POLITECHNIKA LUBELSKA

Beata Pańczyk
Marcin Badurowicz

Programowanie obiektowe
Język C#

PODRĘCZNIKI

Lublin 2013

Programowanie obiektowe
Język C#

Podręczniki – Politechnika Lubelska

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Publikacja współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Beata Pańczyk
Marcin Badurowicz

Programowanie obiektowe
Język C#

Politechnika Lubelska
Lublin 2013

Recenzent:
dr inż. Grzegorz Kozieł

Publikacja dystrybuowana bezpłatnie.

Publikacja finansowana z projektu „Politechnika przyszłości – dostosowanie oferty do potrzeb rynku pracy i GOW"

Projekt „Politechnika przyszłości – dostosowanie oferty do potrzeb rynku pracy i GOW" współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego. Umowy UDA-POKL.04.03.00-00-129/12

Publikacja wydana za zgodą Rektora Politechniki Lubelskiej

ISBN: 978-83-63569-97-6

Wydawca: Politechnika Lubelska
ul. Nadbystrzycka 38D, 20-618 Lublin
Realizacja: Biblioteka Politechniki Lubelskiej
Ośrodek ds. Wydawnictw i Biblioteki Cyfrowej
ul. Nadbystrzycka 36A, 20-618 Lublin
tel. (81) 538-46-59, email: wydawca@pollub.pl
www.biblioteka.pollub.pl
Druk: TOP Agencja Reklamowa Agnieszka Łuczak
www.agencjatop.pl

Elektroniczna wersja książki dostępna w Bibliotece Cyfrowej PL www.bc.pollub.pl
Nakład: 100 egz.

Spis treści

Wstęp

Programowanie obiektowe jest obecnie podstawą programowania różnego rodzaju systemów informatycznych, z uwzględnieniem również programowania aplikacji na urządzenia mobilne. Jest to przedmiot ujęty w standardach kształcenia studentów uczelni technicznych i przede wszystkim do nich jest adresowany ten podręcznik. Znajomość podstaw programowania obiektowego (niezależnie w jakim języku) umożliwia, zgodne z obecnymi trendami, wytworzenie aplikacji z wykorzystaniem bogactwa gotowych bibliotek klas, oferowanych przez współczesne języki programowania (takie jak C#, Java, C++) oraz daje niezbędne podstawy do samodzielnego rozwiązywania specyficznych, coraz bardziej złożonych problemów programistycznych. Wymaga to z jednej strony świadomości istoty rozwiązywanych zagadnień, z drugiej zaś znajomości paradygmatów programowania obiektowego. Znajomość ta jest niezbędna żeby w pełni wykorzystać potencjał programowania obiektowego.

Treść niniejszego podręcznika stanowią wybrane zagadnienia dotyczące programowania obiektowego w języku C#. Teoretyczne podstawy bazują na pozycjach [1-3], które obejmują znacznie obszerniejszy materiał. Niniejszy podręcznik zawiera tylko wyselekcjonowane informacje, które są zwykle omawiane na wykładach. Autorzy ograniczyli się do niezbędnych elementów teorii, bardziej koncentrując się na przykładach, dobranych w taki sposób, aby jak najprościej zobrazować omawiany temat. Każdy rozdział zawiera odpowiednio opracowane zbiory zadań programistycznych. Samodzielna realizacja tych zadań pomoże Czytelnikowi w utrwaleniu i zrozumieniu prezentowanego materiału.

Wszelkie uwagi i propozycje prosimy kierować do autorów na adres:
b.panczyk@pollub.pl lub
m.badurowicz@pollub.pl.

Autorzy

1. Wprowadzenie do programowania w języku C#

1.1. Wstęp

Platforma .NET to idea i sposób budowania aplikacji dla systemu Windows. Podczas instalacji platformy .NET kopiowane są biblioteki DLL, aplikacje dla programistów oraz dokumentacja (dzięki czemu można korzystać z nowych narzędzi oferowanych przez Microsoft). Już Windows 2003 standardowo był wyposażony w platformę .NET i nie ma potrzeby ponownej jej instalacji. Podstawowym składnikiem platformy są biblioteki i technologie wspierające działanie programu, odpowiedzialne za zarządzanie pamięcią aplikacji itp.

Programy pisane w C# wymagają .NET Framework. C# to język programowania, który powstał specjalnie na potrzeby tej platformy; zbudowany jest na bazie C++ oraz Javy i umożliwia obsługę wszystkich technologii .NET Framework.

Pakiet Visual Studio .NET jest zintegrowanym środowiskiem programistycznym dla platformy .NET Framework. Obecnie najnowsza jego wersja to Visual Studio 2013. Jest to środowisko komercyjne, ale do nauki programowania w C# można skorzystać z bezpłatnej wersji Visual Studio Express 2012 for Windows Desktop do pobrania ze strony:

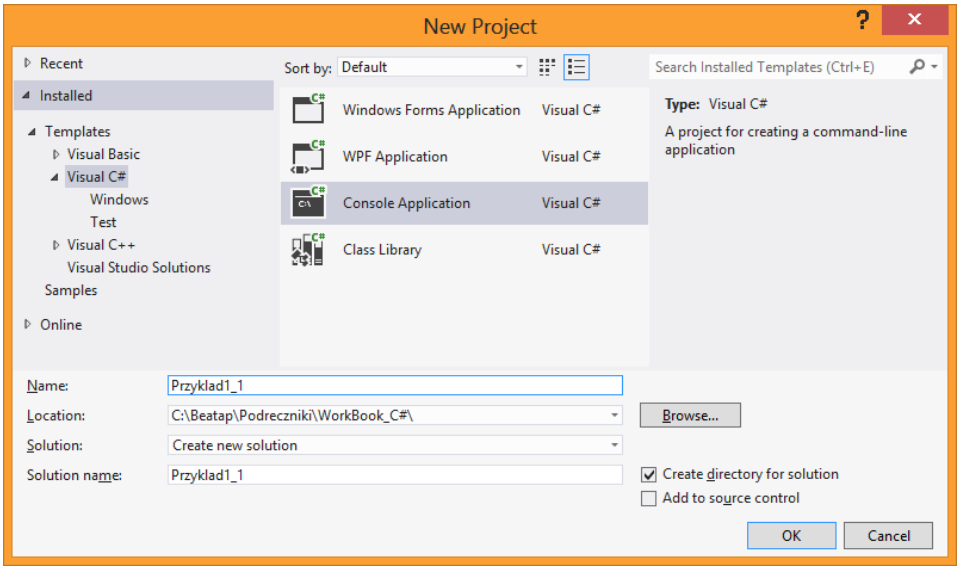
http:// http://www.microsoft.com/visualstudio/plk/downloads.

Wszystkie przykłady realizowane w niniejszym podręczniku wykonane zostały w tej wersji Visual Studio 2012.

Programując w C# można tworzyć różne rodzaje aplikacji:

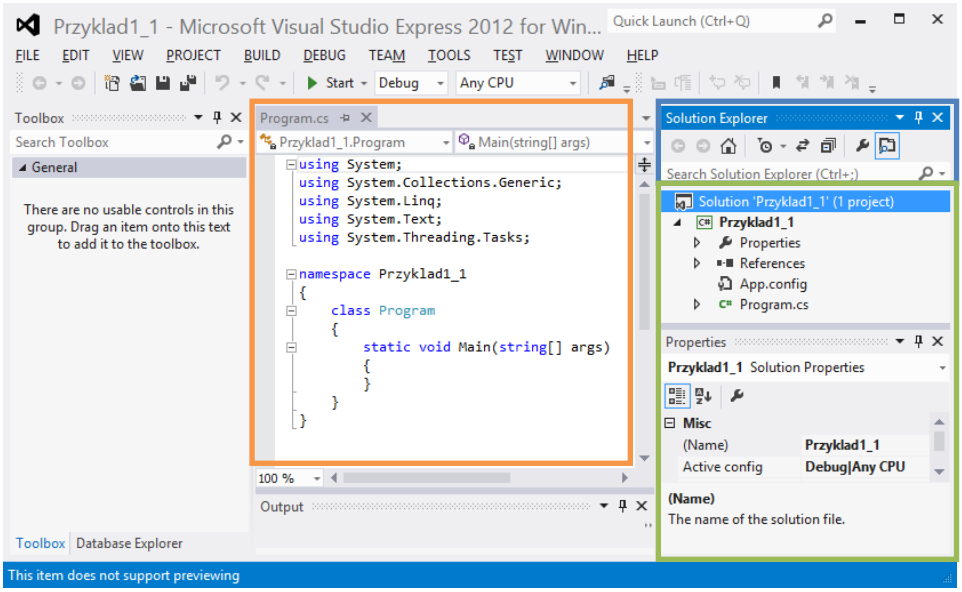
- aplikacje konsolowe, czyli programy uruchamiane w oknie konsoli; nie posiadające interfejsu użytkownika;
- aplikacje *Windows Forms* (WinForms) korzystające z biblioteki wizualnej, umożliwiającej projektowanie graficznego interfejsu użytkownika (ang. Graphical User Interface – GUI). Biblioteka ta jest częścią środowiska .NET Framework, więc komponenty (np. Button, Menu, Label, itp.) mogą zostać użyte w Delphi/C#/Visual Basic.NET;
- aplikacje internetowe (z użyciem technologii ASP.NET lub nowszej ASP.NET MVC);
- aplikacje dla urządzeń mobilnych działające pod kontrolą *Windows Phone*.

Przykłady prezentowane w niniejszym rozdziale dotyczą tworzenia aplikacji konsolowych. Po uruchomieniu środowiska Visual Studio Express 2012, aplikację konsolową możemy utworzyć aktywując kolejno opcje menu:
FILE -> New Project
oraz, po wskazaniu języka programowania C#, wybrać:
Console Application (Rys.1.1).



Rys. 1.1. Tworzenie nowego projektu aplikacji konsolowej w Visual Studio Express 2012 for Windows Desktop

Po prawidłowym utworzeniu aplikacji o nazwie *Przyklad1_1* zobaczymy ekran jak na rysunku 1.2.



Rys. 1.2. Widok po utworzeniu projektu aplikacji konsolowej w Visual Studio Express 2012 for Windows Desktop

1.2. Podstawowe elementy języka

1.2.1. Struktura programu

Pliki programów języka C# posiadają rozszerzenie `.cs`. Zwróćmy uwagę na wejściowy plik projektu o nazwie `program.cs` (pomarańczowa ramka na rysunku 1.2). Jego źródło, dodatkowo uzupełnione komentarzami i instrukcją wyświetlenia tekstu powitalnego na ekranie przedstawia przykład 1.1.

Przykład 1.1. Podstawowy plik program.cs aplikacji konsolowej

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Przyklad1_1
{
```

```
    class Program
    {
        // metoda Main – tu zaczyna się działanie_aplikacji
        static void Main(string[] args)
        {
            // wyświetl powitanie na ekranie:
            System.Console.WriteLine("Witaj C#!");
            //wczytaj linię z klawiatury:
            System.Console.ReadLine();
        }
    }
}
```

Każdy program w C# musi zawierać klasę. Taką klasą w przykładzie 1.1. jest klasa o nazwie `Program`. Każda klasa jest definiowana począwszy od słowa kluczowego `class`, po którym następuje jej nazwa (u nas `Program`) a następnie jej ciało ujęte w nawiasy klamrowe `{ }`.

W podstawowej klasie aplikacji konsolowej (w naszym przykładzie - jest to klasa `Program`) musi istnieć specjalna metoda, w której rozpoczyna się i kończy działanie aplikacji. Nagłówek takiej metody głównej ma postać:

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

Ciało metody `Main` (jak również każdej innej) ujęte jest także w nawiasy klamrowe.

Na chwilę obecną istotne jest, że taka metoda musi istnieć dla każdej aplikacji oraz, że w tej metodzie umieszczamy konieczne deklaracje zmiennych lokalnych i instrukcje realizujące odpowiednie działanie programu. Wszelkie szczegóły dotyczące parametrów i specyfikatorów metod zostaną wyjaśnione kolejno w dalszych rozdziałach.

Środowisko .NET Framework udostępnia programistom szereg gotowych klas i bibliotek, które ułatwiają proces programowania. Główna biblioteka w .NET Framework (`mscorlib.dll`) zawiera przestrzeń nazw `System` z klasą `Console` z metodą `WriteLine()`. Metoda pozwala wyświetlić komunikat na ekranie (wywołania tej metody pokazano również w przykładzie 1.1):

```
System.Console.WriteLine("Witaj C#!");
```

Metoda `ReadLine()` z klasy `Console` pozwala z kolei pobrać ciąg znaków wprowadzony przez użytkownika z klawiatury:

```
System.Console.ReadLine();
```

.NET Framework zawiera tysiące klas i innych typów, każdy posiada inną nazwę, ale w różnych klasach istnieją funkcje (metody) o takich samych nazwach, np. metoda `WriteLine()` klasy `Console` wypisuje tekst na konsoli a metoda `WriteLine()` klasy `TextWriter` zapisuje tekst do pliku – metody

te należą do innych klas. Podobnie jest w przypadku przestrzeni nazw (ang. namespace) - w obrębie kilku przestrzeni nazw mogą istnieć klasy o tej samej nazwie. Przykład 1.2 przedstawia definicję dwóch przestrzeni nazw (przestrzeń `owoce` i przestrzeń `samochody`), każda z klasą o nazwie `Opis`.

Przykład 1.2. Definicja przestrzeni nazw

```
namespace owoce
{
    class Opis
    { }
}
namespace samochody
{
    class Opis
    { }
}
```

Mając tak zdefiniowane przestrzenie nazw jak w przykładzie 1.2 łatwo można zidentyfikować, z której klasy korzystamy, tzn. odwołanie do klasy postaci:

```
samochody.Opis
```

wskazuje na klasę `Opis` z przestrzeni nazw `samochody` a wywołanie postaci:

```
owoce.Opis
```

wskazuje również na klasę `Opis`, ale z przestrzeni nazw `owoce`.

Dla każdej nowej aplikacji tworzonej w Visual Studio, środowisko automatycznie tworzy przestrzeń nazw o nazwie tej aplikacji (w naszym przykładzie jest to `namespace Przyklad1_1`). Pozwala to uniknąć ewentualnych konfliktów nazw stosowanych w innej aplikacji.

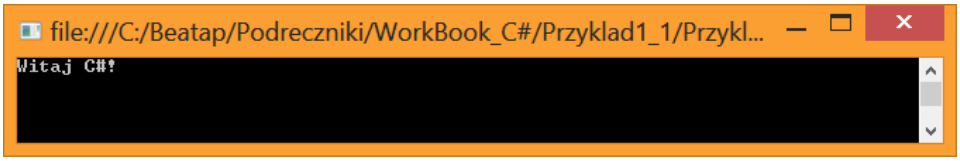
Dodatkowo zamiast pisać za każdym razem nazwę przestrzeni nazw, klasę i metodę - można wykorzystać słowo kluczowe `using`, które informuje, że program korzysta z klas znajdujących się w danej przestrzeni nazw (np. `System`). Visual Studio automatycznie dodaje kilka przestrzeni nazw (w naszym przypadku korzystamy jedynie z przestrzeni `System`). Zatem przykład 1.1. możemy uprościć do postaci przedstawionej jako przykład 1.3. (zwróć uwagę na wiersze pogrubione).

Przykład 1.3. Modyfikacja przykładu 1.1.

```
using System;
namespace Przyklad1_1
{
```

```
    class Program
    {
        // metoda Main – tu zaczyna się działanie_aplikacji
        static void Main(string[] args)
        {   // wyświetl powitanie na ekranie:
            Console.WriteLine("Witaj C#!");
            //wczytaj linię z klawiatury:
            Console.ReadLine();
        }
    }
}
```

Uruchomienie aplikacji w Visual Studio z debugowaniem odbywa się poprzez wybór opcji z menu głównego: *DEBUG -> Start Debugging* lub za pomocą przycisku z ikoną zielonej strzałki i etykietą *Start* (ewentualnie za pomocą klawisza *F5*). Uruchamianie bez debugowania możemy zrealizować wybierając opcje: *DEBUG -> Start Without Debugging* lub za pomocą klawisza *Ctrl+F5*). Wynik działania aplikacji z przykładu 1.3 pokazuje rysunek 1.3.



Rys.1.3. Wynik działania aplikacji z przykładu 1.3.

1.2.1. Zmienne i stałe, typy danych

Każda zmienna wykorzystywana w programie C# musi mieć nazwę oraz zdefiniowany typ, czyli dziedzinę wartości, które jej można przypisać.

Nadając nazwę zmiennej, podobnie jak w innych językach programowania, należy przestrzegać pewnych zasad:

- pierwszym znakiem nazwy zmiennej nie może być cyfra - nazwa musi rozpoczynać się od litery;
- nazwa zmiennej może zawierać na początku znak podkreślenia _, ale nie znaki: ( ) * & ^ % # @ ! / = + - [}] } ' " ; , . ?;
- w nazwach zmiennych można używać polskich znaków lub innych wchodzących w skład Unicode (platforma .NET i język C# obsługują ten standard kodowania).

Podstawowe typy zmiennych języka C# oraz ich wartości przedstawione są w tabeli 1.1.

Tabela 1.1. Podstawowe typy danych C#

Typ	Wartości
byte	od 0 do 255
sbyte	od –128 do 127
short	od –32,768 do 32,767
int	od –2,147,483,648 do 2,147,483,647
uint	od 0 do 4,294,967,295
long	od –9,223,372,036,854,775,808 do 9,223,372,036,854,775,807
ulong	od 0 do 18,446,744,073,709,551,615
float	od –3.402823e38 do 3.402823e38
double	od –1.79769313486232e308 do 1.79769313486232e308
decimal	od –79228162514264337593543950335 do 79228162514264337593543950335
char	pojedynczy znak
string	łańcuch znaków typu char
bool	true lub false

Przy deklaracji zmiennej możliwa jest również inicjalizacja jej wartości początkowej. Sposób deklaracji zmiennych pokazuje przykład 1.4.

Przykład 1.4. Deklaracje zmiennych

```
//deklaracje zmiennych:
string napis;
char znak;
double x, y, z;
//deklaracje z nadaniem wartości początkowej:
int n = 8;
string nazwa = "ABC";
bool dobrze = true;
```

Wartości zmiennych ulegają modyfikacji w trakcie działania programu, natomiast wartość stałej przydzielana jest jeszcze w trakcie pisania kodu. Wartości stałych nie można zmieniać w trakcie działania aplikacji. Raz przypisana wartość pozostaje w pamięci komputera aż do zakończenia działania aplikacji.

Definiowanie wartości stałej odbywa się analogicznie jak deklaracja zmiennej z inicjalizacją, ale dodatkowo poprzedzona jest słowem kluczowym **`const`**:

```
const double EPS = 0.0001;
const int N = 1000;
```

Zgodnie z ogólnie przyjętą konwencją nazwy stałych piszemy wielkimi literami.

1.2.2. Klasa Console i formatowanie tekstu

Podstawowe operacje związane z wyświetlaniem komunikatów na ekranie oraz pobieranie danych z klawiatury realizowane są za pomocą metod klasy `Console`.

Podstawowe metody tej klasy to:

- `WriteLine` – wyświetla komunikat podany jako parametr metody,
- `ReadLine` – pobiera łańcuch znaków z klawiatury (do momentu wciśnięcia klawisza *Enter*),
- `Beep` - odgrywa dźwięk z głośnika systemowego,
- `Clear` - czyści ekran konsoli,
- `ResetColor` - ustawia domyślny kolor tła oraz tekstu,
- `SetCursorPosition` - ustawia pozycję kursora w oknie konsoli,
- `SetWindowPosition` - ustawia położenia okna konsoli,
- `SetWindowSize` - określa rozmiar okna konsoli.

Zwróćmy uwagę na nazwy metod klas. Wszystkie metody klas noszą nazwy zgodnie z następującą konwencją: pierwsza litera jest wielka a jeśli nazwa zawiera kilka słów – kolejne słowa również wyróżnione są wielką literą np.:

`SetCursorPosition.`

Sposób wykorzystania wybranych metod klasy `Console` przedstwia przykład 1.5.

Przykład 1.5. Podstawowe metody klasy Console

```
using System;
class Program
{
    static void Main(string[] args)
    {
        // ustaw rozmiar okna:
        Console.SetWindowSize(60, 30);
        // ustaw położenie tekstu:
        Console.SetCursorPosition(10, 10);
```

```
        // wyświetlenie komunikatu na ekranie:
        Console.WriteLine("Podaj imię:");
        // deklaracja zmiennej string:
        string imie;
        // przypisanie zmiennej imie łańcucha pobranego z klawiatury
        imie = Console.ReadLine();
        // wykorzystanie operatora + w celu konkatenacji łańcuchów:
        Console.WriteLine("Witaj " + imie + ".");
        // oczekiwanie na kolejny łańcuch wpisany z klawiatury:
        Console.ReadLine();
        Console.Clear();
        Console.Beep();
    }
}
```

W środowisku .NET można wykorzystać styl formatowania tekstu podobnie do funkcji `printf` z języka C. Każde miejsce, w którym należy wpisać wartość zmiennej ujęte jest w nawiasy klamrowe `{}`, jak pokazuje przykład 1.6.

Opcjonalnie każdy taki obszar do wypełnienia `{}` może zawierać następujące znaki formatujące (małe lub wielkie litery):

- **C** lub **c** – do formatowania waluty (domyślnie określone w ustawieniach regionalnych Windows, można zmienić za pomocą obiektu `System.Globalization.NumberFormatInfo`),
- **D** lub **d** – formatuje liczby dziesiętne (może również określać minimalną liczbę cyfr użytych do wypełnienia wartości),
- **E** lub **e** – notacja wykładnicza,
- **F** lub **f** – ustalona liczba miejsc po przecinku,
- **G** lub **g** – format ogólny (stałoprzecinkowo lub wykładniczo),
- **N** lub **n** – podstawowy format liczbowy (z przecinkami),
- **X** lub **x** – format heksadecymalny.

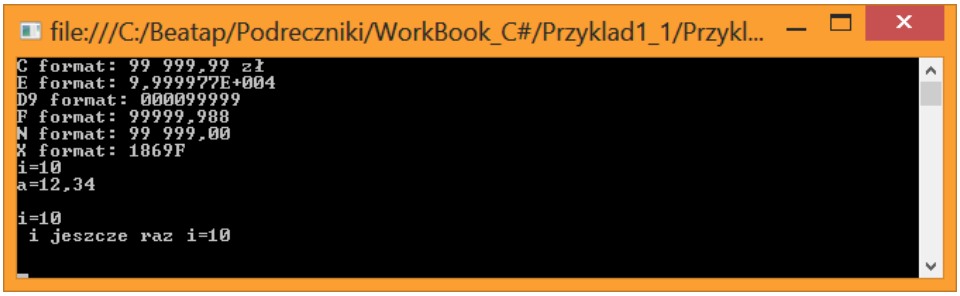
Przykład 1.6 demonstruje możliwości związane z formatowaniem tekstu.

Przykład 1.6. Formatowanie tekstu

```
using System;

class Program
{
    // metoda Main – tu zaczyna się działanie_aplikacji
    static void Main(string[] args)
    {
        Console.WriteLine("C format: {0:C}", 99999.987);
        Console.WriteLine("E format: {0:E}", 99999.76543);
        Console.WriteLine("D9 format: {0:D9}", 99999);
```

```
            Console.WriteLine("F format: {0:F3}", 99999.98765);
            Console.WriteLine("N format: {0:N}", 99999);
            Console.WriteLine("X format: {0:X}", 99999);
            int i = 10;
            double a = 12.34;
            //wyświetlenie domyślnie sformatowanych wartości: i,a
            Console.WriteLine("i={0}\na={1}\n", i, a);
            //możliwy jest również wydruk postaci:
            Console.WriteLine("i={0}\n i jeszcze raz i={0}\n", i);
            Console.ReadLine();
        }
}
```

W wydruku, w miejscu `{0}` zostaje wstawiona wartość pierwszej zmiennej (lub stałej) występującej jako kolejny parametr metody `WriteLine` (np. 99999 lub `i` w naszym przykładzie), `{1}` oznacza miejsce dla wartości kolejnego parametru (np. `a` w przykładzie) itd. Wynik wykonania programu z przykładu 1.6 pokazuje rysunek 1.4.



Rys.1.4. Wynik działania programu z przykładu 1.6.

Te same znaki sterujące mogą zostać użyte do przygotowania odpowiedniego formatu napisu w metodzie `Format` klasy `String` (Przykład 1.7.).

Zastosowanie metody `ReadLine()` do wczytania danych innego typu niż `string` wymaga przeprowadzenia odpowiedniej konwersji np. na typ liczbowy. W tym celu można zastosować metody klasy `Convert`, dostępnej również w przestrzeni nazw `System`.

Przykładowe metody klasy `Convert` to:

- `ToBoolean`,
- `ToByte`,
- `ToChar`,

- `ToInt16` (konwersja na 16-bitową liczbę całkowitą),
- `ToInt32` (konwersja na 32-bitową liczbę całkowitą),
- `ToInt64` (konwertuje na 64-bitową liczbę całkowitą),
- `ToDouble`,
- `ToString`, itp.

Przykład 1.6. Zastosowanie metod klasy Convert i Format

```
//deklaracja zmiennej typu string:
string s1;
//deklaracja zmiennej typu int:
int liczbaInt;
//wczytanie liczby int - wykorzystanie klasy Console i Convert:
Console.WriteLine("Podaj liczbę całkowitą:");
liczbaInt = Convert.ToInt32(Console.ReadLine());
double liczbaDouble;
//wczytanie liczby double - wykorzytsanie klasy Console i Convert:
Console.WriteLine("Podaj liczbę rzeczywistą:");
liczbaDouble = Convert.ToDouble(Console.ReadLine());
// przygotowanie napisu:
s1 = String.Format("x={0:C}, liczbaInt={1}, liczbaDouble={2}",
                   99.999, liczbaInt, liczbaDouble);
// wyświetlenie napisu na konsoli:
Console.WriteLine(s1);
```

W przypadku podania niewłaściwego formatu dla liczby typu `int` czy `double` w przykładzie 1.7, zostanie zgłoszony błąd w trakcie wykonywania programu (*An unhandled exception of type 'System.FormatException' occurred in mscorlib.dll. Additional information: Nieprawidłowy format ciągu wejściowego*). Sposobem rozwiązania tego problemu jest zastosowanie obsługi wyjątków, co jest tematem rozdziału 2.2.5.

1.2.3. Instrukcje sterujące i operatory

Instrukcje sterujące (analogiczne jak w językach C/C++, Java), zestawiono w tabeli 1.2. Podstawowe operatory języka C# przedstawia tabela 1.3.

Tabela 1.2. Podstawowe instrukcje języka C#

Instrukcja	Postać

warunkowa	`if (warunek)`
	`{ // instrukcje do wykonania`
	`}`
	`else`
	`{ // instrukcje do wykonania`
	`}`

wyboru	`switch (wartość)`
	`{ case stała1: // instrukcje do wykonania`
	`break;`
	`case stała2: // instrukcje do wykonania`
	`break;`
	`...`
	`default: // instrukcje do wykonania`
	`break;`
	`}`
	`//UWAGA! – konieczny break w każdym case`

pętli: while	`while (warunek_zakończenia)`
	`{ // instrukcje do wykonania w pętli while`
	`}`

pętli: do	`do`
	`{ // instrukcje do wykonania w pętli do`
	`} while (warunek_zakończenia);`

pętli: for	`for (wartości_startowe; warunek_stopu; licznik_pętli)`
	`{ // instrukcje do wykonania w pętli for`
	`}`

przerwania:	`break;`
kontynuacji:	`continue;`
wyjścia z metody	`return;` `return wynik;`

Tabela 1.3. Podstawowe operatory języka C#

Operatory	
porównania i relacji:	== != > < >= <= is as
arytmetyczne:	+ - * / %
incrementacji i decrementacji:	++ --
logiczne:	&& || !
bitowe:	& | ^ ~ >> <<
przypisania:	= += -= /= *= %=
warunkowy	?:
rzutowania	(typ)

1.2.4. Rzutowanie typów, wartości i referencje

We fragmencie kodu:

```
char c;
byte b;
c = 'A'; //dobrze
b = c; //źle
```

podjęto próbę przypisania zmiennej `b` typu `byte`, wartości zmiennej `c` typu `char`, co spowoduje błąd kompilacji (*Error: Cannot implicitly convert type 'char' to 'byte'. An explicit conversion exists (are you missing a cast?)*). Taka konwersja jest dopuszczalna, ale wówczas jeżeli jawnie zdefiniujemy chęć zmiany typu (informując w ten sposób kompilator, że robimy to świadomie) za pomocą operatora rzutowania na odpowiedni typ:

```
b = (byte) c; //dobrze - wykorzystanie operatora rzutowania
```

Rzutowanie zatem to mechanizm udostępniany przez język programowania, polegający na zmianie typu danej zmiennej, dzięki czemu możliwa jest konwersja na różne typy danych.

Rzutowanie to również sposób na obcinanie części liczb zmienno-przecinkowych, np.:

```
double d;
int i;
d = 454.54;
i = (int)d;
```

Zasadniczo w języku C# rozróżniamy dwa rodzaje typów:

- **typy będące wartościami** – to wszystkie typy danych liczbowych (`int`, `float` itd.) oraz typy wyliczeniowe i struktury. W przypadku typów będących wartościami, przy przypisywaniu jednej zmiennej do drugiej tworzy się lokalna kopia bitowa. Takie zmienne przechowywane są na stosie i usuwane z pamięci gdy wychodzą poza zdefiniowany zasięg (np. poza zasięg metody, w której zostały zadeklarowane);

- **typy referencyjne** – to klasy i interfejsy. Kopiowanie typu referencyjnego (obiektu klasy lub interfejsu) daje w efekcie wiele referencji (odniesień) wskazujących na to samo miejsce w pamięci. Takie zmienne przechowywane są na stercie i usuwane z pamięci gdy wykonywane jest automatyczne czyszczenie pamięci zarządzanej sterty (ang. garbage collection).

Wszystkie typy danych (będące wartościami lub referencjami) wywodzą się ze wspólnej klasy bazowej `Object` z przestrzeni nazw `System`. Typy zdefiniowane w przestrzeni nazw `System` posiadają swoje synonimy w C# (Tabela 1.4).

Bardzo ważna i przydatna w praktyce programowania jest możliwość przechodzenia pomiędzy typami referencyjnymi a wartościami. Mówimy tu o zagadnieniu związanym z tzw. opakowywaniem i odpakowywaniem typu.

Opakowywanie (ang. boxing) to konwersja między typami będącymi wartością a typami referencyjnymi, np.:

```
short i = 5;       //zwykła wartość
object obiekt = i;// opakowanie wartości – zwracana
                  // jest referencja do utworzonego obiektu
                  // (obiekt musi być zainicjalizowany żeby
                  // istniała referencja do niego)
```

Odpakowywanie (ang. unboxing) to konwersja referencji do obiektu na typ będący wartością, np.:

```
short j = (short)obiekt;  //odpakowanie referencji oraz
                          //przywrócenie typu short
```

***Tabela 1.*4. Typy C# i ich obiektowe odpowiedniki**

Typ C#	Klasa z przestrzeni System
byte	System.Byte (8-bitowa liczba całkowita bez znaku)
sbyte	System.Byte (8-bitowa liczba całkowita ze znakiem)
short	System.Int16
int	System.Inr32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
float	System.Single
double	System.Double
decimal	System.Decimal
char	System.Char
string	System.String
bool	System.Boolean

1.2.5. Podstawy definiowania metod

W tym rozdziale podamy jedynie podstawy związane z definiowaniem metod w klasie. Szczegóły związane z tym tematem zostaną wyjaśnione w rozdziale 2.

Każda implementowana metoda musi być składową klasy lub struktury. W C# nie istnieją metody globalne (definiowane poza klasą jak np. w języku C++). Metody mogą przyjmować parametry lub nie, mogą zwracać wartość lub nie, mogą być statyczne lub nie. Metoda statyczna (`static`) może być wywoływana bezpośrednio z poziomu klasy i nie wymaga istnienia instancji obiektu tej klasy.

Statyczne są metody np.: `Main(), Console.Writeln("ABC")` itp.

Metoda musi mieć podany tzw. specyfikator dostępu, określający możliwość dostępu do metody (Tabela 1.5).

Tabela 1.5. Specyfikatory dostępu do metod w C#

Specyfikator	Dostęp
public	zewsząd
private	tylko w obrębie tej klasy (domyślnie)
protected	w obrębie klasy i jej potomnych (chroniony)
internal	w obrębie tej samej przestrzeni nazw (wewnętrzny)
protected internal	chroniony lub wewnętrzny

Metoda może posiadać (lub nie), zestaw parametrów. Każdy parametr może być przekazany do metody za pomocą jednego z modyfikatorów zestawionych w tabeli 1.6.

Tabela 1.6. Modyfikatory parametrów metody w C#

Modyfikator	Znaczenie
brak modyfikatora	domyślnie - parametr wejściowy metody
out	parametr wyjściowy metody
ref	Referencja do parametru we/wy
params	umożliwia przesłanie zmiennej liczby parametrów jako pojedynczego parametru; metoda może mieć tylko jeden modyfikator `params` i musi to być ostatni parametr metody

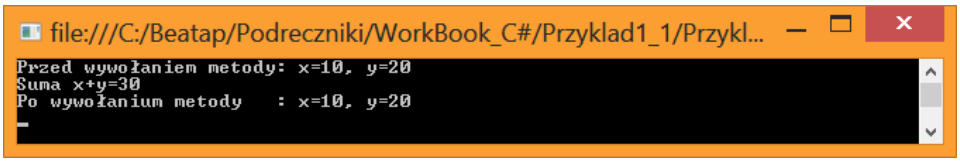
Definiując klasę aplikacji, zwykle istnieje konieczność zdefiniowania szeregu metod w tej klasie. Metody takie mogą być wywoływane w głównej (statycznej) metodzie `Main` aplikacji, o ile tylko będą miały odpowiedni specyfikator

dostępu i będą poprzedzone słowem kluczowym `static` (tylko statyczne metody klasy mogą być wywoływane w innej metodzie statycznej – patrz rozdział 2). Przykłady 1.7-1.10 pokazują zastosowanie odpowiednich modyfikatorów w pomocniczej metodzie `Suma` klasy `Program`.

W przykładzie 1.7, parametry *x, y* metody `Suma` są przekazane bez żadnego modyfikatora (dla typów prostych domyślnie przez wartość). Oznacza to, że metoda będzie tworzyła lokalne kopie swoich parametrów (*x* oraz *y*) i ewentualnej modyfikacji będą poddawane kopie tych parametrów, a to spowoduje, że wartości oryginalnych, aktualnych parametrów nie zostaną zmodyfikowane po wyjściu z metody, co pokazuje rysunek 1.5. W tym przykładzie wynik działania metody jest zwrócony za pomocą instrukcji `return`.

Przykład 1.7. Przekazywanie parametrów przez wartość

```
using System;
class Program
{
    public static int Suma(int x, int y)
    {
        int wynik = x + y;
        x = 100; y = 200;
        return wynik;
    }

    static void Main(string[] args)
    {
        int x = 10, y = 20;
        Console.WriteLine("Przed wywołaniem metody: x={0}, y={1}",
                          x, y);
        //wywołanie metody:
        Console.WriteLine("Suma x+y={0}", Suma(x, y) );
        Console.WriteLine("Po wywołanium metody   : x={0}, y={1}",
                          x, y);
        Console.ReadKey();
    }
}
```



Rys.1.5. Wynik działania programu z przykładu 1.7.

W przykładzie 1.8, parametry metody `Suma` są przekazane dokładnie tak samo jak w przykładzie 1.7, ale wynik działania metody jest umieszczony na liście parametrów i poprzedzony modyfikatorem `out`.Instrukcja `return` w takiej sytuacji została usunięta.

Przykład 1.8. Przekazywanie parametrów – słowo kluczowe out

```
using System;
class Program
{
    public static void Suma(int x, int y, out int wynik)
    {
        wynik = x + y;
    }
    static void Main(string[] args)
    {
        int x = 10, y = 20, wynik;
        //wywołanie metody (parametru out nie trzeba inicjalizować):
        Suma(x, y, out wynik);
        Console.WriteLine("Suma x+y={0}", wynik);
        Console.ReadKey();
    }
}
```

Rezultatem działania przykładu 1.8 będzie wydruk:

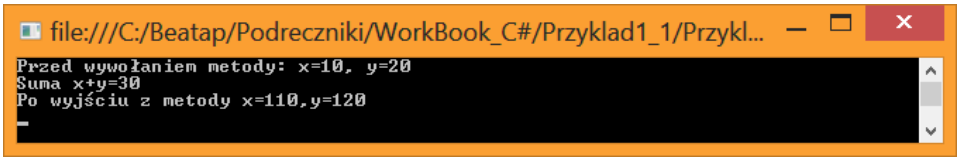
```
Suma x+y=30
```

W przykładzie 1.9 parametry `x` i `y` metody `Suma` są przekazane poprzez referencje (modyfikator `ref`). Oznacza to, że metoda będzie pracowała na swoich argumentach (nie tworzy kopii lokalnych `x` oraz `y`, ponieważ ma dostęp do oryginałów za pomocą otrzymanych referencji-adresów) i ewentualne modyfikacje ich wartości będą zapamiętane i dostępne po wyjściu z metody, co pokazuje rysunek 1.6.

Przykład 1.9. Przekazywanie parametrów – słowo kluczowe ref

```
using System;
class Program
{
    public static void Suma(ref int x, ref int y, out int wynik)
    {
        wynik = x + y; x += 100; y += 100;
    }
    static void Main(string[] args)
    {
        int x = 10, y = 20, wynik;
```

