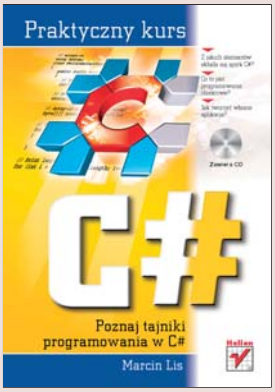
C#. Praktyczny kurs

Autor: Marcin Lis
ISBN: 978-83-246-0818-8
Format: B5, stron: 376
Przykłady na ftp: 199 kB



Poznaj tajniki programowania w C#

- Z jakich elementów składa się język C#?
- Co to jest programowanie obiektowe?
- Jak tworzyć własne aplikacje?

Chcesz poznać jeden z najpopularniejszych języków programowania? A może jesteś już programistą i chcesz poszerzyć swoje kwalifikacje? Zamierzasz wykorzystywać ogromne możliwości platformy .NET? Zainteresuj się językiem C# – połączeniem najlepszych cech Javy i C++, stanowiącym dziś podstawową technologię programistyczną środowiska .NET. C# to obiektowy język programowania cechujący się ogromną wszechstronnością – za jego pomocą można stworzyć aplikacje zarówno dla „dużych" komputerów, jak i urządzeń mobilnych typu Pocket PC. Można również wykorzystać go do budowania aplikacji internetowych w technologii ASP.NET.

„C#. Praktyczny kurs" to wprowadzenie do programowania w tym języku. Dzięki tej książce poznasz podstawowe zagadnienia związane z tworzeniem aplikacji dla systemu Windows z wykorzystaniem języka C# i platformy .NET. Dowiesz się, z jakich elementów składa się C# i na czym polega programowanie obiektowe. Nauczysz się implementować w swoich programach mechanizmy obsługi wyjątków i bez trudu tworzyć aplikacje z graficznym interfejsem użytkownika.

- Typy danych, zmienne i stałe
- Sterowanie przebiegiem programu
- Operacje na tablicach
- Klasy i obiekty
- Praca z systemem plików
- Obsługa wyjątków
- Tworzenie aplikacji z interfejsem graficznym

Zostań profesjonalnym programistą C#

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 3.

Programowanie obiektowe

Każdy program w C# składa się z jednej lub wielu klas. W naszych dotychczasowych 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 naszych kolejnych programów, na których uczyliśmy się struktur języka programowania, ma właśnie tak wyglądać. Teraz nadszedł czas, aby wyjaśnić, dlaczego właśnie tak. Wszystko stanie się jasne po przeczytaniu niniejszego rozdziału.

Podstawy

Pierwsza część rozdziału 3. składa się z trzech lekcji, które przedstawiają podstawy 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`. Ostatnia, 16. lekcja, prezentuje temat konstruktorów, czyli specjalnych metod wywoływanych podczas tworzenia obiektów.

Lekcja 14. Klasy i obiekty

Lekcja 14. rozpoczyna rozdział przedstawiający podstawy programowania obiektowego w C#. Podstawowe pojęcia zostaną tu wyjaśnione na praktycznych przykładach. Zajmiemy się tworzeniem klas, ich strukturą i deklaracjami, poznamy związek między klasą i obiektem. Zostaną przedstawione składowe klasy, czyli pola i metody, dowiemy się też, czym są wartości domyślne pól. Poznamy 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** jako pewien byt programistyczny, który może przechowywać dane i wykonywać operacje, czyli różne zadania. **Klasa** to z kolei definicja, opis takiego obiektu.

Skoro klasa definiuje obiekt, jest zatem również jego typem. Czym jest **typ obiektu**? Zapoznajmy się z 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.

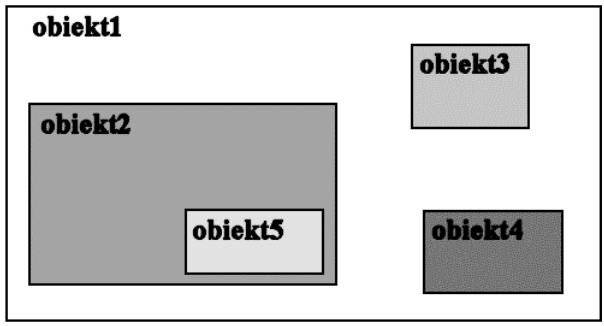
Co może być obiektem? Tak naprawdę — wszystko. W życiu realnym mianem tym określić możemy stół, krzesło, komputer, dom, samochód, radio... Każdy z obiektów ma pewne cechy, właściwości, które go opisują: wielkość, kolor, powierzchnię, wysokość. Co więcej, każdy obiekt może składać się z innych obiektów (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”, „zgaś 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] Jak dowiemy się w dalszej części książki, 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. Przykładowo: jeśli 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, zobaczmy, jak to będzie wyglądało w praktyce.

Pierwsza klasa

Załóżmy, że pisany przez nas program wymaga przechowywania danych odnośnie do punktów na płaszczyźnie. Każdy taki punkt jest charakteryzowany przez dwie wartości: współrzędną `x` oraz współrzędną `y`. Stwórzmy 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 więc dwa pola typu `int`[3]. Pozostaje nam jeszcze wybór nazwy dla takiej klasy. Występują tu takie same ograniczenia, jak w przypadku nazewnictwa zmiennych (patrz 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. Choć jest to możliwe, raczej nie stosuje polskich znaków diakrytycznych. Przyjęte jest również, że w nazwach nie używa się znaku pokreślenia.

