helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

# C#. Programowanie




Autor: Jesse Liberty
Tłumaczenie: Tomasz Walczak
ISBN: 83-246-0213-5
Tytuł oryginału: Programming C#
Format: B5, stron: 600

**Podręcznik podstawowego języka programowania dla platformy .NET**

- Poznaj elementy języka C#
- Stwórz własną aplikację
- Komunikuj się z bazami danych

Język C# został opracowany przez firmę Microsoft jako podstawowe narzędzie programistyczne dla platformy .NET. C# łączący w sobie najlepsze cechy języków C, C++, Java i Visual Basic doskonale nadaje się do tworzenia aplikacji bazujących na komponentach. Jest prosty, bezpieczny ze względu na typy, oparty na obiektach i komponentach i przygotowany do obsługi mechanizmów komunikacji sieciowej. Za pomocą C# tworzone są nie tylko aplikacje na „duże" komputery, ale także programy dla platformy PocketPC. Popularność C# nadal rośnie, a jego producent stale inwestuje w dalszy rozwój tego narzędzia i platformy .NET.

„C#. Programowanie" to podręcznik przedstawiający tajniki tworzenia aplikacji dla .NET w języku C#. Przeczytasz w nim o podstawowych elementach i strukturach C#, zasadach programowania obiektowego oraz obsłudze wyjątków i błędów. Poznasz sposoby tworzenia aplikacji z wykorzystaniem podstawowych technologii tworzenia aplikacji dla platformy .NET – ASP.NET, Windows Forms i ADO.NET. Nauczysz się wykorzystywać metadane i łączyć swoje aplikacje z API Windows oraz obiektami COM.

- Typy w języku C#
- Zmienne i stałe
- Instrukcje, operatory i dyrektywy preprocesora
- Reguły programowania obiektowego
- Operacje na łańcuchach znaków
- Stosowanie wyrażeń regularnych
- Tworzenie aplikacji dla systemu Windows
- Łączenie z bazami danych za pomocą ADO.NET
- Pisanie aplikacji i usług sieciowych
- Zdalne korzystanie z obiektów
- Wątki i synchronizacja
- Operacje na plikach

**Poznaj nowoczesne techniki programowania**

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

# Spis treści

**Przedmowa** ..... **9**

## Część I Język C# 17

**1. Język C# i platforma .NET** ..... **19**
Platforma .NET 19
.NET Framework 20
Kompilacja i język MSIL 22
Język C# 22

**2. Pierwszy program — „Witaj świecie”** ..... **25**
Klasy, obiekty i typy 25
Tworzenie programu „Witaj świecie” 31
Usuwanie błędów w Visual Studio .NET 36

**3. Podstawy języka C#** ..... **39**
Typy 39
Zmienne i stałe 43
Wyrażenia 50
Odstępy 51
Instrukcje 51
Operatory 65
Dyrektywy preprocesora 73

**4. Klasy i obiekty** ..... **77**
Definiowanie klas 78
Tworzenie obiektów 82
Używanie składowych statycznych 88
Usuwanie obiektów 92
Przekazywanie parametrów 95
Przeciążanie metod i konstruktorów 101

Hermetyzacja danych za pomocą właściwości 103
Pola tylko do odczytu 107

**5. Dziedziczenie i polimorfizm ..... 109**
Specjalizacja i uogólnianie 109
Dziedziczenie 111
Polimorfizm 112
Klasy abstrakcyjne 118
Klasa główna — Object 122
Pakowanie i rozpakowywanie typów 124
Zagnieżdżanie klas 126

**6. Przeciążanie operatorów ..... 129**
Używanie słowa kluczowego operator 129
Obsługa innych języków platformy .NET 130
Tworzenie użytecznych operatorów 130
Pary logiczne 130
Operator równości 131
Operatory konwersji 131

**7. Struktury ..... 137**
Definiowanie struktur 137
Tworzenie struktur 140

**8. Interfejsy ..... 145**
Definiowanie i implementowanie interfejsu 146
Dostęp do metod interfejsu 154
Przesłanianie implementacji interfejsu 160
Jawna implementacja interfejsu 164

**9. Tablice, kolekcje i mechanizm indeksowania ..... 173**
Tablice 173
Instrukcja foreach 178
Mechanizm indeksowania 191
Interfejsy kolekcji 199
Ograniczenia 202
Klasa List<T> 207
Kolejki 217

Stosy 219
Słowniki 221

**10. Łańcuchy znaków i wyrażenia regularne ... 225**
Łańcuchy znaków 225
Wyrażenia regularne 239

**11. Obsługa wyjątków ... 251**
Zgłaszanie i przechwytywanie wyjątków 252
Wyjątki jako obiekty 261
Własne wyjątki 263
Ponowne zgłaszanie wyjątków 266

**12. Delegaty i zdarzenia ... 271**
Delegaty 272
Delegaty zbiorowe 281
Zdarzenia 285
Używanie anonimowych metod 296
Pobieranie wartości delegatów zbiorowych 297

**Część II Programowanie w języku C# 305**

**13. Tworzenie aplikacji dla systemu Windows ... 307**
Tworzenie prostego formularza 308
Tworzenie aplikacji dla systemu Windows 313
Komentarze w stylu języka XML 334

**14. Obsługa danych za pomocą ADO.NET ... 337**
Relacyjne bazy danych i SQL 337
Model obiektowy ADO.NET 341
Używanie ADO.NET 343
Używanie zarządzanych dostawców danych OLE DB 345
Używanie kontrolek DataBound 347

**15. Tworzenie aplikacji ASP.NET i Web Services ... 355**
Czym jest Web Forms? 356
Tworzenie formularza Web Form 360
Dodawanie kontrolek 363

Wiązanie danych 366
Usługi Web Service 375
SOAP, WSDL i Discovery 375
Tworzenie usługi Web 376
Tworzenie pośrednika 380

**16. Łączenie różnych technik ... 387**
Ogólny projekt 387
Tworzenie klienta usługi Web 387
Wyświetlanie informacji 396
Przeszukiwanie na podstawie kategorii 404

**Część III CLR i platforma .NET 409**

**17. Podzespoły i kontrola wersji ... 411**
Pliki wykonywalne 411
Metadane 411
Granice zabezpieczeń 412
Manifesty 412
Podzespoły wielomodułowe 413
Podzespoły prywatne 421
Podzespoły współdzielone 421

