartość = {0}.,",
            openWith["rtf"]);

        //zmiana wartości
        openWith["rtf"] = "winword.exe";
        Console.WriteLine("Dla klucza = \"rtf\", wartość = {0}.,",
            openWith["rtf"]);

        //Jeżeli klucz nie istnieje, zostanie utworzony
        openWith["doc"] = "winword.exe";

        //Jeżeli klucz nie istnieje, rzucony zostanie wyjątek
    }
}
```

```
try
{
    Console.WriteLine("Dla klucza = \"tif\", wartość = {0}.",
                      openWith["tif"]);
}
catch (KeyNotFoundException)
{
    Console.WriteLine("Klucz = \"tif\" nie istnieje.");
}

//użycie TryGetValue w celu sprawdzenia, czy para klucz/wartość istnieje
string value = "";
if (openWith.TryGetValue("tif", out value))
{
    Console.WriteLine("Dla klucza = \"tif\", wartość = {0}.", value);
}
else
{
    Console.WriteLine("Klucz = \"tif\" nie istnieje.");
}

//użycie ContainsKey przed dodaniem nowych kluczów
if (!openWith.ContainsKey("ht"))
{
    openWith.Add("ht", "hypertrm.exe");
    Console.WriteLine("Dodano wartość dla klucza = \"ht\": {0}",
                     openWith["ht"]);
}

//Przy dostępie do elementów poprzez pętlę foreach
//zwracane są one jako obiekty KeyValuePair
Console.WriteLine();
foreach (KeyValuePair<string, string> kvp in openWith)
{
    Console.WriteLine("Klucz = {0}, Wartość = {1}",
                     kvp.Key, kvp.Value);
}

//Do pobrania samych wartości użyj właściwości Values
IList<string> ilistValues = openWith.Values;

//Elementy listy są silnie typowane
Console.WriteLine();
foreach (string s in ilistValues)
{
    Console.WriteLine("Wartość = {0}", s);
}

//Właściwość Values jest przydatna do pobierania wartości
//poprzez indeks
Console.WriteLine("\nPobieranie wartości poprzez Values " +
                  "wartość: Values[2] = {0}", openWith.Values[2]);

//Do pobrania jedynie kluczy użyj właściwości Keys
IList<string> ilistKeys = openWith.Keys;

//Elementy listy są silnie typowane
Console.WriteLine();
```

```
foreach (string s in iListKeys)
{
    Console.WriteLine("Klucz = {0}", s);
}

//Właściwość Keys jest przydatna do pobierania kluczy
//poprzez indeks
Console.WriteLine("\nPobieranie wartości poprzez Keys " +
    "właściwość: Keys[2] = {0}", openWith.Keys[2]);

//Metoda Remove usuwa parę klucz/wartość
Console.WriteLine("\nRemove(\"doc\")");
openWith.Remove("doc");

if (!openWith.ContainsKey("doc"))
{
    Console.WriteLine("Klucz \"doc\" nie istnieje.");
}
}
```

## 4.21. Kontra i kowariancja

Gdy przypiszesz metodę do delegatu, kowariancja i kontrawariancja dostarczą elastyczności przy sprawdzaniu typu delegatu z sygnaturą metody. Kowariancja pozwala metodzie mieć typ zwracany bardziej pochodny niż ten zdefiniowany w delegacie. Kontrawariancja pozwala metodzie mieć typy parametrów mniej pochodne od tych w delegacie.

Poniższy przykład demonstruje, jak delegaty mogą być użyte z metodami, które mają typy zwracane pochodne z typu zwracanego w sygnaturze delegatu. Typ danych zwracany przez DogsHandler jest typu Dogs, który pochodzi z typu Mammals zdefiniowanego w delegacie.

```
class Mammals{}
class Dogs : Mammals{};

class Program
{
    //definicja delegatu
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }
}
```

```
static void Test()
{
    HandlerMethod handlerMammals = MammalsHandler;

    //Kowariancja pozwala na takie przypisanie
    HandlerMethod handlerDogs = DogsHandler;
}
```

Teraz kolejny przykład. Poniższy kod prezentuje, jak delegaty mogą być użyte z metodami mającymi parametry typu, który jest typem bazowym sygnatury delegatu. Kontrawariancja pozwala na użycie jednego obsługującego zdarzenia zamiast osobnych.

```
//obsługujący zdarzenia, który akceptuje parametry typu EventArgs
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();

    //Możesz użyć metody, która przyjmuje parametr typu EventArgs
    this.button1.KeyDown += this.MultiHandler;

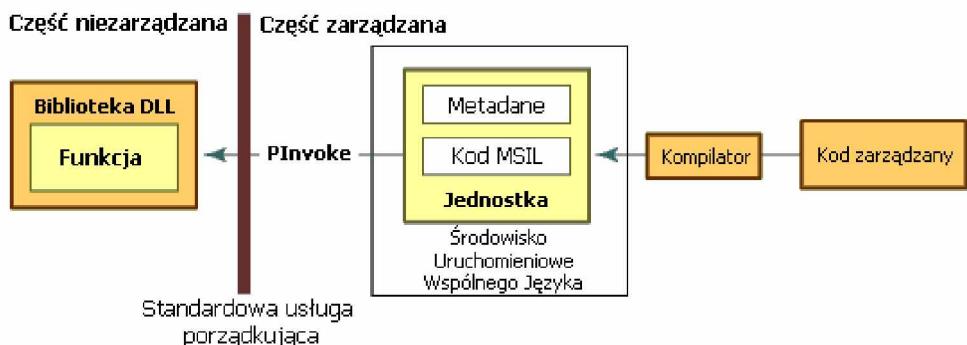
    //Możesz użyć tej samej obsługi zdarzenia
    this.button1.MouseClick += this.MultiHandler;
}
```

## Rozdział 5.

# Język C#. Pozostałe zagadnienia

## 5.1. Wywoływanie funkcji przez PInvoke

PInvoke to skrót od *Platform Invocation Services*. Uslugi 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.



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")] //zimportowanie biblioteki DLL
    public static extern int puts(string c); //deklaracja funkcji z biblioteki
    [DllImport("msvcrt.dll")] //zimportowanie 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 wnętrza, 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 *niezmienne*: 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 odśmiecač 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 tworzysz 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);

//wyświetlanie 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 postanowilem 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, kopowania, usuwania, przenoszenia i otwierania plików.
- ◆ **FileInfo** — dostarcza metody instancji do tworzenia, kopowania, 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 kopować 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 UAEEx)
        {
            Console.WriteLine(UAEEx.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.WriteLine(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 rejestrzu?

```
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 tworzymy, 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("IExplore.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 wykonuje 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 iinstancję 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 oczekивание 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.

Caly 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);

        //Uspij 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(); //Rzuć 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) //Zlap wyjątek, gdy wystąpił  
        {  
            Console.WriteLine("Próbowiesz 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`.



## Rozdział 6.

# Tworzenie interfejsu graficznego aplikacji

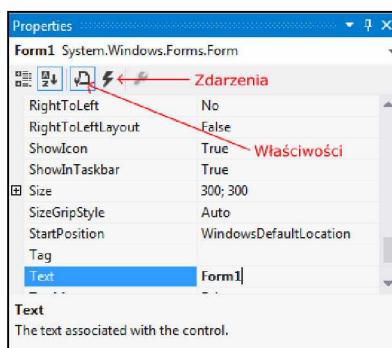
## 6.1. Projektowanie interfejsu graficznego

Uruchom środowisko Microsoft Visual Studio 2012 Ultimate, wybierz z górnego menu *FILE/New/Project...* i zaznacz *Windows Forms Application*, a następnie kliknij *OK*.

Z lewej strony masz wysuwane okienko *Toolbox*, które zawiera kontrolki. Aby wstawić kontrolkę do formularza, kliknij ją, a następnie kliknij miejsce w oknie, gdzie ma się ona znajdująć, i rozciągnij ją według gustu.

Po zaznaczeniu okna głównego lub kontrolki (nazywanej też oknem potomnym) z prawej strony pojawi się okno dokowane *Properties*, które zawiera właściwości zaznaczonego elementu. Okno główne i kontrolki oprócz właściwości mają też zdarzenia. Na widok zdarzeń przełączamy się, klikając w oknie *Properties* ikonę z piorunem (rysunek 6.1).