```
        Console.WriteLine("Przed wywołaniem metody: x={0}, y={1}",
                          x, y);
        //wywołanie metody (ref muszą być wcześniej inicjalizowane):
        Suma(ref x, ref y, out wynik);
        Console.WriteLine("Suma x+y={0}", wynik);
        Console.WriteLine("Po wyjściu z metody x={0},y={1}", x, y);
        Console.ReadKey();
    }
}
```

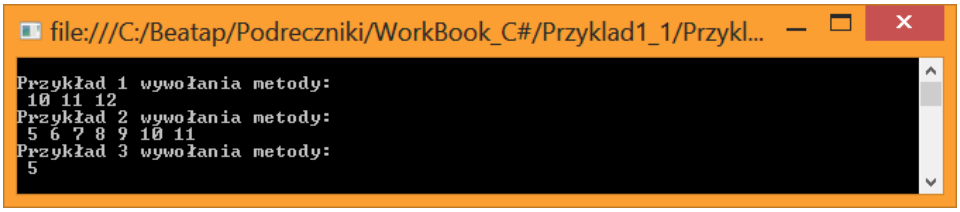


Rys.1.6. Wynik działania programu z przykładu 1.9.

Przykład 1.10 przedstawia sposób wykorzystania modyfikatora `params`, który pozwala przekazać metodzie zmienną liczbę argumentów. Wynik działania kodu z przykładu 1.10 pokazuje rysunek 1.7.

Przykład 1.10. Przekazywanie parametrów – słowo kluczowe params

```
using System;
class Program
{
    public static void TablicaInt(string info, params int[] liczby)
    {
        Console.WriteLine("\n" + info);
        for (int i = 0; i < liczby.Length; i++)
            Console.Write(" " + liczby[i]);
    }

    static void Main(string[] args)
    {
        //zdefiniowanie 3-elementowej tablicy:
        int[] tab = new int[3] { 10, 11, 12 };
        //różne sposoby wywołania metody TablicaInt:
        TablicaInt("Przykład 1 wywołania metody:",tab);
        TablicaInt("Przykład 2 wywołania metody:",5,6,7,8,9,10,11);
        TablicaInt("Przykład 3 wywołania metody:",5);
        Console.ReadKey();
    }
}
```



Rys.1.7. Wynik działania programu z przykładu 1.10.

1.3. Typy złożone

1.3.1. Typ wyliczeniowy

Typy wyliczeniowe stosowane są w przypadku kiedy wartości liczbowe wygodniej jest zastąpić zbiorem symbolicznych nazw. Deklarację typu wyliczeniowego w języku C# należy umieścić poza metodą klasy. Ma ona postać np.:

```
enum Dni { Pn, Wt, Śr, Czw, Pt, So, Nd };
```

Aby wykorzystać wartość typu wyliczeniowego nie trzeba tworzyć nowej zmiennej (instancji) typu a wystarczy odwołać się do konkretnego elementu przy użyciu operatora odwołania np.:

```
Console.WriteLine(Dni.Pn);
```

Domyślnie pierwszy element wyliczenia ma wartość 0, drugi 1 itd. Można to sprawdzić, dokonując rzutowania:

```
Console.WriteLine(Dni.Śr);
```

Możliwe jest również jawne nadanie numeru dla elementu np.:

```
enum Dni { Pn = 10, Wt, Śr, Czw = 50, Pt, So, Nd };
// Pn ma wartość 10, Wt 11 itd.  Czw ma wartość 50, Pt 51.
```

Domyślnie każdy element jest mapowany na liczbę całkowitą. Klasą podstawową typu jest `System.Enum`.

1.3.2. Struktury

Struktury to zorganizowany zbiór danych, niekoniecznie tego samego typu. Struktury w języku C#, podobnie jak klasy mogą posiadać pola, metody a nawet konstruktor. Strukturę można przekazać jako parametr metody.

Deklarację struktury oraz sposób nadania wartości polom struktury przedstawia przykład 1.11.

Przykład 1.11. Definicja i wykorzystanie struktury

```
using System;
struct Student
{
    // Pola struktury Student:
    public string Nazwisko;
    public long Tel;
    public byte Wiek;
}

class Program
{
    static void Main(string[] args)
    {
        Student s1, s2;
        s1.Nazwisko = "Jan Kowalski";
        s1.Wiek = 23;
        s1.Tel = 666666666;
        s2.Nazwisko = "Piotr Nowak";
        s2.Wiek = 43;
        s2.Tel = 555555555;
        Console.ReadKey();
    }
}
```

W odróżnieniu od np. języka C, w strukturze można zadeklarować konstruktor z parametrami, który musi służyć do przypisywania danych polom struktury. Przykład 1.12 pokazuje strukturę z definicją konstruktora oraz sposób nadania wartości polom struktury poprzez wykorzystanie jej konstruktora.

Przykład 1.12. Definicja struktury z konstruktorem

```
using System;
struct Student
{
    public string Nazwisko;
    public long Tel;
    public byte Wiek;
    // konstruktor struktury z parametrami nazw, telefon, w:
    // wartości tych parametrów zostaną przypisane polom zmiennej
    // typu struktury Student
    public Student(string nazw, long telefon, byte w)
    {
```

```
        Nazwisko = nazw;
        Tel = telefon;
        Wiek = w;
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Utworzenie zmiennej s1 typu Student poprzez konstruktor:
        Student s1 = new Student("Jan Kowalski", 666666666, 23);
        // Utworzenie zmiennej s2 typu Student poprzez konstruktor:
        Student s2 = new Student("Piotr Nowak", 555555555, 43);
        Console.ReadKey();
    }
}
```

1.3.3. Tablice

Deklaracja tablicy w C# jest postaci:

```
Typ_elementu[] Nazwa_tablicy;
```

Na przykład, deklaracja tablicy składającej się z elementów typu `int` wygląda następująco:

```
int[] tab1;
```

Przed użyciem tablicy należy zadeklarować, z ilu elementów ma się ona składać, np.:

```
tab1 = new int[5];
```

Deklarację tablicy i liczby jej elementów można zapisać krócej:

```
int[] tab2 = new int[15];
```

Po utworzeniu tablicy każdemu elementowi przypisywana jest domyślna wartość (dla typu `int` jest to 0). Po zadeklarowaniu tablicy można przypisać wartości dla konkretnego elementu, można to zrobić również w momencie deklaracji:

```
char[] znaki = new char[5] { 'W', 'i', 't', 'a', 'j' };
```

Ponieważ kompilator oblicza rozmiar tablicy po liczbie elementów umieszczonych pomiędzy klamrami, powyższą deklarację można uprościć do postaci:

```
char[] znaki = { 'W', 'i', 't', 'a', 'j' };
```

Tablicę dwuwymiarową definiujemy i odwołujemy się do jej elementów np.:

```
string[,] teksty = new string[5, 2];
teksty[0, 0] = "Pn";
teksty[1, 0] = "Wt";
```

Inicjalizacja danych wygląda analogicznie jak w przypadku tablicy jednowymiarowej:

```
int[,] x = new int[3, 3] { { 1,2,3 }, { 4,5,6 }, { 7,8,9 } };
```

lub krótko:

```
int[,] x = new int[,] { { 1,2,3 }, { 4,5,6 }, { 7,8,9 } };
```

albo po prostu:

```
int[,] x = { { 1,2,3 }, { 4,5,6 }, { 7,8,9 } };
```

Deklaracja tablicy trójwymiarowej ma analogicznie postać:

```
string[, ,] teksty3 = new string[2, 4, 2];
```

W języku C# istnieje specjalna pętla `foreach`, dedykowana operacjom na tablicach, której argumentem musi być tablica (Przykład 1.13).

Przykład 1.13. Zastosowanie pętli foreach

```
int[] x = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// dla każdego elementu i należącego do tablicy x wykonaj:
foreach (int i in x)
{  // kolejna iteracja – dostęp do kolejnego elementu i tablicy x:
   Console.WriteLine(i);
}
Console.WriteLine("Rozmiar tablicy x: " + x.Length);
```

Zmienna `i` z przykładu 1.13 nie może być zadeklarowana lub użyta we wcześniejszych fragmentach kodu. Rozmiar tablicy można uzyskać za pomocą konstrukcji `x.Length`.

Zastosowanie zwykłej pętli `for` ma jedną przewagę nad `foreach` - można w niej modyfikować wartości elementów (w `foreach` nie można – przy próbie modyfikacji zgłaszany jest błąd kompilacji). Zastosowanie pętli `for` dla tablicy pokazuje przykład 1.14.

Przykład 1.14. Pętla for – konieczna przy modyfikacjach wartości

```
for (int i = 0; i < x.Length; i++)
{  x[i] = 77;
   Console.WriteLine(x[i]);
}
```

Pętlę `foreach` dla tablic wielowymiarowych i analogiczną pętlę `for` przedstawia przykład 1.15.

Przykład 1.15. Pętla foreah i for dla tablic wielowymiarowych

```
int[,] x = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
foreach (int i in x)
{
   Console.WriteLine(i);
}
//analogiczna petla for:
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 3; j++)
   {
      Console.WriteLine(x[i, j]);
   }
```

W C# można tworzyć tablice, które zawierają kolejne tablice (tablice tablic), np.:

```
int[][] t = new int[2][]; //tablica t zawiera kolejne dwie
// tablice o nieokreślonej jeszcze liczbie elementów
```

Te dwie tablice również muszą zostać utworzone np.:

```
t[0] = new int[5];
t[1] = new int[10];
```

Przypisywanie danych do takich tablic odbywa się następująco:

```
// przypisanie wartości do elementu nr 2 tablicy pierwszej:
t[0][1] = 100; // 0 - numer tablicy (pierwsza)
// przypisanie wartości do elementu 10 tablicy drugiej:
t[1][9] = 1; // 1 – numer tablicy (druga)
```

Przypisanie wartości do elementów w tablicach tablic może być zrealizowane następująco:

```
int[][] t = new int[][] {
        new int[] {1, 2}, // przecinek!
        new int[] {1, 2, 3}
        }; // średnik!
```

lub krótko:

```
int[][] t = {
        new int[] {1, 2}, // przecinek!
        new int[] {1, 2, 3}
        }; // średnik!
```

Elementami tablicy mogą być również typy złożone, na przykład struktury. Przykład 1.16 przedstawia deklarację tablicy struktur `Student` oraz sposób wykorzystania pętli `foreach` dla takiej tablicy.

Przykład 1.16. Tablica struktur i pętla foreach

```
using System;
struct Student
{
    public string Nazwisko;
    public long Tel;
    public byte Wiek;
    public Student(string nazw, long telefon, byte w)
    {
        Nazwisko = nazw;
        Tel = telefon;
        Wiek = w;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Student s1 = new Student("Jan Kowalski", 666666666, 23);
        Student s2 = new Student("Piotr Nowak", 555555555, 43);
        // deklaracja i inicjalizacja tablicy studentów:
        Student[] studenci = { s1, s2 };
        // wyświetlenie informacji o studentach z tablicy:
        foreach (Student i in studenci)
            Console.WriteLine(i.Nazwisko + " " + i.Wiek);
        Console.ReadKey();
    }
}
```

1.3.4. Klasa Array

Wszystkie tablice w C# wywodzą się od wspólnej klasy podstawowej `System.Array`. Klasa `Array` dostarcza podstawowe mechanizmy do manipulacji elementami tablicy. Dzięki metodom tej klasy można pobrać liczbę elementów w tablicy, posortować ją czy przeszukać.

Najważniejsze właściwości klasy `Array` to:

- `Length` - zwraca aktualną liczbę elementów tablicy,
- `LongLength` - zwraca 64-bitową wartość określającą rozmiar wszystkich elementów w przypadku tablic wielowymiarowych,
- `Rank` - zwraca liczbę wymiarów danej tablicy.

Najważniejsze metody klasy `Array` to:

- `Clear()` – umożliwia wyczyszczenie tablicy (ustawia każdemu elementowi wartość 0 lub `null` w zależności od jego typu). Przyjmuje trzy parametry: nazwę tablicy, numer indeksu (od którego ma rozpocząć czyszczenie) i zasięg tego procesu, np. czyszczenie całej zawartości: `Array.Clear(tab, 0, tab.Length);` Metoda nie zmienia rozmiaru czyszczonej tablicy. Do elementów wyczyszczonej tablicy ponownie można przypisywać wartości;
- `Clone()` – zwraca kopię tablicy, z której została wywołana np.: `tab1 = tab.Clone();` Metoda zwraca dane w postaci ogólnego typu `object`, dlatego należy dokonać rzutowania na właściwy typ;
- `Copy()` – inny sposób tworzenia kopii tablicy - umożliwia dodatkowo określenie rozmiarów kopiowania, tj. ile elementów zostanie skopiowanych. Pierwszym parametrem metody jest tablica źródłowa (kopiowana), drugim – tablica, do której skopiowane zostaną elementy, trzecim – liczba kopiowanych elementów;
- `Find()` – umożliwia przeszukanie całej tablicy w celu znalezienia pierwszego elementu spełniającego określone kryterium;
- `FindAll()` – umożliwia przeszukanie całej tablicy w celu znalezienia wszystkich określonych elementów;
- `FindLast()` – działa podobnie jak `Find()`, tylko szuka ostatniego wystąpienia danego elementu - nie kończy pracy, gdy znajdzie pierwszy element pasujący do kryteriów;
- `GetLength()` – zwraca liczbę elementów w tablicy. W parametrze tej metody należy podać numer wymiaru, z którego liczba elementów ma być pobrana (dla tablicy jednowymiarowej jest to 0);
- `GetLongLength()` – analogicznie do `GetLength()`, ale zwraca dane w postaci liczby typu `long`;
- `GetLowerBund()`, `GetUpperBund()` – zwraca numer odpowiednio najmniejszego lub największego indeksu w tablicy. Obie metody mogą działać na tablicach wielowymiarowych (w parametrze należy podać indeks wymiaru);
- `GetValue()` – zwraca wartość danego elementu tablicy, np. `Tab.GetValue(1)` co jest równoważne konstrukcji `Tab[1];`
- `Initialize()` – powoduje przypisanie każdemu elementowi pustej wartości (0, `null` w zależności od typu), np.: `Tab.Initialize();`
- `IndexOf()` – zwraca numer indeksu na podstawie podanej wartości elementu;

- `Resize()` – zmienia rozmiar danej tablicy; pierwszy parametr metody to nazwa tablicy poprzedzona słowem kluczowym `ref`, drugi parametr to nowy rozmiar tablicy;
- `SetValue()` – umożliwia nadanie wartości danemu elementowi tablicy. To samo działanie można zrealizować przy pomocy operatora przypisania`;`

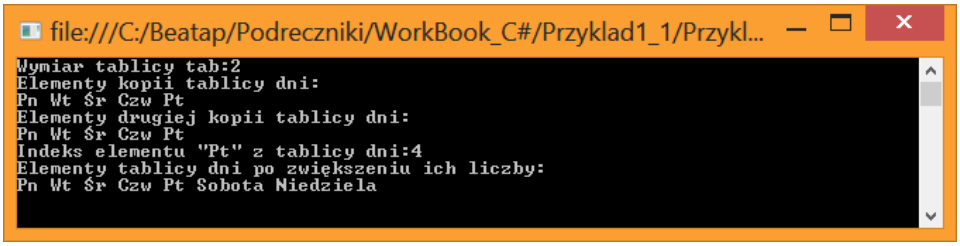
Przykład 1.17 przedstawia zastosowania wybranych metod klasy `Array.`

Przykład 1.17. Wykorzystanie właściwości i metod klasy Array

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int[,] tab = new int[3, 2] { { 1, 2 }, { 1, 2 }, { 1, 2 } };
        //wyświetl wymiar tablicy:
        Console.WriteLine("Wymiar tablicy tab:"+tab.Rank);
        //wyczyść (wyzeruj)zawartość tablicy od pierwszego do
        //ostatniego elementu:
        Array.Clear(tab, 0, tab.Length);
        string[] dni = new string[] {"Pn", "Wt", "Śr", "Czw", "Pt"};
        //utwórz kopię tablicy dni:_
        string[] kopiadni = (string[]) dni.Clone();
        //wyświetl elementy tablicy kopiadni:
        Console.WriteLine("Elementy kopii tablicy dni:");
        foreach (string element in kopiadni)
        {
            Console.Write(element+" ");_
        }
        Console.WriteLine();
        string[] drugakopiadni = new string[dni.Length];_
        // inny sposób tworzenia kopii tablicy:
        Array.Copy(dni, drugakopiadni, dni.Length);
        Console.WriteLine("Elementy drugiej kopii tablicy dni:");
        foreach (string element in drugakopiadni)
        {
            Console.Write(element + " ");
        }
        Console.WriteLine();
        Console.WriteLine("Indeks elementu \"Pt\" z tablicy dni:"+
                          Array.IndexOf(dni, "Pt"));
        //zmień rozmiar tablicy dni:
        Array.Resize(ref dni, dni.Length + 2);
        //ustaw wartość szóstego elementu tablicy dni:
        dni.SetValue("Sobota", 5);
```

```
        //ustaw wartość siódmego elementu tablicy dni:
        dni[6] = "Niedziela";
        Console.WriteLine("Elementy tablicy dni po zwiększeniu ich
                          liczby:");
        foreach (string element in dni)
        {
            Console.Write(element + " ");
        }
        Console.WriteLine();
        Console.ReadKey();
    }
}
```

Wynik działania przykładu 1.17 przedstawia rysunek 1.8.



Rys.1.8. Wynik działania programu z przykładu 1.17.

Przy pracy z tablicą trzeba pamiętać, że w języku C# każda tablica jest reprezentowana przez obiekt klasy `Array`, a zatem jest to typ referencyjny. Pamiętajmy, że w celu utworzenia np. kopii `b` tablicy `a` nie możemy zastosować zwykłego oparatora przypisania (tzn.: `b=a`), gdyż w ten sposób tworzymy jedynie referencję do tablicy `a`, zatem wskazujemy na to samo miejsce w pamięci. Skutkuje to tym, że wszystkie ewentualne modyfikacje wykonywane na elementach tablicy `b`, są w efekcie realizowane na elementach tablicy `a`. Faktyczną kopię tablicy (zlokalizowaną w innym miejscu pamięci) można utworzyć jedynie tak jak pokazano w przykładzie 1.17 stosując metody `Copy()` lub `Clone()` klasy `Array`.

Tworząc metody o parametrach tablicowych pracujemy na referencji do tablicy. W metodzie nie jest tworzona lokalna kopia parametru będącego tablicą, wobec czego wszelkie modyfikacje tablicy w metodzie będą zachowane również po wyjściu z metody. **Parametry tablicowe są domyślnie przekazywane przez referencje**. Przykład 1.18 definiuje cztery metody z parametrami tablicowymi: dwie metody do wyświetlania elementów tablicy typu `string` i `int` oraz dwie analogiczne metody wczytujące elementy do tablicy. Warto zwrócić uwagę, że

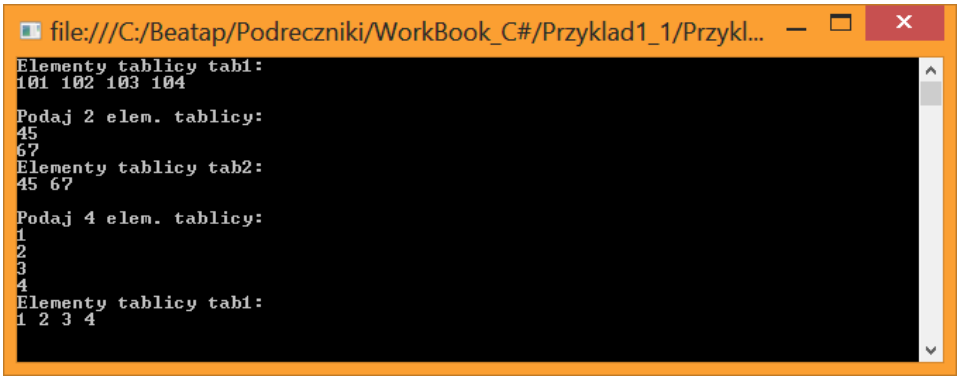
obie metody nazywają się tak samo, ale różnią się typem parametru. W takim przypadku mamy do czynienia z przeciążeniem lub przeładowaniem nazw metod (więcej informacji na ten temat czytelnik znajdzie w rozdziale 2).

Przykład 1.18. Tablice jako parametry metod

```
using System;
class Program
{   //pierwsza metoda do wyświetlania elementów tablicy typu int
    public static void DrukujTab(int[] t)
    {
        for (int i = 0; i < t.Length; i++)
            Console.Write(t[i] + " ");
        Console.WriteLine();
    }
    //druga metoda do wyświetlania elementów tablicy typu string
    public static void DrukujTab(string[] t)
    {
        for (int i = 0; i < t.Length; i++)
            Console.Write(t[i] + " ");
        Console.WriteLine();
    }
    //pierwsza metoda do czytania elementów tablicy typu string
    public static string[] CzytajTab1(int n)
    {   //n - liczba elementów tablicy
        //w metodzie rezerwujemy pamięć na n-elementów typu string:
        string[] t = new string[n];
        Console.WriteLine("\nPodaj " + n + " elem. tablicy:");
        for (int i = 0; i < t.Length; i++)
              t[i] = Console.ReadLine();
        //jako wynik przekazujemy wczytaną tablicę:
        return t;
    }
    //druga metoda do czytania elementów tablicy typu int
    //konieczna jest konwersja łańcucha string na int
    //za pomocą odpowiedniej metody z klasy Convert
    public static int[] CzytajTab2(int n)
    {
        //n - liczba elementów tablicy
        //w metodzie rezerwujemy pamięć na n-elementów typu string:
        int[] t = new int[n];
        Console.WriteLine("\nPodaj " + n + " elem. tablicy:");
        for (int i = 0; i < t.Length; i++)
            t[i] = Convert.ToInt32(Console.ReadLine());
        //jako wynik przekazujemy wczytaną tablicę:
        return t;
    }
    static void Main(string[] args)
```

```
    {
        int[] tab1 = { 101, 102, 103, 104 };
        string[] tab2;
        Console.WriteLine("Elementy tablicy tab1:");
        DrukujTab(tab1);
        tab2 = CzytajTab1(2);
        Console.WriteLine("Elementy tablicy tab2:");
        DrukujTab(tab2);
        tab1 = CzytajTab2(4);
        Console.WriteLine("Elementy tablicy tab1:");
        DrukujTab(tab1);
        Console.ReadKey();
    }
}
```

Wynik działania przykładu 1.18 przedstawia rysunek 1.9.



Rys.1.9. Wynik działania programu z przykładu 1.18.

Podsumowując wiadomości o tablicach języka C#:

- wszystkie tablice wywodzą się od wspólnej klasy podstawowej `System.Array`;
- formalnie tablica jest zbiorem wskaźników do danych tego samego typu dostępnych za pomocą indeksu liczbowego;
- indeksowanie tablic rozpoczyna się od zera (można to zmienić za pomocą metody `CreateInstance` z `System.Array`);
- rozmiar tablicy jest ustalany w chwili jej tworzenia a nie deklarowania – deklarując tablicę o stałym rozmiarze początkowym trzeba zastosować operator `new` (chyba, że się poda listę wartości początkowych);

- każdy element tablicy C# (w przeciwieństwie do C++) jest automatycznie ustawiany na wartość domyślną (0, null, false itp. w zależności od typu elementów).

1.3.5. Klasa String i Char

Łańcuch znakowy w języku C# jest reprezentowany za pomocą prostego typu `string`, który z kolei jest synonimem klasy `String` z przestrzeni nazw `System`. Dzięki temu na łańcuchach możemy wykonywać różne operacje za pomocą metod tej klasy.

Podstawowe metody klasy `System.String` to:

- `Concat()` – konkatenacja (połączenie) łańcuchów,
- `CompareTo()` – porównanie łańcuchów,
- `Copy()` – kopiowanie łańcucha,
- `Format()` – sformatowanie łańcucha,
- `Insert()` – wstawienie łańcucha do innego łańcucha,
- `Remove(), Replace()` – usunięcie lub zamiana wskazanego łańcucha,
- `Join(), Trim()` – połączenie lub rozdzielenie łańcuchów,
- `ToUpper(), ToLower()` – zmiana liter w łańcuchu na duże/małe, itp.

Platforma .NET w całości wspiera system kodowania znaków Unicode, standard obejmujący wszystkie języki świata i eliminujący problem tzw. krzaczków. Łańcuchy (m.in. języka C#), są kodowane w systemie Unicode a to oznacza, że każdy znak zajmuje 2 bajty pamięci. Kodowanie Unicode można również stosować w kodzie źródłowym, używając np. polskich znaków w nazwie zmiennych. Takie podejście ułatwia pisanie aplikacji wielojęzykowych – unika się problemu kodowania i ewentualnej konwersji systemu znaków.

Przykład 1.19 demonstruje zastosowania wybranych metod klasy String.

Przykład 1.19. Zastosowanie metod klasy String

```
string s1= "teścik1";
System.String s2= "teścik2";
string s3;
s3 = s1+s2;
//porównaj łańcuchy s1 i s2:
if (s2.CompareTo(s1)==0) Console.WriteLine("{0}={1}",s1,s2);
else Console.WriteLine("{0}!={1}", s1, s2);
//zastąp łańcuch "teścik" w s3 nowym łańcuchem "test"
s3.Replace("teścik","test");
Console.WriteLine(s3);
```

Jeżeli znaki sterujące (takie jak cudzysłów, apostrof, przejście do nowego wiersza, tabulacja itp.) mają się pojawić w łańcuchu wynikowym należy poprzedzić je znakiem *backslash* np.: \' \" \\ \a \b \n \f \r \t \u \v \0. Istnieje też możliwość zastosowania notacji dosłownej (ang. verbatim strings), tzn. znaki sterujące nie są przetwarzane w napisach słownych jeśli łańcuchy znakowe poprzedzone są prefiksem @ np.:

```
string s4 = @"\n\t Plik: 'C:\WykładyC#\test.cs' ";
Console.WriteLine(s4);
```

Po uformowaniu łańcucha znakowego (obiektu klasy `String`) jego wartość nie może już być modyfikowana ponieważ obiekty `string` w C# są niezmienne. Metody klasy `System.String` w rzeczywistości nie modyfikują łańcucha znaków, ale zwracają jego zmodyfikowaną kopię, np.:

```
System.String s1 = "tekscik1";
System.String s1Upper = s1.ToUpper();
//nie jest tu modyfikowana zawartość bufora ale zwracana
//jest nowa kopia bufora zawierająca wielkie litery
```

Przestrzeń nazw `System.Text`, definiuje inną klasę `String.Builder` – wszystkie modyfikacje instancji obiektu tej klasy oddziałują bezpośrednio na bufor a tym samym operacja jest przeprowadzana sprawniej. W przypadku tworzenia aplikacji intensywnie wykorzystującej dane tekstowe (np. edytor) użycie klasy `System.Text.StringBuilder` poprawia wydajność aplikacji. Przykład 1.20 przedstawia wykorzystanie klasy `StringBuilder` z podprzestrzeni nazw `System.Text`.

Przykład 1.20. Zastosowanie klasy StringBuilder

```
//utwórz obiekt StringBuilder i zmodyfikuj bufor:
StringBuilder sb = new StringBuilder("bufor ");
sb.Append("i dodajmy coś jeszcze");
//zapisz bufor wielkimi literami i przekształć go
//na stały łańcuch czyli obiekt typu string:
string stringfinal = sb.ToString().ToUpper();
Console.WriteLine(stringfinal);
```

Wykonywanie operacji na pojedynczych znakach realizowane jest za pomocą obiektów klasy `System.Char`. Obiekt taki przechowuje pojedynczy znak typu Unicode. Wartość zmiennej typu `char` musi być zawarta w apostrofach np.:

```
char znak = 'A';
char c4 = '\t'; // znak tabulacji
```

Wartość przypisywana do zmiennej typu `char` może być zapisana również w formie heksadecymalnej, można też skorzystać z operatora rzutowania wartości całkowitej na typ `char` np.:

```
char c1 = '\x0058';
char c2 = (char)65;
```

Do podstawowych metod klasy `Char` należą metody:

- `IsControl()` – zwraca `true`, jeżeli znak zawarty w zmiennej jest znakiem kontrolnym (np. \t czy \n);
- `IsDigit()` – zwraca `true`, jeżeli znak zawarty w zmiennej jest liczbą;
- `IsLetter()` – zwraca `true`, gdy znak zawarty w zmiennej jest literą;
- `IsLower()` – zwraca `true`, jeżeli znak zawarty w zmiennej jest małą literą;
- `IsSymbol()` – zwraca `true`, jeżeli znak zawarty w zmiennej jest symbolem;
- `IsWhiteSpace()` – zwraca `true`, jeżeli znak zawarty w zmiennej można zaliczyć do tzw. białych znaków (spacje, znaki nowej linii itp.).

1.4. Zadania do samodzielnego rozwiązania

Zadanie 1.1.

W środowisku Visual C# utworzyć nowy projekt: *FILE->New project –> Console Application*. Uzupełnić wygenerowany schemat programu o metody klasy *Console* jak pokazano poniżej:

```
static void Main(string[] args)
{ Console.WriteLine("Witaj C#");
  Console.ReadKey();
}
```

Następnie należy:

a) skompilować i uruchomić program,
b) zmodyfikować program tak aby wczytywał imię użytkownika a następnie w powitaniu wyświetlał pobrane imię, program ma dodatkowo pobierać wiek użytkownika, przeprowadzić konwersję na wartość całkowitą (klasa **Convert**) oraz wyświetlić na konsoli dodatkowo wiek użytkownika.

UWAGA!

Do wyświetlania informacji zastosować oba sposoby:

a) znany z języka C++/Java operator konkatenacji łańcuchów (+),
b) sposób formatowania C# ({} i argumenty).

Zadanie 1 2.

Napisać program, który wczytuje liczbę całkowitą $0<n<10$ i drukuje trójkąt liczb:

a)
1
22
333
4444
. . .
nnnnnn...n

b)
1
12
123
1234
. . .
1234...n

W przypadku podania wartości spoza przedziału program powtarza wczytanie liczby, aż do momentu podania poprawnej wartości.

Zadanie 1.3.

Napisać program, który będzie rysował trójkąt równoramienny złożony z gwiazdek. Wysokość trójkąta stanowiąca liczbę wierszy będzie daną wejściową programu.
Ostatni wiersz trójkąta powinien stykać się z lewą krawędzią okna. Ilość wierszy ograniczyć do 25.
Np. Ile wierszy? 5

```
    *
   ***
  *****
 *******
*********
```

Zadanie 1.4.

Napisać program, który za pomocą pętli `for` obliczy i wyświetli tabliczkę mnożenia liczb od 1 do 10.

Zadanie 1.5.

Należy:

a) napisać program, który ustali szczęśliwy numerek na dowolną wartość całkowitą >=0 oraz poprosi użytkownika o podanie liczby. Jeśli użytkownik poda liczbę różną od szczęśliwego numerka wyświetlić komunikat „Spróbuj jeszcze raz” i proces zgadywania powtarzać w pętli aż do uzyskania poprawnej liczby (pętla do while). Jeśli użytkownik zgadnie szczęśliwy numerek wyświetlić komunikat „Brawo”;

b) zmodyfikować program tak, aby zliczał i dodatkowo wyświetlał informację za którym razem podano poprawną odpowiedź;
c) zmodyfikować program tak, aby możliwa było tylko 3- krotna próba i po trzecim razie – program kończy działanie komunikatem „Niestety do trzech razy sztuka”; program ma dodatkowo generować losową liczbę z przedziału <1,9>;
d) po realizacji punktu c) pytać użytkownika: "Gramy dalej (t/n)?" i jeśli użytkownik potwierdzi dalszą gre to ponownie realizować zadania z punktu c (zastosować pętlę while).

UWAGA!
Losowanie liczby w .NET realizuje klasa **`Random`**:

```
// tworzenie nowej instancji (obiektu) klasy:
Random Oran = new Random();
// wylosowanie liczby z zakresu 0-999:
int x = Oran.Next(1000);
```

Zadanie 1.6.

Napisać program realizujący komputerową wersję gry w **Oczko.** Gra polega na dobieraniu (generowanych losowo w programie) kolejnych liczb z przedziału <2,11> dotąd, aby osiągnąć sumę posiadanych liczb jak najbliższą (ale nie większą niż) 21. Gracz otrzymuje kolejne losowe liczby dotąd, aż sam zdecyduje, że kończy dobieranie lub otrzyma wynik 21 lub większy. Suma większa lub równa 22 oznacza przegraną. Wyjątkiem od tej reguły jest perskie oczko (dwa asy – wylosowane dwie liczby 11). Perskie oczko zawsze oznacza wygraną.

Zadanie 1.7.

Napisać program, który znajduje w podanym z klawiatury ciągu liczb zawartych pomiędzy 0 a 20 największą i najmniejszą oraz poda ile razy każda z tych dwóch liczb wystąpiła. Zakładamy, że ciąg o nieznanej z góry długości będzie zakończony liczbą -1. W rozwiązaniu nie wolno stosować tablic. W razie podania liczby spoza przedziału nie należy jej uwzględniać a należy poprosić o powtórne podanie liczby z właściwego przedziału <0, 20>. Wynik działania programu powinien wyglądać następująco:

```
Podaj liczbę (-1 kończy program): 12
Podaj liczbę (-1 kończy program): 8
Podaj liczbę (-1 kończy program): 18
Podaj liczbę (-1 kończy program): 7
Podaj liczbę (-1 kończy program): 11
Podaj liczbę (-1 kończy program): 12
```

```
Podaj liczbę (-1 kończy program): 7
Podaj liczbę (-1 kończy program): 9
Podaj liczbę (-1 kończy program): -1

Maksimum równe 18, liczba powtórzeń 1.
Minimum równe 7, liczba powtórzeń 2.
```

Zad 1.8.

Napisać prostą grę - komputer losuje liczbę, a zadaniem użytkownika jest odgadnięcie tej prawidłowej. Gra dwóch użytkowników (zgadują na zmianę), a po każdej próbie program podpowiada, czy należy celować wyżej, czy niżej.
Należy:

a) pobrać od użytkowników ich imiona i przypisać je zmiennym *gracz1* i *gracz2* typu *string*;
b) wylosować liczbę z przedziału <0,100>;
c) zadeklarować zmienną logiczną, w której należy zapisać informację o ewentualnym końcu gry (np. *bool koniec=false;*) oraz zmienną *gracz* z informacją o aktualnym graczu (*gracz=gracz1* lub *gracz=gracz2*);
d) w pętli pobierać strzały graczy i podpowiadać za pomocą stosownych komunikatów np. *"Za mało", "Za dużo", "Trafił gracz: imię."*.

Zadanie 1.9.

Zmodyfikować program z zadania 1.8 tak aby korzystał z pomocniczych metod:

a) metody pobierającej imiona graczy;
b) metody do wyświetlania odpowiedniego komunikatu dla gracza *("Za mało", "Za dużo" lub "Trafił gracz: imię."*).

Zadanie 1.10.

Zdefiniować metodę, która dla zadanej wartości całkowitej n>0 wyznacza jej silnię, tzn. *f(n)=n!*. Napisać program, który prosi użytkownika o podanie dwóch liczb całkowitych ***n>=0, r>=0, n>=r***. W przypadku podania błędnych wartości – powtarzać proces pobrania danych aż do momentu uzyskania prawidłowych liczb.

Funkcję wykorzystać do obliczenia wartość wyrażenia (symbol Newtona):

$$W(n,r) = \binom{n}{r} = \frac{n!}{r!(n-r)!} \ .$$

UWAGA!
Każdą silnię (***n!, r!, (n-r)!***)obliczać korzystając z metody pomocniczej (z odpowiednimi parametrami) a symbol Newtona wyznaczyć za pomocą tylko jednego wyrażenia.

Zadanie 1.11.

Napisać metodę, która będzie otrzymywać jako argumenty dwie liczby rzeczywiste i jeden znak a następnie wykona na tych liczbach operacje arytmetyczną odpowiadającą znakowi i zwróci jej wynik. Mają być realizowane operacje: +, -, *, / . Każdy inny znak ma powodować wykonanie dodawania.
Napisać program, który skorzysta z tej funkcji do wykonania wszystkich czterech operacji na dwóch liczbach a, b≠0 wczytanych z klawiatury.

Zadanie 1.12.

Napisać:

a) metodę, która oblicza współczynniki A, B, C prostej Ax+By+C=0 przechodzącej przez dwa dane punkty K(x1,y1), L(x2,y2) :
   A=y2-y1,
   B=x1-x2,
   C=y1 x2 – y2 x1;
b) metodę, która oblicza odległość punktu S(p,q) od prostej Ax+By+C=0
   $d = \frac{|Ap + Bq + C|}{\sqrt{A^2 + B^2}}$;
c) program, który wczytuje współrzędne trzech punktów D(x1,y1), E(x2,y2) i F(x3,y3) oraz korzystając z powyższych metod oblicza wszystkie wysokości trójkąta DEF (jeśli DEF tworzy trójkąt).

Zadanie 1.13.

Napisać metodę, która zamienia wartości dwóch swoich argumentów całkowitych. Przetestować działanie metody.

Zadanie 1.14.

Napisać:

a) metodę, która wczytuje *n* - elementów całkowitych do tablicy ***t*;**
b) metodę, która wyznacza sumę elementów z tablicy ***t***;
c) metodę, która sprawdza, czy ciąg z tablicy ***t*** jest ciągiem rosnącym;
d) program, który korzystając z powyższych metod wczytuje liczbę całkowitą $n>0$ i wyrazy ciągu $a_0, a_1,..., a_{n-1}$ do tablicy ***a*** oraz sprawdza, czy dany ciąg jest ciągiem rosnącym. Jeśli tak to drukuje sumę elementów tego ciągu.

2. Programowanie obiektowe w C#

2.1. Wstęp

Biblioteka klas, dostępna w pakiecie .NET Framework, składa się z klas, typów i stałych. Elementy biblioteki są pogrupowane w hierarchiczną strukturę, czyli przestrzeń nazw. Nazwy przestrzeni nazw dostarczanych przez Microsoft zawsze zaczynają się od słowa `System` lub `Microsoft`. Przestrzenie nazw wprowadzają pewną hierarchię – programista może udostępnić bibliotekę, np. `Grafika.dll`, która zawiera przestrzenie nazw `Grafika.Figury3D` oraz `Grafika.Figury3D.Kwadryki` oferujące dodatkowe klasy.