**18. Atrybuty i mechanizm refleksji ... 427**
Atrybuty 427
Mechanizm refleksji 433

**19. Szeregowanie i zdalne korzystanie z obiektów ... 443**
Domeny aplikacji 444
Kontekst 453
Zdalne korzystanie z obiektów 455

**20. Wątki i synchronizacja ... 465**
Wątki 466
Synchronizacja 473
Sytuacja wyścigu i zakleszczenie 483

**21. Strumienie ..... 485**
Pliki i katalogi 485
Odczyt i zapis danych 495
Asynchroniczne operacje wejścia i wyjścia 502
Sieciowe operacje wejścia i wyjścia 506
Strumienie w internecie 523
Serializacja 525
Izolowana pamięć 533

**22. Platforma .NET a model COM ..... 537**
Importowanie kontrolek ActiveX 537
Importowanie komponentów COM 544
Eksportowanie komponentów .NET 552
P/Invoke 555
Wskaźniki 557

**Dodatki 563**

**A Słowa kluczowe języka C# ..... 565**

**Skorowidz ..... 571**

ROZDZIAŁ 2.

# Pierwszy program — „Witaj świecie”

Zgodnie z niepisaną tradycją wiele książek programistycznych rozpoczyna się od opisu programu wyświetlającego napis „Witaj świecie”. W tym rozdziale opisuję, jak stworzyć, skompilować i uruchomić prosty program napisany w języku C#. Analiza tego programu pozwoli zaprezentować kluczowe cechy języka C#.

Kod na listingu 2.1 zawiera podstawowe elementy bardzo prostego programu w języku C#.

*Listing 2.1. Prosty program w języku C#*

```
class Hello
{
   static void Main()
   {
      // Używa konsoli systemowej
      System.Console.WriteLine("Witaj świecie");
   }
}
```

Po skompilowaniu i uruchomieniu programu w konsoli zostanie wyświetlony napis „Witaj świecie”. Jednak przed kompilacją i uruchomieniem warto przyjrzeć się bliżej temu prostemu programowi.

## Klasy, obiekty i typy

Istotą programowania obiektowego jest tworzenie nowych typów. *Typ* to reprezentacja jakiegoś bytu. Czasem jest to byt abstrakcyjny, jak tabela danych lub wątek, czasem zaś coś bardziej widocznego, jak przycisk w oknie. Typ definiuje ogólne właściwości i zachowanie bytu.

Jeśli w programie znajdują się trzy egzemplarze klasy przycisku, na przykład przyciski *OK*, *Anuluj* i *Pomoc*, wszystkie one mają rozmiar, ale konkretny rozmiar każdego przycisku może być inny. Podobnie wszystkie przyciski obsługują te same operacje (wyświetlenie, kliknięcie), ale sposób wykonywania tych operacji może być odmienny. Dlatego poszczególne przyciski mogą się różnić szczegółami, ale wszystkie są tego samego typu.

Podobnie jak w wielu innych obiektowych językach programowania, w C# typ definiowany jest przez *klasę*, a konkretne egzemplarze klasy są znane jako *obiekty*. W kolejnych rozdziałach przedstawiam również inne typy języka C#, między innymi wyliczenia, struktury i delegaty, ale na razie proponuję skoncentrować się na klasach.

W programie wyświetlającym napis „Witaj świecie" znajduje się deklaracja tylko jednego typu — klasa `Hello`. Aby zdefiniować typ w języku C#, wystarczy zadeklarować go jako klasę za pomocą słowa kluczowego `class`, nadać mu nazwę — w tym przypadku `Hello` — i zdefiniować potrzebne właściwości i zachowanie. Właściwości i zachowanie klasy w języku C# muszą znajdować się wewnątrz nawiasów (`{}`).



*Dla programistów języka C++.* Po zamykającym nawiasie klasy nie ma średnika.

## Metody

Do cech klasy należą właściwości i zachowanie. Zachowanie zdefiniowane jest za pomocą metod składowych. Właściwości opisuję w rozdziale 3.

*Metoda* to *funkcja* należąca do klasy. Metody składowe są nawet czasem nazywane *funkcjami składowymi*. Metody definiują, co klasa może zrobić i jak się zachowuje. Zwykle nazwy metod odzwierciedlają ich działanie, na przykład `WriteLine()` lub `AddNumbers()`. Jednak w tym przypadku metoda klasy ma specjalną nazwę, `Main()`, która nie opisuje działania, ale informuje środowisko CLR, że jest to główna metoda klasy.



*Dla programistów języka C++.* Nazwa `Main()` w języku C# jest pisana wielką literą i musi być składową klasy (nie może być globalna). Metoda `Main()` może zwracać typ `int` lub `void`.

Środowisko CLR wywołuje metodę `Main()` w momencie uruchomienia programu. `Main()` to punkt startowy programu i każdy program w języku C# musi zawierać tę metodę[1].

Deklaracje metod to kontrakt między twórcą metody i jej użytkownikiem. Często autor metody i jej użytkownik to ta sama osoba, ale nie musi tak być. Zdarza się, że jeden z programistów w zespole tworzy metodę, a inny jej używa.



*Dla programistów języka Java.* `Main()` jest punktem startowym wszystkich programów języka C#, co w pewien sposób przypomina metodę `run()` w apletach i metodę `main()` w programach w języku Java.

Aby zadeklarować metodę, należy określić zwracany przez nią typ oraz podać jej nazwę. Deklaracje metod wymagają także nawiasów, niezależnie od tego, czy metoda przyjmuje parametry. Na przykład wyrażenie:

```
int myMethod(int size)
```

to deklaracja metody o nazwie `myMethod`, która przyjmuje jeden parametr — liczbę całkowitą, która w metodzie jest dostępna jako `size`. Metoda zwraca liczbę całkowitą. Typ zwracanej wartości informuje użytkownika, jaki typ danych zwraca metoda po zakończeniu działania.

[1] Teoretycznie możliwe jest umieszczenie w programie w języku C# wielu metod `Main()`. W takim przypadku należy użyć opcji `/main` wiersza poleceń i wybrać, która klasa zawierająca metodę `Main()` powinna służyć jako punkt startowy.

Niektóre metody nie zwracają żadnej wartości. Mówi się, że zwracają typ `void`, określany przez specjalne słowo kluczowe. Na przykład:

```
void myVoidMethod();
```

to deklaracja metody zwracającej typ `void` i nieprzyjmującej parametrów. W języku C# zawsze trzeba pamiętać o określaniu zwracanego typu, nawet jeśli jest to `void`.