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*.

[3] Tak więc punkty będą mogły przyjmować wyłącznie współrzędne całkowite.

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 deklarowaliśmy zmienne typów prostych (np. `short`, `int`, `char`), czyli pisząc:

```
typ_zmiennej nazwa_zmiennej;
```

Ponieważ typem zmiennej jest nazwa klasy, to jeśli jej nazwą 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 zawiera ona odniesienia do żadnego obiektu. Musimy więc sami utworzyć obiekt klasy `Punkt` i przypisać go tej zmiennej[4]. Ta czynność wykonywana jest za pomocą operatora `new` w postaci:

```
new nazwa_klasy();
```

zatem cała konstrukcja schematycznie wyglądać będzie następująco:

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

w przypadku naszej klasy `Punkt`:

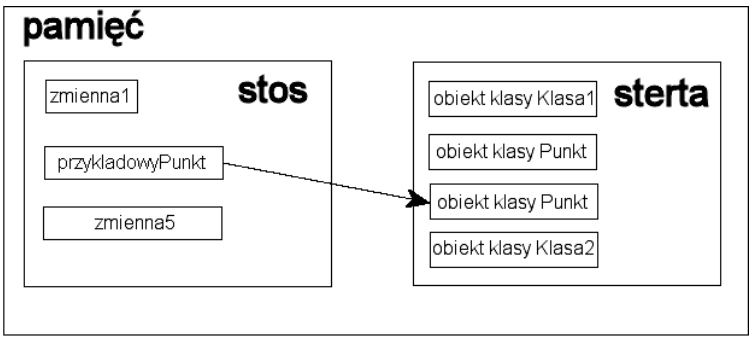
```
Punkt przykladowyPunkt = new Punkt();
```

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();
```

Powinniśmy 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 `Punkt`. Zmienna *`przykladowyPunkt`* zawiera odniesienie do przypisanego jej obiektu klasy `Punkt` i tylko poprzez nią możemy się do tego obiektu odwoływać. Schematycznie obrazuje to rysunek 3.2.

[4] 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 polu `x` obiektu klasy `Punkt` reprezentowanego przez zmienną *`przykladowyPunkt`* wartości `100` będzie wyglądało następująco:

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

Jak użyć klasy?

Spróbujmy teraz przekonać się, że obiekt klasy `Punkt` faktycznie jest w stanie przechowywać dane. Jak pamiętamy 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*

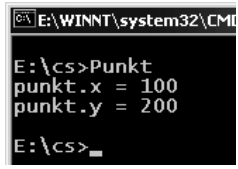
```
using System;

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. W ciele metody znajdują się dwie dobrze nam znane instrukcje, które wyświetlają na ekranie współrzędne punktu. Po umieszczeniu powyższego kodu wewnątrz klasy `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 metodę w sposób identyczny, 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. Wykorzystajmy teraz metodę `Main` do przetestowania nowej konstrukcji. 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).

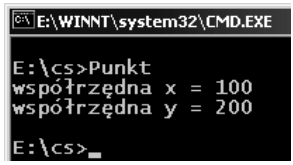
Listing 3.4. *Wywołanie metody WyswietlWspolrzedne*

```
using System;

class Punkt
{
  int x;
  int y;

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

Rysunek 3.4. *Wynik działania metody WyswietlWspolrzedne klasy Punkt*




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

Zobaczmy teraz, w jaki sposób napisać metody, które będą umiały zwracać wyniki. Typ wyniku należy podać przed nazwą metody, zatem jeśli ma ona zwracać liczbę 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` (por. 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 to zostało przedstawione 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);
  }
}
```

Gdybyśmy teraz chcieli 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;
  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 jest 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 jego zarządzaniu. Pojedynczy plik możemy 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 3.8 jest natomiast widoczny kod klasy `Program`, która korzysta z klasy `Punkt`. Tak więc treść z listingu 3.7 zapisujemy w pliku o nazwie *Punkt.cs*, a kod z listingu 3.8 w pliku *Program.cs*.

Listing 3.7. *Prosta klasa Punkt*

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

Listing 3.8. *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 `punk1` klasy `Punkt`, jego składowym przypisywane są wartości 100 i 200, a następnie są one wyświetlane na ekranie. Tego typu konstrukcje były wykorzystywane już kilkukrotnie 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 użyć jako argumentów wywołania kompilatora, czyli w wierszu poleceń wydać komendę:

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

Musimy też wiedzieć, że plik wykonywalny powstały po kompilacji nie składa się wyłącznie z kodu wykonywalnego. W rzeczywistości kod wykonywany na platformie .NET składa się z tak zwanych zestawów (ang. *assembly*). Pojedynczy **zestaw** składa się z tak zwanego 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** to natomiast opis danych i kodu języka pośredniego w danym zestawie, m.in. zawierają 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 zgłębiać się w te kwestie. Nie możemy jednak pominąć zagadnienia przestrzeni nazw.

