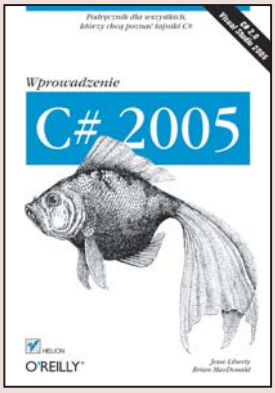
C# 2005. Wprowadzenie



Autorzy: Jesse Liberty, Brian MacDonald
Tłumaczenie: Ireneusz Jakóbik, Tomasz Walczak
ISBN: 83-246-0526-6
Tytuł oryginału: Learning C# 2005: Get Started with C# 2.0 and .NET Programming
Format: B5, stron: 472

Podręcznik dla wszystkich, którzy chcą poznać tajniki C#

- Omówienie środowiska programistycznego Visual C# 2005
- Wprowadzenie do programowania obiektowego w C#
- Tworzenie aplikacji internetowych oraz dla systemu Windows

C# to jeden z podstawowych języków programowania przeznaczonych dla platformy .NET. C#, łączący w sobie najlepsze cechy Javy i C++ szybko stał się jednym z popularniejszych. Wprowadzone na rynek w roku 2005 wersje 2.0 platformy i języka C# przyniosły sporo nowych i przydatnych rozwiązań programistycznych – między innymi nowe typy danych i komponenty. Dzięki nim tworzenie nawet najbardziej złożonych aplikacji stało się znacznie szybsze i prostsze. C# cechuje się niezwykłą wszechstronnością – za jego pomocą można tworzyć zarówno aplikacje dla systemu Windows, jak i dla urządzeń mobilnych. Łatwo również wykorzystać go do pisania aplikacji internetowych w technologii ASP.NET.

„C# 2005. Wprowadzenie" to podręcznik, który objaśnia najważniejsze zagadnienia związane z programowaniem w tym języku. Przeczytasz w nim o platformie .NET oraz opanujesz sposoby wykorzystania środowiska programistycznego Visual C# 2005. Poznasz również elementy języka C# i reguły programowania obiektowego. Nauczysz się wykrywać i usuwać błędy w programach oraz korzystać z komponentów platformy .NET. Stworzysz własne aplikacje dla systemu Windows oraz aplikacje internetowe.

- Struktura aplikacji w języku C#
- środowisko programistyczne Visual C# 2005
- Typy danych i operatory w C#
- Polecenia języka
- Programowanie obiektowe
- Klasy, obiekty i metody
- Operacje na łańcuchach znaków
- Obsługa wyjątków
- Zastosowanie C# do tworzenia aplikacji w technologii ASP.NET

Poznaj język programowania, który zrewolucjonizował rynek

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

Spis treści

ROZDZIAŁ 7.

Klasy i obiekty

W rozdziale 3. opisano typy wbudowane w język C#. Te proste typy pozwalają na przechowywanie wartości liczbowych i łańcuchów tekstowych oraz na posługiwanie się nimi. Jednak prawdziwa potęga C# kryje się w umożliwieniu programiście tworzenia nowych typów, które najlepiej nadają się do rozwiązania danego problemu. To właśnie możliwość tworzenia nowych typów jest cechą charakterystyczną języków zorientowanych obiektowo. Nowe typy są tworzone w C# przez deklarowanie i definiowanie klas.

Poszczególne instancje klasy są nazywane *obiektami*. Różnica między klasą a obiektem jest taka, jak różnica między ideą psa a konkretnym psem, który siedzi przy mnie w chwili, gdy piszę te słowa. Z definicją psa nie można bawić się w aportowanie, ale z instancją psa — tak.

Klasa Pies opisuje, jakie są psy: mają wagę, wzrost, kolor oczu, barwę sierści, charakter itd. Psy mogą też wykonywać czynności: jeść, iść na spacer, szczekać lub spać. Konkretny pies (taki jak mój Milo) ma określoną wagę (28 kilogramów), wzrost (56 centymetrów), kolor oczu (czarne), barwę sierści (żółtą), charakter (aniołek) itd. Może robić różne rzeczy (*metody* w języku programistów), które potrafią robić psy (chociaż przy bliższym poznaniu wydaje się, że jedyną wywoływaną przez niego metodą jest jedzenie).

Ogromną zaletą klas w językach obiektowo zorientowanych jest hermetyzacja ich cech i możliwości w jednej, stanowiącej zamkniętą całość jednostce.

Przykładem niech będzie sortowanie zawartości kontrolki Windows, jaką jest lista wyboru. Lista wyboru jest zdefiniowana jako klasa, a jedną z cech tej klasy jest umiejętność sortowania swojej zawartości. Sortowanie jest *hermetycznie* zamknięte w klasie, a szczegóły sposobu, w jaki lista wyboru sortuje, nie są widoczne dla innych klas. Jeśli lista wyboru ma zostać posortowana, po prostu przekazuje się jej odpowiednie polecenie, a detalami zajmuje się już ona sama.

Wystarczy więc napisać metodę, która nakaże liście wyboru posortowanie się, i właśnie tak się stanie. Nie jest ważne, w jaki sposób lista dokonała sortowania; ważne jest, że potrafi to zrobić.

Jak już powiedziano w rozdziale 6., jest to nazywane hermetyzacją, która razem z polimorfizmem i specjalizacją należy do podstawowych zasad programowania zorientowanego obiektowo. Polimorfizm i dziedziczenie zostaną omówione w rozdziale 11.

Jest taki stary dowcip informatyczny: „Ilu programistów obiektowych potrzeba do wymiany żarówki? Żadnego, wystarczy kazać żarówce, aby sama się wymieniła". W niniejszym rozdziale opisane zostaną cechy języka C# przydatne przy tworzeniu nowych klas. Elementy klasy — jej zachowania i stan — są nazywane wspólnie *składowymi klasy*.

Zachowania klasy określa się pisząc metody (czasami nazywane funkcjami składowymi). Metoda to czynność, którą może wykonać każdy obiekt należący do danej klasy. Na przykład klasa `Pies` może mieć metodę `Szczekaj()`, a klasa `ListBox` — metodę `Sort()`.

Stan klasy jest określony w *polach* (zwanych też zmiennymi składowymi). Pola mogą być typu podstawowego (`int` do przechowywania wieku psa albo zbiór łańcuchów znaków do przechowywania zawartości listy wyboru) lub mogą być obiektami innych klas (na przykład klasa `Pracownik` może posiadać pole typu `Adres`).

Ponadto klasy mogą posiadać *właściwości*, które dla twórcy klasy zachowują się jak metody, ale dla jej klienta wyglądają jak pola. *Klient* to każdy obiekt, który może wchodzić w interakcję z instancjami klasy.

Definiowanie klasy

Podczas definiowania nowej klasy definiuje się cechy wszystkich jej obiektów oraz ich zachowania. Na przykład przy programowaniu własnego okienkowego systemu operacyjnego może być potrzebne utworzenie formantów (w Windows zwanych kontrolkami). Ciekawą kontrolką może być lista wyboru, bardzo przydatna podczas prezentowania użytkownikowi listy, z której może on wybrać interesujące go pozycje.

Listy wyboru mają wiele cech: wysokość, szerokość, położenie, kolor tekstu i inne. Po listach wyboru można spodziewać się także określonych zachowań — można je otwierać, zamykać, sortować itd.

Programowanie zorientowane obiektowo pozwala na utworzenie nowego typu, `ListBox`, który zawiera wszystkie te cechy i zachowania.

Aby utworzyć nowy typ lub klasę, należy najpierw ją zadeklarować, a następnie zdefiniować jej metody i pola. Klasę deklaruje się przy użyciu słowa kluczowego `class`. Pełna składnia wygląda następująco:

```
[atrybuty] [modyfikatory-dostępu] class identyfikator [:klasa-bazowa] {ciało-klasy}
```

Atrybuty są używane w celu udostępnienia specjalnych metadanych o klasie (to znaczy informacji o strukturze albo zastosowaniu klasy). Podczas pisania większości programów w C# atrybuty nie będą potrzebne.

Modyfikatory dostępu zostaną omówione w niniejszego rozdziału (zazwyczaj modyfikatorem dostępu w klasach będzie słowo kluczowe `public`).

Identyfikator to nazwa, którą nadaje się klasie. Zwykle w języku C# nazwy klas są rzeczownikami (Pies, Pracownik, ListaWyboru). Konwencja nazewnictwa zaleca (ale nie nakazuje), aby stosować notację Pascal. W notacji Pascal nie używa podkreśleń ani myślników. Jeśli nazwa składa się z kilku słów (golden retriever), to słowa zestawia się razem, a każde z nich rozpoczyna się wielką literą (GoldenRetriever).