## Komentarze

W programach C# mogą się znajdować komentarze. Na listingu 2.1 w pierwszym wierszu po początkowym nawiasie metody `Main()` znajduje się komentarz:

```
// Używa konsoli systemowej
```

Tekst rozpoczyna się od dwóch ukośników (`//`). Oznaczają one *komentarz*. Komentarz to notatka wpisana przez programistę, która nie wpływa na działanie programu. W języku C# można używać trzech rodzajów komentarzy.

Pierwszy rodzaj, przedstawiony powyżej, oznacza, że cały tekst od symboli (`//`) do końca wiersza jest traktowany jako komentarz. Jest to *komentarz w stylu języka C++*.

Drugi rodzaj komentarzy, nazywany *komentarzem w stylu języka C*, rozpoczyna się od symboli (`/*`) i kończy parą znaków (`*/`). Pozwala to na dodania komentarza ciągnącego się przez kilka wierszy bez konieczności wpisywania w każdym wierszu symboli (`//`). Przykład zastosowania tego rodzaju komentarzy znajduje się na listingu 2.2.

*Listing 2.2. Przykład zastosowania komentarzy w stylu języka C*

```
namespace HalloWorld
{
   class HelloWorld
   {
      static void Main()
      {
         /* Używa konsoli systemowej,
               jak jest to wyjaśnione w tekście */
         System.Console.WriteLine("Witaj świecie");
      }
   }
}
```

Choć zagnieżdżanie komentarzy języka C++ nie jest możliwe, można zagnieżdżać komentarze w stylu języka C++ w komentarzach w stylu języka C. Dlatego zwykle używa się komentarzy w stylu języka C++, a komentarze w stylu języka C służą głównie do „wykomentowania" bloków kodu.

Trzeci i ostatni rodzaj komentarzy obsługiwanych w języku C# służy do powiązania z kodem zewnętrznej dokumentacji opartej na XML, co opisuję w rozdziale 13.

## Aplikacje konsolowe

„Witaj świecie" to przykład programu *konsolowego*. Aplikacja konsolowa zwykle nie ma graficznego interfejsu użytkownika (ang. *graphical user interface — GUI*). Nie ma tam list rozwijanych, przycisków ani okien. Dane wejściowe i wyjściowe przekazywane są przez standardową konsolę — zwykle jest to wiersz poleceń systemu. Ograniczenie się na tym etapie książki

do aplikacji konsolowych pozwala uprościć przykłady i ułatwia przedstawienie podstawowych właściwości języka. W kolejnych rozdziałach częściej występują aplikacje okienkowe i internetowe, przy okazji których przedstawiam narzędzia środowiska Visual Studio .NET służące do projektowania GUI.

Metoda `Main()` w prostym przykładzie przedstawionym na listingu 2.1 przekazuje tekst „Witaj świecie" do *standardowego urządzenia wyjścia*, którym najczęściej jest okno konsoli. Standardowym urządzeniem wyjścia zarządza obiekt zwany `Console`. Jedna z metod tego obiektu nazywa się `WriteLine()`. Przyjmuje ona jako parametr łańcuch znaków, który następnie wyświetla w standardowym urządzeniu wyjścia. Kiedy program zostanie uruchomiony, na ekranie monitora pojawi się okno konsoli wiersza poleceń lub okno DOS-a z napisem „Witaj świecie".

Metody wywoływane są za pomocą operatora kropki (`.`). Dlatego aby wywołać metodę `WriteLine()` obiektu `Console`, należy napisać `Console.WriteLine(…)`, podając w miejscu trzech kropek tekst, który ma zostać wyświetlony.

## Przestrzenie nazw

`Console` to tylko jeden z wielu użytecznych typów wchodzących w skład biblioteki klas platformy .NET. Każda klasa nazywa się inaczej, dlatego FCL zawiera tysiące nazw, jak `ArrayList`, `Hashtable`, `FileDialog`, `DataException` czy `EventArgs`. Istnieją setki, tysiące, a może nawet dziesiątki tysięcy nazw.

Powoduje to pewne problemy. Żaden programista nie jest w stanie zapamiętać wszystkich nazw używanych przez platformę .NET, a wcześniej czy później może zdarzyć się, że programista stworzy obiekt i użyje dla niego nazwy, która jest już wykorzystywana przez inną jednostkę. Co się stanie, kiedy programista kupi klasę `Hashtable` i odkryje, że powoduje ona konflikt nazw z klasą `Hashtable` udostępnianą przez .NET? Należy pamiętać, że każda klasa w języku C# musi mieć niepowtarzalną nazwę, a zwykle nie jest możliwa zmiana nazw w zakupionym kodzie.

Rozwiązaniem tego problemu jest używanie *przestrzeni nazw*. Ograniczają one zasięg nazwy, w wyniku czego nazwa ma znaczenie tylko wewnątrz zdefiniowanej przestrzeni nazw.



*Dla programistów języka C++.* Przestrzenie nazw w języku C++ są ograniczane operatorem zakresu (`::`). W języku C# służy do tego operator kropki (`.`).

*Wskazówka dla programistów języka Java.* Przestrzenie nazw mają wiele podobnych zalet co pakiety.

Jacek jest inżynierem. Słowo „inżynier" może być związane z wieloma zawodami i wprowadzać niepewność. Czy Jacek projektuje budynki? A może opracowuje architekturę systemu operacyjnego?

Można dookreślić to stwierdzenie mówiąc, że Jacek skończył Wyższą Szkołę Budownictwa lub pracuje przy komputerach. Programista języka C# mógłby powiedzieć, że Jacek to raczej `system.inzynier` niż `budynki.inzynier`. Przestrzeń nazw, w tym przypadku `system` lub `budynki`, ogranicza zasięg słowa występującego po operatorze kropki. Pozwala to utworzyć „przestrzeń", w której nazwa ma określone znaczenie.

Może się też okazać, że Jacek nie jest dowolnym `system.inzynier`. Możliwe, że specjalizuje się w tworzeniu architektury systemów Linux. Wtedy obiekt reprezentujący Jacka można

opisać bardziej szczegółowo jako `system.linux.inzynier`. Taka klasyfikacja sugeruje, że przestrzeń nazw `linux` ma znaczenie w obrębie przestrzeni nazw `system`, a `inzynier` w tym kontekście ma znaczenie w przestrzeni `linux`. Jeśli w przyszłości okaże się, że Krzysztof jest `budynki.gospodarcze.inzynier`, łatwo będzie się zorientować, jakim jest inżynierem i czym się zajmuje. Dwa wystąpienia nazwy `inzynier` mogą współistnieć dzięki temu, że każda nazwa znajduje się we własnej przestrzeni.