Otóż 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 powstawanie konfliktów nazw, czyli powstawania 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 za pomocą słowa `namespace`, a jej składowe należy umieścić w występującym dalej nawiasie klamrowym. Schematycznie wygląda to tak:

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

Przykładowo jeden programista może pracować nad biblioteką klas dotyczących grafiki trójwymiarowej, a drugi nad bibliotekę klas wspomagających tworzenie grafiki na płaszczyźnie. Każdemu z nich możemy przydzielić osobne przestrzenie nazw, np. o nazwach `Grafika2D` i `Grafika3D`. W takiej sytuacji każdy z nich będzie mógł utworzyć własną klasę o nazwie `Punkt` i obie te klasy będzie można jednocześnie wykorzystać w jednej aplikacji. Klasy te mogłyby mieć definicje takie jak na listingach 3.9 i 3.10.

Listing 3.9. *Klasa Punkt w przestrzeni nazw Grafika2D*

```
namespace Grafika2D
{
  class Punkt
  {
    public int x;
    public int y;
  }
}
```

Listing 3.10. *Klasa Punkt w przestrzeni nazw Grafika3D*

```
namespace Grafika3D
{
  class Punkt
  {
    public double x;
    public double y;
  }
}
```

Jak skorzystać z jednej z tych klas w jakimś programie? Mamy dwie możliwości. Pierwsza z nich to podanie pełnej nazwy klasy wraz z nazwą przestrzeni nazw. Pomiędzy nazwą klasy a nazwą przestrzeni należy umieścić znak kropki. Przykładowo odwołanie do klasy `Punkt` z przestrzeni `Grafika2D` miałoby postać:

```
Grafika2D.Punkt
```

Sposób ten został przedstawiony na listingu 3.11.

Listing 3.11. *Użycie przestrzeni nazw Grafika2D*

```
using System;

public class Program
{
  public static void Main()
  {
    Grafika2D.Punkt punkt1 = new Grafika2D.Punkt();
    punkt1.x = 100;
    punkt1.y = 200;
    Console.WriteLine("punkt1.x = " + punkt1.x);
    Console.WriteLine("punkt1.y = " + punkt1.y);
  }
}
```

Drugi sposób to użycie dyrektywy `using`, w postaci:

```
using nazwa_przestrzeni;
```

Należy ją umieścić na samym początku pliku. Oznacza ona nic innego, jak informację dla kompilatora, że chcemy korzystać z klas zdefiniowanych w przestrzeni o nazwie *nazwa_przestrzeni*. Liczba umieszczonych na początku pliku instrukcji `using` nie jest ograniczona. Jeśli chcemy skorzystać w kilku przestrzeni nazw, używamy kilku dyrektyw `using`. Jasne jest więc już, co oznacza zwrot:

```
using System;
```

wykorzystywany w praktycznie wszystkich dotychczasowych przykładach. To deklaracja, że chcemy korzystać z przestrzeni nazw o nazwie `System`. Była nam ona niezbędna, gdyż w tej właśnie przestrzeni jest umieszczona klasa `Console` zawierająca metody `Write` i `WriteLine`. Domyślamy się też zapewne, że moglibyśmy pominąć dyrektywę `using System`, ale wtedy instrukcja wyświetlająca wiersz tekstu na ekranie musiałby przyjmować postać:

```
System.Console.WriteLine("tekst");
```

Tak więc nasz pierwszy program z listingu 1.1 równie dobrze mógłby mieć postać widoczną na listingu 3.12.

Listing 3.12. *Pominięcie dyrektywy using System*

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

Nie będzie także zaskoczeniem, że gdybyśmy chcieli, aby w programie z listingu 3.11 nie trzeba było odwoływać się przestrzeni nazw `Grafika2D` przy każdym wystąpieniu klasy `Punkt`, to należałoby użyć instrukcji `using Grafika2D`, tak jak zostało to zaprezentowane na listingu 3.13.

Listing 3.13. *Użycie instrukcji using Grafika2D*

```
using System;
using Grafika2D;

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);
  }
}
```

Pozostaje jeszcze kwestia jednoczesnego użycia klas `Punkt` z przestrzeni `Grafika2D` i `Grafika3D`. Można oczywiście użyć dwóch następujących po sobie instrukcji `using`:

```
using Grafika2D;
using Grafika3D;
```