Jak już wspomniano, dziedziczenie jest jednym z filarów programowania obiektowo zorientowanego. Opcjonalna *klasa bazowa* zostanie omówiona przy okazji opisywania dziedziczenia w rozdziale 11.

Definicje składowych klasy, które stanowią *ciało klasy*, są zamknięte w nawiasach klamrowych (`{}`):

```
class Dog
{
   int age; // wiek psa
   int weight; // waga psa
   Bark() { // … }
   Eat () { // … }
}
```

Metody zawarte wewnątrz definicji klasy `Dog` opisują wszystko to, co może zrobić pies. Pola (zmienne składowe), takie jak `age` (wiek) czy `weight` (waga), opisują cechy i stany obiektu klasy `Dog`.

Tworzenie obiektów

Aby utworzyć instancję albo *obiekt* klasy `Dog`, należy zadeklarować obiekt i przydzielić mu miejsce w pamięci komputera. Te dwa kroki są niezbędne do *utworzenia* obiektu. Oto jak to zrobić.

Najpierw deklaruje się obiekt poprzez napisanie nazwy jego klasy (`Dog`), po którym następuje identyfikator (nazwa) obiektu, czyli instancji klasy:

```
Dog milo; // zadeklarowanie milo jako instancji klasy Dog
```

W podobny sposób deklarowane są zmienne lokalne; najpierw podaje się typ (w tym przypadku `Dog`), a następnie identyfikator (`milo`). Warto też zwrócić uwagę, że tak jak w przypadku zmiennych, identyfikator obiektu zapisano w konwencji Camel. Notacja Camel jest podobna do notacji Pascal, z tym że pierwsza litera pozostaje mała. Tak więc nazwa zmiennej lub obiektu może wyglądać następująco: `myDog`, `designatedDriver` albo `plantManager`.

Sama deklaracja jednak nie tworzy obiektu. Aby utworzyć instancję danego typu, należy przydzielić mu miejsce w pamięci poprzez użycie słowa kluczowego `new`:

```
milo = new Dog(); przydzielenie pamięci dla milo
```

Deklarację i przydzielenie pamięci obiektowi typu `Dog` można połączyć w jedną linię kodu:

```
Dog milo = new Dog();
```

Powyższy fragment kodu deklaruje `milo` jako obiekt typu `Dog` i jednocześnie tworzy nową instancję tego typu. Znaczenie nawiasów zostanie wyjaśnione w dalszej części niniejszego rozdziału, przy okazji omawiania konstruktora.

W C# *wszystko* wydarza się wewnątrz klas. Żadna metoda nie może działać poza klasą, nawet `Main()`. Metoda `Main()` jest punktem startowym; jest wywoływana przez system operacyjny, i to właśnie od niej rozpoczyna się wykonywanie programu. Zazwyczaj tworzy się niewielką klasę po to, aby umieścić w niej metodę `Main()`, ponieważ `Main()`, tak jak każda inna metoda, musi znajdować się wewnątrz klasy. Niektóre z przykładów zamieszczonych w niniejszej książce korzystają w tym celu z klasy `Tester`:

```
public class Tester
{
   public static void Main()
   {
      // …
   }
}
```

Pomimo że klasa `Tester` została utworzona po to, aby umieścić w niej metodę `Main()`, to nie powołano do życia żadnego obiektu typu `Tester`. Aby to zrobić, należałoby napisać:

```
Tester myTester = new Tester(); // utworzenie instancji obiektu typu Tester
```

Jak okaże się w dalszej części tego rozdziału, utworzenie obiektu klasy `Tester` pozwoli na wywoływanie innych metod w utworzonym obiekcie (`myTester`).

Tworzenie klasy Time

Klasa pozwalająca na przechowywanie i wyświetlanie bieżącej godziny może być bardzo przydatna. Wewnętrzny stan klasy powinien odzwierciedlać bieżący rok, miesiąc, dzień, godzinę, minutę i sekundę. Być może potrzebna będzie też możliwość wyświetlania czasu w różnych formatach.

Platforma .NET udostępnia w pełni funkcjonalną klasę `DateTime`. Uproszczona klasa `Time` została utworzona wyłącznie po to, aby pokazać sposób, w jaki klasa może zostać zaprojektowana i zaimplementowana.

Taką klasę można zaimplementować definiując pojedynczą metodę i sześć zmiennych, w sposób pokazany w przykładzie z listingu 7.1.

Klasa a obiekt

Jednym ze sposobów na zrozumienie różnicy między klasą a obiektem (instancją klasy) jest rozważenie różnicy między typem `int` a zmienną typu `int`.

Typowi nie da się przypisać wartości:

```
int = 5; // błąd
```

Zamiast tego wartość przypisuje się obiektowi danego typu (w tym przypadku: zmiennej typu `int`):

```
int myInteger;
myInteger = 5; // poprawnie
```

Podobnie, nie da się przypisać wartości polom w klasie; wartości przypisuje się polom w obiekcie. Nie można napisać:

```
Dog.Weight = 5;
```

Takie przypisanie nie ma żadnego znaczenia. Nie jest prawdą, że każdy pies waży pięć kilogramów. Zamiast tego należy napisać:

```
milo.Weight = 5;
```

Oznacza to, że waga konkretnego psa (Milo) wynosi pięć kilogramów.

Listing 7.1. Klasa Time

```
using System;

public class Time
{
   // zmienne prywatne
   private int year;
   private int month;
   private int date;
   private int hour;
   private int minute;
   private int second;
```

```
    // metody publiczne
    public void DisplayCurrentTime()
    {
       Console.WriteLine( "Namiastka metody DisplayCurrentTime()" );
    }
}
public class Tester
{
    static void Main()
    {
       Time timeObject = new Time();
       timeObject.DisplayCurrentTime();
    }
}
```

Powyższy kod tworzy nowy typ zdefiniowany przez użytkownika: `Time`. Definicja klasy `Time` zaczyna się zadeklarowaniem zmiennych składowych: `year`, `month`, `date`, `hour`, `minute` i `second`.

Słowo kluczowe `private` oznacza, że zmienne mogą być dostępne wyłącznie dla metod tej klasy. Słowo kluczowe `private` jest modyfikatorem dostępu, co będzie wyjaśnione w dalszej części niniejszego rozdziału.

Wielu programistów C# woli umieszczać pola klasy w jednym miejscu, na początku lub na końcu deklaracji klasy, chociaż nie jest to wymogiem języka.

Jedyną metodą zadeklarowaną w klasie `Time` jest `DisplayCurrentTime()`. Zwracaną przez metodę `DisplayCurrentTime()` wartością jest `void`, co oznacza, że w rzeczywistości metodzie wywołującej nie zostanie zwrócona żadna wartość. Ciało metody zostało też tymczasowo zastąpione namiastką.

Namiastka metody to tymczasowe rozwiązanie stosowane podczas pisania programu, które pozwala skupić się na ogólnej strukturze, bez potrzeby uzupełniania każdego szczegółu tworzonej klasy. Namiastkę ciała metody zazwyczaj tworzy się poprzez pominięcie jej całej struktury logicznej i zaznaczenie obecności metody wyświetlanym tekstem:

```
public void DisplayCurrentTime()
{
   Console.WriteLine( "Namiastka metody DisplayCurrentTime()" );
}
```

Po zamykającym nawiasie klamrowym zdefiniowana jest druga klasa — `Tester`. `Tester` zawiera znaną już metodę `Main()`. W metodzie `Main()` zdefiniowano instancję klasy `Time` o nazwie `timeObject`:

```
Time timeObject = new Time();
```

Z technicznego punktu widzenia nienazwany obiekt typu `Time` jest tworzony na stercie, a referencja do tego obiektu jest zwracana i użyta do zainicjowania zmiennej o nazwie `timeObject`. Ponieważ proces ten jest trochę zagmatwany, prościej będzie powiedzieć, że został utworzony obiekt typu `Time` o nazwie `timeObject`.

Ponieważ `timeObject` jest instancją typu `Time`, metoda `Main()` może uzyskać dostęp do metody `DisplayCurrentTime()` za pośrednictwem obiektów tego właśnie typu i wywołać ją w celu wyświetlenia czasu:

```
timeObject.DisplayCurrentTime();
```

Metodę obiektu wywołuje się poprzez napisanie nazwy obiektu (`timeObject`), po którym znajduje się operator kropki (`.`) i nazwa metody (`DisplayCurrentTime`) z listą parametrów (w tym przypadku pustą). Sposób przekazywania wartości inicjalizujących zmienne składowe metody zostanie omówiony w dalszej części niniejszego rozdziału przy okazji omawiania konstruktorów.

Modyfikatory dostępu

Modyfikator dostępu określa, które metody klasy (w tym także metody innych klas) mogą używać zmiennych i metod składowych danej klasy. W tabeli 7.1 opisano modyfikatory dostępu języka C#.