Podstawowe przestrzenie nazw dostępne w .NET Framework zestawiono w tabeli 2.1.

Tabela 2.1. Podstawowe przestrzenie nazw w .NET

Przestrzeń nazw	Opis
System	klasy używane w każdym programie — np.String
System.IO	obsługa strumieni we/wy
System.Threading	klasy dla aplikacji wielowątkowych
System.Security	klasy związane z bezpieczeństwem oraz kryptografią
System.Net	klasy umożliwiające programowanie sieciowe oraz dostęp do wielu usług m.in. DNS czy HTTP
System.Data	klasy związane z dostępem do baz danych
System.Windows.Forms	klasy biblioteki Windows Forms
System.Web	klasy oraz inne przestrzenie związane z dostępem do sieci, usługami sieciowymi oraz protokołem HTTP

Klasa jest podstawą programowania na platformie .NET. Biblioteka klas .NET Framework (ang. .NET Framework Class Library - FCL) stanowi zestaw setek klas, bibliotek, interfejsów i typów, z których możemy korzystać programując w C#. Możemy też budować własne klasy, ukierunkowane na konkretną funkcjonalność.

Niniejszy rozdział opisuje elementy języka C# bezpośrednio związane z możliwościami programowania obiektowego. Rozdział 2.2 przedstawia podstawowe pojęcia związane z definiowaniem klas, natomiast rozdział 2.3 prezentuje bardziej zaawansowane mechanizmy programowania obiektowego dostępne w C#.

2.2. Podstawy programowania obiektowego

Klasa to typ danych. Definiując własną klasę (nowy typ danych) definiujemy jej składniki, którymi sa pola i metody. Metody (funkcje klasy) realizują zwykle różne operacje na swoich polach. Definiowanie klasy możemy utożsamiać z tworzeniem pól formularza, którym później będziemy przypisywać wartości. Obiekt jest zmienną typu klasy (instancją klasy). Tworząc obiekt danej klasy, zwykle polom klasy przypisujemy konkretne wartości, czyli wypełniamy, przygotowany wcześniej formularz, konkretnymi danymi.

2.2.1. Klasa i obiekt

Klasę deklaruje się słowem kluczowym `class`. Klasa musi mieć nazwę, może być pusta, lecz należy użyć klamer { }. Klasa może być umieszczona w przestrzeni nazw, ale nie musi np.:

```
class Nowa {}
```

Tworzenie instancji klasy (obiektu) jest realizowane za pomocą operatora `new`, jak demonstruje przykład 2.1.

Przykład 2.1. Deklaracja klasy Nowa poza przestrzenią nazw

```
using System;
// deklaracja klasy poza przestrzenią nazw:
class Nowa { }
namespace Przyklad2
{   // klasa z metodą Main() w przestrzeni nazw:
    class Program
    {
        static void Main(string[] args)
        {   // utworzenie instancji (obiektu) klasy Nowa:
            Nowa no = new Nowa();
            // utworzenie instancji (obiektu) klasy Int32:
            System.Int32 i = new System.Int32();
            // utworzenie obiektu klasy Int32 można zrobić prościej:
            int j;
        }
    }
}
```

Środowisko .NET Framework dopuszcza możliwość zagnieżdżania typów, w tym klas, np.:

```
class Figura
{
    public class Punkt { }
}
```

Tworzenie instancji dla zagnieżdżanego obiektu ma postać:

```
Figura.Punkt fp = new Figura.Punkt();
```

Pola klasy

Zmienne mogą być umieszczone w klasach lub w ich metodach. Zmienna umieszczona w klasie (nie wewnątrz metody) nazywana jest polem klasy. Pola to zatem zmienne lub stałe, deklarowane na użytek klasy lub udostępnione na zewnątrz do użytku programisty. Dostęp do zawartości pól jest realizowany za pomocą operatora odwołania: '**.**' (kropka) (Przykład 2.2).

Przykład 2.2. Deklaracja klasyFigura z polem info

```
using System;
class Figura {
    //publiczne pole klasy:
    public String info = "Klasa Figura";_
}
class Program {
    static void Main(string[] args)
    {   //utworzenie obiektu f1 klasy Figura:
        Figura f1 = new Figura();
        //odwołanie się do pola info z obiektu f1 klasy Figura
        Console.Write(f1.info);
        Console.Read();
    }
}
```

Metody klasy

Język C# nie umożliwia deklarowania funkcji poza klasą - wszystkie funkcje są metodami a w klasie może być wiele metod. Deklaracja metody ma ogólną postać:

```
typ_zwracanego_wyniku nazwaMetody(lista_parametrow)
    {
        //ciało metody
        return wynik;
    }
```

lub w przypadku gdy metoda nie zwraca żadnego wyniku (co sygnalizuje specjalny typ `void`), pomijana jest instrukcja `return`:

```
void nazwaMetody(lista_parametrow)
    {
        //ciało metody
    }
```

Typ `void` jest aliasem dla typu .NET Framework `System.void`.

Przeciążanie metod

Język C# nie pozwala nadawać domyślnych wartości parametrów dla metod (w przeciwieństwie np. do C++), rozwiązaniem jest wykorzystanie mechanizmu przeciążania metod.

Przeciążanie (lub inaczej przeładowywanie) metod to technika często spotykana w środowisku .NET Framework. W klasie mogą istnieć metody o tej samej nazwie, jeśli tylko posiadają różną liczbę parametrów lub parametry są różnych typów. Na przykład przeciążenie metody obliczającej sumę wartości swoich parametrów można zadeklarować jak w przykładzie 2.3. Przy takich definicjach metody przeciążonej `suma`, w zależności od tego jakiego typu i ile parametrów dostarczymy – będzie wywołana właściwa metoda.

Przykład 2.3. Deklaracja metod przeciążonych

```
using System;
class Test{
    //pole klasy:
    public String info = "Klasa Test";
    // metoda suma z jednym parametrem int:
    public int suma(int a)
    {   return a + a;
    }
    // metoda suma z dwoma parametrami int:
    public int suma(int a, int b)
    {   return a + b;
    }
    // metoda suma z dwoma parametrami double:
 public double suma(double a, double b)
    {   return a + b;
    }
}
class Program {
    static void Main(string[] args)
    {   //utworzenie obiektu t1 klasy Test:
        Test t1 = new Test();
        //odwołanie się do przeciążonej metody suma:
        t1.suma(5);
        t1.suma(3, 4);
        t1.suma(1.5, 6.78);
        Console.Read();
    }
}
```

Konstruktor i destruktor

Konstruktor to opcjonalny element klasy, to specjalna metoda, wywoływana automatycznie w momencie tworzenia instancji klasy. Jest to metoda publiczna (`public`) o takiej samej nazwie jak klasa (w której jest zadeklarowany) i nie może zwracać żadnej wartości (nawet typu `void`), ale może być przeciążany. Zwyle w konstruktorze dokonuje się inicjalizacji pól klasy na różne sposoby, dlatego najczęściej spotykaną metodą przeciążaną jest właśnie konstruktor (Przykład 2.4). Różne konstruktory zdefiniowane w klasie udostępniają użytkownikowi możliwość tworzenia obiektów na różne sposoby. Jeśli w klasie nie został zdefiniowany żaden konstruktor, to automatycznie tworzony jest konstruktor bezparametrowy, który umożliwia utworzenie obiektu z domyślnymi wartościami (najczęściej pola klasy w takim konstruktorze są zerami odpowiednich typów, np. 0 dla liczb, łańcuch pusty dla typu string, `null` dla obiektów itp.). Jeśli natomiast w klasie zdefiniujemy przynajmniej jeden konstruktor z parametrem, to należy samodzielnie zdefiniować konstruktor bez parametrów (w takim przypadku nie jest on już tworzony automatycznie). Przykład 2.4 definiuje różne konstruktory, również konstruktor bezparametrowy.

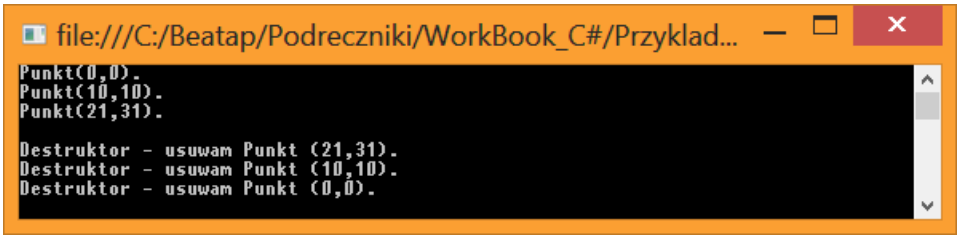
Przeciwieństwem konstruktora jest destruktor, czyli metoda specjalna wywoływana po zakończeniu korzystania z danej klasy. Przy zamykaniu aplikacji specjalny mechanizm środowiska .NET Framework (ang. garbage collection) zwalnia pamięć zarezerwowaną przez poszczególne instancje klas programu. W momencie usuwania danego obiektu klasy z pamięci wywoływany jest jej destruktor. W przeciwieństwie do konstruktora, destruktor w klasie może być tylko jeden i musi nosić taką samą nazwę jak klasa, w której się znajduje, poprzedzoną znakiem tyldy: '~'. Destruktor jest metodą bezparametrową i również nie przekazuje żadnego wyniku.

Przykład 2.4. Klasa Punkt z konstruktorami i destruktorem

```
using System;
class Punkt
{
    //pola klasy - współrzędne punktu na płaszczyźnie (x,y):
    int x, y;
    //konstruktor 1 z dwoma parametrami
    //ustawia punkt (x1,y1) o współrzędnych argumentów:
    public Punkt(int x1, int y1)
    {
        x = x1;
        y = y1;
        //sygnalizacja wywołania konstruktora
        //(na potrzeby przykładu):
        Console.Beep();
    }
```

```
    //konstruktor 2 z jednym parametrem
    //ustawia punkt (z,z) o współrzędnych argumentu:
    public Punkt(int z)
    {
        x = z;
        y = z;
        //sygnalizacja wywołania konstruktora
        //(na potrzeby przykładu):
        Console.Beep();
    }
    //konstruktor bezparametrowy
    //ustawia punkt o współrzędnych (0,0)
    public Punkt()
    {
        x = 0;
        y = 0;
        //sygnalizacja wywołania konstruktora
        // (na potrzeby przykładu):
        Console.Beep();
    }
    //destruktor:
    ~Punkt()
    {
        Console.WriteLine("Destruktor – usuwam Punkt
                          ({0},{1}).",x,y);
        Console.Beep(); //sygnalizacja wywołania destruktora
    }
    //zwykła metoda klasy - nie posiada parametrów
    //wyświetla współrzędne pól klasy:
    public void Pokaz()
    {
        Console.WriteLine("Punkt({0},{1}).", x, y);_
    }
}
class Program {
    static void Main(string[] args)
    {   //utworzenie punktów korzystając z 3 konstruktorów:
        Punkt p1 = new Punkt(); //konstruktor bezargumentowy
                                //tworzy punkt (0,0)
        Punkt p2 = new Punkt(10);//konstruktor z 1 argumentem
                                 //tworzy punkt (10,10)
        Punkt p3 = new Punkt(21, 31);//konstruktor z 2 argumentami
                                     //tworzy punkt (21,31)
        //wyświetlenie współrzędnych utworzonych punktów:
        p1.Pokaz(); p2.Pokaz(); p3.Pokaz();
        Console.Read();
    }
}
```

Wynik działania przykładu 2.4 przedstawia rysunek 2.1.



Rys.2.1. Wynik działania programu z przykładu 2.4.

Odwołanie do aktualnej instancji (obiektu) klasy można zrealizować używając ukrytego wskaźnika do obiektu `this`. Jest to tzw. **autoreferencja** do obiektu. Nie jest wymagana, ale czasami konieczna (jak w przypadku kiedy nazwy parametrów są identyczne jak nazwy pól klasy). Na przykład korzystając z autoreferencji `this`, konstruktor klasy `Punkt` z dwoma parametrami z przykładu 2.4, można zapisać następująco:

```
public Punkt(int x, int y)
 {
     this.x = x; //this.x - pole klasy, x- parametr konstruktora
     this.y = y; //this.y - pole klasy, y- parametr konstruktora
 }
```

W tym przypadku autoreferencja `this` jest konieczna, aby jednoznacznie zdefiniować, że wartości parametrów konstruktora `x` i `y` przypisujemy do pól instancji tej klasy: `this.x` i `this.y`.

Przeciążając konstruktory możemy również skorzystać z autoreferencji do konstruktora tejże klasy. Definicja klasy `Punkt` może wtedy wyglądać tak jak pokazuje przykład 2.5. Zwróćmy uwagę, że w tym przypadku zdecydowanie ograniczyliśmy liczbę linii kodu.

Przykład 2.5. Klasa Punkt z autoreferencją do pierwszego konstruktora

```
class Punkt
{
    //pola klasy - współrzędne punktu na płaszczyźnie (x,y):
    int x, y;
```

```
    //konstruktor 1 z dwoma parametrami:
    //ustawia punkt (x,y) o współrzędnych argumentów:
    public Punkt(int x, int y)
    {
        this.x = x;
        this.y = y;
      }
    //konstruktor 2 z wykorzystaniem autoreferencji do pierwszego
    //ustawia punkt (x,x) o współrzędnych argumentu:
    public Punkt(int x):this(x,x) {}
    //konstruktor bezparametrowy z wykorzystaniem autoreferencji
    //do pierwszego - ustawia punkt o współrzędnych (0,0)
    public Punkt() : this(0, 0) { }

    //destruktor i metoda Pokaz bez zmian
}
```

Sposób tworzenia obiektów klasy `Punkt` pozostaje bez zmian, tak jak w przykładzie 2.4.

Modyfikatory dostępu

Jedną z ważnych cech programowania obiektowego jest hermetyzacja, tzn. ukrycie pewnych składników klasy oraz określenie, dla kogo i które składniki będą dostępne spoza klasy. Dostęp do elementów klasy (metod, pól, właściwości) określają tzw. modyfikatory dostępu, czyli słowa kluczowe zestawione w tabeli 1.5.

Dla przypomnienia są to modyfikatory:

- private (prywatny) – domyślnie (dostępność tylko w obrębie klasy),
- protected (chroniony) – dostępność dla klas potomnych,
- public (publiczny) – pełen dostęp publiczny (zewsząd),
- internal (wewnętrzny) – najrzadziej wykorzystywany modyfikator dostępu, określa, że z danego elementu będzie można skorzystać jedynie w obrębie podzespołu, w którym jest zastosowany.

Modyfikator dostępu musi poprzedzać deklarację danego elementu klasy. Gdy wystąpi słowo kluczowe określające modyfikator dostępu, wszystkie dalsze deklaracje będą traktowane zgodnie z jego znaczeniem, ale dobrą praktyką jest poprzedzanie każdej deklaracji elementu klasy odpowiednim modyfikatorem. Język C# udostępnia dodatkowo możliwość deklarowania pól tylko do odczytu. Takie pole jest poprzedzone słowem kluczowym `readonly` a jego wartość można nadać bezpośrednio w kodzie (w trakcie deklaracji), tak jak w przypadku stałych, lub w ciele konstruktora. Próba zmiany wartości pola w innym miejscu

w kodzie kończy się błędem. Jest to rozwiązanie pośrednie pomiędzy polami stałymi (`const`) a zwykłymi np.:

```
class Przyklad
{
    private readonly int x = 10;
    public Przyklad()
    {       x = 20;  //w konstruktorze jest OK
    }
   public void Zmien()
   {
       x = 30; //poza konstruktorem - błąd!
   }
}
```

Zbiór publicznych składowych klasy, które są dostępne z instancji obiektu (są to wszystkie elementy zadeklarowane w klasie jako `public`) tworzy publiczny interfejs klasy.

Na publiczny interfejs w C# składają się:

- metody – wykonujące pewne zadania, modelujące zachowania klasy,
- właściwości – zamaskowane metody dostępu do prywatnych pól i ich modyfikacji (dokładniejsze informacje na temat właściwości można znaleźć w kolejnych rozdziałach),
- pola publiczne (generalnie nie zbyt dobra idea, ale dostępna w C#).

2.2.2. Typy referencyjne a wartości

Wprowadzenie do typów referencyjnych i wartości zostało przedstawione w rozdziale 1.5. Przypomnijmy, że przekazywanie metodom parametrów przez wartość (domyślnie dla typów prostych, struktur i wyliczeń) polega na tym, iż w metodzie tworzone są kopie parametrów (oryginalne wartości zmiennych przekazanych do metody nie zostają zmieniane).

W celu przekazania parametrów metodzie przez referencję (domyślnie dla klas i interfejsów) możemy skorzystać z modyfikatorów:

- `out` – umożliwia metodzie modyfikację wartości, jaka została do niej przekazana oryginalnie;
- `ref` – j.w. ale zmienna musi zostać zainicjalizowana przez wartości i referencje.

Programowanie z użyciem typów referencyjnych to wydajny sposób tworzenia metod (zwłaszcza dla dużych parametrów). W przypadku referencji do metody przekazywany jest jedynie adres komórki pamięci, w której znajduje się wartość (sama wartość nie jest kopiowana).

Przykłady 2.6-2.8 demonstrują sposób przekazywania parametrów metodzie. Warto porównać wyniki działania przykładów zestawione na rysunku 2.2.

Przykład 2.6. Przekazywanie parametrów przez wartość

```
using System;
class Program
{
    static void Zamiana(int x,int y)
    {  int pom=x;
       x=y;
       y=pom;
    }
    static void Main(string[] args)
    {
        int a=2,b=3;
        Console.Write(" Przed zamiana: a= " +a+ " b= " +b+ "\n");
        Zamiana(a,b);
        Console.Write(" Po zamianie: a= " +a+ " b= " +b+ "\n ");
        Console.Read();
    }
}
```

Przykład 2.7. Przekazywanie parametrów przez referencję (ref)

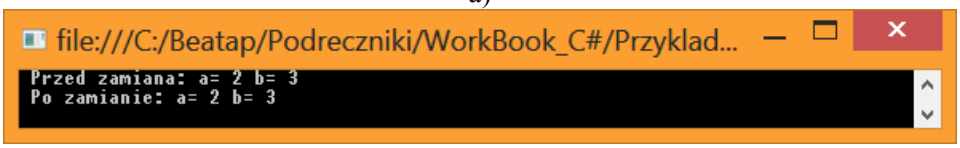
```
using System;
class Program
{

    static void Zamiana(ref int x, ref int y)
    {
        int pom = x;
        x = y;
        y = pom;
    }
    static void Main(string[] args)
    {
        int a = 2, b = 3;
        Console.Write(" Przed zamiana: a=" + a + " b=" + b + "\n");
        Zamiana(ref a, ref b);
        Console.Write(" Po zamianie: a= "+ a + " b= " + b + "\n ");
        Console.Read();
    }
}
```

Przykład 2.8. Przekazywanie parametrów przez referencję (out)

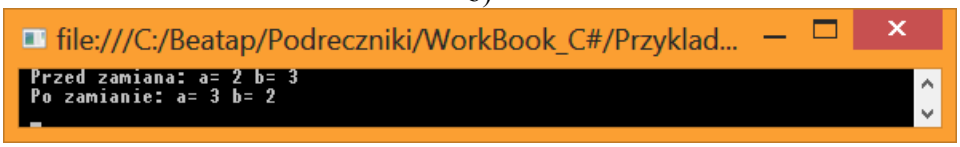
```
using System;
class Program
{
    static void Zamiana(int x, int y, out int x1, out int y1)
    {
```

```
        x1 = y;
        y1 = x;
    }
    static void Main(string[] args)
    {
        int a = 2, b = 3, a1, b1;
        Console.Write(" Przed zamiana: a=" + a + " b= " + b + "\n");
        Zamiana(a, b, out a1, out b1);
        Console.Write(" Po zamianie: a1="+ a1 + " b1=" + b1 + "\n");
        Console.Read();
    }
}
```

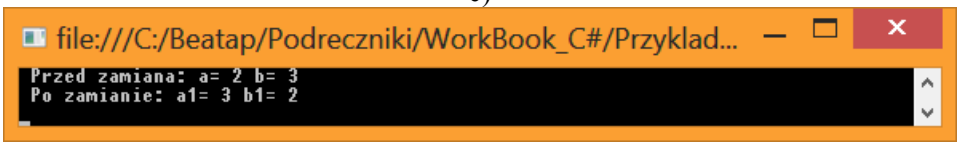
a)



b)



c)



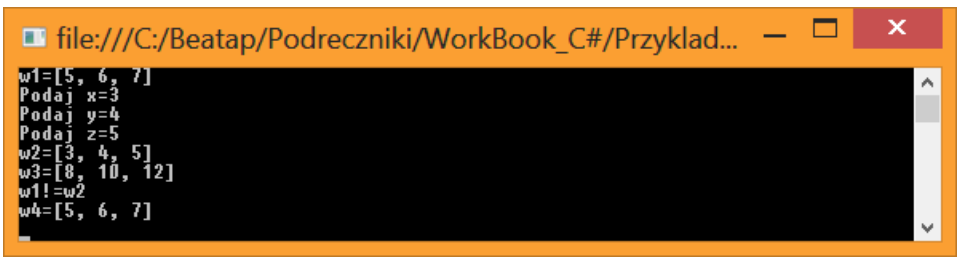
Rys. 2.2. Wynik działania programu z a) przykładu 2.6, b) przykładu 2.7, c) przykładu 2.8

Kolejny przykład 2.9 przedstawia różne sposoby przekazywania obiektów jako parametrów metod przykładowej klasy `Wektor`. Klasa `Wektor` pozwala wykonywać operacje na wektorach w przestrzeni 3D. Każdy wektor posiada 3 współrzędne `x`, `y`, `z`, które są zdefiniowane jako prywatne pola klasy. Metody klasy umożliwiają operacje np. obliczania długości wektora, dodawania i porównywania dwóch obiektów klasy `Wektor`, itp. Warto zwrócić uwagę, że niektóre metody muszą wykorzystać autoreferencję do obiektu (np. metoda `CzyRowne`). Zauważmy ponadto, że metody klasy mogą wywoływać inne metody tej samej klasy (np. `metoda CzyRozne` korzysta z metody `CzyRowne`).

Przykład 2.9. Klasa Wektor – metody z parametrami obiektowymi

```
using System;
class Wektor         //początek definicji klasy
{
    double x, y, z;  //pola prywatne – współrzędne wektora [x, y, z]
    //metody publicznego interfejsu klasy
    //***********************************
    public Wektor()  //konstruktor bezparametrowy
    {
        x = 0; y = 0; z = 0;
    }
    public Wektor(double x, double y, double z) //konstruktor
    {
        this.x = x; this.y = y; this.z = z;
    }
    public void Czytaj()
    {
        Console.Write("Podaj x=");
        x = Convert.ToDouble(Console.ReadLine());
        Console.Write("Podaj y=");
        y = Convert.ToDouble(Console.ReadLine());
        Console.Write("Podaj z=");
        z = Convert.ToDouble(Console.ReadLine());
    }
    public void Wyswietl()
    {
        Console.WriteLine("[{0}, {1}, {2}]", x, y, z);
    }
    public double DlugoscWektora()
    {
        return Math.Sqrt(x * x + y * y + z * z);
    }
    public bool CzyRowne(Wektor w)
    {
        return x.Equals(w.x) && y.Equals(w.y) && z.Equals(w.z);
    }
    public bool CzyRozne(Wektor w)
    {
        return !this.CzyRowne(w);
    }
    public Wektor SumaWektorow(Wektor w)
    {
        Wektor p = new Wektor();
        p.x = x + w.x; p.y = y + w.y; p.z = z + w.z;
        return p;
    }
    public Wektor DluzszyWektor(Wektor w)
    {
```

```
        if (w.DlugoscWektora() > this.DlugoscWektora()) return w;
        else return this;
    }
}       //koniec klasy Wektor

class Program
{
    static void Main(string[] args)
    {   Wektor w1 = new Wektor(5, 6, 7);
        Console.Write("w1="); w1.Wyswietl();
        Wektor w2 = new Wektor(), w4;
        w2.Czytaj();
        Console.Write("w2="); w2.Wyswietl();
        Wektor w3 = new Wektor();
        w3 = w1.SumaWektorow(w2);
        Console.Write("w3="); w3.Wyswietl();
        if (w2.CzyRowne(w1)) Console.WriteLine("w1==w2");
        else Console.WriteLine("w1!=w2");
        w4 = new Wektor();
        w4 = w1.DluzszyWektor(w2);
        Console.Write("w4="); w4.Wyswietl();
        Console.Read();
    }
}
```



Rys. 2.3. Wynik działania programu z przykładu 2.9

2.2.3. Dziedziczenie i polimorfizm

W języku C# możemy deklarować klasę składową wewnątrz innej klasy. Taka wewnętrzna klasa musi mieć ustalony poziom widoczności (public, private, protected, internal). Na tej samej zasadzie należy określić widoczność całej klasy. Widoczność metody wskazuje, które metody mają być dostępne z danej instancji obiektu. Widoczność klasy określa, które części systemu mogą tworzyć obiekty danego typu.

Klasy (ale również struktury i typy enum) mogą być:

- `public` – obiekty klas publicznych mogą być tworzone przez dowolne inne obiekty w obrębie tej samej lub innej przestrzeni nazw,
  - np.: `public class Punkt { },`
- `internal` (domyślnie) - obiekty klas wewnętrznych mogą być tworzone jedynie przez obiekty znajdujące się w tej samej przestrzeni nazw (nie są dostępne dla obiektów z innych przestrzeni),
  - np.: `internal class Punkt { }.`

Dziedziczenie

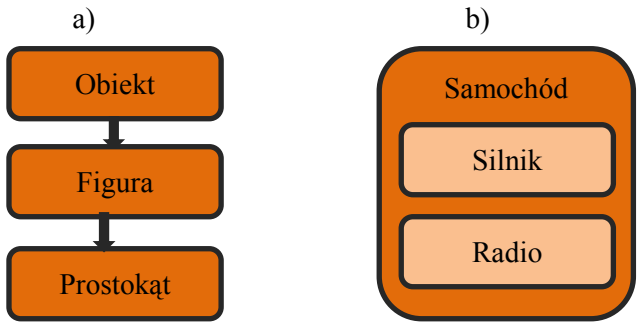
W programowaniu obiektowym występują trzy podstawowe idee:

- hermetyzacja (ang. encapsulation) - określa jak dokładnie dany język jest w stanie ukryć wewnętrzną implementację obiektów (realizowana za pomocą modyfikatorów dostępu);
- dziedziczenie (ang. inheritance) - określa do jakiego stopnia język umożliwia wielokrotne wykorzystywanie kodu;
- polimorfizm (ang. polymorphism) – określa na ile język pozwala obchodzić się z powiązanymi ze sobą (w relacji dziedziczenia) obiektami w podobny sposób.

Powtórne wykorzystywanie kodu może być realizowane na dwa sposoby: przy wykorzystaniu relacji **is-a** i **has-a**.

Relacja **is-a** utożsamia klasyczne dziedziczenie, tzn. jedna klasa jest rozszerzeniem innej klasy i dziedziczy po niej pola i metody (ze specyfikatorem `public` lub `protected`). Taki związek między klasami określa się relacją **'jest'** (inaczej związek **generalizacji** lub **specjalizacji**) (Rys.2.4a). Zgodnie z rysunkiem 2.4a każdy obiekt klasy `Prostokąt` jest również obiektem klasy `Figura` (przejmuje, czyli dziedziczy pewne własności i metody z figury), a każdą figurę możemy traktować jako pewien ogólny typ `Obiekt`. Klasa nadrzędna `Obiekt` definiuje pewne wspólne właściwości i metody związane z każdym obiektem. W C# taką klasą bazową dla wszystkich pozostałych klas (wszystkie klasy po niej dziedziczą) jest klasa `Object`.

Dziedziczenie w relacji **has-a** to inny sposób wielokrotnego użycia kodu, realizujący model zawierania i delegacji (relacja zawiera). W tym przypadku nie ustala się relacji między klasą podstawową a podklasą, ale jedna klasa może zawierać drugą i eksponować część lub całość jej działania (Rys.2.4b). Zgodnie z rysunkiem 2.4b obiekt klasy `Samochód` zawiera obiekt klasy `Silnik` oraz obiekt klasy `Radio`.



Rys. 2.4. Relacja dziedziczenia a) jest (is-a), b) zawiera (has-a)

Biblioteka klas środowiska .NET Framework jest oparta na dziedziczeniu. Klasa `Object` z przestrzeni nazw `System` to główna klasa bazowa w .NET Framework. Kolejne klasy, np. `Consolas`, `Int32` to klasy potomne dziedziczące z klasy podstawowej `Object` określone pola i metody. Deklarując nową klasę następuje automatycznie dziedziczenie po klasie `System.Object` (nawet jeżeli jawnie tego nie określimy).

Dziedziczenie w C# realizowane jest za pomocą deklaracji:

```
class Nazwa_klasy : Nazwa_klasy_bazowej { }
```

Na przykład, jeśli istnieje klasa bazowa `A:`

```
class A
{
    public void Wyswietl()
    { Console.WriteLine("Metoda klasy A"); }
}
```

to możemy zdefiniować klasę `B` dziedziczącą po `A`, która będzie mogła korzystać ze wszystkich pól i metod publicznych (`public`) lub chronionych (`protected`) klasy `A`, nie mając dostępu do jej pól prywatnych:

```
class B : A {}
```

Po takiej deklaracji dziedziczenia możemy utworzyć obiekt klasy `B` i skorzystać z publicznej metody `Wyswietl`() klasy `A`, np.:

```
B b1 = new B();
b1.Wyswietl();
```

Właściwości

Dobrą praktyką w programowaniu obiektowym jest deklaracja pól jako prywatnych elementów klasy. Poza klasycznymi polami klasy, język C#

udostepnia tzw. właściwości, które służą do komunikowania się ze „światem zewnętrznym" i podobnie jak pola gromadzą dane, ale dodatkowo istnieje możliwość zaprogramowania różnych czynności podczas, na przykład, przypisywania im wartości. Właściwości mogą być prywatne, ale najczęściej są publiczne i udostępniają publiczne metody do odczytu/zapisu informacji. Biblioteka klas środowiska .NET Framework oparta jest właśnie na właściwościach oraz metodach. Pola klas są polami prywatnymi, wykorzystywanymi na potrzeby danej klasy a właściwości udostępniają publiczny interfejs (metody `get/set`) umożliwiający użytkownikowi dostęp do odpowiednich danych (Przykład 2.10).

Przykład 2.10. Definicja właściwości z metodami get/set

```
class Przyklad
{
    private int liczba = 12; // liczba (pole prywatne)- małe litery
    public  int Liczba  // Liczba (publiczna właściwość)- pierwsza
                        // duża litera
    {
        get { return liczba;  } //metoda właściwości get (pobierz)
        set { liczba = value; } //metoda właściwości set (ustaw)
    }
  }
```

Próba odczytania wartości `liczba` (pola prywatnego klasy!) spowoduje zwrócenie wartości pola poprzez metodę `get` właściwości `Liczba`) np:

```
Przyklad p1 = new Przyklad();
//Próba dostępu do pola prywatnego liczba:
Console.WriteLine("Miesiąc:"+p1.liczba); //Błąd (pole private)
//Próba dostępu do właściwości Liczba:
Console.WriteLine("Miesiąc:"+p1.Liczba);//OK (właściwość public)
```

Właściwości zapewniają dogodny sposób na przypisywanie i odczytywanie danych, ukrywając przy tym szczegóły ich weryfikacji (Przykład 2.11). Słowo kluczowe `get` jest używane do zwracania wartości pola, a `set` do ustawiania nowych wartości. Właściwość nie może pozostać pusta, tj. musi posiadać blok `set` lub `get`. Właściwości nie posiadające bloku `set` są tylko do odczytu.

Przykład 2.11. Właściwość z weryfikacją danych w metodzie set

```
using System;
class Przyklad{
    private int godzina; //pole prywatne
    public int Godzina //właściwość publiczna
    {
        get { return godzina;  }
```

```
        set {
             if (value >= 0 && value <= 24)
             {
                godzina = value;
             }
               else godzina=0; //domyślnie
            }
    } //koniec właściwości
    public string Info() //zwykła metoda publiczna klasy
    {
        return "Jest godzina " + godzina;
    }
}

class Program
{
    static void Main(string[] args)
    {   Przyklad p=new Przyklad();
        p.Godzina=13;
        Console.WriteLine(p.Info());
        Console.Read();
    }
}
```

Właściwości również mogą posiadać modyfikatory dostępu, które mogą być przypisywane blokom `set/get` np.

```
   public int Godzina
       {
           get { return godzina; }
           protected set
           {   //to samo co w przykładzie 2.10
           }
       }
```

Bloki `set/get` właściwości klas nazywa się **akcesorami get/set**.
Kolejne możliwości oferowane prez akcesory właściwości `get/set` przedstawia przykład 2.12.

Przykład 2.12. Praca z akcesorami get/set

```
using System;
class Student
{
    private string nazwisko = ""; //dostęp prywatny!
    private long numerIndeksu = 0; //dostęp prywatny!
    public string Nazwisko
    {  //publiczne metody get/set
        get { return nazwisko; }
```

```
        set { nazwisko = value; }
    }
    public long NumerIndeksu
    {  //publiczne metody get/set
        get
        { return numerIndeksu; }
        set
        {
            if (value < 0 || value > 999999) numerIndeksu = 0;
            else numerIndeksu = value;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Student s1 = new Student();
        s1.Nazwisko = "Kowalski";
        s1.NumerIndeksu = 666666;
        Console.WriteLine("Student: " + s1.Nazwisko + ", numer
indeksu: " + s1.NumerIndeksu);

        Console.Read();
    }
}
```

W przypadku właściwości `Nazwisko` w przykładzie 2.12 nie wykonujemy żadnych dodatkowych działań związanych z weryfikacją danych. W takim przypadku można uprościć kod właściwości do postaci:

```
public string Nazwisko  { get; set; }
```

Składniki statyczne klasy

Składnikami statycznymi klasy (słowo kluczowe `static`) mogą być:

- klasy – są ładowane do pamięci w momencie, gdy jest ładowany program lub przestrzeń nazw, która je zawiera ,
- metody – są związane z samą klasą (nie musi istnieć obiekt klasy, aby można było skorzystać z metody statycznej),
- właściwości – analogicznie jak metody.

Zwykłe metody klasy (nie statyczne) nie mogą być wywoływane bez uprzedniego utworzenia instancji klasy (obiektu). Do metod statycznych odwołujemy się z nazwy samej klasy a nie z obiektu, zatem w metodach

statycznych nie można używać słowa kluczowego `this` (ponieważ żaden obiekt tej klasy może jeszcze nie istnieć) ani odnosić się do elementów nie statycznych (w metodach statycznych korzysta się tylko z pól i metod statycznych). Do metody statycznej nie można się odnieść z obiektu. Przykładem metody statycznej jest główna metoda każdego programu:

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

Przykład 2.13 przedstawia sposób definiowania metody statycznej i korzystania z niej w programie.

Przykład 2.13. Klasa z metodą statyczną

```
using System;
class Przyklad
{
    public Przyklad()
    {
        Console.WriteLine("Tworzenie nowego obiektu!");
    }
    static public void Metoda_statyczna()
    {
        Console.WriteLine("Metoda statyczna!");
    }
}
class Program
{
    static void Main(string[] args)
    {
        //odwołanie do metody statycznej:
        Przyklad.Metoda_statyczna();
        //utworzenie obiektu klasy Przyklad:
        Przyklad p = new Przyklad();
        //próba odwołania się do metody statycznej z obiektu p:
        p.Metoda_styczna();// Błąd!
        Console.Read();
    }
}
```

Cała klasa jest statyczna, gdy wszystkie elementy tej klasy są statyczne. Statyczne klasy posiadają statyczne i prywatne konstruktory, które nie mogą posiadać modyfikatorów dostępu oraz parametrów. Nie można jawnie wywołać takiego konstruktora - jest on wywoływany wówczas, gdy nastąpi pierwsze odwołanie do klasy.

Przykładem klasy statycznej w C# jest klasa `Math`, której wszystkie metody np. `Sin()`, `Cos()`, `Pow()` itd. są metodami statycznymi tej klasy i można się do nich odwołać tylko z nazwy klasy, np.:

```
Math.Cos(20);
```

Polimorfizm i metody wirtualne

Polimorfizm (gr. polymorphos – wielopostaciowy) jest związany z dziedziczeniem i realizuje możliwość operowania na obiektach należących do różnych klas ustawionych w hierarchii dziedziczenia.

Tworząc hierarchię klas bardzo często nadpisujemy (predefiniowujemy) metody klasy bazowej w klasach pochodnych, modyfikując w ten sposób ich działania dla obiektów klas pochodnych. Taką sytuację pokazuje przykład 2.14.

Przykład 2.14. Przesłonięcie metody w klasie pochodnej

```
class A{
    public void Run()
    {
        Console.WriteLine("Metoda Run() z klasy A");
    }
}
class B : A{
    public void Run()  //przesłaniamy metodę nadklasy
    {
        Console.WriteLine("Metoda Run() z klasy B");
    }
}
```

Przy odwołaniu się do metody `Run()` z obiektu klasy `B` zdefiniowanej w przykładzie 2.14:

```
B b = new B();
b.Run(); //wywołanie nadpisanej w klasie B metody Run
```

wyświetli się ostrzeżenie o nadpisaniu metody `Run()` z klasy `A`.

