# C# 3.0 dla .NET 3.5. Księga eksperta


Autor: Joseph Mayo
Tłumaczenie: Tomasz Bienkiewicz, Jacek Janusz
ISBN: 978-83-246-2141-5
Tytuł oryginału: C# 3.0 Unleashed: With the .NET Framework 3.5
Format: 172 × 245, stron: 1112
Oprawa: twarda

**Kompletne źródło wiedzy na temat C# i .NET!**

- Jak tworzyć interfejs użytkownika?
- Jak uzyskać dostęp do danych z wykorzystaniem LINQ?
- Jak wdrażać napisany kod?

C# to jeden z głównych języków, który możesz wykorzystać jeżeli chcesz tworzyć rozwiązania dla platformy .NET. Jego Nnajnowsza wersja 3.0 wniosła wprowadziła wiele udoskonaleń nowości takich jak, a wśród nich: typy domniemane, typy anonimowe, uproszczone inicjowanie obiektów oraz nowe słowa kluczowe ułatwiające korzystanie z zapytań SQL. Jednak oprócz tych nowości, w książce znajdziesz również wyczerpujący opis wszystkich elementów języka C# – począwszy od składni, skończywszy na wdrażaniu kodu. Nawiązując do najlepszych tradycji tej serii „Księga eksperta" książka „C# 3.0. Księga eksperta„ stanowi kompletne źródło wiedz na temat języka C# oraz platformy .NET w wersji 3.5.

Joe Mayo podzielił książkę na dziesięć głównych części. Wśród nich znajdziesz te poświęcone podstawom języka C#. Dowiesz się zatem, co to jest tak naprawdę platforma .NET, poznasz środowisko programistyczne Visual Studio 2008 oraz zaznajomisz się z jego e elementami, składnią, wyrażeniami i instrukcjami języka C#. Część pierwsza tworzy podwaliny Twojej przygody z C# i platformą .NET. Kolejne części zawierają coraz bardziej zaawansowaną wiedzę. Szczególną uwagę warto zwrócić na te poświęcone programowaniu obiektowemu, dostępowi do danych z wykorzystaniem LINQ, tworzeniu interfejsu użytkownika czy też wdrażaniu kodu. Jeżeli jesteś programistą C# lub chcesz rozpocząć przygodę z tym językiem i platformą .NET jest to obowiązkowa pozycja w Twojej biblioteczce!

**Twórz zaawansowane rozwiązania wykorzystując najlepsze narzędzia!**



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

# Spis treści

# Rozdział 8.
# Projektowanie obiektów

W świecie rzeczywistym, który różni się od *Second Life*[1] i innych światów wirtualnych, mamy codziennie do czynienia z wieloma rodzajami problemów. Zastanów się, w jaki sposób się je rozwiązuje. Na przykład czego potrzebowałbyś do zorganizowania specjalnego przyjęcia? Musiałbyś ustalić datę i godzinę spotkania, a także rodzaje rozrywek i zestawy potraw, zatrudnić pracowników i przygotować zaproszenia. Wszystkie te elementy musiałbyś skoordynować za pomocą planu działań.

Z punktu widzenia oprogramowania powyżej wspomniane elementy są obiektami, takimi jak klasa stworzona przez użytkownika lub struktura. Będą one następnie wzbogacone o możliwość przechowywania składników klasy, które definiują jej atrybuty oraz zachowanie. Atrybuty i sposób zachowania mogą zostać odwzorowane na elementy języka C#. Atrybut mógłby być polem lub właściwością, a zachowanie mogłoby być metodą lub zdarzeniem. Oto świat obiektowy — dzięki definiowaniu obiektów, które reprezentują fragmenty świata rzeczywistego, możesz tworzyć bardziej sensowne systemy.

W tym podrozdziale zostanie zaprezentowane, w jaki sposób tworzy się obiekty oraz definiuje ich składniki.

# Elementy obiektu

Obiekt powinien być samodzielny i zaprojektowany do realizacji jednego celu. Wszystkie jego elementy powinny być zgodne i skutecznie ze sobą współpracować, aby wykonać założone zadanie. Oto szkielet prostej klasy:

```
class WebSite
{
    // konstruktory

    // destruktory

    // pola

    // metody

    // właściwości

    // indeksatory

    // zdarzenia

    // obiekty zagnieżdżone
}
```

[1] *Second Life* — częściowo płatny wirtualny świat 3D, udostępniony publicznie w 2003 roku przez firmę Linden Lab, mieszczącą się w San Francisco — *przyp. tłum.*

W powyższym przykładzie słowo kluczowe `class` oznacza klasę — typ referencyjny zdefiniowany przez użytkownika. Słowo `WebSite` jest nazwą klasy, a jej elementy zawarte są wewnątrz nawiasów klamrowych. Zamiast klasy mogłaby zostać zdefiniowana struktura, lecz wiązałoby się to z powstaniem typu wartościowego, zgodnie z tym, co przedstawiono w rozdziale 4., zatytułowanym „Typy referencyjne i wartościowe".