Tabela 7.1. Modyfikatory dostępu

Modyfikator dostępu	Ograniczenia
`public`	Brak ograniczeń. Składowe oznaczone jako `public` są dostępne dla wszystkich metod dowolnej klasy.
`private`	Składowe klasy A oznaczone jako `private` są dostępne tylko dla metod klasy A.
`protected`	Składowe klasy A oznaczone jako `protected` są dostępne tylko dla metod klasy A i dla metod klas pochodnych od A. Modyfikator dostępu `protected` jest stosowany z klasami pochodnymi, jak to zostanie wyjaśnione w rozdziale 11.
`internal`	Metody klasy A oznaczone jako `internal` są dostępne dla metod każdej klasy z zestawu klasy A. *Zestaw* to zbiór plików, które z punku widzenia programisty są pojedynczym wykonywalnym plikiem typu exe lub DLL.
`protected internal`	Metody klasy A oznaczone jako `protected internal` są dostępne metodom klasy A, metodom klas pochodnych od A i metodom z zestawu klasy A. Modyfikator ten jest odpowiednikiem `protected` *lub* `internal`. Nie ma pojęcia równoznacznego z `protected i internal`.

Metody publiczne są częścią publicznego interfejsu klasy; definiują sposób, w jaki klasa się zachowuje. Metody prywatne to „metody pomocnicze" wykorzystywane przez metody publiczne w celu wykonania pracy powierzonej klasie. Ponieważ wewnętrzny sposób działania klasy jest prywatny, metody pomocnicze nie muszą (i nie powinny) być udostępniane innym klasom.

Zarówno klasa `Time`, jak i metoda `DisplayCurrentTime()` zostały zadeklarowane jako publiczne, tak więc dowolna inna klasa może z nich korzystać. Gdyby `DisplayCurrentTime()` była prywatna, to nie byłoby możliwe jej wywołanie przez jakąkolwiek inną metodę lub klasę oprócz `Time`. W przykładzie z listingu 7.2 metoda `DisplayCurrentTime()` została wywołana przez metodę należącą do klasy `Tester` (a nie `Time`) Jest to dopuszczalne, ponieważ klasa (`Time`) oraz metoda (`DisplayCurrentTime()`) zostały oznaczone jako publiczne.

Dobrym zwyczajem jest jawne określanie dostępu do wszystkich metod i danych składowych klasy. Chociaż można polegać na domyślnym deklarowaniu składowych klasy jako prywatnych, to jednak jawne określanie dostępu wskazuje na przemyślaną decyzję programisty i dodatkowo dokumentuje kod programu.

Argumenty metod

Zachowanie klasy jest zdefiniowane przez jej metody. Aby metody jak najlepiej wypełniały swoje zadania, możliwe jest zdefiniowanie *parametrów*, czyli informacji przekazywanej metodzie w chwili jej wywołania. Tak więc zamiast pisać jedną metodę, która posortuje zawartość listy

wyboru od A do Z, i drugą, która posortuje zawartość listy od Z do A, wystarczy napisać ogólną metodę `Sort()` i przekazać jej parametr określający sposób sortowania.

Metody mogą mieć dowolną liczbę parametrów. Lista parametrów jest umieszczana w nawiasach znajdujących się po nazwie metody. Typ każdego parametru określa identyfikator przed jego nazwą.

Nazwy „argument" i „parametr" często są stosowane przemiennie, chociaż niektórzy programiści kładą nacisk na rozróżnienie między deklaracją parametru a przekazaniem argumentu w momencie wywoływania metody.

Na przykład poniższa deklaracja definiuje metodę o nazwie `MyMethod()`, która zwraca wartość `void` (czyli nie zwraca żadnej wartości) i jest wywoływana z dwoma parametrami (typu `int` i `Button`):

```
void MyMethod (int firstParam, Button secondParam)
{
   // ...
}
```

Wewnątrz ciała metody parametry zachowują się jak zmienne lokalne, które zostały zainicjalizowane przekazanymi wartościami. Listing 7.2 ilustruje sposób przekazywania argumentów do metody. W tym przypadku są to zmienne o typach `int` i `float`.

Listing 7.2. Przekazywanie parametrów

```
using System;
public class MyClass
{
   public void SomeMethod( int firstParam, float secondParam )
   {
      Console.WriteLine( "Oto otrzymane parametry: {0}, {1}", firstParam, secondParam );
   }
}
public class Tester
{
   static void Main()
   {
      int howManyPeople = 5;
      float pi = 3.14f;
      MyClass mc = new MyClass();
      mc.SomeMethod( howManyPeople, pi );
   }
}
```

Wynik działania programu wygląda następująco:

```
Oto otrzymane parametry: 5, 3,14
```

Warto zapamiętać, że przy przekazywaniu zmiennej typu `float` z miejscami po przecinku (3.14) należy dołączyć literę „f" (3.14f), aby zasygnalizować kompilatorowi, że zmienna jest typu `float`, a nie `double`.

Metoda `SomeMethod()` pobiera dwa parametry, `firstParam` i `secondParam`, a następnie wyświetla je korzystając z metody `Console.WriteLine()`. `firstParam` jest typu `int`, a `secondParam` typu `float` i oba są traktowane jak zmienne lokalne w metodzie `SomeMethod()`. Można posługiwać się nimi w tej metodzie, ale poza nią są nieosiągalne, a po jej zakończeniu niszczone.

W metodzie wywołującej (`Main()`) tworzone i inicjalizowane są dwie zmienne (`howManyPeople` i `pi`). Zmienne te są przekazywane jako parametry do metody `SomeMethod()`. Kompilator przepisuje wartość `howManyPeople` do `firstParam`, a wartość `pi` do `secondParam` na podstawie miejsc, jakie argumenty zajmują na liście parametrów.

Konstruktory

Warto zwrócić uwagę na to, że w przykładzie z listingu 7.1 instrukcja tworząca obiekt typu `Time` wygląda podobnie jak wywołanie metody `Time()`:

```
Time timeObject = new Time();
```

Rzeczywiście, w momencie tworzenia nowej instancji obiektu wywoływana jest metoda zwana *konstruktorem*. Za każdym razem kiedy definiuje się klasę, można też zdefiniować własny konstruktor, ale jeśli się tego nie zrobi, to kompilator automatycznie i niezauważalnie sam go udostępni.

Zadaniem konstruktora jest utworzenie instancji obiektu zdefiniowanego przez klasę i nadanie mu poprawnego stanu. Przed uruchomieniem konstruktora obiekt jest tylko fragmentem pamięci; po zakończeniu pracy konstruktora obiekt jest już pełnoprawnym przedstawicielem danej klasy.

W klasie `Time` z listingu 7.2 nie zdefiniowano konstruktora, więc kompilator udostępnił go automatycznie. Konstruktor udostępniony przez kompilatora tworzy obiekt, ale nie podejmuje żadnej innej akcji.

Każdy konstruktor, który nie pobiera żadnych argumentów, jest nazywany *konstruktorem domyślnym*. Konstruktor udostępniany przez kompilator nie pobiera argumentów, a więc jest konstruktorem domyślnym. Takie nazewnictwo wprowadza niemało zamętu. Można utworzyć swój własny konstruktor domyślny, ale jeśli nie utworzy się żadnego konstruktora, to kompilator posłuży się swoim domyślnym konstruktorem.

Jeśli zmienne składowe nie zostaną jawnie zainicjalizowane, to przyjmą one wartości zerowe (zmienne całkowite przyjmą 0, łańcuchy znaków będą pustymi łańcuchami itd.). W tabeli 7.2 wymieniono wartości zerowe przypisywane poszczególnym typom.

Tabela 7.2. Typy podstawowe i ich wartości zerowe

Typ	Wartość zerowa
Numeryczny (`int`, `long` itd.)	0
`bool`	false
`char`	'\0' (null)
`enum`	0
Referencyjny	null

Zwykle definiuje się własny konstruktor i przekazuje mu argumenty w taki sposób, aby konstruktor mógł nadać obiektowi wartość początkową. W przykładzie z listingu 7.3 przekazywane są: bieżący rok, miesiąc, dzień (itd.), dzięki czemu obiekt zostanie utworzony z wartościami początkowymi.

Sposób deklarowania konstruktora jest taki sam, jak każdej innej metody składowej, z następującymi wyjątkami:

- Nazwa konstruktora musi być taka sama jak nazwa klasy.
- Konstruktor nie zwraca żadnej wartości (nawet typu `void`).

Jeśli do konstruktora mają zostać przekazane argumenty, to ich listę definiuje się tak samo jak w przypadku jakiejkolwiek innej metody. W przykładzie z listingu 7.3 zadeklarowano konstruktora dla klasy `Time`, który pobiera sześć argumentów: rok, miesiąc, dzień, godzina, minuta i sekunda dla każdego nowego obiektu typu `Time`, który jest tworzony.

