MARCIN LIS

WYDANIE III

C#

PRAKTYCZNY KURS

Helion

Opieka redakcyjna: Ewelina Burska

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

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cshpk3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z ćwiczeniami i listingami wykorzystanymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/cshpk3.zip

ISBN: 978-83-283-1456-6

Printed in Poland.

Spis treści

Rozdział 3.

Programowanie obiektowe

Każdy program w C# składa się z jednej lub wielu klas. W dotychczas prezentowanych przykładach była to tylko jednak klasa o nazwie `Program`. Przypomnijmy sobie naszą pierwszą aplikację, wyświetlającą na ekranie napis. Jej kod wyglądał następująco:

```
using System;

public class Program
{
  public static void Main()
  {
    Console.WriteLine("Mój pierwszy program!");
  }
}
```

Założyliśmy wtedy, że szkielet kolejnych programów, na których demonstrowano struktury języka programowania, ma właśnie tak wyglądać. Teraz nadszedł czas, aby dowiedzieć się, dlaczego właśnie tak. Wszystko zostanie wyjaśnione w niniejszym rozdziale.

Podstawy

Pierwsza część rozdziału 3. składa się z trzech lekcji, w których podjęto tematykę podstaw programowania obiektowego w C#. W lekcji 14. jest omawiana budowa klas oraz tworzenie obiektów. Zostały w niej przedstawione pola i metody, sposoby ich deklaracji oraz wywoływania. Lekcja 15. jest poświęcona argumentom metod oraz technice przeciążania metod, została w niej również przybliżona wykorzystywana już wcześniej metoda `Main`. W ostatniej, 16. lekcji, zaprezentowano temat konstruktorów, czyli specjalnych metod wywoływanych podczas tworzenia obiektów.

Lekcja 14. Klasy i obiekty

Lekcja 14. rozpoczyna rozdział przedstawiający podstawy programowania obiektowego w C#. Najważniejsze pojęcia zostaną tu wyjaśnione na praktycznych przykładach. Zajmiemy się tworzeniem klas, ich strukturą i deklaracjami, przeanalizujemy związek między klasą i obiektem. Zostaną przedstawione składowe klasy, czyli pola i metody, będzie też wyjaśnione, czym są wartości domyślne pól. Opisane zostaną również relacje między zadeklarowaną na stosie zmienną obiektową (inaczej referencyjną, odnośnikową) a utworzonym na stercie obiektem.

Podstawy obiektowości

Program w C# składa się z klas, które są z kolei opisami obiektów. To podstawowe pojęcia związane z programowaniem obiektowym. Osoby, które nie zetknęły się dotychczas z programowaniem obiektowym, mogą potraktować **obiekt** (ang. *object*) jako pewien byt programistyczny, który może przechowywać dane i wykonywać operacje, czyli różne zadania. **Klasa** (ang. *class*) to z kolei definicja, opis takiego obiektu.

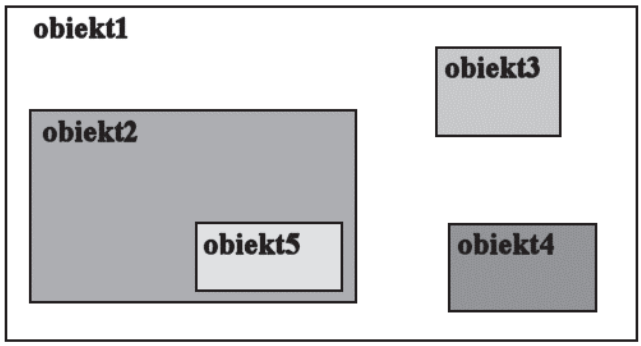
Skoro klasa definiuje obiekt, jest zatem również jego typem. Czym jest **typ obiektu**? Przytoczmy jedną z definicji: „Typ jest przypisany zmiennej, wyrażeniu lub innemu bytowi programistycznemu (danej, obiektowi, funkcji, procedurze, operacji, metodzie, parametrowi, modułowi, wyjątkowi, zdarzeniu). Specyfikuje on rodzaj wartości, które może przybierać ten byt. (…) Jest to również ograniczenie kontekstu, w którym odwołanie do tego bytu może być użyte w programie”[1]. Innymi słowy, typ obiektu określa po prostu, czym jest dany obiekt. Tak samo jak miało to miejsce w przypadku zmiennych typów prostych. Jeśli mieliśmy zmienną typu `int`, to mogła ona przechowywać wartości całkowite. Z obiektami jest podobnie. Zmienna obiektowa hipotetycznej klasy `Punkt` może przechowywać obiekty klasy (typu) `Punkt`[2]. Klasa to zatem nic innego jak definicja nowego typu danych.