W żaden sposób nie rozwiąże to jednak problemu. Jak bowiem kompilator (ale także i programista) miałby wtedy odróżnić, o którą z klas chodzi, kiedy nazywają się one tak samo. Dlatego też w takim wypadku za każdym razem trzeba w odwołaniu podawać, o którą przestrzeń nazw chodzi. Stąd, jeśli chcemy zdefiniować dwa obiekty: `punkt1` klasy `Punkt` z przestrzeni `Grafika2D` i `punkt2` klasy `Punkt` z przestrzeni `Grafika3D`, należy użyć instrukcji:

```
Grafika2D.Punkt punkt1 = new Grafika2D.Punkt();
Grafika3D.Punkt punkt2 = new Grafika3D.Punkt();
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 14.1

Napisz przykładową klasę `LiczbaCalkowita`, która będzie przechowywała wartość całkowitą. Klasa ta powinna zawierać metodę `WyswietlLiczbe`, która będzie wyświetlała na ekranie przechowywaną wartość, oraz metodę `PobierzLiczbe` zwracającą przechowywaną wartość.

Ćwiczenie 14.2

Napisz kod przykładowej klasy `Prostokat` zawierającej cztery pola przechowujące współrzędne czterech rogów prostokąta.

Ćwiczenie 14.3

Do utworzonej w ćwiczeniu 14.2 klasy `Prostokat` dopisz metody zwracające współrzędne wszystkich czterech rogów oraz metodę wyświetlającą wartości współrzędnych.

Ćwiczenie 14.4

Do klas `LiczbaCalkowita` i `Prostokat` dopisz przykładową metodę `Main` testującą ich zachowanie.

Ćwiczenie 14.5

Napisz klasę `Protokat` przechowującą jedynie współrzędne lewego górnego i prawego dolnego rogu (wystarczają one do jednoznacznego wyznaczenia prostokąta na płaszczyźnie). Dodaj metody podające współrzędne każdego rogu.

Ćwiczenie 14.6

Do klasy `Prostokat` z ćwiczeń 14.2 i 14.3 dopisz metodę sprawdzającą, czy wprowadzone współrzędne faktycznie definiują prostokąt (cztery punkty na płaszczyźnie dają dowolny czworokąt, który nie musi mieć kształtów prostokąta).

Lekcja 15. Argumenty i przeciążanie metod

O tym, że metody mogą mieć argumenty, wiemy z lekcji 14. Czas dowiedzieć się, jak się nimi posługiwać. Właśnie temu tematowi została poświęcona cała lekcja 15. Dowiemy się, w jaki sposób przekazuje się metodom argumenty typów prostych oraz referencyjnych, poznamy też przeciążanie metod, czyli technikę umożliwiającą umieszczenie w jednej klasie kilku metod o tej samej nazwie. Nieco miejsca poświęcimy także metodzie `Main`, od której zaczyna się wykonywanie aplikacji. Zobaczymy również, w jaki sposób przekazać aplikacji parametry z wiersza poleceń.

Argumenty metod

W lekcji 14. dowiedzieliśmy się, że każda metoda może mieć argumenty. **Argumenty metody** to inaczej dane, które można jej przekazać. Metoda może mieć dowolną liczbę argumentów umieszczonych w nawiasie okrągłym za jej nazwą. Poszczególne argumenty oddzielamy od siebie znakiem przecinka. Schematycznie wygląda to następująco:

```
typ_wyniku nazwa_metody(typ_argumentu_1 nazwa_argumentu_1, typ_argumentu_2
nazwa_argumentu_2, ... , typ_argumentu_N nazwa_argumentu_N)
{
  /* treść metody */
}
```

Przykładowo w klasie `Punkt` przydałyby się metody umożliwiające ustawianie współrzędnych. Jest tu możliwych kilka wariantów — zacznijmy od najprostszych: napiszemy dwie metody, `UstawX` i `UstawY`. Pierwsza będzie odpowiedzialna za przypisanie przekaza-

nej jej wartości polu `x`, a druga — polu `y`. Zgodnie z podanym powyżej schematem pierwsza z nich powinna wyglądać następująco:

```
void UstawX(int wspX)
{
  x = wspX;
}
```

natomiast druga:

```
void UstawY(int wspY)
{
  y = wspY;
}
```

Metody te nie zwracają żadnych wyników, co sygnalizuje słowo `void`, przyjmują natomiast jeden parametr typu `int`. W ciele każdej z metod następuje z kolei przypisanie wartości przekazanej w parametrze odpowiedniemu polu: `x` w przypadku metody `UstawX` oraz `y` w przypadku metody `UstawY`.

W podobny sposób możemy napisać metodę, która będzie jednocześnie ustawiała pola `x` i `y` klasy `Punkt`. Oczywiście będzie ona przyjmowała dwa argumenty. Jak wiemy, w deklaracji należy oddzielić je przecinkiem, zatem będzie ona wyglądała następująco:

```
void UstawXY(int wspX, int wspY)
{
  x = wspX;
  y = wspY;
}
```

Metoda `UstawXY` nie zwraca żadnego wyniku, ale przyjmuje dwa argumenty: `wspX`, `wspY`, oba typu `int`. W ciele tej metody argument `wspX` (dokładniej — jego wartość) zostaje przypisany polu `x`, a `wspY` — polu `y`. Jeśli teraz dodamy do klasy `Punkt` wszystkie trzy powstałe wyżej metody, otrzymamy kod widoczny na listingu 3.14.

Listing 3.14. *Metody ustawiające pola klasy Punkt*

```
using System;

class Punkt
{
  int x;
  int y;

  int PobierzX()
  {
    return x;
  }
  int PobierzY()
  {
    return y;
  }
  void UstawX(int wspX)
  {
    x = wspX;
```

```
    }
    void UstawY(int wspY)
    {
      y = wspY;
    }
    void UstawXY(int wspX, int wspY)
    {
      x = wspX;
      y = wspY;
    }
    void WyswietlWspolrzedne()
    {
      Console.WriteLine("współrzędna x = " + x);
      Console.WriteLine("współrzędna y = " + y);
    }
  }
```

Warto teraz napisać dodatkową metodę `Main`, która przetestuje nowe metody. Dzięki temu będziemy mogli sprawdzić, czy wszystkie trzy działają zgodnie z naszymi założeniami. Taka przykładowa metoda jest widoczna na listingu 3.15[5].

Listing 3.15. *Metoda Main testująca metody ustawiające współrzędne*