Listing 7.3. Tworzenie konstruktora

```
using System;
public class Time
{
   // prywatne zmienne składowe
   int year;
   int month;
   int date;
   int hour;
   int minute;
   int second;

   // metoda publiczna
   public void DisplayCurrentTime()
   {
      System.Console.WriteLine( "{0}-{1}-{2} {3}:{4}:{5}", year, month, date, hour,
         minute, second );
   }

   // konstruktor
   public Time( int theYear, int theMonth, int theDate, int theHour, int theMinute,
      int theSecond )
   {
      year = theYear;
      month = theMonth;
      date = theDate;
      hour = theHour;
      minute = theMinute;
      second = theSecond;
   }
}

public class Tester
{
   static void Main()
   {
      Time timeObject = new Time( 2008, 8, 1, 9, 35, 20 );
      timeObject.DisplayCurrentTime();
   }
}
```

Wynik działania programu wygląda następująco:

```
2008-8-1 9:35:20
```

W powyższym przykładzie konstruktor pobiera serię wartości całkowitych i w oparciu o te parametry inicjalizuje zmienne składowe.

W tym, podobnie jak w każdym innym przykładowym programie, pominięto sprawdzanie błędów, aby przykłady były prostsze. Oczywiście pozwala to na wprowadzenie takiej daty jak 45 lipca 2005 roku, godzina 29:32, jednak nie powinno się tego robić.

Po zakończeniu działania konstruktora został utworzony i zainicjalizowany obiekt typu `Time`. Wartości jego zmiennych składowych zostają wyświetlone po wywołaniu z metody `Main()` metody `DisplayCurrentTime()`.

Warto zamienić w komentarz jedno z przypisań wartości i uruchomić program ponownie. Okaże się, że każda ze zmiennych składowych, która nie została zainicjalizowana, ma wartość zero. Warto pamiętać o tym, że zmienne składowe powinny być inicjalizowane. Jeśli tego się nie zrobi, konstruktor przypisze im wartości zerowe.

Inicjalizator

Zamiast konstruktora, do nadawania zmiennym składowym wartości początkowych można użyć *inicjalizatora*. Inicjalizator tworzy się przypisując wartość początkową zmiennej składowej:

```
private int second = 30; // inicjalizator
```

Jeśli zmienna wewnętrzna przechowująca sekundy — niezależnie od tego, jaki ustawiono czas — ma zawsze być inicjalizowana wartością 30, to można przepisać klasę `Time` w taki sposób, aby wartość ta była nadawana inicjalizatorem, jak pokazano to w przykładzie z listingu 7.4.

Listing 7.4. Zastosowanie inicjalizatora

```
using System;
public class Time
{
   // prywatne zmienne składowe
   int year;
   int month;
   int date;
   int hour;
   int minute;
   int second = 30;

   // metoda publiczna
   public void DisplayCurrentTime()
   {
      System.Console.WriteLine( "{0}-{1}-{2} {3}:{4}:{5}", year, month, date, hour,
         minute, second );
   }

   // konstruktor
   public Time( int theYear, int theMonth, int theDate, int theHour, int theMinute )
   {
      year = theYear;
      month = theMonth;
      date = theDate;
      hour = theHour;
      minute = theMinute;
   }
}

public class Tester
{
   static void Main()
   {
      Time timeObject = new Time( 2008, 8, 1, 9, 35 );
      timeObject.DisplayCurrentTime();
   }
}
```

Wynik działania programu wygląda następująco:

```
2008-8-1 9:35:30
```

Bez udziału inicjalizatora konstruktor nada każdej całkowitej zmiennej składowej wartość zero (0). W powyższym przykładzie zmienna `second` została jednak zainicjalizowana wartością 30:

```
private int second = 30; // inicjalizator
```

Później w niniejszym rozdziale okaże się, że może istnieć więcej niż tylko jeden konstruktor. Jeśli potrzeba przypisać wartość 30 zmiennej `second` za pomocą większej liczby konstruktorów, można uniknąć problemów związanych z koniecznością zachowania zgodności wszystkich tych konstruktorów ze sobą, inicjalizując zmienną `second` zamiast przypisywać wartość 30 w każdym z konstruktorów z osobna.

Słowo kluczowe this

Słowo kluczowe `this` odnosi się do bieżącej instancji obiektu. Referencja `this` jest ukrytym parametrem w każdej niestatycznej metodzie klasy (metody statyczne zostaną omówione w dalszej części niniejszego rozdziału). Referencja `this` jest zwykle używana na jeden z trzech sposobów. Pierwszy sposób polega na kwalifikowaniu składowych instancji, które mają takie same nazwy jak nazwy parametrów:

```
public void SomeMethod (int hour)
{
   this.hour = hour;
}
```

W powyższym przykładzie metoda `SomeMethod()` pobiera parametr (`hour`) o takiej samej nazwie, jaką ma zmienna składowa klasy. Referencja `this` pozwala na uniknięcie niejednoznaczności — `this.hour` odnosi się do zmiennej składowej, a `hour` — do parametru.

Dzięki referencji `this` można sprawić, że przypisania zmiennych składowych staną się czytelniejsze:

```
public void SetTime(year, month, date, newHour, newMinute, newSecond)
{
   this.year = year;          // użycie "this" jest wymagane
   this.month = month;        // wymagane
   this.date = date;          // wymagane
   this.hour = newHour;       // użycie "this" jest opcjonalne
   this.minute = newMinute;   // opcjonalne
   this.second = newSecond;   // także w porządku
}
```

Jeśli nazwa parametru jest taka sama jak nazwa zmiennej składowej, to wtedy referencja `this` *musi* zostać użyta, aby zapewnić rozróżnienie obu nazw. Jeśli jednak nazwy są różne (takie jak `newMinute` i `newSecond`), to stosowanie referencji `this` nie jest wymagane.

Argument na korzyść nadawania zmiennym składowym i parametrom takich samych nazw jest taki, że dzięki temu związek między zmienną a parametrem jest oczywisty. Argumentem przeciw jest brak jasności, do czego odnosi się nazwa w danej chwili.

Drugim zastosowaniem referencji `this` jest przekazanie bieżącego obiektu jako parametru innej metodzie, tak jak w poniższym kodzie:

```
class SomeClass
{
   public void FirstMethod(OtherClass otherObject)
   {
      otherObject.SecondMethod(this);
   }
   // …
}
```

W powyższym fragmencie występują dwie klasy: `SomeClass` i `OtherClass`. `SomeClass` posiada metodę o nazwie `FirstMethod()`, a `OtherClass` metodę o nazwie `SecondMethod()`.

Z metody `FirstMethod()` należy wywołać metodę `SecondMethod()`, przekazując jej jako parametr bieżący obiekt (będący instancją klasy `SomeClass`) w celu dalszego przetwarzania. Aby tego dokonać, przekazuje się referencję `this`, która odnosi się do bieżącej instancji klasy `SomeClass`.

Trzecie zastosowanie słowa kluczowego `this` ma związek z mechanizmami indeksowania, które zostaną opisane w rozdziale 12.

Składowe statyczne i składowe instancji

Pola, właściwości i metody klasy mogą być *składowymi instancji* lub *składowymi statycznymi*. Składowe instancji są powiązane z obiektami danego typu, podczas gdy składowe statyczne są związane z klasą, a nie z konkretną instancją obiektu. Metody są metodami składowymi instancji, chyba że jawnie zostaną określone słowem kluczowym `static`.

Ogromna większość metod będzie metodami instancji. Semantyką metody instancji jest podejmowanie działania w danym obiekcie. Czasami jednak wygodniej będzie wywołać metodę bez potrzeby posiadania instancji klasy. W tym właśnie celu używa się metod statycznych.

Dostęp do składowej statycznej jest możliwy poprzez nazwę klasy, w której jest ona zadeklarowana. Przykładowo zostały utworzone dwie instancje klasy `Button`, o nazwach `btnUpdate` i `btnDelete`.

Niech w klasie `Button` istnieje metoda składowa o nazwie `Draw()` i metoda statyczna `GetButtonCount()`. Zadaniem `Draw()` jest narysowanie bieżącego przycisku, a `GetButtonCount()` ma zwracać liczbę przycisków widocznych aktualnie na formularzu.

Dostęp do metody składowej odbywa się przez instancję klasy, czyli przez obiekt:

```
btnUpdate.SomeMethod();
```

Dostęp do metody statycznej odbywa się przez nazwę klasy, a nie poprzez instancję jej typu:

```
Button.GetButtonCount();
```

Wywoływanie metod statycznych

Metody statyczne działają w klasie, nie w instancji klasy. Nie posiadają referencji `this`, ponieważ nie istnieje obiekt, na który mogłyby wskazywać.