Co może być obiektem? Tak naprawdę — wszystko. W życiu codziennym mianem tym określić możemy stół, krzesło, komputer, dom, samochód, radio… Każdy z obiektów ma pewne cechy, właściwości, które go opisują: wielkość, kolor, powierzchnię, wysokość. Co więcej, każdy obiekt może składać się z innych obiektów (rysunek 3.1). Na przykład mieszkanie składa się z poszczególnych pomieszczeń, z których każde może być obiektem; w każdym pomieszczeniu mamy z kolei inne obiekty: sprzęty domowe, meble itd.

Obiekty oprócz tego, że mają właściwości, mogą wykonywać różne funkcje, zadania. Innymi słowy, każdy obiekt ma przypisany pewien zestaw poleceń, które potrafi wykonywać. Na przykład samochód „rozumie” polecenia „uruchom silnik”, „wyłącz silnik”, „skręć w prawo”, „przyspiesz” itp. Funkcje te składają się na pewien interfejs udostępniany nam przez tenże samochód. Dzięki interfejsowi możemy wpływać na zachowanie samochodu i wydawać mu polecenia.

[1] K. Subieta, *Wytwarzanie, integracja i testowanie systemów informatycznych*, PJWSTK, Warszawa 1997.

[2] W dalszej części książki zostanie pokazane, że takiej zmiennej można również przypisać obiekty klas potomnych lub nadrzędnych w stosunku do klasy `Punkt`.



Rysunek 3.1. *Obiekt może zawierać inne obiekty*

W programowaniu jest bardzo podobnie. Za pomocą klas staramy się opisać obiekty, ich właściwości, zbudować konstrukcje, interfejs, dzięki któremu będziemy mogli wydawać polecenia realizowane potem przez obiekty. Obiekt powstaje jednak dopiero w trakcie działania programu jako instancja (wystąpienie, egzemplarz) danej klasy. Obiektów danej klasy może być bardzo dużo. Jeśli na przykład klasą będzie *Samochód*, to instancją tej klasy będzie konkretny egzemplarz o danym numerze seryjnym.

Ponieważ dla osób nieobeznanych z programowaniem obiektowym może to wszystko brzmieć nieco zawile, od razu zobaczmy, jak to będzie wyglądało w praktyce.

Pierwsza klasa

Załóżmy, że pisany przez nas program wymaga przechowywania danych odnoszących się do punktów na płaszczyźnie, ekranie. Każdy taki punkt jest charakteryzowany przez dwie wartości: współrzędną `x` oraz współrzędną `y`. Utwórzmy więc klasę opisującą obiekty tego typu. Schematyczny szkielet klasy wygląda następująco:

```
class nazwa_klasy
{
  //treść klasy
}
```

W treści klasy definiujemy pola i metody. Pola służą do przechowywania danych, metody do wykonywania różnych operacji. W przypadku klasy, która ma przechowywać dane dotyczące współrzędnych `x` i `y`, wystarczą dwa pola typu `int` (przy założeniu, że wystarczające będzie przechowywanie wyłącznie współrzędnych całkowitych). Pozostaje jeszcze wybór nazwy dla takiej klasy. Występują tu takie same ograniczenia jak w przypadku nazewnictwa zmiennych (por. lekcja 5.), czyli nazwa klasy może składać się jedynie z liter (zarówno małych, jak i dużych), cyfr oraz znaku podkreślenia, ale nie może zaczynać się od cyfry. Można stosować znaki polskie znaki (choć wielu programistów używa wyłącznie alfabetu łacińskiego, nawet jeśli nazwy pochodzą z języka polskiego). Przyjęte jest również, że w nazwach nie używa się znaku podkreślenia.