```
  public static void Main()
  {
    Punkt pierwszyPunkt = new Punkt();
    Punkt drugiPunkt = new Punkt();

    pierwszyPunkt.UstawX(100);
    pierwszyPunkt.UstawY(100);

    Console.WriteLine("pierwszyPunkt:");
    pierwszyPunkt.WyswietlWspolrzedne();

    drugiPunkt.UstawXY(200, 200);

    Console.WriteLine("\ndrugiPunkt:");
    drugiPunkt.WyswietlWspolrzedne();
  }
```

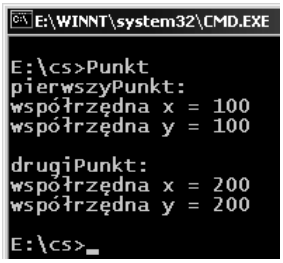
Na początku tworzymy dwa obiekty typu (klasy) `Punkt`, jeden z nich przypisujemy zmiennej o nazwie `pierwszyPunkt`, drugi zmiennej o nazwie `drugiPunkt`[6]. Następnie wykorzystujemy metody `UstawX` i `UstawY` do przypisania polom obiektu `pierwszyPunkt` wartości `100`. W kolejnym kroku za pomocą metody `WyswietlWspolrzedne` wyświetlamy te wartości na ekranie. Dalej wykorzystujemy metodę `UstawXY`, aby przypisać polom

[5] W listingach na umieszczonych się na płycie CD znajduje się pełny kod klasy `Punkt` zawierający widoczną metodę `Main`.

[6] W rzeczywistości zmiennym zostały przypisane referencje (odniesienia) do utworzonych na stercie obiektów. Można jednak stosować przestawioną tu uproszczoną terminologię, w której referencję utożsamia się z obiektem.

obiektu drugiPunkt wartości 200, oraz wyświetlamy je na ekranie, również za pomocą metody WyswietlWspolrzedne. Po wykonaniu tego programu otrzymamy widok jak na rysunku 3.5.

Rysunek 3.5. *Efekt wykonania programu z listingu 3.15*



Obiekt jako argument

Argumentem przekazanym metodzie może być również obiekt (ściślej: referencja do obiektu), nie musimy ograniczać się jedynie do typów prostych. Podobnie metoda może zwracać obiekt w wyniku swojego działania. W obu wymienionych sytuacjach postępowanie jest identyczne, jak w przypadku typów prostych. Przykładowo metoda UstawXY w klasie Punkt mogłaby przyjmować jako argument obiekt tej klasy, a nie dwie liczby typu int, tak jak zaprogramowaliśmy to we wcześniejszych przykładach (listing 3.14). Metoda taka wyglądałaby następująco:

```
void UstawXY(Punkt punkt)
{
  x = punkt.x;
  y = punkt.y;
}
```

Argumentem jest w tej chwili obiekt punkt klasy Punkt. W ciele metody następuje skopiowanie wartości pól z obiektu przekazanego jako argument do obiektu bieżącego, czyli przypisanie polu x wartości zapisanej w punkt.x, a polu y wartości zapisanej w punkt.y.

Podobnie możemy umieścić w klasie Punkt metodę o nazwie PobierzXY, która zwróci w wyniku nowy obiekt klasy Punkt o współrzędnych takich, jakie zostały zapisane w polach obiektu bieżącego. Metoda taka będzie miała postać:

```
Punkt PobierzXY()
{
  Punkt punkt = new Punkt();
  punkt.x = x;
  punkt.y = y;
  return punkt;
}
```

Jak widzimy, nie przyjmuje ona żadnych argumentów, nie ma przecież takiej potrzeby; z deklaracji wynika jednak, że zwraca ona obiekt klasy Punkt. W ciele metody najpierw tworzymy nowy obiekt klasy Punkt, przypisując go zmiennej referencyjnej o nazwie punkt, a następnie przypisujemy jego polom wartości pól x i y z obiektu bieżącego.

Ostatecznie za pomocą instrukcji `return` powodujemy, że obiekt `punkt` staje się wartością zwracaną przez metodę. Klasa `Punkt` po wprowadzeniu takich modyfikacji będzie miała postać widoczną na listingu 3.16.

Listing 3.16. *Nowe metody klasy Punkt*

```
using System;

class Punkt
{
  int x;
  int y;

  int PobierzX()
  {
    return x;
  }
  int PobierzY()
  {
    return y;
  }
  void UstawX(int wspX)
  {
    x = wspX;
  }
  void UstawY(int wspY)
  {
    y = wspY;
  }
  void UstawXY(Punkt punkt)
  {
    x = punkt.x;
    y = punkt.y;
  }
  Punkt PobierzXY()
  {
    Punkt punkt = new Punkt();
    punkt.x = x;
    punkt.y = y;
    return punkt;
  }
  void WyswietlWspolrzedne()
  {
    Console.WriteLine("współrzędna x = " + x);
    Console.WriteLine("współrzędna y = " + y);
  }
}
```

Aby lepiej uzmysłowić sobie sposób działania wymienionych metod, napiszemy teraz kod metody `Main`, który będzie je wykorzystywał. Należy go dodać do klasy naszej najnowszej wersji klasy `Punkt`. Kod ten został zaprezentowany na listingu 3.17.

Listing 3.17. *Kod metody Main*