Podobnie, jeśli okaże się, że klasa `Hashtable` platformy .NET znajduje się w przestrzeni nazw `System.Collections`, a programista utworzył własną klasę `Hashtable` w przestrzeni `ProgCSharp.DataStructures`, nie spowoduje to konfliktu, ponieważ nazwa klasy występuje w dwóch różnych przestrzeniach nazw.

Na listingu 2.1 fakt przynależności klasy `Console` do przestrzeni nazw `System` jest wyrażony za pomocą poniższego wiersza:

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

## Operator kropki (.)

Na listingu 2.1 operator kropki (`.`) pozwala uzyskać dostęp do metody i danych klasy (w tym przypadku jest to metoda `WriteLine()`), a także do określenia przestrzeni nazw (w tym przypadku do zaznaczenia, że klasa `Console` znajduje się w przestrzeni nazw `System`). Ten mechanizm działa poprawnie, ponieważ w obu przypadkach kompilator przeszukuje podaną przestrzeń, aby znaleźć konkretną nazwę. Przeszukiwanie rozpoczyna się od przestrzeni nazw `System`, która zawiera wszystkie obiekty systemowe udostępniane przez FCL. Typ `Console` znajduje się właśnie w przestrzeni nazw `System`, a metoda `WriteLine()` jest funkcją składową klasy `Console`.

W wielu przypadkach przestrzenie nazw są podzielone na podprzestrzenie. Na przykład przestrzeń nazw `System` zawiera wiele podprzestrzeni, jak `Data`, `Configuration` czy `Collections`. Z kolei sama przestrzeń nazw `Collections` zawiera wiele kolejnych podprzestrzeni.

Przestrzenie nazw pomagają w organizacji i porównywaniu typów. Kiedy programista pisze złożoną aplikacją w języku C#, może chcieć stworzyć własną hierarchię przestrzeni nazw. Hierarchia ta może być dowolnie głęboka. Używanie przestrzeni nazw pomaga w podziale hierarchii obiektów i przezwyciężeniu jej złożoności.

## Słowo kluczowe using

Zamiast pisać słowo `System` przed nazwą klasy `Console`, można określić, że w kodzie używane będą nazwy z przestrzeni nazw `System`. Służy do tego dyrektywa `using`:

```
using System;
```

umieszczana na początku kodu, jak na listingu 2.3.

*Listing 2.3. Słowo kluczowe using*

```
using System;
class Hello
{
   static void Main()
   {
```

```
        // Klasa Console z przestrzeni nazw System
        Console.WriteLine("Witaj świecie");
    }
}
```

Warto zwrócić uwagę, że dyrektywa `using` znajduje się przed definicją klasy `Hello`. Domyślnie w środowisku Visual Studio .NET do każdej aplikacji konsolowej dodawane są trzy dyrektywy `using` — `System`, `System.Collections.Generics` oraz `System.Text`.

Choć można określić, że w programie używane są obiekty z przestrzeni nazw `System`, w odróżnieniu od niektórych języków nie można zrobić tego samego w przypadku obiektu `System.Console`. Kod przedstawiony na listingu 2.4 nie skompiluje się.

*Listing 2.4. Niepoprawny kod w języku C#*

```
using System.Console;
class Hello
{
   static void Main()
   {
      // Konsola z przestrzeni nazw System
      WriteLine("Witaj świecie");
   }
}
```

Próba kompilacji takiego kodu spowoduje wygenerowanie błędu:

```
error CS0138: A using namespace directive can only be applied
to namespaces;  System.Console  is a type not a namespace
```



Użytkownicy Visual Studio będą wiedzieć, kiedy popełnili błąd, ponieważ po wpisaniu `using System` i kropki środowisko wyświetla listę poprawnych przestrzeni nazw, a `Console` nie jest jedną z nich.

Słowo kluczowe `using` pozwala oszczędzić wpisywania dużych ilości kodu, jednak zmniejsza korzyści używania przestrzeni nazw, zaśmiecając zasięg wieloma nieodróżnialnymi nazwami. Często stosowanym rozwiązaniem jest używanie słowa kluczowego `using`, kiedy korzysta się z wbudowanych i własnych przestrzeni nazw, a unikanie tego w przypadku komponentów dostarczanych przez innych producentów.



Niektóre grupy programistyczne stosują politykę wpisywania długich nazw obiektów, wraz z przestrzeniami nazw (na przykład `System.Console.WriteLine()` zamiast `Console.WriteLine()`). Ma to służyć jako forma dokumentowania kodu. Jednak w przypadku wielokrotnie zagnieżdżonych przestrzeni nazw takie rozwiązanie może szybko stać się niewygodne.

## Wrażliwość na wielkość znaków

W języku C# wielkość znaków ma znaczenie. Oznacza to, że `writeLine` to coś innego niż `WriteLine`, a jeszcze czymś innym jest `WRITELINE`. Niestety, w przeciwieństwie do VB, w środowisku języka C# błędna wielkość liter nie jest automatycznie poprawiana. Błąd spowodowany przez napisanie tego samego słowa za pomocą liter o odmiennej wielkości może być trudny do wykrycia.



Użyteczną sztuczką jest umieszczenie kursora nad nazwą, w której jedynym błędem jest niepoprawna wielkość znaków, a następnie użycie kombinacji klawiszy *Ctrl+Space*. Mechanizm uzupełniania kodu Intellisense zmieni wtedy nazwę na poprawną.

Aby zapobiec takim czasochłonnym i męczącym błędom, warto ustalić konwencję nazywania zmiennych, funkcji, stałych i innych elementów kodu. W tej książce do nazw zmiennych stosuję notację wielbłądzią (na przykład `jakasNazwaZmiennej`), a do nazw funkcji, stałych i właściwości używam notacji z języka Pascal (na przykład `JakasFunkcja`).



Jedyna różnica między notacją wielbłądzią a notacją z języka Pascal polega na rozpoczynaniu nazw wielką literą w tym drugim przypadku.

Microsoft udostępnia wskazówki dotyczące stylu kodowania, które są bardzo dobrym punktem wyjścia, a często w zupełności wystarczają do tworzenia czytelnego kodu. Można je znaleźć na stronie *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconNETFrameworkDesingGuidlines.asp*.