### Często używane słowo: obiekt

**Słowo *obiekt* jest używany na tak wiele sposobów, że nie dziwi to, iż wprowadza ono wielu ludzi w zakłopotanie. W informatyce obiekt jest przedmiotem, który reprezentuje pewną jednostkę w obszarze problemu do rozwiązania. Według terminologii .NET definicja obiektu zwana jest często typem. W programowaniu zorientowanym obiektowo obiekt jest konkretyzacją (instancją) typu. W języku C# słowo `object` oznacza typ bazowy dla wszystkich innych typów.**

**W tym rozdziale używam powyżej wspomnianej definicji informatycznej, aby zaprezentować, w jaki sposób w języku C# należy tworzyć obiekty. Możesz odwiedzić stronę *http://pl.wikipedia.org/wiki/Obiekt*, aby dowiedzieć się, ile znaczeń ma słowo *obiekt*.**

W kolejnych podrozdziałach zapoznasz się szczegółowo ze wszystkimi składnikami obiektu. Są to między innymi pola, konstruktory, destruktory, metody, właściwości, indeksatory i zdarzenia.

## Elementy statyczne i instancyjne

Każdy składnik obiektu może zostać przypisany do jednej z dwóch kategorii: elementu statycznego lub instancyjnego. Gdy tworzona jest kopia obiektu, powstaje jego nowy egzemplarz. W tym przypadku obiekt jest całkowicie niezależną jednostką, posiadającą swój własny zestaw atrybutów i sposób zachowania. Gdyby została stworzona druga instancja obiektu, miałaby ona niezależny zestaw danych w porównaniu z pierwszym egzemplarzem. W języku C# składniki obiektu należą domyślnie do instancji, chyba że zostaną zdefiniowane jako statyczne. Przykładem egzemplarza obiektu jest klasa `Customer`, która przechowuje odmienne informacje dla każdego klienta.

Jeśli użyjesz statycznego modyfikatora dla jakiegoś składnika obiektu, w danym momencie będzie istnieć tylko jedna jego kopia, bez względu na to, ile stworzono egzemplarzy obiektu. Elementy statyczne są przydatne podczas definiowania dostępu do pól statycznych (stanu obiektu). W bibliotece klas platformy .NET (FCL) można znaleźć wartościowe przykłady użycia metod statycznych, takie jak klasy `System.Math`, `System.IO.Path` i `System.IO.Directory`. Każda z nich ma elementy statyczne, które przetwarzają dane wejściowe i zwracają wartość. Ponieważ nie zależą one od żadnego stanu obiektu, wygodne jest zdefiniowanie ich jako składników statycznych, co zapobiega powstaniu narzutu podczas tworzenia instancji.

# Pola

Pola zawierają zasadniczą część danych w klasie. Określają one stan obiektu. Są elementami klasy w przeciwieństwie do lokalnych zmiennych, które są definiowane wewnątrz metod i właściwości.

Pola mogą być inicjalizowane podczas deklaracji lub później, w zależności od stylu lub rodzaju wymagań. Każdy ze sposobów inicjalizacji ma swoje wady i zalety.

Na przykład ostrożną metodą może być zapewnienie, że wszystkie pola będą zawierały domyślne wartości, co wymusza ich inicjalizację podczas deklaracji lub zaraz po niej. Jest to zachowanie bezpieczne, a być może również zasadniczo wspomaga planowanie projektu poprzez żądanie zastanowienia się nad właściwościami definiowanych danych. Oto przykład deklaracji pola:

```
string siteName = "Zaawansowana strona o bezpieczeństwie komputerowym";
```

Deklaracja pola i jego inicjalizacja mogą zostać przeprowadzone w jednym wierszu. Nie jest to jednak koniecznie wymagane. Pola mogą być deklarowane w jednym wierszu, a inicjalizowane później, jak przedstawiono na poniższym przykładzie:

```
string url;

// gdzieś w dalszej części kodu
url = "http://www.comp_sec-mega_site.com";
```

W razie konieczności w jednym wierszu można deklarować wiele pól. Muszą one być od siebie oddzielane przecinkami. Takie wiersze mogą nawet zawierać deklarację jednego lub większej liczby pól, jak zaprezentowano w poniższym przykładzie:

```
string siteName, url, description = "Informacje o bezpieczeństwie
↪komputerowym";
```

Wszystkie trzy powyżej zadeklarowane pola są łańcuchami. Pole `description` zostało zainicjalizowane przy użyciu łańcucha literalnego. Inne pola są wciąż niezainicjalizowane, mogłyby jednak zostać zainicjalizowane w taki sam sposób, jak pole `description`.