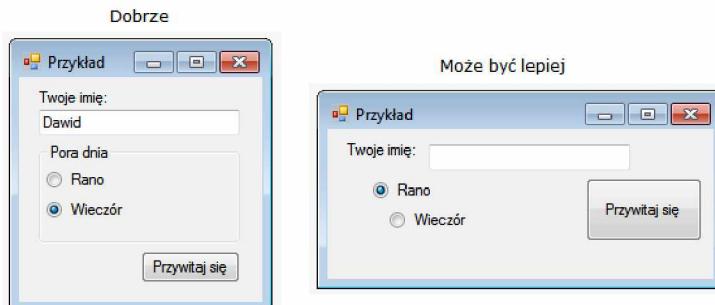
**Rysunek 6.1.**  
Okno dokowane  
z właściwościami  
zaznaczonego elementu



Jeżeli chodzi o projektowanie interfejsu użytkownika, to należy tak układać elementy, aby interfejs był jak najbardziej przejrzysty i trzymał się zasad, jakie występują w większości aplikacji. Na rysunku 6.2 możesz zobaczyć dla porównania interfejs dobrze zaprojektowany (lewa strona rysunku) i interfejs źle zaprojektowany (prawa strona rysunku).

#### Rysunek 6.2.

Dobrze (lewa strona rysunku) i niezbyt dobrze (prawa strona rysunku) zaprojektowany interfejs



## 6.2. Wejście klawiatury

Gdy naciskamy przycisk na klawiaturze i okno naszej aplikacji jest aktywne, to wywoływane jest zdarzenie o nazwie `KeyPress`. Za chwilę dowiesz się, jak użyć tego zdarzenia, aby rozpoznać wciśnięty klawisz i wykonać wtedy odpowiednią akcję.

Wstaw do formularza kontrolkę `TextBox` (z okna *Toolbox* zawierającego dostępne kontrole). Teraz w oknie *Properties* przejdź do widoku zdarzeń (*Events*) i kliknij pole tekstowe obok zdarzenia o nazwie `KeyPress`. Utworzona zostanie metoda o nazwie `textBox1_KeyPress` i od razu zostaniesz przeniesiony do edycji kodu tej nowo wygenerowanej metody.

Załóżmy, że chcemy, aby do kontrolki można było wpisywać tylko liczby w systemie binarnym. Dlatego musimy przefiltrować wprowadzane znaki i odrzucać wszystkie znaki oprócz znaku 0 i 1.

Tak wygląda kod wykonujący powyżej opisaną czynność:

```
if (e.KeyChar != '0' && e.KeyChar != '1') //gdy znak różny od '0' i '1'  
{  
    e.Handled = true; //Zdarzenie zostało obsłużone  
}
```

Powyższy kod sprawdza, czy wciśnięty znak to '0' lub '1'. Jeżeli tak, to wykonuje się domyślna obsługa zdarzenia (znak zostaje wpisany do kontrolki `TextBox`). Jeśli jednak znak jest inny niż '0' lub '1', to sami obsługujemy zdarzenie (tak naprawdę to nic nie robimy) i ustawiamy właściwość `Handled` na `true` (czyli informujemy, że sami obsłużyliśmy zdarzenie).

Zdarzenia dotyczące klawiatury, jakie mogą zostać wywołane, przedstawia tabela 6.1.

**Tabela 6.1.** Zdarzenia dotyczące klawiatury

Nazwa zdarzenia	Opis
KeyDown	Zdarzenie jest wywoływanie, gdy użytkownik wcisnie klawisz.
KeyPress	Zdarzenie jest wywoływanie, gdy użytkownik wcisnie klawisz lub klawisze i rezultatem tego jest znak (np. wcisnięcie klawisza A na klawiaturze).
KeyUp	Zdarzenie jest wywoływanie, gdy użytkownik puszcza klawisz.

## 6.3. Wejście myszy

Zarówno klawiatura, jak i mysz wywołują zdarzenia. Przykładowym zdarzeniem może być podwójne kliknięcie lewym przyciskiem myszy. Zdarzenia wywoływane przez mysz przedstawia tabela 6.2.

**Tabela 6.2.** Zdarzenia dotyczące myszy

Nazwa zdarzenia	Opis
Click	Zdarzenie wywoływanie, gdy przycisk myszy zostaje zwolniony. Przeważnie przed zdarzeniem MouseUp.
MouseClick	Zdarzenie wywoływanie, gdy użytkownik kliknie myszą kontrolkę.
DoubleClick	Zdarzenie wywoływanie, gdy użytkownik podwójnie kliknie kontrolkę.
MouseDoubleClick	Zdarzenie wywoływanie, gdy użytkownik podwójnie kliknie kontrolkę myszą.
MouseDown	Zdarzenie wywoływanie, gdy kursor myszy znajduje się nad kontrolką i jest wcisnięty.
MouseEnter	Zdarzenie wywoływanie, gdy kursor myszy wchodzi na obszar kontrolki.
MouseHover	Zdarzenie wywoływanie, gdy kursor myszy zatrzyma się nad kontrolką.
MouseLeave	Zdarzenie wywoływanie, gdy kursor myszy opuszcza obszar kontrolki.
MouseMove	Zdarzenie wywoływanie, gdy kursor myszy porusza się po obszarze kontrolki.
MouseUp	Zdarzenie wywoływanie, gdy kursor myszy jest nad kontrolką i przycisk myszy zostaje zwolniony.
MouseWheel	Zdarzenie wywoływanie, jeśli użytkownik przekręci kółko myszki, gdy kontrolka jest aktywna.

## 6.4. Symulowanie klawiatury i myszy

Akcje wykonywane przez klawiaturę i mysz możemy symulować. Właśnie to zagadnienie zostanie zaraz przedstawione.

## Symulowanie klawiatury

Załóżmy, że chcemy zautomatyzować operację arytmetyczną wykonywaną przez uruchomiony kalkulator systemu Windows.



Być może trzeba będzie zmienić parametry funkcji FindWindow, gdyż aplikacja kalkulatora może się różnić zależnie od wersji systemu Windows. Poniższy kod został przetestowany na systemie Windows 7.

Oto przykład:

```
//funkcja pobierająca uchwyt do okna aplikacji
[DllImport("USER32.DLL", CharSet = CharSet.Unicode)]
public static extern IntPtr FindWindow(string lpClassName,
    string lpWindowName);

//funkcja ustawiająca określone okno na pierwszy plan
[DllImport("USER32.DLL")]
public static extern bool SetForegroundWindow(IntPtr hWnd);

private void button1_Click(object sender, EventArgs e)
{
    //Pobierz uchwyt do aplikacji kalkulatora
    //Klasę i nazwę okna pomógł określić program Spy++
    IntPtr calculatorHandle = FindWindow("CalcFrame", "Calculator");

    //Sprawdź, czy kalkulator jest uruchomiony
    if (calculatorHandle == IntPtr.Zero)
    {
        MessageBox.Show("Calculator is not running.");
        return;
    }

    //Ustaw aplikację kalkulatora na pierwszy plan i wyslij do niej klawisze
    SetForegroundWindow(calculatorHandle);
    SendKeys.SendWait("111");
    SendKeys.SendWait("*");
    SendKeys.SendWait("11");
    SendKeys.SendWait "=";
}
```

## Symulowanie myszy

Najlepszym sposobem symulowania zdarzeń myszy jest wywołanie metody `OnEventName`. Opcja ta jest jednak przeważnie możliwa tylko przy własnych kontrolkach i formularzach, gdyż metody wywołujące zdarzenia są chronione.

## 6.5. Przeciągnij i upuść

Technika ta polega na tym, aby użytkownik mógł przeciągać elementy jednej kontolki do drugiej lub przeciągać pliki do kontrolki.