## Słowo kluczowe static

Przed metodą `Main()` przedstawioną na listingu 2.1 znajduje się modyfikator. Zaraz przed zwracanym typem `void` (co oznacza, że metoda nie zwraca żadnej wartości) znajduje się słowo kluczowe `static`:

```
static void Main()
```

Słowo kluczowe `static` informuje kompilator, że można wywołać metodę `Main()` bez konieczności tworzenia obiektu typu `Hello`. To dość złożone zagadnienie opisuję bardziej szczegółowo w kolejnych rozdziałach. Jednym z problemów uczenia się nowego języka programowania jest konieczność poznawania niektórych skomplikowanych mechanizmów, zanim możliwe jest ich pełne zrozumienie. Na razie można więc traktować deklarację metody `Main()` jako czystą magię.

# Tworzenie programu „Witaj świecie"

Istnieją przynajmniej dwa sposoby pisania, kompilowania i uruchamiania programów przedstawionych w tej książce. Można użyć zintegrowanego środowiska programowania (ang. *Integrated Development Environment — IDE*) Visual Studio .NET lub edytora tekstu i kompilatora uruchamianego z wiersza poleceń (wraz z pewnymi dodatkowymi narzędziami opisanymi w kolejnych rozdziałach).

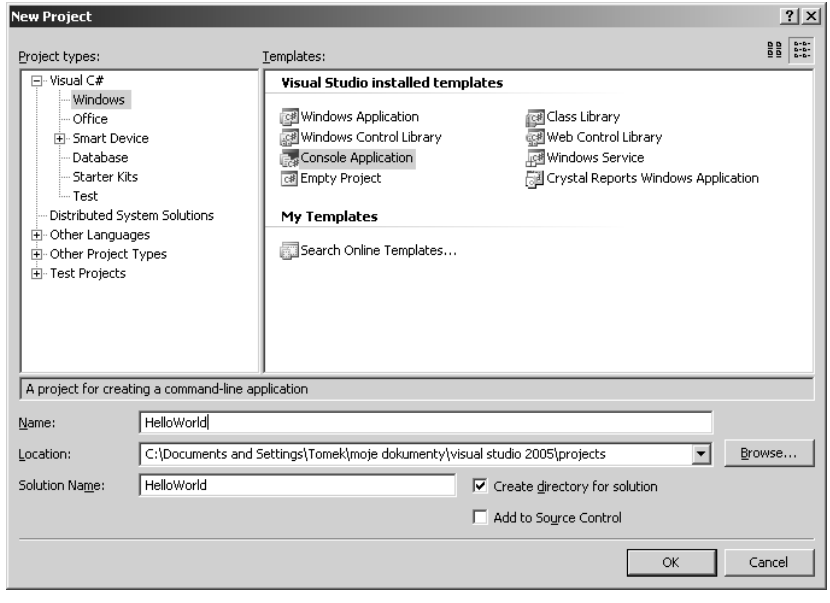
Choć *możliwe* jest tworzenie programów bez korzystania z Visual Studio .NET, używanie dobrego IDE związane jest z wieloma udogodnieniami. Są to między innymi automatyczne wcinanie kodu, mechanizm uzupełniania słów Intellisense, kolorowanie składni czy integracja z systemem pomocy. Co najważniejsze, IDE zawiera wydajne mechanizmy do usuwania błędów i mnóstwo innych narzędzi.

Zakładam po cichu, że Czytelnicy tej książki używają Visual Studio .NET. Jednak opis przykładowych programów kładzie większy nacisk na sam język i platformę niż na używane narzędzia. Można przepisać wszystkie przykłady korzystając z Notatnika lub Emacs, zapisać je

z rozszerzeniem *.cs* i skompilować za pomocą kompilatora uruchamianego z wiersza poleceń, który udostępniany jest wraz z .NET Framework SDK. Można także użyć jednego ze zgodnych z .NET narzędzi, jak Mono lub Microsoft Shared Source CLI. Warto zauważyć, że niektóre przykłady w późniejszych rozdziałach wymagają użycia narzędzi Visual Studio .NET do tworzenia formularzy Windows i Web, jednak zdeterminowani programiści mogą nawet te przykłady wpisać samodzielnie, korzystając z Notatnika.

## Pisanie kodu „Witaj świecie"

Aby utworzyć program „Witaj świecie" w środowisku IDE, należy wybrać Visual Studio .NET z menu *Start* lub kliknąć ikonę na pulpicie, a następnie wybrać z menu paska narzędzi *File→New→Project*. Spowoduje to otwarcie okna *New Project*. Przy pierwszym użyciu Visual Studio okno *New Project* może pojawić się bez dalszych zapytań. Okno *New Project* przedstawione jest na rysunku 2.1.

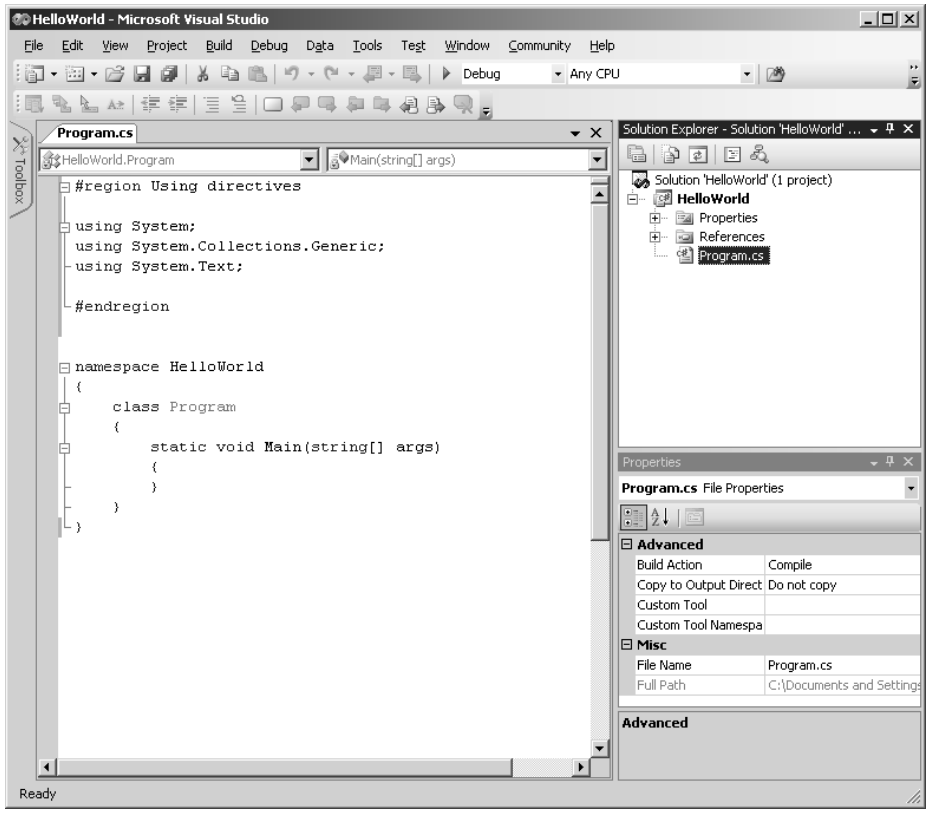



*Rysunek 2.1. Tworzenie aplikacji konsolowej w środowisku Visual Studio .NET*

Aby utworzyć aplikację, należy zaznaczyć *Visual C#* w oknie *Project Types*, a następnie wybrać opcję *Console Application* w oknie *Templates*. W środowisku Express Edition języka Visual C# wystarczy wybrać opcję Console Application.