## Pola stałe

Gdy wartość pola jest znana wcześniej i nie zmienia się, możesz stworzyć stałą. Pole stałe gwarantuje, że jego wartość nie zostanie zmieniona podczas wykonania programu. Może być ono wielokrotnie odczytywane, nie można jednak do niego zapisywać ani w jakikolwiek sposób go modyfikować.

Pola stałe są wydajne. Ich wartości są znane podczas kompilacji. Pozwala to na zastosowanie pewnych optymalizacji niedostępnych dla innych rodzajów pól. Pola stałe są również z definicji statyczne. Oto przykład:

```
const string http = "http://";
```

W powyższym przykładzie przedstawiono deklarację stałej łańcuchowej zainicjalizowanej łańcuchem literalnym. Stałe mogą być inicjalizowane wartościami literalnymi. Jest to dobry wybór, ponieważ takie wartości się nie zmieniają. Zastanów się nad sposobem, w jaki czasami wprowadza się adresy internetowe w przeglądarce: użytkownik podaje fragment adresu, zakładając, że protokół internetowy będzie zgodny ze standardami WWW. Prostą metodą uzyskania tej funkcjonalności jest zadeklarowanie pola stałego definiującego protokół HTTP jako domyślny przedrostek dla wszystkich adresów sieciowych.

Stałe całkowite mogą być implementowane przy użyciu słowa kluczowego `const`, lecz często dużo wygodniejsze jest definiowanie ich w postaci typów wyliczeniowych. Ich użycie wymusza również implementację o ściślejszej kontroli typu. W rozdziale 6., zatytułowanym „Użycie tablic i typów wyliczeniowych", omówiono szczegółowo typy wyliczeniowe.

## Pola readonly

Pola `readonly` są podobne do pól stałych, ponieważ nie mogą być modyfikowane po zainicjalizowaniu. Największą różnicą między tymi dwoma rodzajami pól jest moment inicjalizacji: stałe są inicjalizowane podczas kompilacji, natomiast pola `readonly` po uruchomieniu programu. Istnieją pewne powody, aby tak postępować, między innymi uniwersalność i udostępnianie większej funkcjonalności dla użytkowników.

Czasami wartość zmiennej pozostaje nieznana aż do uruchomienia programu. Może ona zależeć od różnych warunków i logiki programu. Pola `readonly` są inicjalizowane podczas tworzenia egzemplarza obiektu.

Pole `currentDate` w poniższym przykładzie zostaje zainicjalizowane wartością równą czasowi jego utworzenia:

```
readonly DateTime currentDate = DateTime.Now;
```

Ponieważ czas utworzenia obiektu jest wartością, której nie można ustalić podczas kompilacji programu, użycie modyfikatora `readonly` jest najodpowiedniejszą metodą rozwiązania tego przypadku.

# Metody

Metody będą jednymi z najczęściej używanych przez Ciebie składników obiektów. Metoda w języku C# jest zbliżona do funkcji, procedur, podprogramów i tym podobnych elementów w innych językach programowania. O metodach C# można powiedzieć bardzo dużo — więcej dowiesz się o nich w rozdziale 10., zatytułowanym „Metody kodowania i operatory tworzone przez użytkownika". Teraz przedstawiam prosty przykład:

```
void MyMethod()
{
    // instrukcje programu
}
```

Powyższa metoda nie zwraca żadnej wartości, dlatego zwracany typ jest równy `void`. Metoda nazywa się `MyMethod`. Powinieneś jednakże nadawać metodom takie nazwy, które reprezentują ich działanie. Zaprezentowana metoda nie ma żadnych parametrów, lecz w dalszym ciągu do nazwy musisz dołączać nawiasy okrągłe. Blok kodu występujący wewnątrz nawiasów klamrowych jest treścią metody.

# Właściwości

Właściwości w języku C# pozwalają na ochronę dostępu do stanu Twojego obiektu. Możesz używać ich jak pól, lecz działają one bardziej jak metody. W poniższych podrozdziałach zostanie zaprezentowane, w jaki sposób należy deklarować i używać właściwości. Poznasz również nową opcję wersji 3.0 języka C#, zwaną właściwością automatyczną (ang. *autoimplemented property*), oraz dowiesz się, jak w środowisku VS2008 należy używać gotowego fragmentu kodu dla właściwości.