```
public static void Main()
{
  Punkt pierwszyPunkt = new Punkt();
  Punkt drugiPunkt;

  pierwszyPunkt.UstawX(100);
  pierwszyPunkt.UstawY(100);

  Console.WriteLine("Obiekt pierwszyPunkt ma współrzędne:");
  pierwszyPunkt.WyswietlWspolrzedne();
  Console.Write("\n");

  drugiPunkt = pierwszyPunkt.PobierzXY();

  Console.WriteLine("Obiekt drugiPunkt ma współrzędne:");
  drugiPunkt.WyswietlWspolrzedne();
  Console.Write("\n");

  Punkt trzeciPunkt = new Punkt();
  trzeciPunkt.UstawXY(drugiPunkt);

  Console.WriteLine("Obiekt trzeciPunkt ma współrzędne:");
  trzeciPunkt.WyswietlWspolrzedne();
  Console.Write("\n");
}
```

Na początku deklarujemy zmienne `pierwszyPunkt` oraz `drugiPunkt`. Zmiennej `pierwszyPunkt` przypisujemy nowo utworzony obiekt klasy `Punkt` (rysunek 3.7 a). Następnie wykorzystujemy znane nam dobrze metody `UstawX` i `UstawY` do przypisania polom `x` i `y` wartości `100` oraz wyświetlamy te dane na ekranie, korzystając z metody `wyswietlWspolrzedne`.

W kolejnym kroku zmiennej `drugiPunkt`, która, jak pamiętamy, nie została wcześniej zainicjowana, przypisujemy obiekt zwrócony przez metodę `PobierzWspolrzedne` wywołaną na rzecz obiektu `pierwszyPunkt`. A zatem zapis:

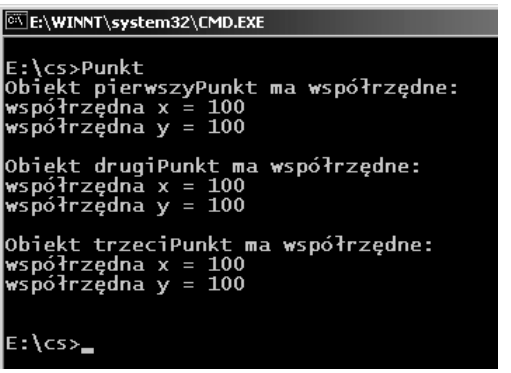
```
drugiPunkt = pierwszyPunkt.PobierzWspolrzedne();
```

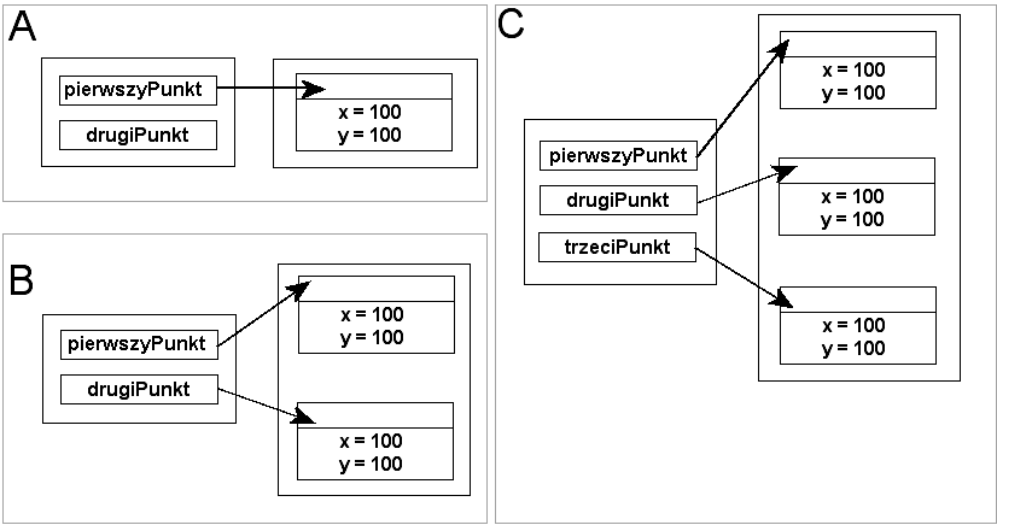
oznacza, że wywoływana jest metoda `PobierzWspolrzedne` obiektu `punkt`, a zwrócony przez nią wynik jest przypisywany zmiennej `drugiPunkt`. Jak wiemy, wynikiem działania tej metody będzie obiekt klasy `Punkt` będący kopią obiektu `pierwszyPunkt`, czyli zawierający w polach `x` i `y` takie same wartości, jakie są zapisane w polach obiektu `pierwszyPunkt`. To znaczy, że po wykonaniu tej instrukcji zmienna `drugiPunkt` zawiera referencję do obiektu, w którym pola `x` i `y` mają wartość `100` (rysunek 3.7 b). Obie wartości wyświetlamy na ekranie za pomocą instrukcji `WyswietlWspolrzedne`.

W trzeciej części programu tworzymy obiekt `trzeciPunkt` (`Punkt trzeciPunkt = new Punkt();`) i wywołujemy jego metodę `ustawXY`, aby wypełnić pola `x` i `y` danymi. Metoda ta jako parametr przyjmuje obiekt klasy `Punkt`, w tym przypadku obiekt `drugiPunkt`. Zatem po wykonaniu instrukcji wartości pól `x` i `y` obiektu `trzeciPunkt` będą takie same,

jak pól `x` i `y` obiektu `drugiPunkt` (rysunek 3.7 c). Nic zatem dziwnego, że wynik działania programu z listingu 3.17 jest taki, jak zaprezentowany na rysunku 3.6. Z kolei rysunek 3.7 przedstawia schematyczne zależności pomiędzy zmiennymi i obiektami występującymi w metodzie `Main`.

Rysunek 3.6. *Utworzenie trzech takich samych obiektów różnymi metodami*





Rysunek 3.7. *Kolejne etapy powstawania zmiennych i obiektów w programie z listingu 3.17*

W fazie pierwszej, na samym początku programu, mamy jedynie dwie zmienne: `pierwszyPunkt` i `drugiPunkt`. Tylko pierwszej z nich jest przypisany obiekt, druga jest po prostu pusta (zawiera wartość `null`). Przedstawia to rysunek 3.7 a. W części drugiej przypisujemy zmiennej `drugiPunkt` obiekt, który jest kopią obiektu `pierwszyPunkt` (rysunek 3.7 b), a w trzeciej tworzymy obiekt `trzeciPunkt` i wypełniamy go danymi pochodzącymi z obiektu `drugiPunkt`. Tym samym ostatecznie otrzymujemy trzy zmienne i trzy obiekty (rysunek 3.7 c).

Przeciążanie metod

W trakcie naszej pracy nad kodem klasy `Punkt` powstały dwie metody o takiej samej nazwie, ale różnym kodzie. Chodzi oczywiście o metody `ustawXY`. Pierwsza wersja przyjmowała jako argumenty dwie liczby typu `int`, a druga miała tylko jeden argument, którym był obiekt klasy `Punkt`. Okazuje się, że takie dwie metody mogą współistnieć w klasie `Punkt` i z obu z nich możemy korzystać w kodzie programu.

Ogólnie rzecz ujmując, w każdej klasie może istnieć dowolna liczba metod, które mają takie same nazwy, o ile tylko różnią się argumentami. Mogą one — ale nie muszą — również różnić się typem zwracanego wyniku. Nazywamy to **przeciążaniem metod**. Skonstruujmy zatem taką klasę `Punkt`, w której znajdą się obie wersje metody `ustawXY`. Kod tej klasy jest przedstawiony na listingu 3.18.

Listing 3.18. *Przeciążone metody UstawXY w klasie Punkt*

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

  void ustawXY(int wspX, int wspY)
  {
    x = wspX;
    y = wspY;
  }
  void ustawXY(Punkt punkt)
  {
    x = punkt.x;
    y = punkt.y;
  }
}
```