Następnie można podać nazwę projektu i wybrać katalog, w którym przechowywane będą pliki. Potem wystarczy kliknąć przycisk *OK* i pojawi się nowe okno, w którym można wpisać kod z listingu 2.1, co pokazane jest na rysunku 2.2.

Warto zwrócić uwagę, że Visual Studio .NET automatycznie tworzy przestrzeń nazw według nazwy projektu (w tym przypadku jest to `HelloWorld`), a także dodaje dyrektywy `using`




*Rysunek 2.2. Edytor przygotowany do pracy z nowym projektem*

przestrzeni nazw `System`, `System.Collections.Generic` i `System.Text`, ponieważ potrzebuje ich prawie każdy program.

Visual Studio .NET tworzy klasę o nazwie `Program`, którą programista może nazwać w inny sposób. Przy zmianie nazwy klasy warto zmienić również nazwę pliku. Przy zmianie nazwy pliku środowisko Visual Studio automatycznie zmienia nazwę klasy. Aby odtworzyć kod z listingu 2.1, można na przykład zmienić nazwę pliku *Program.cs* (znajduje się ona w oknie *Solution Explorer*) na *hello.cs* a nazwę klasy *Program* na *HelloWorld*. Jeśli wykona się te czynności w odwrotnej kolejności, Visual Studio automatycznie zmieni nazwę klasy na *Hello*.

Visual Studio tworzy szkielet programu. Aby napisać program z listingu 2.1, należy usunąć argumenty `(string[] args)` metody `Main()`, a następnie wpisać w ciele tej metody dwa poniższe wiersze kodu:

```
// Używa konsoli systemowej
System.Console.WriteLine("Witaj świecie");
```

Można też napisać kod, nie używając środowiska Visual Studio. W tym celu należy otworzyć *Notatnik* lub inny edytor tekstu, wpisać kod z listingu 2.1, zapisać plik jako tekstowy i nadać mu nazwę *hello.cs*.

## Kompilacja i uruchamianie „Witaj świecie"

W Visual Studio istnieje wiele sposobów na skompilowanie i uruchomienie programu „Witaj świecie". Zwykle robi się to za pomocą poleceń z menu na pasku narzędzi, przycisków lub, w wielu przypadkach, używając klawiszy skrótu.



Klawisze skrótu można ustawić wybierając opcję *Tools→Options→Keyboard*. W książce przedstawiam najpopularniejsze domyślne klawisze skrótu.

Na przykład, aby skompilować program „Witaj świecie", należy przycisnąć kombinację klawiszy *Ctrl+Shift+B* lub wybrać opcję *Build→Build Solution*. Jeszcze inne rozwiązanie to kliknięcie przycisku *Build* na pasku narzędzi *Build*. Jeśli pasek ten nie jest widoczny, należy kliknąć prawym przyciskiem myszy na pasku narzędzi i uaktywnić go. Pasek narzędzi *Build* przedstawiony jest na rysunku 2.3. Przycisk *Build* znajduje się po lewej stronie i jest zaznaczony na rysunku.




*Rysunek 2.3. Pasek narzędzi Build*

Aby uruchomić program „Witaj świecie" w trybie bez usuwania błędów, należy przycisnąć kombinację klawiszy *Ctrl+F5*, wybrać opcję *Debug→Start Without Debugging* z menu *IDE* z paska narzędzi lub kliknąć przycisk *Start Without Debugging* na pasku narzędzi *IDE Build*. Ta ostatnia możliwość przedstawiona jest na rysunku 2.4. Jeśli przycisk ten nie jest widoczny, należy zmienić ustawienia paska narzędzi. Możliwe jest uruchomienie programu bez wcześniejszej jego budowy. Ustawienia opcji (można je zmienić, wybierając *Tools→Options*) mogą spowodować, że IDE zapisze plik, zbuduje program, a następnie uruchomi go, zapewne pytając o pozwolenie wykonania każdej kolejnej czynności.



*Rysunek 2.4. Przycisk Start Without Debugging*



Gorąco zalecam zapoznanie się ze środowiskiem Visual Studio 2005. Jest to podstawowe narzędzie programisty tworzącego aplikację na platformę .NET i warto dobrze poznać jego działanie. Czas poświęcony na zapoznanie się ze środowiskiem Visual Studio zwróci się wielokrotnie w trakcie kolejnych miesięcy programowania. Proponuję odłożyć na chwilę książkę i przyjrzeć się dobrze środowisku.

Aby skompilować i uruchomić program „Witaj świecie" używając kompilatora C# uruchamianego z wiersza poleceń, udostępnianego wraz z .NET Framework SDK, z Mono (*http://www.mono-project.com*) lub z Shared Source CLI (*http://msdn.microsoft.com/net/sscli/*), należy wykonać następujące czynności:

1. Zapisać kod z listingu 2.1 w pliku *hello.cs*.
2. Otworzyć okno poleceń platformy .NET (*Start→Programs→Visual Studio .NET→Visual Studio Tools→Visual Studio Command Prompt*). Użytkownicy systemu Unix powinni uruchomić konsolę, xterm lub inne narzędzie umożliwiające wpisywanie poleceń powłoki.
3. W wierszu poleceń należy wpisać:

   ```
   csc /debug hello.cs
   ```

   jeśli używa się kompilatora z .NET lub z Shared Source CLI, albo:

   ```
   mcs -debug hello.cs
   ```

   jeśli używa się kompilatora z Mono. Powoduje to utworzenie pliku *.EXE*. Jeśli program zawiera błędy, kompilator wyświetli je. Opcja wiersza poleceń `/debug` powoduje umieszczenie w kodzie symboli, które umożliwiają uruchomienie pliku o rozszerzeniu *EXE* w programie uruchomieniowym lub zobaczenie numerów wierszy kodu w czasie podglądania stosu. Obraz stosu zostanie wyświetlony, jeśli program wygeneruje nieobsługiwany błąd.
4. Aby uruchomić program, należy wpisać:

   ```
   hello
   ```

   jeśli używa się kompilatora z .NET, lub:

   ```
   clix hello.exe
   ```

   jeśli używa się kompilatora z Shared Source CLI, a:

   ```
   mono hello.exe
   ```

   w przypadku korzystania z kompilatora z Mono.