Metody statyczne nie mogą mieć bezpośredniego dostępu do niestatycznych składowych. Metoda `Main()` jest metodą statyczną i aby mogła wywołać niestatyczną metodę jakiejkolwiek klasy, w tym także swojej własnej klasy, musi utworzyć obiekt.

Na potrzeby następnego przykładu potrzebne będzie utworzenie w Visual Studio 2005 nowej aplikacji konsolowej o nazwie `StaticTester`. VS.NET utworzy przestrzeń nazw `StaticTester` i klasę o nazwie `Program`. Należy zmienić nazwę `Program` na `Tester`, pozbyć się wszystkich komentarzy, które Visual Studio umieściło powyżej linii z metodą `Main()`, i skasować parametr `args` z tej metody. Po skończonym zabiegu kod źródłowy powinien wyglądać następująco:

```
using System;
namespace StaticTester
{
   class Tester
   {
      static void Main()
      {
      }
   }
}
```

Jest to całkiem dobry punkt wyjściowy. Do tej pory wszystkie programy wykonywały swoją pracę w metodzie `Main()`, ale teraz zostanie utworzona metoda będąca obiektem o nazwie `Run()`. Praca programu nie będzie już wykonywana w metodzie `Main()`, tylko w metodzie `Run()`.

W obrębie klasy `Tester` (ale nie w metodzie `Main()`) należy zadeklarować nową metodę składową o nazwie `Run()`. Podczas deklarowania metody należy podać modyfikator dostępu (`public`), zwracany typ, identyfikator i nawiasy:

```
public void Run()
```

W nawiasach umieszcza się parametry, ale ponieważ metoda `Run()` nie posiada żadnych parametrów, więc nawiasy pozostaną puste. Między nawiasami klamrowymi ograniczającymi ciało metody należy wpisać instrukcję wyświetlającą na ekranie konsoli słowa: „Witaj świecie”:

```
public void Run()
{
   Console.WriteLine("Witaj świecie");
}
```

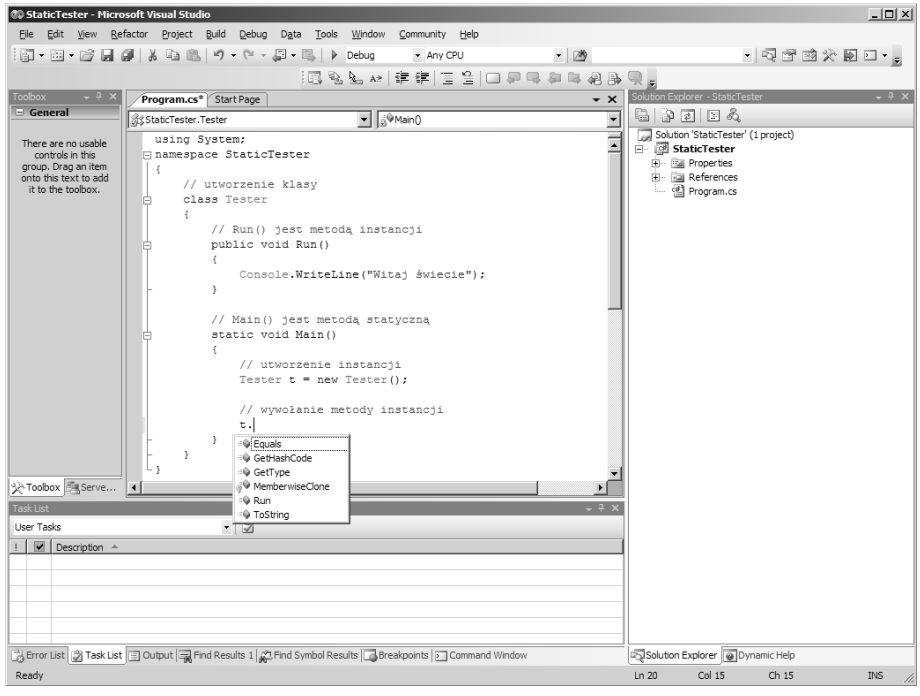
Metoda `Run()` jest metodą instancji, natomiast metoda `Main()` metodą statyczną, która nie może bezpośrednio wywołać metody `Run()`. Konieczne zatem będzie utworzenie obiektu klasy `Tester`, za pośrednictwem którego będzie wywołana metoda `Run()`:

```
Tester t = new Tester()
```

Podczas wpisywania słowa kluczowego `new` IntelliSense stara się pomóc, podpowiadając nazwę klasy. Klasa `Tester` znajduje się na wyświetlonej liście, ponieważ jest dostępną klasą, tak samo jak i inne widoczne na liście klasy.

W następnej linii kodu zostanie wywołana metoda `Run()` w obiekcie `t` klasy `Tester`. Po wpisaniu „`t`” i wprowadzeniu kropki IntelliSense pokaże wszystkie publiczne metody klasy `Tester`, jak to pokazano na rysunku 7.1.

Warto zwrócić uwagę, że klasa `Tester` posiada wiele innych metod, które nie zostały utworzone w prezentowanym przykładzie (`Equals()`, `Finalize()` i inne). Każda klasa w C# jest obiektem, a metody te są częścią klasy `Object`, która zostanie omówiona w rozdziale 11.



Rysunek 7.1. IntelliSense

Po skończeniu wpisywania kodu program powinien wyglądać tak, jak przedstawia to listing 7.5.

Listing 7.5. Metody instancji

```
using System;
namespace StaticTester
{
   // utworzenie klasy
   class Tester
   {
      // Run() jest metodą instancji
      public void Run()
      {
         Console.WriteLine( "Witaj świecie" );
      }

      // Main() jest metodą statyczną
      static void Main()
      {
         // utworzenie instancji
         Tester t = new Tester();

         // wywołanie metody instancji
         t.Run();
      }
   }
}
```

Wynik działania programu wygląda następująco:

```
Witaj świecie
```

Według powyższego schematu będzie pisana większość aplikacji konsolowych w tej książce. Zadania metody `Main()` zostaną ograniczone do ustanowienia obiektu i wywołania metody `Run()`.

Używanie pól statycznych

Częstym zastosowaniem statycznych zmiennych składowych (pól statycznych) jest przechowywanie liczby istniejących obiektów danej klasy. W poniższym przykładzie zostanie utworzona klasa `Cat`, która może być wykorzystana w symulacji sklepu zoologicznego.

Dla potrzeb przykładu, klasa `Cat` została ograniczona do niezbędnego minimum. Program przedstawiono na listingu 7.6, po którym został on przeanalizowany.

Listing 7.6. Pola statyczne

```
using System;

namespace Test
{
   // deklaracja ograniczonej do minimum klasy Cat
   public class Cat
   {
      // prywatna składowa statyczna przechowująca liczbę utworzonych obiektów klasy Cat
      private static int instances = 0;
      private int weight;
      private String name;

      // konstruktor obiektów typu Cat zwiększa zmienną liczącą koty
      public Cat( String name, int weight )
      {
         instances++;
         this.name = name;
         this.weight = weight;
      }

      // metoda statyczna podająca bieżącą liczbę obiektów typu Cat
      public static void HowManyCats()
      {
         Console.WriteLine( "Zaadoptowano {0} kot(a/y/ów)", instances );
      }
      public void TellWeight()
      {
         Console.WriteLine( "{0} waży {1} kilogram(-/y/ów)", name, weight );
      }
   }

   class Tester
   {
      public void Run()
      {
         Cat.HowManyCats();
         Cat frisky = new Cat( "Filemon", 2 );
         frisky.TellWeight();
         Cat.HowManyCats();
         Cat whiskers = new Cat( "Bonifacy", 3 );
         whiskers.TellWeight();
         Cat.HowManyCats();
      }

      static void Main()
      {
         Tester t = new Tester();
         t.Run();
      }
   }
}
```

Wynik programu wygląda następująco:

```
Zaadoptowano 0 kot(a/y/ów)
Filemon waży 2 kilogram(-/y/ów)
Zaadoptowano 1 kot(a/y/ów)
Bonifacy waży 3 kilogram(-/y/ów)
Zaadoptowano 2 kot(a/y/ów)
```

Klasa `Cat` zaczyna się od zadeklarowania statycznej zmiennej składowej o nazwie `instances`, która zostaje zainicjalizowana wartością 0. Ta statyczna zmienna będzie przechowywać liczbę tworzonych obiektów klasy `Cat`. Za każdym razem gdy zostanie wywołany konstruktor tworzący nowy obiekt, zmienna `instances` zostanie zwiększona o 1.

W klasie `Cat` zadeklarowano także dwie zmienne składowe: `name` i `weight`. W nich zapisane będą imię i waga należące do każdego kota (czyli obiektu typu `Cat`).