Klasa ta zawiera w tej chwili dwie przeciążone metody o nazwie `ustawXY`. Jest to możliwe, ponieważ przyjmują one różne argumenty: pierwsza metoda — dwie liczby typu `int`, druga — jeden obiekt klasy `Punkt`. Obie metody realizują takie samo zadanie, tzn. ustawiają nowe wartości w polach `x` i `y`. Możemy przetestować ich działanie, dopisując do klasy `Punkt` metodę `Main` w postaci widocznej na listingu 3.19.

Listing 3.19. *Metoda Main do klasy Punkt z listingu 3.18*

```
public static void Main()
{
  Punkt punkt1 = new Punkt();
  Punkt punkt2 = new Punkt();

  punkt1.ustawXY(100, 100);
  punkt2.ustawXY(200,200);

  System.Console.WriteLine("Po pierwszym ustawieniu współrzędnych:");
  System.Console.WriteLine("x = " + punkt1.x);
  System.Console.WriteLine("y = " + punkt1.y);
  System.Console.WriteLine("");
```

```
    punkt1.ustawXY(punkt2);

    System.Console.WriteLine("Po drugim ustawieniu współrzędnych:");
    System.Console.WriteLine("x = " + punkt1.x);
    System.Console.WriteLine("y = " + punkt1.y);
  }
```

Działanie tej metody jest proste i nie wymaga wielu wyjaśnień. Tworzymy na początku dwa obiekty klasy `Punkt` i przypisujemy je zmiennym `punkt1` oraz `punkt2`. Następnie korzystamy z pierwszej wersji przeciążonej metody `ustawXY`, aby przypisać polom `x` i `y` pierwszego obiektu wartość `100`, a polom `x` i `y` drugiego obiektu — `200`. Dalej wyświetlamy zawartość obiektu `punkt1` na ekranie. Potem wykorzystujemy drugą wersję metody `ustawXY` w celu zmiany zawartości pól obiektu `punkt1`, tak aby zawierały wartości zapisane w obiekcie `punkt2`. Następnie ponownie wyświetlamy wartości pól obiektu `punkt1` na ekranie.