## Deklarowanie właściwości

Oto przykład prostej właściwości:

```
private string m_description;

public string Description
{
   get
   {
      return m_description;
   }

   set
   {
      m_description = value;
   }
}
```

Właściwość rozpoczyna się od modyfikatora dostępu `public`. Oznacza to, że kod znajdujący się poza aktualną klasą będzie miał dostęp do tej właściwości. Kolejnym elementem jest typ właściwości: `string`. Nazwą właściwości jest `Description`. Zawiera ona oba akcesory: `get` i `set`.

Akcesory `get` i `set` mogą zawierać dowolnie zdefiniowaną przez użytkownika logikę. Akcesor `get` zwraca wartość, natomiast akcesor `set` ustawia ją. Zwróć uwagę na obecność słowa kluczowego `value` w akcesorze `set`; przechowuje ono dowolną wartość, która została przypisana do właściwości.

## Użycie właściwości

Oto przykład prezentujący użycie właściwości:

```
static void Main()
{
WebSite site = new WebSite();

site.Description = "świetna strona";
string desc = site.Description;
}
```

Załóżmy, że właściwość `Description` została zdefiniowana wewnątrz klasy `WebSite`. Możesz następnie stworzyć egzemplarz tej klasy — w powyższym kodzie jest to obiekt `site`. Poprzez niego masz dostęp do właściwości `Description`. Pamiętaj, że została ona zdefiniowana jako `public`, co oznacza, że jest widoczna poza klasą `WebSite`. Element `m_description` został jednak zdefiniowany jako `private`, dlatego nie jest on dostępny dla powyższego kodu.

Zwróć uwagę na to, w jaki sposób do elementu `site.Description` został przypisany łańcuch `"świetna strona"`. Gdy ta operacja ma miejsce, następuje wywołanie akcesora `set` dla właściwości `Description`. Oprócz tego słowo kluczowe `value` wewnątrz akcesora `set` przechowuje przypisany łańcuch `"świetna strona"`.

Następnie przyjrzyj się, jak do zmiennej `desc` przypisano wartość elementu `site.Description`. Podczas tej operacji wywołany zostaje akcesor `get` dla właściwości `Description`. Zwraca on wartość, która została przypisana do pola `m_description`. Zostaje ona z kolei przypisana do zmiennej `desc`.

Powyższy przykład prezentuje, w jaki sposób właściwość może używać pojedynczego pola jako swojej lokalnej pamięci pomocniczej. Wykonywana jest tylko operacja ustawiania lub pobierania wartości pola z tej pamięci — jest to powszechnie stosowany sposób działania. W rozdziale 9., zatytułowanym „Implementacja reguł zorientowanych obiektowo", dowiesz się więcej o obiektowo zorientowanej zasadzie hermetyzacji, która wyjaśni, dlaczego powinno się używać właściwości w powyżej zaprezentowany sposób, zamiast bezpośrednio sięgać do pól klasy. W pierwszym rzędzie będziesz chciał zapewnić rozłączność obiektów oraz poprawić zarządzanie kodem — powyżej zalecane użycie właściwości jest w tym bardzo pomocne. W następnym podrozdziale przedstawiony zostanie łatwiejszy sposób rozwiązania prostego zadania polegającego jedynie na ustawianiu i pobieraniu wartości z lokalnej pamięci pomocniczej.

## Właściwości automatyczne

Wykorzystanie właściwości obejmujących swoim zasięgiem jedynie pojedyncze pole jest tak powszechne, że w wersji 3.0 języka C# wprowadzono właściwości automatyczne. Oto przykład:

```
public int Rating { get; set; }
```

Właściwość `Rating` dotyczy pewnej wartości o typie `int`. Ponieważ wywołujący kod nie ma dostępu do lokalnej zmiennej pomocniczej, dlatego jej nazwa nie będzie mu do niczego potrzebna. Kompilator języka C# automatycznie utworzy pole o typie `int`.

Inną korzyścią z używania właściwości automatycznej jest eliminowanie pokusy pojawiającej się u programistów i polegającej na zajmowaniu się kodowaniem na poziomie lokalnej pamięci pomocniczej, zamiast obsługi samej właściwości. Na przykład w przypadku właściwości `Des ↪criptio`n kod znajdujący się w definiującej ją klasie ma dostęp do pola `m_description`. Może się jednak zdarzyć, że po jakimś czasie zmodyfikujesz kod implementujący tę właściwość, dzięki czemu będzie on zawierał reguły biznesowe przetwarzające wartość, która została do niej przypisana. Możliwe będzie, że inny kod w tej samej klasie nie zostanie odpowiednio zmieniony, aby używać zaimplementowanych reguł biznesowych, co spowoduje powstanie błędu w oprogramowaniu.