W klasie `Cat` zadeklarowano poza tym dwie metody: `HowManyCats()` i `TellWeight()`. `HowManyCats()` jest metodą statyczną, ponieważ liczba obiektów typu `Cat` nie jest cechą żadnego z tych obiektów, lecz cechą całej klasy. Metoda `TellWeight()` jest metodą instancji, gdyż imię i waga są cechami każdego z poszczególnych kotów (obiektów typu `Cat`).

Metoda `Main()` wywołuje metodę statyczną `HowManyCats()` bezpośrednio poprzez klasę:

```
Cat.HowManyCats();
```

Następnie metoda `Main()` tworzy obiekt typu `Cat` i wywołuje metodę składową instancji `TellWeight()` za pośrednictwem instancji typu `Cat`:

```
Cat frisky = new Cat();
frisky.TellWeight();
```

Przy każdym utworzeniu nowego obiektu typu `Cat` metoda `HowManyCats()` wyświetla bieżącą liczbę kotów.

Usuwanie obiektów

W przeciwieństwie do innych języków programowania (takich jak C, C++, Pascal itd.), C# oferuje *mechanizm przywracania pamięci*. Obiekty są usuwane po tym, jak spełnią swoją rolę. Nie ma potrzeby martwić się o czyszczenie pamięci po obiektach, chyba że korzystano z niezarządzanych lub rzadkich zasobów. Niezarządzane zasoby to funkcje systemu operacyjnego poza platformą .NET, natomiast rzadkie zasoby to zasoby, które występują w ograniczonych ilościach, wynikających na przykład z praw licencyjnych (takich jak połączenia z bazą danych).

Po skończeniu pracy z zasobem niezarządzanym należy w jawny sposób go uwolnić. Jawne zarządzanie zasobem tego typu odbywa się za pośrednictwem *destruktora*, który jest wywoływany przez mechanizm przywracania pamięci podczas usuwania obiektu.

Treść niniejszego podrozdziału jest raczej trudna i został on tu zamieszczony po to, aby omówienie tematu było pełne. Można go w tej chwili pominąć i powrócić do niego później. W końcu książka ta należy do Czytelnika, który ją kupił (kupił, prawda?).

Destruktor w języku C# jest deklarowany przy użyciu tyldy:

```
~MyClass(){}
```

Powyższa składnia jest tłumaczona przez kompilator na następujący kod:

```
protected override void Finalize()
{
   try
   {
      // w tym miejscu wykonuje się praca
   }
   finally
   {
      base.Finalize();
   }
}
```

Z tego powodu niektórzy programiści nazywają destruktora *finalizatorem*.

Destruktora nie można wywołać jawnie; destruktor musi zostać wywołany przez mechanizm przywracania pamięci. Jeśli obsługuje się cenne niezarządzane zasoby (takie jak deskryptory plików), które chce się jak najszybciej zamknąć i uwolnić, to należy zaimplementować interfejs `IDisposable` (więcej informacji na temat interfejsów znajduje się w rozdziale 13.).

Interfejs `IDisposable` wymaga utworzenia metody o nazwie `Dispose()`, która będzie wywoływana przez klientów.

Jeśli udostępni się metodę `Dispose()`, to należy powstrzymać mechanizm przywracania pamięci przed wywoływaniem destruktora obiektu. Aby to zrobić, wywołuje się metodę statyczną `GC.SuppressFinalize()`, przekazując jej jako argument referencję `this` wskazującą na obiekt. Wtedy destruktor może wywołać metodę `Dispose()`. Można zatem napisać:

```
using System;
class Testing : IDisposable
{
   bool is_disposed = false;
   protected virtual void Dispose( bool disposing )
   {
      if ( !is_disposed ) // usuń tylko raz!
      {
         if ( disposing )
         {
            Console.WriteLine( "Poza destruktorem, można tworzyć odwołania do innych
            obiektów" );
         }
         // wykonaj przywracanie pamięci
         Console.WriteLine( "Przywracanie..." );
      }
      this.is_disposed = true;
   }
   public void Dispose()
   {
      Dispose( true );
      // komunikat dla mechanizmu przywracania pamięci, aby nie finalizował
      GC.SuppressFinalize( this );
   }
   ~Testing()
   {
      Dispose( false );
      Console.WriteLine( "W destruktorze." );
   }
}
```

Dla niektórych obiektów lepsze będzie wywołanie metody `Close()` (zastosowanie metody `Close()` ma większy sens niż metoda `Dispose()` na przykład w przypadku obiektów plikowych). Można to zaimplementować tworząc prywatną metodę `Dispose()` oraz publiczną metodę `Close()` i wywołując metodę `Dispose()` z metody `Close()`.

Ponieważ nie ma pewności, że klient prawidłowo wywoła metodę `Dispose()` i ponieważ finalizacja jest niedeterministyczna (to znaczy, że nie ma się wpływu na to, kiedy zostanie uruchomiony mechanizm przywracania pamięci), C# udostępnia instrukcję `using`, aby zagwarantować wywołanie metody `Dispose()` w najbliższym możliwym czasie. Cała sztuka polega na tym, by zadeklarować, które obiekty są używane, i utworzyć dla nich obszar za pomocą nawiasów klamrowych. Kiedy zostanie osiągnięty zamykający nawias klamrowy, metoda `Dispose()`zostanie automatycznie wywołana w obiekcie, jak zostało to pokazane poniżej:

```
using System.Drawing;
class Tester
{
   public static void Main()
   {
      using (Font theFont = new Font("Arial", 10.0f))
      {
         // użycie czcionki
      }
   }
}
```

Ponieważ system Windows pozwala na utworzenie tylko niewielkiej liczby obiektów typu `Font`, należy pozbyć się ich przy pierwszej nadarzającej się okazji. W powyższym fragmencie kodu obiekt `Font` jest tworzony za pomocą instrukcji `using`. Gdy instrukcja `using` skończy swoje działanie, zostanie wywołana metoda `Dispose()` gwarantująca wyczyszczenie pamięci po obiekcie typu `Font`.

Przydzielanie pamięci: stos kontra sterta

Obiekty tworzone w obrębie metod są nazywane *zmiennymi lokalnymi*. Są one lokalne w danej metodzie, w przeciwieństwie do zmiennych składowych, które przynależą do obiektu. Obiekt jest tworzony wewnątrz metody, jest w niej używany, a kiedy metoda kończy działanie, zostaje zniszczony. Zmienne lokalne nie stanowią części stanu obiektu — są tymczasowymi przechowalniami wartości, przydatnymi tylko w obrębie danej metody.

Lokalne zmienne typu wbudowanego, takiego jak `int`, są tworzone w części pamięci zwanej *stosem*. Przydział i zwalnianie pamięci dla stosu odbywa się w chwili wywoływania metod. Kiedy metoda rozpoczyna działanie, jej zmienne lokalne są tworzone na stosie. Kiedy metoda kończy się, zmienne lokalne są niszczone.

Zmienne takie są nazywane lokalnymi, ponieważ istnieją (i są widoczne) wyłącznie podczas działania metody. Mówi się o nich, że mają *zasięg lokalny*. Kiedy metoda kończy swoje działanie, zmienne wychodzą *poza zasięg* i są niszczone.

W języku C# zmienne zostały podzielone na dwa typy: typy wartościowe i typy referencyjne. *Typy wartościowe* są tworzone na stosie. Wszystkie typy wbudowane (`int`, `long`) są typu wartościowego (tak jak i struktury, omówione dalej w niniejszym rozdziale), a zatem są tworzone na stosie.

Klasy, z drugiej strony, są *typu referencyjnego*. Typy referencyjne tworzone są w niezróżnicowanym bloku pamięci, nazywanym *stertą*. Kiedy deklarowany jest obiekt typu referencyjnego, tak naprawdę deklarowana jest *referencja*, to znaczy zmienna wskazująca na inny obiekt. Referencja działa jak nazwa zastępcza przydzielona obiektowi.

Tak więc w instrukcji:

```
Dog milo = new Dog();
```

operator `new` tworzy na stercie obiekt typu `Dog` i zwraca do niego referencję, która jest przypisana do `milo`. A zatem `milo` jest obiektem referencyjnym odnoszącym się do umieszczonego na stercie obiektu typu `Dog`. Zwykle mówi się, że `milo` jest referencją do typu `Dog` lub że `milo` jest obiektem typu `Dog`, ale z technicznego punktu widzenia nie jest to poprawne. W rzeczywistości `milo` jest obiektem referencyjnym odnoszącym się do nienazwanego obiektu typu `Dog`, umieszczonego na stercie.

Referencja `milo` działa jak nazwa zastępcza dla tego nienazwanego obiektu. Jednak ze względów praktycznych można traktować `milo` jak gdyby rzeczywiście był obiektem typu `Dog`.

