Dołącz do grona profesjonalnych programistów C#!

- Poznaj podstawy języka C# i zasady korzystania z platformy .NET
- Dowiedz się, jak używać popularnych środowisk programistycznych
- Naucz się tworzyć aplikacje różnego typu w języku C#

Wydanie II

Marcin Lis

Helion

Zawiera CD

Redaktor prowadzący: Ewelina Burska

Projekt okładki: Maciej Pasek

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?cshpk2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Materiały do książki można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/cshpk2.zip

ISBN: 978-83-246-3870-3

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 wyjaśnić, dlaczego właśnie tak. Wszystko przedstawi niniejszy rozdział.

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** jako pewien byt programistyczny, który może przechowywać dane i wykonywać operacje, czyli różne zadania. **Klasa** to z kolei definicja, opis takiego obiektu.

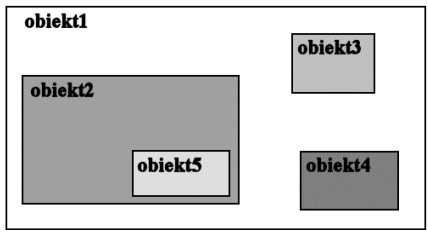
Skoro klasa definiuje obiekt, jest zatem również jego typem. Czym jest **typ obiektu**? Przytoczmy jedną z definicji: „Typ jest przypisany zmiennej, wyrażeniu lub innemu bytowi programistycznemu (danej, obiektowi, funkcji, procedurze, operacji, metodzie, parametrowi, modułowi, wyjątkowi, zdarzeniu). 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”,

[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*

„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.

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 polskie znaki diakrytyczne (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`:

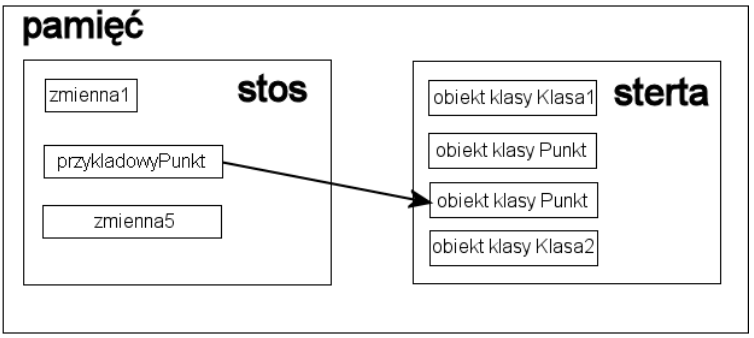
```
Punkt przykladowyPunkt = new Punkt();
```

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;
```

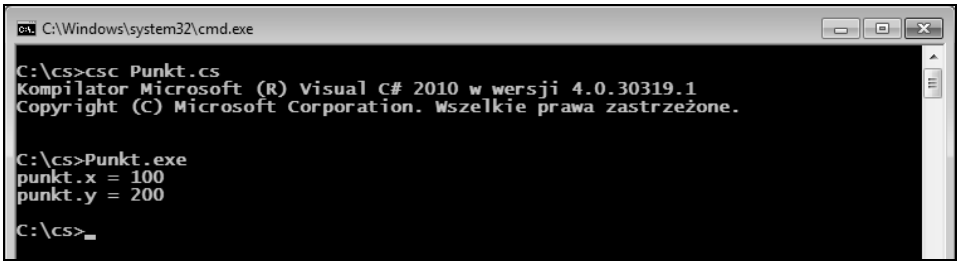
Jak użyć klasy?

Spróbujmy teraz się przekonać, 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 więc posługiwać nią tak, jakbyśmy posługiwali się samym obiektem. Pisząc `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** (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:

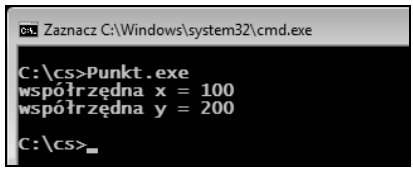
```
nazwa_zmiennej.nazwa_metody(argumenty_metody);
```

Oczywiście, jeśli dana metoda nie ma argumentów, po prostu je pomijamy. Przy czym termin *wywołanie* oznacza 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
{
```



Rysunek 3.4. *Wynik działania metody WyswietlWspolrzedne klasy 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();
  }
}
```

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 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 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 przypisywany jest efekt działania (zwrócona wartość) metody `PobierzX`, a drugiej — efekt działania metody `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 `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;
}
```

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

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 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 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 pliku lub też w 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 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 biblioteką klas wspomagających tworzenie grafiki na płaszczyźnie. Można zatem przygotować dwie osobne przestrzenie nazw, np. o nazwach `Grafika2D` i `Grafika3D`. W takiej sytuacji każdy programista 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 3.10 i 3.11.

Listing 3.10. *Klasa Punkt w przestrzeni nazw Grafika2D*

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

Listing 3.11. *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? Istnieją 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. Na przykład odwołanie do klasy `Punkt` z przestrzeni `Grafika2D` miałoby postać:

```
Grafika2D.Punkt
```

Sposób ten został przedstawiony na listingu 3.12.

Listing 3.12. *Użycie klasy Punkt 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. Nie 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 z kilku przestrzeni nazw, używamy kilku dyrektyw `using`. Jasne jest więc już, co oznacza fragment:

```
using System;
```

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

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

Tak więc nasz pierwszy program z listingu 1.1 równie dobrze mógłby mieć postać widoczną na listingu 3.13.

Listing 3.13. *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 3.12 nie trzeba było odwoływać się do przestrzeni nazw `Grafika2D` przy każdym wystąpieniu klasy `Punkt`, należałoby użyć instrukcji `using Grafika2D`, tak jak zostało to zaprezentowane na listingu 3.14.

Listing 3.14. *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 ustalić, 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. Jeśli więc 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, wiadomo z lekcji 14. Czas dowiedzieć się, jak się nimi posługiwać. Właśnie temu tematowi została poświęcona cała lekcja 15. Będzie w niej wyjaśnione, w jaki sposób przekazuje się metodom argumenty typów prostych oraz referencyjnych, będzie też omówione przeciążanie metod, czyli technika umożliwiająca umieszczenie w jednej klasie kilku metod o tej samej nazwie. Nieco miejsca zostanie także poświęcone metodzie `Main`, od której zaczyna się wykonywanie aplikacji. Zobaczymy również, w jaki sposób przekazać aplikacji parametry z wiersza poleceń.

Argumenty metod

W lekcji 14. powiedziano, 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 przekazanej 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 można napisać metodę, która będzie jednocześnie ustawiała pola `x` i `y` klasy `Punkt`. Oczywiście będzie ona przyjmowała dwa argumenty, które w deklaracji należy oddzielić przecinkiem. Zatem cała konstrukcja będzie 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 3.15.

Listing 3.15. *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 klasy `Punkt`. Dzięki temu będziemy mogli sprawdzić, czy wszystkie trzy działają zgodnie z naszymi założeniami. Taka przykładowa metoda jest widoczna na listingu 3.16[5].

Listing 3.16. *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 obiektu `drugiPunkt` wartości `200`, oraz wyświetlamy je na ekranie, również za pomocą metody `WyswietlWspolrzedne`. Po skompilowaniu i uruchomieniu tego programu otrzymamy widok jak na rysunku 3.5.

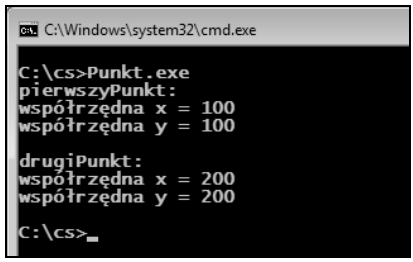
Obiekt jako argument

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

[5] Na listingach zamieszczonych 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 przedstawioną tu uproszczoną terminologię, w której referencję utożsamia się z obiektem.

Rysunek 3.5. *Efekt wykonania programu z listingu 3.16*



postępowanie jest takie same 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 zostało to zaprogramowane we wcześniejszych przykładach (listing 3.15). 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 widać, nie przyjmuje ona żadnych argumentów, nie ma przecież takiej potrzeby; z deklaracji wynika jednak, że zwraca 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 3.17.

Listing 3.17. *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 najnowszej wersji klasy `Punkt` z listingu 3.17. Kod ten został zaprezentowany na listingu 3.18.

Listing 3.18. *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");
    }
```

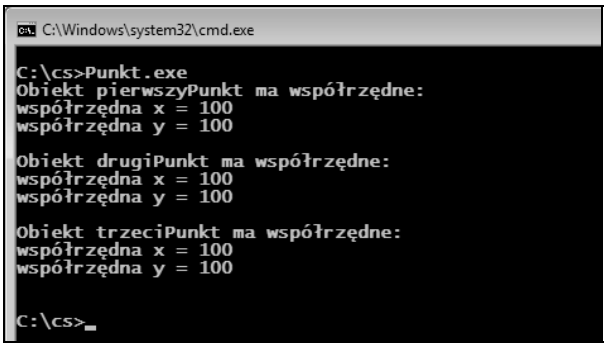
Na początku deklarujemy zmienne `pierwszyPunkt` oraz `drugiPunkt`. Zmiennej `pierwszy↪Punkt` przypisujemy nowo utworzony obiekt klasy `Punkt` (rysunek 3.7 A). Następnie wykorzystujemy znane nam dobrze metody `UstawX` i `UstawY` do przypisania polom `x` i `y` wartości `100` oraz wyświetlamy te dane na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

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:

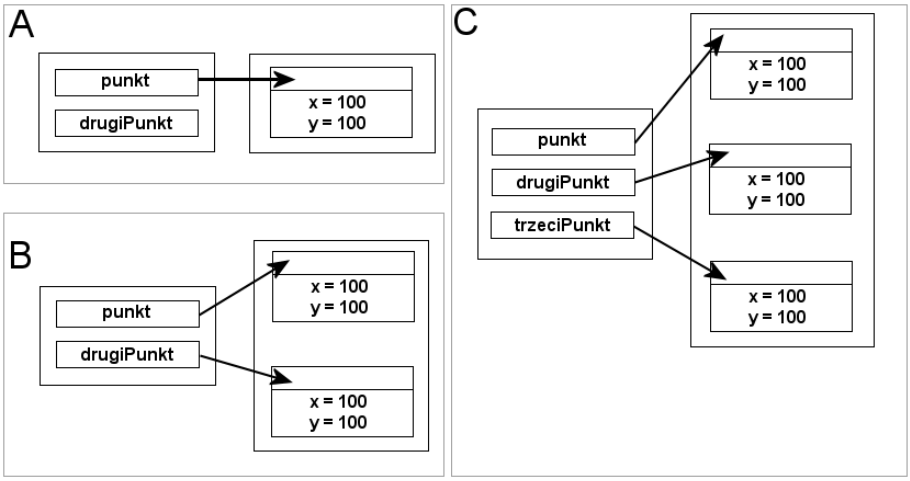
```
drugiPunkt = pierwszyPunkt.PobierzWspolrzedne();
```

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 `pierwszy↪Punkt`. 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 3.18 jest taki jak zaprezentowany na rysunku 3.6. Z kolei na rysunku 3.7 przedstawione zostały 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`). Przedstawiono to na rysunku 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 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 można 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. Taka funkcjonalność nosi nazwę **przeciążania metod** (ang. *methods overloading*). Skonstruujmy zatem taką klasę `Punkt`, w której znajdą się obie wersje metody `ustawXY`. Kod tej klasy został przedstawiony na listingu 3.19.

Listing 3.19. *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 3.20.

Listing 3.20. *Metoda Main do klasy Punkt z listingu 3.19*

```
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. Na początku tworzymy 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
}
```

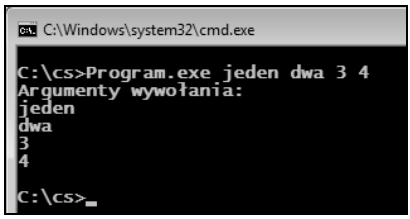
Tablica `args` zawiera parametry wywołania programu, czyli argumenty przekazane z wiersza poleceń. O tym, że tak jest w istocie, można się przekonać, uruchamiając program widoczny na listingu 3.21. Wykorzystuje on pętlę `for` do przejrzenia i wyświetlenia na ekranie zawartości wszystkich komórek tablicy `args`. Przykładowy wynik jego działania jest widoczny na rysunku 3.8.

Listing 3.21. *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]);
    }
  }
}
```

Sposoby przekazywania argumentów

Argumenty metod domyślnie przekazywane są **przez wartość** (ang. *by value*). To oznacza, że wewnątrz metody dostępna jest tylko kopia argumentu, a w związku z tym jakiekolwiek zmiany jego wartości będą wykonywane na tej kopii i obowiązywały wyłącznie wewnątrz metody. Jeśli mamy na przykład metodę `Zwieksz` o postaci:



Rysunek 3.8. *Program wyświetlający parametry jego wywołania*

```
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** (ang. *by reference*). 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 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). Praktyczne różnice w opisanych sposobach przekazywania argumentów zostały zobrazowane w przykładzie widocznym na listingu 3.22.

Listing 3.22. *Różnice w sposobach przekazywania argumentów*

```
using System;

public class Program
```

```
{
  public void Zwieksz1(int arg)
  {
    arg++;
  }
  public void Zwieksz2(ref int arg)
  {
    arg++;
  }
  public void Zwieksz3(out int arg)
  {
    //int wartosc = arg;
    //arg++;
    arg = 10;
    arg++;
  }
  public static void Main(String[] args)
  {
    int liczba1 = 100, liczba2;
    Program pg = new Program();

    pg.Zwieksz1(liczba1);
    Console.WriteLine("Po wywołaniu Zwieksz1(liczba1):");
    Console.WriteLine(liczba1);

    pg.Zwieksz2(ref liczba1);
    Console.WriteLine("Po wywołaniu Zwieksz2(ref liczba1):");
    Console.WriteLine(liczba1);

    //pg.Zwieksz2(ref liczba2);

    pg.Zwieksz3(out liczba1);
    Console.WriteLine("Po wywołaniu Zwieksz3(out liczba1):");
    Console.WriteLine(liczba1);

    pg.Zwieksz3(out liczba2);
    Console.WriteLine("Po wywołaniu Zwieksz3(out liczba2):");
    Console.WriteLine(liczba2);
  }
}
```

W kodzie zostały zdefiniowane trzy metody przyjmujące jeden argument typu `int`, zajmujące się zwiększaniem jego wartości. Pierwsza z nich (`Zwieksz1`) jest standardowa — argument nie zawiera żadnych modyfikatorów, a jego wartość jest zwiększana o jeden za pomocą operatora ++. Druga (`Zwieksz2`) ma identyczną konstrukcję, ale przed argumentem został zastosowany modyfikator `ref`. To oznacza, że zmienna przekazywana jako argument będzie musiała być zainicjowana. W trzeciej metodzie (`Zwieksz3`) użyty został modyfikator `out`. To oznacza, że jej pierwsze dwie instrukcje są nieprawidłowe (dlatego zostały ujęte w komentarz). Otóż taki argument musi zostać zainicjowany wewnątrz metody przed wykonaniem jakiejkolwiek operacji na nim. Prawidłowa jest zatem dopiero trzecia instrukcja przypisująca argumentowi wartość `10`, a także czwarta — zwiększająca tę wartość o jeden (po pierwszym przypisaniu wartości można już wykonywać dowolne operacje).

Działanie metod `Zwieksz` jest testowane w metodzie `main`, od której zaczyna się wykonywanie kodu programu. Zostały w niej zadeklarowane dwie zmienne: `liczba1` i `liczba2`, pierwsza z nich została też od razu zainicjalizowana wartością `100`, natomiast druga pozostała niezainicjowana. Następnie powstał obiekt `pg` klasy `Program`. Jest to konieczne, ponieważ aby korzystać z metod zdefiniowanych w klasie `Program`, niezbędny jest obiekt tej klasy[7]. Dalsze bloki kodu to wywołania kolejnych wersji metod `Zwieksz` i wyświetlanie bieżącego stanu zmiennej użytej w wywołaniu.

W pierwszym bloku używana jest metoda `Zwieksz1`, której w tradycyjny sposób przekazywany jest argument w postaci zmiennej `liczba1`. To oznacza, że metoda otrzymuje w istocie kopię zmiennej i operuje na tej kopii. A zatem zwiększenie wartości argumentu (`arg++`) obowiązuje wyłącznie w obrębie metody. Wartość zmiennej `liczba1` w metodzie `main` nie ulegnie zmianie (będzie równa `100`).

W drugim bloku kodu używana jest metoda `Zwieksz2`, której przekazywany jest przez referencję z użyciem słowa `ref` argument w postaci zmiennej `liczba1`. Jest to prawidłowe wywołanie, gdyż `liczba1` została zainicjowana w metodzie `main` i w związku z tym ma określoną wartość. Takie wywołanie oznacza jednak, że we wnętrzu metody `Zwieksz2` operacje wykonywane są na zmiennej `liczba1` (a nie na jej kopii, jak miało to miejsce w przypadku metody `Zwieksz1`). W efekcie po wywołaniu `pg.Zwieksz2(ref liczba1)` zmienna `liczba1` w metodzie `main` będzie miała wartość `101` (została zwiększona przez instrukcję `arg++` znajdującą się w metodzie `Zwieksz2`).

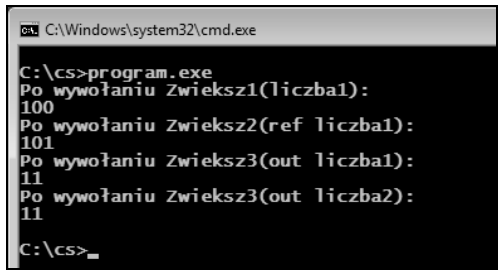
Trzeci blok kodu zawiera tylko jedną instrukcję: `pg.Zwieksz2(ref liczba2);`, która została ujęta w komentarz, gdyż jest nieprawidłowa. Z użyciem słowa `ref` nie można przekazać metodzie `Zwieksz2` argumentu w postaci zmiennej `liczba2`, ponieważ ta zmienna nie została zainicjowana. Tymczasem słowo `ref` oznacza, że taka inicjalizacja jest wymagana. Usunięcie komentarza z tej instrukcji spowoduje więc błąd kompilacji.

W piątym bloku kodu używana jest metoda `Zwieksz3`, której przekazywany jest przez referencję z użyciem słowa `out` argument w postaci zmiennej `liczba1`. Takie wywołanie jest prawidłowe, zmienna przekazywana z użyciem słowa `out` może być wcześniej zainicjalizowana, należy jednak pamiętać, że jej pierwotna wartość zostanie zawsze zmieniona w wywoływanej metodzie. Dlatego po tym wywołaniu zmienna `liczba1` będzie miała wartość `11` (wartość wynikającą z przypisań wykonywanych w metodzie `Zwieksz3`).

W czwartym bloku kodu używana jest metoda `Zwieksz3`, której przekazywany jest przez referencję z użyciem słowa `out` argument w postaci zmiennej `liczba2`. To wywołanie jest również właściwe — słowo `out` wskazuje, że zmienna `liczba2` nie musi być zainicjowana przed przekazaniem do metody, ponieważ ta operacja zostanie wykonana właśnie w metodzie. W efekcie po wykonaniu metody `Zwieksz3` zmienna `liczba2` otrzyma wartość `11`.

Ostatecznie zatem po skompilowaniu i uruchomieniu programu z listingu 3.22 na ekranie pojawi się widok przedstawiony na rysunku 3.9.

[7] Inaczej metody musiałyby być zadeklarowane jako statyczne. Ta kwestia zostanie wyjaśniona w lekcji 19.



Rysunek 3.9. *Stan zmiennych przy różnych wywołaniach metod*

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 15.1

Do klasy `Punkt` z listingów 3.15 i 3.16 dopisz metody `UstawX` i `UstawY`, które jako argument będą przyjmowały obiekt klasy `Punkt`.

Ćwiczenie 15.2

W klasie `Punkt` z listingu 3.15 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 zadania 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`.

Ćwiczenie 15.4

Napisz kod przykładowej klasy o nazwie `Dzialania`. Umieść w niej metody `Dodaj` i `Odejmij` oraz pole o nazwie `wynik` (w deklaracji pola użyj słowa `public`, podobnie jak na listingu 3.8). Metoda `Dodaj` powinna przyjmować dwa argumenty oraz zapisywać wynik ich dodawania w polu `wynik`. Metoda `Odejmij` powinna działać analogicznie, z tą różnicą, że rezultatem jej wykonania powinna być różnica przekazanych argumentów.

Ćwiczenie 15.5

W oparciu o kod z ćwiczenia 15.4 napisz taką wersję klasy `Dzialania`, która wynik wykonywanych operacji będzie zapisywała w pierwszym argumencie, a jego pierwotna zawartość znajdzie się w polu `wynik`. Pamiętaj o takim sposobie przekazywania argumentów, aby wynik operacji dodawania lub odejmowania był dostępny po wywołaniu dowolnej z metod.

Ćwiczenie 15.6

Napisz przykładowy program ilustrujący działanie klas z ćwiczeń 15.4 i 15.5. Zastanów się, jakie modyfikacje musisz wprowadzić, aby móc skorzystać z tych klas w jednym programie.

Lekcja 16. Konstruktory i destruktory

Lekcja 16. w większej części jest poświęcona **konstruktorom**, czyli specjalnym metodom wykonywanym podczas tworzenia obiektu. Można się z niej dowiedzieć, jak powstaje konstruktor, jak umieścić go w klasie, a także czy może przyjmować argumenty. Nie zostaną też pominięte informacje o sposobach przeciążania konstruktorów oraz o wykorzystaniu słowa kluczowego `this`. Na zakończenie przedstawiony zostanie też temat **destruktorów**, które są wykonywane, kiedy obiekt jest usuwany z pamięci.

Czym jest konstruktor?

Po utworzeniu obiektu w pamięci wszystkie jego pola zawierają wartości domyślne. Wartości te dla poszczególnych typów danych zostały przedstawione w tabeli 3.1.

Tabela 3.1. *Wartości domyślne niezainicjowanych pól obiektu*

Typ	Wartość domyślna
`byte`	`0`
`sbyte`	`0`
`short`	`0`
`ushort`	`0`
`int`	`0`
`uint`	`0`
`long`	`0`
`ulong`	`0`
`decimal`	`0.0`
`float`	`0.0`
`double`	`0.0`
`char`	`\0`
`bool`	`false`
obiektowy	`null`

Najczęściej jednak chcemy, aby pola te zawierały jakieś konkretne wartości. Przykładowo moglibyśmy życzyć sobie, aby każdy obiekt klasy `Punkt` powstałej w lekcji 14. (listing 3.1) otrzymywał współrzędne: `x = 1` i `y = 1`. Oczywiście można po każdym utworzeniu obiektu przypisywać wartości tym polom, np.:

```
Punkt punkt1 = new Punkt();
punkt1.x = 1;
punkt1.y = 1;
```

Można też dopisać do klasy `Punkt` dodatkową metodę, na przykład o nazwie `inicjuj` (albo `init`, `initialize` lub podobnej), w postaci:

```
void init()
{
  x = 1;
  y = 1;
}
```

i wywoływać ją po każdym utworzeniu obiektu. Widać jednak od razu, że żadna z tych metod nie jest wygodna. Przede wszystkim wymagają one, aby programista zawsze pamiętał o ich stosowaniu, a jak pokazuje praktyka, jest to zwykle zbyt optymistyczne założenie. Na szczęście obiektowe języki programowania udostępniają dużo wygodniejszy mechanizm konstruktorów. Otóż konstruktor jest to specjalna metoda, która jest wywoływana zawsze w trakcie tworzenia obiektu w pamięci. Nadaje się więc doskonale do jego zainicjowania.

Metoda będąca konstruktorem nigdy nie zwraca żadnego wyniku i musi mieć nazwę zgodną z nazwą klasy, czyli schematycznie wygląda to następująco:

```
class nazwa_klasy
{
  nazwa_klasy()
  {
    //kod konstruktora
  }
}
```

Jak widać, przed definicją nie umieszcza się nawet słowa `void`, tak jak miałoby to miejsce w przypadku zwykłej metody. To, co będzie robił konstruktor, czyli jakie wykona zadania, zależy już tylko od programisty.

Dopiszmy zatem do klasy `Punkt` z listingu 3.1 (czyli jej najprostszej wersji) konstruktor, który będzie przypisywał polom `x` i `y` każdego obiektu wartość `1`. Wygląd takiej klasy zaprezentowano na listingu 3.23.

Listing 3.23. *Prosty konstruktor dla klasy Punkt*

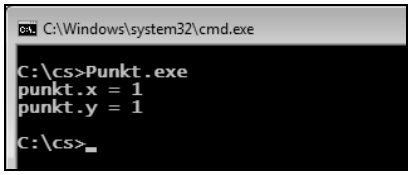
```
class Punkt
{
  int x;
  int y;
  Punkt()
  {
    x = 1;
    y = 1;
  }
}
```

Jak widać, wszystko jest tu zgodne z podanym wyżej schematem. Konstruktor nie zwraca żadnej wartości i ma nazwę zgodną z nazwą klasy. Przed nazwą nie występuje słowo `void`. W jego ciele następuje proste przypisanie wartości polom obiektu. O tym, że konstruktor faktycznie działa, można się przekonać, pisząc dodatkowo metodę `Main`, w której skorzystamy z obiektu nowej klasy `Punkt`. Taka przykładowa metoda `Main` jest widoczna na listingu 3.24.

Listing 3.24. *Metoda Main testująca konstruktor klasy Punkt*

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

Metoda ta ma wyjątkowo prostą konstrukcję, jedyne jej zadania to utworzenie obiektu klasy `Punkt` i przypisanie odniesienia do niego zmiennej `punkt1` oraz wyświetlenie zawartości jego pól na ekranie. Dzięki temu przekonamy się, że konstruktor faktycznie został wykonany, zobaczymy bowiem widok zaprezentowany na rysunku 3.10.



Rysunek 3.10. *Konstruktor klasy Punkt faktycznie został wykonany*

Argumenty konstruktorów

Konstruktor nie musi być bezargumentowy, może również przyjmować argumenty, które zostaną wykorzystane, bezpośrednio lub pośrednio, na przykład do zainicjowania pól obiektu. Argumenty przekazuje się dokładnie tak samo jak w przypadku zwykłych metod (por. lekcja 15.). Schemat takiego konstruktora byłby więc następujący:

```
class nazwa_klasy
{
  nazwa_klasy(typ1 argument1, typ2 argument2,..., typN argumentN)
  {
  }
}
```

Oczywiście, jeśli konstruktor przyjmuje argumenty, to przy tworzeniu obiektu należy je podać, czyli zamiast stosowanej do tej pory konstrukcji:

```
nazwa_klasy zmienna = new nazwa_klasy()
```

trzeba zastosować wywołanie:

```
nazwa_klasy zmienna = new nazwa_klasy(argumenty_konstruktora)
```

W przypadku naszej klasy `Punkt` byłby przydatny np. konstruktor przyjmujący dwa argumenty, które oznaczałyby współrzędne punktu. Jego definicja, co nie jest z pewnością żadnym zaskoczeniem, wyglądać będzie następująco:

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

Kiedy zostanie umieszczony w klasie `Punkt`, przyjmie ona postać widoczną na listingu 3.25.

Listing 3.25. *Konstruktor przyjmujący argumenty*

```
class Punkt
{
  int x;
  int y;
  Punkt(int wspX, int wspY)
  {
    x = wspX;
    y = wspY;
  }
}
```

Teraz podczas każdej próby utworzenia obiektu klasy `Punkt` będziemy musieli podawać jego współrzędne. Jeśli na przykład początkowa współrzędna `x` ma mieć wartość `100`, a początkowa współrzędna `y` — `200`, powinniśmy zastosować konstrukcję:

```
Punkt punkt = new Punkt(100, 200);
```

Taka instrukcja może być umieszczona w metodzie `Main` (analogicznie do przykładu z listingu 3.24) testującej zachowanie tej wersji konstruktora, której przykładowa postać została zaprezentowana na listingu 3.26.

Listing 3.26. *Testowanie konstruktora przyjmującego argumenty*

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

Przeciążanie konstruktorów

Konstruktory, tak jak zwykłe metody, mogą być przeciążane, tzn. każda klasa może mieć kilka konstruktorów, o ile tylko różnią się one przyjmowanymi argumentami. Do tej pory powstały dwa konstruktory klasy `Punkt`: pierwszy bezargumentowy i drugi przyjmujący dwa argumenty typu `int`. Dopiszmy zatem jeszcze trzeci, który jako argument będzie przyjmował obiekt klasy `Punkt`. Jego postać będzie zatem następująca:

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

Zasada działania jest prosta: polu `x` jest przypisywana wartość pola `x` obiektu przekazanego jako argument, natomiast polu `y` — wartość pola `y` tego obiektu. Można teraz zebrać wszystkie trzy napisane dotychczas konstruktory i umieścić je w klasie `Punkt`. Będzie ona wtedy miała postać przedstawioną na listingu 3.27.

Listing 3.27. *Trzy konstruktory w klasie Punkt*

```
class Punkt
{
  int x;
  int y;
  Punkt()
  {
    x = 1;
    y = 1;
  }
  Punkt(int wspX, int wspY)
  {
    x = wspX;
    y = wspY;
  }
  Punkt(Punkt punkt)
  {
    x = punkt.x;
    y = punkt.y;
  }
}
```

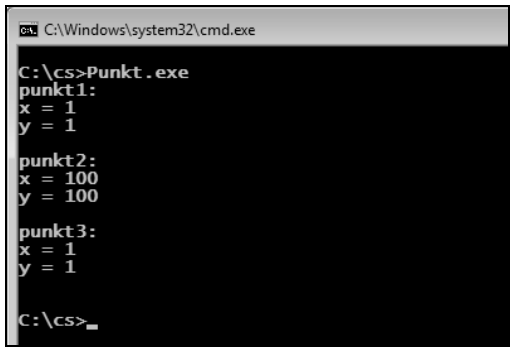
Taka budowa klasy `Punkt` pozwala na osobne wywoływanie każdego z trzech konstruktorów, w zależności od tego, który z nich jest najbardziej odpowiedni w danej sytuacji. Warto teraz na konkretnym przykładzie przekonać się, że tak jest w istocie. Dopiszemy więc do klasy `Punkt` metodę `Main`, w której utworzymy trzy obiekty typu `Punkt`, a każdy z nich będzie tworzony za pomocą innego konstruktora. Taka przykładowa metoda jest widoczna na listingu 3.28.

Listing 3.28. *Użycie przeciążonych konstruktorów*

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

  System.Console.WriteLine("punkt1:");
  System.Console.WriteLine("x = " + punkt1.x);
  System.Console.WriteLine("y = " + punkt1.y);
  System.Console.WriteLine("");

  Punkt punkt2 = new Punkt(100, 100);

  System.Console.WriteLine("punkt2:");
```

```
    System.Console.WriteLine("x = " + punkt2.x);
    System.Console.WriteLine("y = " + punkt2.y);
    System.Console.WriteLine("");

    Punkt punkt3 = new Punkt(punkt1);

    System.Console.WriteLine("punkt3:");
    System.Console.WriteLine("x = " + punkt3.x);
    System.Console.WriteLine("y = " + punkt3.y);
    System.Console.WriteLine("");
  }
```

Pierwszy obiekt — `punkt1` — jest tworzony za pomocą konstruktora bezargumentowego, który przypisuje polom `x` i `y` wartość `1`. Obiekt drugi — `punkt2` — jest tworzony poprzez wywołanie drugiego ze znanych nam konstruktorów, który przyjmuje dwa argumenty odzwierciedlające wartości `x` i `y`. Oba pola otrzymują wartość `100`. Konstruktor trzeci, zastosowany wobec obiektu `punkt3`, to nasza najnowsza konstrukcja. Jako argument przyjmuje on obiekt klasy `Punkt`, w tym przypadku obiekt wskazywany przez `punkt1`. Ponieważ w tym obiekcie oba pola mają wartość `1`, również pola obiektu `punkt3` przyjmą wartość `1`. W efekcie działania programu na ekranie zobaczymy widok zaprezentowany na rysunku 3.11.



Rysunek 3.11. *Wykorzystanie trzech różnych konstruktorów klasy Punkt*

Słowo kluczowe this

Słowo kluczowe `this` to nic innego jak odwołanie do obiektu bieżącego. Można je traktować jako referencję do aktualnego obiektu. Najłatwiej pokazać to na przykładzie. Załóżmy, że mamy konstruktor klasy `Punkt`, taki jak na listingu 3.25, czyli przyjmujący dwa argumenty, którymi są liczby typu `int`. Nazwami tych argumentów były `wspX` i `wspY`. Co by się jednak stało, gdyby ich nazwami były `x` i `y`, czyli gdyby jego deklaracja wyglądała jak poniżej?

```
Punkt(int x, int y)
{
}
```

Co należy wpisać w jego treści, aby spełniał swoje zadanie? Gdybyśmy postępowali w sposób podobny jak w przypadku klasy z listingu 3.25, otrzymalibyśmy konstrukcję:

```
Punkt(int x, int y) {
  x = x;
  y = x;
}
```

Oczywiście, nie ma to najmniejszego sensu[8]. W jaki bowiem sposób kompilator ma ustalić, kiedy chodzi nam o argument konstruktora, a kiedy o pole klasy, jeśli ich nazwy są takie same? Oczywiście sam sobie nie poradzi i tu właśnie z pomocą przychodzi nam słowo `this`. Otóż jeśli chcemy zaznaczyć, że chodzi nam o składową klasy (np. pole, metodę), korzystamy z odwołania w postaci:

```
this.nazwa_pola
```

lub:

```
this.nazwa_metody(argumenty)
```

Wynika z tego, że poprawna postać opisywanego konstruktora powinna wyglądać następująco:

```
Punkt(int x, int y)
{
  this.x = x;
  this.y = y;
}
```

Instrukcję `this.x = x` rozumiemy jako: „Przypisz polu `x` wartość przekazaną jako argument o nazwie `x`", a instrukcję `this.y = y` analogicznie jako: „Przypisz polu `y` wartość przekazaną jako argument o nazwie `y`".

Słowo `this` pozwala również na inną ciekawą konstrukcję. Umożliwia mianowicie wywołanie konstruktora z wnętrza innego konstruktora. Może to być przydatne w sytuacji, kiedy w klasie mamy kilka przeciążonych konstruktorów, a zakres wykonywanego przez nie kodu się pokrywa. Nie zawsze takie wywołanie jest możliwe i niezbędne, niemniej taka możliwość istnieje, trzeba więc wiedzieć, jak takie zadanie zrealizować.

Jeżeli za jednym z konstruktorów umieścimy dwukropek, a za nim słowo `this` i listę argumentów umieszczonych w nawiasie okrągłym, czyli zastosujemy konstrukcję o schemacie:

```
class nazwa_klasy
{
  nazwa_klasy(argumenty):this(argument1, argument2, ... , argumentN)
  {
  }
  //pozostałe konstruktory
}
```

to przed widocznym konstruktorem zostanie wywołany ten, którego argumenty pasują do wymienionych w nawiasie po `this`. Jest to tak zwane zastosowanie **inicjalizatora** lub **listy inicjalizacyjnej**. Przykład kodu wykorzystującego taką technikę jest widoczny na listingu 3.29.

[8] Chociaż formalnie taki zapis jest w pełni poprawny.

Listing 3.29. *Wywołanie konstruktora z wnętrza innego konstruktora*

```
class Punkt
{
  int x;
  int y;
  Punkt(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
  Punkt():this(1, 1)
  {
  }
  public static void Main()
  {
    Punkt punkt1 = new Punkt(100, 200);
    Punkt punkt2 = new Punkt();
    System.Console.WriteLine("punkt1.x = " + punkt1.x);
    System.Console.WriteLine("punkt1.y = " + punkt1.y);
    System.Console.WriteLine("");
    System.Console.WriteLine("punkt2.x = " + punkt2.x);
    System.Console.WriteLine("punkt2.y = " + punkt2.y);
  }
}
```

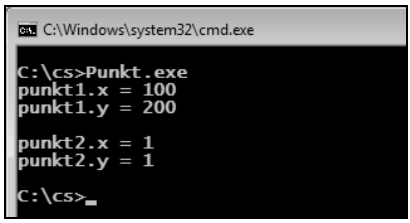
Klasa `Punkt` ma teraz dwa konstruktory. Jeden z nich ma postać standardową — przyjmuje po prostu dwa argumenty typu `int` i przypisuje ich wartości polom `x` i `y`. Drugi z konstruktorów jest natomiast bezargumentowy, a jego zadaniem jest przypisanie polom `x` i `y` wartości `1`. Nie dzieje się to jednak w sposób znany z dotychczasowych przykładów. Wnętrze tego konstruktora jest puste[9], a wykorzystywana jest lista inicjalizacyjna — `Punkt():this(1, 1)`. Dzięki temu jest wywoływany konstruktor, którego argumenty są zgodne z podanymi na liście, a więc konstruktor przyjmujący dwa argumenty typu `int`.