## Gotowy fragment kodu dla właściwości w środowisku VS2008

W środowisku VS2008 istnieje kolejny przydatny snippet, którego możesz użyć podczas kodowania właściwości. Oto sposób jego wykorzystania:

1. Kliknij myszą wewnątrz klasy `WebSite`, aby umieścić tam kursor (jeśli Twój projekt nie zawiera klasy `WebSite`, stwórz ją).
2. Wprowadź z klawiatury ciąg znaków `pro` i naciśnij *Tab*, co spowoduje, że w edytorze pojawi się słowo `prop`. Dla właściwości istnieją również inne gotowe fragmenty kodów, lecz w tej chwili chcemy zająć się jedynie kodem `prop`.
3. Ponownie naciśnij klawisz *Tab*, aby uzyskać szkielet kodu dla właściwości. Podświetlone zostanie pole typu, które domyślnie równe jest `int`.
4. Wpisz `Web`, a następnie naciśnij *Tab*. W polu typu pojawi się słowo `WebSite`.
5. Naciśnij klawisz *Tab*, co spowoduje, że kursor przemieści się do następnego pola, którym będzie nazwa właściwości.
6. Wprowadź z klawiatury `BetaSite` i naciśnij klawisz *Enter*. Kursor przemieści się na koniec fragmentu kodu.

Stworzyliśmy właściwość automatyczną. Snippet `propg` tworzy właściwość z akcesorem `get`, a fragmenty kodu `propa` i `propdp` tworzą właściwości dołączone i zależne, których używa się w aplikacjach wykorzystujących systemy Windows Presentation Foundation i Windows Workflow. Zostaną one zaprezentowane w rozdziale 26., zatytułowanym „Tworzenie aplikacji Windows Presentation Foundation (WPF)", i rozdziale 28., zatytułowanym „Wykorzystanie ASP.NET AJAX w aplikacjach sieciowych".

# Indeksatory

Indeksatory pozwalają na tworzenie obiektów, które mogą być następnie używane jak tablice. Można potraktować ich implementację jako połączenie tablicy, właściwości i metody.

Indeksatory zachowują się jak tablice, ponieważ używają składni wykorzystującej nawiasy kwadratowe, aby uzyskać dostęp do ich elementów. Zestaw klas platformy .NET używa indeksatorów w tym samym celu. Ich elementy są dostępne za pomocą indeksów.

Indeksatory zostały zaimplementowane jak właściwości, ponieważ wykorzystują akcesory `get` i `set` oraz ich składnię. Za pomocą akcesora `get` zwracają odpowiednią wartość wskazywaną przez indeks. Podobnie też przy użyciu akcesora `set` ustawiają wartość odpowiadającą danemu indeksowi.

Indeksatory wykorzystują również listę parametrów, podobnie jak ma to miejsce w przypadku metod. Lista parametrów jest zawarta w nawiasach okrągłych. Parametry mają zazwyczaj typ `int`, dzięki czemu klasa może udostępniać operacje podobne do tablic. Innymi użytecznymi typami są również `string` i `enum`. Oto przykład:

```
const int MinLinksSize = 0;
const int MaxLinksSize = 10;
string[] m_links = new string[MaxLinksSize];

public string this[int i]
{
   get
   {
      if (i >= MinLinksSize && i < MaxLinksSize)
      {
         return m_links[i];
      }
      return null;
   }

   set
   {
      if (i >= MinLinksSize && i < MaxLinksSize)
         {
            m_links[i] = value;
         }
   }
}

// kod w innej klasie

static void Main()
{
   WebSite site = new WebSite();
```

```
        site[0] = "http://www.mysite.com";
        string link = site[0];
    }
```

Indeksator z powyższego przykładu używa parametru całkowitoliczbowego. Akcesory `get` i `set` chronią przed próbami dostępu do elementów spoza zakresu.

Sposób użycia tego indeksatora jest bardzo podobny do wykorzystania tablicy. Pod koniec powyższego przykładu pojawia się instancja klasy `WebSite` będąca obiektem zawierającym indeksator. Podobnie jak ma to miejsce w przypadku właściwości, odczytywanie indeksatora wywołuje akcesora `get`, natomiast przypisywanie do indeksatora wywołuje akcesora `set`. Liczba znajdująca się w nawiasach jest przekazywana do parametru `i`, który wykorzystuje się w akcesorach; w powyższym przykładzie parametr `i` ma wartość `0`. Słowo kluczowe `value` przechowuje przypisaną wartość, która w tym przypadku jest równa `"http://www.mysite.com"`.