W oknie powinien pojawić się wtedy napis „Witaj świecie".

## Kompilacja JIT

W trakcie kompilacji *hello.cs* za pomocą polecenia `csc` powstaje plik wykonywalny. Należy jednak pamiętać, że zawiera on kod w języku MSIL, opisanym w rozdziale 1.

Co ciekawe, kiedy napisze się aplikację w języku VB.NET lub w dowolnym języku zgodnym systemem .NET CLS, zostanie ona skompilowana do bardzo podobnego kodu w języku MSIL. Kod IL utworzony na podstawie kodu pisanego w różnych językach jest praktycznie nierozróżnialny.

Oprócz tworzenia kodu IL, który jest podobny do kodu bajtowego języka Java, kompilator tworzy w pliku o rozszerzeniu *.EXE* segment tylko do odczytu, w którym znajduje się standardowy nagłówek plików wykonywalnych systemu Win32. W tym segmencie kompilator określa punkt startowy, do którego w momencie uruchamiania programu przechodzi program ładujący systemu operacyjnego.

System operacyjny nie potrafi jednak wykonać kodu IL, dlatego punkt startowy prowadzi do uruchomienia kompilatora JIT platformy .NET (kompilator ten został opisany w rozdziale 1.). W wyniku kompilacji JIT powstaje kod maszynowy procesora, taki sam, jak w zwykłych plikach wykonywalnych. Kluczową cechą kompilacji JIT jest to, że funkcje kompilowane są jedynie wtedy, kiedy są używane, bezpośrednio przed ich wykonaniem.

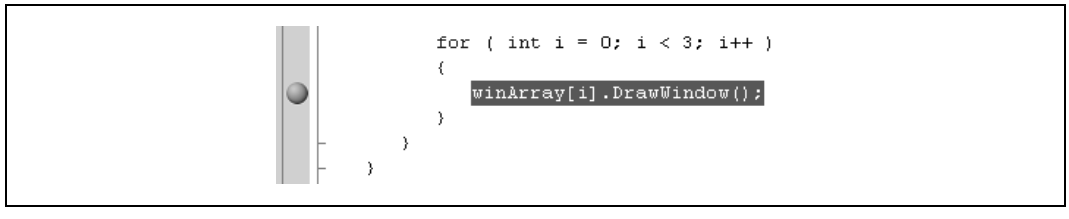
# Usuwanie błędów w Visual Studio .NET

Najważniejszym elementem każdego środowiska programistycznego jest program uruchomieniowy. Narzędzie udostępniane w Visual Studio jest bardzo wydajne i czas poświęcony na jego dobre poznanie na pewno nie będzie stracony. Podstawy używania programu uruchomieniowego są bardzo proste. Wystarczy nauczyć się wykonywać trzy kluczowe czynności:

- ustawianie punktów przerwania i wykonywanie programów do danego punktu,
- wkraczanie w ciało metody i przeskakiwanie nad wywołaniami metod,
- sprawdzanie i zmiana wartości zmiennych, danych składowych i innych wartości.

Ten rozdział nie obejmuje całej dokumentacji programu uruchomieniowego, ale wymienione powyżej czynności są tak istotne, że z pewnością wystarczą do rozpoczęcia pracy.

Program uruchomieniowy umożliwia dostęp do tych samych operacji na różne sposoby, zwykle przez opcje menu i przyciski. Najłatwiejszy sposób na ustawienie punktu przerwania to kliknięcie na lewym marginesie. Środowisko IDE zaznacza punkty przerwania jako czerwone kropki, jak przedstawione to jest na rysunku 2.5.

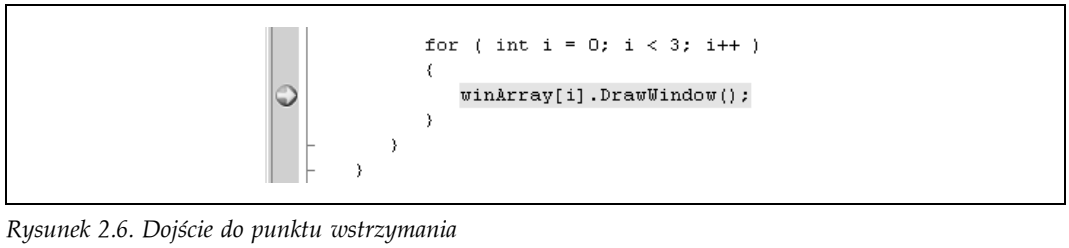



*Rysunek 2.5. Punkt przerwania*



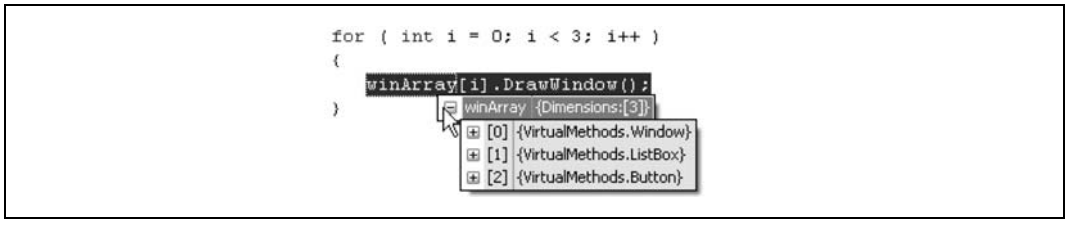
Analiza działania programu uruchomieniowego wymaga przykładowego kodu. W tym punkcie używam fragmentów kodu z rozdziału 5., który na razie może być niezrozumiały. Czytelnicy znający język C++ lub Java powinni jednak rozumieć jego znaczenie.

Aby uruchomić program uruchomieniowy, należy wybrać opcję *Debug→Start* lub przycisnąć klawisz *F5*. Program zostanie wtedy skompilowany, a następnie wykonany do punktu przerwania. W tym momencie wykonywanie zatrzyma się, a żółta strzałka wskaże instrukcję, która powinna zostać wykonana jako kolejna, co pokazane jest na rysunku 2.6.