Jak więc zadziała kod metody `Main`? Najpierw za pomocą konstruktora dwuargumentowego jest tworzony obiekt `punkt1`. Tu nie dzieje się nic nowego, pola otrzymują zatem wartości `100` i `200`. Następnie powstaje obiekt `punkt2`, a do jego utworzenia jest wykorzystywany konstruktor bezargumentowy. Ponieważ korzysta on z listy inicjalizacyjnej, najpierw zostanie wywołany konstruktor dwuargumentowy, któremu w postaci argumentów zostaną przekazane dwie wartości `1`. A zatem oba pola obiektu `punkt2` przyjmą wartość `1`. Przekonujemy się o tym, wyświetlając wartości pól obu obiektów na ekranie (rysunek 3.12).

Argumentem przekazywanym na liście inicjalizacyjnej może też być argument konstruktora (patrz też zadanie 16.4 w podrozdziale „Ćwiczenia do samodzielnego wykonania"). Schematycznie można taką sytuację przedstawić następująco:

```
class nazwa_klasy
{
  nazwa_klasy(argument)
```

[9] Oczywiście nie jest to obligatoryjne. Konstruktor korzystający z listy inicjalizacyjnej może również zawierać instrukcje wykonujące inne czynności.



Rysunek 3.12. *Efekt użycia konstruktora korzystającego z listy inicjalizacyjnej*

```
    {
    }
    nazwa_klasy(argument1, argument2):this(argument1)
    {
    }
    //pozostałe konstruktory
}
```

W takim przypadku argument o nazwie *argument1* zostanie użyty zarówno w konstruktorze jednoargumentowym, jak i dwuargumentowym.

Niszczenie obiektu

Osoby, które programowały w językach obiektowych, takich jak np. C++ czy Object Pascal, zastanawiają się zapewne, jak w C# wygląda destruktor i kiedy zwalniamy pamięć zarezerwowaną dla obiektów. Skoro bowiem operator `new` pozwala na utworzenie obiektu, a tym samym na zarezerwowanie dla niego pamięci operacyjnej, logicznym założeniem jest, że po jego wykorzystaniu pamięć tę należy zwolnić. Ponieważ jednak takie podejście, tzn. zrzucenie na barki programistów konieczności zwalniania przydzielonej obiektom pamięci, powodowało powstawanie wielu błędów, w nowoczesnych językach programowania stosuje się inne rozwiązanie. Otóż za zwalnianie pamięci odpowiada środowisko uruchomieniowe, a programista praktycznie nie ma nad tym procesem kontroli[10].

Zajmuje się tym tak zwany **odśmiecacz** (ang. *garbage collector*), który czuwa nad optymalnym wykorzystaniem pamięci i uruchamia proces jej odzyskiwania w momencie, kiedy wolna ilość oddana do dyspozycji programu zbytnio się zmniejszy. Jest to wyjątkowo wygodne podejście dla programisty, zwalnia go bowiem z obowiązku zarządzania pamięcią. Zwiększa jednak narzuty czasowe związane z wykonaniem programu, wszak sam proces odśmiecania musi zająć czas procesora. Niemniej dzisiejsze środowiska uruchomieniowe są na tyle dopracowane, że w większości przypadków nie ma najmniejszej potrzeby zaprzątania myśli tym problemem.

Trzeba jednak zdawać sobie sprawę, że środowisko .NET jest w stanie automatycznie zarządzać wykorzystywaniem pamięci, ale tylko tej, która jest alokowana standardowo, czyli za pomocą operatora `new`. W nielicznych przypadkach, np. w sytuacji, gdyby stworzony przez nas obiekt wykorzystywał jakieś specyficzne zasoby, które nie mogą być zwolnione automatycznie, o posprzątanie systemu trzeba zadbać samodzielnie.

[10] Aczkolwiek wywołując metodę `System.GC.Collect()`, można wymusić zainicjowanie procesu odzyskiwania pamięci. Nie należy jednak tego wywołania nadużywać.

C# w tym celu wykorzystuje się **destruktory**[11], które są wykonywane zawsze, kiedy obiekt jest niszczony, usuwany z pamięci. Wystarczy więc, jeśli klasa będzie zawierała taki destruktor, a przy niszczeniu jej obiektu zostanie on wykonany. W ciele destruktora można wykonać dowolne instrukcje sprzątające. Destruktor deklaruje się tak jak konstruktor, z tą różnicą, że nazwę poprzedzamy znakiem tyldy, ogólnie:

```
class nazwa_klasy
{
  ~nazwa_klasy()
  {
    //kod destruktora
  }
}
```

Destruktora należy jednak używać tylko i wyłącznie w sytuacji, kiedy faktycznie niezbędne jest zwolnienie alokowanych niestandardowo zasobów. Nie należy natomiast umieszczać w kodzie pustych destruktorów, gdyż obniży to wydajność aplikacji[12].

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 16.1

Napisz klasę, której zadaniem będzie przechowywanie liczb typu `int`. Dołącz jednoargumentowy konstruktor przyjmujący argument typu `int`. Polu klasy nadaj nazwę `liczba`, tak samo nazwij argument konstruktora.

Ćwiczenie 16.2

Do klasy powstałej w ćwiczeniu 16.1 dopisz przeciążony konstruktor bezargumentowy ustawiający jej pole na wartość `-1`.

Ćwiczenie 16.3

Napisz klasę zawierającą dwa pola: pierwsze typu `double` i drugie typu `char`. Dopisz cztery przeciążone konstruktory: pierwszy przyjmujący jeden argument typu `double`, drugi przyjmujący jeden argument typu `char`, trzeci przyjmujący dwa argumenty — pierwszy typu `double`, drugi typu `char` — i czwarty przyjmujący również dwa argumenty — pierwszy typu `char`, drugi typu `double`.

Ćwiczenie 16.4

Zmień kod klasy powstałej w ćwiczeniu 16.3 tak, aby w konstruktorach dwuargumentowych były wykorzystywane konstruktory jednoargumentowe.

---

[11] W rzeczywistości destruktor jest tłumaczony wewnętrznie (przez kompilator) na wywołanie metody `Finalize` (co dodatkowo jest obejmowane blokiem obsługi sytuacji wyjątkowych), można więc z równie dobrym skutkiem umieścić zamiast niego w klasie taką metodę. Użycie destruktora wydaje się jednak czytelniejsze.

[12] Ze względu na specjalne traktowanie takich obiektów przez środowisko uruchomieniowe.

Ćwiczenie 16.5

Napisz kod klasy przechowującej dane określające prostokąt na płaszczyźnie; zapamiętywane mają być współrzędne lewego górnego rogu oraz prawego dolnego rogu. Do klasy dodaj jeden konstruktor przyjmujący cztery argumenty liczbowe, które będą określały współrzędne lewego górnego rogu oraz szerokość i wysokość prostokąta.

Dziedziczenie

Dziedziczenie to jeden z fundamentów programowania obiektowego. Umożliwia sprawne i łatwe wykorzystywanie już raz napisanego kodu czy budowanie hierarchii klas przejmujących swoje właściwości. Ten podrozdział zawiera trzy lekcje przybliżające temat dziedziczenia. W lekcji 17. zaprezentowane są podstawy, czyli sposoby tworzenia klas potomnych oraz zachowania konstruktorów klasy bazowej i potomnej. W lekcji 18. poruszony został temat specyfikatorów dostępu pozwalających na ustalanie praw dostępu do składowych klas. W lekcji 19. przedstawiono techniki przesłaniania pól i metod w klasach potomnych oraz składowe statyczne.

Lekcja 17. Klasy potomne

W lekcji 17. przedstawione zostały podstawy dziedziczenia, czyli budowania nowych klas na bazie już istniejących. Każda taka nowa klasa przejmuje zachowanie i właściwości klasy bazowej. Zobaczymy, jak tworzy się klasy potomne, jakie podstawowe zależności występują między klasą bazową a potomną oraz jak zachowują się konstruktory w przypadku dziedziczenia.

Dziedziczenie

Na początku lekcji 14. utworzyliśmy klasę `Punkt`, która przechowywała informację o współrzędnych punktu na płaszczyźnie. W trakcie dalszych ćwiczeń rozbudowaliśmy ją o dodatkowe metody, które pozwalały na ustawianie i pobieranie tych współrzędnych. Zastanówmy się teraz, co byśmy zrobili, gdybyśmy chcieli określać położenie punktu nie w dwóch, ale w trzech wymiarach, czyli gdyby do współrzędnych `x` i `y` trzeba było dodać współrzędną `z`. Pomysłem, który się od razu nasuwa, jest napisanie dodatkowej klasy, np. o nazwie `Punkt3D`, w postaci:

```
class Punkt3D
{
  int x;
  int y;
  int z;
}
```

Do tej klasy należałoby dalej dopisać pełny zestaw metod, które znajdowały się w klasie `Punkt`, takich jak `PobierzX`, `PobierzY`, `UstawX`, `UstawY` itd., oraz dodatkowe metody operujące na współrzędnej `z`. Zauważmy jednak, że w takiej sytuacji w dużej części po prostu powtarzamy już raz napisany kod. Czym bowiem będzie się różniła metoda `UstawX` klasy `Punkt` od metody `UstawX` klasy `Punkt3D`? Oczywiście niczym. Po prostu `Punkt3D` jest pewnego rodzaju rozszerzeniem klasy `Punkt`. Rozszerza ją o dodatkowe możliwości (pola, metody), pozostawiając stare właściwości bez zmian. Zamiast więc pisać całkiem od nowa klasę `Punkt3D`, lepiej spowodować, aby przejęła ona wszystkie możliwości klasy `Punkt`, wprowadzając dodatkowo swoje własne. Jest to tak zwane dziedziczenie, czyli jeden z fundamentów programowania obiektowego. Powiemy, że klasa `Punkt3D` dziedziczy z `Punkt`, czyli przejmuje jej składowe oraz dodaje swoje własne.

W C# dziedziczenie jest wyrażane za pomocą symbolu dwukropka, a cała definicja schematycznie wygląda następująco:

```
class klasa_potomna : klasa_bazowa
{
  //wnętrze klasy
}
```

Zapis taki oznacza, że klasa potomna dziedziczy z klasy bazowej. Zobaczmy, jak taka deklaracja będzie wyglądała w praktyce dla wspomnianych klas `Punkt` i `Punkt3D`. Jest to bardzo proste:

```
class Punkt3D : Punkt
{
  int z;
}
```

Taki zapis oznacza, że klasa `Punkt3D` przejęła wszystkie właściwości klasy `Punkt`, a dodatkowo otrzymała pole typu `int` o nazwie `z`. Przekonajmy się, że tak jest w istocie. Niech klasy `Punkt` i `Punkt3D` wyglądają tak, jak na listingu 3.30.

Listing 3.30. *Dziedziczenie pomiędzy klasami*

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

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

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

class Punkt3D : Punkt
{
  public int z;
}
```

Klasa `Punkt` ma tu postać znaną z wcześniejszych przykładów. Zawiera dwa pola, `x` i `y`, oraz sześć metod: `PobierzX` i `PobierzY` (zwracające współrzędne `x` i `y`), `UstawX`, `UstawY` i `UstawXY` (ustawiające współrzędne) oraz `WyswietlWspolrzedne` (wyświetlającą wartości pól `x` i `y` na ekranie). Ponieważ klasa `Punkt3D` dziedziczy z klasy `Punkt`, również zawiera wymienione pola i metody oraz dodatkowo pole o nazwie `z`. Nowością jest użycie przed każdą składową (polem lub metodą) słowa `public`. Oznacza ono, że składowe są dostępne publicznie, a więc można się do nich bezpośrednio odwoływać. Ta kwestia zostanie dokładniej wyjaśniona w kolejnej lekcji.

Kod z listingu 3.30 można zapisać w jednym pliku, np. o nazwie *Punkt.cs*, lub też w dwóch. Skorzystajmy z tej drugiej możliwości i zapiszmy kod klasy `Punkt` w pliku *Punkt.cs*, a klasy `Punkt3D` w pliku *Punkt3D.cs*. Napiszemy też teraz dodatkową klasę `Program`, widoczną na listingu 3.31, testującą obiekt klasy `Punkt3D`. Pozwoli to naocznie przekonać się, że na takim obiekcie zadziałają wszystkie metody, które znajdowały się w klasie `Punkt`.

Listing 3.31. *Testowanie klasy Punkt3D*

```
using System;

public class Program
{
  public static void Main()
  {
    Punkt3D punkt = new Punkt3D();

    Console.WriteLine("x = " + punkt.x);
    Console.WriteLine("y = " + punkt.y);
    Console.WriteLine("z = " + punkt.z);
    Console.WriteLine("");

    punkt.UstawX(100);
    punkt.UstawY(200);

    Console.WriteLine("x = " + punkt.x);
```

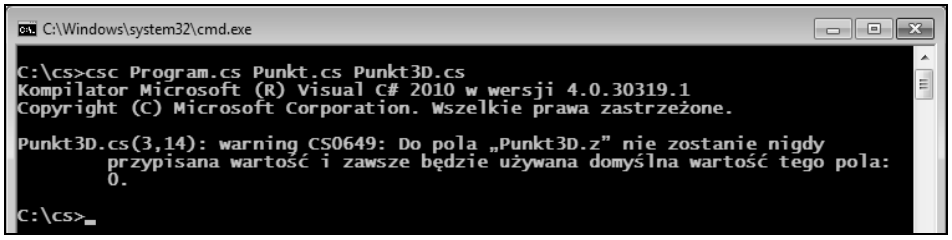
```
        Console.WriteLine("y = " + punkt.y);
        Console.WriteLine("z = " + punkt.z);
        Console.WriteLine("");

        punkt.UstawXY(300, 400);

        Console.WriteLine("x = " + punkt.x);
        Console.WriteLine("y = " + punkt.y);
        Console.WriteLine("z = " + punkt.z);
    }
}
```

Na początku definiujemy zmienną klasy `Punkt3D` o nazwie `punkt` i przypisujemy jej nowo utworzony obiekt klasy `Punkt3D`. Wykorzystujemy oczywiście dobrze nam znany operator `new`. Następnie wyświetlamy na ekranie wartości wszystkich pól tego obiektu. Wiemy, że są to trzy pola, `x`, `y`, `z`, oraz że powinny one otrzymać wartości domyślne równe `0` (por. tabela 3.1). Następnie wykorzystujemy metody `UstawX` oraz `UstawY`, aby przypisać polom `x` i `y` wartości `100` oraz `200`. W kolejnym kroku ponownie wyświetlamy zawartość wszystkich pól na ekranie. W dalszej części kodu wykorzystujemy metodę `UstawXY` do przypisania polu `x` wartości `300`, a polu `y` wartości `400` i jeszcze jeden raz wyświetlamy zawartość wszystkich pól na ekranie.

Możemy więc skompilować program. Ponieważ składa się on z trzech plików: *Program.cs*, *Punkt.cs* i *Punkt3D.cs*, w wierszu poleceń trzeba wydać komendę:

```
csc Program.cs Punkt.cs Punkt3D.cs
```

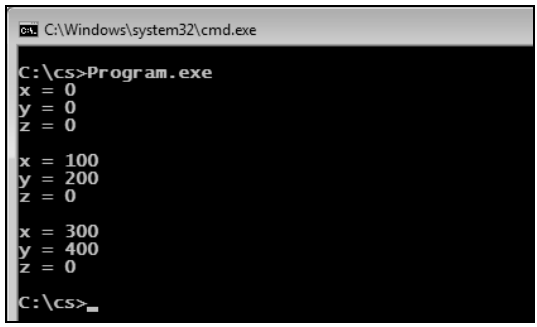
Kompilator wyświetli ostrzeżenie widoczne na rysunku 3.13 (w angielskiej wersji językowej kompilatora ma on brzmienie: `warning CS0649: Field 'Punkt3D.z' is never assigned to, and will always have its default value 0`). Jest to informacja o tym, że nie wykorzystujemy pola `z` i że będzie ono miało cały czas wartość `0`, czym oczywiście nie musimy się przejmować — jest to prawda, faktycznie nigdzie nie ustawiliśmy wartości pola `z`.



Rysunek 3.13. *Ostrzeżenie kompilatora o niezainicjalizowanym polu z*

Po uruchomieniu zobaczymy widok zaprezentowany na rysunku 3.14. Jest to też najlepszy dowód, że faktycznie klasa `Punkt3D` odziedziczyła wszystkie pola i metody klasy `Punkt`.

Klasa `Punkt3D` nie jest jednak w takiej postaci w pełni funkcjonalna, należałoby przecież dopisać metody operujące na nowym polu `z`. Na pewno przydatne będą: `UstawZ`, `PobierzZ` oraz `UstawXYZ`. Oczywiście metoda `UstawZ` będzie przyjmowała jeden argument



Rysunek 3.14. *Klasa Punkt3D przejęła pola i metody klasy Punkt*

typu `int` i przypisywała jego wartość polu `z`, metoda `pobierzZ` będzie zwracała wartość pola `z`, natomiast `ustawXYZ` będzie przyjmowała trzy argumenty typu `int` i przypisywała je polom `x`, `y` i `z`. Z pewnością nie jest żadnym zaskoczeniem, że metody te będą wyglądały tak, jak jest to zaprezentowane na listingu 3.32. Można się również zastanowić nad dopisaniem metod analogicznych do `ustawXY`, czyli metod `ustawXZ` oraz `ustawYZ`, to jednak będzie dobrym ćwiczeniem do samodzielnego wykonania.

Listing 3.32. *Metody operujące na polu z*

```
class Punkt3D : Punkt
{
  public int z;
  public void UstawZ(int wspZ)
  {
    z = wspZ;
  }
  public int PobierzZ()
  {
    return z;
  }
  public void UstawXYZ(int wspX, int wspY, int wspZ)
  {
    x = wspX;
    y = wspY;
    z = wspZ;
  }
}
```

Konstruktory klasy bazowej i potomnej

Klasom widocznym na listingach 3.30 i 3.32 brakuje konstruktorów. Przypomnijmy sobie, że w trakcie prac nad klasą `Punkt` powstały aż trzy konstruktory (listing 3.27 z lekcji 16.):

- bezargumentowy, ustawiający wartość wszystkich pól na `1`;
- dwuargumentowy, przyjmujący dwie wartości typu `int`;
- jednoargumentowy, przyjmujący obiekt klasy `Punkt`.

Można je z powodzeniem dopisać do kodu widocznego na listingu 3.30. Niestety, żaden z nich nie zajmuje się polem z, którego w klasie Punkt po prostu nie ma. Konstruktory dla klasy Punkt3D musimy więc napisać osobno. Nie jest to skomplikowane zadanie, zostały one zaprezentowane na listingu 3.33.

Listing 3.33. *Konstruktory dla klasy Punkt3D*