# Gdzie mogą zostać użyte typy częściowe?

Typy częściowe, wprowadzone w wersji 2.0 języka C#, pozwalają na podział definicji pojedynczego typu na wiele części. Mimo że części mogą znajdować się w tym samym pliku, służą one zazwyczaj do umieszczenia definicji obiektu w wielu plikach. Podstawowym zadaniem typów częściowych jest wsparcie w oddzielaniu kodu wygenerowanego maszynowo od kodu tworzonego przez programistę. Na przykład w środowisku VS2008 kreatory projektów i elementów wykorzystujących ASP.NET i Windows Form tworzą szkielet klas umieszczonych w dwóch plikach. Zmniejsza to ilość kodu, z którym musisz bezpośrednio pracować, ponieważ Twoja część znajduje się w jednym pliku, natomiast w drugim zawarty jest segment wygenerowany maszynowo.

Składnia identyfikująca typ częściowy zawiera definicję klasy (lub struktury) z użytym modyfikatorem `partial`. W czasie kompilacji język C# identyfikuje wszystkie klasy zdefiniowane z tym samym modyfikatorem `partial` i łączy je w pojedynczy typ. W poniższym kodzie zaprezentowano składnię typów częściowych:

```
using System;

partial class Program
{
    static void Main()
    {
        m_someVar = 5;
    }
}

// kod umieszczony w innym pliku

using System;

partial class Program
```

```
{
   private static int m_someVar;
}
```

Powyższy kod znajduje się w dwóch różnych plikach. Drugi plik rozpoczyna się od drugiego użycia instrukcji `using System`. W prosty sposób zaprezentowałem tu deklarację typu częściowego, której obie części zawierają modyfikator `partial`. Zwróć uwagę na to, że pole `m_someVar` zostało zadeklarowane w jednej części, a użyte w metodzie `Main` znajdującej się w drugiej części. Podczas działania programu obie części tworzą jedną klasę, co nie stanowi żadnego problemu.

Zaprezentowana została podstawowa składnia typu częściowego; z jego działaniem będziesz mógł zapoznać się w rozdziale 25. „Tworzenie aplikacji w oparciu o formularze Windows Forms", rozdziale 26. „Tworzenie aplikacji Windows Presentation Foundation (WPF)" i rozdziale 27. „Tworzenie aplikacji sieciowych za pomocą ASP.NET".

# Klasy statyczne

Standardowe klasy mogą zawierać elementy instancyjne i statyczne. Czasem jest jednak wymagane, aby klasa składała się jedynie z elementów statycznych. W tym przypadku możesz utworzyć klasę statyczną. Oto przykład:

```
public static class CustomMathLib
{
   public static double DoAdvancedCalculation(double param1, double param2)
   {
      return -1;
   }
}
```

Jak przedstawiono powyżej, w celu utworzenia klasy statycznej wystarczy użyć modyfikatora `static`. Wynika stąd, że wszystkie elementy klasy muszą również być statyczne.

# Klasa System.Object

Jak dowiedziałeś się już w rozdziale 4., wszystkie typy pochodzą z klasy `System.Object`. Ze względu na ten związek dziedziczenia wszystkie obiekty zawierają w sobie także elementy tej klasy. W tym podrozdziale zostanie zaprezentowane, czym są te elementy i jak należy ich używać.

## Sprawdzanie typu obiektu

W rozdziale 2., zatytułowanym „Wprowadzenie do języka C# i środowiska Visual Studio 2008", poznałeś operator `typeof`. Przyjrzałeś się jego praktycznemu wykorzystaniu w rozdziale 6. podczas pracy z klasą `Enum`. Cechą operatora `typeof` jest to, że wymaga on znajomości typu użytego

parametru. Czasem jednakże będziesz miał do czynienia z obiektem, którego typu być może nie będziesz znać. Na przykład podczas użycia ogólnej metody z parametrem typu obiektowego będziesz chciał sprawdzić, czy aktualnie używany typ jest prawidłowy. Z taką sytuacją spotkasz się w rozdziale 9. podczas własnej implementacji metody `Equals`. Aby uzyskać typ egzemplarza obiektu, możesz wywołać jego metodę `GetType`:

```
Type siteType = site.GetType();
```

W powyższym przykładzie wywołano metodę `GetType` dla instancji `WebSite`. Być może jest to klasa pochodząca z `WebSite`, dla której należy poznać dokładny typ.