Aby uniknąć takiego ostrzeżenia można zmodyfikować nagłówek metody `Run()` w klasie `B` i poprzedzić go słowem kluczowym `new` (jest to wtedy informacja dla kompilatora o świadomym nadpisaniu metody klasy bazowej):

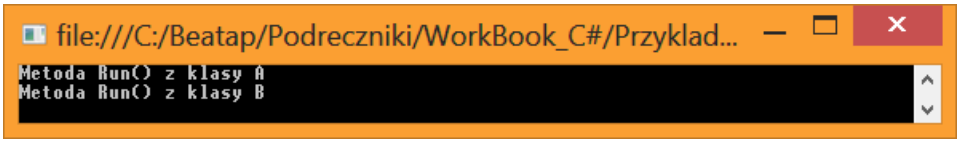
```
new public void Run() //świadomie przesłaniamy metodę nadklasy
```

Elementy statyczne mogą być również przesłaniene w klasach potomnych. Przesłonięcie metody lub pola w klasie pochodnej nie oznacza, że tracimy dostęp do przesłoniętych składników klasy bazowej. Dostep do nadpisanej metody lub pola jest możliwy dzięki wykorzystaniu słowa kluczowego `base`, co pokazuje przykład 2.15.

Przykład 2.15. Korzystanie z metod nadklasy (słowo kluczowe base)

```
using System;
class A{
    public void Run()
    {
        Console.WriteLine("Metoda Run() z klasy A");_
    }
}
class B : A{
    new public void Run() //świadomie przesłaniamy metodę nadklasy
    {
        base.Run();  //korzystamy z metody nadklasy
        Console.WriteLine("Metoda Run() z klasy B");
    }
    public B():base() //korzystamy z konstruktora nadklasy
    {
        //dodatkowy kod dla konstruktora klasy B
    }
}
class Program
{
    static void Main(string[] args)
    {
        B b = new B();
        b.Run(); //wywołanie nadpisanej w klasie B metody Run
        Console.Read();
    }
}
```

Wynik działania przykładu 2.15 przedstawia rysunek 2.5.



Rys. 2.5. Wynik działania przykładu 2.15.

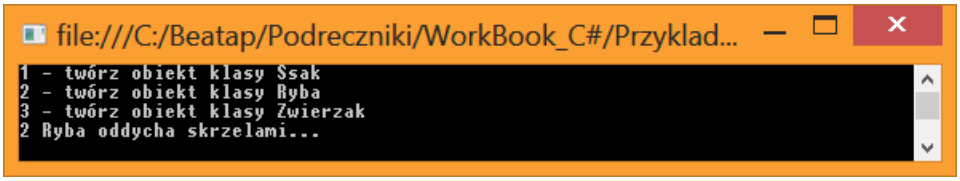
Z polimorfizmem ściśle związane są metody wirtualne. Metoda wirtualna jest przygotowana do zastąpienia jej innym kodem w klasie potomnej. Taką metodę tworzy się dodając do jej deklaracji słowo kluczowe `virtual` (przykład 2.16). W programie z przykładu 2.16 obiekt `zwierz` jest tworzony w zależności od wyboru użytkownika. Z hierarchii dziedziczenia wynika, że każdy obiekt klasy

pochodnej jest jednocześnie obiektem klasy bazowej (np. `Ryba` to `Zwierz` i `Ssak` to `Zwierz`). Polimorfizm (wielopostaciowość) jest tu realizowany na bazie wirtualnej metody `Oddychanie()`, która jest wywoływana z odpowiedniej klasy, w zależności od typu utworzonego obiektu z jakim faktycznie mamy do czynienia (`Ryba`, `Ssak` lub `Zwierz`) (Rys.2.6).

Przykład 2.16. Metody wirtualne i polimorfizm

```
using System;
class Zwierzak {
    public int Wiek;
    public virtual void Oddychanie()
    {
        Console.WriteLine(" Zwierzak oddycha...");
    }
}
class Ssak : Zwierzak {
    public override void Oddychanie()
    {
        Console.WriteLine(" Ssak oddycha płucami...");
    }
}
class Ryba : Zwierzak {
    public override void Oddychanie()
    {
        Console.WriteLine(" Ryba oddycha skrzelami...");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Zwierzak zwierz;
        Console.WriteLine("1 - twórz obiekt klasy Ssak");_
        Console.WriteLine("2 - twórz obiekt klasy Ryba");
        Console.WriteLine("3 - twórz obiekt klasy Zwierzak");
        //wczytaj znak z Consoli i przypisz go do obiektu klasy
ConsoleKeyInfo:
        ConsoleKeyInfo Key = Console.ReadKey();
        switch (Key.KeyChar)_
        {  case '1': zwierz = new Ssak();break;
           case '2': zwierz = new Ryba(); break;
           default: zwierz = new Zwierzak(); break;
        }

        // wywołanie metody Oddychanie
        zwierz.Oddychanie();
```

```
        Console.Read();
    }
}
```



Rys. 2.6. Wynik działania przykładu 2.16.

Metoda statyczna nie może być wirtualna. Metody wirtualne są wywoływane na rzecz obiektu klasy, a metoda statyczna związana jest z istnieniem samej klasy i nie może być wołana z jej obiektu.

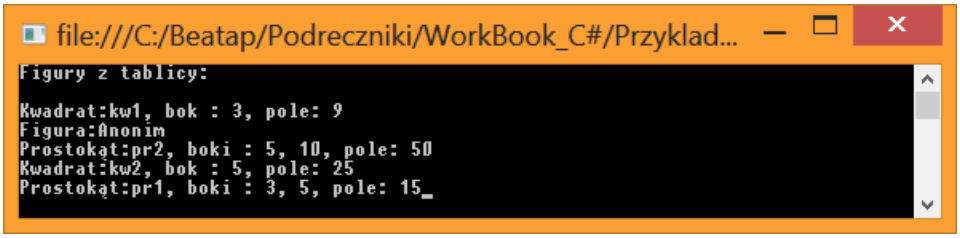
Kolejny przykład 2.17 z wykorzystaniem metod wirtualnych dotyczy dziedziczenia figur i definiuje metodę wirtualną do obliczania pola figury. W programie dodatkowo utworzono tablicę figur oraz wyświetlono informacje o każdej z nich.

Przykład 2.17. Metody wirtualne i polimorfizm, tablica obiektów

```
using System;
class Figura
{
    protected string nazwa;
    public Figura() { nazwa = "Anonim"; }
    public Figura(string n) { nazwa = n; }
    public virtual void Wyswietl()
    { Console.Write("\nFigura:" + nazwa); }
}
class Kwadrat : Figura
{
    protected double a = 1;
    public Kwadrat() { nazwa = "Kwadrat"; }
    public Kwadrat(string k) : base(k) { }
    public Kwadrat(string k, double bok) : base(k) { a = bok; }
    public virtual double ObliczPole() { return a * a; }
    public override void Wyswietl()
    {
        Console.Write("\nKwadrat:" + nazwa + ", bok : " +
         a + ", pole: " + ObliczPole());
    }
```

```
}
class Prostokat : Kwadrat
{
    protected double b = 1;
    public Prostokat() { nazwa = "Prostokat"; }
    public Prostokat(string k) : base(k) { }
    public Prostokat(string k, double bok1, double bok2)
        : base(k)
    {
        a = bok1;
        b = bok2;
    }
    public override double ObliczPole() { return a * b; }
    public override void Wyswietl()
    {
        Console.Write("\nProstokąt:" + nazwa + ", boki : "
        + a + ", " + b + ", pole: " + ObliczPole());
    }
}

class Program
{
    static void Main(string[] args)
    {
        Figura f = new Figura();
        Kwadrat k1 = new Kwadrat("kw1", 3);
        Kwadrat k2 = new Kwadrat("kw2", 5);
        Prostokat p1 = new Prostokat("pr1", 3, 5);
        Prostokat p2 = new Prostokat("pr2", 5, 10);
        Figura[] figury = { k1, f, p2, k2, p1 };
        Console.WriteLine("Figury z tablicy:");
        foreach (Figura el in figury)
            el.Wyswietl();
        //lub zwykłym for:
        //for (int i = 0; i < figury.Length; i++)
        //figury[i].Wyswietl();
        Console.Read();
    }
}
```

Wynik działania programu z przykładu 2.17 przedstawia rysunek 2.5.



Rys. 2.6. Wynik działania przykładu 2.17.

Zamiast w każdej klasie definiować metodę `Wyswietl()`, można skorzystać z metody `ToString()`, która jest dostępna dla każdego obiektu, ponieważ każda klasa dziedziczy (nie jawnie) po bazowej klasie `Object`. W klasie `Object` zdefiniowana jest publiczna metoda wirtualna `ToString()`, przygotowana do przesłonięcia. Przykład 2.18 jest lekką modyfikacją przykładu 2.17, do którego dodano metodę `ToString()` dla klasy `Figura`. Wynikiem działania metody `ToString()` jest oczywiście łańcuch z opisem danego obiektu. Brak nadpisania tej metody nie wyklucza jej wykorzystania – zwracanym łańcuchem jest wówczas domyślny opis obiektu. Konwersja na łańcuch jest realizowana automatycznie.

Przykład 2.18. Predefiniowana metoda ToString()

```
class Figura
{
    protected string nazwa;
    public Figura() { nazwa = "Anonim"; }
    public Figura(string n) { nazwa = n; }
    public virtual void Wyswietl()
    {
        Console.Write("\nFigura:" + nazwa);
    }
    public override string ToString()
    {
        return String.Format("\nFigura:" + nazwa);
    }
}
```

Wykorzystanie tak przesłoniętej metody `ToString()` jak w przykładzie 2.18 jest następujące:

```
Figura f = new Figura();
f.Wyswietl();         //wykorzystanie własnej metody
Console.WriteLine(f); //wykorzystanie metody przesłoniętej
```

W przypadku dużych projektów można podzielić kod klasy na kilka różnych modułów korzystając ze słowa kluczowego `partial` (Przykład 2.19). Taki kod w trakcie kompilacji jest łączony w jedną klasę. Rozwiązanie to jest stosowne w projektach typu `Windows Forms`, o czym będzie mowa w rozdziale 3.

Przykład 2.19. Podział kodu klasy na moduły

```
partial class Przyklad {
    public Przyklad() { }
}
partial class Przyklad {
    public void Wyswietl() { }
}
```

2.2.4. Elementy abstrakcyjne i interfejsy

Bardzo często w programowaniu obiektowym wykorzystuje się elementy abstrakcyjne. Klasa abstrakcyjna (poprzedzona słowem kluczowym `abstract`) nie ma implementacji (tj. definicji metod), zawiera jedynie nagłówki metod abstrakcyjnych. Może zawierać również zwykłe składniki.

Deklaracja klasy abstrakcyjnej ma postać:

```
abstract class Zwierzak
{
    public int Wiek;

    public abstract void Oddychanie();
}
```

Klasa abstrakcyjna jest fundamentem do budowania kolejnych klas. Nie wszystkie metody klasy abstrakcyjnej muszą być abstrakcyjne. Jednak jeżeli klasa zawiera przynajmniej jeden element abstrakcyjny, to również musi być opatrzona klauzulą `abstract`. Metody abstrakcyjne nie posiadają kodu, nie mogą być opatrzone klauzulą `virtual` lub `static`.

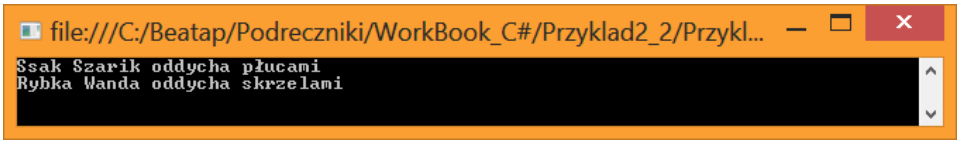
Nie można tworzyć obiektów klasy abstrakcyjnej. Klasa abstrakcyjna jest konkretyzowana poprzez utworzenie zwykłej klasy dziedziczącej po klasie abstrakcyjnej. Takie klasy potomne muszą implementować (definiować ich działanie) abstrakcyjne metody bazowej klasy abstrakcyjnej.

Przykład 2.20 pokazuje sposób definiowania klasy abstrakcyjnej `Zwierzak` z abstrakcyjną metodą `Oddychanie()` oraz dwie klasy konkretne: `Ryba` i `Ssak` dziedziczące po klasie `Zwierzak` i implementujące metodę `Oddychanie()`.

Przykład 2.20. Definicja klasy abstrakcyjnej i jej wykorzystanie

```
using System;

//klasa abstrakcyjna Zwierzak:
abstract class Zwierzak
{
    public string nazwa = "Zwierzak";
    public abstract void Oddychanie();
}

//klasa konkretna Ryba dziedzicząca po Zwierzak
//implementuje abstrakcyjną metodę Oddychanie()
class Ryba : Zwierzak
{
    public Ryba(string n) { nazwa = n; }
    public override void Oddychanie()
    { Console.WriteLine("Rybka " + nazwa + " oddycha skrzelami"); }
}

//klasa konkretna Ssak dziedzicząca po Zwierzak
//implementuje abstrakcyjną metodę Oddychanie()
class Ssak : Zwierzak //klasa konkretna
{
    public Ssak(string n) { nazwa = n; }
    public override void Oddychanie()
    { Console.WriteLine("Ssak " + nazwa + " oddycha płucami"); }
}

class Program
    {
        static void Main(string[] args)
        {
            Ssak s = new Ssak("Szarik");
            Ryba r = new Ryba("Wanda");
            s.Oddychanie();
            r.Oddychanie();
            Console.ReadKey();
        }
    }
```

Wynik działania programu z przykładu 2.20 przedstawia rysunek 2.7.



Rys. 2.7. Wynik działania przykładu 2.20.

Interfejsy przypominają klasy abstrakcyjne, ale mogą zawierać jedynie deklaracje metod oraz właściwości, nie zawierają zwykłych pól. Z definicji wszystkie metody interfejsu są publiczne i abstrakcyjne, dlatego przed nazwą metody nie umieszcza się już słowa `abstract`. Interfejsy deklaruje się z użyciem słowa kluczowego `interface` np.:

```
interface IKlasa
{
    int Potega(int X, int Y); // tylko nagłówek metody bez ciała
}
```

Klasy języka C# mogą dziedziczyć jedynie elementy pojedynczej klasy a interfejs definiuje wyłącznie mechanizm pośredniczący w komunikacji klienta z obiektem. Za obsługę interfejsu i odpowiednią implementację każdej z jego funkcji odpowiada klasa. Klasa może dziedziczyć wiele interfejsów, ale musi implementować ich wszystkie metody np.:

```
class Klasa : IKlasa
{
    public int Potega(int X, int Y)
    { return X * Y; }
}
```

Dobrą praktyką jest specyficzne nazewnictwo interfejsów polegające na dodaniu przed nazwą dużej litery `I`. Przykład 2.21 definiuje dwa interfejsy i klasę dziedziczącą po obu interfejsach i implementującą ich metody.

Przykład 2.21. Klasa dziedzicząca po dwóch interfejsach

```
interface IA {
    void metoda();
}
interface IB {
    void metoda();
}
class C : IA, IB {
    void IA.metoda()
    {
```

```
        Console.WriteLine("Metoda implementująca IA.metoda");
    }
    void IB.metoda()
    {
        Console.WriteLine("Metoda implementująca IB.metoda");
    }
}
```

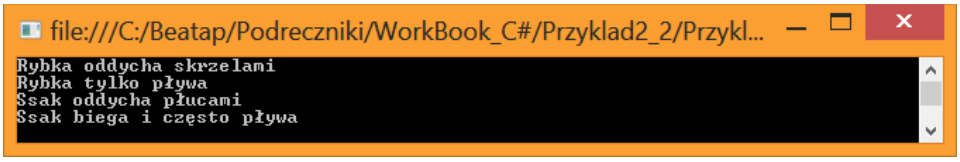
Kolejny przykład 2.22 pokazuje zastosowanie interfejsu `IZycie` w klasie `Ryba` i `Ssak`. Obie klasy dzidziczą po tym interfejsie i implementują jego metody `Oddychanie()` i `Poruszanie()`. Warto zwrócić uwagę, że każdy obiekt klasy `Ryba` i każdy obiekt klasy `Ssak` jest również obiektem klasy interfejsu `IZycie`. W przykładzie utworzono również tablicę takich obiektów oraz wykorzystano pętle `foreach` w celu przetestowania działania zaimplementowanych metod.

Przykład 2.22. Definicja i zastosowanie interfejsu

```
using System;

interface IZycie
{
    void Oddychanie();
    void Poruszanie();
}

class Ryba : IZycie
{
    public void Oddychanie()
    { Console.WriteLine("Rybka oddycha skrzelami"); }
    public void Poruszanie()
    { Console.WriteLine("Rybka tylko pływa"); }
}

class Ssak : IZycie
{
    public void Oddychanie()
    { Console.WriteLine("Ssak oddycha płucami"); }
    public void Poruszanie()
    { Console.WriteLine("Ssak biega i często pływa"); }
}

class Program
    {
        static void Main(string[] args)
        {
            Ssak s = new Ssak();
```

```
            Ryba r = new Ryba();
            IZycie[] z = { r, s }; //Ssak oraz Ryba są również
                                   // obiektami klasy interfejsu IZycie
            foreach (IZycie el in z)
            {
                el.Oddychanie();
                el.Poruszanie();
            }

            Console.ReadKey();
        }
    }
```

Wynik działania programu z przykładu 2.22 przedstawia rysunek 2.8.



Rys. 2.8. Wynik działania przykładu 2.22.

Interfejsy mogą dziedziczyć po sobie podobnie jak klasy. Wszystkie elementy interfejsu są domyślnie traktowane jako publiczne. Interfejs nie może zawierać elementów statycznych.

W tym miejscu warto wspomnieć również o operatorach `is` i `as`:

- operator `is` sprawdza typ obiektu i stosowany jest w połączeniu z klasami, działa podobnie jak porównanie operatorem `==` (jednak błędne jest sprawdzenie postaci: `obiekt == typ`);
- operator `as` służy do konwersji - ale nie typów; dzięki niemu możliwa jest zmiana właściwości komponentu, jeśli tylko znany jest jego typ (nie jest konieczna informacja o nazwie komponentu).

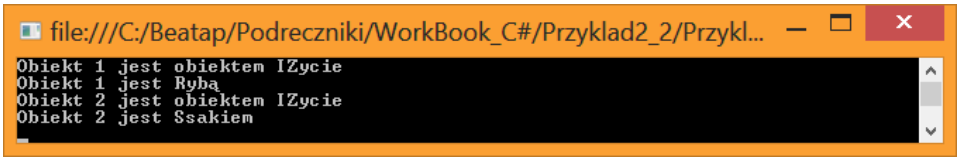
Przykład 2.23 i 2.24, bazując na definicji interfejsu i klas z przykładu 2.22, przedstawia sposób wykorzystania operatorów `is` i `as`.

Przykład 2.23. Wykorzystanie operatora is

```
Ssak s = new Ssak();
Ryba r = new Ryba();
IZycie[] z = { r, s };
int licznik = 0;
```

```
foreach (IZycie el in z)
{
  licznik++;
  if (el is IZycie)
   Console.WriteLine("Obiekt " + licznik + " jest obiektem IZycie");
  if (el is Ryba)
   Console.WriteLine("Obiekt " + licznik + " jest Rybą");
  if (el is Ssak)
    Console.WriteLine("Obiekt " + licznik + " jest Ssakiem");
}
```

Wynik działania programu z przykładu 2.23 przedstawia rysunek 2.9.

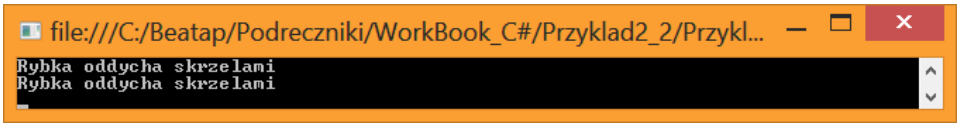


Rys. 2.9. Wynik działania przykładu 2.23.

Przykład 2.24. Wykorzystanie operatora as

```
Ssak s = new Ssak();
Ryba r1 = new Ryba();
Ryba r2 = new Ryba();
IZycie[] z = { r1, s, r2 };
foreach (IZycie el in z)
{
  if (el is Ryba)
  {
     Ryba r = el as Ryba;
     r.Oddychanie();
    //tu mozna też modyfikować właściwości el jako Ryby
  }
}
```

Wynik działania programu z przykładu 2.24 przedstawia rysunek 2.10.



Rys. 2.10. Wynik działania przykładu 2.24.

2.2.5. Obsługa wyjątków

Wyjątek (ang. exception) to mechanizm kontroli przepływu występujący w językach programowania i służący do obsługi zdarzeń wyjątkowych (błędów). Nowoczesne kompilatory starają się wykryć takie sytuacje i nie dopuścić do kompilacji.

Na przykład kod:

```
int  X; X = 10 / 0;
```

nie zostanie skompilowany, natomiast kod:

```
int X, Y, Z; X = 10; Y = 0; Z = X / Y;
```

zostanie skompilowany, ale w trakcie działania programu wyświetlony będzie błąd (*An unhandled exception of type 'System.DivideByZeroException' occurred*).

Mechanizm wyjątków pozwala przechwycić niedopuszczalne i wyjątkowe sytuacje i odpowiednio zareagować, np. wyświetleniem stosownego komunikatu. W tym celu kompilator musi zostać odpowiednio poinformowany, jakie miejsce kodu jest narażone na wystąpienie błędu.

Do podstawowych pojęć związanych z mechanizmem obsługi wyjątków należą:

- typ reprezentujący szczegóły wyjątkowych okoliczności,
- metoda, która zgłasza (wyrzuca) wyjątek,
- blok kodu, który wywoła metodę przygotowaną na zaistnienie wyjątku,
- blok kodu, który ma przetworzyć (lub złapać) wyjątek.

Do przechwycenia wyjątku służy instrukcja `try-catch` uzupełniona ewentualnie blokiem `finally`, z którego instrukcje będą wykonane niezależnie od tego czy wystąpił wyjątek czy nie. Blok `finally` można pominąć. Przykład 2.25 pokazuje wykorzystanie bloku `try-catch`.

Przykład 2.25. Wykorzystanie bloku try-catch

```
int X, Y, Z; X = 10; Y = 0;
try
{  //spróbuj wykonać dzielenie,
   //jeśli się nie uda przejdź do bloku catch:
   Z = X / Y;
}
```

```
catch
{
   Console.WriteLine("Nie dziel przez zero!");
}
finally
{
   Console.WriteLine("Kod do wykonania niezależnie od tego"+
                "czy wystąpił wyjatek czy nie");
}
```

W przypadku wystąpienia większej liczby wyjątków różnego typu, bloki `try-catch` można zagnieżdżać tak jak w przykładzie 2.26.

Przykład 2.26. Zagnieżdżanie bloków try-catch

```
try{
    // kod
    try {
         // kod
         try { }
         catch
         {  //błąd
         }
         finally { }
     }
    catch { //inny błąd
          }
    }
catch { //jeszcze inny  błąd
      }
finally {  // blok finally
        }
```

Wyjątki nie obsługiwane przez aplikację, są obsługiwane przez system (komunikat o błędzie, informujący np. o nieprawidłowym odwołaniu do pamięci, jest często niejasny). Informacje o wyjątku są dostarczane przez system w formie obiektu klasy, której klasą bazową jest `System.Exception`. Środowisko .NET Framework posiada kolekcję wyjątków dziedziczonych po tej klasie. Klasa `Exception` oferuje metody, które możemy wykorzystać, aby podać dokładne informacje o zgłoszonym wyjątku (czyli o obiekcie klasy `Exception` lub jej klasy potomnej np. klasy `DivideByZeroException`), co pokazuje przykład 2.27.

Przykład 2.27. Wyświetlenie informacji o obiekcie klasy Exception

```
int X, Y, Z; X = 10; Y = 0;
try{
```

```
    Z = X / Y;
}
catch (Exception e)
{
   //pokaż informacje o wyjątku - skorzystaj z właściwoci
   //Message dla obiektu e klasy Exception:
   Console.WriteLine(e.Message);
}
```

Parametr `e` klasy `Exception` w bloku `catch` jest opcjonalny, ale jeżeli znamy klasę wyjątku jak w przykładzie 2.27, możemy napisać zamiast `Exception`:

```
catch (DivideByZeroException e) //....
```

W przykładzie 2.27 skorzystaliśmy z właściwości `Message` obiektu `Exception`. Klasa `Exception` udostępnia również inne właściwości:

- `Data` - dodatkowe informacje na temat źródła wystąpienia wyjątku,
- `HelpLink` – zwraca adres URL do pliku pomocy szczegółowo opisującego błąd,
- `Message` – komunikat błędu,
- `Source` – umożliwia odczytanie lub przypisanie nazwy aplikacji lub obiektu, w którym wystąpił błąd,
- `StackTrace` – zawiera łańcuch znakowy identyfikujący sekwencję wywołań, która spowodowała błąd,
- `InnerException` – można użyć w celu uzyskania informacji o poprzednim wyjątku, który spowodował zaistnienie obecnego wyjątku,
- `TargetSite` – umożliwia odczytanie metody, w której wystąpił błąd.

Znając klasy ewentualnych wyjątków, mamy możliwość ich selektywnej obsługi. W przykładzie 2.28 podjęto próbę przypisania wartości dziesiątemu elementowi tablicy, której deklaracja przewiduje tylko 6 elementów. Uruchomienie tego kodu wyrzuci wyjątek `IndexOutOfRangeException` i sterowanie zostanie przekazane do bloku `catch`, przechwytującego wyjątek tej klasy. W przypadku wystąpienia innego wyjątku wykona się kod z kolejnego bloku `catch`.

Przykład 2.28. Selektywna obsługa wyjątków

```
string[] Abc = new string[6];
try
{
   Abc[10] = "Test";
}
```

```
catch (IndexOutOfRangeException e)
{
   Console.WriteLine("Za duży indeks!");
}
catch
{
   Console.WriteLine("Inny błąd");
}
```

Metoda, która „wyrzuca” wyjątek stosuje instrukcję `throw`, np.:

```
try {
       throw new IndexOutOfRangeException();
    }
catch (Exception e)
    {
       Console.WriteLine(e.Message);
     }
```

Na potrzeby programu można zadeklarować własne klasy obsługi wyjątków, np. do walidacji danych:

```
try
{
   //pobierz dane
}
catch
{
   throw new ZleDaneException(); //zwracamy obiekt naszej klasy
}
```

Nasza klasa `ZleDaneException` powinna decydować, co program ma w danej sytuacji zrobić (np. wyświetlić komunikat na konsoli lub wyświetlić komunikat w nowym okienku itp.). W konstruktorze klasy do odpowiednich właściwości przypisywane są dane, które mogą pomóc w ewentualnym odszukaniu i naprawie usterki. Zalecane jest deklarowanie 3 konstruktorów dla klas wyjątków, każdy z innymi parametrami, np.:

```
public class ZleDaneException : System.Exception
{
  public ZleDaneException(string Message) : base(Message)
  { this.Source = "ZleDaneException";
    // inne instrukcje
  }
  public ZleDaneException()
  {
    //instrukcje
  }
  public ZleDaneException(string Message, Exception inner) :
                                          base(Message, inner)
```

```
    {
      //instrukcje
    }
}
```

Ogólnie rozróżnia się dwa rodzaje wyjątków:

- na poziomie systemu `System.SystemException` (wyjątki na poziomie biblioteki klas bazowych .NET), np.: `ArgumentOutOfRangeException`, `IndexOutOfRangeException`, `StackOverFlowException` itp.
- wyjątki na poziomie aplikacji: `System.ApplicationException`.

2.2.6. Czas życia obiektów

W przeciwieństwie do C++, programiści C# nigdy bezpośrednio nie usuwają obiektów z pamięci (w C# nie istnieje słowo kluczowe delete). Obiekty w .NET są umieszczane w regionie pamięci zwanym **zarządzaną stertą** i czekają na automatyczne usunięcie przez środowisko uruchomieniowe w nieokreślonej przyszłości. Użycie słowa kluczowego `new` powoduje umieszczenie obiektu na zarządzanej stercie. Środowisko uruchomieniowe .NET niszczy obiekt, gdy nie jest już potrzebny (tzn. gdy w aktualnym zasięgu nie ma już więcej referencji do obiektu), np. w metodzie `Main()` mamy:

```
static void Main(string[] args)
{
  //utwórz lokalną zmienną Wektor
  Wektor w=new Wektor();
  //...
  Console.ReadKey();
}
//jeśli w jest jedyną referencją do typu Wektor
//to można ją zwolnić w chwili wyjścia poza zasięg metody
```

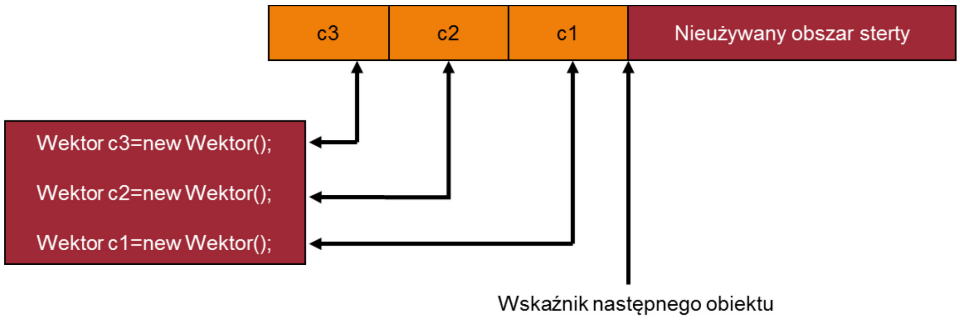
Nic jednak nie gwarantuje, że pamięć zajęta przez obiekt zostanie odzyskana w chwili zakończenia metody `Main()`. Wiadomo jedynie, że w momencie przeprowadzania przez środowisko uruchomieniowe .NET następnego ‘zbierania śmieci’ (ang. garbage collection), obiekt `w` jest gotowy do zniszczenia.

Sterta zarządzana to coś więcej niż fragment pamięci wykorzystywany przez środowisko w procesie czyszczenia pamięci. .NET łączy bloki wolnej pamięci w celu optymalizacji – do obsługi tego zadania wykorzystywany jest wskaźnik (tzw. wskaźnik nowego obiektu) określający gdzie dokładnie zostanie umieszczony na stercie następny obiekt.

Tworzenie nowego obiektu związane jest z przeprowadzeniem następujących działań (Rys.2.11):

- obliczenia całkowitej pamięci dla obiektu,

- sprawdzenia czy jest miejsce na stercie na umieszczenie obiektu (jeśli tak uruchamiany jest konstruktor obiektu i zwracana jest referencja do pamięci zajmowanej przez obiekt identyczna z pozycją wskazywaną przez wskaźniki nowego obiektu),
- przed zwróceniem referencji środowisko uaktualnia wskaźnik do nowego obiektu, tak aby wskazywał następną wolną pozycję na stercie.



Rys.2.11. Tworzenie nowego obiektu na stercie

Czyszczenie pamięci przeprowadzane jest w przypadku gdy na stercie nie ma wystarczającej wolnej pamięci dla nowego obiektu. Podczas czyszczenia pamięci sprawdzane są wszystkie obiekty znajdujące się na stercie (pod kątem zbędnych referencji) w celu określenia czy nadal są wykorzystywane przez aplikację i pamięć jest porządkowana – jest to realizowane automatycznie w sposób niewidoczny.

Trudno jest przewidzieć dokładnie kiedy obiekt zostanie usunięty z pamięci – ułatwia to programowanie, ale pozostają resztki obiektów utrzymujących powiązania z nie zarządzanymi zasobami (np. połączeniami z bazą danych) dłużej niż jest to potrzebne. Jedną z decyzji pozostawionych programiście C# jest stwierdzenie czy klasa ma zawierać metodę `System.Object.Finalize()`, czy nie (domyślnie ta metoda nie wykonuje niczego).

W większości przypadków nie ma potrzeby tworzenia własnej implementacji metody `Finalize()`. Jeśli jednak zachodzi taka potrzeba to C# nie pozwala bezpośrednio przesłonić tej metody i zamiast tego trzeba skorzystać z destruktora (jak w C++), np.:

```
class FinalizedWektor
{    ~FinalizedWektor()
     {  Console.WriteLine("Likwidacja obiektu Wektor");
     }
}
```

2.3. Zaawansowane mechanizmy obiektowe

2.3.1. Delegaty, metody anonimowe i zdarzenia

Delegat to nowy typ danych; może być zadeklarowany w klasie jako typ zagnieżdżony lub poza klasą. Deklaracja delegata jest podobna do deklaracji metody uzupełniona słówkiem kluczowym `delegate`. Delegat nie posiada implementacji np.:

```
public delegate int Delegat1(int X);
```

Wykorzystanie delegata związane jest z utworzeniem nowej zmiennej wskazującej na metodę, której nagłówek jest taki sam jak nagłówek delegata (tzn. metoda musi mieć takie same parametry i zwracać wynik tego samego typu co delegat, np.:

```
static int metoda1(int X)  { return X * X;  }
static int metoda2(int X)  { return X*X*X;  }
```

Delegaty:

- mogą zostać zainicjalizowane jak w przypadku zwykłych klas – należy wówczas utworzyć obiekt delegata i jako parametr podać w nim nazwę metody, na którą ma wskazywać np.:
  ```
  Delegat1 d1 = new Delegat1(metoda1);
  Delegat1 d2 = new Delegat1(metoda2);
  ```
  i wówczas wywołanie `d1()` jest równoznaczne z wywołaniem metody `metoda1()` a wywołanie `d2()` jest równoważne wywołaniu `metoda2()`.
- można też utworzyć zmienną wskazującą na delegat i przypisać do niej wartość:
  ```
  Delegat1 d3 = metoda1;
  ```

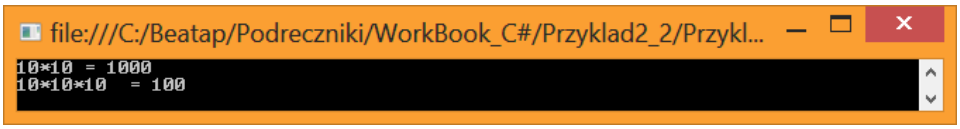
Przykład 2.29 pokazuje sposób deklaracji i wykorzystania delegata w programie.

Przykład 2.29. Wykorzystanie delegata

```
using System;

//deklaracja delegata:
public delegate int Delegat1(int X);
class Program
{
    //deklaracja metod, które mozna przypisać delegatowi:
    static int metoda1(int X) { return X * X; }
    static int metoda2(int X) { return X * X * X; }
```

```
    static void Main(string[] args)
    {
        Delegat1 d1 = new Delegat1(metoda2);
        Console.WriteLine("10*10 = " + d1(10));
        d1 = metoda1;
        Console.WriteLine("10*10*10  = " + d1(10));
        Console.ReadKey();
    }
    }
```

Wynik uruchomienia programu z przykładu 2.29 przedstawia rysunek 2.12.



Rys. 2.12. Wynik działania przykładu 2.29.

Przykład 2.30 pokazuje, że parametrem metody może być delegat. Wówczas wywołanie metody z parametrem będącym delegatem powoduje, że w rzeczywistości wywoływana jest metoda, do której odnosi się delegat.

Przykład 2.30. Delegat jako parametr metody

```
using System;

//deklaracja delegata:
public delegate void Delegat2(string Wiadomosc);
class Program
{
    //deklaracja metody z parametrem delagatem:
    static void Info(int X, int Y, Delegat2 d1)
    {
        d1("Rezultat: " + (X + Y));
    }
    static void m1(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        Delegat2 d = m1;
        //wywołanie metody Infor z parametrem – metodą m1
```

```
        //przypisaną delegatowi d:
       Info(25, 35, d);
        Console.ReadKey();
    }
   }
```

Kolejny przykład 2.31 również przedstawia zastosowanie delegata jako parametru metody. Wynik działania programu pokazany jest na rysunku 2.13.

Przykład 2.31. Delegat jako parametr metody

```
using System;
public delegate int Delegat1(int X);
class Program
{
  static int f2(int X) { return X * X; }
  static int f3(int X) { return X * X * X; }

  static void Main(string[] args)
  {
   Delegat1 d1 = new Delegat1(f2);
   int n = 0; string info = "";
   Console.Write("Kwadrat (n=2) czy sześcian (n=3)? Podaj n:");
   n = Console.Read(); Console.WriteLine("n=" + (char)n);
   switch (n)
   {
   case '3': d1 = f3; info = "Sześcian o boku 10, objetosc = ";
             break;
    default: d1 = f2; info = "Kwadrat o boku 10, pole = ";
             break;
   }
  Console.WriteLine(info + d1(10));
  Console.ReadKey();
  }
}
```

Deklaracja delegata może być realizowana także poprzez przypisanie metody anonimowej (bez nazwy), która ma być wywoływana po użyciu delegata np.:

```
Delegat1 d2 = delegate(int X)
{
   Console.WriteLine("Kwadrat liczby:"+X*X);
   return X * X;
};
```

W takim przypadku wywołanie:

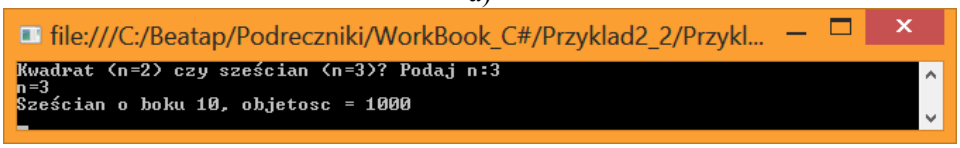
```
d2(20);
```

do zmiennej `d2` (typu `Delegat1`) przypisuje kod, wykonywany w momencie wywołania `d2()`.

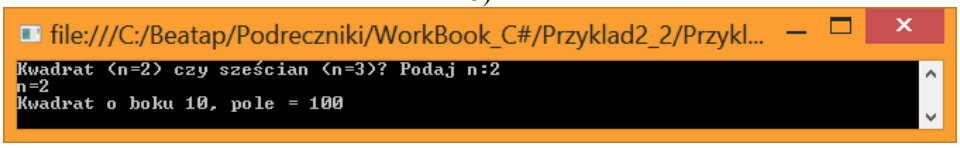
Metody anonimowe tworzone są podobnie jak zwykłe metody, ale nazwę metody zastępuje słowo kluczowe `delegate` (konieczny jest też średnik po klamrze zamykającej kod metody anonimowej!).