```
class Punkt3D : Punkt
{
  public int z;
  public Punkt3D()
  {
    x = 1;
    y = 1;
    z = 1;
  }
  public Punkt3D(int wspX, int wspY, int wspZ)
  {
    x = wspX;
    y = wspY;
    z = wspZ;
  }
  public Punkt3D(Punkt3D punkt)
  {
    x = punkt.x;
    y = punkt.y;
    z = punkt.z;
  }
  /*
  ...pozostałe metody klasy Punkt3D...
  */
}
```

Jak widać, pierwszy konstruktor nie przyjmuje żadnych argumentów i przypisuje wszystkim polom wartość 1. Konstruktor drugi przyjmuje trzy argumenty: wspX, wspY oraz wspZ, wszystkie typu int, i przypisuje otrzymane wartości polom x, y i z. Konstruktor trzeci otrzymuje jako argument obiekt klasy Punkt3D i kopiuje z niego wartości pól. Oczywiście, pozostałe metody klasy Punkt3D pozostają bez zmian, nie zostały one uwzględnione na listingu, aby niepotrzebnie nie powielać prezentowanego już kodu (są one natomiast uwzględnione na listingach znajdujących się na płycie CD oraz na FTP).

Jeśli przyjrzymy się dokładnie napisanym właśnie konstruktorom, zauważymy z pewnością, że w znacznej części ich kod dubluje się z kodem konstruktorów klasy Punkt. Dokładniej są to te same instrukcje, uzupełnione dodatkowo o instrukcje operujące na wartościach pola z. Spójrzmy, konstruktory:

```
Punkt3D(int wspX, int wspY, int wspZ)
{
  x = wspX;
  y = wspY;
  z = wspZ;
}
```

oraz:

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

są przecież prawie identyczne! Jedyna różnica to dodatkowy argument i dodatkowa instrukcja przypisująca jego wartość polu `z`. Czy nie lepiej byłoby zatem wykorzystać konstruktor klasy `Punkt` w klasie `Punkt3D` lub ogólniej — konstruktor klasy bazowej w konstruktorze klasy potomnej? Oczywiście, że tak. Nie można jednak wywołać konstruktora tak jak zwyczajnej metody — do tego celu służy specjalna konstrukcja ze słowem `base`, o ogólnej postaci:

```
class klasa_potomna : klasa_bazowa
{
  klasa_potomna(argumenty):base(argumenty)
  {
    /*
    ...kod konstruktora...
    */
  }
}
```

Zauważmy, że bardzo przypomina to opisaną wcześniej składnię ze słowem `this`. Różnica jest taka, że `this` służy do wywoływania konstruktorów w ramach jednej klasy, a `base` do wywoływania konstruktorów klasy bazowej. Jeśli zatem w klasie `Punkt` będą istniały konstruktory takie jak widoczne na listingu 3.34, to będzie można je wywoływać w klasie `Punkt3D` w sposób zaprezentowany na listingu 3.35.

Listing 3.34. *Konstruktory w klasie Punkt*

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

  public Punkt()
  {
    x = 1;
    y = 1;
  }
  public Punkt(int wspX, int wspY)
  {
    x = wspX;
    y = wspY;
  }
  public Punkt(Punkt punkt)
  {
    x = punkt.x;
    y = punkt.y;
  }
  /*
  ...dalszy kod klasy Punkt...
  */
}
```

Listing 3.35. *Wywołanie konstruktorów klasy bazowej*

```
class Punkt3D : Punkt
{
  public int z;
  public Punkt3D():base()
  {
    z = 1;
  }
  public Punkt3D(int wspX, int wspY, int wspZ):base(wspX, wspY)
  {
    z = wspZ;
  }
  public Punkt3D(Punkt3D punkt):base(punkt)
  {
    z = punkt.z;
  }
  /*
  ...pozostałe metody klasy Punkt3D...
  */
}
```

W pierwszym konstruktorze występuje ciąg `base()`, co powoduje wywołanie bezargumentowego konstruktora klasy bazowej. Taki konstruktor (bezargumentowy) istnieje w klasie `Punkt`, konstrukcja ta nie budzi więc żadnych wątpliwości. W konstruktorze drugim w nawiasie za `base` występują dwa argumenty typu `int` (`base(wspX, wspY)`). Ponieważ w klasie `Punkt` istnieje konstruktor dwuargumentowy, przyjmujący dwie wartości typu `int`, również i ta konstrukcja jest jasna — zostanie on wywołany i będą mu przekazane wartości `wspX` i `wspY` przekazane w wywołaniu konstruktora klasy `Punkt3D`.

Konstruktor trzeci przyjmuje jeden argument typu (klasy) `Punkt3D` i przekazuje go jako argument w wywołaniu `base` (`base(punkt)`). W klasie `Punkt` istnieje konstruktor przyjmujący jeden argument klasy… no właśnie, w klasie `Punkt` przecież wcale nie ma konstruktora, który przyjmowałby argument tego typu! Jest co prawda konstruktor:

```
Punkt(Punkt punkt)
{
  //instrukcje konstruktora
}
```

ale przecież przyjmuje on argument typu `Punkt`, a nie `Punkt3D`. Tymczasem klasa z listingu 3.35 skompiluje się bez żadnych problemów! Jak to możliwe? Przecież nie zgadzają się typy argumentów! Otóż okazuje się, że jeśli oczekujemy argumentu klasy `X`, a podany zostanie argument klasy `Y`, która jest klasą potomną dla `X`, błędu nie będzie. W takiej sytuacji nastąpi tak zwane rzutowanie typu obiektu, czym jednak zajmiemy się dokładniej dopiero w rozdziale 6. Na razie wystarczy zapamiętać zasadę: w miejscu, gdzie powinien być zastosowany obiekt pewnej klasy `X`, można zastosować również obiekt klasy potomnej dla `X`[13].

[13] Istnieją wszakże sytuacje, kiedy nie będzie to możliwe. Ten temat zostanie poruszony w dalszej części książki.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 17.1

Zmodyfikuj kod klasy `Punkt` z listingu 3.30 w taki sposób, aby nazwy parametrów w metodach `UstawX`, `UstawY` oraz `UstawXY` miały takie same nazwy jak nazwy pól, czyli `x` i `y`. Zatem nagłówki tych metod mają wyglądać następująco:

```
void UstawX(int x)
void UstawY(int y)
void UstawXY(int x, int y)
```

Ćwiczenie 17.2

Dopisz do klasy `Punkt3D` zaprezentowanej na listingu 3.32 metodę `UstawXZ` oraz `UstawYZ`.

Ćwiczenie 17.3

Napisz przykładową klasę `Program` wykorzystującą wszystkie trzy konstruktory klasy `Punkt3D` z listingu 3.33.

Ćwiczenie 17.4

Zmodyfikuj kod z listingu 3.33 w taki sposób, aby w żadnym z konstruktorów nie występowało bezpośrednie przypisanie wartości do pól klasy. Możesz użyć metody `UstawXYZ`.

Ćwiczenie 17.5

Napisz kod klasy `KolorowyPunkt` będącej rozszerzeniem klasy `Punkt` o informację o kolorze. Kolor ma być określany dodatkowym polem o nazwie `kolor` i typie `int`. Dopisz metody `UstawKolor` i `PobierzKolor`, a także odpowiednie konstruktory.

Ćwiczenie 17.6

Dopisz do klasy `Punkt3D` z listingu 3.35 konstruktor, który jako argument będzie przyjmował obiekt klasy `Punkt`. Wykorzystaj w tym konstruktorze wywołanie `base`.

Lekcja 18. Modyfikatory dostępu

Modyfikatory dostępu (nazywane również specyfikatorami dostępu, ang. *access modifiers*) pełnią ważną funkcję w C#, pozwalają bowiem na określenie praw dostępu do składowych klas, a także do samych klas. Występują one w kilku rodzajach, które zostaną przedstawione właśnie w lekcji 18.

Określanie reguł dostępu

W dotychczasowych naszych programach zarówno przed słowem `class`, jak i przed niektórymi składowymi, pojawiało się czasem słowo `public`. Jest to tak zwany **specyfikator** lub **modyfikator dostępu** i oznacza, że dana klasa jest publiczna, czyli że mogą z niej korzystać (mogą się do niej odwoływać) wszystkie inne klasy. Każda klasa, pole oraz metoda[14] mogą być:

- publiczne (`public`),
- chronione (`protected`),
- wewnętrzne (`internal`),
- wewnętrzne chronione (`protected internal`),
- prywatne (`private`).

Typowa klasa, czyli o takiej postaci jak dotychczas stosowana, np.:

```
class Punkt
{
}
```

może być albo publiczna (`public`), albo wewnętrzna (`internal`)[15]. Domyślnie jest wewnętrzna, czyli dostęp do niej jest możliwy w obrębie jednego zestawu (por. lekcja 14.). Dopuszczalna jest zmiana sposobu dostępu na publiczny przez użycie słowa `public`:

```
public class Punkt
{
}
```

Użycie słowa `public` oznacza zniesienie wszelkich ograniczeń w dostępie do klasy (ale już nie do jej składowych, dostęp do składowych klasy definiuje się osobno). W tej fazie nauki różnice nie są jednak istotne, gdyż i tak korzystamy zawsze z jednego zestawu tworzącego konkretny program, a więc użycie bądź nieużycie słowa `public` przy klasie nie wywoła żadnych negatywnych konsekwencji.

W przypadku składowych klas obowiązują następujące zasady. Publiczne składowe określa się słowem `public`, co oznacza, że wszyscy mają do nich dostęp oraz że są dziedziczone przez klasy pochodne. Do składowych prywatnych (`private`) można dostać się tylko z wnętrza danej klasy, natomiast do składowych chronionych (`protected`) można uzyskać dostęp z wnętrza danej klasy oraz klas potomnych. Znaczenie tych specyfikatorów dostępu jest praktycznie takie samo jak w innych językach obiektowych, na przykład w Javie.

W C# do dyspozycji są jednak dodatkowo specyfikatory `internal` i `protected internal`. Słowo `internal` oznacza, że dana składowa klasy będzie dostępna dla wszystkich klas

[14] Dotyczy to także struktur, interfejsów, wyliczeń i delegacji. Te zagadnienia będą omawiane w dalszej części książki.

[15] Stosowanie pozostałych modyfikatorów jest możliwe w przypadku klas wewnętrznych (zagnieżdżonych), które zostaną omówione w rozdziale 6.

z danego zestawu (por. lekcja 14.). Z kolei `protected internal`, jak łatwo się domyślić, jest kombinacją `protected` oraz `internal` i oznacza, że dostęp do składowej mają zarówno klasy potomne, jak i klasy z danego zestawu. Niemniej tymi dwoma specyfikatorami nie będziemy się zajmować, przyjrzymy się za to bliżej modyfikatorom `public`, `private` i `protected`.

Jeśli przed daną składową nie wystąpi żaden modyfikator, to będzie ona domyślnie prywatna. To właśnie dlatego w niektórych dotychczasowych przykładach poziom dostępu był zmieniany na publiczny, tak aby do składowych można się było odwoływać z innych klas.

Specyfikator dostępu należy umieścić przed nazwą typu, co schematycznie wygląda następująco:

```
specyfikator_dostępu nazwa_typu nazwa_pola
```

Podobnie jest z metodami — specyfikator dostępu powinien być pierwszym elementem deklaracji, czyli ogólnie napiszemy:

```
specyfikator_dostępu typ_zwracany nazwa_metody(argumenty)
```

Znaczenie modyfikatorów w przypadku określania reguł dostępu do całych klas jest podobne, z tym zastrzeżeniem, że modyfikatory `protected` i `private` mogą być stosowane tylko w przypadku klas zagnieżdżonych (patrz lekcja 32.). Domyślnym poziomem dostępu (czyli gdy przed jej nazwą nie występuje żadne określenie reguł dostępu) do klasy jest `internal`.

Dostęp publiczny — public

Jeżeli dana składowa klasy jest publiczna, oznacza to, że mają do niej dostęp wszystkie inne klasy, czyli dostęp ten nie jest w żaden sposób ograniczony. Weźmy np. pierwotną wersję klasy `Punkt` z listingu 3.1 (lekcja 14.). Gdyby pola `x` i `y` tej klasy miały być publiczne, musiałaby ona wyglądać tak, jak na listingu 3.36.

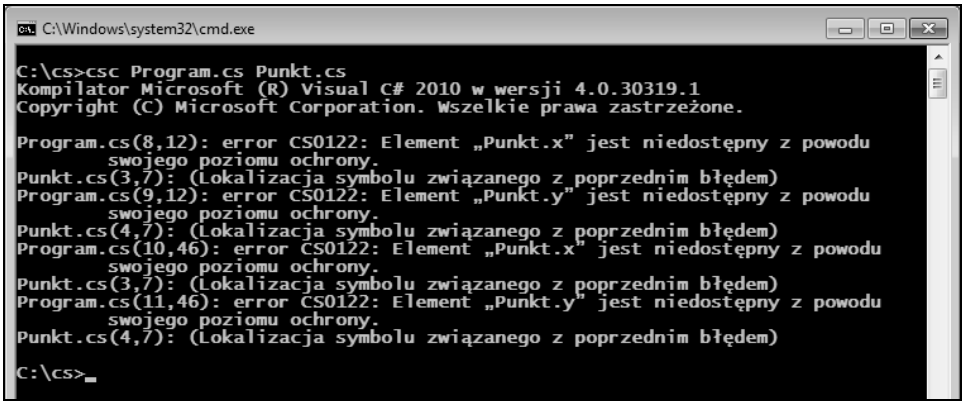
Listing 3.36. *Publiczne składowe klasy Punkt*

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

O tym, że poziom dostępu do pól tej klasy zmienił się, można się przekonać w prosty sposób. Użyjmy klasy `Program` z listingu 3.9 i klasy `Punkt` z listingu 3.1. Tworzony jest tam obiekt klasy `Punkt`, jego polom `x` i `y` są przypisywane wartości `100` i `200`, a następnie są one odczytywane i wyświetlane na ekranie. Próba kompilacji takiego zestawu klas się nie uda. Po wydaniu w wierszu poleceń komendy:

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

zakończy się błędem kompilacji widocznym na rysunku 3.15. Nic w tym dziwnego, skoro domyślny poziom dostępu nie pozwala klasie `Program` na bezpośrednie odwoływanie się do składowych klasy `Punkt` (zgodnie z podanym wyżej opisem domyślnie składowe klasy są prywatne).



Rysunek 3.15. *Próba dostępu do prywatnych składowych kończy się błędami kompilacji*

Zupełnie inaczej będzie w przypadku tej samej klasy `Program` oraz klasy `Punkt` z listingu 3.36. Ponieważ w takim przypadku dostęp do pól `x` i `y` będzie publiczny, program uda się skompilować bez problemów.

Warto przy tym wspomnieć, że zaleca się, aby dostęp do pól klasy nie był publiczny, a ich odczyt i modyfikacja odbywały się poprzez odpowiednio zdefiniowane metody. Dlaczego tak jest, zostanie pokazane w dalszej części lekcji. Gdybyśmy chcieli dopisać do klasy `Punkt` z listingu 3.36 publiczne wersje metod `PobierzX`, `PobierzY`, `UstawX` i `UstawY`, przyjęłaby ona postać widoczną na listingu 3.37.

Listing 3.37. *Publiczne pola i metody klasy Punkt*

```
class Punkt
{
  public int x;
  public int y;
  public int PobierzX()
  {
    return x;
  }
  public int PobierzY()
  {
    return y;
  }
  public void UstawX(int wspX)
  {
    x = wspX;
  }
  public void UstawY(int wspY)
  {
```

```
    y = wspY;
  }
}
```

Gdyby natomiast klasa `Punkt` z listingu 3.36 miała być publiczna, to wyglądałaby tak jak na listingu 3.38. Z reguły główne klasy określane są jako publiczne, tak aby można było się do nich dowolnie odwoływać, natomiast klasy pomocnicze, usługowe wobec klasy głównej, określane są jako wewnętrzne (`internal`), tak aby dostęp do nich był jedynie z wnętrza danego zestawu.

Listing 3.38. *Publiczna klasa Punkt*

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

Dostęp prywatny — private

Składowe oznaczone słowem `private` to takie, które są dostępne jedynie z wnętrza danej klasy. To znaczy, że wszystkie metody danej klasy mogą je dowolnie odczytywać i zapisywać, natomiast dostęp z zewnątrz jest zabroniony zarówno dla zapisu, jak i odczytu. Jeżeli zatem w klasie `Punkt` z listingu 3.36 zechcemy jawnie ustawić oba pola jako prywatne, będzie ona miała postać widoczną na listingu 3.39.

Listing 3.39. *Klasa Punkt z prywatnymi polami*

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

O tym, że dostęp spoza klasy został zabroniony, przekonamy się, próbując dokonać kompilacji podobnej do tej w poprzednim podpunkcie, tzn. używając klasy `Program` z listingu 3.9 i klasy `Punkt` z listingu 3.39. Efekt będzie taki sam jak na rysunku 3.15. Tak więc do składowych prywatnych na pewno nie można się odwołać spoza klasy, w której zostały zdefiniowane. Ta uwaga dotyczy również klas potomnych.

W jaki zatem sposób odwołać się do pola prywatnego? Przypomnijmy opis prywatnej składowej klasy: jest to taka składowa, która jest dostępna z wnętrza danej klasy, czyli dostęp do niej mają wszystkie metody klasy. Wystarczy zatem, jeśli napiszemy publiczne metody pobierające i ustawiające pola prywatne, a będziemy mogli wykonywać na nich operacje. W przypadku klasy `Punkt` z listingu 3.39 niezbędne byłyby metody `UstawX`, `UstawY`, `PobierzX` i `PobierzY`. Klasa `Punkt` zawierająca prywatne pola `x` i `y` oraz wymienione metody o dostępie publicznym została przedstawiona na listingu 3.40.

Listing 3.40. *Dostęp do prywatnych pól za pomocą publicznych metod*

```
class Punkt
{
  private int x;
  private int y;
  public int PobierzX()
  {
    return x;
  }
  public int PobierzY()
  {
    return y;
  }
  public void UstawX(int wspX)
  {
    x = wspX;
  }
  public void UstawY(int wspY)
  {
    y = wspY;
  }
}
```

Takie metody pozwolą nam już na bezproblemowe odwoływanie się do obu prywatnych pól. Teraz program z listingu 3.9 trzeba by poprawić tak, aby wykorzystywał nowe metody, czyli zamiast:

```
punkt.x = 100;
```

napiszemy:

```
punkt.UstawX(100);
```

a zamiast:

```
Console.WriteLine("punkt.x = " + punkt.x);
```

napiszemy:

```
Console.WriteLine("punkt.x = " + punkt.PobierzX());
```

Podobne zmiany trzeba będzie wprowadzić w przypadku dostępu do pola `y`.