Naszą klasę nazwiemy zatem, jakżeby inaczej, `Punkt` i będzie ona miała postać widoczną na listingu 3.1. Kod ten zapiszemy w pliku o nazwie *Punkt.cs*.

Listing 3.1. *Klasa przechowująca współrzędne punktów*

```
class Punkt
{
  int x;
  int y;
}
```

Ta klasa zawiera dwa pola o nazwach x i y, które opisują współrzędne położenia punktu. Pola definiujemy w taki sam sposób jak zmienne.

Kiedy mamy zdefiniowaną klasę Punkt, możemy zadeklarować zmienną typu Punkt. Robimy to podobnie jak wtedy, gdy deklarowaliśmy zmienne typów prostych (np. short, int, char), czyli pisząc:

```
typ_zmiennej nazwa_zmiennej;
```

Ponieważ typem zmiennej jest nazwa klasy (klasa to definicja typu danych), to jeśli nazwą zmiennej ma być przykladowyPunkt, deklaracja przyjmie postać:

```
Punkt przykladowyPunkt;
```

W ten sposób powstała zmienna odnośnikowa (referencyjna, obiektowa), która domyślnie jest pusta, tzn. nie zawiera żadnych danych. Dokładniej rzecz ujmując, po deklaracji zmienna taka zawiera wartość specjalną null, która określa, że nie ma ona odniesienia do żadnego obiektu. Musimy więc sami utworzyć obiekt klasy Punkt i przypisać go tej zmiennej[3]. Obiekty tworzy się za pomocą operatora new w postaci:

```
new nazwa_klasy();
```

zatem cała konstrukcja schematycznie wyglądać będzie następująco:

```
nazwa_klasy nazwa_zmiennej = new nazwa_klasy();
```

a w przypadku naszej klasy Punkt:

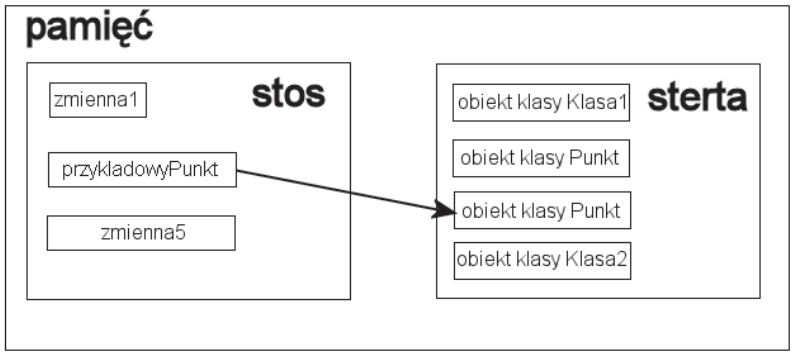
```
Punkt przykladowyPunkt = new Punkt();
```

Oczywiście, podobnie jak w przypadku zmiennych typów prostych (por. lekcja 5.), również i tutaj można oddzielić deklarację zmiennej od jej inicjalizacji, zatem równie poprawna jest konstrukcja w postaci:

```
Punkt przykladowyPunkt;
przykladowyPunkt = new Punkt();
```

Koniecznie trzeba sobie dobrze uzmysłowić, że po wykonaniu tych instrukcji w pamięci powstają dwie różne struktury. Pierwszą z nich jest powstała na tak zwanym *stosie* (ang. *stack*) zmienna referencyjna *przykladowyPunkt*, drugą jest powstały na tak zwanej *stercie* (ang. *heap*) obiekt klasy (typu) Punkt. Zmienna *przykladowyPunkt* zawiera odniesienie do przypisanego jej obiektu klasy Punkt i tylko poprzez nią możemy się do tego obiektu odwoływać. Schematycznie zobrazowano to na rysunku 3.2.