Aby kontrolka mogła przyjmować przeciagane elementy, należy ustawić jej właściwość AllowDrop na true.i użyć jednego lub więcej następujących zdarzeń:

- ◆ DragEnter — użytkownik przeciąga obiekt na obszar kontrolki, ale nie upuścił go jeszcze.
- ◆ DragOver — obiekt jest przenoszony na obszar kontrolki.
- ◆ DragDrop — użytkownik przeciągnął obiekt na obszar kontrolki i upuścił go.
- ◆ DragLeave — obiekt przeciągany na obszar kontrolki jest przeniesiony z powrotem.

Wstaw teraz do formularza kontrolkę o nazwie ListBox i oprogramuj jej zdarzenia DragEnter oraz DragDrop następująco:

```
private void listBox1_DragEnter(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop, false) == true)
    {
        e.Effect = DragDropEffects.All;
    }
}

private void listBox1_DragDrop(object sender, DragEventArgs e)
{
    string[] files = (string[])e.Data.GetData(DataFormats.FileDrop);

    foreach (string file in files)
    {
        listBox1.Items.Add(file);
    }
}
```

Od tej pory bez problemu możesz przeciągać pliki na kontrolkę, która po przeciągnięciu na nią pliku wyświetli ścieżkę do pliku i nazwę pliku.

## 6.6. Przegląd wybranych kontrolek

Poniżej znajduje się lista kontrolek dostępnych w Windows Forms oraz krótki opis ich zastosowania.

- ◆ BackgroundWorker — pozwala formularzowi lub kontrolce wykonywać operacje asynchronicznie.

- ◆ **BindingNavigator** — dostarcza nawigację i manipulację interfejsem użytkownika dla kontrolek związanych z danymi.
- ◆ **Button** — reprezentuje przycisk, który użytkownik może wcisnąć, aby wykonać określoną akcję.
- ◆ **CheckBox** — reprezentuje pole wielokrotnego wyboru.
- ◆ **CheckedListBox** — wyświetla listę elementów, które mają dołączone pola wielokrotnego wyboru.
- ◆ **ColorDialog** — pozwala użytkownikowi wybrać kolor z palety kolorów i dodać własny kolor do palety.
- ◆ **ComboBox** — lista rozwijana.
- ◆ **DataGridView** — pozwala wyświetlać w przejrzysty sposób dane tabelaryczne.
- ◆ **DateTimePicker** — pozwala wybrać użytkownikowi datę i czas z listy.
- ◆ **DomainUpDown** — pozwala wybrać użytkownikowi element z listy z przyciskami w góre i w dół.
- ◆ **FolderBrowserDialog** — wyświetla okno dialogowe wyboru katalogu lub utworzenia go.
- ◆ **FontDialog** — wyświetla czcionki zainstalowane w systemie.
- ◆ **GroupBox** — pozwala łączyć kontrolki w grupy.
- ◆ **Label** — wyświetla tekst, który nie może być edytowany przez użytkownika (tzw. etykieta).
- ◆ **LinkLabel** — etykieta z hiperłączem.
- ◆ **ListBox** — pozwala użytkownikowi wybrać element z predefiniowanej listy.
- ◆ **ListView** — wyświetla elementy z ikonami.
- ◆ **MaskedTextBox** — pozwala na formatowane wprowadzanie danych.
- ◆ **NumericUpDown** — pozwala użytkownikowi wybrać liczbę (wartość można zwiększać i zmniejszać przyciskami góra/dół).
- ◆ **OpenFileDialog** — wyświetla okno wyboru pliku do otwarcia.
- ◆ **Panel** — pozwala grupować kontrolki. Wspiera przewijanie.
- ◆ **PrintDialog** — okno z ustawieniami drukowania.
- ◆ **ProgressBar** — wyświetla postęp określonej operacji w postaci paska.
- ◆ **RadioButton** — pole jednokrotnego wyboru.
- ◆ **RichTextBox** — do operacji na tekście ze wsparciem dla formatowania.
- ◆ **SaveFileDialog** — okno zapisu pliku.
- ◆ **SoundPlayer** — pozwala odtwarzać dźwięk.

- ♦ `TabControl` — zakładki, które mogą zawierać inne kontrolki.
- ♦ `TextBox` — pozwala edytować tekst, obsługa wielu linijek.
- ♦ `Timer` — wywołuje zdarzenie w określonych odstępach czasowych.
- ♦ `ToolTip` — wyświetla podpowiedź.
- ♦ `TrackBar` — suwak pozwalający wybrać wartość liczbową.
- ♦ `TreeView` — kontrolka widoku drzewa.
- ♦ `WebBrowser` — przeglądarka internetowa.

## 6.7. Wstęp do Windows Presentation Foundation

*Windows Presentation Foundation* jest systemem prezentacyjnym następnej generacji. Jądro tego systemu jest niezależne od rozdzielczości oraz oparte na silniku renderującym bazowanym na wektorach.

Cechy (ang. *features*) systemu WPF:

- ♦ Język XAML
- ♦ Kontrolki
- ♦ Bindowanie danych
- ♦ Układ (ang. *layout*)
- ♦ Grafika 2D i 3D
- ♦ Animacja
- ♦ Style
- ♦ Szablony
- ♦ Dokumenty
- ♦ Media
- ♦ Typografia

System WPF jest włączony do Microsoft .NET Framework.

### Tworzenie projektu WPF

Uruchom Microsoft Visual Studio 2012 Ultimate i z górnego menu wybierz *FILE/New Project...*, zaznacz *WPF Application* i kliknij *OK*.

W oknie dokowanym Solution Explorer z prawej strony znajdują się pliki, jakie zawiera projekt. Pliki z rozszerzeniem XAML możesz edytować wizualnie, klikając prawym przyciskiem myszy nazwę pliku i wybierając *View Designer*. Natomiast pliki o rozszerzeniu CS to pliki z kodem programu w języku C#.

## Przykład: „Witaj, świecie WPF!”

*MainWindow.xaml* zawiera kod:

```
<Window x:Class="WpfApplication1.MainWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Witaj, świecie WPF!" Height="350" Width="525">
    <Grid>
        <Button Content="Witaj!" Height="20" HorizontalAlignment="Left"
               Margin="58,65,0,0" Name="button1" VerticalAlignment="Top" Width="50"
               Click="button1_Click" />
        <TextBox Height="24" HorizontalAlignment="Left" Margin="27,35,0,0"
                 Name="textBox1" VerticalAlignment="Top" Width="81" />
        <Label Content="Twoje imię:" Height="25" HorizontalAlignment="Left"
               Margin="22,12,0,0" Name="label1" VerticalAlignment="Top" Width="66"
               ContentStringFormat="" />
    </Grid>
</Window>
```

*MainWindow.xaml.cs* zawiera kod:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace WpfApplication1
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, RoutedEventArgs e)
```

```
        MessageBox.Show("Witaj, " + textBox1.Text + "!");
    }
}
```

Rezultat działania aplikacji przedstawia rysunek 6.3.

**Rysunek 6.3.**

Program „*Witaj świecie,  
WPF!*”. Rezultat działania





## Rozdział 7.

# Podstawy programowania sieciowego

## 7.1. System DNS

Rozwinięcie skrótu *DNS* to *Domain Name System* (po polsku oznacza to System Nazw Domenowych). Jest to system zamieniający adresy internetowe zrozumiale dla użytkowników na adresy IP zrozumiałe dla urządzeń tworzących sieć komputerową.

Klasa o nazwie *DNS* znajduje się w przestrzeni nazw *System.Net*.

Oto prosty przykład zamieniający adres URL na adres IP:

```
using System;
using System.Net;

class Program
{
    static void Main()
    {
        string helion = "www.helion.pl"; //zmienna string z adresem URL
        //Pobierz adres IP
        IPAddress[] addresslist = Dns.GetHostAddresses(helion);

        foreach (IPAddress theaddress in addresslist)
        {
            //Wyświetl adres(y) IP na ekranie
            Console.WriteLine(theaddress.ToString());
        }
    }
}
```

## 7.2. Wysyłanie wiadomości e-mail

Najpierw kod, którego zadaniem jest wysłać wiadomość na podany adres e-mail:

```
using System;
using System.Net;
using System.Net.Mail;

class Program
{
    static void Main()
    {
        SmtpClient smtpClient = new SmtpClient();
        NetworkCredential basicCredential = new NetworkCredential();
        MailMessage message = new MailMessage();
        MailAddress fromAddress = new MailAddress("nadawca@serwer.com");
        int error = 0;

        basicCredential.UserName = " użytkownik@serwer.com";
        basicCredential.Password = "hasło";

        smtpClient.Host = "smtp.serwer.com";
        smtpClient.UseDefaultCredentials = false;
        smtpClient.Credentials = basicCredential;
        smtpClient.EnableSsl = true; //true, gdy serwer SMTP wymaga szyfrowania

        message.From = fromAddress;
        message.Subject = "Temat wiadomości";

        message.IsBodyHtml = true; //true, gdy wiadomość w formacie HTML
        message.Body = "<strong>Treść wiadomości</strong>";
        message.To.Add("odbiorca@serwer.com");

        try
        {
            smtpClient.Send(message);
        }
        catch
        {
            error = 1;
            Console.WriteLine("Wystąpił błąd.");
        }
        finally
        {
            if(error == 0)
                Console.WriteLine("Wiadomość została wysłana.");
        }
    }
}
```

W powyższym kodzie używamy przestrzeni nazw `System.Net` i `System.Net.Mail`. Klasa `NetworkCredential` reprezentuje poświadczenia takie jak użytkownik i hasło. Klasa `MailMessage` reprezentuje wiadomość, którą wysyłamy. A wysyłaniem wiadomości e-mail zajmuje się obiekt klasy `SmtpClient`.

## 7.3. Protokół FTP

Protokoł FTP służy między innymi do przesyłania plików na serwer, aby udostępnić określone pliki innym użytkownikom internetu.

### Przykład: Jak wysłać plik na serwer FTP?

```
using System;
using System.Net;
using System.IO;

class Program
{
    static void Main()
    {
        FtpWebRequest ftp = (FtpWebRequest)WebRequest.
            Create("ftp://ftp.serwer.com/www/plik.txt");
        ftp.Credentials = new NetworkCredential("użytkownik", "hasło");
        ftp.KeepAlive = true;
        ftp.UseBinary = true;
        ftp.Method = WebRequestMethods.Ftp.UploadFile;
        FileStream fs = File.OpenRead(@"C:\plik.txt"); //ściezka lokalna
        byte[] buffer = new byte[fs.Length];
        fs.Read(buffer, 0, buffer.Length);
        fs.Close();
        Stream ftpstream = ftp.GetRequestStream();
        ftpstream.Write(buffer, 0, buffer.Length);
        ftpstream.Close();
    }
}
```

## 7.4. Gniazda (ang. Sockets)

W tym rozdziale napiszemy prostą aplikację klient-serwer opartą na protokole TCP. Zadaniem klienta będzie wysłanie przez sieć wiadomości do serwera. Natomiast serwer po otrzymaniu wiadomości o treści "cd" otworzy i zamknie napęd CD-ROM oraz wyśle potwierdzenie do klienta.

Kod serwera (aplikacja konsolowa):

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Runtime.InteropServices;

class Program
{
    [DllImport( "winmm.dll", EntryPoint="mciSendStringA",
    CharSet=CharSet.Ansi )]
```

```
protected static extern int mciSendString(string lpstrCommand,
    StringBuilder
    lpstrReturnString,
    int uReturnLength,
    IntPtr hwndCallback );

static void Main()
{
    IPAddress localAddr = IPAddress.Parse("127.0.0.1");
    TcpListener serverSocket = new TcpListener(localAddr, 8888);
    TcpClient clientSocket = default(TcpClient);
    serverSocket.Start();
    Console.WriteLine(" >> Serwer uruchomiony");
    clientSocket = serverSocket.AcceptTcpClient();
    Console.WriteLine(" >> Połaczenie od klienta zaakceptowane");

    while ((true))
    {
        try
        {
            NetworkStream networkStream = clientSocket.GetStream();
            byte[] bytesFrom = new byte[10025];
            networkStream.Read(bytesFrom, 0, (int)clientSocket.ReceiveBufferSize);
            string dataFromClient = System.Text.Encoding.ASCII.GetString(bytesFrom);
            dataFromClient = dataFromClient.Substring(0, dataFromClient.IndexOf("$"));
            if (dataFromClient == "cd")
            {
                mciSendString("set cdaudio door open", null, 0, IntPtr.Zero);
                mciSendString("set cdaudio door closed", null, 0, IntPtr.Zero);
            }
            Console.WriteLine(" >> Polecenie od klienta - " + dataFromClient);
            string serverResponse = "Wysunięto taczkę napędu CD.";
            Byte[] sendBytes = Encoding.ASCII.GetBytes(serverResponse);
            networkStream.Write(sendBytes, 0, sendBytes.Length);
            networkStream.Flush();
        }
        catch
        {
            break;
        }
    }

    clientSocket.Close();
    serverSocket.Stop();
    Console.WriteLine(" >> wyjście");
    Console.ReadLine();
}
}
```

Kod klienta (aplikacja Windows Forms z przyciskiem button1 i etykietą label1):

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
```

```
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        System.Net.Sockets.TcpClient clientSocket = new System.Net.Sockets.TcpClient();

        public Form1()
        {
            InitializeComponent();
        }

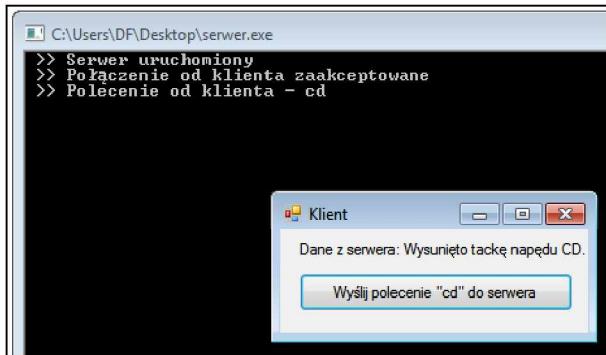
        private void button1_Click(object sender, EventArgs e)
        {
            NetworkStream serverStream = clientSocket.GetStream();
            byte[] outStream = System.Text.Encoding.ASCII.GetBytes("cd$");
            serverStream.Write(outStream, 0, outStream.Length);
            serverStream.Flush();

            byte[] inStream = new byte[10025];
            serverStream.Read(inStream, 0, (int)clientSocket.ReceiveBufferSize);
            string returndata = System.Text.Encoding.ASCII.GetString(inStream);
            label1.Text = "Dane z serwera: " + returndata;
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            int error = 0;
            try
            {
                clientSocket.Connect("127.0.0.1", 8888);
            }
            catch
            {
                error = 1;
                label1.Text = "Nie mogę się połączyć z serwerem.";
            }
            finally
            {
                if (error == 0)
                    label1.Text = "Połączono z serwerem.";
            }
        }
    }
}
```

Program podczas działania przedstawia rysunek 7.1.

**Rysunek 7.1.**  
*Aplikacja klient-serwer  
podczas działania*



# Rozdział 8.

# Asembler IL

## 8.1. Co to jest?

*Common Intermediate Language* (z ang. Wspólny Język Pośredni, w skrócie CIL lub IL) jest to język najniższego poziomu dla platformy .NET, który może zostać odczytany i zrozumiany przez człowieka. Można go nazwać Asemblerem platformy Microsoft .NET. Kod CIL jest tłumaczony bezpośrednio na kod bajtowy i wykonywany przez maszynę wirtualną. Język ten jest obiektowy (wsparia klasy, dziedziczenie, polymorfizm, obsługę błędów) oraz w całości oparty na stosie.

Kompilator Asemblera IL (*ilasm.exe*) jest dołączony do środowiska Microsoft Visual Studio. Plikom źródłowym z kodem w języku CIL przyjęto nadawać rozszerzenie *\*.il*.

## 8.2. Program „Witaj, świecie!”

Program „Witaj, świecie!” prezentuje się następująco:

```
.assembly extern mscorlib{}

.assembly HelloWorld
{
    .ver 1:0:1:0
}
.method static void main()
{
    .entrypoint
    .maxstack 1

    ldstr "Witaj, świecie!"
    call void [mscorlib]System.Console::WriteLine(string)

    ret
}
```

Dyrektywa `.entrypoint` określa, od jakiej metody rozpocznie się wykonywanie programu. Natomiast `.maxstack` określa, że metoda będzie używała tylko jednego miejsca na stosie. Każda metoda CIL musi zadeklarować, ile miejsc na stosie będzie używała. W platformie .NET jednostka jest modelem wykonywalnym lub biblioteką. Pierwsza dyrektywa `.assembly` odwołuje się do zewnętrznej jednostki, biblioteki `mscorlib`, która zawiera między innymi metody do operacji wejścia/wyjścia. Druga dyrektywa `.assembly` określa nazwę jednostki, której moduł będzie częścią, oraz numer wersji (`.ver`). Instrukcja `ldstr` odkłada na stos referencję do napisu "Witaj, świecie!", a instrukcja `call` wywołuje metodę `WriteLine` wyświetlającą napis odłożony na stos na standardowym wyjściu.

## 8.3. Kompilacja i uruchamianie

W swoim systemie kliknij menu *Start*, następnie *Wszystkie programy*, dalej wybierz *Microsoft Visual Studio 2012/Visual Studio Tools* i uruchom narzędzie *Developer Command Prompt for VS2012*.

Teraz za pomocą polecenia `cd` przejdź do katalogu z kodem źródłowym programu napisanym w Asemblerze IL (plik źródłowy powinien mieć rozszerzenie `*.il`).

```
cd C:\ilasm
```

Program komplujemy za pomocą polecenia:

```
ilasm hello.il
```

A uruchamiamy za pomocą polecenia:

```
hello.exe
```



Zauważ, że program „Witaj, świecie!” w Asemblerze IL zajmuje jedynie 2 kilobajty (zależnie od systemu rozmiar może się nieznacznie zmienić).

## 8.4. Zmienne lokalne

Popatrz na poniższy program:

```
.assembly extern mscorel{  
}  
.assembly sumprog  
{  
    .ver 1:0:1:0  
}  
.method static void main()
```

```
{  
    .entrypoint  
    .maxstack 4  
    .locals init (int32 first, int32 second)  
  
    ldstr "First number: "  
    call void [mscorlib]System.Console::Write(string)  
    call string [mscorlib]System.Console::ReadLine()  
    call int32 [mscorlib]System.Int32::Parse(string)  
  
    ldstr "Second number: "  
    call void [mscorlib]System.Console::Write(string)  
    call string [mscorlib]System.Console::ReadLine()  
    call int32 [mscorlib]System.Int32::Parse(string)  
  
    add  
    call void [mscorlib]System.Console::Write(int32)  
  
    ret  
}
```

Zadaniem tego programu jest pobranie od użytkownika dwóch liczb i wyświetlenie ich sumy. Dyrektywa `.maxstack` z wartością 4 informuje, że będą użyte 4 miejsca na stosie (dwa miejsca na napisy i dwa na zmienne lokalne). Dalej mamy deklaracje dwóch zmiennych lokalnych (po dyrektywie `.locals`). Potem za pomocą instrukcji `ldstr` odkładamy na stos referencję do napisu. Następnie wywołujemy metodę `Write`, która zdejmuję ze stosu napis i wyświetla go na ekranie. Potem wywołujemy metodę `ReadLine`, która pobiera napis od użytkownika i odkłada go na stos. Następna jest metoda `Parse`, która pobiera napis ze stosu, zamienia go na liczbę i odkłada na stos. Analogicznie jest przy pobieraniu drugiej wartości. Na końcu mamy instrukcję `add`, która dodaje dwie liczby ze stosu, odkłada wynik na stosie, a metoda `Write` wyświetla ten wynik na ekranie.

Przykładowe wyjście programu:

```
C:\ilasm>sumprog  
Pierwsza liczba: 5  
Druga liczba: 2  
7  
C:\ilasm>
```

## 8.5. Metody

Poniżej prosta metoda, która pobiera dwie liczby całkowite jako parametry i zwraca sumę tych wartości.

```
.method static int32 sum(int32, int32)  
{  
    .maxstack 2  
    ldarg.0  
    ldarg.1
```

```
    add
    ret
}
```

Odkłada ona na stos dwie wartości i sumuje je za pomocą instrukcji add, a następnie wraca do miejsca wywołania za pomocą instrukcji ret.

Poniżej cały program korzystający z metody sum:

```
.assembly extern mscorel{ }

.assembly MethodsExample
{
    .ver 1:0:1:0

.method static void main()
{
    .entrypoint
    .maxstack 4

    .locals init (int32 first,
                  int32 second)

    ldstr "First number: "
    call void [mscorel]System.Console::Write(string)
    call string [mscorel]System.Console::ReadLine()
    call int32 [mscorel]System.Int32::Parse(string)

    ldstr "Second number: "
    call void [mscorel]System.Console::Write(string)
    call string [mscorel]System.Console::ReadLine()
    call int32 [mscorel]System.Int32::Parse(string)

    call int32 sum(int32, int32)
    call void [mscorel]System.Console::Write(int32)

    ret
}

.method static int32 sum(int32, int32)
{
    .maxstack 2

    ldarg.0
    ldarg.1
    add

    ret
}
```

## 8.6. Rozgałęzienia

Dwuargumentowe instrukcje rozgałęzień to: `beq`, `bne`, `bgt`, `bge`, `blt`, `ble`, a jednoargumentowe to: `brfalse`, `brtrue`.

Poniżej przykład użycia tych instrukcji poprzez napisanie funkcji zwracającej wartość maksymalną z dwóch podanych liczb:

```
.assembly extern mscorelib{ }

.assembly Maximum
{
    .ver 1:0:1:0
}

.method static void main()
{
    .entrypoint
    .maxstack 4

    .locals init (int32 first,
                  int32 second)

    ldstr "Pierwsza liczba: "
    call void [mscorelib]System.Console::Write(string)
    call string [mscorelib]System.Console::ReadLine()
    call int32 [mscorelib]System.Int32::Parse(string)

    ldstr "Druga liczba: "
    call void [mscorelib]System.Console::Write(string)
    call string [mscorelib]System.Console::ReadLine()
    call int32 [mscorelib]System.Int32::Parse(string)

    ldstr "Maksimum to "
    call void [mscorelib]System.Console::Write(string)

    call int32 max(int32, int32)
    call void [mscorelib]System.Console::Write(int32)

    ret
}

.method static int32 max(int32 a, int32 b)
{
    .maxstack 2

    ldarg.0
    ldarg.1
    bge    firstBigger
    ldarg.1
    ret
}
```

```
firstBigger:  
    ldarg.0  
    ret  
}
```

Przykładowe wyjście programu:

```
C:\ilasm>maximum  
Pierwsza liczba: 2  
Druga liczba: 7  
Maksimum to 7  
C:\ilasm>
```

## 8.7. Pętle

W języku CIL możemy również stosować pętle. Dla przykładu napiszemy program wyświetlający pierwsze 10 kwadratów liczb całkowitych, czyli:

```
1  
4  
9  
16  
25  
36  
49  
64  
81  
100
```

Użyjemy głównej metody z dwiema lokalnymi zmiennymi: k zainicjalizowaną wartością 1 i n zainicjalizowaną wartością 10. Cała strategia polega na zwiększaniu wartości k w przedziale od 1 do 10 i wyświetlanie  $k^2$  przy każdej iteracji. Należy także przy każdej iteracji porównywać wartość k z wartością n, a zrobimy to poprzez odkładanie k oraz n na stos i użycie instrukcji:

```
bgt done
```

aby sprawdzić, czy już należy zakończyć pętlę. Instrukcja br usuwa wartości k oraz n ze stosu. Gdy nie wystąpi skok na koniec programu, to odkładamy dwie kopie zmiennej k na stos i stosujemy instrukcję mul, która mnoży te wartości. Wynik jest zowany na stosie do wyświetlenia go.

Oto cały program:

```
.assembly extern mscorelib{  
  
.assembly LoopExample  
{  
    .ver 1:0:1:0  
}  
  
.method static void main()
```

```
{  
    .entrypoint  
    .maxstack 2  
  
    .locals init (int32 k, int32 n)  
  
    ldc.i4.1      //k = 1  
    stloc.0  
    ldc.i4.10     //n = 10  
    stloc.1  
    topOfLoop:  
    ldloc.0        //k  
    ldloc.1        //n  
    bgt done       //Jeżeli k > n, to kończymy  
    //print k*k  
    ldloc.0  
    ldloc.0  
    mul  
    call void [mscorlib]System.Console::WriteLine(int32)  
    //k = k + 1  
    ldloc.0  
    ldc.i4.1  
    add  
    stloc.0  
    br topOfLoop  
done:  
    ret  
}
```

## 8.8. Przegląd wybranych instrukcji

Jak już wspomniałem, język CIL jest oparty na stosie. Wszystkie instrukcje szukają swoich argumentów na stosie, usuwają je ze stosu, wykonują określone operacje i wynik odkładają z powrotem na stos. Podobnie metody — gdy są wywoływane, pobierają argumenty ze stosu, a przed powrotem odkładają na stos rezultat.

### Instrukcje odkładające wartość na stos

Ta rodzina instrukcji odkłada wartość na stos. Instrukcje te mają postać:

ldc.i4 wartość

Powyższa instrukcja ładuje 4-bajtową (32-bitową) stałą wartość na stos.

Na przykład:

ldc.i4 100

załaduje na stos wartość 100.

Istnieje specjalny format instrukcji, który ładuje na stos małe wartości z zakresu od 0 do 8.

```
ldc.i4.0  
ldc.i4.1  
ldc.i4.8
```

Powyższe instrukcje ładują na stos wartości: 0, 1 oraz 8. Dodatkowo instrukcja:

```
ldc.i4.m1
```

ładuje na stos wartość -1.

Jest też rodzina instrukcji, które ładują na stos wartości zmiennych lokalnych. Zmienne lokalne są deklarowane w następujący sposób:

```
.locals init (int32 a, int32 b, int32 c)
```

który alokuje trzy zmienne lokalne a, b oraz c, przypisując im odpowiednio pozycje 0, 1 oraz 2.

Te wartości mogą zostać odłożone na stos w następujący sposób:

```
ldloc.0 //Odlóż a  
ldloc.2 //Odlóż c
```

Podobnie istnieje rodzina instrukcji odkładających na stos parametry metod. Parametry mają przypisane pozycje zależne od pozycji na liście parametrów metody.

Na przykład metoda:

```
.method int32 myMethod(int32 a, int32b)
```

ma dwa parametry: a oraz b, o pozycjach odpowiednio 0 i 1. Instrukcje ldarg odkładają wartości parametrów na stos:

```
ldarg.0 //Odlóż argument a  
ldarg.1 //Odlóż argument b
```

Referencja do napisu może być odłożona na stos również w następujący sposób:

```
ldstr string
```

## Instrukcje zdejmujące wartość ze stosu

Instrukcje stloc zdejmują wartości z wierzchołka stosu i zapisują je do zmiennych lokalnych.

```
stloc.0 //Zapisz do zmiennej lokalnej pod pozycją 0  
stloc.4 //Zapisz do zmiennej lokalnej pod pozycją 4
```

## Instrukcje rozgałęzień

Instrukcje rozgałęzień określają skoki, zarówno warunkowe, jak i bezwarunkowe.

Skok bezwarunkowy jest określony przez:

br etykieta

Skoki warunkowe są następujące:

beq etykieta  
bne etykieta  
bgt etykieta  
bge etykieta  
blt etykieta  
ble etykieta

Powysze instrukcje pobierają dwie wartości ze stosu, porównują je i wykonują skok, jeżeli pierwsza wartość jest równa, różna, większa, większa lub równa, mniejsza, mniejsza lub równa.

Dwie instrukcje przedstawione poniżej pobierają jedną wartość ze stosu i odpowiednio wykonują skok zależnie od tego, czy wartość jest prawdą (true), czy fałszem (false).

brfalse etykieta  
brtrue etykieta

## Instrukcje arytmetyczne

Poniższe instrukcje dodają, odejmują i mnożą dwie wartości. Te dwie wartości są usuwane ze stosu i zastępowane odpowiednio przez sumę, różnicę lub iloczyn.

add  
sub  
mul

Dodatkowo instrukcja neg zastępuje wartość na stosie wartością negatywną.

## Pozostałe instrukcje

Jeszcze dwie instrukcje są warte zainteresowania:

box int32  
pop

Pierwsza zdejmuje ze stosu 32-bitową liczbę całkowitą i zamienia ją na obiekt (przydatne, gdy mamy liczbę, a jakąś metodą wymaga obiektu). Natomiast druga instrukcja zdejmuje wartość ze stosu i porzuca ją (nie zapisuje jej nigdzie).



## Rozdział 9.

# Podstawy tworzenia aplikacji w stylu Metro dla Windows 8

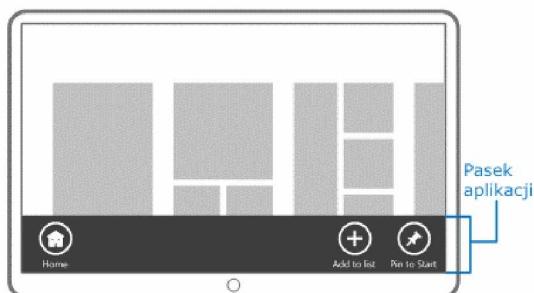
## 9.1. Co to są aplikacje Metro?

Aplikacje Metro to nowy typ aplikacji, które są uruchamiane na urządzeniach z systemem Windows 8. Programy te mają jedno okno, ale wiele widoków. Wspierają też różne układy i widoki.

Jako urządzeń wejściowych możesz użyć touchpada, pióra, myszy czy klawiatury — aplikacja zawsze będzie działała dobrze.

Aplikacje mają nowe kontrolki i powierzchnie interfejsu. Jedną z nowości jest pasek aplikacji (ang. *App bar*), który pojawia się, gdy użytkownik przesunie palec do środka od górnej lub dolnej krawędzi ekranu. Przedstawiony został on na rysunku 9.1. Kolejną nowością jest pasek funkcji (ang. *The charms*), który zawiera funkcje, takie jak: szukaj, udostępnij, połącz, ustawienia oraz przycisk start.

**Rysunek 9.1.**  
*Pasek aplikacji*  
(ang. *App bar*)



Następną cechą aplikacji Metro jest to, że używają one kafelków, a nie ikon, tak jak standardowe aplikacje. Gdy zainstalujesz aplikację, na ekranie startowym pojawia się jej kafelek, którego kliknięcie powoduje uruchomienie wybranej aplikacji.

Aplikacje Metro możesz pisać w języku, który już znasz (chodzi o język C#). Możesz także użyć języków takich jak C++, Visual Basic czy JavaScript.

## 9.2. Potrzebne narzędzia

Jeżeli chcesz wydawać aplikacje Metro, będziesz potrzebował systemu Windows 8 oraz kilku narzędzi deweloperskich.

System Windows 8 Enterprise w 90-dniowej wersji testowej możesz pobrać z adresu:

<http://go.microsoft.com/fwlink/?LinkId=238220>

Będziesz potrzebował również środowiska Microsoft Visual Studio 2012. Możesz użyć środowiska w wersji Ultimate, ale działać ono będzie za darmo tylko przez 90 dni. Możesz także skorzystać z darmowej wersji Express, którą pobierzesz z adresu:

<http://go.microsoft.com/fwlink/?LinkId=238221>

## 9.3. Uzyskiwanie licencji dewelopera

Licencja dewelopera pozwala Ci instalować, wydawać i testować aplikacje Metro, zanim jeszcze sklep Windows je przetestuje i podpisze certyfikatem.

Wyżej wspomnianą licencję możesz uzyskać poprzez środowisko Visual Studio. Gdy po raz pierwszy uruchomisz na swoim komputerze środowisko Microsoft Visual Studio 2012, zostaniesz poinformowany o licencji dewelopera. Gdy ją przeczytasz, musisz ją zaakceptować przyciskiem *I Agree*. Gdy zainstalujesz licencję na swoim komputerze, nie będziesz więcej razy o niej powiadamiany, chyba że licencja wygaśnie (lub usuniesz ją) albo spróbujesz uruchomić aplikację ze sklepu Windows bez certyfikatu.

Raz otrzymaną licencję możesz odnowić. Gdy używasz Microsoft Visual Studio Express 2012 dla Windows 8, kliknij *Store/Acquire Developer License*. Gdy natomiast używasz wersji innej niż Express, kliknij *Project/Store/Acquire Developer License*.

Jeżeli masz licencję dewelopera, możesz uruchamiać aplikacje ze sklepu Windows nieprzetestowane i bez certyfikatu, ale robisz to na własną odpowiedzialność.

## 9.4. Program „Witaj, świecie Metro!”

Aby ukończyć ten poradnik, musisz mieć system Windows 8 and Microsoft Visual Studio 2012 dla Windows 8. Potrzebna Ci będzie także licencja dewelopera (w przednim rozdziale zostało opisane, jak ją uzyskać).

### Tworzenie nowego projektu

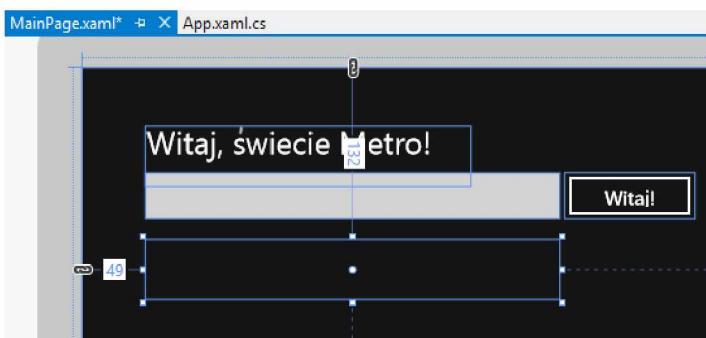
1. Najpierw uruchom Microsoft Visual Studio 2012 Ultimate for Windows 8.
2. Wybierz *FILE/New/Project...*.
3. Rozwiń gałąź *Installed/Templates*, następnie rozwiń *Visual C# i wybierz Windows Metro Style.*
4. W panelu środkowym wybierz szablon *Blank App.*
5. W polu *Name* wpisz "HelloWorld".
6. Kliknij przycisk *OK*, aby utworzyć projekt.

### Zmodyfikuj stronę startową

W okienku *Solution Explorer* kliknij podwojnie *MainPage.xaml*. Kliknij z lewej strony ekranu na wysuwane okienko *Toolbox*, znajdź na nim kontrolkę *TextBlock* i wstaw ją do okna projektowania. Następnie w oknie *Properties* zmień właściwość *Text* kontrolki na *Witaj, świecie Metro!* i właściwość *FontSize* na 24. Niżej wstaw kontrolkę *TextBox* i wyczyść jej właściwość *Text*. Ustaw także w polu *Name* nazwę kontrolki. Wpisz tam *inputName*. Obok kontrolki *TextBox* wstaw kontrolkę *Button* i ustaw jej właściwość *Content* na *Witaj!*. Poniżej wstaw jeszcze jedną kontrolkę *TextBlock* i wyczyść jej właściwość *Text*, a właściwość *FontSize* ustaw na 24. Ustaw jej nazwę (*Name* w oknie *Properties*) na *outputHello*. Wszystko powinno wyglądać tak, jak na rysunku 9.2.

Rysunek 9.2.

Okno programu  
„Witaj, świecie Metro!”  
podczas projektowania



## Dodaj obsługę zdarzeń

Teraz dodamy zdarzenie, jakie zostanie wywołane po kliknięciu przycisku z napisem *Witaj!*, zatem zaznacz ten przycisk. Dalej w oknie *Properties* przełącz się na widok zdarzeń, klikając ikonę z piorunem. Znajdź zdarzenie *Click* i kliknij dwa razy pole obok tego zdarzenia. Wygenerowany zostanie kod zdarzenia, do którego wpisz:

```
outputHello.Text = "Witaj " + inputName.Text + "!";
```

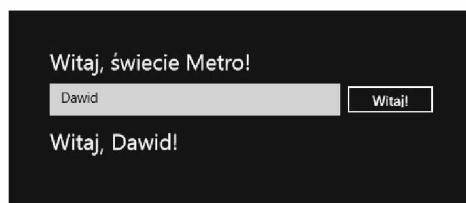
## Uruchom aplikację

Aby skompilować projekt, kliknij w górnym menu *BUILD/Build Solution*. Teraz uruchomimy program — kliknij *DEBUG/Start Debugging* (lub wcisnij klawisz *F5*).

Program powinien wyglądać i działać tak, jak na rysunku 9.3.

Rysunek 9.3.

Program „Witaj, świecie Metro!” podczas działania



## 9.5. Przegląd wybranych kontrolek

### App bar

Pasek narzędziowy wyświetlający polecenia specyficzne dla aplikacji (rysunek 9.4).

Rysunek 9.4.

Przykładowy wygląd kontrolki App bar

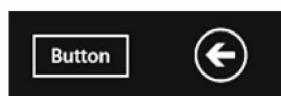


### Button

Przycisk wywołujący zdarzenie *Click* podczas kliknięcia (rysunek 9.5).

Rysunek 9.5.

Przykładowy wygląd kontrolki Button



## Check box

Kontrolka, którą użytkownik może zaznaczyć lub nie (rysunek 9.6).

Rysunek 9.6.

Przykładowy wygląd kontrolki Check box



## Combo box

Lista rozwijana z elementami. Użytkownik może wybrać jeden element z listy (rysunek 9.7).

Rysunek 9.7.

Przykładowy wygląd kontrolki Combo box

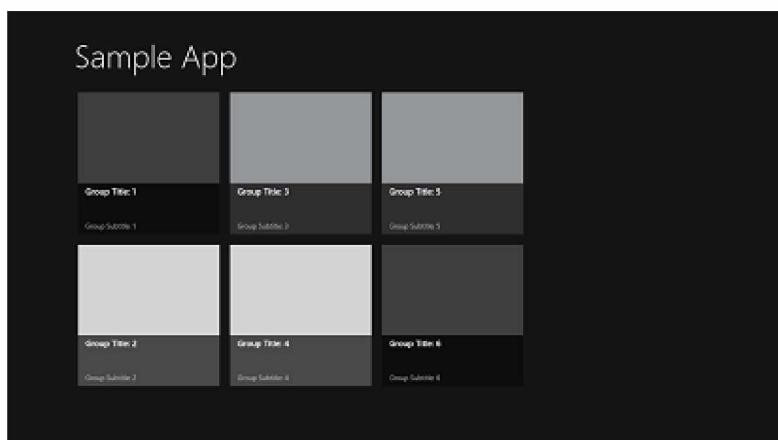


## Grid view

Kontrolka prezentująca kolekcję elementów w wierszach i kolumnach (rysunek 9.8).

Rysunek 9.8.

Przykładowy wygląd kontrolki Grid view



## Hyperlink

Reprezentuje hiperłącze (rysunek 9.9).

**Rysunek 9.9.**  
Przykładowy wygląd  
kontrolki *Hyperlink*



## List box

Kontrolka prezentująca nierożwianą listę elementów, z której użytkownik może wybierać (rysunek 9.10).

**Rysunek 9.10.**  
Przykładowy wygląd  
kontrolki *List box*



## List view

Kontrolka prezentująca kolekcję elementów, którą użytkownik może przewijać pionowo (rysunek 9.11).

**Rysunek 9.11.**  
Przykładowy wygląd  
kontrolki *List view*



## Password box

Kontrolka do wpisywania hasła (rysunek 9.12).

**Rysunek 9.12.**

Przykładowy wygląd  
kontrolki Password box



## Progress bar

Kontrolka do wyświetlania postępu (rysunki 9.13 i 9.14).

**Rysunek 9.13.**

Kontrolka Progress  
bar wyświetlająca  
określony postęp



**Rysunek 9.14.**

Kontrolka Progress  
bar wyświetlająca  
nieokreślony postęp



## Progress ring

Kontrolka wyświetlająca nieokreślony postęp w postaci pierścienia (rysunek 9.15).

**Rysunek 9.15.**

Przykładowy wygląd  
kontrolki Progress ring



## Radio button

Kontrolka pozwalająca wybrać jedną opcję z grupy dostępnych opcji (rysunek 9.16).

**Rysunek 9.16.**

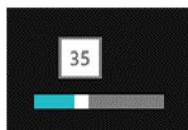
Przykładowy wygląd  
kontrolki Radio button



## Slider

Kontrolka pozwala użytkownikowi wybrać wartość z określonego przedziału (rysunek 9.1).

**Rysunek 9.17.**  
*Przykładowy wygląd kontrolki Slider*



## Text block

Kontrolka wyświetlająca tekst, którego użytkownik nie może edytować (rysunek 9.18).

**Rysunek 9.18.**  
*Przykładowy wygląd kontrolki Text block*



## Text box

Kontrolka wyświetlająca tekst jednolinijkowy lub wielolinijkowy (rysunek 9.19).

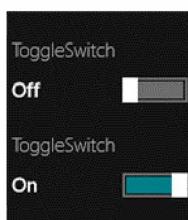
**Rysunek 9.19.**  
*Przykładowy wygląd kontrolki Text box*



## Toggle switch

Przełącznik przyjmujący jeden z dwóch możliwych stanów (rysunek 9.20).

**Rysunek 9.20.**  
*Przykładowy wygląd kontrolki Toggle switch*

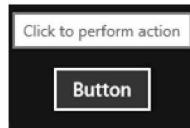


## Tooltip

Kontrolka podpowiedzi. Wyświetla informacje o obiekcie (rysunek 9.21).

**Rysunek 9.21.**

Przykładowy wygląd  
kontrolki Tooltip





## Dodatek A

# Słowa kluczowe języka C#

Słowa kluczowe są predefiniowanymi zarezerwowanymi identyfikatorami, które mają specjalne znaczenie dla kompilatora. Nie mogą być one używane jako zwykłe identyfikatory, o ile nie mają znaku @ jako prefiksu. Na przykład @if jest dozwolonym identyfikatorem, natomiast if już nie, gdyż jest to słowo kluczowe.

abstract	event	new	struct
as	explicit	null	switch
base	extern	object	this
bool	false	operator	throw
break	finally	out	true
byte	fixed	override	try
case	float	params	typeof
catch	for	private	uint
char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort
continue	in	return	using
decimal	int	sbyte	virtual
default	interface	sealed	volatile
delegate	internal	short	void
do	is	sizeof	while
double	lock	stackalloc	-
else	long	static	-
enum	namespace	string	-



## Dodatek B

# Zestaw instrukcji Asemblera IL

Poniżej opis zestawu instrukcji Asemblera IL platformy .NET.

## Operacje arytmetyczne

### Dodawanie

add  
add.ovf  
add.ovf.un

### Odejmowanie

sub  
sub.ovf  
sub.ovf.un

### Mnożenie

mul  
mul.ovf  
mul.ovf.un

## Dzielenie

div  
div.un

## Modulo

rem  
rem.un

## Wartość negatywna

neg

## Operacje bitowe

### Koniunkcja

and

### Alternatywa

or

### Negacja

not

### Alternatywa wykluczająca

xor

### Przesunięcie bitowe w prawo

shr  
shr.un

### Przesunięcie bitowe w lewo

shl

# Operacje odkładania na stos

ldarg.0  
ldarg.1  
ldarg.2  
ldarg.3  
ldarg  
ldarga  
ldargs  
ldarga.s  
ldc.i4.0  
ldc.i4.1  
ldc.i4.2  
ldc.i4.3  
ldc.i4.4  
ldc.i4.5  
ldc.i4.6  
ldc.i4.7  
ldc.i4.8  
ldc.i4.m1  
ldc.i4.s  
ldc.i4  
ldc.i8  
ldc.r4  
ldc.r8  
ldind.i  
ldind.i1  
ldind.i2  
ldind.i4  
ldind.i8  
ldind.r4  
ldind.r8  
ldind.u1  
ldind.u2  
ldind.u4  
ldind.ref  
ldloc  
ldloca  
ldloc.0  
ldloc.1  
ldloc.2  
ldloc.3  
ldloc.s  
ldloca.s  
ldftn  
ldnull  
ldelem  
ldelem.i  
ldelem.i1  
ldelem.i2