Dostęp chroniony — protected

Składowe klasy oznaczone słowem `protected` to składowe chronione. Są one dostępne jedynie dla metod danej klasy oraz klas potomnych. Oznacza to, że jeśli mamy przykładową klasę `Punkt`, w której znajdzie się chronione pole o nazwie `x`, to w klasie `Punkt3D`, o ile jest ona klasą pochodną od `Punkt`, również będziemy mogli odwoływać się do pola `x`. Jednak dla każdej innej klasy, która nie dziedziczy z `Punkt`, pole `x` będzie niedostępne. W praktyce klasa `Punkt` — z polami `x` i `y` zadeklarowanymi jako chronione — będzie wyglądała tak, jak na listingu 3.41.

Listing 3.41. *Chronione pola w klasie Punkt*

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

Jeśli teraz z klasy `Punkt` wyprowadzimy klasę `Punkt3D` w postaci widocznej na listingu 3.42, to będzie ona miała (odmiennie niż byłoby to w przypadku składowych prywatnych) pełny dostęp do składowych `x` i `y` klasy `Punkt`.

Listing 3.42. *Klasa dziedzicząca z Punkt*

```
class Punkt3D : Punkt
{
  protected int z;
}
```

Dlaczego ukrywamy wnętrze klasy?

W tym miejscu pojawi się zapewne pytanie, dlaczego chcemy zabraniać bezpośredniego dostępu do niektórych składowych klas, stosując modyfikatory `private` i `protected`. Otóż chodzi o ukrycie implementacji wnętrza klasy. Programista, projektując daną klasę, udostępnia na zewnątrz (innym programistom) pewien interfejs służący do posługiwania się jej obiektami. Określa więc sposób, w jaki można korzystać z danego obiektu. To, co znajduje się we wnętrzu, jest ukryte; dzięki temu można całkowicie zmienić wewnętrzną konstrukcję klasy, nie zmieniając zupełnie sposobu korzystania z niej.

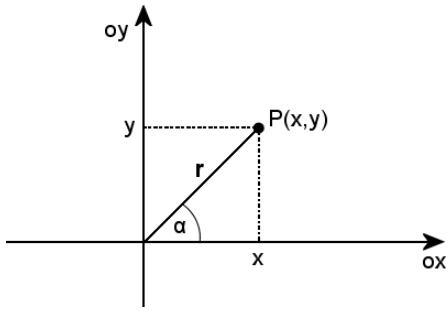
To, że takie podejście może nam się przydać, można pokazać nawet na przykładzie tak prostej klasy, jaką jest klasa `Punkt`. Załóżmy, że ma ona postać widoczną na listingu 3.40. Pola `x` i `y` są prywatne i zabezpieczone przed dostępem z zewnątrz, operacje na współrzędnych możemy wykonywać wyłącznie dzięki publicznym metodom: `PobierzX`, `PobierzY`, `UstawX`, `UstawY`. Program przedstawiony na listingu 3.43 będzie zatem działał poprawnie.

Listing 3.43. *Program korzystający z klasy Punkt*

```
using System;

public class Program
{
  public static void Main()
  {
    Punkt punkt1 = new Punkt();
    punkt1.UstawX(100);
    punkt1.UstawY(200);
    Console.WriteLine("punkt1.x = " + punkt1.PobierzX());
```

```
      Console.WriteLine("punkt1.y = " + punkt1.PobierzY());
    }
  }
```

Załóżmy teraz, że zostaliśmy zmuszeni (obojętnie, z jakiego powodu) do zmiany sposobu reprezentacji współrzędnych na tak zwany układ biegunowy, w którym położenie punktu jest opisywane za pomocą dwóch parametrów: kąta α oraz odległości od początku układu współrzędnych (rysunek 3.16). W klasie `Punkt` nie będzie już zatem pól `x` i `y`, przestaną mieć więc sens wszelkie odwołania do nich. Gdyby pola te były zadeklarowane jako publiczne, spowodowałoby to spory problem. Nie dość, że we wszystkich programach wykorzystujących klasę `Punkt` trzeba by zmieniać odwołania, to dodatkowo należałoby w każdym takim miejscu dokonywać przeliczania współrzędnych. Wymagałoby to wykonania ogromnej pracy, a ponadto pojawiłoby się w ten sposób sporo możliwości powstania niepotrzebnych błędów.



Rysunek 3.16. *Położenie punktu reprezentowane za pomocą współrzędnych biegunowych*

Jednak dzięki temu, że pola `x` i `y` są prywatne, a dostęp do nich odbywa się przez publiczne metody, wystarczy, że tylko odpowiednio zmienimy te metody. Jak się za chwilę okaże, można całkowicie tę klasę przebudować, a korzystający z niej program z listingu 3.43 nie będzie wymagał nawet najmniejszej poprawki.

Najpierw trzeba zamienić pola `x` i `y` typu `int` na pola reprezentujące kąt i odległość. Kąt najlepiej reprezentować za pomocą jego funkcji trygonometrycznej — wybierzmy np. sinus. Nowe pola nazwiemy więc `sinusalfa` oraz `r` (będzie reprezentowało odległość punktu od początku układu współrzędnych). Zatem podstawowa wersja nowej klasy `Punkt` będzie miała postać:

```
public class Punkt
{
  private double sinusalfa;
  private double r;
}
```

Dopisać należy teraz wszystkie cztery metody pierwotnie operujące na polach `x` i `y`. Aby to zrobić, musimy znać wzory przekształcające wartości współrzędnych kartezjańskich (tzn. współrzędne (x, y)) na układ biegunowy (czyli kąt i moduł) oraz wzory odwrotne, czyli przekształcające współrzędne biegunowe na kartezjańskie. Wyprowadzenie tych wzorów nie jest skomplikowane, wystarczy znajomość podstawowych funkcji

trygonometrycznych oraz twierdzenia Pitagorasa. Jednak książka ta to kurs programowania, a nie lekcja matematyki, wzory zostaną więc przedstawione już w gotowej postaci[16]. I tak (dla oznaczeń jak na rysunku 3.16):

$$x = r \times \sqrt{1 - \sin^2(\alpha)}$$

$$y = r \times \sin(\alpha)$$

oraz:

$$r = \sqrt{x^2 + y^2}$$

$$\sin(\alpha) = \frac{y}{r}$$

Mając te dane, możemy przystąpić do napisania odpowiednich metod. Zacznijmy od metody `PobierzY`. Jej postać będzie następująca:

```
public int PobierzY()
{
  double y = r * sinusalfa;
  return (int) y;
}
```

Deklarujemy zmienną pomocniczą `y` typu `double` i przypisujemy jej wynik mnożenia wartości pól `r` oraz `sinusalfa` — zgodnie z podanymi wyżej wzorami. Ponieważ metoda ma zwrócić wartość `int`, a wynikiem obliczeń jest wartość `double`, przed zwróceniem wyniku dokonujemy konwersji na typ `int`. Odpowiada za to konstrukcja `(int) y`[17,18]. W analogiczny sposób napiszemy metodę `PobierzX`, choć będziemy musieli oczywiście wykonać nieco więcej obliczeń. Metoda ta wygląda następująco:

```
public int PobierzX()
{
  double x = r * Math.Sqrt(1 - sinusalfa * sinusalfa);
  return (int) x;
}
```

Tym razem deklarujemy, podobnie jak w poprzednim przypadku, pomocniczą zmienną `x` typu `double` oraz przypisujemy jej wynik działania: `r * Math.Sqrt(1 - sinusalfa * sinusalfa)`. `Math.Sqrt` — to standardowa metoda obliczająca pierwiastek kwadratowy z przekazanego jej argumentu (czyli np. wykonanie instrukcji `Math.sqrt(4)` da

[16] W celu uniknięcia umieszczania w kodzie klasy dodatkowych instrukcji warunkowych, zaciemniających sedno zagadnienia, przedstawiony kod i wzory są poprawne dla dodatnich współrzędnych `x`. Uzupełnienie klasy `Punkt` w taki sposób, aby możliwe było także korzystanie z ujemnych wartości `x`, można potraktować jako ćwiczenie do samodzielnego wykonania.

[17] Nie jest to sposób w pełni poprawny, gdyż pozbywamy się zupełnie części ułamkowej, zamiast wykonać prawidłowe zaokrąglenie, a w związku z tym w wynikach mogą się pojawić drobne nieścisłości. Żeby jednak nie zaciemniać przedstawianego zagadnienia dodatkowymi instrukcjami, musimy się z tą drobną niedogodnością pogodzić.

[18] W tej instrukcji jest wykonywane tzw. rzutowanie typu; temat ten zostanie jednak omówiony dokładnie dopiero w lekcji 27., w rozdziale 6.

w wyniku 2) — wykorzystywaliśmy ją już w programach rozwiązujących równania kwadratowe. W tym przypadku ten argument to `1 - sinusalfa * sinusalfa`, czyli `1 - sinusalfa`^{2}, zgodnie z podanym wzorem na współrzędną `x`. Wykonujemy mnożenie zamiast potęgowania, gdyż jest ono po prostu szybsze i wygodniejsze.

Pozostały jeszcze do napisania metody `UstawX` i `UstawY`. Pierwsza z nich będzie mieć następującą postać:

```
public void UstawX(int wspX)
{
  int x = wspX;
  int y = PobierzY();

  r = Math.Sqrt(x * x + y * y);
  sinusalfa = y / r;
}
```

Ponieważ zarówno parametr `r`, jak i `sinusalfa` zależą od obu współrzędnych, trzeba je najpierw uzyskać. Współrzędna `x` jest oczywiście przekazywana jako argument, natomiast `y` uzyskujemy, wywołując napisaną przed chwilą metodę `PobierzY`. Dalsza część metody `UstawX` to wykonanie działań zgodnych z podanymi wzorami[19]. Podobnie jak w przypadku `PobierzY`, zamiast potęgowania wykonujemy zwykłe mnożenie `x * x` i `y * y`. Metoda `UstawY` ma prawie identyczną postać, z tą różnicą, że skoro będzie jej przekazywana wartość współrzędnej `y`, to musimy uzyskać jedynie wartość `x`, czyli początkowe instrukcje będą następujące:

```
int x = PobierzX();
int y = wspY;
```

Kiedy złożymy wszystkie napisane do tej pory elementy w jedną całość, uzyskamy klasę `Punkt` w postaci widocznej na listingu 3.44 (na początku została dodana dyrektywa `using`, tak aby można było swobodnie odwoływać się do klasy `Math` zdefiniowanej w przestrzeni nazw `System`). Jeśli teraz uruchomimy program z listingu 3.43, przekonamy się, że wynik jego działania z nową klasą `Punkt` jest taki sam jak w przypadku jej poprzedniej postaci. Mimo całkowitej wymiany wnętrza klasy `Punkt` program zadziała tak, jakby nic się nie zmieniło.

Listing 3.44. *Nowa wersja klasy Punkt*

```
using System;

class Punkt
{
  private double sinusalfa;
  private double r;

  public int PobierzX()
  {
    double x = r * Math.Sqrt(1 - sinusalfa * sinusalfa);
    return (int) x;
```

[19] Jak można zauważyć, taki kod nie będzie działał poprawnie dla punktu o współrzędnych (0,0). Niezbędne byłoby wprowadzenie dodatkowych instrukcji warunkowych.

```
  }

  public int PobierzY()
  {
    double y = r * sinusalfa;
    return (int) y;
  }

  public void UstawX(int wspX)
  {
    int x = wspX;
    int y = PobierzY();

    r = Math.Sqrt(x * x + y * y);
    sinusalfa = y / r;
  }

  public void UstawY(int wspY)
  {
    int x = PobierzX();
    int y = wspY;

    r = Math.Sqrt(x * x + y * y);
    sinusalfa = y / r;
  }
}
```

Jak zabronić dziedziczenia?

W praktyce programistycznej można spotkać się z sytuacjami, kiedy konieczne jest zabronienie dziedziczenia. Innymi słowy będziemy chcieli spowodować, aby z naszej klasy nie można było wyprowadzać klas potomnych. Służy do tego słowo kluczowe `sealed`, które należy umieścić przed nazwą klasy zgodnie ze schematem:

```
specyfikator_dostępu sealed class nazwa_klasy
{
  //składowe klasy
}
```

Nie ma przy tym formalnego znaczenia to, czy słowo `sealed` będzie przed czy za specyfikatorem dostępu, czyli czy napiszemy np. `public sealed class`, czy `sealed public class`, niemniej dla przejrzystości i ujednolicenia notacji najlepiej konsekwentnie stosować tylko jeden przedstawionych sposobów.

Przykładowa klasa `Wartosc` tego typu została przedstawiona na listingu 3.45.

Listing 3.45. *Zastosowanie modyfikatora sealed*

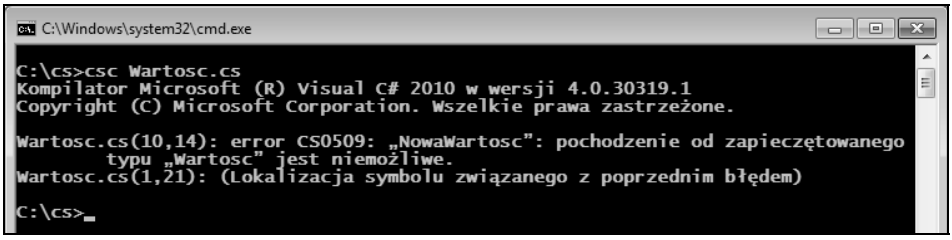
```
public sealed class Wartosc
{
  public int liczba;
  public void Wyswietl()
```

```
  {
    System.Console.WriteLine(liczba);
  }
}
```

Z takiej klasy nie można wyprowadzić żadnej innej. Tak więc przedstawiona na listingu 3.46 klasa `NowaWartosc` dziedzicząca z `Wartosc` jest niepoprawna. Kompilator C# nie dopuści do kompilacji takiego kodu i zgłosi komunikat o błędzie zaprezentowany na rysunku 3.17.

Listing 3.46. *Niepoprawne dziedziczenie*

```
public class NowaWartosc : Wartosc
{
  public int drugaLiczba;
  /*
  ... dalsze składowe klasy ...
  */
}
```



Rysunek 3.17. *Próba nieprawidłowego dziedziczenia kończy się błędem kompilacji*

Tylko do odczytu

W C# można zadeklarować pole klasy jako tylko do odczytu, co oznacza, że przypisanej mu wartości nie można będzie zmieniać. Takie pola oznacza się modyfikatorem `readonly`, który musi wystąpić przed nazwą typu, schematycznie:

```
specyfikator_dostępu readonly typ_pola nazwa_pola;
```

lub

```
readonly specyfikator_dostępu typ_pola nazwa_pola;
```

Tak więc poprawne będą poniższe przykładowe deklaracje:

```
readonly int liczba;
readonly public int liczba;
public readonly int liczba;
```

Wartość takiego pola może być przypisana albo w momencie deklaracji, albo też w konstruktorze klasy i nie może być później zmieniana.

Pola readonly typów prostych

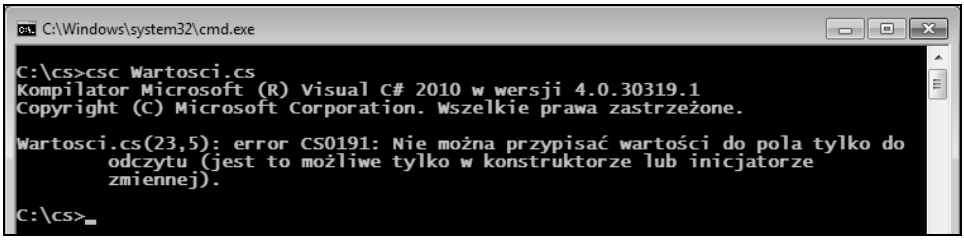
Przykładowa klasa zawierająca pola tylko do odczytu została przedstawiona na listingu 3.47.

Listing 3.47. *Klasa zawierająca pola tylko do odczytu*

```
public class Wartosci
{
  public readonly int liczba1 = 100;
  public readonly int liczba2;
  public int liczba3;
  public Wartosci()
  {
    //prawidłowo: inicjalizacja pola liczba2
    liczba2 = 200;

    //prawidłowo: można zmienić wartość pola w konstruktorze
    liczba1 = 150;
  }
  public void Obliczenia()
  {
    //prawidłowo: odczyt pola liczba1, zapis pola liczba3
    liczba3 = 2 * liczba1;

    //prawidłowo: odczyt pól liczba1 i liczba2, zapis pola liczba3
    liczba3 = liczba2 + liczba1;

    //nieprawidłowo: niedozwolony zapis pola liczba1
    //liczba1 = liczba2 / 2;

    System.Console.WriteLine(liczba1);
    System.Console.WriteLine(liczba2);
    System.Console.WriteLine(liczba3);
  }
  public static void Main()
  {
    Wartosci w = new Wartosci();
    w.Obliczenia();
  }
}
```

Zostały tu zadeklarowane trzy pola, `liczba1`, `liczba2 i liczba3`, wszystkie publiczne o typie `int`. Dwa pierwsze są również polami tylko do odczytu, a zatem przypisanych im wartości nie wolno modyfikować poza konstruktorem. W klasie znalazły się również konstruktor oraz metoda `Obliczenia`, która wykonuje działania, wykorzystując wartości przypisane zadeklarowanym polom. W konstruktorze polu `liczba2` została przypisana wartość `200`, a polu `liczba1` wartość `150`. Oba przypisania są prawidłowe, mimo że `liczba1` miało już ustaloną wcześniej wartość. W konstruktorze można bowiem przypisać nową wartość polu tylko do odczytu i jest to jedyne miejsce, w którym taka operacja jest prawidłowa.

W metodzie `Obliczenia` najpierw zmiennej `liczba3` przypisujemy wynik mnożenia `2 * liczba1`. Jest to poprawna instrukcja, gdyż wolno odczytywać wartość pola tylko do odczytu `liczba1` oraz przypisywać wartości zwykłemu polu `liczba3`. Podobną sytuację mamy w przypadku drugiego działania. Trzecia instrukcja przypisania została ujęta w komentarz, gdyż jest nieprawidłowa i spowodowałaby błąd kompilacji widoczny na rysunku 3.18. Występuje tu bowiem próba przyporządkowania wyniku działania `liczba2 / 2` polu `liczba1`, w stosunku do którego został użyty modyfikator `readonly`. Takiej operacji nie wolno wykonywać, zatem po usunięciu znaków komentarza z tej instrukcji kompilator zaprotestuje. Do klasy `Wartosci` dopisana została też metoda `Main` (por. lekcja 15.), w której tworzymy nowy obiekt klasy `Wartosci` i wywołujemy jego metodę `Obliczenia`.