[3] Osoby programujące w C++ powinny zwrócić na to uwagę, gdyż w tym języku już sama deklaracja zmiennej typu klasowego powoduje wywołanie domyślnego konstruktora i utworzenie obiektu.



Rysunek 3.2. *Zależność między zmienną odnośnikową a wskazywanym przez nią obiektem*

Jeśli chcemy odwołać się do danego pola klasy, korzystamy z operatora `.` (kropka), czyli używamy konstrukcji:

```
nazwa_zmiennej_obiektowej.nazwa_pola_obiektu
```

Przykładowo przypisanie wartości `100` polu `x` obiektu klasy `Punkt` reprezentowanego przez zmienną *`przykladowyPunkt`* będzie wyglądało następująco:

```
przykladowyPunkt.x = 100;
```

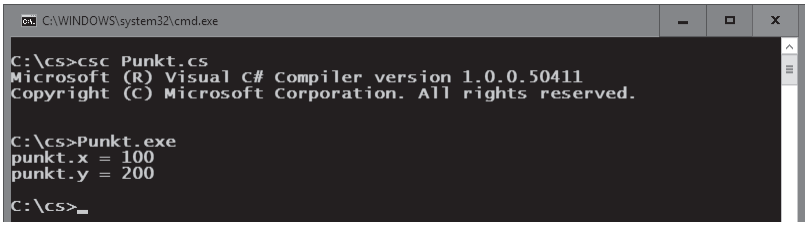
Jak użyć klasy?

Spróbujmy teraz przekonać się, że obiekt klasy `Punkt` faktycznie jest w stanie przechowywać dane. Jak wiadomo z poprzednich rozdziałów, aby program mógł zostać uruchomiony, musi zawierać metodę `Main` (więcej o metodach już w kolejnym podpunkcie, a o metodzie `Main` w jednej z kolejnych lekcji). Dopiszmy więc do klasy `Punkt` taką metodę, która utworzy obiekt, przypisze jego polom pewne wartości oraz wyświetli je na ekranie. Kod programu realizującego takie zadanie jest widoczny na listingu 3.2.

Listing 3.2. *Użycie klasy Punkt*

```
using System;

class Punkt
{
  int x;
  int y;

  public static void Main()
  {
    Punkt punkt1 = new Punkt();
    punkt1.x = 100;
    punkt1.y = 200;
    Console.WriteLine("punkt.x = " + punkt1.x);
    Console.WriteLine("punkt.y = " + punkt1.y);
  }
}
```

Struktura klasy `Punkt` jest taka sama jak w przypadku listingu 3.1, z tą różnicą, że do jej treści została dodana metoda `Main`. W tej metodzie deklarujemy zmienną klasy `Punkt` o nazwie `punkt1` i przypisujemy jej nowo utworzony obiekt tej klasy. Dokonujemy zatem jednoczesnej deklaracji i inicjalizacji. Od tej chwili zmienna `punkt1` wskazuje na obiekt klasy `Punkt`, możemy się zatem posługiwać nią tak, jakbyśmy posługiwali się samym obiektem. Pisząc zatem `punkt1.x = 100`, przypisujemy wartość `100` polu `x`, a pisząc `punkt.y = 200`, przypisujemy wartość `200` polu `y`. W ostatnich dwóch liniach korzystamy z instrukcji `Console.WriteLine`, aby wyświetlić wartość obu pól na ekranie. Efekt jest widoczny na rysunku 3.3.