Konsekwencją stosowania typów referencyjnych jest możliwość posiadania więcej niż tylko jednej referencji wskazującej na ten sam obiekt. Przykład z listingu 7.7 ukazuje różnice między tworzeniem typów wartościowych a tworzeniem typów referencyjnych. Szczegółowa analiza kodu znajduje się po wydruku przedstawiającym wynik działania programu.

Listing 7.7. Tworzenie typów wartościowych i typów referencyjnych

```
using System;
namespace heap
{
   public class Dog
   {
      public int weight;
   }

   class Tester
   {
      public void Run()
      {
         // utworzenie zmiennej całkowitej
         int firstInt = 5;

         // utworzenie drugiej zmiennej całkowitej
         int secondInt = firstInt;

         // wyświetlenie obu zmiennych
         Console.WriteLine( "firstInt: {0} secondInt: {1}", firstInt, secondInt );

         // zmiana drugiej zmiennej
         secondInt = 7;

         // wyświetlenie obu zmiennych
         Console.WriteLine( "firstInt: {0} secondInt: {1}", firstInt, secondInt );

         // utworzenie obiektu typu Dog
         Dog milo = new Dog();

         // przypisanie wartości do wagi
         milo.weight = 5;

         // utworzenie drugiej referencji do obiektu
         Dog fido = milo;

         // wyświetlenie ich wartości
         Console.WriteLine( "milo: {0}, fido: {1}", milo.weight, fido.weight );

         // przypisanie nowej wartości drugiej referencji
         fido.weight = 7;

         // wyświetlenie obu wartości
         Console.WriteLine( "milo: {0}, fido: {1}", milo.weight, fido.weight );
      }
```

```
        static void Main()
        {
           Tester t = new Tester();
           t.Run();
        }
     }
  }
```

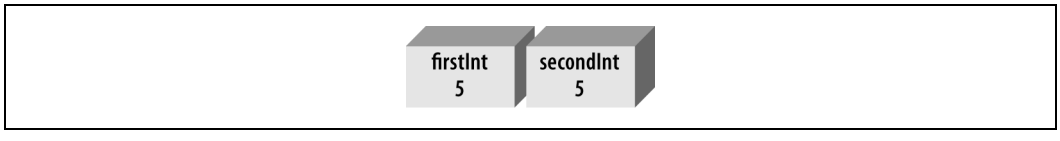
Wynik działania programu wygląda następująco:

```
firstInt: 5 secondInt: 5
firstInt: 5 secondInt: 7
milo: 5, fido: 5
milo: 7, fido: 7
```

Na początku programu tworzona jest zmienna całkowita `firstInt`, która jest inicjalizowana wartością 5. Następnie utworzona zostaje druga zmienna całkowita, `secondInt`, która zostaje zainicjalizowana wartością zmiennej `firstInt`. Wyświetlone zostają ich wartości:

```
firstInt: 5 secondInt: 5
```

Obie wartości są identyczne. Ponieważ `int` jest typem wartościowym, utworzona została kopia zmiennej `firstInt`, a jej wartość została przypisana zmiennej `secondInt`. Obie zmienne są niezależne od siebie, tak jak to pokazano na rysunku 7.2.

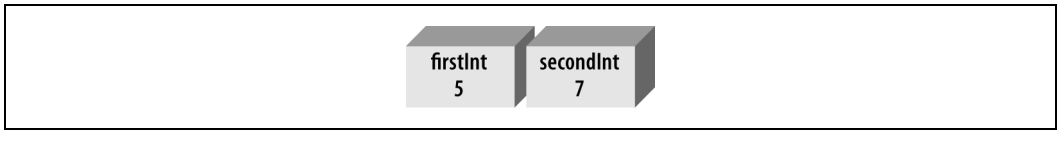


Rysunek 7.2. secondInt jest kopią firstInt

Następnie program przypisuje nową wartość zmiennej `secondInt`:

```
secondInt = 7;
```

Ponieważ obie zmienne są typu wartościowego i są niezależne od siebie, pierwsza zmienna pozostaje nienaruszona. Została zmieniona tylko kopia (rysunek 7.3).



Rysunek 7.3. Tylko kopia jest zmieniona

Kiedy zostają wyświetlone ich wartości, okazuje się, że są różne:

```
firstInt: 5 secondInt: 7
```

Następnym krokiem jest utworzenie prostej klasy `Dog` z jedną zmienną składową o nazwie `weight`. Warto zauważyć, że zmiennej nadano atrybut `public`, co oznacza, że dowolna metoda dowolnej klasy może mieć do niej dostęp. `public` to *modyfikator dostępu* (zwykle nie jest pożądane, aby zmienne składowe były publiczne, ale w tym przypadku postąpiono tak, żeby uprościć przykład).

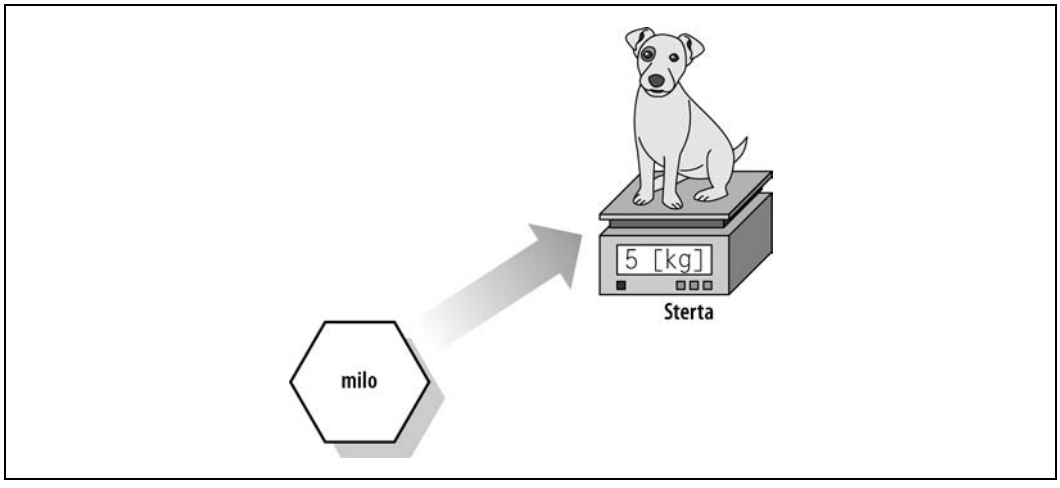
Po utworzeniu obiektu typu `Dog` referencja do tego obiektu zostaje zapisana pod nazwą `milo`:

```
Dog milo = new Dog();
```

Polu `weight` w obiekcie `milo` zostaje przypisana wartość 5:

```
milo.weight = 5;
```

Zwykle mówi się, że zmiennej `weight` obiektu `milo` nadano wartość 5, ale w rzeczywistości została ustawiona wartość nienazwanego obiektu na stercie, do którego odnosi się referencja o nazwie `milo`, tak jak to pokazano na rysunku 7.4.



Rysunek 7.4. milo jest referencją odnoszącą się do nienazwanego obiektu typu Dog

Następnie została utworzona druga referencja do obiektu typu `Dog` i zainicjalizowana wartością obiektu `milo`. W ten sposób powstała nowa referencja do tego samego obiektu na stercie:

```
Dog fido = milo;
```

Warto zauważyć, że pod względem składni przypomina to zadeklarowanie drugiej zmiennej typu `int` i zainicjalizowanie jej wartością już istniejącej zmiennej:

```
int secondInt = firstInt;
Dog fido = milo;
```

Różnica polega na tym, że `Dog` jest typem referencyjnym, więc `fido` nie jest kopią `milo`. Jest drugą referencją do tego samego obiektu, na który wskazuje `milo`. A zatem na stercie istnieje jeden obiekt, do którego odnoszą się dwie referencje, co pokazano na rysunku 7.5.