Rysunek 3.18. *Próba przypisania wartości zmiennej typu readonly*

Pola readonly typów odnośnikowych

Zachowanie pól z modyfikatorem `readonly` w przypadku typów prostych jest jasne — nie wolno zmieniać ich wartości. To znaczy wartość przypisana polu pozostaje niezmienna przez cały czas działania programu. W przypadku typów odnośnikowych jest oczywiście tak samo, trzeba jednak dobrze uświadomić sobie, co to wówczas oznacza. Otóż pisząc:

```
nazwa_klasy nazwa_pola = new nazwa_klasy(argumenty_konstruktora)
```

polu *nazwa_pola* przypisujemy referencję do nowo powstałego obiektu klasy *nazwa_klasy*. Przykładowo w przypadku klasy `Punkt`, którą przedstawiono na początku rozdziału, deklaracja:

```
Punkt punkt = new Punkt()
```

oznacza przypisanie zmiennej `punkt` referencji do powstałego na stercie obiektu klasy `Punkt` (por. lekcja 14.).

Gdyby pole to było typu `readonly`, tej referencji nie byłoby wolno zmieniać, jednak nic nie stałoby na przeszkodzie, aby modyfikować pola obiektu, na który ta referencja wskazuje. Czyli po wykonaniu instrukcji:

```
readonly Punkt punkt = new Punkt();
```

możliwe byłoby odwołanie w postaci (zakładając publiczny dostęp do pola `x`):

```
punkt.x = 100;
```

Aby lepiej to zrozumieć, spójrzmy na kod przedstawiony na listingu 3.48.

Listing 3.48. *Odwołania do pól typu readonly*

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

public class Program
{
  public readonly Punkt punkt = new Punkt();
  public void UzyjPunktu()
  {
    //prawidłowo, można modyfikować pola obiektu punkt
    punkt.x = 100;
    punkt.y = 200;

    //nieprawidłowo, nie można zmieniać referencji typu readonly
    //punkt = new Punkt();
  }
}
```

Są tu widoczne dwie publiczne klasy: `Program` i `Punkt`. Klasa `Punkt` zawiera dwa publiczne pola typu `int` o nazwach `x` i `y`. Klasa `Program` zawiera jedno publiczne pole tylko do odczytu o nazwie `Punkt`, któremu została przypisana referencja do obiektu klasy `Punkt`. Ponieważ pole jest publiczne, mają do niego dostęp wszystkie inne klasy; ponieważ jest typu `readonly`, nie wolno zmieniać jego wartości. Ale uwaga: zgodnie z tym, co zostało napisane we wcześniejszych akapitach, nie wolno zmienić referencji, ale nic nie stoi na przeszkodzie, aby modyfikować pola obiektu, na który ona wskazuje. Dlatego też pierwsze dwa odwołania w metodzie `UzyjPunktu` są poprawne. Wolno przypisać dowolne wartości polom `x` i `y` obiektu wskazywanego przez pole `punkt`. Nie wolno natomiast zmieniać samej referencji, zatem ujęta w komentarz instrukcja `punkt = new Punkt()` jest nieprawidłowa.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 18.1

Zmień kod z listingu 3.9 tak, aby poprawnie współpracował z klasą `Punkt` z listingu 3.40.

Ćwiczenie 18.2

Zmodyfikuj kod z listingu 3.44 tak, aby dawał prawidłowe wyniki również dla ujemnych współrzędnych `y` oraz by poprawnie obsługiwany był punkt o współrzędnych (0,0). Nie zmieniaj zastosowanych wzorów.

Ćwiczenie 18.3

Napisz kod klasy realizującej zadanie odwrotne do przykładu z listingu 3.44. Dane wewnętrzne powinny być przechowywane w postaci pól `x` i `y`, natomiast metody powinny obsługiwać dane w układzie biegunowym (`pobierzR`, `ustawR`, `pobierzSinusalfa`, `ustawSinusalfa`).

Lekcja 19. Przesłanianie metod i składowe statyczne

W lekcji 15. został przedstawiony termin przeciążania metod; teraz będzie wyjaśnione, w jaki sposób dziedziczenie wpływa na przeciążanie, oraz zostanie przybliżona technika przesłaniania pól i metod. Technika ta pozwala na bardzo ciekawy efekt umieszczenia składowych o identycznych nazwach zarówno w klasie bazowej, jak i potomnej. Drugim poruszanym tematem będą z kolei składowe statyczne, czyli takie, które mogą istnieć nawet wtedy, kiedy nie istnieją obiekty danej klasy.

Przesłanianie metod

Zastanówmy się, co się stanie, kiedy w klasie potomnej ponownie zdefiniujemy metodę o takiej samej nazwie i takich samych argumentach jak w klasie bazowej. Albo inaczej: jakiego zachowania metod mamy się spodziewać w przypadku klas przedstawionych na listingu 3.49.

Listing 3.49. *Przesłanianie metod*

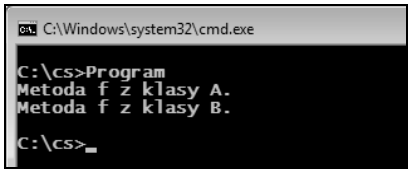
```
public class A
{
  public void f()
  {
    System.Console.WriteLine("Metoda f z klasy A.");
  }
}

public class B : A
{
  public void f()
  {
    System.Console.WriteLine("Metoda f z klasy B.");
  }
}
```

W klasie `A` znajduje się bezargumentowa metoda o nazwie `f`, wyświetlająca na ekranie informację o nazwie klasy, w której została zdefiniowana. Klasa `B` dziedziczy z klasy `A`, zgodnie z zasadami dziedziczenia przejmuje więc metodę `f` z klasy `A`. Tymczasem

w klasie `B` została ponownie zadeklarowana bezargumentowa metoda `f` (również wyświetlająca nazwę klasy, w której została zdefiniowana, czyli tym razem klasy `B`). Wydawać by się mogło, że w takim wypadku wystąpi konflikt nazw (dwukrotne zadeklarowanie metody `f`). Jednak próba kompilacji wykaże, że kompilator nie zgłasza żadnych błędów — pojawi się jedynie ostrzeżenie (o tym za chwilę). Dlaczego konflikt nazw nie występuje? Otóż zasada jest następująca: jeśli w klasie bazowej i pochodnej występuje metoda o tej samej nazwie i argumentach, metoda z klasy bazowej jest przesłaniana (ang. *override*) i mamy do czynienia z tzw. **przesłanianiem metod** (ang. *methods overriding*). A zatem w obiektach klasy bazowej będzie obowiązywała metoda z klasy bazowej, a w obiektach klasy pochodnej — metoda z klasy pochodnej.

Sprawdźmy to. Co pojawi się na ekranie po uruchomieniu klasy `Program` z listingu 3.50, która korzysta z obiektów klas `A` i `B` z listingu 3.49? Oczywiście najpierw tekst `Metoda f z klasy A.`, a następnie tekst `Metoda f z klasy B.` (rysunek 3.19). Skoro bowiem obiekt `a` jest klasy `A`, to wywołanie `a.f()` powoduje wywołanie metody `f` z klasy `A`. Z kolei obiekt `b` jest klasy `B`, zatem wywołanie `b.f()` powoduje uruchomienie metody `f` z klasy `B`.

Rysunek 3.19. *Efekt wywołania przesłoniętej metody*

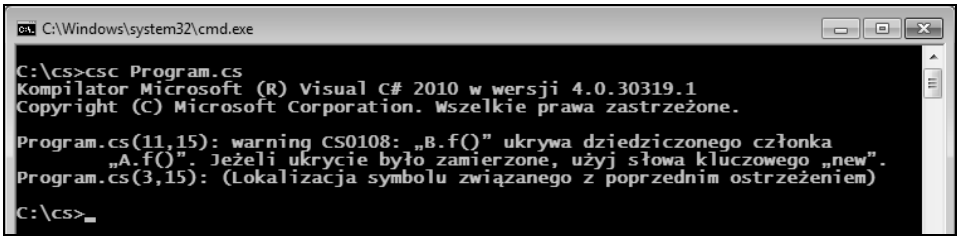


Listing 3.50. *Użycie obiektów klas A i B*

```
public class Program
{
  public static void Main()
  {
    A a = new A();
    B b = new B();

    a.f();
    b.f();
  }
}
```

Wróćmy teraz do ostrzeżenia wygenerowanego przez kompilator przy kompilacji współpracujących ze sobą klas z listingów 3.49 i 3.50. Jest ono widoczne na rysunku 3.20. Otóż kompilator oczywiście wykrył istnienie metody o takiej samej deklaracji w klasach bazowej (`A`) i potomnej (`B`) i poinformował nas o tym. Formalnie należy bowiem określić sposób zachowania takich metod. Zostanie to dokładniej wyjaśnione w rozdziale 6., omawiającym zaawansowane zagadnienia programowania obiektowego.

Na razie przyjmijmy, że w prezentowanej sytuacji, czyli wtedy, gdy w klasie potomnej ma zostać zdefiniowana nowa metoda o takiej samej nazwie, argumentach i typie zwracanym, do jej deklaracji należy użyć słowa kluczowego `new`, które umieszcza się przed typem zwracanym. To właśnie sugeruje komunikat kompilatora z rysunku 3.20.



Rysunek 3.20. *Ostrzeżenie generowane przez kompilator*

Tak więc schematyczna deklaracja takiej metody powinna mieć postać:

```
specyfikator_dostępu new typ_zwracany nazwa_metody(argumenty)
{
  //wnętrze metody
}
```

lub:

```
new specyfikator_dostępu typ_zwracany nazwa_metody(argumenty)
{
  //wnętrze metody
}
```

W naszym przypadku klasa `B` powinna więc wyglądać tak jak na listingu 3.51.

Listing 3.51. *Użycie modyfikatora new*

```
public class B : A
{
  public new void f()
  {
    System.Console.WriteLine("B");
  }
}
```

Może się w tym miejscu pojawić pytanie, czy jest w takim razie możliwe wywołanie przesłoniętej metody z klasy bazowej. Jeśli pytanie to brzmi zbyt zawile, to — na przykładzie klas z listingu 3.49 — chodzi o to, czy w klasie `B` można wywołać metodę `f` z klasy `A`. Nie jest to zagadnienie czysto teoretyczne, gdyż w praktyce programistycznej takie odwołania często upraszczają kod i ułatwiają tworzenie spójnych hierarchii klas. Skoro tak, to odwołanie takie oczywiście jest możliwe. Jak pamiętamy z lekcji 17., jeśli trzeba było wywołać konstruktor klasy bazowej, używało się słowa `base`. W tym przypadku jest podobnie. Odwołanie do przesłoniętej metody klasy bazowej uzyskujemy dzięki wywołaniu w schematycznej postaci:

```
base.nazwa_metody(argumenty);
```

Wywołanie takie najczęściej stosuje się w metodzie przesłaniającej (np. metodzie `f` klasy `B`), ale możliwe jest ono również w dowolnej innej metodzie klasy pochodnej. Gdyby więc metoda `f` klasy `B` z listingu 3.49 miała wywoływać metodę klasy bazowej, kod tej klasy powinien przyjąć postać widoczną na listingu 3.52.

Listing 3.52. *Wywołanie przesłoniętej metody z klasy bazowej*

```
public class B : A
{
  public new void f()
  {
    base.f();
    System.Console.WriteLine("Metoda f z klasy B.");
  }
}
```

Przesłanianie pól

Pola klas bazowych są przesłaniane w sposób analogiczny do metod. Jeśli więc w klasie pochodnej zdefiniujemy pole o takiej samej nazwie jak w klasie bazowej, bezpośrednio dostępne będzie tylko to z klasy pochodnej. Przy deklaracji należy oczywiście użyć modyfikatora `new`. Taka sytuacja jest zobrazowana na listingu 3.53.

Listing 3.53. *Przesłonięte pola*

```
public class A
{
  public int liczba;
}

public class B : A
{
  public new int liczba;
}
```

W klasie `A` zostało zdefiniowane pole o nazwie `liczba` i typie `int`. W klasie `B`, która dziedziczy z `A`, ponownie zostało zadeklarowane pole o takiej samej nazwie i takim samym typie. Trzeba sobie jednak dobrze uświadomić, że każdy obiekt klasy `B` będzie w takiej sytuacji zawierał dwa pola o nazwie `liczba` — jedno pochodzące z klasy `A`, drugie z `B`. Co więcej, tym polom można przypisywać różne wartości. Zilustrowano to w programie widocznym na listingu 3.54.

Listing 3.54. *Odwołania do przesłoniętych pól*

```
using System;

public class Program
{
  public static void Main()
  {
    B b = new B();
    b.liczba = 10;
    ((A)b).liczba = 20;

    Console.Write("Wartość pola liczba z klasy B: ");
    Console.WriteLine(b.liczba);
    Console.Write("Wartość pola liczba odziedziczonego z klasy A: ");
    Console.WriteLine(((A)b).liczba);
```

```
    }
}
```

Tworzymy najpierw obiekt `b` klasy `B`, odbywa się to w standardowy sposób. Podobnie pierwsza instrukcja przypisania ma dobrze już nam znaną postać:

```
b.liczba = 10;
```

W ten sposób ustalona została wartość pola `liczba` zdefiniowanego w klasie `B`. Dzieje się tak dlatego, że to pole przesłania (przykrywa) pole o tej samej nazwie, pochodzące z klasy `A`. Klasyczne odwołanie powoduje więc dostęp do pola zdefiniowanego w klasie, która jest typem obiektu (w tym przypadku obiekt `b` jest typu `B`). W obiekcie `b` istnieje jednak również drugie pole o nazwie `liczba`, odziedziczone z klasy `A`. Do niego również istnieje możliwość dostępu. W tym celu jest wykorzystywana tak zwana technika rzutowania, która zostanie zaprezentowana w dalszej części książki. Na razie przyjmijmy jedynie, że konstrukcja:

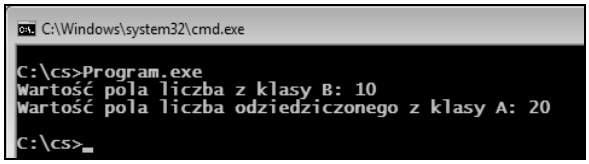
```
((A)b)
```

oznacza: „Potraktuj obiekt klasy `B` tak, jakby był obiektem klasy `A`". Tak więc odwołanie:

```
((A)b).liczba  = 20;
```

to nic innego jak przypisanie wartości `20` polu `liczba` pochodzącemu z klasy `A`.

O tym, że obiekt `b` faktycznie przechowuje dwie różne wartości, przekonujemy się, wyświetlając je na ekranie za pomocą metody `WriteLine` klasy `Console`. Po kompilacji i uruchomieniu programu zobaczymy widok taki jak przedstawiony na rysunku 3.21.



Rysunek 3.21. *Odwołania do dwóch pól zawartych w obiekcie klasy B*

Składowe statyczne

Składowe statyczne to takie, które istnieją nawet wtedy, gdy nie istnieje żaden obiekt danej klasy. Każda taka składowa jest wspólna dla wszystkich obiektów klasy. Składowe te są oznaczane słowem `static`. W dotychczasowych przykładach wykorzystywaliśmy jedną metodę tego typu — `Main`, od której rozpoczyna się wykonywanie programu.

Metody statyczne

Metodę statyczną oznaczamy słowem `static`, które powinno znaleźć się przed typem zwracanym. Zwyczajowo umieszcza się je zaraz za specyfikatorem dostępu[20], czyli schematycznie deklaracja metody statycznej będzie wyglądała następująco:

[20] W rzeczywistości słowo kluczowe `static` może pojawić się również przed specyfikatorem dostępu, ta kolejność nie jest bowiem istotna z punktu widzenia kompilatora. Przyjmuje się jednak, że — ze względu na ujednolicenie notacji — o ile występuje specyfikator dostępu metody, słowo `static` powinno znaleźć się za nim; na przykład: `public static void main`, a nie `static public void main`.

```
specyfikator_dostępu static typ_zwracany nazwa_metody(argumenty)
{
  //treść metody
}
```

Przykładowa klasa z zadeklarowaną metodą statyczną może wyglądać tak, jak zostało to przedstawione na listingu 3.55.

Listing 3.55. *Klasa zawierająca metodę statyczną*

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

Tak napisaną metodę można wywołać tylko przez zastosowanie konstrukcji o ogólnej postaci:

```
nazwa_klasy.nazwa_metody(argumenty_metody);
```

W przypadku klasy `A` wywołanie tego typu miałoby następującą postać:

```
A.f();
```

Nie można natomiast zastosować odwołania poprzez obiekt, a więc instrukcje:

```
A a = new A();
a.f();
```

są nieprawidłowe i spowodują błąd kompilacji.

Na listingu 3.56 jest przedstawiona przykładowa klasa `Program`, która korzysta z takiego wywołania. Uruchomienie tego kodu pozwoli przekonać się, że faktycznie w przypadku metody statycznej nie trzeba tworzyć obiektu.

Listing 3.56. *Wywołanie metody statycznej*

```
public class Program
{
  public static void Main()
  {
    A.f();
  }
}
```

Dlatego też metoda `Main`, od której rozpoczyna się wykonywanie kodu programu, jest metodą statyczną, może bowiem zostać wykonana, mimo że w trakcie uruchamiania aplikacji nie powstały jeszcze żadne obiekty.

Musimy jednak zdawać sobie sprawę, że metoda statyczna jest umieszczana w specjalnie zarezerwowanym do tego celu obszarze pamięci i jeśli powstaną obiekty danej klasy, to będzie ona dla nich wspólna. To znaczy, że dla każdego obiektu klasy nie tworzy się kopii metody statycznej.

Statyczne pola

Do pól oznaczonych jako statyczne można się odwoływać podobnie jak w przypadku statycznych metod, czyli nawet wtedy, gdy nie istnieje żaden obiekt danej klasy. Pola takie deklaruje się, umieszczając przed typem słowo `static`. Schematycznie deklaracja taka wygląda następująco:

```
static typ_pola nazwa_pola;
```

lub:

```
specyfikator_dostępu static typ_pola nazwa_pola;
```

Jeśli zatem w naszej przykładowej klasie `A` ma się pojawić statyczne pole o nazwie `liczba` typu `int` o dostępie publicznym, klasa taka będzie miała postać widoczną na listingu 3.57[21].

Listing 3.57. *Umieszczenie w klasie pola statycznego*

```
public class A
{
  public static int liczba;
}
```

Do pól statycznych nie można odwołać się w sposób klasyczny, tak jak do innych pól klasy — poprzedzając je nazwą obiektu (oczywiście, jeśli wcześniej utworzymy dany obiekt). W celu zapisu lub odczytu należy zastosować konstrukcję:

```
nazwa_klasy.nazwa_pola
```

Podobnie jak metody statyczne, również i pola tego typu znajdują się w wyznaczonym obszarze pamięci i są wspólne dla wszystkich obiektów danej klasy. Tak więc niezależnie od liczby obiektów danej klasy pole statyczne o danej nazwie będzie tylko jedno. Przypisanie i odczytanie zawartości pola statycznego klasy `A` z listingu 3.57 może zostać zrealizowane w sposób przedstawiony na listingu 3.58.

Listing 3.58. *Użycie pola statycznego*

```
public class Program
{
  public static void Main()
  {
    A.liczba = 100;
```

[21] Podobnie jak w przypadku metod statycznych, z formalnego punktu widzenia słowo `static` może się znaleźć przed specyfikatorem dostępu, czyli na przykład: `static public int liczba`. Jednak dla ujednolicenia notacji oraz zachowania zwyczajowej konwencji zapisu będzie konsekwentnie stosowana forma zaprezentowana w powyższym akapicie, czyli: `public static int liczba`.