Rysunek 3.3. *Wynik działania klasy Punkt z listingu 3.2*

Metody klas

Klasy oprócz pól przechowujących dane zawierają także metody, które wykonują zapisane przez programistę operacje. Definiujemy je w ciele (czyli wewnątrz) klasy pomiędzy znakami nawiasu klamrowego. Każda metoda może przyjmować argumenty oraz zwracać wynik. Schematyczna deklaracja metody wygląda następująco:

```
typ_wyniku nazwa_metody(argumenty_metody)
{
  instrukcje metody
}
```

Po umieszczeniu w ciele klasy deklaracja taka będzie natomiast wyglądała tak:

```
class nazwa_klasy
{
  typ_wyniku nazwa_metody(argumenty_metody)
  {
    instrukcje metody
  }
}
```

Jeśli metoda nie zwraca żadnego wyniku, jako typ wyniku należy zastosować słowo `void`; jeśli natomiast nie przyjmuje żadnych parametrów, pomiędzy znakami nawiasu okrągłego nie należy nic wpisywać. Aby zobaczyć, jak to wygląda w praktyce, do klasy `Punkt` dodamy prostą metodę, której zadaniem będzie wyświetlenie wartości współrzędnych `x` i `y` na ekranie. Nadamy jej nazwę `WyswietlWspolrzedne`, zatem jej wygląd będzie następujący:

```
void WyswietlWspolrzedne()
{
  Console.WriteLine("współrzędna x = " + x);
  Console.WriteLine("współrzędna y = " + y);
}
```

Słowo `void` oznacza, że metoda nie zwraca żadnego wyniku, a brak argumentów pomiędzy znakami nawiasu okrągłego wskazuje, że metoda ta żadnych argumentów nie przyjmuje. We wnętrzu metody znajdują się dwie dobrze nam znane instrukcje, które wyświetlają na ekranie współrzędne punktu. Po umieszczeniu powyższego kodu w klasie `Punkt` przyjmie ona postać widoczną na listingu 3.3.

Listing 3.3. *Dodanie metody do klasy Punkt*

```
using System;

class Punkt
{
  int x;
  int y;

  void WyswietlWspolrzedne()
  {
    Console.WriteLine("współrzędna x = " + x);
    Console.WriteLine("współrzędna y = " + y);
  }
}
```

Po utworzeniu obiektu danej klasy możemy **wywołać** (uruchomić) metodę w taki sam sposób, w jaki odwołujemy się do pól klasy, tzn. korzystając z operatora `.` (kropka). Jeśli zatem przykładowa zmienna `punkt1` zawiera referencję do obiektu klasy `Punkt`, prawidłowym wywołaniem metody `WyswietlWspolrzedne` będzie:

```
punkt1.WyswietlWspolrzedne();
```

Ogólnie wywołanie metody wygląda następująco:

```
nazwa_zmiennej.nazwa_metody(argumenty_metody);
```

Oczywiście, jeśli dana metoda nie ma argumentów, po prostu je pomijamy. Przy czym termin *wywołanie* oznacza po prostu wykonanie kodu (instrukcji) zawartego w metodzie.

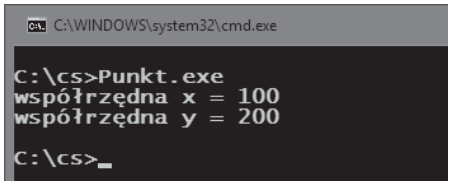
Użyjmy zatem metody `Main` do przetestowania nowej konstrukcji. W tym celu zmodyfikujemy program z listingu 3.2 tak, aby wykorzystywał metodę `WyswietlWspolrzedne`. Odpowiedni kod jest zaprezentowany na listingu 3.4. Wynik jego działania jest łatwy do przewidzenia (rysunek 3.4).