Jeśli za pośrednictwem referencji `fido` zmieniona zostanie wartość zmiennej `weight`:

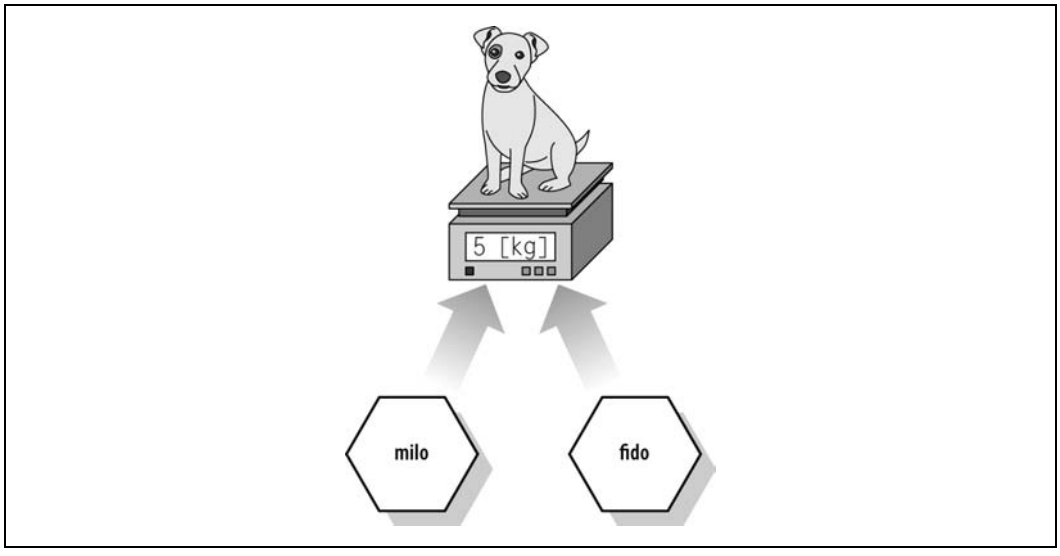
```
fido.weight = 7;
```

to zostanie zmieniona też wartość obiektu, do którego odnosi się referencja `milo`. Odzwierciedla do wynik działania programu:

```
milo: 7, fido: 7
```

Nie oznacza to, że `milo` został zmieniony przez `fido`. Dzieje się tak, ponieważ zmiana wartości nienazwanego obiektu na stercie, na który wskazuje `fido`, zmienia jednocześnie wartość `milo`, gdyż obie referencje wskazują na ten sam nienazwany obiekt.

Gdyby podczas tworzenia `fido` zostało użyte słowo kluczowe `new`, na stercie zostałaby utworzona nowa instancja typu `Dog`, a `fido` i `milo` nie wskazywałyby na ten sam obiekt typu `Dog`.



Rysunek 7.5. fido jest drugą referencją do obiektu typu Dog

Jeżeli potrzebna jest klasa, która zachowuje się jak typ wartościowy, można utworzyć strukturę (`struct`) (opisaną w ramce). Użycie struktur jest na tyle niezwykłe, że oprócz opisania ich w ramce, nie poświęciłem im więcej miejsca w tej książce. Od pięciu lat zajmuję się zawodowo pisaniem programów w C# i mój jedyny kontakt ze strukturami polegał na omawianiu ich podczas wykładów, a nie na praktycznym ich stosowaniu.

Podsumowanie

- Podczas definiowania klasy deklaruje się jej nazwę słowem kluczowym `class`, jej metody, pola, delegaty, zdarzenia i właściwości.
- Aby utworzyć obiekt, należy zadeklarować jego identyfikator poprzedzony nazwą klasy, podobnie do sposobu, w jaki deklaruje się zmienną lokalną. Następnie należy przydzielić pamięć nienazwanemu obiektowi, który zostanie umieszczony na stercie. Służy do tego słowo kluczowe `new`.
- Można zdefiniować referencję do istniejącego obiektu poprzez zadeklarowanie klasy oraz identyfikatora, a następnie przypisanie tego identyfikatora istniejącemu obiektowi. Od tej chwili oba identyfikatory będą odnosić się do tego samego, nienazwanego obiektu na stercie.
- Metodę obiektu wywołuje się podając nazwę metody z nawiasami, poprzedzoną operatorem kropki i nazwą obiektu. Jeśli metoda pobiera parametry, wypisuje się je w nawiasach.
- Modyfikatory dostępu określają, które metody mają dostęp do zmiennych i metod klasy. Wszystkie składowe danej klasy są widoczne dla wszystkich metod tej klasy.
- Składowe oznaczone słowem kluczowym `public` są widoczne dla wszystkich metod wszystkich klas.
- Składowe oznaczone słowem kluczowym `private` są widoczne tylko dla metod tej samej klasy.

Struktury

Chociaż struktury są typem wartościowym, to przypominają klasy, ponieważ mogą zawierać konstruktory, właściwości, metody, pola i operatory (wszystkie omówione w niniejszej książce). Struktury wspierają też mechanizmy indeksowania (opisane w rozdziale 12.).

Z drugiej strony, struktury nie wspierają dziedziczenia, destruktorów (rozdział 11.) ani inicjalizacji pól. Strukturę definiuje się niemal tak samo jak klasę:

```
[atrybuty] [modyfikator-dostępu] struct identyfikator [:lista-interfejsów]
{ składowe-struktury }
```

Struktury domyślnie wywodzą się z klasy `Object` (tak samo jak wszystkie typy w C#, łącznie z typami wbudowanymi), lecz nie mogą dziedziczyć z żadnych innych klas ani ze struktur (tak jak mogą klasy). Struktury są też domyślnie *zamknięte* (co znaczy, że żadna klasa ani struktura nie może po niej dziedziczyć; omówiono to w rozdziale 11.), co nie jest prawdą w przypadku klas.

Zaletą struktur miała być ich „lekkość" (wymagają mało pamięci), ale możliwości ich stosowania są tak bardzo ograniczone, a oszczędności na tyle małe, że większość programistów prawie w ogóle ich nie używa.

Uwaga programiści C++! Struktury w C++ są identyczne z klasami (z wyjątkiem dostępności), czego nie można powiedzieć o strukturach w C#.

- Składowe oznaczone słowem kluczowym `protected` są widoczne dla metod tej samej klasy i metod od niej pochodnych.
- Konstruktor jest specjalną metodą wywoływaną podczas tworzenia nowego obiektu. Jeśli nie zdefiniuje się żadnego konstruktora dla swojej klasy, to kompilator wywoła domyślny konstruktor, który nic nie robi. Konstruktor domyślny to taki konstruktor, który nie pobiera parametrów. Można tworzyć swoje własne konstruktory domyślne dla własnych klas.
- W czasie definiowania zmiennych składowych można je zainicjalizować określonymi wartościami.
- Słowo kluczowe `this` odnosi się do bieżącej instancji obiektu.
- Każda niestatyczna metoda klasy ma domyślną zmienną `this`, która jest jej przekazywana.
- Składowe statyczne są związane z daną klasą, a nie z konkretną instancją jej typu. Składowe statyczne są deklarowane słowem kluczowym `static`, a wywoływane za pośrednictwem nazwy klasy. Metody statyczne nie mogą mieć parametru `this`, ponieważ nie ma obiektu, na który mogłyby wskazywać.
- Klasy w języku C# nie wymagają destruktora, ponieważ obiekty niebędące w użyciu są usuwane automatycznie.
- W przypadku gdy klasa korzysta z niezarządzanych zasobów, należy wywołać metodę `Dispose()`.
- Zmienne typu wartościowego są tworzone na stosie. Kiedy metoda kończy swoje działanie, zmienne wychodzą poza jej zasięg i są niszczone.
- Obiekty typu referencyjnego są tworzone na stercie. W momencie deklaracji instancji typu referencyjnego tworzona jest referencja odnosząca się do miejsca, w którym dany obiekt znajduje się w pamięci komputera. Jeśli referencja zostanie utworzona w obrębie metody, to po zakończeniu działania metody referencja jest usuwana. Jeśli nie ma innych referencji odnoszących się do obiektu na stercie, to mechanizm przywracania pamięci usuwa także sam obiekt.

Pytania

Pytanie 7.1. Jaka jest różnica między klasą a obiektem?

Pytanie 7.2. Gdzie tworzone są typy referencyjne?

Pytanie 7.3. Gdzie tworzone są typy wartościowe?

Pytanie 7.4. Co oznacza słowo kluczowe `private`?

Pytanie 7.5. Co oznacza słowo kluczowe `public`?

Pytanie 7.6. Jaka metoda jest wywoływana podczas tworzenia obiektu?

Pytanie 7.7. Co to jest konstruktor domyślny?

Pytanie 7.8. Jakie typy może zwrócić konstruktor?

Pytanie 7.9. W jaki sposób inicjalizuje się wartość zmiennej składowej klasy?

Pytanie 7.10. Do czego odnosi się słowo kluczowe `this`?

Pytanie 7.11. Jaka jest różnica między metodą statyczną a metodą obiektu?

Pytanie 7.12. Jakie jest działanie instrukcji `using`?

Ćwiczenia

Ćwiczenie 7.1. Napisz program z klasą `Math`, która ma cztery metody: `Add()`, `Subtract()`, `Multiply()` i `Divide()` (każda pobierająca dwa argumenty). W metodzie `Main()` program ma wywoływać wszystkie metody.

Ćwiczenie 7.2. Zmodyfikuj program z ćwiczenia 7.1 w taki sposób, aby niepotrzebne było tworzenie obiektu typu `Math` w celu wywoływania poszczególnych metod.