```
      System.Console.WriteLine("Pole liczba klasy A ma wartość {0}.",
        A.liczba);
    }
  }
```

Odwołanie do pola statycznego może też mieć miejsce wewnątrz klasy. Nie trzeba wtedy stosować przedstawionej konstrukcji, przecież pole to jest częścią klasy. Dlatego też do klasy `A` można by dopisać przykładową metodę `f` o postaci:

```
public void f(int wartosc)
{
  liczba = wartosc;
}
```

której zadaniem jest zmiana wartości pola statycznego.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 19.1

Napisz klasę `Punkt` przechowującą współrzędne punktów na płaszczyźnie oraz klasę `Punkt3D` przechowującą współrzędne punktów w przestrzeni trójwymiarowej. W obu przypadkach przygotuj metodę `odleglosc`, której zadaniem będzie zwrócenie odległości punktu od początku układu współrzędnych.

Ćwiczenie 19.2

Napisz klasę `Punkt3D` dziedziczącą z klasy `Punkt` zaprezentowanej na listingu 3.8. Umieść w niej pole typu `int` o nazwie `z`. W obu klasach zdefiniuj publiczną metodę `WyswietlWspolrzedne` wyświetlającą wartości współrzędnych na ekranie. W metodzie `WyswietlWspolrzedne` z klasy `Punkt3D` nie używaj odwołań do pól `x` i `y`.

Ćwiczenie 19.3

Napisz klasę `Dodawanie`, zawierającą statyczną metodę `Dodaj` przyjmującą dwa argumenty typu `int`. Metoda ta powinna zwrócić wartość będącą wynikiem dodawania obu argumentów.

Ćwiczenie 19.4

Napisz klasę `Przechowalnia` zawierającą statyczną metodę o nazwie `Przechowaj` przyjmującą jeden argument typu `int`. Klasa ta ma zapamiętywać argument przekazany metodzie `Przechowaj` w taki sposób, że każde wywołanie tej metody spowoduje zwrócenie poprzednio zapisanej wartości i zapamiętanie aktualnie przekazanej.

Ćwiczenie 19.5

Napisz kod przykładowej klasy (o dowolnej nazwie) i umieść w niej statyczną prywatną metodę `Wyswietl`, wyświetlającą na ekranie dowolny napis. Przygotuj też osobną klasę `Program` i spraw, aby metoda `Wyswietl` została wywołana, tak aby efekt jej działania pojawił się na ekranie.

Lekcja 20. Właściwości i struktury

Lekcja 20. poświęcona jest dwóm różnym zagadnieniom — właściwościom oraz strukturom. Zostanie w niej pokazane, czym są te konstrukcje programistyczne oraz jak i kiedy się nimi posługiwać. Nie zostaną też pominięte informacje o tym, czym są tzw. akcesory `get` i `set` oraz jak tworzyć właściwości tylko do zapisu lub tylko do odczytu.

Właściwości

Struktura właściwości

Opisanymi dotychczas składowymi klas były pola i metody. W C# uznaje się, że pola z reguły powinny być prywatne, a dostęp do nich powinien być realizowany za pomocą innych konstrukcji, np. metod. To dlatego we wcześniejszych przykładach, np. w klasie `Punkt`, stosowane były metody takie jak `UstawX` czy `PobierzY`. Istnieje jednak jeszcze jeden, i to bardzo wygodny, sposób dostępu, jakim są właściwości (ang. *properties*). Otóż **właściwość** (ang. *property*) to jakby połączenie możliwości, jakie dają pola i metody. Dostęp bowiem wygląda tak samo jak w przypadku pól, ale w rzeczywistości wykonywane są specjalne metody dostępowe zwane **akcesorami** (ang. *accessors*). Ogólny schemat takiej konstrukcji jest następujący:

```
[modyfikator_dostępu] typ_właściwości nazwa_właściwości
{
  get
  {
    //instrukcje wykonywane podczas pobierania wartości
  }
  set
  {
    //instrukcje wykonywane podczas ustawiania wartości
  }
}
```

Akcesory `get` i `set` są przy tym niezależne od siebie. Akcesor `get` powinien w wyniku swojego działania zwracać (za pomocą instrukcji `return`) wartość takiego typu, jakiego jest właściwość, natomiast `set` otrzymuje przypisywaną mu wartość w postaci argumentu o nazwie `value`.

Załóżmy więc, że w klasie `Kontener` umieściliśmy prywatne pole o nazwie `_wartosc` i typie `int`. Do takiego pola, jak już wiadomo z lekcji 18., nie można się bezpośrednio odwoływać spoza klasy. Do jego odczytu i zapisu można więc użyć albo metod, albo właśnie właściwości. Jak to zrobić, zobrazowano w przykładzie widocznym na listingu 3.59.

Listing 3.59. *Użycie prostej właściwości*

```
public class Kontener
{
  private int _wartosc;
  public int wartosc
  {
    get
    {
      return _wartosc;
    }
    set
    {
      _wartosc = value;
    }
  }
}
```

Klasa zawiera prywatne pole `_wartosc` oraz publiczną właściwość `wartosc`. Wewnątrz definicji właściwości znalazły się akcesory `get` i `set`. Oba mają bardzo prostą konstrukcję: `get` za pomocą instrukcji `return` zwraca po prostu wartość zapisaną w polu `_wartosc`, natomiast `set` ustawia wartość tego pola za pomocą prostej instrukcji przypisania. Słowo `value` oznacza tutaj wartość przekazaną akcesorowi w instrukcji przypisania. Zobaczmy, jak będzie wyglądało wykorzystanie obiektu typu `Kontener` w działającym programie. Jest on widoczny na listingu 3.60.

Listing 3.60. *Użycie klasy Kontener*

```
using System;

public class Program
{
  public static void Main()
  {
    Kontener obj = new Kontener();
    obj.wartosc = 100;
    Console.WriteLine(obj.wartosc);
  }
}
```

W metodzie `Main` klasy `Program` jest tworzony i przypisywany zmiennej `obj` nowy obiekt klasy `Kontener`. Następnie właściwości `wartosc` tego obiektu jest przypisywana wartość `100`. Jak widać, odbywa się to dokładnie w taki sam sposób jak w przypadku pól. Odwołanie do właściwości następuje za pomocą operatora oznaczanego symbolem kropki, a przypisanie — za pomocą operatora `=`. Jednak wykonanie instrukcji:

```
obj.wartosc = 100;
```

oznacza w rzeczywistości przekazanie wartości 100 akcesorowi set związanemu z właściwością wartosc. Wartość ta jest dostępna wewnątrz akcesora poprzez słowo value. Tym samym wymieniona instrukcja powoduje zapamiętanie w obiekcie wartości 100. Przekonujemy się o tym, odczytując zawartość właściwości w trzeciej instrukcji metody Main i wyświetlając ją na ekranie. Oczywiście odczytanie właściwości to nic innego jak wywołanie akcesora get.

Właściwości a sprawdzanie poprawności danych

Właściwości doskonale nadają się do sprawdzania poprawności danych przypisywanych prywatnym polom. Załóżmy, że mamy do czynienia z klasą o nazwie Data zawierającą pole typu byte określające dzień tygodnia, takie że 1 to niedziela, 2 — poniedziałek itd. Jeśli dostęp do tego pola będzie się odbywał przez właściwość, to łatwo będzie można sprawdzać, czy aby na pewno przypisywana mu wartość nie przekracza dopuszczalnego zakresu 1 – 7. Napiszmy więc treść takiej klasy; jest ona widoczna na listingu 3.61.

Listing 3.61. *Sprawdzanie poprawności przypisywanych danych*

```
public class Data
{
  private byte _dzien;
  public byte DzienTygodnia
  {
    get
    {
      return _dzien;
    }
    set
    {
      if(value > 0 && value < 8)
      {
        _dzien = value;
      }
    }
  }
}
```

Ogólna struktura klasy jest podobna do tej zaprezentowanej na listingu 3.59 i omówionej w poprzednim podpunkcie. Inaczej wygląda jedynie akcesor set, w którym znalazła się instrukcja warunkowa if. Bada ona, czy wartość value (czyli ta przekazana podczas operacji przypisania) jest większa od 0 i mniejsza od 8, czyli czy zawiera się w przedziale 1 – 7. Jeśli tak, jest przypisywana polu _dzien, a więc przechowywana w obiekcie; jeśli nie, nie dzieje się nic. Spróbujmy więc zobaczyć, jak w praktyce zachowa się obiekt takiej klasy przy przypisywaniu różnych wartości właściwości DzienTygodnia. Odpowiedni przykład jest widoczny na listingu 3.62.

Listing 3.62. *Użycie klasy Data*

```
using System;

public class Program
```

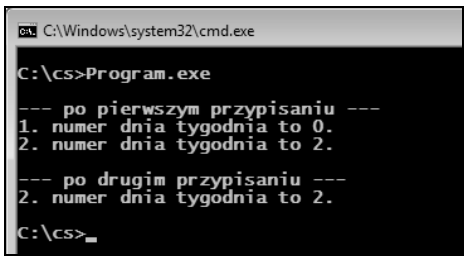
```
{
  public static void Main()
  {
    Data pierwszaData = new Data();
    Data drugaData = new Data();
    pierwszaData.DzienTygodnia = 8;
    drugaData.DzienTygodnia = 2;

    Console.WriteLine("\n--- po pierwszym przypisaniu ---");
    Console.Write("1. numer dnia tygodnia to ");
    Console.WriteLine("{0}.", pierwszaData.DzienTygodnia);
    Console.Write("2. numer dnia tygodnia to ");
    Console.WriteLine("{0}.", drugaData.DzienTygodnia);

    drugaData.DzienTygodnia = 9;
    Console.WriteLine("\n--- po drugim przypisaniu ---");
    Console.Write("2. numer dnia tygodnia to ");
    Console.WriteLine("{0}.", drugaData.DzienTygodnia);
  }
}
```

Najpierw tworzone są dwa obiekty typu `Data`. Pierwszy z nich jest przypisywany zmiennej `pierwszaData`, a drugi zmiennej `drugaData`. Następnie właściwości `DzienTygodnia` obiektu `pierwszaData` jest przypisywana wartość `8`, a obiektowi `drugaData` wartość `2`. Jak już wiadomo, pierwsza z tych operacji nie może zostać poprawnie wykonana, gdyż dzień tygodnia musi zawierać się w przedziale 1 – 7. W związku z tym wartość właściwości (oraz związanego z nią pola `_dzien`) pozostanie niezmieniona, a więc będzie to wartość przypisywana niezainicjowanym polom typu `byte`, czyli `0`. W drugim przypadku operacja przypisania może zostać wykonana, a więc wartością właściwości `DzienTygodnia` obiektu `drugaData` będzie `2`.

O tym, że oba przypisania działają zgodnie z powyższym opisem, przekonujemy się, wyświetlając wartości właściwości obu obiektów za pomocą instrukcji `Console.Write` i `Console.WriteLine`. Później wykonujemy jednak kolejne przypisanie, o postaci:

```
drugaData.DzienTygodnia = 9;
```

Ono oczywiście również nie może zostać poprawnie wykonane, więc instrukcja ta nie zmieni stanu obiektu `drugaData`. Sprawdzamy to, ponownie odczytując i wyświetlając wartość właściwości `DzienTygodnia` tego obiektu. Ostatecznie po kompilacji i uruchomieniu na ekranie zobaczymy widok zaprezentowany na rysunku 3.22.



Rysunek 3.22. *Wynik testowania właściwości DzienTygodnia*

Sygnalizacja błędów

Przykład z poprzedniego podpunktu pokazywał, w jaki sposób sprawdzać poprawność danych przypisywanych właściwości. Nie uwzględniał jednak sygnalizacji błędnych danych. W przypadku zwykłej metody ustawiającej wartość pola informacja o błędzie mogłaby być zwracana jako rezultat działania. W przypadku właściwości takiej możliwości jednak nie ma. Akcesor nie może przecież zwracać żadnej wartości. Można jednak w tym celu wykorzystać technikę tzw. wyjątków. Wyjątki zostaną omówione dopiero w kolejnym rozdziale, a zatem Czytelnicy nieobeznani z tą tematyką powinni pominąć ten punkt i powrócić dopiero po zapoznaniu się z materiałem przedstawionym w lekcjach z rozdziału 4.

Poprawienie kodu z listingu 3.61 w taki sposób, aby w przypadku wykrycia przekroczenia dopuszczalnego zakresu danych był generowany wyjątek, nie jest skomplikowane. Kod realizujący takie zadanie został przedstawiony na listingu 3.63.

Listing 3.63. *Sygnalizacja błędu za pomocą wyjątku*

```
using System;

public class ValueOutOfRangeException : Exception
{
}

public class Data
{
  private byte _dzien;
  public byte DzienTygodnia
  {
    get
    {
      return _dzien;
    }
    set
    {
      if(value > 0 && value < 8)
      {
        _dzien = value;
      }
      else
      {
        throw new ValueOutOfRangeException();
      }
    }
  }
}
```

Na początku została dodana klasa wyjątku `ValueOutOfRangeException` dziedzicząca bezpośrednio z `Exception`. Jest to nasz własny wyjątek, który będzie zgłaszany po ustaleniu, że wartość przekazana akcesorowi `set` jest poza dopuszczalnym zakresem. Treść klasy `Data` nie wymagała wielkich zmian. Instrukcja `if` akcesora `set` została zmieniona na instrukcję warunkową `if…else`. W bloku `else`, wykonywanym, kiedy wartość wskazywana przez `value` jest mniejsza od 1 lub większa od 7, za pomocą instrukcji `throw`

zgłaszany jest wyjątek typu `ValueOutOfRangeException`. Obiekt wyjątku tworzony jest za pomocą operatora new. W jaki sposób można obsłużyć błąd zgłaszany przez tę wersję klasy `Data`, zobrazowano w programie widocznym na listingu 3.64.

Listing 3.64. *Obsługa błędu zgłoszonego przez akcesor set*

```
using System;

public class Program
{
  public static void Main()
  {
    Data pierwszaData = new Data();
    try
    {
      pierwszaData.DzienTygodnia = 8;
    }
    catch(ValueOutOfRangeException)
    {
      Console.WriteLine("Wartość poza zakresem.");
    }
  }
}
```

Utworzenie obiektu jest realizowane w taki sam sposób jak w poprzednich przykładach, natomiast instrukcja przypisująca wartość `8` właściwości `DzienTygodnia` została ujęta w blok `try`. Dzięki temu, jeśli ta instrukcja spowoduje zgłoszenie wyjątku, zostaną wykonane instrukcje znajdujące się w bloku `catch`. Oczywiście w tym przypadku mamy pewność, że wyjątek zostanie zgłoszony, wartość `8` przekracza bowiem dopuszczalny zakres. Dlatego też po uruchomieniu programu na ekranie ukaże się napis `Wartość poza zakresem.`.

Właściwości tylko do odczytu

We wszystkich dotychczasowych przykładach właściwości miały przypisane akcesory `get` i `set`. Nie jest to jednak obligatoryjne. Otóż jeśli pominiemy `set`, to otrzymamy właściwość tylko do odczytu. Próba przypisania jej jakiejkolwiek wartości skończy się błędem kompilacji. Przykład obrazujący to zagadnienie jest widoczny na listingu 3.65.

Listing 3.65. *Właściwość tylko do odczytu*

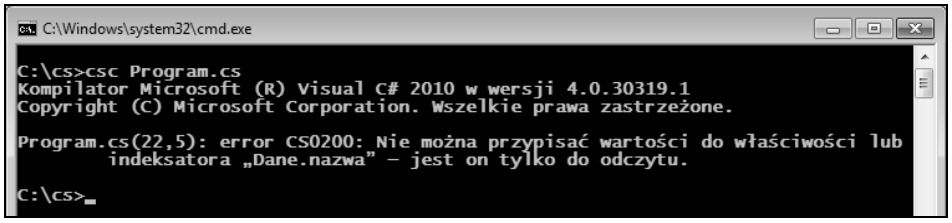
```
using System;

public class Dane
{
  private string _nazwa = "Klasa Dane";
  public string nazwa
  {
    get
    {
      return _nazwa;
    }
  }
```

```
}

public class Program
{
  public static void Main()
  {
    Dane dane1 = new Dane();
    string napis = dane1.nazwa;
    Console.WriteLine(napis);
    //dane1.nazwa = "Klasa Data";
  }
}
```

Klasa `Dane` ma jedno prywatne pole typu `string`, któremu został przypisany łańcuch znaków `Klasa Dane`. Oprócz pola znajduje się w niej również właściwość `nazwa`, w której został zdefiniowany jedynie akcesor `get`, a jego zadaniem jest zwrócenie zawartości pola `_nazwa`. Akcesora `set` po prostu nie ma, co oznacza, że właściwość można jedynie odczytywać. W klasie `Program` został utworzony nowy obiekt typu `Dane`, a następnie została odczytana jego właściwość `nazwa`. Odczytana wartość została przypisana zmiennej `napis` i wyświetlona na ekranie za pomocą instrukcji `Console.WriteLine`. Te wszystkie operacje niewątpliwie są prawidłowe, natomiast oznaczona komentarzem:

```
dane1.nazwa = "Klasa Data";
```

— już nie. Ponieważ nie został zdefiniowany akcesor `set`, nie można przypisywać żadnych wartości właściwości `nazwa`. Dlatego też po usunięciu komentarza i próbie kompilacji zostanie zgłoszony błąd widoczny na rysunku 3.23.



Rysunek 3.23. *Próba przypisania wartości właściwości tylko do odczytu kończy się błędem kompilacji*

Właściwości tylko do zapisu

Skoro, jak zostało to opisane w poprzedniej części lekcji, usunięcie akcesora `set` sprawiało, że właściwość można było tylko odczytywać, logika podpowiada, że usunięcie akcesora `get` spowoduje, iż właściwość będzie można tylko zapisywać. Taka możliwość jest rzadziej wykorzystywana, niemniej istnieje. Jak utworzyć właściwość tylko do zapisu, zobrazowano na listingu 3.66.

Listing 3.66. *Właściwość tylko do zapisu*

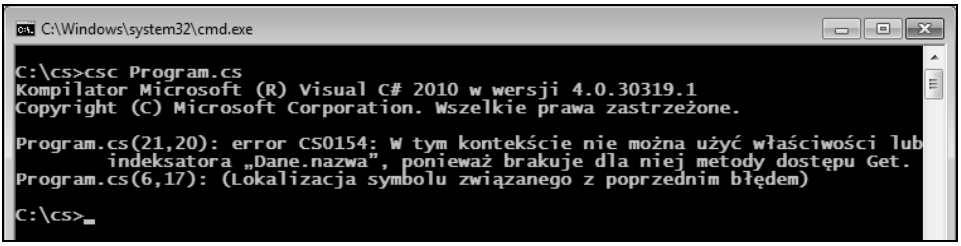
```
using System;

public class Dane
{
```

```
  private string _nazwa = "";
  public string nazwa
  {
    set
    {
      _nazwa = value;
    }
  }
}

public class Program
{
  public static void Main()
  {
    Dane dane1 = new Dane();
    dane1.nazwa = "Klasa Dane";
    //string napis = dane1.nazwa;
  }
}
```

Klasa `Dane` zawiera teraz takie samo pole jak w przypadku przykładu z listingu 3.65, zmienił się natomiast akcesor właściwości `nazwa`. Tym razem zamiast `get` jest `set`. Skoro nie ma `get`, oznacza to, że właściwość będzie mogła być tylko zapisywana. Tak też dzieje się w metodzie `Main` klasy `Program`. Po utworzeniu obiektu typu `Dane` i przypisaniu go zmiennej `dane1`, właściwości `nazwa` jest przypisywany ciąg znaków `Klasa Dane`. Taka instrukcja zostanie wykonana prawidłowo. Inaczej jest w przypadku ujętej w komentarz instrukcji:

```
string napis = dane1.nazwa;
```

Nie może być ona poprawnie wykonana, właściwość `nazwa` jest bowiem właściwością tylko do zapisu. W związku z tym usunięcie komentarza spowoduje błąd kompilacji widoczny na rysunku 3.24.



Rysunek 3.24. *Błąd związany z próbą odczytania właściwości tylko do zapisu*

Właściwości niezwiązane z polami

W dotychczasowych przykładach z tego rozdziału właściwości były powiązane z prywatnymi polami klasy i pośredniczyły w zapisie i odczycie ich wartości. Nie jest to jednak obligatoryjne; właściwości mogą być całkowicie niezależne od pól. Można sobie wyobrazić różne sytuacje, kiedy zapis czy odczyt właściwości powoduje dużo bardziej złożoną reakcję niż tylko przypisanie wartości jakiemuś polu; mogą to być np. operacje

na bazach danych czy plikach. Te zagadnienia wykraczają poza ramy niniejszej publikacji, można jednak wykonać jeszcze jeden prosty przykład, który pokaże właściwość tylko do odczytu zawsze zwracającą taką samą wartość. Jest on widoczny na listingu 3.67.

Listing 3.67. *Właściwość niezwiązana z polem*

```
using System;

public class Dane
{
  public string nazwa
  {
    get
    {
      return "Klasa Dane";
    }
  }
}

public class Program
{
  public static void Main()
  {
    Dane dane1 = new Dane();
    Console.WriteLine(dane1.nazwa);
    Console.WriteLine(dane1.nazwa);
  }
}
```

Klasa `Dane` zawiera wyłącznie właściwość `nazwa`, nie ma w niej żadnego pola. Istnieje także tylko jeden akcesor, którym jest `get`. Z każdym wywołaniem zwraca on wartość typu `string`, którą jest ciąg znaków `Klasa Dane`. Ten ciąg jest niezmienny. W metodzie `Main` klasy `Program` został utworzony nowy obiekt typu `Dane`, a wartość jego właściwości `nazwa` została dwukrotnie wyświetlona na ekranie za pomocą instrukcji `Console.WriteLine`. Oczywiście, ponieważ wartość zdefiniowana w `get` jest niezmienna, każdy odczyt właściwości `nazwa` będzie dawał ten sam wynik.

Struktury

Tworzenie struktur

W C# oprócz klas mamy do dyspozycji również struktury. Składnia obu tych konstrukcji programistycznych jest podobna, choć zachowują się one inaczej. Struktury najlepiej sprawują się przy reprezentacji niewielkich obiektów zawierających po kilka pól i ewentualnie niewielką liczbę innych składowych (metod, właściwości itp.). Ogólna definicja struktury jest następująca:

```
[modyfikator_dostępu] struct nazwa_struktury
{
  //składowe struktury
}
```

Składowe struktury definiuje się tak samo jak składowe klasy. Gdybyśmy na przykład chcieli utworzyć strukturę o nazwie `Punkt` przechowującą całkowite współrzędne `x` i `y` punktów na płaszczyźnie, powinniśmy zastosować konstrukcję przedstawioną na listingu 3.68.

Listing 3.68. *Prosta struktura*

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

Jak skorzystać z takiej struktury? Tu właśnie ujawni się pierwsza różnica między klasą a strukturą. Otóż ta druga jest traktowana jak typ wartościowy (taki jak `int`, `byte` itp.), co oznacza, że po pierwsze, nie ma konieczności jawnego tworzenia obiektu, a po drugie, obiekty będące strukturami są tworzone na stosie, a nie na stercie. Tak więc zmienna przechowująca strukturę zawiera sam obiekt struktury, a nie jak w przypadku typów klasowych — referencję. Spójrzmy zatem na listing 3.69. Zawiera on prosty program korzystający ze struktury `Punkt` z listingu 3.68.

Listing 3.69. *Użycie struktury Punkt*

```
using System;

public class Program
{
  public static void Main()
  {
    Punkt punkt;
    punkt.x = 100;
    punkt.y = 200;
    Console.WriteLine("punkt.x = {0}", punkt.x);
    Console.WriteLine("punkt.y = {0}", punkt.y);
  }
}
```

W metodzie `Main` klasy `Program` została utworzona zmienna `punkt` typu `Punkt`. Jest to równoznaczne z powstaniem instancji tej struktury, obiektu typu `Punkt`. Zwróćmy uwagę, że nie został użyty operator `new`, a więc zachowanie jest podobne jak w przypadku typów prostych. Kiedy pisaliśmy np.:

```
int liczba;
```

od razu powstawała gotowa do użycia zmienna `liczba`. O tym, że faktycznie tak samo jest w przypadku struktur, przekonujemy się, przypisując polom `x` i `y` wartości `100` i `200`, a następnie wyświetlając je na ekranie za pomocą instrukcji `Console.WriteLine`.

Nie oznacza to jednak, że do tworzenia struktur nie można użyć operatora `new`. Otóż instrukcja w postaci:

```
Punkt punkt = new Punkt();
```

również jest prawidłowa. Trzeba jednak wiedzieć, że nie oznacza to tego samego. Otóż jeśli stosujemy konstrukcję o schematycznej postaci:

```
nazwa_struktury zmienna;
```

pola struktury pozostają niezainicjowane i dopóki nie zostaną zainicjowane, nie można z nich korzystać. Jeśli natomiast użyjemy konstrukcji o postaci:

```
nazwa_struktury zmienna = new nazwa_struktury();
```

to zostanie wywołany konstruktor domyślny i wszystkie pola zostaną zainicjowane wartościami domyślnymi dla danego typu (patrz tabela 3.1 z lekcji 16.). Te różnice zostały zobrazowane w przykładzie z listingu 3.70.

Listing 3.70. *Różne sposoby tworzenia struktur*

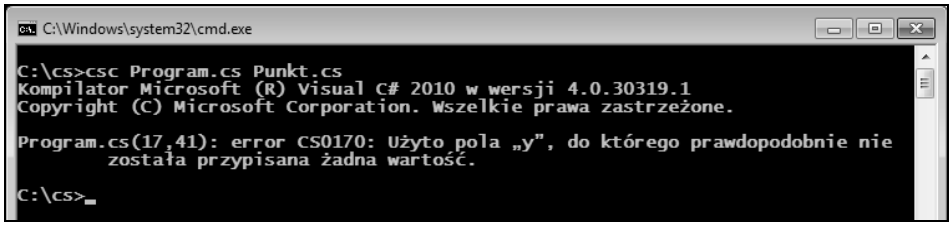
```
using System;

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

    punkt1.x = 100;
    punkt2.x = 100;

    Console.WriteLine("punkt1.x = {0}", punkt1.x);
    Console.WriteLine("punkt1.y = {0}", punkt1.y);

    Console.WriteLine("punkt2.x = {0}", punkt2.x);
    //Console.WriteLine("punkt2.y = {0}", punkt2.y);
  }
}
```

Powstały tu dwie zmienne, a więc i struktury typu `Punkt`: `punkt1` i `punkt2`. Pierwsza z nich została utworzona za pomocą operatora `new`, a druga tak jak zwykła zmienna typu prostego. W związku z tym ich zachowanie będzie nieco inne. Po utworzeniu struktur zostały zainicjowane ich pola `x`, w obu przypadkach przypisano wartość `100`. Następnie za pomocą dwóch instrukcji `Console.WriteLine` na ekranie zostały wyświetlone wartości pól `x` i `y` struktury `punkt1`. Te operacje są prawidłowe. Ponieważ do utworzenia struktury `punkt1` został użyty operator `new`, został też wywołany konstruktor domyślny, a pola otrzymały wartość początkową równą `0`. Niezmienione w dalszej części kodu pole `y` będzie więc miało wartość `0`, która może być bez problemu odczytana.

Inaczej jest w przypadku drugiej zmiennej. O ile polu `x` została przypisana wartość i instrukcja:

```
Console.WriteLine("punkt2.x = {0}", punkt2.x);
```

może zostać wykonana, to pole `y` pozostało niezainicjowane i nie można go odczytywać. W związku z tym instrukcja ujęta w komentarz jest nieprawidłowa, a próba jej wykonania spowodowałaby błąd kompilacji przedstawiony na rysunku 3.25.



Rysunek 3.25. *Próba odwołania do niezainicjowanego pola struktury*

Konstruktory i inicjalizacja pól

Składowe struktur nie mogą być inicjalizowane w trakcie deklaracji. Przypisanie wartości może odbywać się albo w konstruktorze, albo po utworzeniu struktury przez zwykłe operacje przypisania. Oznacza to, że przykładowy kod widoczny na listingu 3.71 jest nieprawidłowy i spowoduje błąd kompilacji.

Listing 3.71. *Nieprawidłowa inicjalizacja pól struktury*

```
public struct Punkt
{
  public int x = 100;
  public int y = 200;
}
```

Struktury mogą zawierać konstruktory, z tym zastrzeżeniem, że nie można definiować domyślnego konstruktora bezargumentowego. Taki konstruktor jest tworzony automatycznie przez kompilator i nie może być redefiniowany. Jeśli chcielibyśmy wyposażyć strukturę `Punkt` w dwuargumentowy konstruktor ustawiający wartości pól `x` i `y`, powinniśmy zastosować kod widoczny na listingu 3.72.

Listing 3.72. *Konstruktor struktury Punkt*

```
public struct Punkt
{
  public int x;
  public int y;
  public Punkt(int wspX, int wspY)
  {
    x = wspX;
    y = wspY;
  }
}
```

Użycie takiego konstruktora mogłoby wyglądać na przykład następująco:

```
Punkt punkt1 = new Punkt(100, 200);
```

Należy też zwrócić uwagę, że inaczej niż w przypadku klas wprowadzenie konstruktora przyjmującego argumenty nie powoduje pominięcia przez kompilator bezargumentowego konstruktora domyślnego. Jak zostało wspomniane wcześniej, do struktur

konstruktor domyślny jest dodawany zawsze. Tak więc używając wersji struktury `Punkt` widocznej na listingu 3.72, nadal można tworzyć zmienne za pomocą konstrukcji typu:

```
Punkt punkt2 = new Punkt();
```

Struktury a dziedziczenie

Struktury nie podlegają dziedziczeniu względem klas i struktur. Oznacza to, że struktura nie może dziedziczyć z klasy ani z innej struktury, a także że klasa nie może dziedziczyć ze struktury. Struktury mogą natomiast dziedziczyć po interfejsach. Temat interfejsów zostanie omówiony dopiero w rozdziale 6., tam też został opublikowany kod interfejsu `IPunkt`, który został wykorzystany w poniższym przykładzie. Tak więc Czytelnicy, którzy nie mieli do tej pory do czynienia z tymi konstrukcjami programistycznymi, mogą na razie pominąć tę część lekcji.

Dziedziczenie struktury po interfejsie wygląda tak samo jak w przypadku klas. Stosowana jest konstrukcja o ogólnej postaci:

```
[modyfikator_dostępu] struct nazwa_struktury : nazwa_interfejsu
{
  //wnętrze struktury
}
```

Gdyby więc miała powstać struktura `Punkt` dziedzicząca po interfejsie `IPunkt` (rozdział 6., lekcja 30., listing 6.24), to mogłaby ona przyjąć postać widoczną na listingu 3.73.

Listing 3.73. *Dziedziczenie po interfejsie*

```
public struct Punkt : IPunkt
{
  private int _x;
  private int _y;
  public int x
  {
    get
    {
      return _x;
    }
    set
    {
      _x = value;
    }
  }
  public int y
  {
    get
    {
      return _y;
    }
    set
    {
      _y = value;
```

```
        }
    }
}
```

W interfejsie `IPunkt` zdefiniowane zostały dwie publiczne właściwości: `x` i `y`, obie z akcesorami `get` i `set`. W związku z tym takie elementy muszą się też pojawić w strukturze. Wartości `x` i `y` muszą być jednak gdzieś przechowywane, dlatego struktura zawiera również prywatne pola `_x` i `_y`. Budowa akcesorów jest tu bardzo prosta. Akcesor `get` zwraca w przypadku właściwości `x` — wartość pola `_x`, a w przypadku właściwości `y` — wartość pola `_y`. Zadanie akcesora `set` jest oczywiście odwrotne, w przypadku właściwości `x` ustawia on pole `_x`, a w przypadku właściwości `y` — pole `_y`.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 20.1

Napisz kod klasy `Punkt` zawierającej właściwości `x` i `y` oraz klasy `Punkt3D` dziedziczącej z `Punkt`, zawierającej właściwość `z`.

Ćwiczenie 20.2

Napisz kod klasy `Punkt` zawierającej właściwości `x` i `y`. Dane o współrzędnych `x` i `y` mają być przechowywane w tablicy liczb typu `int`.

Ćwiczenie 20.3

Napisz kod klasy zawierającej właściwość `liczba` typu rzeczywistego. Kod powinien działać w taki sposób, aby przypisanie wartości właściwości `liczba` powodowało zapisanie jedynie połowy przypisywanej liczby, a odczyt powodował zwrócenie podwojonej zapisanej wartości.

Ćwiczenie 20.4

Napisz kod klasy zawierającej właściwość przechowującą wartość całkowitą. Każdy odczyt tej właściwości powinien powodować zwrócenie kolejnego wyrazu ciągu opisanego wzorem $a_{n+1} = 2 \times (a_n - 1) - 2$.

Ćwiczenie 20.5

Do struktury z listingu 3.73 dopisz dwuargumentowy konstruktor ustawiający wartość jej pól. Zastanów się, czy modyfikacja pól może się odbywać poprzez właściwości `x` i `y`.

Skorowidz

E

F

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

C# Praktyczny kurs

Wydanie II

C# to nowoczesny, w pełni obiektowy następca języków C i C++, łączący w sobie ich legendarną wydajność z mechanizmami, które doskonale sprawdziły się w środowiskach Javy i Delphi. Opracowany przez firmę Microsoft jako podstawowe narzędzie programistyczne dla platformy .NET, język C# zdobywa coraz większą popularność również za sprawą poziomu bezpieczeństwa, możliwości przenoszenia kodu oraz dostępu do wielu przydatnych narzędzi, które znacznie ułatwiają tworzenie różnego rodzaju programów, w tym aplikacji WWW, sieciowych, bazodanowych i graficznych.

Niezależnie od tego, czy dopiero zaczynasz swoją przygodę z programowaniem, czy masz już pewne doświadczenie w korzystaniu z innych języków, książka *C#. Praktyczny kurs. Wydanie II* będzie dla Ciebie doskonałym wprowadzeniem w arkana praktycznego stosowania języka C#. Dzięki niej poznasz wszystkie niezbędne informacje teoretyczne i szybko zaczniesz pisać swoje pierwsze programy. Dowiesz się jak używać podstawowych konstrukcji języka oraz jak korzystać z zaawansowanych mechanizmów obiektowych, obsługiwać wyjątki i przeprowadzać operacje wejścia-wyjścia, a nawet tworzyć interfejsy graficzne aplikacji.

- **Podstawowe informacje na temat języka C# i platformy .NET**
- **Przegląd i instalacja narzędzi przydatnych programiście C#**
- **Struktura programu w C#, proste typy danych i ich zastosowanie**
- **Podstawowe konstrukcje języka i operacje na zmiennych**
- **Instrukcje sterujące i korzystanie z tablic**
- **Podstawy programowania obiektowego**
- **Elementy składowe klas i ich używanie**
- **Dziedziczenie i polimorfizm**
- **Interfejsy, klasy zagnieżdżone i typy uogólnione**
- **Obsługa wyjątków standardowych i definiowanie własnych**
- **Operacje na strumieniach danych, plikach i katalogach**
- **Korzystanie z komponentów graficznych i obsługa zdarzeń**

Poznaj nowoczesny język programowania!
Poznaj C#!

sięgnij po WIĘCEJ

KOD KORZYŚCI

helion.pl
księgarnia internetowa

Nr katalogowy: 8115

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

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

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

ISBN 978-83-246-3870-3

9 788324 638703

Cena 59,00 zł

Informatyka w najlepszym wydaniu