Listing 3.4. *Wywołanie metody WyswietlWspolrzedne*

```
using System;

class Punkt
{
  int x;
  int y;

  void WyswietlWspolrzedne()
  {
    Console.WriteLine("współrzędna x = " + x);
    Console.WriteLine("współrzędna y = " + y);
```

```
  }

  public static void Main()
  {
    Punkt punkt1 = new Punkt();
    punkt1.x = 100;
    punkt1.y = 200;
    punkt1.WyswietlWspolrzedne();
  }
}
```

Rysunek 3.4. *Wynik działania metody WyswietlWspolrzedne z klasy Punkt*



Przedstawiony kod jest w istocie złożeniem przykładów z listingów 3.2 i 3.3. Klasa `Punkt` z listingu 3.3 została uzupełniona o nieco zmodyfikowany kod metody `Main`, pobrany z listingu 3.2. W metodzie tej jest więc tworzony nowy obiekt typu `Punkt` i ustalane są wartości jego pól `x` i `y`. Do wyświetlenia wartości zapisanych w `x` i `y` jest natomiast używana metoda `WyswietlWspolrzedne`.

Zobaczmy teraz, w jaki sposób napisać metody, które będą mogły zwracać wyniki. Typ wyniku należy podać przed nazwą metody, zatem jeśli ma ona zwracać wartość typu `int`, deklaracja powinna wyglądać następująco:

```
int nazwa_metody()
{
  //instrukcje metody
}
```

Sam wynik zwracamy natomiast przez zastosowanie instrukcji `return`. Najlepiej zobaczyć to na praktycznym przykładzie. Do klasy `Punkt` dodamy zatem dwie metody — jedna będzie podawała wartość współrzędnej `x`, druga `y`. Nazwiemy je odpowiednio `PobierzX` i `PobierzY`. Wygląd metody `PobierzX` będzie następujący:

```
int PobierzX()
{
  return x;
}
```

Przed nazwą metody znajduje się określenie typu zwracanego przez nią wyniku — skoro jest to `int`, oznacza to, że metoda ta musi zwrócić jako wynik liczbę całkowitą z przedziału określonego przez typ `int` (tabela 2.1). Wynik jest zwracany dzięki instrukcji `return`. Zapis `return x` oznacza zwrócenie przez metodę wartości zapisanej w polu `x`.

Jak łatwo się domyślić, metoda `PobierzY` będzie wyglądała analogicznie, z tym że będzie w niej zwracana wartość zapisana w polu `y`. Pełny kod klasy `Punkt` po dodaniu tych dwóch metod będzie wyglądał tak, jak przedstawiono na listingu 3.5.

Listing 3.5. *Metody zwracające wyniki*

```
using System;

class Punkt
{
  int x;
  int y;

  int PobierzX()
  {
    return x;
  }

  int PobierzY()
  {
    return y;
  }

  void WyswietlWspolrzedne()
  {
    Console.WriteLine("współrzędna x = " + x);
    Console.WriteLine("współrzędna y = " + y);
  }
}
```

Jeśli teraz zechcemy przekonać się, jak działają nowe metody, możemy wyposażyć klasę `Punkt` w metodę `Main` testującą ich działanie. Mogłaby ona mieć postać widoczną na listingu 3.6.

Listing 3.6. *Metoda Main testująca działanie klasy Punkt*

```
public static void Main()
{
  Punkt punkt1 = new Punkt();
  punkt1.x = 100;
  punkt1.y = 200;
  int wspX = punkt1.PobierzX();
  int wspY = punkt1.PobierzY();
  Console.WriteLine("współrzędna x = " + wspX);
  Console.WriteLine("współrzędna y = " + wspY);
}
```

Początek kodu jest tu taki sam jak we wcześniej prezentowanych przykładach — powstaje obiekt typu `Punkt` i są w nim zapisywane przykładowe współrzędne. Następnie tworzone są dwie zmienne typu `int`: `wspX` i `wspY`. Pierwszej przypisywana jest wartość zwrócona przez metodę `PobierzX`, a drugiej — wartość zwrócona przez metodę `PobierzY`. Wartości zapisane w zmiennych są następnie wyświetlane w standardowy sposób na ekranie.