Argumenty metody Main

Każdy program musi zawierać punkt startowy, czyli miejsce, od którego zacznie się jego wykonywanie. W C# takim miejscem jest metoda o nazwie `Main` i następującej deklaracji:

```
public static void Main()
{
  //treść metody Main
}
```

Jeśli w danej klasie znajdzie się metoda w takiej postaci, od niej właśnie zacznie się wykonywanie kodu programu. Teraz powinno być już jasne, dlaczego dotychczas prezentowane przykładowe programy miały schematyczną konstrukcję:

```
class Program
{
  public static void main()
  {
    //tutaj instrukcje do wykonania
  }
}
```

Ta konstrukcja może mieć również nieco inną postać. Otóż metoda `Main` może przyjąć argument, którym jest tablica ciągów znaków. Zatem istnieje również jej przeciążona wersja o schematycznej postaci:

```
public static void Main(String[] args)
{
  //treść metody Main
}
```

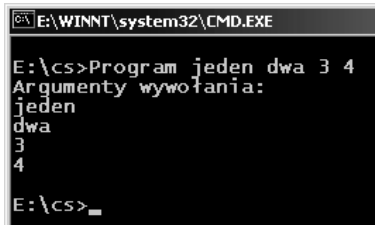
Tablica `args` zawiera parametry wywołania programu, czyli argumenty przekazane z wiersza poleceń. O tym, że tak jest w istocie, możemy przekonać się, uruchamiając program widoczny na listingu 3.20. Wykorzystuje on pętlę `for` do przejrzenia i wyświetlenia na ekranie zawartości wszystkich komórek tablicy `args`. Przykładowy wynik jego działania jest widoczny na rysunku 3.8.

Listing 3.20. *Odczytanie argumentów podanych z wiersza poleceń*

```
using System;

public class Program
{
  public static void Main(String[] args)
  {
    Console.WriteLine("Argumenty wywołania:");
    for(int i = 0; i < args.Length; i++)
    {
      Console.WriteLine(args[i]);
    }
  }
}
```

Rysunek 3.8. *Program wyświetlający parametry jego wywołania*



Sposoby przekazywania argumentów

Argumenty metod domyślnie przekazywane są przez wartość. To oznacza, że wewnątrz metody dostępna jest tylko kopia argumentu i jakiekolwiek zmiany jego wartości będą wykonywane na tej kopii i obowiązywały wyłącznie wewnątrz metody. Przykładowo jeśli mamy metodę `Zwieksz` o postaci:

```
public void Zwieksz(int arg)
{
  arg++;
}
```

i w którymś miejscu programu wywołamy ją, przekazując jako argument zmienną `liczba`, np. w następujący sposób:

```
int liczba = 100;
Zwieksz(liczba);
Console.WriteLine(liczba);
```

To metoda `Zwieksz` otrzyma do dyspozycji kopię wartości zmiennej `liczba` i zwiększenie wykonywane przez instrukcję `arg++;` będzie obowiązywało tylko w obrębie tej metody. Instrukcja `Console.WriteLine(liczba);` spowoduje więc wyświetlenie wartości 100.

To zachowanie można zmienić — argumenty mogą być również przekazywane przez referencję. Metoda otrzyma wtedy w postaci argumentu referencję do zmiennej i będzie mogła bezpośrednio operować na tej zmiennej (a nie na jej kopii). W takiej sytuacji należy zastosować słowa `ref` lub `out`. Różnica jest taka, że w pierwszym przypadku

przekazywana zmienna musi być zainicjowana przed przekazaniem jej jako argument, a w przypadku drugim musi zaś być zainicjowana wewnątrz metody. Metoda `Zwieksz` mogłaby mieć zatem postać:

```
public void Zwieksz(ref int arg)
{
  arg++;
}
```

Wtedy fragment kodu:

```
int liczba = 100;
Zwieksz(ref liczba);
Console.WriteLine(liczba);
```

spowodowałby faktyczne zwiększenie zmiennej `liczba` o 1 i na ekranie, dzięki działaniu instrukcji `Console.WriteLine(liczba);`, pojawiłaby się wartość 101. Należy przy tym zwrócić uwagę, że słowo `ref` (a także `out`) musi być użyte również w wywołaniu metody (a nie tylko przy jej deklaracji).

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 15.1

Do klasy `Punkt` z listingu 3.14 dopisz metody `UstawX` i `UstawY`, które jako parametr będą przyjmowały obiekt klasy `Punkt`.

Ćwiczenie 15.2

W klasie `Punkt` z listingu 3.14 zmień kod metod `UstawX` i `UstawY`, tak aby zwracały one poprzednią wartość zapisywanych pól. Zadaniem metody `UstawX` jest więc zmiana wartości pola `x` i zwrócenie jego poprzedniej wartości. Metoda `UstawY` ma wykonywać analogiczne czynności w stosunku do pola `y`.

Ćwiczenie 15.3

Do klasy `Punkt` z ćwiczenia 15.2 dopisz metodę `UstawXY` przyjmującą jako argument obiekt klasy `Punkt`. Polom `x` i `y` należy przypisać wartości pól `x` i `y` przekazanego obiektu. Metoda ma natomiast zwrócić obiekt klasy `Punkt` zawierający stare wartości `x` i `y`.

Lekcja 16. Konstruktory

Lekcja 16. jest poświęcona **konstruktorom**, czyli specjalnym metodom wykonywanym podczas tworzenia obiektu. Dowiemy się, jak powstaje konstruktor, jak umieścić go w klasie, a także czy może przyjmować argumenty. Znajdą się tu również informacje o sposobach przeciążania konstruktorów oraz o wykorzystaniu słowa kluczowego `this`.