## Porównywanie referencji

Inną metodą klasy `System.Object` jest `ReferenceEquals`, która działa z obiektami o typie referencyjnym. Dzięki niej będziesz mógł się dowiedzieć, czy dwie zmienne zawierają referencję do tego samego obiektu. Oto przykład:

```
WebSite site2 = site;
bool isSameObject = object.ReferenceEquals(site, site2);
```

Ponieważ w powyższym kodzie przypisano zmienną `site` do `site2`, ich referencje będą takie same, dlatego wywołanie metody `ReferenceEquals` zwróci `true`. Mógłbyś mieć dwa odmienne obiekty o tych samych wartościach, lecz metoda `ReferenceEquals` zwróciłaby w tym przypadku `false`, ponieważ referencje nie byłyby sobie równe.

## Sprawdzanie równości

Przeznaczeniem metody `Equals` jest sprawdzanie, czy wartości są sobie równe. Dla obiektów o typie wartościowym metoda ta sprawdza automatycznie wszystkie ich elementy. W przypadku obiektów o typie referencyjnym metoda `Equals` wywołuje jednakże `ReferenceEquals`, która zwraca informację o równości referencji. W rozdziale 9. zostanie przedstawione, w jaki sposób do swoich obiektów możesz dodać metodę `Equals`, aby zdefiniować równość wartości. Oto kilka przykładów prezentujących użycie instancyjnych i statycznych metod `Equals` z klasy `System.Object`:

```
WebSite site3 = new WebSite();
WebSite site4 = new WebSite();
site3.Description = "Informacje o C#";
site4.Description = site3.Description;

bool isSiteEqual = site3.Equals(site4);
isSiteEqual = object.Equals(site3, site4);
```

W powyższym przykładzie właściwość `site4.Description` została ustawiona na wartość właściwości `site3.Description`, dzięki czemu oba obiekty uzyskały taką samą wartość. Nie ma to jednakże znaczenia, ponieważ oba wywołania metody `Equals` zwracają `false`. Dzieje się tak, gdyż zmienne odnoszą się do dwóch różnych obiektów. W rozdziale 9. pokażę, jak można to naprawić.

## Uzyskiwanie wartości mieszających

Często wykorzystywaną operacją podczas programowania w języku C# jest praca z tablicami mieszającymi (ang. *hash tables*), które przechowują obiekty w oparciu o ich niezmienne klucze. Tablice te są również nazywane w innych językach asocjacjami, tablicami asocjacyjnymi lub słownikami. Aby wspomóc tworzenie kluczy, klasa `System.Object` zawiera metodę `GetHashCode` przedstawioną poniżej:

```
int hashCode = site3.GetHashCode();
```

W zasadzie nie będziesz wywoływać metody `GetHashCode` w powyżej przedstawiony sposób, chyba że zamierzasz zaimplementować swoją własną tablicę mieszającą. Zostanie ona wywołana podczas użycia klasy `Hashtable` lub ogólnej klasy `Dictionary`, co zaprezentowano w rozdziale 17., zatytułowanym „Parametryzowanie typów poprzez szablony klas i tworzenie iteratorów". Domyślna implementacja metody `GetHashCode` z klasy `System.Object` nie gwarantuje zwracania unikalnej wartości, jednak w rozdziale 9. dowiesz się, jak można zdefiniować własną metodę tego typu.

## Klonowanie obiektów

Klasa `System.Object` zawiera także metodę zwaną `MemberwiseClone` tworzącą kopie obiektów. Oto przykład:

```
// element klasy WebSite
public WebSite GetCopy()
{
   return MemberwiseClone() as WebSite;
}
```

Ponieważ metoda `MemberwiseClone` została zdefiniowana z modyfikatorem `protected`, musiałem wywołać ją z wnętrza klasy `WebSite`. Oznacza to, że dostęp do tej metody mają tylko klasy pochodne. Więcej o użyciu modyfikatora `protected` i innych sposobach definiowania dostępu dowiesz się w rozdziale 9. Oto kod, który wykorzystuje tę metodę:

```
WebSite beta = new WebSite();
WebSite site5 = new WebSite();
site5.BetaSite = beta;

WebSite site6 = site5.GetCopy();

bool areSitesEqual = ReferenceEquals(site5, site6);
bool areBetasEqual = ReferenceEquals(site5.BetaSite, site6.BetaSite);

Console.WriteLine("Równość obiektów site: {0}, równość właściwości beta: {1}",
↪areSitesEqual, areBetasEqual);
```

Zwróć uwagę na to, że właściwość `BetaSite` dla obiektu `WebSite` została ustawiona na wartość `beta`. Oznacza to, że obiekt `WebSite` zawiera pole z referencją do innej instancji klasy `WebSite`. Podczas wywołania metody `GetCopy`, która z kolei wywołuje `MemberwiseClone`, do zmiennej `site6` przypisany zostaje nowy obiekt będący kopią `site5`.