Warto tu zauważyć, że zmienne `wspX` i `wspY` pełnią funkcję pomocniczą — dzięki nim kod jest czytelniejszy. Nic jednak nie stoi na przeszkodzie, aby wartości zwrócone przez metody były używane bezpośrednio w instrukcjach `Console.WriteLine`[4]. Metoda

[4] Po wyjaśnieniach przedstawionych w lekcji 3. można się domyślić, że to, co do tej pory było nazywane instrukcją `WriteLine`, jest w rzeczywistości wywołaniem metody o nazwie `WriteLine`.

`Main` mogłaby więc mieć również postać przedstawioną na listingu 3.7. Efekt działania byłby taki sam.

Listing 3.7. *Alternatywna wersja metody Main*

```
public static void Main()
{
  Punkt punkt1 = new Punkt();
  punkt1.x = 100;
  punkt1.y = 200;
  Console.WriteLine("współrzędna x = " + punkt1.PobierzX());
  Console.WriteLine("współrzędna y = " + punkt1.PobierzY());
}
```

Jednostki kompilacji, przestrzenie nazw i zestawy

Każdą klasę można zapisać w pliku o dowolnej nazwie. Często przyjmuje się jednak, że nazwa pliku powinna być zgodna z nazwą klasy. Jeśli zatem istnieje klasa `Punkt`, to jej kod powinien znaleźć się w pliku *Punkt.cs*. W jednym pliku może się też znaleźć kilka klas. Wówczas jednak zazwyczaj są to tylko jedna klasa główna oraz dodatkowe klasy pomocnicze. W przypadku prostych aplikacji tych zasad nie trzeba przestrzegać, ale w przypadku większych programów umieszczenie całej struktury kodu w jednym pliku spowodowałoby duże trudności w zarządzaniu nim. Pojedynczy plik można nazwać jednostką kompilacji lub modułem.

Wszystkie dotychczasowe przykłady składały się zawsze z jednej klasy zapisywanej w jednym pliku. Zobaczmy więc, jak mogą współpracować ze sobą dwie klasy. Na listingu 3.8 znajduje się nieco zmodyfikowana treść klasy `Punkt` z listingu 3.1. Przed składowymi zostały dodane słowa `public`, dzięki którym będzie istniała możliwość odwoływania się do nich z innych klas. Ta kwestia zostanie wyjaśniona dokładniej w jednej z kolejnych lekcji. Na listingu 3.9 jest natomiast widoczny kod klasy `Program`, która korzysta z klasy `Punkt`. Tak więc treść z listingu 3.8 zapiszemy w pliku o nazwie *Punkt.cs*, a kod z listingu 3.9 w pliku *Program.cs*.

Listing 3.8. *Prosta klasa Punkt*

```
class Punkt
{
  public int x;
  public int y;
}
```

Listing 3.9. *Klasa Program korzystająca z obiektu klasy Punkt*

```
using System;

public class Program
{
  public static void Main()
  {
    Punkt punkt1 = new Punkt();
```

```
      punkt1.x = 100;
      punkt1.y = 200;
      Console.WriteLine("punkt1.x = " + punkt1.x);
      Console.WriteLine("punkt1.y = " + punkt1.y);
    }
  }
```

W klasie `Program` znajduje się metoda `Main`, od której rozpoczyna się wykonywanie kodu aplikacji. W tej metodzie tworzony jest obiekt `punkt1` klasy `Punkt`, jego składowym przypisywane są wartości `100` i `200`, a następnie są one wyświetlane na ekranie. Tego typu konstrukcje były wykorzystywane już kilkakrotnie we wcześniejszych przykładach.

Jak teraz przetworzyć oba kody na plik wykonywalny? Nie jest to skomplikowane, po prostu nazwy obu plików (`Program.cs i Punkt.cs`) należy zastosować jako argumenty wywołania kompilatora, czyli w wierszu poleceń wydać komendę:

```
csc Program.cs Punkt.cs
```

Trzeba też wiedzieć, że plik wykonywalny powstały po kompilacji nie zawiera tylko kodu wykonywalnego. W rzeczywistości kod wykonywany na platformie .NET składa się z tak zwanych zestawów (ang. *assembly*). Pojedynczy **zestaw** składa się z manifestu, metadanych oraz kodu języka pośredniego IL. **Manifest** to wszelkie informacje o zestawie, takie jak nazwy plików składowych, odwołania do innych zestawów, numer wersji itp. **Metadane** natomiast to opis danych i kodu języka pośredniego w danym zestawie, zawierający m.in. definicje zastosowanych typów danych.