Zamiast tworzyć metodę i dodawać ją do delegata przez nazwę, tworzy się metodę bez nazwy (ale ze słowem kluczowym `delegate`) i od razu ją rejestruje (czyli "chowa" w delegacie). Metodę tą można wywołać tylko korzystając z delegata. Jest to bardzo wygodne rozwiązanie, kiedy szybko trzeba stworzyć i przekazać dalej krótką metodę.

a)



b)



Rys. 2.13. Przykładowe wyniki działania programu 2.31.

Istnieje możliwość przypisywania do delegata więcej niż jednej metody – wtedy przy wywołaniu delegata wywołane zostaną wszystkie przypisane do niego metody. Przypisywanie dodatkowych metod do delegata realizuje operator `+=` oraz `-=` (Przykład 2.32). Są to tzw. delegaty złożone.

Przykład 2.32. Przypisywanie do delegata więcej niż jednej metody

```
using System;

public delegate int Delegat1(int X);

class Program
{
```

```
    static int f2(int X)
    {
       Console.WriteLine("Kwadrat:"); return X * X;
    }
    static int f3(int X)
    {
       Console.WriteLine("Sześcian:");
       return X * X * X;
    }

    static void Main(string[] args)
    {
        Delegat1 d1 = f2;
        d1 += f3;
        Console.WriteLine("f2, f3 :" + d1(10));
        d1 -= f3;
        Console.WriteLine("f2: " + d1(10));
        Console.ReadKey();
    }
}
```

Delegaty (czasem też nazywane delegacjami) są to zatem obiekty, w których przechowujemy referencje do metod. Dla znających C++, delegaty są podobne do wskaźników na funkcję. Delegata możemy traktować jako metodę, która nie ma swojego ciała, ale może wywoływać ciała innych metod.

Wyrażenie lambda

Wróćmy po raz kolejny do definicji metody anonimowej. Przy założeniu, że mamy definicję delegata:

```
public delegate int Delegat1(int X);
```

możemy do niego ‘schować’ metodę anonimową np.:

```
Delegat1 d2 = delegate(int X)
{
   Console.WriteLine("Kwadrat liczby:"+X*X);
   return X * X;
};
```

Zamiast tworzyć metodę anonimową możemy skorzystać z tzw. wyrażenia lambda i powyższy kod zapisać w równoważnej postaci:

```
Delegat1 d2 = (int X)=>
{
   Console.WriteLine("Kwadrat liczby:" + X * X);
   return X * X;
};
```

Warto zauważyć, że jedyne co zmieniliśmy w stosunku do poprzedniego zapisu, to zastąpienie słowa kluczowego **`delegate`** symbolem: **`=>`**.

Wyrażenia lambda pozwalają szybciej tworzyć metody anonimowe. Zapis wyrażenia można jeszcze skrócić do postaci:

```
Delegat1 d2 = X=>X*X;
```

i korzystać z delegata `d2` w sposób następujący.:

```
Console.WriteLine("Kwadrat liczby 5: " + d2(5) );
```

Gdy metoda (tzn. wyrażenie lambda) przyjmuje jeden argument nie musimy otaczać go nawiasami. Gdy w ciele metody istnieje tylko jedna instrukcja, nie musimy korzystać z klamer i zrezygnowaliśmy też z `return`.

Zdarzenia

Ważnym elementem programowania, zwłaszcza aplikacji z graficznym interfejsem użytkownika są zdarzenia. Delegaty deklaruje się z użyciem słowa kluczowego `delegate`, a zdarzenia z użyciem słowa `event`. Szerzej na temat obsługi zdarzeń czytelnik dowie się w rozdziale 3.

2.3.2. Kolekcje i typy generyczne

Indeksatory

Przy pracy z obiektami często zachodzi potrzeba wykorzystania mechanizmu indeksowania. Taki mechanizm pozwala odwoływać się do obiektu jak do zwykłych tablic (przy pomocy symboli [ ]) za pomocą specjalnego obiektu indeksatora zdefiniowanego w danej klasie. Indeksator można wykorzystać przy odwoływaniu się do elementów tablicy (ogólniej listy obiektów):

```
TypListy Lista1 = new TypListy();
//wykorzystanie indeksatora z parametrem typu string:
Lista1["klucz1"] = "Tekst1";
```

Dodatkowo można kontrolować proces pobierania oraz przypisywania takich elementów. Indeksatory w klasie deklaruje się podobnie jak właściwości, tylko zamiast nazwy właściwości stosuje się słowo kluczowe `this`, np.:

```
class TypListy
{
    public string this[string index]
    { get; set; }
}
```

Indeksator:

- deklaruje się z użyciem słowa kluczowego `this`,
- musi posiadać typ zwrotny,

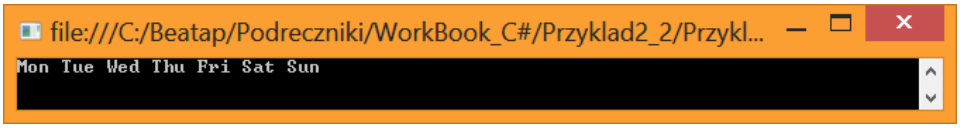
- musi posiadać akcesor `get` i/lub `set`,
- w jednej klasie może być wiele indeksatorów, pod warunkiem że będą miały różne parametry (w przykładzie powyżej parametr indeksatora jest typu `string`).

Przykład 2.33 przedstawia klasę z indeksatorem oraz jej wykorzystanie w programie.

Przykład 2.33. Klasa z indeksatorem

```
using System;
class TypListy
{   //pola klasy:
    private string[,] Dni; //tablica dwuwymiarowa
    private int Id; //określa język (polski/angielski
    //konstruktor klasy:
    public TypListy(string jezyk)
    {
        //inicjalizacja tablicy nazwami dni
        //w języku polskim i angielskim:
        Dni = new string[,]
              {  {"Pn", "Wt", "Śr", "Czw", "Pt", "So", "Nd"},
               {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"}
              };
        //język ustalany w zależności od parametru konstruktora
        Id = jezyk == "pl" ? 0 : 1;
    }
    //indeksator z parametrem typu int:
    public string this[int index]
    {
        get { return Dni[Id, index]; }
    }
}

class Program
{
    static void Main(string[] args)
    {
        TypListy l1 = new TypListy("en");
       //wykorzystanie indeksatora w pętli:
        for (int i = 0; i < 7; i++)
        {
            Console.Write("{0} ", l1[i] );
        }
        Console.ReadKey();
    }
}
```

Wynik działania programu z przykładu 2.33 przedstawia rysunek 2.14.



Rys. 2.14. Przykładowe wyniki działania programu 2.33.

W języku C# nie są dostępne tablice asocjacyjne. Można zamiast tego skorzystać z indeksatorów, które dopuszczają użycie łańcuchów w nazwach indeksu. To rozwiązanie jest jednak nieco problematyczne i lepiej używać list i słowników, o których będzie mowa w dalszej części rozdziału. Nie ma również możliwości przypisania danych do indeksatora (ang. *indexer cannot be assigned to -- it is read only*). Indeksatory nie mogą być opatrzone modyfikatorem `static`. Mechanizm indeksowania można traktować jako rozbudowany system tablic, ponieważ dają one większą kontrolę nad odczytywaniem danych.

Przykład 2.34 przedstawia definicję klasy z indeksatorem o parametrze łańcuchowym.

Przykład 2.34. Indeksator z parametrem łańcuchowym

```
using System;
class TypListyl
{
    private string[] DniKrotko;
    private string[] Dni;
    public TypListyl()
    {
        DniKrotko = new string[] { "Pn", "Wt", "Śr", "Czw", "Pt",
                                   "So", "Nd" };
        Dni = new string[] { "Poniedziałek", "Wtorek", "Środa",
                    "Czwartek", "Piątek", "Sobota", "Niedziela" };
    }
    //indeksator z parametrem typu string:
    public string this[string index]
    {
        get
        {
            int IndexOf = System.Array.IndexOf(DniKrotko, index);
            return Dni[IndexOf];
        }
```

```
        set
        {
            int IndexOf = System.Array.IndexOf(DniKrotko, index);
            Dni[IndexOf] = value;
        }
    }
}


class Program
{
    static void Main(string[] args)
    {
        TypListyl l1 = new TypListyl();
        //wykorzystanie indeksatora z parametrem string:
        Console.Write("\n" + l1["Śr"]);
        l1["Śr"] = "Wednesday";
        Console.Write("\n" + l1["Śr"]);

        Console.ReadKey();
    }
}
```

Kolekcje

Tablice w języku C# mają wiele ograniczeń, m.in.:

- brak jest możliwości nadawania określonych indeksów dla elementów tablicy (obowiązuje indeksowanie od 0),
- trudności w usuwaniu lub dodawaniu kolejnych elementów tablicy,
- deklarując tablicę należy podać z góry jej rozmiar lub zainicjalizować jej elementy.

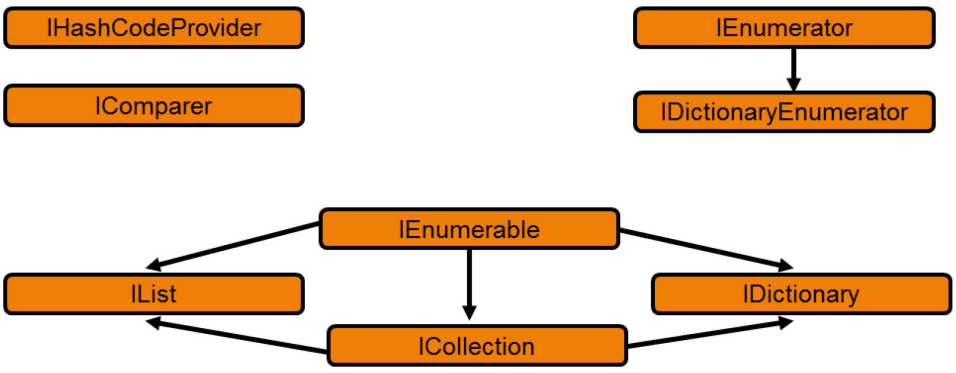
Kolekcje to bardziej zaawansowany mechanizm przechowywania zbioru różnych obiektów. Odpowiednie klasy dostępne w .NET dostarczają bardziej zaawansowanych metod służących do operacji na zbiorze elementów kolekcji. Takie klasy służące do operowania na zbiorach danych zawarte są w przestrzeni nazw `System.Collections`.

Klasa `Array` należy do przestrzeni nazw `System`, ale implementuje również metody interfejsów związanych z kolekcjami: `ICollection, ICloneable, IList` oraz `IEnumerable`. `Array` jest najprostszą klasą wykorzystującą mechanizm kolekcji.

Interfejsy, zdefiniowane wewnątrz przestrzeni `System.Collections`, definiujące odpowiednie metody (implementowane w klasach reprezentujących kolekcje) to:

- `IEnumerable` – udostępnia interfejs `IEnumerator` dla danego obiektu (umożliwia przeglądanie elementów kolekcji w jednym kierunku),
- `ICollection` – dziedziczy metody `IEnumerable` i definiuje ogólne cechy charakterystyczne klasy modelującej zbiór (np.rozmiar kolekcji),
- `IList` – umożliwia dodawanie, usuwanie oraz indeksowanie elementów w listach obiektów,
- `IDictionary` – daje obiektom możliwość przedstawienia zawartości w postaci par klucz (nazwa) - wartość,
- `IEnumerator` – zwykle wykorzystywany do przeprowadzania iteracji podtypów za pomocą instrukcji `foreach`,
- `IComparer` – pozwala porównać dwa obiekty,
- `IDictionaryEnumerator` – definiuje właściwości umożliwiające pobieranie kluczy i wartości słowników (obiektów klas `IDictionary`),
- `IHashCodeProvider` – definiuje metodę zwracającą unikalny kod danego obiektu.

Relacje pomiędzy interfejsami kolekcji przedstawia rysunek 2.15.



Rys. 2.15. Relacje pomiędzy interfejsami implementowanymi w kolekcjach

Podstawowe metody i właściwości poszczególnych interfejsów, implementowane w klasach kolekcji to:

- `IEnumerable` – bazowy interfejs definiuje tylko jedną metodę:
  - `GetEnumerator()` – zwraca egzemplarz klasy, który implementuje interfejs `IEnumerable`;

- `ICollection` – interfejs dziedziczy po `IEnumerable`, definiuje:
  - właściwość `Count` – zwraca liczbę elementów w kolekcji,
  - `metodę CopyTo()`– umożliwia skopiowanie obiektu zawartego w danej kolekcji do tablicy (`System.Array`). Metoda ta posiada dwa parametry: nazwę tablicy (do której zostaną skopiowane dane) i indeks (liczony od zera), od którego rozpocznie się kopiowanie;
- `IDictionary` – definiuje metody oraz właściwości umożliwiające korzystanie ze słowników (umożliwiają przechowywanie danych, do których dostęp uzyskuje się, podając odpowiedni klucz - nie indeks, jak to jest w tablicach oraz listach. Interfejs, podobnie jak `IList`, definiuje właściwości `IsFixedSize`, `IsReadOnly` i `Items` oraz metody `Add()`, `Clear()`, `Contains()`, `Remove()`. Dodatkowo posiada dwie właściwości:
  - `Keys` – zwraca kolekcję składającą się z kluczy danego słownika,
  - `Values` – zwraca kolekcję składającą się z wartości danego słownika;
- `IEnumerator` – definiuje:
  - właściwość `Current` – zwraca bieżący element kolekcji,
  - metodę `MoveNext()` – do przechodzenia do kolejnego elementu kolekcji,
  - metodę `Reset()` – umożliwia przejście do pierwszego elementu;
- `IList` – dziedziczy po interfejsach `ICollection` oraz IEnumerable, bazowy interfejs dla list – definiuje metody oraz właściwości umożliwiające dodawanie i kasowanie elementów listy:
  - `IsFixedSize` – właściwość określa, czy lista ma stały rozmiar,
  - `IsReadOnly` – właściwość określa, czy dana lista zwraca swoje wartości jedynie do odczytu,
  - `Item` – właściwość umożliwia pobranie lub ustawienie elementów pod danym indeksem,
  - `Add()` – metoda umożliwia dodanie elementów do listy,
  - `Clear()` – usuwa elementy z danej listy,
  - `Contains()` – sprawdza, czy lista zawiera daną wartość,
  - `IndexOf()` – zwraca indeks danej wartości,
  - `Insert()` – umożliwia wstawienie elementu pod wskazaną pozycję,
  - `Remove()` – usuwa pierwsze wystąpienie danego elementu,
  - `RemoveAt()` – usuwa element określony danym indeksem.

Podstawowe klasy (implementujące powyższe interfejsy) z przestrzeni `System.Collections` to:

- `ArrayList` – tablica obiektów o dynamicznie ustalonym rozmiarze,
- `Hashtable` – reprezentuje zbiór obiektów identyfikowanych za pomocą numerycznego klucza,
- `Queue` – reprezentuje kolejkę FIFO (ang. First In First Out),
- `SortedList` – lista uporządkowana, w której można sięgać do elementów o danym indeksie,
- `Stack` – stos, reprezentuje kolejkę LIFO (ang. Last In First Out).

Stos – klasa Stack

Stos w języku C# jest reprezentowany przez klasę `Stack` z przestrzeni `System.Collections.`

Klasa `Stack`:

- reprezentuje stos, czyli strukturę danych, w której można dodawać kolejne elementy, a usunąć tylko ten dodany na końcu,
- implementuje interfejsy `ICollection`, `IEnumerable`, `ICloneable`,
- nie umożliwia usuwania dowolnego elementu stosu (w przeciwieństwie do list, które oferują taką możliwość),
- w konstruktorze klasy można określić przewidywalną liczbę elementów lub wywołać konstruktor bezparametrowy (domyślna liczba elementów to 10 - w razie potrzeby zostaje automatycznie zwiększona),
- udostępnia następujące metody:
  - `Push()` – umieszcza na stosie kolejny element,
  - `Peek()` – pobiera ostatni element stosu,
  - `Pop()` – zwraca ostatni element, po czym go usuwa.

Aby wyświetlić elementy ze stosu należy zadeklarować zmienną wskazującą na interfejs `IEnumerator` i użyć metody `GetEnumerator()`, która zwraca implementowany obiekt, jak pokazuje przykład 2.35. Zwróćmy uwagę, że aby korzystać z klas kolekcji należy dodatkowo wskazać przestrzeń nazw:

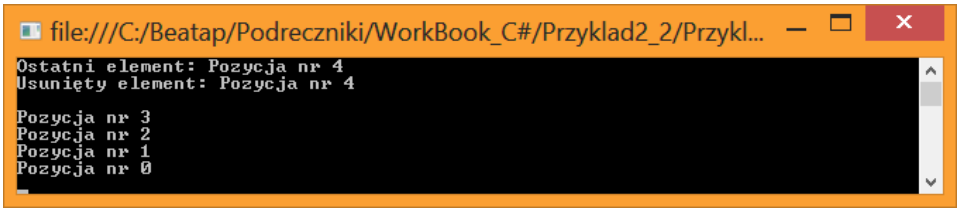
```
using System.Collections;
```

Wynik działania przykładu przedstawia rysunek 2.16.

Przykład 2.35. Praca ze stosem

```
using System;
using System.Collections;

class Program
{
```

```
    static void Main(string[] args)
    {
        Stack Stos1 = new Stack(); //utwórz oobiekt klasy Stack
        // dodaj elementy na Stos1
        for (int i = 0; i < 5; i++)
         {
               Stos1.Push("Pozycja nr " + i);
         }
         Console.WriteLine("Ostatni element: " + Stos1.Peek() );
         Console.WriteLine("Usunięty element: " + Stos1.Pop() );
         Console.WriteLine();
        //pobierz enumerator dla stosu Stos1:
        IEnumerator ie = Stos1.GetEnumerator();
        //wykorzystaj enumerator ie do wyświetlenia elementów stosu:
          while ( ie.MoveNext() )
         {
             Console.WriteLine( ie.Current.ToString() );
         }
        Console.ReadKey();
    }
}
```



Rys. 2.16. Wynik działania programu z przykładu 2.35

Kolejka – klasa Queue

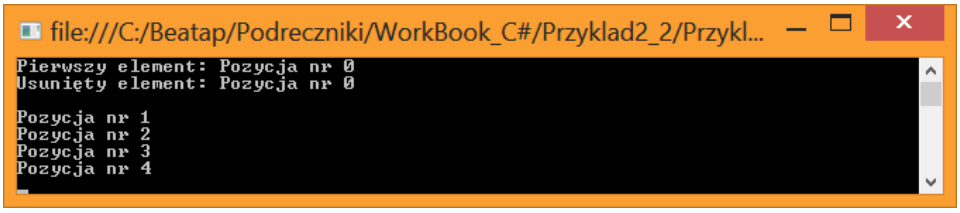
Kolejną klasą z przestrzeni kolekcji jest klasa reprezentująca kolejkę `Queue`, która:

- implementuje te same interfejsy, co klasa `Stack`,
- działa zgodnie z zasadą FIFO (ang. First-in First-out),
- nie posiada metody umożliwiającej usunięcie dowolnego elementu,
- udostępnia metody:
  - `Enqueue()` – dodaje obiekt na koniec kolejki,
  - `Dequeue()` – usuwa i zwraca obiekt z początku kolejki,
  - `Peek()` – zwraca obiekt z początku kolejki bez usuwania go.

Aby usunąć ostatni element kolejki, najpierw należy usunąć wszystkie pozostałe utworzone wcześniej, co przedstawia przykład 2.36. Wynik działania przykładu pokazuje rysunek 2.17. Aby wyświetlić elementy kolejki również potrzebujemy utworzyć enumerator dla kolejki (analogicznie jak w przypadku stosu).

Przykład 2.36. Praca z kolejką

```
using System;
using System.Collections;
class Program
{
    static void Main(string[] args)
    {
        Queue Kolejka1 = new Queue();
        for (int i = 0; i < 5; i++)
        {
            Kolejka1.Enqueue("Pozycja nr " + i);
        }
      Console.WriteLine("Pierwszy element: " + Kolejka1.Peek() );
      Console.WriteLine("Usunięty element: " + Kolejka1.Dequeue() );
      Console.WriteLine();
        IEnumerator ie = Kolejka1.GetEnumerator();
        while ( ie.MoveNext() )
        {
            Console.WriteLine( ie.Current.ToString() );
        }
        Console.ReadKey();
    }
}
```



Rys. 2.17. Wynik działania programu z przykładu 2.36

Klasa ArrayList

Klasa `ArrayList` jest najlepszą alternatywą pomiędzy klasą `Stack` a `Queue`.

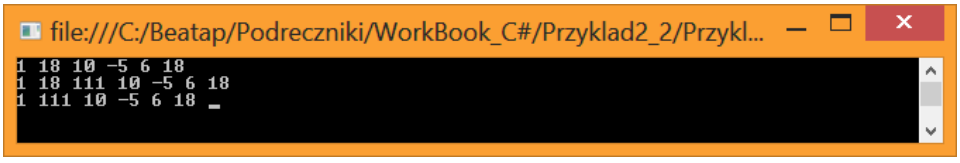
Klasa `ArrayList`:

- posiada metody jak klasa `Array` – działanie z nią jest podobne do obsługi zwykłych tablic, ale można do niej dodawać elementy różnych typów,
- implementuje interfejsy: `IList`, `ICollection`, `IEnumerable`, `ICloneable`.

Pracę z obiektem klasy `ArrayList` przedstawia przykład 2.37. Wynik działania kodu z tego przykładu pokazuje rysunek 2.18.

Przykład 2.37. Praca z ArrayList

```
using System;
using System.Collections;
class Program
{
    static void Main(string[] args)
    {
        ArrayList liczby = new ArrayList();
        //dodaj tablicę elementów do listy:
        liczby.AddRange(new int[] { 1, 18, 10, -5, 6 });
        //wstaw nowy element (na końcu listy):
        liczby.Add(18);
        //wypisz bieżące wartości:
        foreach ( int m in liczby ) Console.Write( m + " ");
        Console.WriteLine();
        //wstaw element na pozycji o indeksie 2
        liczby.Insert(2, 111);
        //ponownie wypisz bieżące wartości:
        foreach (int m in liczby) Console.Write(m + " ");
        Console.WriteLine();
        liczby.Remove(18); //usuń pierwsze wystąpienie obiektu 18
        //ponownie wypisz bieżące wartości:
        foreach (int m in liczby)
            Console.Write(m + " ");
        Console.ReadKey();
    }
}
```



Rys. 2.18. Wynik działania programu z przykładu 2.37

Typy generyczne

Typy generyczne to nowy element począwszy od środowiska .NET Framework 2.0. Poprzednia wersja 1.1 nie posiadała możliwości wykorzystania typów generycznych. Typy te są wzorowane na technologii szablonów (`templates`) z języka C++.

Na przykład, wracając do klasy `ArrayList`, dodanie elementów do kolekcji tego typu i wyświetlenie ich realizuje fragment kodu:

```
ArrayList l1 = new ArrayList();
l1.Add("Ania");
l1.Add("Jasio");
l1.Add(123);
l1.Add(4.56);
for (int i = 0; i < l1.Count; i++)
{
    Console.WriteLine("Indeks: {0} wartość: {1}", i, l1[i]);
}
```

Parametr metody `Add()` jest typu `Object` (a wszystkie typy .NET Framework dziedziczą po tej klasie), dzięki czemu można do listy dodawać elementy różnych typów. Przy każdym wywołaniu metody `Add()` klasy `ArrayList` program wykonuje opakowywanie (ang. boxing) typów, a przy wyświetlaniu - odpakowywanie (ang. unboxing) . Przy dużej ilości danych trwa to dość długo.

Jeśli w kolekcji mają być umieszczone dane tego samego typu - lepiej wykorzystać klasy oferowane przez przestrzeń nazw:

`System.Collection.Generic`,

która zawiera odpowiedniki klas `Stack` oraz `Queue`, a które działają szybciej na danych tego samego typu.

W przestrzeni nazw `System.Collection.Generic` nie ma klasy `ArrayList` – jej generycznym odpowiednikiem jest klasa `List`. Przy deklarowaniu i tworzeniu obiektu klasy `List` (w miejsce `T`) trzeba podać typ danych listy (np. `int, double, string`):

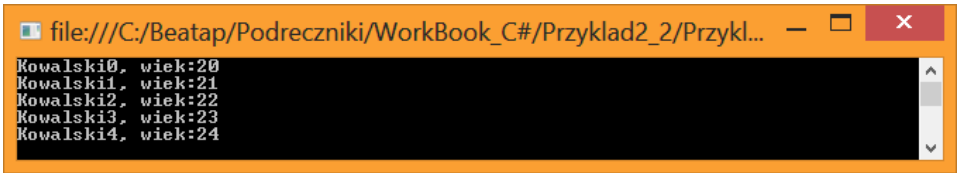
```
List<T> Lista1 = new List<T>();
```

Przykład 2.38 przedstawia pracę z listą struktur `Student`. Wynik działania kodu pokazuje rysunek 2.19.

Przykład 2.38. Praca z listą struktur

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

```
public struct Student
{
    public string Nazwisko;
    public int Wiek;
}
class Program
{
    static void Main(string[] args)
    {
        List<Student> Lista1 = new List<Student>();
        Student s1 = new Student();
        for (int i = 0; i < 5; i++)
        {
            s1.Nazwisko = "Kowalski" + i;
            s1.Wiek = 20 + i;
            Lista1.Add(s1);
        }
        for (int i = 0; i < Lista1.Count; i++)
        {
            Console.WriteLine( Lista1[i].Nazwisko + ", wiek:"
                + Lista1[i].Wiek );
        }
        Console.ReadKey();
    }
}
```



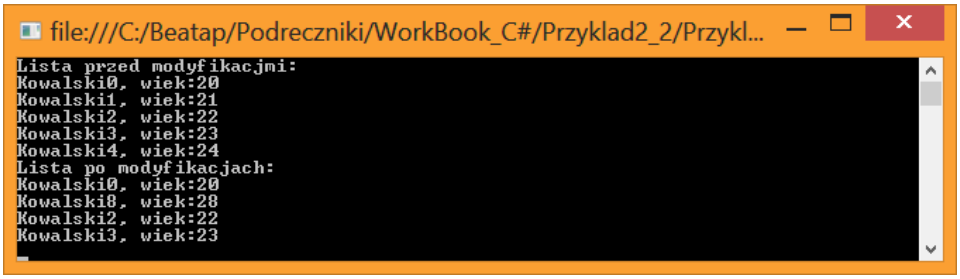
Rys. 2.19. Wynik działania programu z przykładu 2.38

W przypadku klasy `List` możliwe są modyfikacje listy np. wstawianie elementów do listy, co pokazuje przykład 2.39 (modyfikacja przykładu 2.38). Wynik po zmianach widoczny jest na rysunku 2.20.

Przykład 2.39. Modyfikacja elementów listy

```
    static void Main(string[] args)
    {
        List<Student> Lista1 = new List<Student>();
        Student s1 = new Student();
```

```
        for (int i = 0; i < 5; i++)
        {
            s1.Nazwisko = "Kowalski" + i;
            s1.Wiek = 20 + i;
            Lista1.Add(s1);
        }
        Console.WriteLine("Lista przed modyfikacjmi:");
        for (int i = 0; i < Lista1.Count; i++)
        {
            Console.WriteLine(Lista1[i].Nazwisko + ", wiek:" +
                Lista1[i].Wiek);
        }

        Lista1.RemoveAt(1); //usuń element z pozycji 1
        Lista1.RemoveAt(3); //usuń element z pozycji 3
        s1.Nazwisko = "Kowalski8";
        s1.Wiek = 28;
        Lista1.Insert(1, s1); //dodaj element na pozycję 1
        Console.WriteLine("Lista po modyfikacjach:");
        for (int i = 0; i < Lista1.Count; i++)
        {
            Console.WriteLine(Lista1[i].Nazwisko + ", wiek:" +
                Lista1[i].Wiek);
        }

        Console.ReadKey();
    }
```



Rys. 2.20. Wynik działania programu z przykładu 2.39

Kolejną klasą z przestrzeni `System.Collections.Generic` jest klasa słownikowa `Dictionary`. Zamiast indeksu numerycznego wykorzystuje się tu klucz. Każdy element z kolekcji indeksowany jest unikatowym kluczem i posiada wartość określonego typu. Najczęściej klucz i wartość są typu `string`. Deklarując instancję klasy, należy podać zarówno typ klucza, jak i wartości np.: `Dictionary<string, string> Slownik;`

```
Slownik=new Dictionary<string,string>;
```

Korzystanie ze słowników odbywa się podobnie jak ze zwykłych list, ale słowniki dają możliwość określenia typu klucza (w przypadku list klucz jest zawsze typu `int`). W przypadku korzystania ze słownika jak z listy, należy utworzyć zmienną wskazującą na klasę:

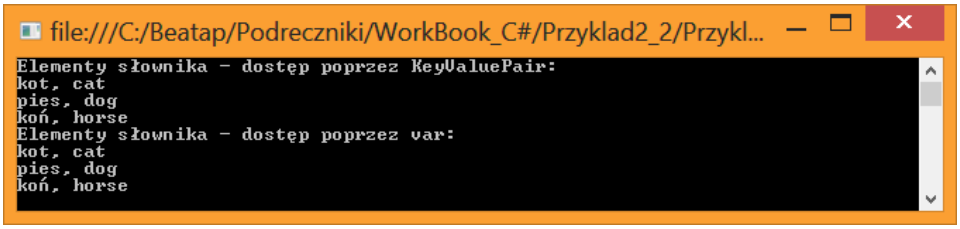
```
Dictionary<int, string> d1;
```

gdzie kluczem słownika jest liczba typu `int`, a wartością ciąg typu `string`.

Pracę z klasą `Dictionary` przedstawia przykład 2.40, a wynik działania widoczny jest na rysunku 2.21.

Przykład 2.40. Praca z klasą Dictionary

```
using System;
using System.Collections;
using System.Collections.Generic;
class Program
{
    static void Main(string[] args)
    {   Dictionary <string, string> d1 =
                    new Dictionary<string, string>();
        //przypisz kluczom wartości:
        d1["kot"] = "cat";
        d1["pies"] = "dog";
        d1["koń"] = "horse";
        Console.WriteLine("Elementy słownika - dostęp poprzez
                           KeyValuePair:");
        //wykorzystaj obiekt KeyValuePair do przeglądania słownika:
        foreach (KeyValuePair<string, string> para in d1)
           {
               Console.WriteLine("{0}, {1}", para.Key, para.Value);
           }
        Console.WriteLine("Elementy słownika - dostęp poprzez
                           var:");
        //wykorzystaj słowo var do przeglądania słownika
        foreach (var para in d1)
          {
              Console.WriteLine("{0}, {1}", para.Key, para.Value);
          }
        Console.ReadKey();
    }
}
```



Rys. 2.21. Wynik działania programu z przykładu 2.40

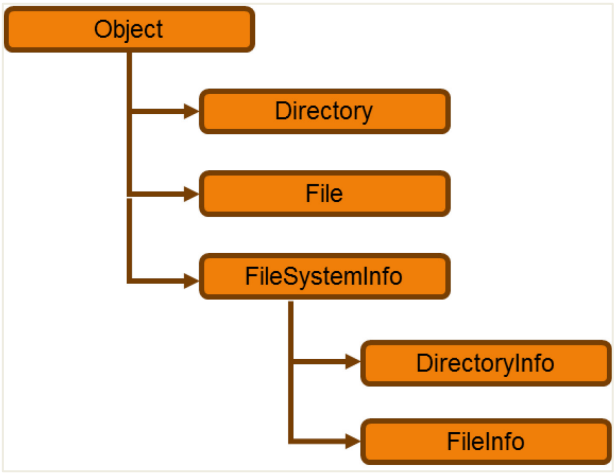
2.3.3. Przestrzeń nazw System.IO

Przestrzeń nazw `System.IO` to część bibliotek klas bazowych poświęcona obsłudze operacji we/wy wykonywanych na plikach i w pamięci. Przestrzeń zawiera zbiór typów do manipulowania fizycznymi katalogami i plikami, typy do obsługi czytania i zapisywania danych do buforów łańcuchowych i różnych obszarów pamięci.

Najważniejsze klasy przestrzeni System.IO to:

- `Directory`, `DirectoryInfo`, `File`, `FileInfo` – klasy do działania na katalogach i plikach,
- `StreamWriter`, `StreamReader` – zapis/odczyt informacji tekstowych z/do pliku (pamięcią bazową jest bufor łańcuchowy),
- `FileStream` – umożliwia bezpośredni dostęp do pliku (np. wyszukiwanie danych) z danymi reprezentowanymi jako strumień bajtów,
- `MemoryStream` – bezpośredni dostęp do danych strumieniowych składowanych w pamięci a nie w fizycznym pliku,
- `BufferedStream` – zapewnia tymczasowe przechowywanie strumienia bajtów, który można później zapisać w pamięci trwałej,
- `BinaryReader`, `BinaryWriter` – zapis/odczyt prostych typów danych (liczby całkowite, wartości logiczne itp.) w postaci wartości binarnych.

Hierarchia klas związanych z klasami `File` i `Directory` przedstawiona jest na rysunku 2.22.



Rys. 2.22. Hierarchia klas DirectoryInfo i FileInfo

Abstrakcyjna klasa bazowa `FileSystemInfo`, po której dziedziczą klasy `DirectoryInfo` i `FileInfo` posiada m.in. następujące właściwości: `Attributes`, `FileAttributes`, `CreationTime`, `Exists`, `Extension`, `FullName`, `LastAccessTime`, `LastWriteTime`, `Name`.

Klasa FileInfo

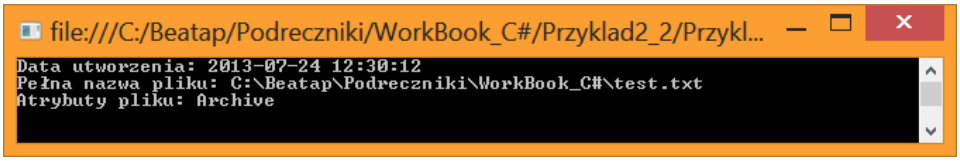
Klasa `FileInfo` dziedziczy po `FileSystemInfo` oraz posiada metody:

- `AppendText()` – tworzy typ `StreamWriter` załączający tekst do pliku,
- `CopyTo()` – tworzy kopię pliku,
- `Create()` – tworzy nowy plik i zwraca typ `FileStream`,
- `CreateText()` – tworzy typ `StreamWriter`, który zapisuje dane do nowego pliku tekstowego,
- `Delete()`, `MoveTo()` – usuwa /zmienia lokalizację pliku,
- `Open()` – otwiera plik z różnymi prawami zapisu i dostępu,
- `OpenRead()` – tworzy typ `FileStream` tylko do odczytu,
- `OpenWrite()` – tworzy typ `FileStream` do odczytu i zapisu,
- `OpenText()` – tworzy typ `StreamReader`.

Przykład 2.41 przedstawia podstawy pracy z klasą `FileInfo`. Wynik działania kodu demonstruje rysunek 2.23.