Oto coś, co może Cię zaskoczyć: metoda `MemberwiseClone` przeprowadza tylko płytkie kopiowanie. Płytkie kopiowanie wykonuje kopię obiektów wyłącznie na pierwszym poziomie grafu obiektowego. Obie zmienne `site5` i `site6` znajdują się właśnie na tym poziomie grafu obiektowego. Instancja `beta` klasy `Website` znajduje się jednakże w obiekcie `site5` na drugim poziomie grafu obiektowego. Oznacza to, że jedynym elementem, który zostanie skopiowany podczas wywołania metody `MemberwiseClone`, będzie referencja przechowywana we właściwości `Beta ↪Site` obiektu `site5`. Tak więc mimo że `site5` i `site6` są odmiennymi obiektami, które naprawdę zostały skopiowane, jednakże odwołują się do tego samego obiektu przypisanego do `BetaSite`. Poniżej przedstawiony wynik działania kodu potwierdza tę uwagę:

```
Równość obiektów site: False, równość właściwości beta: True
```

Zapamiętaj, że metoda `MemberwiseClone` wykonuje płytkie kopiowanie. W celu przeprowadzenia głębokiego kopiowania będziesz musiał zaimplementować własną metodę.

## Używanie obiektów jako łańcuchów

Być może do tej pory zwróciłeś uwagę na jeden fakt: metodę `ToString` możesz wywołać dla dowolnego elementu. Metoda `Console.WriteLine` wywołuje domyślnie `ToString` dla każdego parametru, który zostanie do niej dostarczony. Dzieje się tak, ponieważ `ToString` jest elementem klasy `System.Object`. Oto kilka przykładów:

```
string siteStr = site6.ToString();
string fiveStr = 5.ToString();

Console.WriteLine("site6: {0}, pięć: {1}", siteStr, fiveStr);
```

Do klasy `WebSite` nie dołączyliśmy metody `ToString`, lecz mimo to może ona zostać wywołana. Możesz również wywołać ją dla wartości literalnej, jak zaprezentowano w powyższym przykładzie w przypadku liczby `5`. Oto wynik działania kodu:

```
site6: Rozdzial_08.WebSite, pięć: 5
```

Wynik działania dla liczby `5` nie jest zaskakujący, lecz zauważ, co uzyskano dla zmiennej `site6`. `Rozdzial_08` jest przestrzenią nazw programu, której używam w tym rozdziale, a `WebSite` jest nazwą klasy. Jest to pełna kwalifikowana nazwa typu, którą domyślnie otrzymasz za pomocą metody `ToString`.

### Domyślna wartość ToString podczas debugowania programu

**Może się zdarzyć, że będziesz debugował swój program lub wydrukujesz wyniki zwracane przez metodę `ToString`, i zobaczysz pełną kwalifikowaną nazwę typu. Może to oznaczać, że obserwujesz lub wykorzystujesz niewłaściwy obiekt.**

**Na przykład w środowisku ASP.NET możesz do list przypisać obiekty `ListItem`, które zawierają właściwości `Text` i `Value`. Jednakże przypadkiem mógłbyś wywołać dla obiektu `ListItem` metodę `ToString`, zamiast właściwości `Text`, co było Twoim pierwotnym zamiarem.**

# Podsumowanie

Zapoznałeś się już z elementami klas i wiesz już, z czego składa się obiekt. Właściwości są przydatne podczas opisywania stanu obiektów. Możesz również używać obiektów jak tablic dzięki zaimplementowaniu indeksatorów.

Środowisko VS2008 wykorzystuje typy częściowe dla ułatwienia pracy z technologiami ASP.NET i Windows Forms dzięki umieszczaniu klas w różnych plikach.

Klasa `System.Object`, która — jak już wiesz — jest podstawową klasą dla wszystkich typów języka C#, zawiera elementy mogące zostać wykorzystane przez Ciebie. Możesz porównywać obiekty pod względem równości referencji i wartości. W następnym rozdziale poznasz różne koncepcje związane z programowaniem obiektowym, między innymi polimorfizm. Możesz go wykorzystać, aby przesłonić `Equals` i inne metody pochodzące z `System.Object`, a przez to uzyskać bardziej funkcjonalne klasy.