```
ldelem.i4
ldelem.i8
ldelem.r4
ldelem.r8
ldelem.ref
ldelem.u1
ldelem.u2
ldelem.u4
ldelema
ldfld
ldflda
ldobj
ldsfld
ldsflda
ldstr
```

## Operacje zdejmowania ze stosu i zapisywania

```
starg
starg.s
stloc.0
stloc.1
stloc.2
stloc.3
stloc.s
stlelem
stlelem.i
stlelem.i1
stlelem.i2
stlelem.i4
stlelem.i8
stlelem.r4
stlelem.r8
stelem.ref
stfld
stsfld
stind.i
stind.i1
stind.i2
stind.i4
stind.i8
stind.r4
stind.r8
stind.ref
stloc
stobj
dup
pop
```

## Konwersje

conv.i  
conv.i1  
conv.i2  
conv.i4  
conv.i8  
conv.ovf.i  
conv.ovf.i.un  
conv.ovf.il  
conv.ovf.il.un  
conv.ovf.i2  
conv.ovf.i2.un  
conv.ovf.i4  
conv.ovf.i4.un  
conv.ovf.i8  
conv.ovf.i8.un  
conv.ovf.u  
conv.ovf.u.un  
conv.ovf.ul  
conv.ovf.il.un  
conv.ovf.i2  
conv.ovf.i2.un  
conv.ovf.i4  
conv.ovf.i4.un  
conv.ovf.i8  
conv.ovf.i8.un  
conv.r.un  
conv.r4  
conv.r8  
conv.u  
conv.u1  
conv.u2  
conv.u4  
conv.u8

## Porównywanie

ceq  
cgt  
cgt.un  
clt  
clt.un

## Skoki bezwarunkowe

br  
br.s  
jmp

## Skoki warunkowe

beq  
beq.s  
bge  
bge.s  
bge.un  
bge.un.s  
bgt  
bgt.s  
bgt.un  
bgt.un.s  
ble  
ble.s  
ble.un  
ble.un.s  
blt  
blt.s  
blt.un  
blt.un.s  
bne.un  
bne.un.s  
brfalse.s  
brtrue.s

## Wywoływanie metod i powrót

call  
calli  
callvirt  
ret

## Opakowywanie

box  
unbox  
unbox.any

## Wyjątki

throw  
rethrow  
endfilter  
endfinally

## Bloki pamięci

cpblk  
initblk

## Wskaźniki

arglist  
cpobj  
ldvirtftn  
mkrefany

## Pozostałe

ldlen  
sizeof  
break  
ldtoken  
refanytype  
refanyval  
castclass  
ckfinite  
initobj  
isinst  
leave.s  
leave  
newarr  
newobj  
nop



# 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 polecen, 52  
asembler IL, 165

## B

biblioteka mscorlib, 166  
biblioteki DLL, 127, 141  
błędы, 146  
błędы 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  
komplikacja 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  
NameValuePairCollection, 110  
NetworkCredential, 160  
Object, 94  
Process, 142  
SmtpClient, 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  
zagnieżdzone, 68  
zapieczętowane, 78  
klawiatura, 150  
klucz, 140  
kod nienadzorowany, unsafe code, 60  
kolekcje, 99  
komentarz  
    blokowy, 20  
    liniowy, 20  
    XML, 20  
komplilator, 9  
komplilator ilasm.exe, 165  
kompresja, 139  
koniuunkcja  
    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  
    plytka, 85  
kopiowanie katalogów, 135  
kowarianca, 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  
niezmiennosć obiektów String, 130

## O

obiekt, 23  
obiekty String, 130  
obsługa  
    blędów, 146  
    zdarzeń, 178  
odczyt z pliku, 79  
odczyt z pliku tekowego, 138  
odpluskiwacz, Debugger, 10  
odsmeiacz 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

przesłanianie 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  
sterta, 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

- górnego menu, 16

- instalacja, 12

- kompilacja, 15

- konfiguracja, 14

odpluskwianie, 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

wskazniki, 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