Przykład 2.41 .Praca z klasą FileInfo i jej metodąCreate

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {
        //utwórz nowy plik na dysku
        FileInfo f = new
           FileInfo(@"C:\Beatap\Podreczniki\WorkBook_C#\test.txt");
        FileStream fs = f.Create();
        //wyswietl najważniejsze info o pliku test.txt
        Console.WriteLine("Data utworzenia: " + f.CreationTime );
        Console.WriteLine("Pełna nazwa pliku: " + f.FullName );
        Console.WriteLine("Atrybuty pliku: " + f.Attributes );
        //zamknij strumień do pliku i usuń plik
        fs.Close();
        f.Delete();

        Console.ReadKey();
    }
}
```



Rys. 2.23. Wynik działania programu z przykładu 2.41

Typy wyliczeniowe z przestrzeni System.IO

Do otwierania istniejących plików i tworzenia nowych plików z większą precyzją niż za pomocą metody `Create()`, można użyć metody `Open()` (Przykład 2.42) z tej samej klasy. Podczas otwarcia należy określić sposób korzystania z pliku za pomocą ustawienia atrybutów, zdefiniowanych w przestrzeni nazw `System.IO` przy pomocy typów wyliczeniowych:

- `System.IO.FileMode` – definiuje sposób otwarcia lub utworzenia pliku,
- `System.IO.FileAccess` – definiuje tryb dostępu do pliku,
- `System.IO.FileShare` – definiuje zasady współużytkowania pliku z innymi programami (procesami).

Typ wyliczeniowy `FileMode` posiada następujące wartości:

- `FileMode.Append` – otwiera plik w celu dopisywania do pliku, jeżeli żądany plik nie istnieje jest tworzony. Atrybut ten może pracować tylko w połączeniu z atrybutem `FileAccess.Write`,
- `FileMode.Create` – otwiera plik do zapisu. W przypadku, gdy plik nie istnieje jest tworzony, jeżeli istnieje jest nadpisywany,
- `FileMode.CreateNew` – otwiera plik do zapisu. W przypadku, gdy plik nie istnieje jest tworzony, jeżeli istnieje wyrzucany jest wyjątek,
- `FileMode.Open` – otwiera plik do odczytu lub zapisu w zależności od atrybutu `FileAccess`. Jeżeli plik nie istnieje wyrzucany jest wyjątek,
- `FileMode.OpenOrCreate` – otwiera plik do odczytu lub zapisu w zależności od atrybutu `FileAccess`. Otwiera istniejący plik, jeżeli plik nie istnieje jest tworzony,
- `FileMode.Truncate` – otwiera plik do zapisu, otwiera istniejący plik kasując jego zawartość.

Typ wyliczeniowy `FileAccess` udostepnia wartości:

- `FileAccess.Read` – dane mogą być czytane z pliku,
- `FileAccess.Write` – dane mogą być zapisywane do pliku,
- `FileAccess.ReadWrite` – dane mogą być czytane z i zapisywane do pliku.

Typ wyliczeniowy `FileShare` definiuje wartości:

- `FileShare.None` – zabrania udostępniania bieżącego pliku do chwili jego zamknięcia,
- `FileShare.Read` – pozwala otworzyć powtórnie plik, ale tylko do odczytu,
- `FileShare.ReadWrite` – pozwala powtórnie otworzyć plik do zapisu lub odczytu,
- `FileShare.Write` – pozwala powtórnie otworzyć plik, ale tylko do zapisu.

Przykład 2.42. Praca z klasą FileInfo i jej metodą Open

```
//otwórz lub utwórz nowy plik z uprawnieniami zapisu i odczytu
//ale bez udostępniania go i zapisz uchwyt do pliku
//w obiekcie FileStream
FileInfo f = new
        FileInfo(@"C:\Beatap\Podreczniki\WorkBook_C#\test2.txt");
```

```
FileStream fs = f.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite,
                       FileShare.None);
//wykonaj operacje na pliku
//zamknij strumień do pliku i usuń plik
fs.Close();
f.Delete();
```

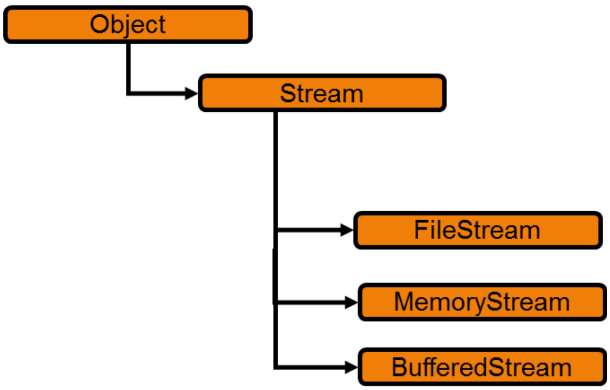
Strumienie

Strumień to warstwa abstrakcyjna, która umożliwia zapis i odczyt danych, nie tylko z pliku, ale z różnych źródeł. W celu skorzystania z pliku, należy skojarzyć z nim odpowiedni strumień.

Na strumieniu możemy wykonywać różne operacje (ich dostępność zależy od typu strumienia i sposobu jego otwarcia), np.:

- czytanie danych (ang. reading) – pobieranie danych ze strumienia i umieszczanie ich w pewnej strukturze danych,
- zapis danych (ang. writting) – wstawienie danych do strumienia z pewnej struktury danych,
- ustawienie bieżącej pozycji w strumieniu (ang. seeking).

Rysunek 2.24 przedstawia klasę abstrakcyjną `Stream` i klasy po niej dziedziczące.



Rys. 2.24. Hierarchia klas strumieniowych

Do najważniejszych składowych klasy `Stream` należą:

- `CanRead`, `CanSeek`, `CanWrite` – sprawdza czy bieżący strumień obsługuje odczyt/wyszukiwanie/zapis,
- `Length` – zwraca wyrażoną w bajtach długość strumienia,

- `Position` – ustala pozycję w bieżącym strumieniu,
- `Read()`, `ReadByte()`,
- `Write()`, `WriteByte()`,
- `Seek()`,
- `SetLenght()`,
- `Close()`, `Flush()`.

Przykład 2.43 przedstawia podstawy pracy z klasą `FileStream` (która reprezentuje strumień skojarzony z plikiem), a przykład 2.44 pracę z klasą `MemoryStream` (reprezentuje strumień skojarzony z pamięcią). Zwróćmy uwagę na analogie przy pracy z obiema klasami.

Przykład 2.43. Praca z klasą FileStream

```
//utwórz nowy plik w bieżącym katalogu
FileStream fs = new FileStream("dane.dat", FileMode.OpenOrCreate,
                               FileAccess.ReadWrite);
//zapisz bajty w pliku
for (int i = 0; i < 256; i++) fs.WriteByte((byte)i);
//zresetuj wskaźnik położenia
fs.Position = 0;
//odczytaj bajty z pliku
for (int i = 0; i < 256; i++) Console.Write( fs.ReadByte() + " ");
//zamknij strumień
fs.Close();
```

Przykład 2.44. Praca z klasą MemoryStream

```
//zapis następuje nie do fizycznego pliku, ale do pamięci
//utwórz strumień pamięciowy o ustalonej pojemności
MemoryStream ms = new MemoryStream();
ms.Capacity = 256;
//zapisz bajty do strumienia
for (int i = 0; i < 256; i++) ms.WriteByte((byte)i ) ;
//zresetuj wskaźnik położenia
ms.Position = 0;
//odczytaj bajty ze strumienia
for (int i = 0; i < 256; i++) Console.Write( ms.ReadByte() + " ");
//zamknij strumień
ms.Close();
```

Najwazniejsze składowe klasy `MemoryStream` to:

- `Capacity` – odczytuje lub ustawia liczbę bajtów dla tego strumienia,
- `GetBuffer()` – zwraca tablicę nieoznaczonych bajtów, z których ten strumień został utworzony,

- `ToArray()` – zapisuje zawartość całego strumienia w tablicy bajtów niezależnie od właściwości `Position`,
- `WriteTo()` – zapisuje całą zawartość tego typu `MemoryStream` w innym typie wyprowadzonym ze `Stream`, np. w pliku (Przykład 2.45).

Przykład 2.45. Praca z klasą MemoryStream i FileStream

```
//utwórz strumień pamięciowy o ustalonej pojemności
MemoryStream ms = new MemoryStream();
ms.Capacity = 256;
//zapisz bajty do strumienia
for (int i = 0; i < 256; i++) ms.WriteByte((byte)i);
//przekaż dane z pamięci do pliku
FileStream fs = new FileStream("dane.dat", FileMode.Create,
                               FileAccess.ReadWrite);
ms.WriteTo(fs);
//przekaż dane z pamięci do tablicy bajtów
byte[] tabzpamieci = ms.ToArray();
//zamknij strumienie:
fs.Close();
ms.Close();
```

Przy pracy ze strumieniami można użyć bufora (obiektu klasy `BufferedStream`) jako tymczasowego miejsca zapisywania lub wczytywania informacji. Wykorzystanie takiego bufora optymalizuje proces zapisu/odczytu poprzez redukcję operacji na fizycznym pliku (operacje są wykonywane na buforze w pamięci) (Przykład 2.46).

Przykład 2.46. Praca z klasą BufferedStream

```
FileStream fs = new FileStream("dane.dat", FileMode.Create,
                               FileAccess.ReadWrite);
//utwórz bufor powiązany z istniejącym fs (typu FileStream)
BufferedStream bs = new BufferedStream(fs);
//dodaj do bufora kilka bajtów
byte[] tab = { 127, 0x77, 0x4, 0x0, 0x16 };
bs.Write(tab, 0, tab.Length);
//przekaż zmiany do pliku
bs.Close(); //automatycznie wyczyść bufor
```

Operacje plikowe mogą generować liczne wyjątki (typu `IOException` np.: brak odpowiednich uprawnień do pliku, brak żądanego pliku, brak miejsca na dysku, niedostępny dysk sieciowy), dlatego zawsze powinny być umieszczone w bloku `try` (Przykład 2.47).

Przykład 2.47. Operacje plikowe w bloku try

```
FileStream zrodlo = null, cel = null;
try
{
   //operacje na pliku
}
catch (IOException ex)
{  Console.WriteLine("Problemy z plikiem.\n{0}", ex.Message);
}
finally
{
   if (zrodlo != null) zrodlo.Close();
   if (cel != null) cel.Close();
}
```

Kolejny program z przykładu 2.48 realizuje operację szyfrowania danych z pliku:

- pobiera nazwę pliku do szyfrowania,
- pobiera nazwę pliku zaszyfrowanego,
- pobiera wartość klucza (maski) szyfrującego,
- wczytuje po jednym bajcie z pliku źródłowego, następnie wykonuje operację bitową `xor` na kluczu i wartości wczytanej, a wynik wyrażenia zapisuje do pliku zaszyfrowanego,
- czynność powtarza do osiągnięcia końca pliku do szyfrowania.

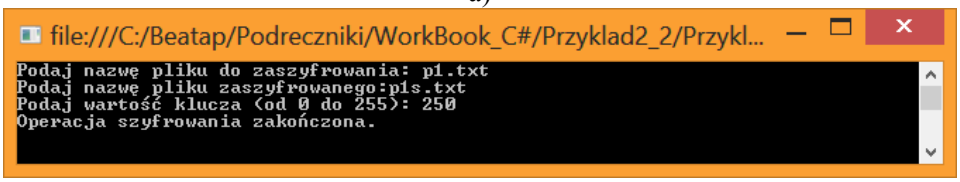
Podając jako plik do szyfrowania plik wcześniej zaszyfrowany i klucz, który został użyty do szyfrowania tego pliku nastąpi jego rozszyfrowanie (Rys.2.25).

Przykład 2.48. Szyfrowanie danych z pliku

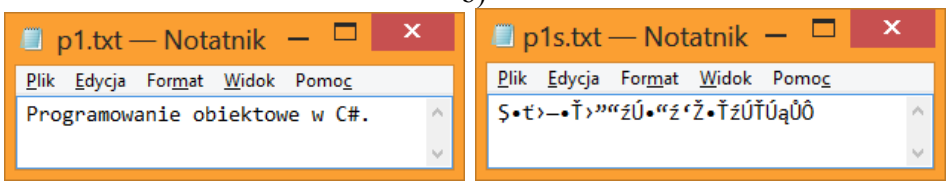
```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {
        Console.Write("Podaj nazwę pliku do zaszyfrowania: ");
        string nazwaZrodla = Console.ReadLine();
        Console.Write("Podaj nazwę pliku zaszyfrowanego:");
        string nazwaCelu = Console.ReadLine();
        Console.Write("Podaj wartość klucza (od 0 do 255): ");
        byte klucz = Convert.ToByte(Console.ReadLine());
        FileStream zrodlo = null, cel = null;
```

```
        try
        {
            zrodlo = new FileStream(nazwaZrodla, FileMode.Open);
            cel = new FileStream(nazwaCelu, FileMode.Create);
            int i = zrodlo.ReadByte();
            while (i != -1)
            {
                cel.WriteByte((byte)(i ^ klucz));
                i = zrodlo.ReadByte();
            }
        }
        finally
        {
            if (zrodlo != null) zrodlo.Close();
            if (cel != null) cel.Close();
        }
        Console.WriteLine("Operacja szyfrowania zakończona.");
        Console.ReadKey();
    }
}
```

a)



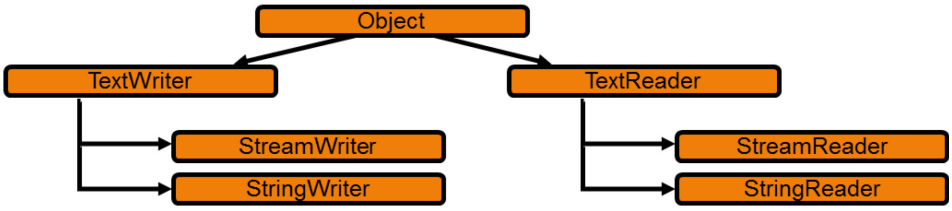
b)



Rys. 2.25. a) Wynik działania programu z przykładu 2.48, b) plik p1.txt poddany szyfrowaniu, i plik p1s.txt zaszyfrowany

Klasy StreamWriter i StreamReader

Do pracy z danymi znakowymi (w formacie Unicode) wykorzystuje się klasy `StreamWriter` i `StreamReader`, których miejsce w hierarchii dziedziczenia klas przedstawia rysunek 2.26. Klasy te nie zapewniają dostępu do swojej zawartości (np. w celu wyszukiwania danych).



Rys. 2.26. Klasy StreamReader i StreamWriter w hierarchii dziedziczenia

Klasa abstrakcyjna `TextWriter`, po której dziedziczą klasy `StreamWriter` i `StringWriter`, posiada następujące składowe:

- `Close()` – zamyka proces zapisujący i zwalnia wszystkie skojarzone z nim zasoby, jednocześnie automatycznie opróżniany jest bufor;
- `Flush()` – czyści wszystkie bufory związane z bieżącym procesem zapisywania i powoduje zapisanie wszystkich zbuforowanych danych na odpowiednim urządzeniu, ale proces zapisujący nie jest zamykany;
- `NewLine` – używana do tworzenia znaku nowego wiersza dla wyprowadzonej klasy zapisującej. Domyślnym zakończeniem wiersza jest znak powrotu karetki, po którym następuje wysuw nowego wiersza ("\r\n");
- `Write()` – zapisuje wiersz w strumieniu tekstowym bez znaku nowego wiersza;
- `WriteLine()` - zapisuje wiersz w strumieniu tekstowym ze znakiem nowego wiersza.

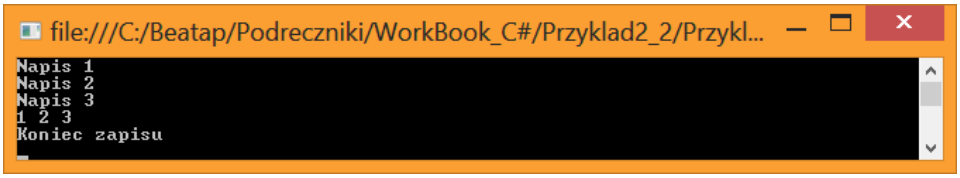
Klasa abstrakcyjna `TextReader`, po której dziedziczą klasy `StreamReader` i `StringReader` udostępnia składowe:

- `Peek()` – zwraca następny dostępny znak, nie zmieniając pozycji czytnika;
- `Read()` – odczytuje dane ze strumienia wejściowego;
- `ReadBlock()` – odczytuje maksymalną liczbę znaków z bieżącego strumienia i zapisuje dane w buforze, zaczynając od miejsca wskaźnika;
- `ReadLine()` – odczytuje z bieżącego strumienia wiersz znaków i zwraca dane jako łańcuch znaków (pusty łańcuch oznacza koniec pliku);
- `ReadToEnd()` – wczytuje wszystkie znaki między bieżącym położeniem a końcem typu `textReader` i zwraca je jako jeden łańcuch znaków.

Przykład 2.49 przedstawia podstawowy schemat pracy przy zapisie i odczycie danych z pliku tekstowego. Wynik działania programu pokazuje rysunek 2.27.

Przykład 2.49. Praca z klasą StreamWriter i StreamReader

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {
         //zapis do pliku tekstowego:
        //*************************
        //utworz plik w katalogu aplikacji
        FileInfo f = new FileInfo("Test.txt");
        //uzyskaj typ StreamWriter i zapisz do niego tekst
        StreamWriter sw = f.CreateText();
        sw.WriteLine("Napis 1");
        sw.WriteLine("Napis 2");
        sw.WriteLine("Napis 3");
        for (int i = 1; i <= 3; i++)
            sw.Write(i + " ");
        sw.Write(sw.NewLine);
        sw.WriteLine("Koniec zapisu");
        //zamkniecie zapisu automatycznie opróżnia bufor
        sw.Close();

        //odczyt z pliku tekstowego:
        //*************************
        StreamReader sr = File.OpenText("Test.txt");
        string input = null;
        while ((input = sr.ReadLine()) != null)
            Console.WriteLine(input);
        sr.Close();

        Console.ReadKey();
    }
}
```



Rys. 2.27. Wynik działania programu z przykładu 2.49

Klasy StringWriter i StringReader

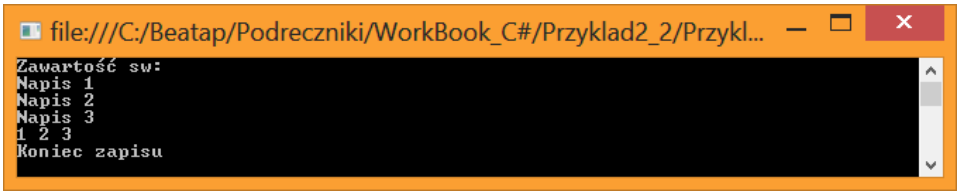
Dzięki klasom `StringWriter` i `StringReader` informacje tekstowe można traktować jako strumień znaków pamięciowych np. gdy do danego bufora chcemy dołączyć informacje znakowe. Aby uzyskać dostęp do tego bufora z instancji typu `StringWriter` można wywołać :

- przesłoniętą metodę `ToString()`, aby uzyskać typ `System.String` (Przykład 2.50 i rysunek.2.28),
- metodę `GetStringBuilder()`, która zwraca instancję klasy `System.Text.StringBuilder` (Przykład 2.51 i rysunek.2.29).

Klasy `StringWriter/StreamWriter` i `StringReader/StreamReader` dziedziczą funkcje tej samej klasy bazowej (Rys. 2.26).

Przykład 2.50. Praca z klasą StringWriter i metodą ToString

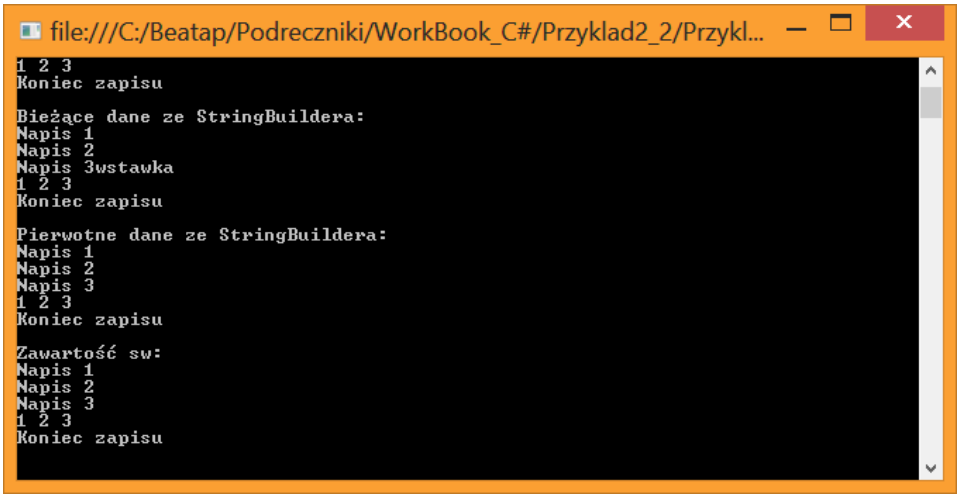
```
//utwórz typ StringWriter i coś do niego zapisz
StringWriter sw = new StringWriter();
sw.WriteLine("Napis 1");
sw.WriteLine("Napis 2");
sw.WriteLine("Napis 3");
for (int i = 1; i <= 3; i++) sw.Write(i + " ");
sw.Write(sw.NewLine);
sw.WriteLine("Koniec zapisu");
//zamkniecie zapisu automatycznie opróżnia bufor
sw.Close();
Console.WriteLine("Zawartość sw:\n" + sw.ToString() );
```



Rys. 2.28. Wynik działania programu z przykładu 2.50

Przykład 2.51. Praca z klasą StringWriter i StringBuilder

```
//sw - jak w poprzednim przykładzie
//utwórz wewnętrzny typ StringBuilder (konieczne using System.Text)
StringBuilder str = sw.GetStringBuilder();
string wszystkiedane = str.ToString();
Console.WriteLine("Wszystkie dane ze StringBuildera:\n" +
                  wszystkiedane);

//wstaw element do bufora w pozycji 25
str.Insert(25, "wstawka");
wszystkiedane = str.ToString();
Console.WriteLine("Bieżące dane ze StringBuildera:\n" +
wszystkiedane);

//usuń wstawiony łańcuch
str.Remove(25, "wstawka".Length);
wszystkiedane = str.ToString();
Console.WriteLine("Pierwotne dane ze StringBuildera:\n" +
                  wszystkiedane);
Console.WriteLine("Zawartość sw:\n" + sw.ToString());
sw.Close();
```



Rys. 2.29. Wynik działania programu z przykładu 2.51

Przykład kolejny pokazuje w jaki sposób można odczytać dane za pomocą obiektu klasy `StringReader`. Porównaj instrukcje odczytu danych z przykładów 2.49 i 2.52.

Przykład 2.52. Praca z klasą StringReader

```
//odczytaj dane za pomocą StringReadera
StringReader sr = new StringReader(sw.ToString());
string input = null;
while ((input = sr.ReadLine()) != null)
      Console.WriteLine(input);
sr.Close();
```

Klasy BinaryWriter i BinaryReader

Klasy `BinaryWriter` i `BinaryReader` pochodzą bezpośrednio z klasy `System.Object` i umożliwiają odpowiednio zapis nieciągłych typów danych do strumienia i odczyt takich danych ze strumienia.

Najważniejsze składowe klasy `BinaryWriter` to:

- `BaseStream` – reprezentuje strumień powiązany z procesem odczytu danych binarnych;
- `Close()` – zamyka strumień binarny;
- `Flush()` – opróżnia strumień binarny;
- `Seek()` – ustawia położenie w bieżącym strumieniu;
- `Write()` – zapisuje wartość do bieżącego strumienia.

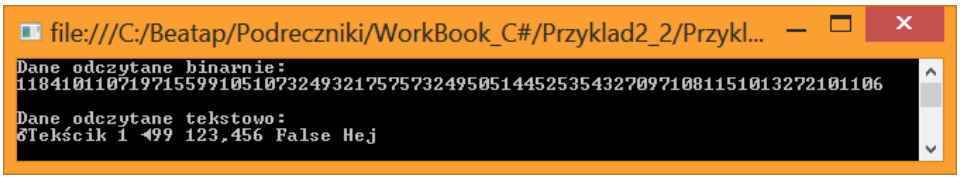
Najważniejsze składowe klasy `BinaryReader` to:

- `BaseStream` – umożliwia dostęp do wykorzystywanego strumienia;
- `Close()` – zamyka strumień binarny;
- `PeekChar()` – zwraca następny dostępny znak bez przesuwania wskaźnika położenia w strumieniu;
- `Read()` – odczytuje określony zbiór znaków lub bajtów i zapisuje je w tablicy;
- `ReadXXX()` – w klasie `BinaryReader` zdefiniowano wiele metod `ReadXXX()`, które przechwytują ze strumienia następny typ (np. `ReadBoolean()`, `ReadByte()`, `ReadChar()`, `ReadInt32()`, `ReadDouble()`).

Przykład 2.53 pokazuje podstawy pracy z klasami `BinaryReader` i `BinaryWriter`. Wynik działania przykładu demonstruje rysunek 2.30.

Przykład 2.53. Praca z klasą BinaryReader i BinaryWriter

```
FileStream fs = new FileStream("bin.dat", FileMode.OpenOrCreate,
                    FileAccess.ReadWrite);
//zapisz dane binarne
BinaryWriter bw = new BinaryWriter(fs);
bw.Write("Tekścik 1 ");
int n = 99; float a = 123.456F; Boolean b = false;
bw.Write(n + " " + a + " " + b + " ");
char[] z = { 'H', 'e', 'j' };
bw.Write(z);
//wyzeruj wewnętrzny wskaźnik położenia:
bw.BaseStream.Position = 0;

//odczytaj dane binarne w postaci bajtów:
Console.WriteLine("Dane odczytane binarnie:");
BinaryReader br = new BinaryReader(fs);
while (br.PeekChar() != -1)
{
   Console.Write( br.ReadByte() );
}
//odczytaj dane binarne w postaci znaków:
br.BaseStream.Position=0;
Console.WriteLine("\n\nDane odczytane tekstowo:");
while ( br.PeekChar() != -1)
{
   Console.Write(br.ReadChar());
}
bw.Close();
br.Close();
fs.Close();
```



Rys. 2.30. Wynik działania programu z przykładu 2.53

Podsumowując, klasy reprezentujące strumienie w .NET możemy podzielić na trzy grupy przedstawione w tabeli 2.1.

Tabela 2.1. Grupy strumieni w .NET

Grupa	Opis	Przykładowe klasy
1	strumienie bezpośrednio podłączone do źródła	○ `System.IO.FileStream`, ○ `System.IO.MemoryStream`, ○ `System.Net.Sockets.NetworkStream`;
2	strumienie pośredniczące w komunikacji ze strumieniami grupy 1	○ `System.IO.BufferedStream`, ○ `System.Security.Cryptography.CryptoStream`
3	klasy używane do odczytu lub zapisu z/do strumienia	○ `System.IO.BinaryReader`, `System.IO.BinaryWriter` - odczyt lub zapis danych binarnych, ○ `System.IO.StreamReader`, `System.IO.StreamWriter` - odczyt lub zapis danych tekstowych.

Poszczególne grupy udostępniają różny stopień abstrakcji przesyłania danych do strumienia:

- grupa 1 – udostępnia najprostszy interfejs, w którym do strumienia można zapisywać oraz z niego pobierać tylko pojedyncze bajty;
- grupa 2 – pozwala na dodanie szyfrowania lub buforowania strumienia;
- grupa 3 – przy pomocy obiektów klas tej grupy można wymieniać ze strumieniem informacje w formie napisów, liczb i innych obiektów.

2.3.4. Serializacja obiektów

Pojęcie serializacji obiektów związane jest z utrwaleniem danych o stanie obiektu w określonej lokalizacji (np. w strumieniu pamięci, fizycznym pliku). Deserializacja to z kolei proces wczytywania utrwalonych danych z tej lokalizacji i i tworzenia nowego obiektu na bazie zachowanych wartości stanu.

Serializacja (ang. serialization) zatem to proces przekształcania stanu obiektu do liniowej sekwencji danych. Sekwencja ta zawiera wszystkie informacje niezbędne do zrekonstruowania (deserializacji) stanu obiektu i późniejszego wykorzystania tych informacji.

Proces serializacji w .NET realizowany jest następująco:

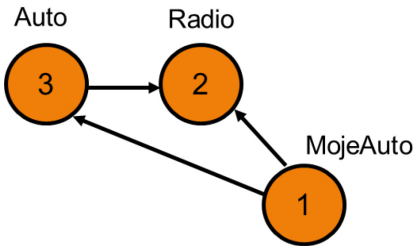
- kiedy dany obiekt zostanie zapisany do strumienia, wszystkie referencje do obiektów wymaganych przez główny obiekt są również automatycznie serializowane (kiedy np. serializowana jest klasa pochodna, każdy serializowany obiekt w łańcuchu dziedziczenia może zapisać w strumieniu niestandardowe dane o własnym stanie). Jeśli dany typ zawiera serializowane zmienne składowe, również są one automatycznie uwzgledniane w procesie serializacji;
- po zapisaniu zbioru obiektów do strumienia (np. graf obiektów zapisano w pamięci), strumień ten można przesłać dalej (np. na zdalny komputer, zapisać w pliku itp.).

W procesie serializacji tworzony jest graf obiektów. Graf obiektów (ang. object graph) to łańcuch powiązanych ze sobą obiektów serializowanych do strumienia. To prosty sposób dokumentowania wzajemnych związków między obiektami, ale nie w rozumieniu klasycznych wzajemności obiektowych (czyli relacji ‘jest’:is-a, ‘zawiera’:has-a). Każdemu obiektowi przyporządkowuje się unikatową wartość liczbową, a po niej graf wszystkich powiązanych z tym obiektem elementów (numery przypisane elementom grafu obiektów są przypadkowe i nie mają żadnego znaczenia na zewnątrz).

Na przykład, dla zbioru klas reprezentujących modele samochodów, klasa bazowa `Auto` jest związana relacją ‘zawiera’ z klasą `Radio`. Klasa o nazwie `MojeAuto` dziedziczy po `Auto`. Graf obiektów modelujący te zależności ma postać jak na rysunku 2.31. Przedstawione zależności można wyrazić wzorem:

```
[Auto 3, ref 2], [Radio 2], [MojeAuto 1, ref 3, ref 2].
```

Serializując instancje typu `MojeAuto`, dzięki grafowi obiektów typy `Radio` i `Auto` zostaną w tym procesie uwzględnione a graf jest tworzony automatycznie bez udziału programisty.



Rys. 2.31. Przykładowy graf serializacji obiektów

Aby przygotować obiekt do serializacji w .NET należy wyposażyć każdą klasę w atrybut `[Serializable]`. Jeśli klasa zawiera dane składowe, które nie powinny być serializowane to pola takie należy oznaczyć atrybutem `[NonSerialized]`, np.:

```
[Serializable]
public class Radio
{
    [NonSerialized]
    private int IdObiektu = 10;
    //dane serializowane:
    public Radio() {  }
    public void On(bool wlaczone) {  }
}
```

Aby serializować obiekty należy wskazać odpowiedni format serializacji. Przestrzeń nazw `System.Runtime.Serialization.Formatters` zawiera dwie dodatkowe przestrzenie nazw (`Binary` i `SOAP`), w których zdefiniowano tzw. formatery:

- format binarny z przestrzeni:
  `System.Runtime.Serialization.Formatters.Binary;`
- format XML z przestrzeni:
  `System.Xml.Serialization`.

Formatery implementują interfejs `IFormatter` z metodami `Serialize()` i `Deserialize()` oraz interfejsu `IRemotingFormatter`, dzięki czemu dostarczają wspólny zbiór funkcji.

Najważniejsze składowe klasy `BinaryFormatter` to:

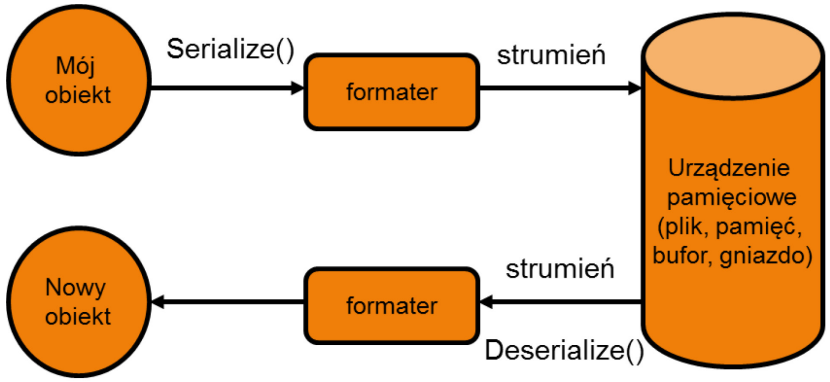
- `Deserialize()` – deserializuje strumień bajtów do grafu obiektów,

- `Serialize()` – serializuje obiekt lub graf powiązanych ze sobą obiektów do strumienia,

Do pracy z formaterem binarnym należy skorzystać z przestrzeni nazw:

- `using System.IO;`
- `using System.Runtime.Serialization.Formatters.Binary;`

Proces serializacji i deserializacji można zobrazować schematem jak na rysunku 2.32.



Rys. 2.32. Schemat serializacji i deserializacji

Przykład 2.54 przedstawia definicje klas `Auto`, `Radio` i `MojeAuto` przeznaczonych do serializacji oraz wykorzystanie serializacji z formaterem binarnym.

Przykład 2.54. Serializacja obiektów z formaterem binarnym

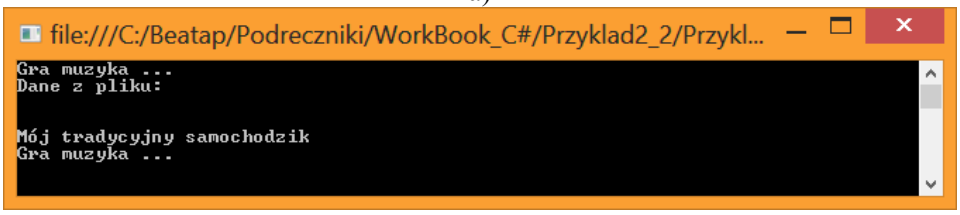
```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]  //klasa Radio przeznaczona do serializacji
public class Radio
{
    [NonSerialized]
    private int obiektId = 9;
    //inne serializowane dane stanu:
    public Radio() { }
    public void On(bool wlaczone)
```

```
    {
        if (wlaczone == true) Console.WriteLine("Gra muzyka ... ");
        else Console.WriteLine("Cisza ... ");
    }
}

[Serializable] //klasa Auto przeznaczona do serializacji
public class Auto
{
    protected string nazwa;
    protected int maxPredkosc;
    public bool radioWlaczone;
    protected Radio radio = new Radio();
    public Auto(string nn, int ms)
    { nazwa = nn; maxPredkosc = ms; }
    public Auto() { }
    public String Nazwa
    { get;set; }
    public int MaxPredkosc
    { get; set; }
    public void WlaczRadio(bool wlaczone)
    { radio.On(wlaczone);
      radioWlaczone = wlaczone;
    }
}

[Serializable] //klasa MojeAuto przeznaczona do serializacji
public class MojeAuto : Auto
{
    public bool czySportowe;
    public MojeAuto() { }
    public MojeAuto(string nn, int ms, bool sport)
        : base(nn, ms)
    {
        czySportowe = sport;
    }
    public void Info()
    {
        if (czySportowe) Console.WriteLine("Mój sportowy
                                            samochodzik");
        else Console.WriteLine("Mój tradycyjny samochodzik");
        if (radioWlaczone) Console.WriteLine("Gra muzyka ...");
        else Console.WriteLine("Cisza ...");
    }
}
```

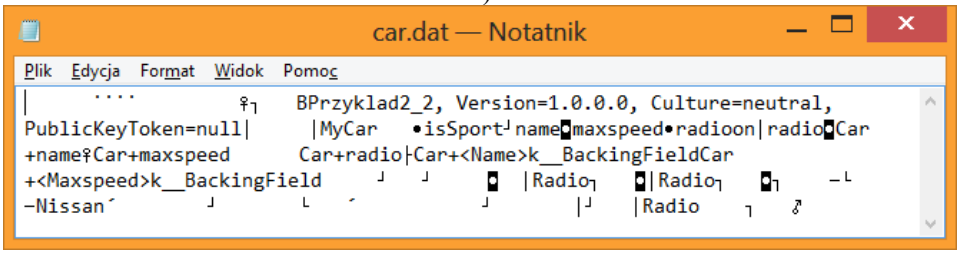
```
class Program
{
    static void Main(string[] args)
    {
        MojeAuto Auto = new MojeAuto("Nissan", 180, false);
        Auto.WlaczRadio(true);
        //utwórz strumień do pliku aby zapisać w nim stan obiektu
        FileStream fs = File.Create("Auto.dat");
        //przenieś graf obiektów do strumienia
        BinaryFormatter bf = new BinaryFormatter();
        bf.Serialize(fs, Auto);
        fs.Close();

        //wczytaj Auto ze strumienia binarnego
        fs = File.OpenRead("Auto.dat");
        MojeAuto AutozPliku = (MojeAuto)bf.Deserialize(fs);
        Console.WriteLine("Dane z pliku:\n" + AutozPliku.Nazwa +
                          "\n");
        AutozPliku.Info();
        fs.Close();
        Console.ReadKey();
    }
}
```

Wynik działania programu z przykładu 2.54 przedstawia rysunek 2.33.

a)



b)



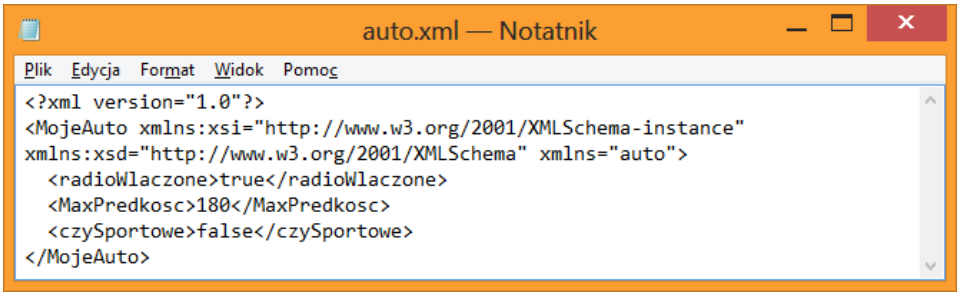
Rys. 2.33. a) Wynik działania programu z przykładu 2.54 i b) wynikowy plik binarny

Przykład 2.55 przedstawia wykorzystanie formatera XML, a rysunek 2.34 wynikowy plik XML. Należy pamiętać o dodaniu przestrzeni nazw:

```
using System.Xml.Serialization;
```

Przykład 2.55. Serializacja obiektów z formaterem XML

```
//definicje klas - jak w przykładzie 2.54
//należy użyć: using System.Xml.Serialization;
MojeAuto Auto = new MojeAuto("Nissan", 180, false);
Auto.WlaczRadio(true);
//utwórz strumień do pliku aby zapisać w nim stan obiektu
FileStream fs = File.Create("auto.xml");
//przenieś graf obiektów do pliku XML
XmlSerializer xs = new XmlSerializer(typeof(MojeAuto), "auto");
xs.Serialize(fs, Auto);
fs.Close();
//wczytaj auto z pliku XML
fs = File.OpenRead("auto.xml");
MojeAuto AutozPliku = (MojeAuto)xs.Deserialize(fs);
Console.WriteLine("Dane z pliku XML:\n" + AutozPliku.Nazwa);
AutozPliku.Info();
fs.Close();
```



Rys. 2.34. Wynikowy plik *auto.xml* z przykładu 2.56

2.4. Zadania do samodzielnego rozwiązania

Zadanie 2.1.

Na podstawie przykładu 2.9 napisać klasę, która będzie reprezentowała obiekt wektora w tablicy **x** *[x1, x2, x3]* i umożliwiała wykonywanie operacji na takich wektorach. Klasa powinna posiadać prywatne pole wskazujące liczbę współrzędnych wektora (*n=3*), prywatne pole będące jednowymiarową tablicą *x* (o trzech elementach) oraz publiczny interfejs składający się z co najmniej

dwóch konstruktorów (jeden konstruktor bezparametrowy ustawia wektor *[0,0,0]*, drugi konstruktor z trzema parametrami ustawia wektor na dowolne wartości *[x1, x2, x3]*). Publiczny interfejs ma zawierać ponadto:

a) metodę `Wczytaj`, która wczytuje współrzędne wektora **x**=*[x1,x2,x3]*;
b) metodę `Wyswietl`, która wyświetla współrzędne wektora **x**=*[x1,x2,x3]*;
c) metodę `DlugoscWektora`, która oblicza długość wektora **x**=*[x1,x2,x3];*
d) metodę `SumaWektorow`, która dla dwóch wektorów **x** i **y** wyznacza współrzędne wektora ***z*** =**x**+**y**;

Napisać aplikację, która korzystając z obiektów klasy `Wektor`, wczytuje współrzędne dwóch wektorów **a**=[*a1,a2,a3]* i **b**=*[b1,b2,b3],* wyznacza wektory **c**=**a**+**b**, **d**=**b**+**c**, **e**=**c**+**d** oraz drukuje długości wektorów **a**, **b**, **c**, **d**, **e** oraz współrzędne wektorów **c**, **d** i **e.**

Wskazówka 1

Parametrem metody `SumaWektorow()` ma być jeden obiekt klasy `Wektor`, drugi obiekt jest wektorem, na rzecz którego wołamy metodę – dostęp do niego mamy dzięki autoreferencji `this`. Wynikiem działania metody ma być nowy obiekt `Wektor`, który będzie reprezentował sumę wektorów.