Wszystko to może być umieszczone w jednym lub też kilku plikach (*exe*, *dll*). We wszystkich przykładach w tej książce będziemy mieli do czynienia tylko z zestawami jednoplikowymi i będą to pliki wykonywalne typu *exe*, generowane automatycznie przez kompilator, tak że nie będziemy musieli zagłębiać się w te kwestie. Nie można jednak pominąć zagadnienia przestrzeni nazw.

Przestrzeń nazw to ograniczenie widoczności danej nazwy, ograniczenie kontekstu, w którym jest ona rozpoznawana. Czemu to służy? Otóż pojedyncza aplikacja może się składać z bardzo dużej liczby klas, a jeszcze więcej klas znajduje się w bibliotekach udostępnianych przez .NET. Co więcej, nad jednym projektem zwykle pracują zespoły programistów. W takiej sytuacji nietrudno o pojawianie się konfliktów nazw, czyli powstawanie klas o takich samych nazwach. Tymczasem nazwa każdej klasy musi być unikatowa. Ten problem rozwiązują właśnie przestrzenie nazw. Jeśli bowiem klasa zostanie umieszczona w danej przestrzeni, to będzie widoczna tylko w niej. Będą więc mogły istnieć klasy o takiej samej nazwie, o ile tylko zostaną umieszczone w różnych przestrzeniach nazw. Taką przestrzeń definiuje się za pomocą słowa `namespace`, a jej składowe należy umieścić w występującym dalej nawiasie klamrowym. Schematycznie wygląda to tak:

```
namespace nazwa_przestrzeni
{
  elementy przestrzeni nazw
}
```

Skorowidz

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

PRAKTYCZNY KURS

C# należy do rodziny obiektowych języków programowania. Co to oznacza? Otóż znajduje się on w doborowym towarzystwie Javy, C++ i wielu innych języków, za pomocą których można tworzyć wspaniałe aplikacje, ale jednocześnie jest od nich odrobinę łatwiejszy. Jeśli dopiero zaczynasz przygodę z programowaniem, szybciej uda Ci się go opanować, a jeżeli jest to dla Ciebie któryś kolejny język, zrozumienie jego składni i reguł nie powinno Ci sprawić większych trudności — szczególnie jeśli do nauki wykorzystasz tę książkę.

Ten znakomity, praktyczny podręcznik pozwoli Ci przećwiczyć używanie i sposób działania wszystkich elementów C# — różnych typów danych, zmiennych i operatorów, instrukcji i tablic. Zobaczysz, jak korzystać z pętli i jak zachowują się obiekty. Poznasz najróżniejsze rodzaje klas, opanujesz wygodne sposoby stosowania dziedziczenia i nauczysz się obsługiwać błędy. W dalszej części książki znajdziesz zaawansowane zagadnienia programowania obiektowego i odkryjesz, jak projektować aplikacje z interfejsem graficznym. Krótko mówiąc, po starannym wykonaniu ćwiczeń będziesz w stanie zaprojektować i zbudować własną aplikację z użyciem języka C#!

- Typy danych, zmienne i instrukcje sterujące
- Tablice i pętle
- Klasy i obiekty
- Dziedziczenie
- Obsługa błędów i wyjątków
- System wejścia–wyjścia
- Polimorfizm i interfejsy
- Klasy zagnieżdżone i typy uogólnione
- Aplikacje z interfejsem graficznym

Stwórz własną aplikację w języku C#!

Helion

41171 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

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

ISBN 978-83-283-1456-6

9 788328 314566

Informatyka w najlepszym wydaniu

cena: 59,00 zł