Wskazówka2

Przykładowe pola i konstruktory mogą być zdefiniowane następująco:

```
class Wektor
{
    readonly int n = 3;
    double[] x;
    public Wektor()
    {
        x = new double[n];
        x.Initialize(); //inicjalizacja tablicy zerami
    }
    public Wektor(double[] t)
    {
        n = t.Length;
        x = new double[n];
        x = (double[])t.Clone(); //skopiowanie tablicy t do x
    }

    //pozostałe metody klasy
    //...
}
```

Wskazówka 3
Wszystkie metody interfejsu (`Wyswietl`(), `Czytaj()` itp.) powinny realizować pracę z tablicą **x** w pętli `for`. Aby skorzystać z *i*-tej współrzędnej np. obiektu `w` typu `Wektor` należy zastosować odwołanie postaci: `w.x[i]`.

Zadanie 2.2.

1) Zdefiniować klasę `WektorN`, która będzie modyfikacją klasy `Wektor` z zadania 2.1. Klasa ma realizować te same zadania co klasa `Wektor`, ale dla dowolnego wymiaru wektora (*n* – ma być parametrem konstruktora). Napisać program testujący metody tej klasy.
2) Do klasy `WektorN` dodać dodatkowe pole typu `string` reprezentujące nazwę wektora. Uwzględnić je w konstruktorze oraz w metodach `Wczytaj()` i `Wyswietl()` (przy wydrukach ma się pojawiać dodatkowo nazwa wczytywanego/wyświetlanego wektora).

Zadanie 2.3.

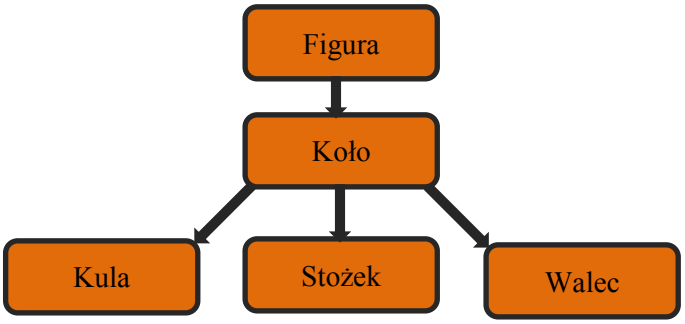
Na podstawie klasy `WektorN` zrealizować klasę `Macierz` do reprezentacji macierzy o wymiarze *n x n* w postaci tablicy dwuwymiarowej (prywatne pole klasy). Klasa ma zawierać odpowiednie konstruktory oraz:

a) metodę, która dla obiektu klasy `Macierz` wczytuje elementy tablicy *n x n*, założyć, że *n*<=10 (w przypadku nieprawidłowej wartości przyjąć *n*=1),
b) metodę, która drukuje elementy tablicy *n x n*;
c) metodę, która mając dane dwa obiekty **X** i **Y** klasy `Macierz` oblicza elementy macierzy **Z** = **X** + **Y**;
d) metodę, która mając dany obiekt **X** klasy `Macierz` wyznacza obiekt **Z**=2**X**;
e) metodę, która mając dane dwa obiekty **X** i **Y** klasy `Macierz` wyznacza obiekt **Z**=**X***__Y__;
f) metodę, która mając dany obiekt **X** wyznacza sumę elementów macierzy tego obiektu;

Napisać program, który wczytuje wartość *n* oraz *n x n* elementów dla obiektów **A** i **B** klasy `Macierz`, wyznacza obiekty **C**=**A**+**B**, **D**=2(**A**+**B**) oraz drukuje macierze i sumy elementów macierzy dla obiektów **C**, **D**.

Zadanie 2.4.

Zrealizować klasę bazową `Figura` oraz klasy pochodne jak pokazano na rysunku 2.35.



Rys.2.35. Schemat dziedziczenia klas

1) Klasa `Figura` ma zawierać:
   a) pole: `nazwa` (domyślnie – figura geometryczna),
   b) odpowiednie konstruktory,
   c) metodę `Wyswietl()` – wyświetlającą komunikat: *„Figura geometryczna”*.
2) Podklasa `Kolo` ma zawierać:
   a) dodatkowe pole definiujące daną figurę (promień koła),
   b) odpowiednie konstruktory (skorzystać z konstruktorów klasy bazowej),
   c) metodę `Pole(),`
   d) predefiniowaną metodę `Wyswietl()`, która wyświetla komunikat postaci (skorzystać z metody nadklasy): *”Figura geometryczna – **nazwa**, pole figury: ...”*.
3) Kolejne podklasy mają zawierać:
   a) dodatkowe pola definiujące daną figurę (wysokość stożka/walca),
   b) konstruktory,
   c) dodatkową metodę `Objetosc(),`
   d) predefiniowaną metodę `Pole()`obliczającą pole powierzchni bocznej figury,
   e) predefiniowaną metodę `Wyswietl()`, która wyświetla komunikat postaci (skorzystać z metody nadklasy): *”Figura geometryczna – **nazwa**, pole figury: ..., objętość: ...”*.
4) Napisać program testowy definiujący po jednym obiekcie każdej klasy i wyświetlający komunikaty na temat każdego obiektu.
5) Utworzyć testową tablicę obiektów `Figura` (np.jeden obiekt klasy `Figura`, dwa obiekty klasy `Kolo`, dwa `Walce`, dwa `Stożki` – rozmieszczone

w dowolnej kolejności w tablicy) oraz za pomocą pętli `foreach` wyświetlić informacje na temat każdego obiektu z tablicy.

Zadanie 2.5.

Zmodyfikować klasy z zadania 2.4 tak aby korzystały z właściwości.

Zadanie 2.6.

Zdefiniować interfejs `I2D` z metodą `Pole()` i interfejs `I3D` z metodą `Objetosc()`. Zrealizować zadanie 2.4 z zastosowaniem tych interfejsów.

3. Wprowadzenie do programowania aplikacji z graficznym interfejsem użytkownika

3.1. Wstęp

Na platformie .NET dostępne są trzy przestrzenie nazw z zestawem typów do tworzenia graficznego interfejsu użytkownika (ang. Graphical User Interface – GUI):

- `Windows.Forms` – do tworzenia tradycyjnych aplikacji pulpitowych,
- `Web Forms` – do tworzenia aplikacji webowych ASP.NET,
- `Mobile Forms` – do tworzenia aplikacji na urządzenia mobilne.

W tych przestrzeniach zdefiniowano wiele podobnie nazywających się klas (np.. `Button`, `CheckBox` itp.) z wieloma podobnymi składowymi (np. `Text`, `BackColor`), które jednak nie mają wspólnej implementacji i nie można ich traktować jednakowo.

Tematem niniejszego rozdziału będą podstawy niezbędne do programowania aplikacji z graficznym interfejsem użytkownika z wykorzystaniem przestrzeni nazw `Windows.Forms`.

3.2. Przestrzeń nazw Windows.Forms

Przestrzeń nazw `Windows.Forms` jest dołączana do każdego projektu aplikacji z GUI.

Zawiera różne kategorie m.in.:

- komponenty, czyli niewidoczne elementy programu, „klocki" działające w tle, które mogą mieć duży wpływ na działanie aplikacji, lecz nie są częścią jej interfejsu graficznego;
- kontrolki, czyli komponenty wizualne do tworzenia interfejsu programu (GUI);
- elementy pozycjonowania, które umożliwiają zarządzanie elementami interfejsu (np. komponent umożliwiający tworzenie zakładek, komponent `Panel` do grupowania innych kontrolek);
- menu i paski narzędziowe;
- okna dialogowe.

Do najważniejszych klas przestrzeni `Windows.Forms` zaliczamy:

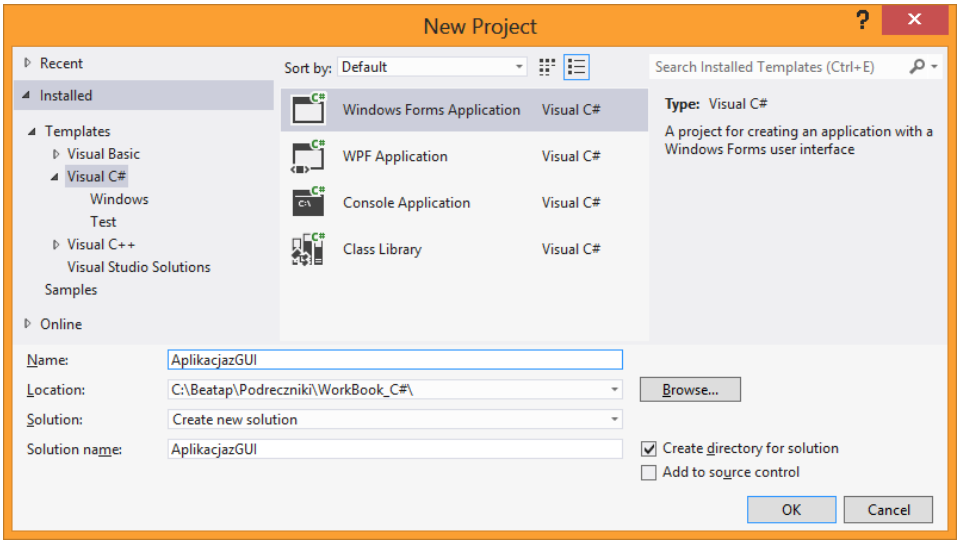
- `Application`,
- klasy kontrolek GUI: `Button`, `CheckBox`, `ComboBox`, `DataGrid`, `GroupBox`, `ListBox`, `PictureBox`, itp.,
- `Form` – główna klasa aplikacji GUI,
- okna dialogowe: `ColorDialog`, `OpenFileDialog`, `SaveFileDialog`, `FontDialog`, `FolderBrowserDialog`, itp.,
- typy do budowy menu: `Menu`, `MainMenu`,`MenuItem`, `ContextMenu`
- typy pomocnicze: `ClipBoard`, `Help`, `Timer`, `Screen`, `ToolTip`, `Cursors`, itp.,
- kontrolki potomne: `StatusBar`, `ScrollBar`, `ToolBar`, itp.

3.3. Struktura aplikacji z GUI

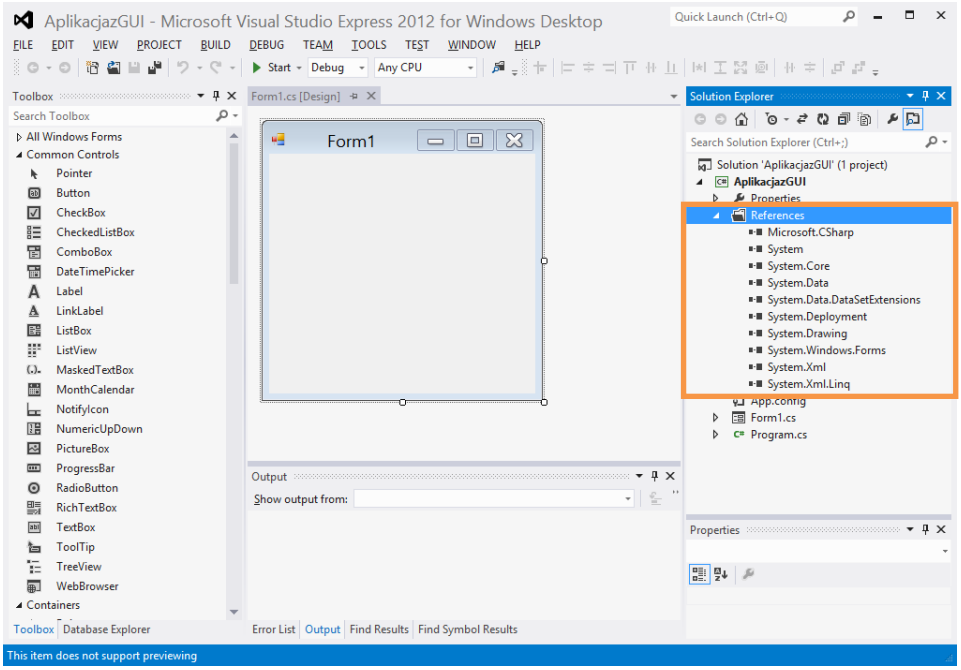
Podczas tworzenia nowego projektu z GUI (Windows Forms Application) automatycznie dołączane są odpowiednie podzespoły umożliwiające jego prawidłową kompilację. Tworzenie nowego projektu aplikacji z GUI w środowisku VisualStudio przedstawia rysunek 3.1. Strukturę takiego projektu pokazuje rysunek 3.2.

Najważniejsze okna do pracy z aplikacją z GUI to (Rys. 3.2):

- **Solution Explorer** (zasoby projektu) z automatycznie dołączonymi przestrzeniami nazw (Rys. 3.3),
- okienko do projektowania GUI (D**esign**) z nazwą formatki i odpowiadającym jej plikiem z kodem C# (np. *form1.cs*), na której będziemy tworzyć interfejs graficzny z wykorzystaniem kontrolek GUI (Rys.3.4),
- **Toolbox** – okienko udostepniające kontrolki GUI (do przeciągnięcia myszką na obszar formatki) (Rys. 3.5),
- **Properties** – okienko z właściwościami aktualnej kontrolki (zaznaczonej w obszarze formatki) (Rys.3.6).
- **Output** – okienko z komunikatami o błędach.

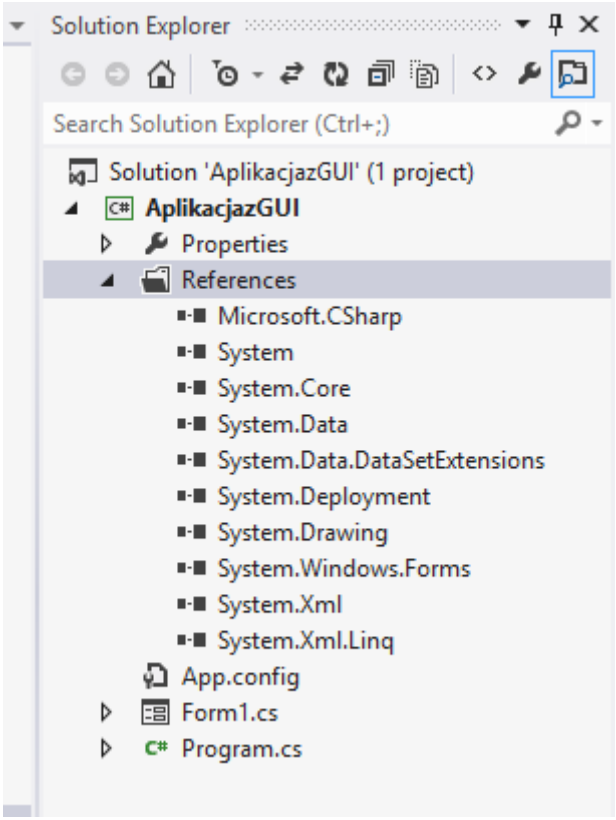


Rys. 3.1. Tworzenie nowego projektu aplikacji z GUI w VisualStudio



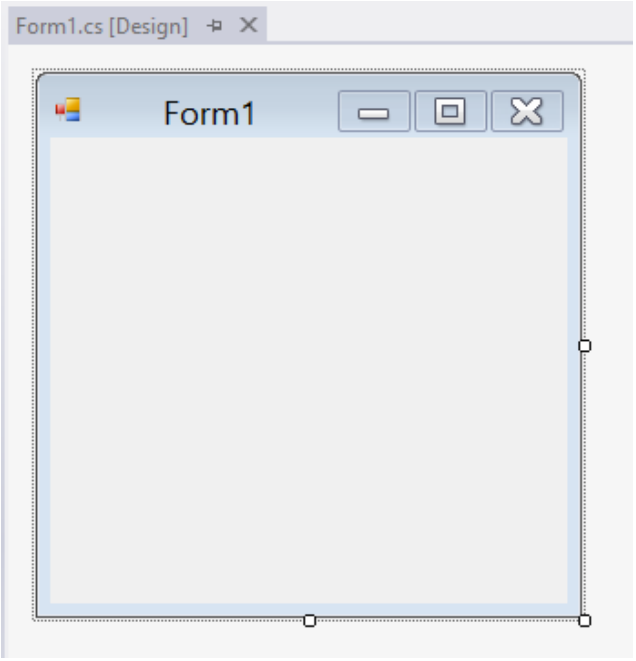
Rys. 3.2. Struktura projektu z GUI

Okienko **Solution Explorer** przedstwione jest na rysunku 3.3.

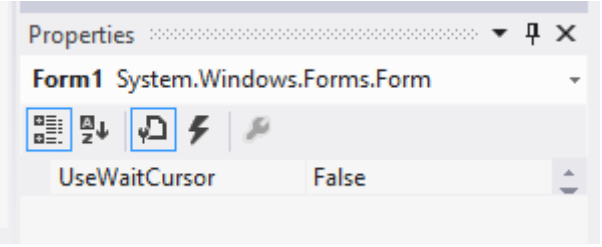


Rys. 3.3. Zasoby projektu i przestrzenie nazw automatycznie dołączane do projektu (podstawowe pliki źródłowe projektu to *program.cs* i *Form1.cs*)

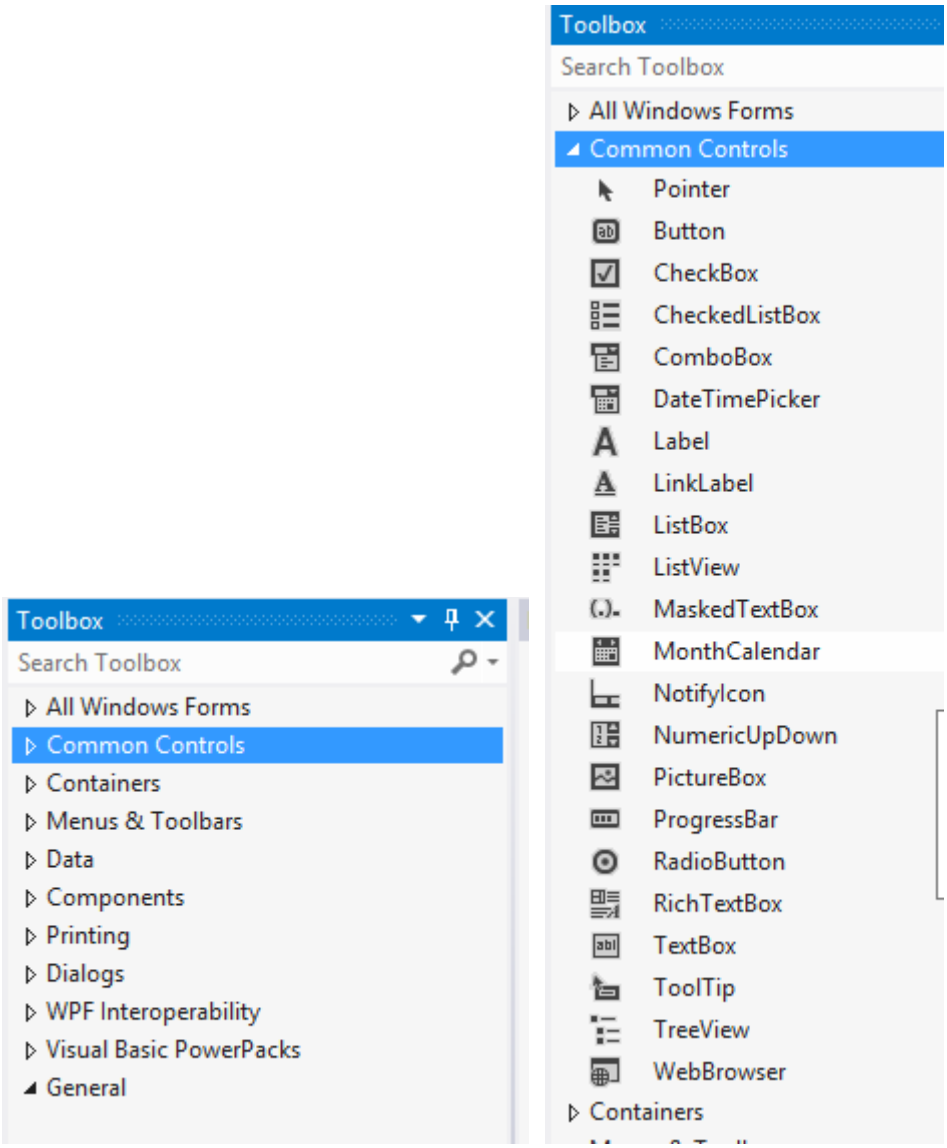
Okienko do graficznego projektowania GUI przedstawia rysunek 3.4. Dołączanie kontrolek GUI do tego okna odbywa się poprzez wybór odpowiedniej kontrolki dostępnej w oknie **Toolbox** (Rys.3.6) i przeciągnięcie jej myszą na wskazany obszar na formatce. Okienko z właściwościami aktywnej kontrolki (w tym przypadku formatki o nazwie domyślnej *Form1*) przedstawia Rys.3.5.



Rys. 3.4. Okienko do projektowania GUI (design)



Rys. 3.5. Okienko **Properties** dla Form1



Rys. 3.6. Okienko **Toolbox** z dostępnymi kategoriami kontrolek (z lewej) i kategorie kontrolek dostępne w CommonControls (z prawej)

Wejściowym plikiem aplikacji z GUI jest plik **program.cs** (patrz Rys.3.3). Kod źródłowy tego pliku przedstawia przykład 3.1.

Przykład 3.1. Kod źródłowy pliku program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace AplikacjazGUI
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Działanie aplikacji z GUI realizowane jest na bazie trzech metod klasy `Application`:

- `Application.EnableVisualStyles();`
- `Application.SetCompatibleTextRenderingDefault(false);`
- `Application.Run(new Form1());`

Rola dwóch pierwszych metod polega na ustawieniu opcji związanych z renderowaniem wynikowego okienka aplikacji. Metoda `Run()` zajmuje się wywołaniem konstruktora naszej formatki o nazwie (domyślnie) `Form1`. Fragment klasy `Form1` (`public partial class Form1`) przedstawia przykład 3.2. (patrz Rys.3.3)

Przykład 3.2. Fragment klasy Form1 z pliku Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;
using System.Windows.Forms;

namespace AplikacjazGUI
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

W konstruktorze formatki `Form1`, wywołana jest tylko jedna metoda inicjalizująca komponenty na formatce: `InitializeComponent()`. Poza kodem zawartym w pliku *Form1.cs* (jak pokazano w przykładzie 3.2), środowisko Visual Studio automatycznie generuje plik *Form1.designer.cs*, w którym do klasy formatki dołącza obiekty kontrolek (jako pola klasy `Form1`) umieszczanych przez programistę na formatce w widoku *design* oraz dodatkowo ustawia ich właściwości, zgodnie z życzeniem programisty. Fragment pliku *Form1.designer.cs* przedstawia przykład 3.3. Zdefiniowano tu dwie ważne metody:

```
protected override void Dispose(bool disposing)
```

oraz

```
private void InitializeComponent().
```

Na etapie początkowym projektowania aplikacji z GUI mamy do dyspozycji jedynie jeden komponent generujący okienko ramowe (kontener do umieszczania pozostałych kontrolek GUI) aplikacji (formatka o domyślnej nazwie *Form1*).

Przykład 3.3. Fragment klasy Form1 z pliku Form1.designer.cs

```
namespace AplikacjazGUI
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
```

```
        /// <param name="disposing">true if managed resources should
        /// be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.AutoScaleMode =
                            System.Windows.Forms.AutoScaleMode.Font;
            this.Text = "Form1";
        }

        #endregion
    }
}
```

Metoda `InitializeComponent()` jest modyfikowana automatycznie przez *Form Designer* i odzwierciedla modyfikacje formatki i jej kontrolek wprowadzone w Visual Studio. Klasa wyprowadzona z `Form` wywołuje metodę `InitializeComponent()` w zasięgu domyślnego konstruktora. Przesłonięta metoda `Dispose()` wywoływana jest automatycznie gdy formatka ma zostać zniszczona (bezpieczny sposób na uwolnienie wszystkich alokowanych zasobów).

3.4. Klasa Application

Klasa `System.Windows.Forms.Application` pozwala kontrolować niskopoziomowe zachowanie aplikacji Windows Forms i zawiera metody oraz właściwości odpowiadające za obsługę komunikatów oraz wątków.

Podstawowe metody klasy to:

- `Run()` – rozpoczyna wykonywanie standardowej pętli komunikatów aplikacji w bieżącym wątku,

- `Restart()` – ponownie uruchamia aplikację,
- `Exit()` – zamyka aplikację,
- `ExitThread()` – wychodzi z pętli komunikatów bieżącego wątku i zamyka wszystkie należące do niego okna.

Właściwości klasy to np.:

- `CompanyName`,
- `ProductName`,
- `ProductVersion`,
- `ExecutablePath` (wyświetla ścieżkę do programu, włączając w to jego nazwę), itp.

Do najważniejszych zadarzeń związanych z klasą `Application` należą zdarzenia:

- `ThreadExit` – pojawia się kiedy wątek w aplikacji ma się zakończyć,
- `ApplicationExit` – pojawia się tuż przed zamknięciem aplikacji,

Wiele zdarzeń związanych z kontrolkami GUI używa delegacji typu `System.EventHandler`, postaci:

```
public delegate void EventHandler(object sender,
                                  EventArgs e);
```

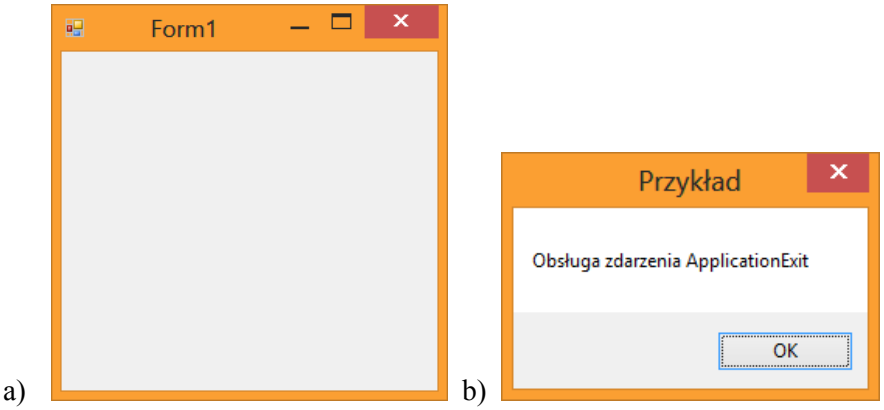
Przykład 3.4. przedstawia logikę obsługi zdarzeń w aplikacjach Windows Forms na przykładzie zdarzenia aplikacji `ApplicationExit`. Obsługa zdarzenia zostaje oddelegowana do metody `Form_OnExit`. Parametrem metody jest obiekt, dla którego zaszło zdarzenie (`object sender`) oraz samo zdarzenie jako obiekt `e` (typu `EventArgs`). Zwróćmy uwagę, że delegacja obsługi zdarzenia jest realizowana w konstruktorze, po wywołaniu metody `InitializeComponent()`.

Przykład 3.4. Obsługa zdarzenia ApplicationExit

```
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace AplikacjazGUI
{
    public partial class Form1 : Form
    {
```

```
        public Form1()
        {
            InitializeComponent();
            //przechwyć zdarzenie ApplicationExit
            Application.ApplicationExit +=
                    new EventHandler(Form_OnExit);
        }
        //obsługa zdarzenia
        private void Form_OnExit(object sender, EventArgs e)
        {
            //pokaż okienko typu MessageBox z komunikatem:
            MessageBox.Show("Obsługa zdarzenia ApplicationExit",
                            "Przykład");
        }
    }
}
```

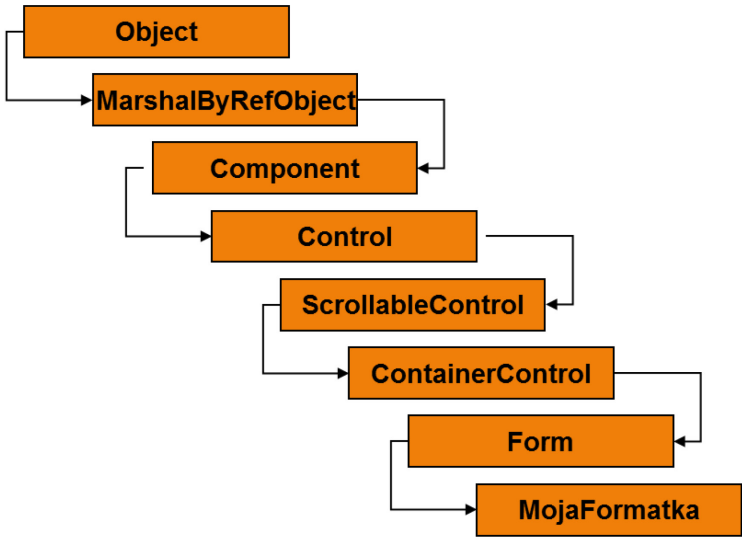
W momencie zamykania aplikacji z przykładu 3.4 pojawi się okienko z komunikatem widoczne na rysunku 3.7.



Rys. 3.7. a) Widok okna aplikacji, b) wynik obsługi zdarzenia zamknięcia aplikacji z przykładu 3.4

3.5. Anatomia formatki

Obiekt klasy `Form` to kontener (okno aplikacji), do którego możemy dodawać kontrolki GUI. Miejsce klasy `Form` w hierarchii dziedziczenia klas przedstawia rysunek 3.8.



Rys. 3.8. Klasa Form w hierarchii dziedziczenia klas

3.5.1. Klasa Component i Control

Klasa `Component` implementuje `interfejs IComponent` i metodę `Dispose()` (wywoływaną kiedy komponent nie jest już potrzebny). Kiedy formatka zostaje zamknięta – metoda `Dispose()` zostaje wywoływana automatycznie dla typu `Form` i dla wszystkich kontrolek znajdujących się na formatce.

Klasa `Control` zawiera zachowania wymagane przez każdy typ związany z interfejsem graficznym (rozmiar i położenie kontrolki, pozwala przechwytywać dane wprowadzane za pomocą myszy i klawiatury).

Najważniejsze właściwości klasy `Control` to:

- `AllowDrop` – czy dany komponent może być rodzicem dla innych,
- `Anchor` – w jaki sposób komponent ma zmieniać swoje rozmiary, jeżeli kontrolka znajdująca się w nim zmienia swój rozmiar,
- `AutoSize` – czy komponent ma mieć rozmiar zgodny z wymiarami treści, która się w nim znajduje,
- `BackColor`, `BackgroundImage`, `Font`, `ForeColor`, `Cursor` – opisują podstawowe właściwości komponentu (kolor tła, obraz w tle, właściwości czcionki, kolor pierwszoplanowy, właściwości kursora),
- `Enabled` – czy komponent będzie wyłączony,

- `Dock` – do której krawędzi kontenera, kontrolka będzie dokowana,
- `Height`, `Width` – umożliwia odczytanie lub przypisanie szerokości kontrolki,
- `Left`, `Right` – umożliwia określenie (w pikselach) odległości lewej/prawej krawędzi od obszaru formularza lub innego komponentu,
- `Location`, `Size` - właściwość typu `Point` – położenie/rozmiar kontrolki,
- `Name` – nazwa komponentu,
- `Opacity` – nieprzejrzystość kontrolki w % (0.0 – pełna przezroczystość),
- `Padding` – odstępy (min. odległości) wewnątrz danej kontrolki,
- `Parent` – kontrolka macierzysta dla danego komponentu,
- `Top` – określa odległość (w pikselach) od górnej krawędzi formularza lub komponentu,
- `Visible` – określa widoczność komponentu.

Najwazniejsze metody klasy `Control` to:

- `GetStyle()`, `SetStyle()` – pobranie/ustawienie stylu dla kontrolki,
- `Hide()`, `Show()` - pokazywanie/ukrywanie kontrolki,
- `Invalidate()` – wymusza ponowne narysowanie kontrolki (można określić tylko odświeżany prostokąt a nie cały obszar kliencki),
- `ResetFont()`, `ResetCursor()`, `ResetForeColor()`, `ResetBackColor()`,
- `Refresh()` – wymusza unieważnienie typu `Control` i ponowne narysowanie go (i wszystkich jego potomków),
- `SetBounds()`, `SetLocation()`, `SetClientArea()`.

Zdarzenia związane z klasą `Control` to:

- wyzwalane w odpowiedzi na dane wprowadzane myszą:
  - `Click`, `DoubleClick`,
  - `MouseEnter`, `MouseLeave`,
  - `MouseDown`, `MouseUp`,
  - `MouseMove`, `MouseHover`, `MouseWheel`,
- wyzwalane w odpowiedzi na dane wprowadzane z klawiatury:
  - `KeyPress`,
  - `KeyUp`,
  - `KeyDown`.

Obsługując zdarzenia kontrolki związane z myszą mamy do dyspozycji następujące właściwości typu `MouseEventArgs`:

- `Button` – sprawdza, który przycisk myszy został wciśnięty,

- `Clicks` – sprawdza liczbę naciśnięć i zwolnień przycisku myszy,
- `Delta` – sprawdza o ile jednostek (dodatnich lub ujemnych) obrócono kółko myszy,
- X, Y – pobiera współrzędne kliknięcia myszą.

Obsługę zdarzenia `MouseUp` i śledzenie ruchu myszy (zdarzenie `MouseMove`) pokazuje przykład 3.5.

Przykład 3.5. Obsługa zdarzenia myszyMouseUp i OnMouseMove

```
public class Form1 : Form
{
    public static int Main(string[] args)
    {
        Application.Run(new Form1()); return 0;
    }
    public Form1()
    {  //śledzimy ruch myszy
        this.MouseUp += new MouseEventHandler(OnMouseUp);
        this.MouseMove += new MouseEventHandler(OnMouseMove);
    }

    public void OnMouseMove(object sender, MouseEventArgs e)
    {  //po kliknięciu wyświetla się komunikat:
        MessageBox.Show = string.Format("Przestań klikać!");
    }
    public void OnMouseUp(object sender, MouseEventArgs e)
    {
        if (e.Button == MouseButtons.Left)
            MessageBox.Show("Lewy");
        if (e.Button == MouseButtons.Right)
            MessageBox.Show("Prawy");
        if (e.Button == MouseButtons.Middle)
            MessageBox.Show("Środek");
    }
}
```

Do obsługi zdarzeń można również wykorzystać inne podejście a mianowicie przesłonić odpowiednie metody klasy bazowej `Control`. W takiej technice przykład 3.5 możemy zapisać w postaci przykładu 3.6.

Prezykład 3.6. Obsługa zdarzeń – przesłonięcie metod klasy Control

```
public Form1()
{ // przesłaniając metody nie trzeba już tego pisać
  // this.MouseUp += new MouseEventHandler(OnMouseUp);
  // this.MouseMove += new MouseEventHandler(OnMouseUp);
```

```
}
protected override void OnMouseUp( /*object sender,*/
                                   MouseEventArgs e)
{  //to samo
}
protected override void OnMouseMove( /*object sender,*/
                                     MouseEventArgs e)
{ this.Text = string.Format("Bieżąca pozycja {0}, {1}", e.X, e.Y);
}
```

Obsługując zdarzenia związane z klawiaturą mamy do dyspozycji następujące właściwości typu `KeyEventArgs`:

- `Alt` – wartość informująca o naciśnięciu klawisza *Alt,*
- `Control` – wartość informująca o naciśnięciu *Control,*
- `Handled` – uzyskuje lub modyfikuje wartość informującą czy zdarzenie zostało obsłużone,
- `KeyCode` – uzyskuje kod klawisza dla zdarzenia `KeyDown` lub `KeyUp`,
- `KeyData` – uzyskuje dane klawisza dla zdarzenia `KeyDown` lub `KeyUp`,
- `Modifiers` – informuje o naciśnięciu jednego z klawiszy modyfikujących (*Ctrl*, *Shift*, *Alt)*,
- `Shift` – wartość informująca o naciśnięciu klawisza *Shift.*

3.5.2. Klasa ScrollableControl i ContainerControl

Klasa `ScrollableControl` zawiera składowe do obsługi pionowych i poziomych pasków przewijania reprezentowanych przez klasy kontrolek `ScrollBar` (pasek poziomy: `HScrollBar` i pasek pionowy: `VScrollBar`).

Klasa `ContainerControl` obsługuje stan aktywności (focus) danego elementu interfejsu graficznego. Jest przydatna w momencie tworzenia formatki zawierającej wiele kontrolek potomnych, przy czym użytkownik może używać klawisza *Tab* do zmiany zaznaczenia.

Podstawowe składowe klasy `ContainerControl` to:

- `ActiveControl`, `ParentForm` – właściwości te pozwalają odczytać i ustawić aktywną kontrolkę oraz pobrać referencję do formatki, na której kontrolka ta się znajduje;
- `ProcessTabKey()` – metoda pozwala programistycznie aktywować klawisz *Tab*, aby uaktywnić nastepną dostępną kontrolkę.

3.5.3. Klasa Form

Do najważniejszych metod klasy `Form` zaliczamy:

- `Activate()` – włącza formatkę i uaktywnia ją,
- `Close()` – zamyka formatkę,

- `CenterToScreen()` – umieszcza formatkę w środku ekranu,
- `LayoutMDI()` – umieszcza wszystkie formatki potomne w obrębie formatki nadrzędnej,
- `OnResize()` – można przesłonić w celu reagowania na zdarzenie `Resize`.

Zdarzenia klasy `Form` to:

- `Activate` – wysyłane, kiedy formatka jest przenoszona na pierwszy plan aktywnej aplikacji;
- `Closed`, `Closing` – zdarzenia te pozwalają ustalić, kiedy formatka ma zostać zamknięta;
- `MDIChildActive` – wysyłane, kiedy uaktywniane jest okno potomne.

3.5.4. Obsługa zdarzeń w Visual Studio

W aplikacji Windows Forms tworzonej w środowisku Visual Studio z przykładową kontrolką `Button` umieszczoną na formatce, po podwójnym kliknięciu na kontrolkę automatycznie generowany jest kod metody do obsługi zdarzenia kliknięcia na przycisk:

```
private void button1_Click(object sender, EventArgs e) {  }
```

W pliku `*.designer.cs` znajdują się instrukcje odpowiedzialne za utworzenie przycisku `Button` na form
ularzu (jako pole prywatne formatki):

```
private System.Windows.Forms.Button button1;
```

oraz za przypisanie określonej metody do danego zdarzenia:

```
this.button1.Click +=
                    new System.EventHandler(this.button1_Click);
```

Zdarzenie `Click` zadeklarowane jest w klasie `Control` jako:

```
public event EventHandler Click;
```

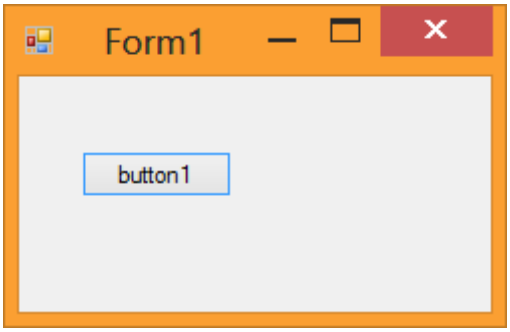
`EventHandler` nie należy do słów kluczowych języka C# - to nazwa delegata określonego w przestrzeni `System` jako:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Metody zdarzeniowe przypisywane do zdarzenia `Click` muszą mieć sygnaturę zgodną z delegatem `EventHandler` - pierwszym parametrem sygnatury delegata jest parametr typu `object`, drugim typu `EventArgs`. Klasa `EventArgs` jest jedynie klasą bazową dla innych, które przechowują informacje o zdarzeniu.

Przykład 3.7 przedstawia kod pliku *Form1.cs* dla formatki z przyciskiem (Rys.3.9). Przykład 3.8 pokazuje kod automatycznie wygenerowanej metody

obsługi zdarzenia kliknięcia na przycisk, umieszczonej w pliku *Form1.designer.cs*.



Rys.3.9. Formatka z przyciskiem

Przykład 3.7. Plik Form1.cs

```
using System;
using System.Windows.Forms;

namespace AplikacjazGUI
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            //instrukcje do wykonania po kliknięciu na przycisk
        }
    }
}
```

Przykład 3.8. Kod z wygenerowanego pliku Form1.designer.cs

```
namespace AplikacjazGUI
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
```

```
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources
        /// should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(31,
37);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(75, 23);
            this.button1.TabIndex = 0;
            this.button1.Text = "button1";
            this.button1.UseVisualStyleBackColor = true;
            this.button1.Click += new
                        System.EventHandler(this.button1_Click);
            //
            // Form1
            //
            this.AutoScaleDimensions =
                        new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode =
                        System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(284, 256);
            this.Controls.Add(this.button1);
```

```
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);

        }

        #endregion

        private System.Windows.Forms.Button button1;
    }
}
```

3.5.5. Okna dialogowe SaveFileDialog i OpenFileDialog

Do tworzenia aplikacji współpracującej z plikami bardzo przydatne są okna dialogowe `SaveFileDialog` i `OpenFileDialog`, które możemy znaleźć w Visual Studio w oknie **Toolbox** w kategorii **Dialogs.**

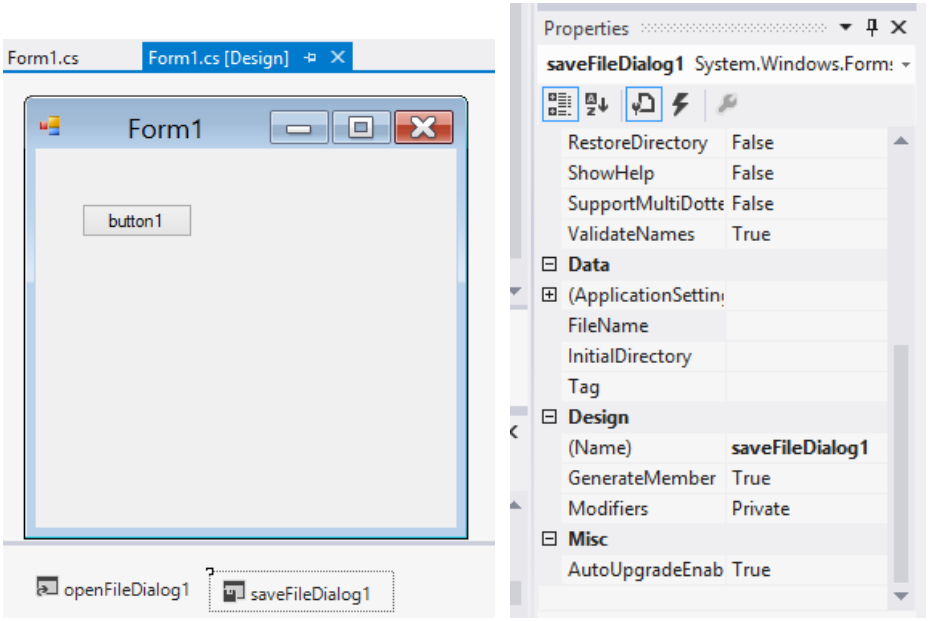
Manualne utworzenie okna dialogowego (jak również każdej innej kontrolki) to utworzenie prywatnego pola w klasie formatki np.::

```
  private SaveFileDialog saveFileDialog = new
                      SaveFileDialog();
  private OpenFileDialog openFileDialog = new
                      OpenFileDialog();
```

Konfigurację wyglądu i zachowania tak utworzonych okienek można wykonać z wykorzystaniem okna **Properties** w Visual Studio (Rys. 3.10) lub ręcznie np.;

```
public Form1()
{
  InitializeComponent();
  saveFileDialog.InitialDirectory = ".";
  saveFileDialog.Filter = "car files (*.car)|*.car|*.*|";
  saveFileDialog.FilterIndex = 1;
  saveFileDialog.RestoreDirectory = true;
  saveFileDialog.FileName = "carDoc";
  openFileDialog.InitialDirectory = ".";
  openFileDialog.Filter = "car files (*.car)|*.car|*.*|";
  openFileDialog.FilterIndex = 1;
  openFileDialog.RestoreDirectory = true;
}
```

Rysunek 3.9 pokazuje miejsce osadzenia okienek dialogowych w przypadku wykorzystania okna **Toolbox** w Visual Studio i przeciągnięcia kontrolek na okno przykładowej formatki *Form1* oraz okienko **Properties** z możliwością konfiguracji wskazanego obiektu GUI.



Rys.3.10. Okna dialogowe *FileOpenDialog* i *FileSaveDialog* w widoku *Form1.cs* (Design) oraz okienko **Properties** z właściwościami dla kontrolki *saveFileDialog*

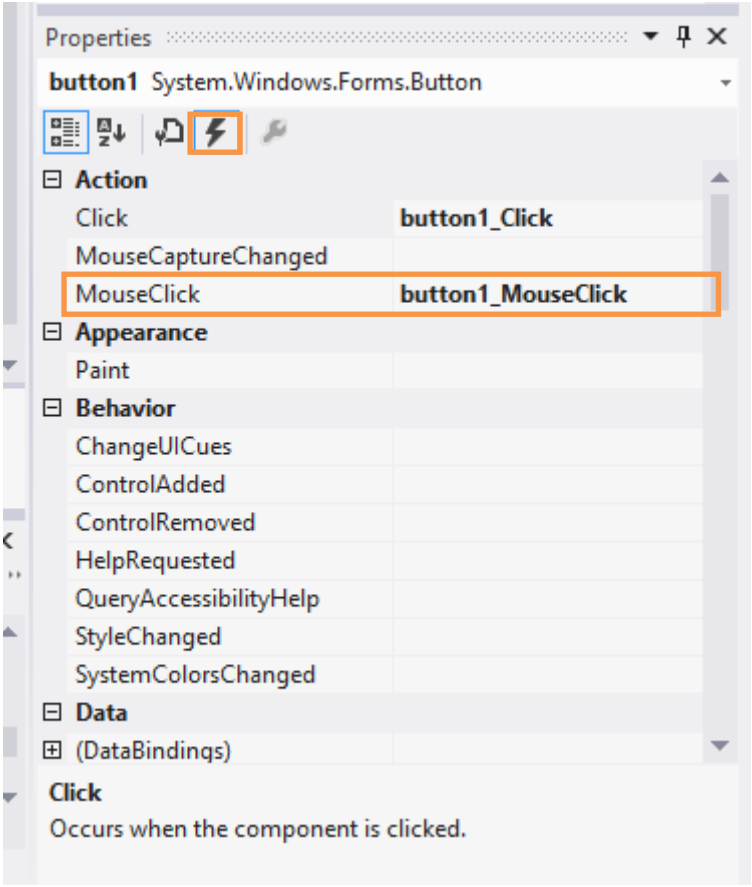
3.5.6. Obsługa zdarzeń w Visual Studio

Projektując interfejs graficzny w środowisku Visual Studio korzystamy z dostępnych kontrolek przeciąganych na obszar formatki. Właściwości każdej kontrolki można ustawiać programowo lub wykorzystać (w fazie projektowania) okno **Properties.** Obsługę zdarzeń związanych z konkretną kontrolką możemy również zrealizować korzystając ze wsparcia jakie oferuje nam środowisko Visual Studio. Na przykład, dla przycisku na formatce możemy automatycznie wygenerować metody do obsługi różnych zdarzeń, wybranych w okienku **Properties**, ale na zakładce z ikoną piorunka (Rys. 3.11).

Automatyczne wygenerowanie nagłówka metody do zdarzeń wskazanych w tym okienku następuje po podwójnym kliknięciu na wybrane zdarzenie. Na przykład, dla zdarzenia `MouseClick`, w pliku *Form1.cs* został wygenerowany nagłówek postaci:

```
private void button1_MouseClick(object sender, MouseEventArgs e)
{   }
```

Proces projektowania GUI w Visual Studio jest bardzo prosty. Najtrudniejszym zadaniem dla programisty jest oczywiście oprogramowanie automatycznie generowanych metod do obsługi zdarzeń.



Rys. 3.11. Okienko **Properties** z widokiem do obsługi zdarzeń

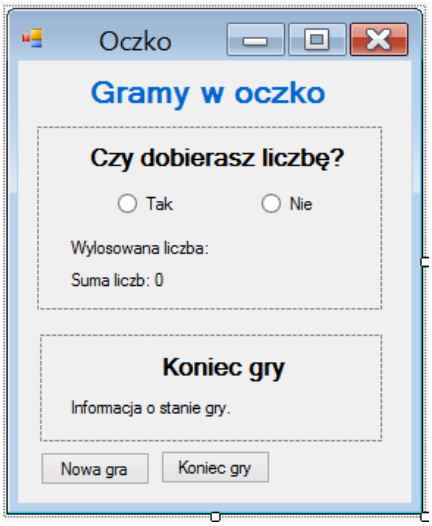
3.6. Zadania do samodzielnego rozwiązania

Zadania w tym rozdziale, poza tworzeniem aplikacji z graficznym interfejsem użytkownika, demonstrują również mechanizmy omówione w rozdziale 2.3, dotyczące zaawansowanych możliwości programowania obiektowego w C#.

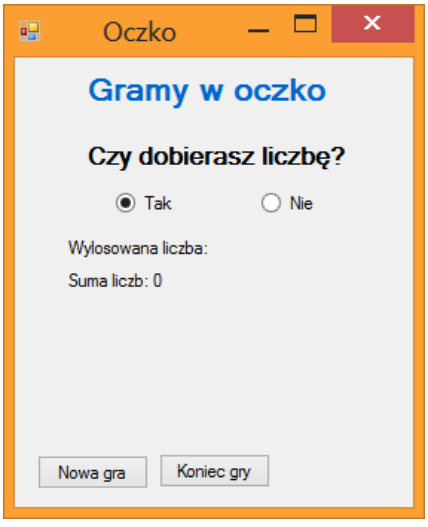
Zadanie 3.1.

a) W środowisku Visual Studio utworzyć nowy projekt *Windows Forms Application*, na którym należy zaprojektować interfejs graficzny do gry w Oczko (Rys. 3.12a). GUI aplikacji składa się z etykiety tytułowej (kontrolka *Label* z grupy *Common Controls* w okienku **Toolbox** Visual Studio), dwóch paneli (kontrolka *Panel* z grupy *Containers*) oraz z dwóch przycisków na dole okna aplikacji (kontrolka *Button* z grupy *Common Controls*). Pierwszy panel zawiera etykietę (*Label)*, dwa przyciski typu radio (kontrolka *RadioButton* z grupy *CommonControl*) oraz dwie kolejne etykiety, w których będzie umieszczona informacja o wylosowanej liczbie oraz o sumie już wylosowanych liczb. Drugi panel (na początku ukryty) pokazuje się w momencie, gdy użytkownik przerwie dobieranie kart lub gdy suma już wylosowanych liczb przekroczy 21. Drugi panel zawiera w ety-kiecie informację o rezultacie gry. Pozmieniaj odpowiednio właściwości kontrolek korzystając z okienka **Properties** (np. właściwość **Text**, **Font**, **ForeColor** itp.). Dla panelu drugiego ustaw właściwość widoczności: **Visible** na **false** (panel pokazuje się dopiero, gdy użytkownik nie dobiera już liczb). Widok po uruchomieniu aplikacji przedstawia rysunek 3.12 b.

a)



b)



Rys.3.12. a) Interfejs graficzny aplikacji do gry w Oczko widoczny w okienku projektowania Visual Studio, b) GUI aplikacji po uruchomieniu

b) Do realizacji logiki gry do zasobów projektu (w okienku **Solution Explorer** po wskazaniu folderu z zasobami aplikacji należy wykorzystać menu kontekstowe dostępne pod prawym klawiszem myszy *Add->Class*) dodaj pomocniczą klasę (plik *Gracz.cs*) `Gracz`, której schemat może wyglądać następująco:

```
class Gracz
{
    //prywatne pola klasy
    private int Suma = 0;
    private int Liczba = 0;
    private int StaraLiczba = 0;
    private bool Koniec = false;

    //publiczna metoda klasy
    public void LosujLiczbe(int l)
    {
        StaraLiczba = Liczba;
        Liczba = l;
        //oblicz sume
        //sprawdz sume
    }

    //właściwości klasy – tylko do odczytu:
    public bool GetKoniec
    {
        get { return Koniec; }
    }

    public int GetLiczba
    {
        //uzupełnić
    }

    public int GetSuma
    {
        //uzupełnić
    }

    public string getInfo
    {
        get
        {
            string Info = "";
            if (Suma == 21)
                Info = " Wygrałeś - Oczko";
```

```
                else
                    Info = " ...";
                //uzupełnić

                return Info;
            }
        }
    }
```

c) W klasie formatki `Form1` zadeklarować dwa pola prywatne np.:

```
Gracz g;
Random r;
```

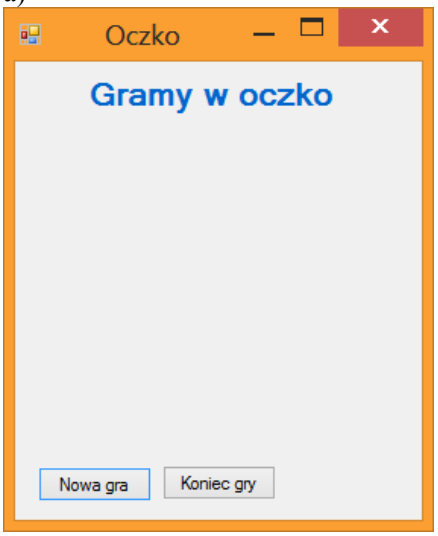
a konstruktor formatki może mieć postać np.:

```
public Form1()
 {
     InitializeComponent();
     panel1.Visible = false;
     panel2.Visible = false;
     g = new Gracz();
     r = new Random();
 }
```

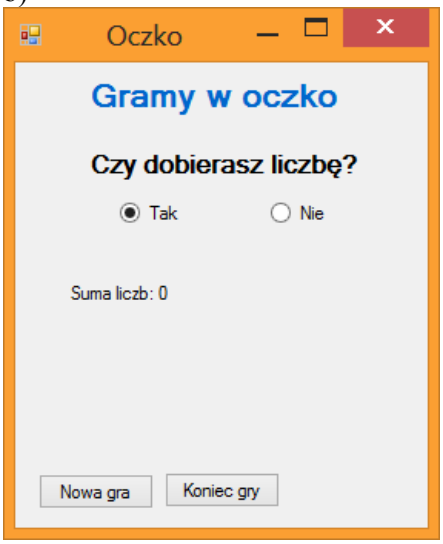
d) w celu opóźnienia działania programu (przycisk radio odznaczyć po 3 sekundach) można zastosować metodę:
`System.Threading.Thread.Sleep(300);`

Przykładowe działanie aplikacji pokazują rysunki 3.13 i 3.14.

a)

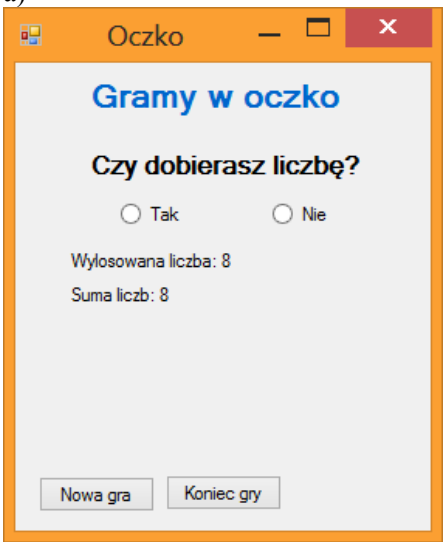


b)

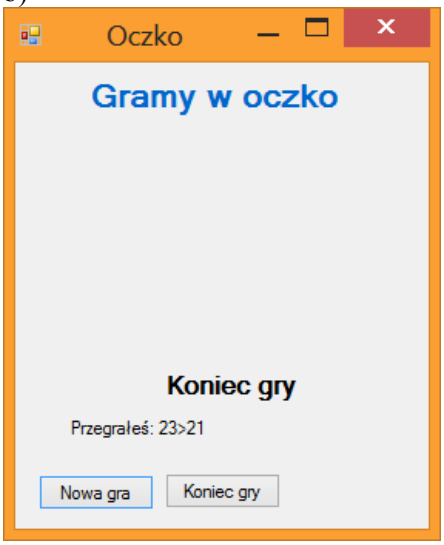


Rys.3.13. a) Widok po uruchomieniu, b) Widok po wyborze nowej gry

a)



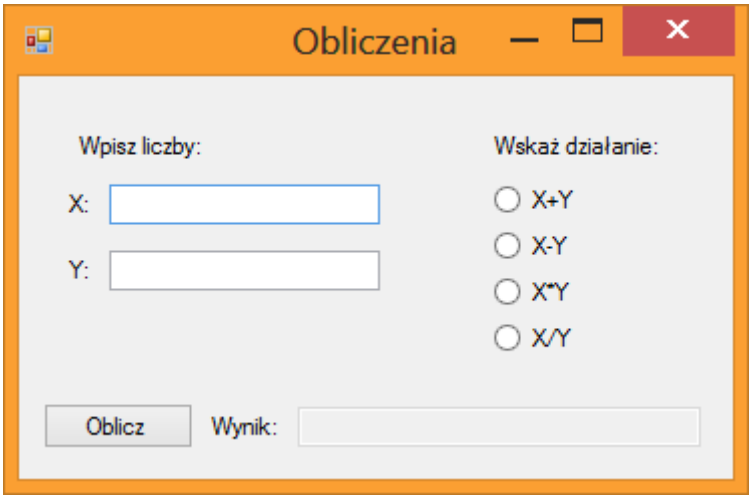
b)



Rys.3.14. a) Widok w trakcie gry, b) Zakończenie gry

Zadanie 3.2.

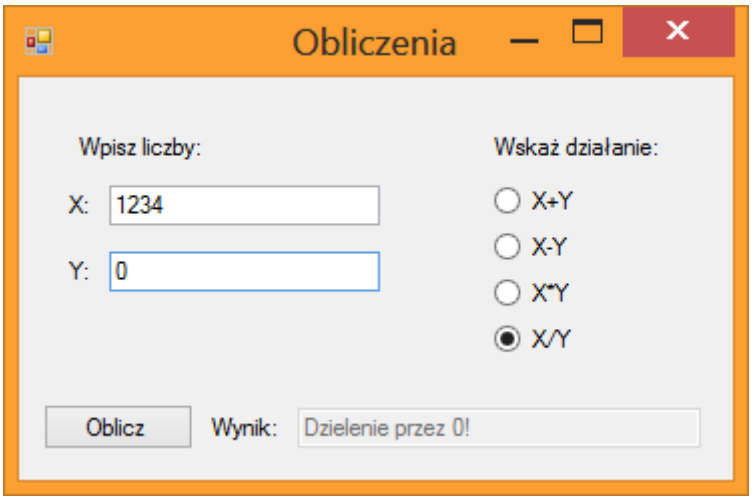
a) Utworzyć aplikację **Obliczenia** z GUI jak pokazano na rysunku 3.15.



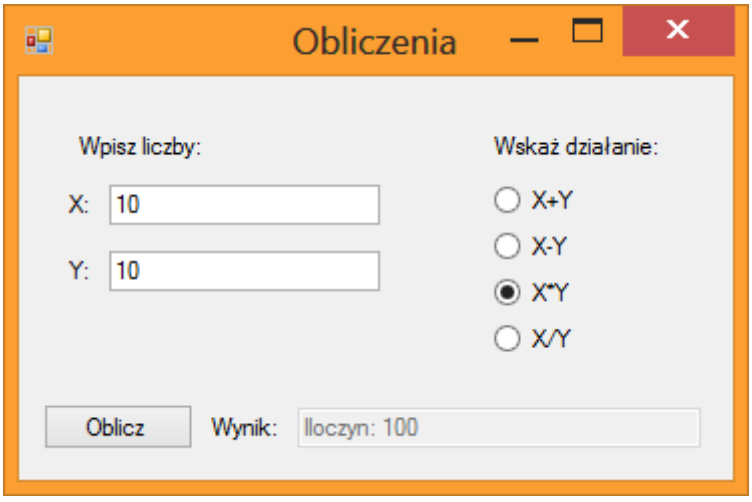
Rys.3.15. Widok formularza obliczeniowego

b) zdefiniować delegata **Dzialanie** oraz odpowiadające mu metody **Dodawanie, Odejmowanie**, **Mnozenie** i **Dzielenie**.
c) dodać obsługę przycisku **Oblicz**, który w zależności od wybranego działania utworzy odpowiedni obiekt delegata i wyświetli wynik działania wskazanego .przez użytkownika,
d) zmodyfikować kod tak aby wykorzystać wyrażenie lambda.

Efekt działania aplikacji pokazują rysunki 3.16 i 3.17.



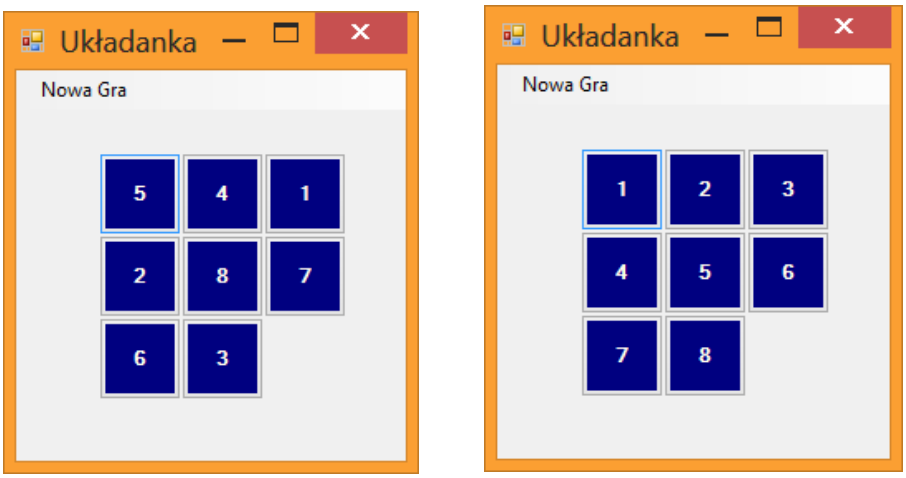
Rys.3.16. Widok formularza obliczeniowego



Rys.3.17. Widok formularza obliczeniowego

Zadanie 3.3.

Zrealizować aplikację **Układanka**. Przykładowe okno aplikacji przedstawia rys.3.18.



Rys.3.18. Plansza gry - początkowa i końcowa

Celem gry jest ułożenie przycisków z cyframi w odpowiedniej kolejności. Przyciski są tworzone jako elementy listy (`List`) w pętli w pomocniczej klasie `Gra`. Dla każdego przycisku programowo (bez korzystania z narzędzia designer) ustawiane są jego właściwości (rozmiary, położenie, kolor tła, kolor tekstu, tekst opisujący przycisk itp.) oraz tworzony jest uchwyt do metody obsługującej zdarzenie kliknięcia na przycisk.

Klasa **Gra** może być postaci:

```
class Gra
    {   readonly int ile=9;  //liczba przycisków na planszy
        List <Button> tab=new List<Button>();   //lista przycisków
        List<int> opis; //lista kolejnych liczb
                        // 1..ile (opisy na przyciskach)
        int niewidoczny;//przycisk z liczbą=ile będzie niewidoczny
        int n;//pierwiastek z ile (plansza ma wymiar nxn)

        public Gra(int ile=9)      //konstruktor klasy
        {
            this.ile = ile;
            opis = new List<int>();
            for (int i=1;i<=ile;i++)
             opis.Add(i); //dodajemy kolejne liczby 1..ile do listy
            Random r=new Random();
            //utworz tablice przycisków:
            n=(int)Math.Sqrt(ile); //plansza układanki nxn
```

```
        for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            {   tab.Add(new Button()); //dodaj kolejny przycisk
                                       //do listy
                int i1 = tab.Count-1;
                //losujemy indeks z listy
                int indeks = r.Next(0, opis.Count - 1);
                //ustawiamy opis na przycisku:
                tab[i1].Text = opis[indeks] + "";
                // jeśli wylosowany indeks jest równy ile
                // przycisk o indeksie i1 będzie niewidoczny
                if (opis[indeks] == ile) niewidoczny = i1;
                //usuwamy wylosowaną liczbę z listy –
                //(losujemy bez powtórzeń)
                opis.RemoveAt(indeks);
                //ustawiamy własności przycisku
                //na liście o indeksie i1:
                tab[i1].Visible = true;
                tab[i1].Size = new System.Drawing.Size(50, 50);
                tab[i1].Location =
                   new System.Drawing.Point(50 + j*50, 50+i*50);
                //dodajemy uchwyt do metody obsługującej
                //kliknięcie:
                tab[i1].Click +=
                    new System.EventHandler(this.button_Click);
            }
            tab[niewidoczny].Visible = false;
        } //koniec konstruktora

       //metody klasy Gra:
        public List<Button> PobierzListePrzyciskow()
        {
            return tab;     //metoda zwraca liste z przyciskami
        }

        private void button_Click(object sender, EventArgs e)
        {
            Button k = (Button) sender;  //k – przycisk kliknięty
            //sprawdz_ktory przycisk kliknięty - pobierz jego indeks
            int ktory=tab.IndexOf(k);
            //jesli przycisk leży obok niewidocznego - zamien
            //przyciski miejscami
            sprawdz_i_zamien(k,ktory);
       }
```

```
        private void sprawdz_i_zamien(Button k,int ktory)
        {
               bool zamiana=false;
               //sprawdz czy klikniety przycisk sąsiaduje z
               //niewidocznym
               //uzupełnić

               //zamieniamy:
               if (zamiana)
               {
                   //uzupełnić
                   niewidoczny = który;
               }
               sprawdz_czy_koniec();
        }

        private void sprawdz_czy_koniec()
        {
            //uzupełnić
        }
}
```

W klasie formularza *Windows Form* należy utworzyć obiekt **Gra**, pobrać jego listę przycisków oraz dodać je do listy kontrolek formatki, tak jak to pokazano na listingu:

```
public partial class Form1 : Form
    {
        int ile = 9;
        Gra g;
        List <Button>lista;
        public Form1()
        {  InitializeComponent();
           g = new Gra(ile);
           ustaw_plansze();
        }
        private void ustaw_plansze()
        {   lista=g.PobierzListePrzyciskow();
            for (int i=0;i<ile;i++)
                Controls.Add(lista[i]);
        }
}
```

Do menu formatki dodać opcje wyboru rozmiaru planszy (3x3, 4x4, 5x5). Obsługa wyboru opcji menu ma powodować utworzenie nowej gry z odpowiednią liczbą przycisków (rys.3.19). Będzie konieczne opracowanie metody usuwającej kontrolki (Controls.Remove) dodane we wcześniejszej grze do klasy formatki.

Rys. 3.19. Plansza gry 5x5

Zadanie 3.4.

Zrealizować aplikację do ewidencjonowania pojazdów. Na formatce umieścić kontrolkę *MenuStrip* oraz *DataGridView* (rys. 3.20).

Następnie należy:

a) do projektu dodac nowy plik z klasą publiczną `Car` z właściwościami `Marka`, `Kolor` i `Rocznik` oraz z odpowiednim konstruktorem (patrz podpunkt c),

b) w klasie formatki zdefiniować pole z listą samochodów np.:
```
private ArrayList cars = null;
```
oraz pole (tabelę danych), w którym wyświetlimy dane o pojazdach np.:
```
private DataTable ewidencja = null; //tabela z danymi
```

c) w konstruktorze formatki dodać kilka samochodów do listy:
```
cars = new ArrayList();
cars.Add(new Car("Toyota","Srebrny",1991));
```
oraz wyświetlić dane na siatce za pomocą metody: `UpdateGrid();`

d) metoda `UpdateGrid` może być postaci:

```
private void UpdateGrid()
    {
        if (cars != null)
        {   //utwórz obiekt DataTable o nazwie ewidencja:
            ewidencja =
                        new DataTable("Ewidencja pojazdów");
            //utwórz obiekty DataColumn odpowiadające polom Car
            DataColumn marka = new DataColumn("Marka pojazdu");
            //utwórz kolumnę kolor
            //utwórz kolumnę Rocznik
            //dodaj kolumny do tabeli danych
            ewidencja.Columns.Add(marka);
            //dodaj pozostałe dwie kolumny
            //wykonaj iteracje po elementach listy w celu
            //utworzenia wierszy
            foreach (Car c in cars)
            {
                DataRow wiersz;
                wiersz = ewidencja.NewRow();
                wiersz["Marka pojazdu"] = c.Marka;
                wiersz["Kolor"] = c.Kolor;
                wiersz["Rocznik"] = c.Rok;
                //dodaj rekord do ewidencji:
                ewidencja.Rows.Add(wiersz);
            }
            //przypisz tę tablicę danych do siatki:
            dataGridView1.DataSource = ewidencja;
        }
    }
```

e) uzupełnić opcje menu aplikacji (formatki 1):
- *Utwórz nowy pojazd*
- *Wyczyść ewidencję pojazdów*
- *Zapisz plik ewidencji*
- *Otwórz plik ewidencji*
- *Koniec*

f) do aplikacji dodać kolejną formatkę (w oknie *Solution Explorer* po wskazaniu zasobów projektu pod prawym klawiszem myszy wybrać opcję *Add* a następnie *Windows Form*) wyświetlającą formularz umożliwiający dodanie nowego pojazdu do ewidencji (Rys.3.21-3.22).

Uwaga! W celu poprawnego działania aplikacji w kodzie formatki 2

do przycisku *Dodaj* należy przypisać właściwość *DialogResult.OK* natomiast do przycisku *Anuluj* właściwość *DialogResult.Cancel*

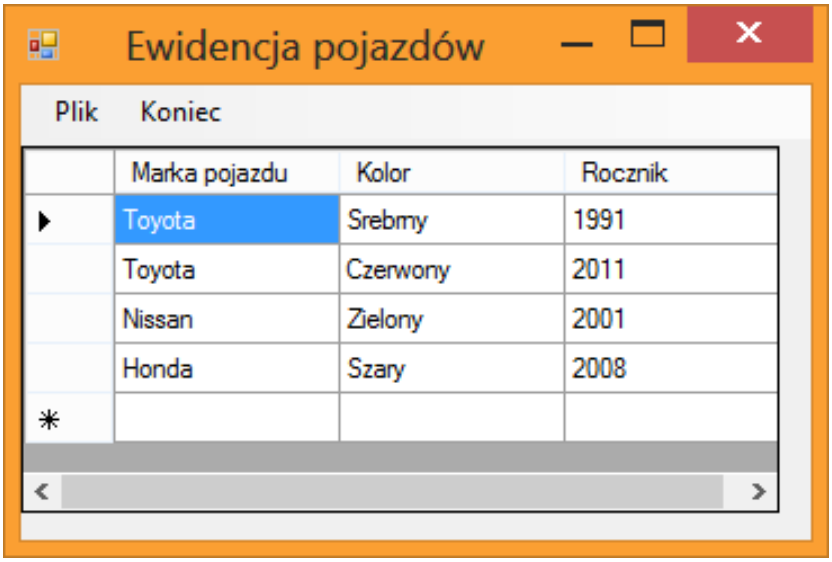
Przykładowy kod formatki ma postać:

```
public partial class Form2 : Form
{
    public Car cc; //pole Car tworzone za pomoca formularza

    public Form2()
    {
        cc = new Car();
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        cc = new Car(comboBox1.Text, comboBox2.Text,
                            Convert.ToInt32(comboBox3.Text));
    }
}
```

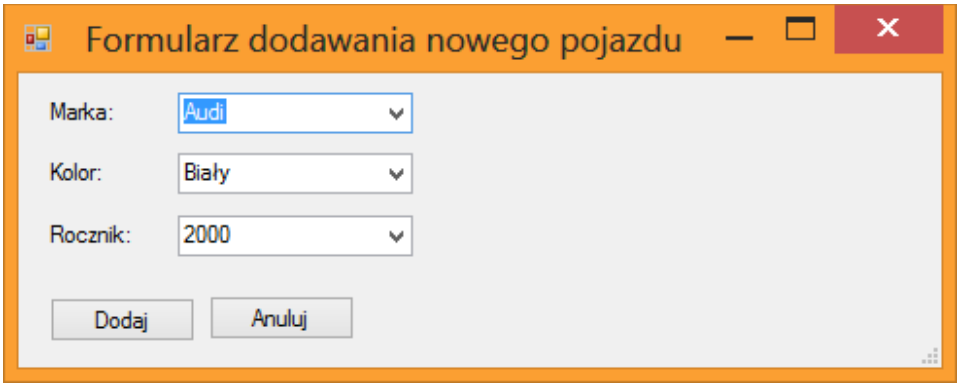
W formatce 1 obsługa wyboru opcji *Utwórz nowy pojazd* powinna otwierać okno drugiego formularza i sprawdzać czy użytkownik wybrał przycisk *Dodaj* w celu dodania nowego samochodu do ewidencji:

```
Form2 d = new Form2();
if (d.ShowDialog() == DialogResult.OK)
{
    //dodaj nowy samochód (z obiektu d) do kolekcji cars
    //zaktualizuj widok – UpdateGrid()
}
```

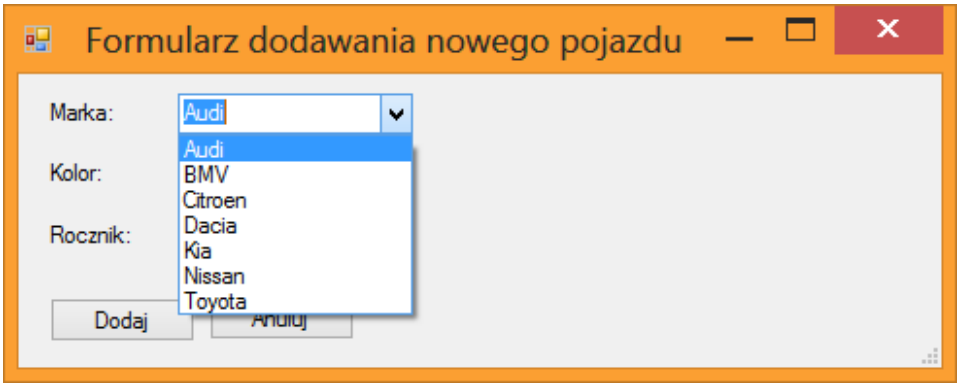
g) do okna formatki 1 dodać kolejne kontrolki *SaveFileDialog* i *OpenFileDialog*, z których należy skorzystać w procesie zapisu lub odczytu (serializacji/deserializacji) danych z pliku wskazanego w oknie przez użytkownika.



Rys.3.20. GUI aplikacji **Ewidencja**



Rys. 3.21. Formularz dodawania nowego pojazdu do ewidencji – widok początkowy



Rys. 3.22. Formularz dodawania nowego pojazdu do ewidencji - możliwości wyboru na kontrolce ComboBox

Bibliografia

1. Andrew Troelsen, Język C# 2010 i platforma .NET 4, PWN, 2011
2. Marcin Lis, C#. Praktyczny kurs, Helion, 2012
3. http://msdn.microsoft.com/en-us/library/aa288436(v=vs.71).aspx, C# tutorial
4. http://arvangen.wordpress.com/2011/08/27/c-var-delegacje-metody-anonimowe-i-wyrazenia-lambda/

Indeks

K

L

M

O

P

R

S

T

U

V

W

Z