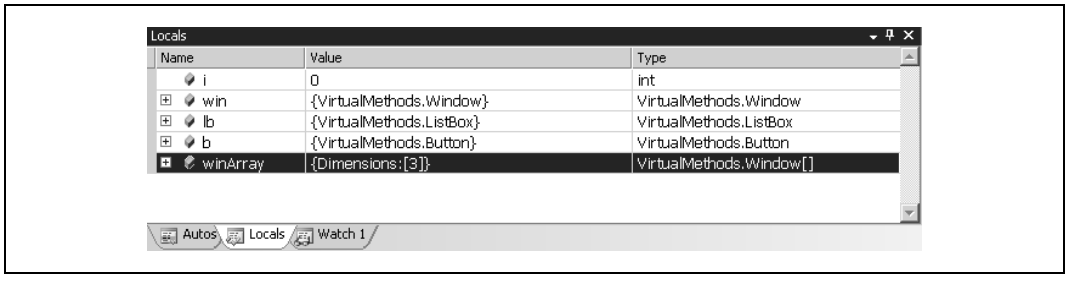

*Rysunek 2.6. Dojście do punktu wstrzymania*

Po dojściu do punktu wstrzymania łatwo jest sprawdzić wartości poszczególnych obiektów. Na przykład można sprawdzić wartość zmiennej i umieszczając nad nią kursor i odczekując chwilę. Pojawi się wtedy lista taka jak na rysunku 2.7.



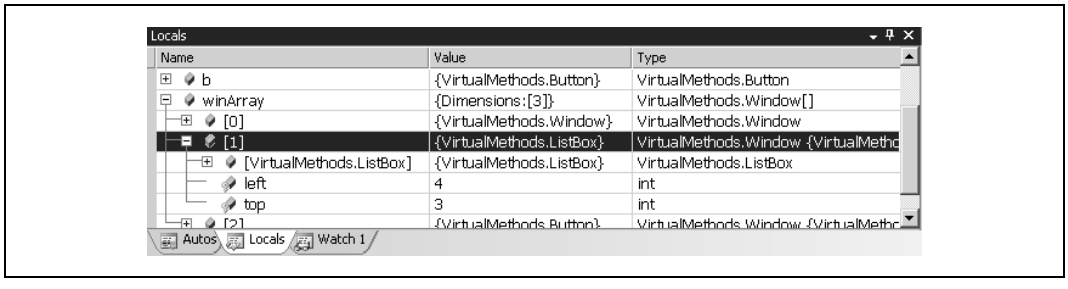

*Rysunek 2.7. Sprawdzanie wartości*

Środowisko programu uruchomieniowego udostępnia także liczne użyteczne okna, jak okno *Locals* wyświetlające wartości wszystkich zmiennych lokalnych (patrz rysunek 2.8).




*Rysunek 2.8. Okno Locals*

W przypadku zmiennych należących do typów wbudowanych, na przykład liczb całkowitych, wyświetlana jest ich wartość. Obiekty złożonych typów posiadają przy sobie znak plus (+). Można je rozwinąć, jak widać na rysunku 2.9. Więcej informacji o obiektach i ich danych znajduje się w kolejnych rozdziałach.

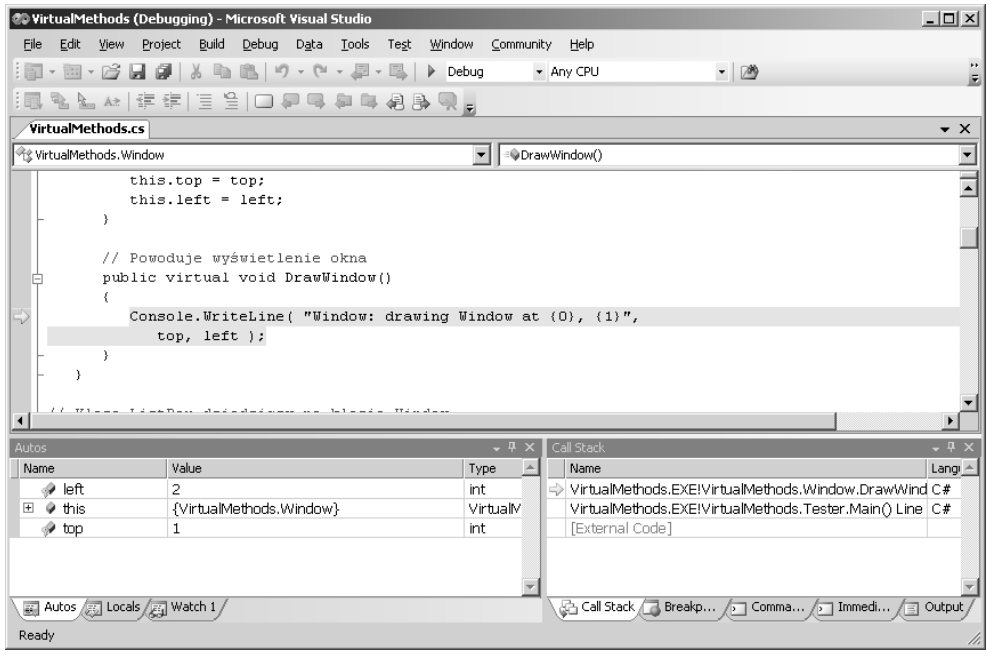



*Rysunek 2.9. Obiekt rozwinięty w oknie Locals*

Można wkroczyć w ciało kolejnej metody przyciskając klawisz *F11*. W tym przypadku spowoduje to wejście w ciało metody `DrawWindow()` klasy `Window`, co przedstawione jest na rysunku 2.10.

Można zobaczyć, że kolejną instrukcją jest teraz `WriteLine()` w metodzie `DrawWindow()`. Okno *Autos* zostało uaktualnione i przedstawia aktualny stan obiektów.

Oczywiście to nie wszystkie możliwości programu uruchomieniowego, jednak materiał przedstawiony w tym rozdziale pozwala na rozpoczęcie pracy z tym narzędziem. Pisanie krótkich programów demonstracyjnych i sprawdzanie ich działania za pomocą programu uruchomieniowego pozwala udzielić odpowiedzi na wiele pytań programistycznych. Dobry program uruchomieniowy jest, w pewnym sensie, najbardziej istotnym narzędziem języka programowania.




*Rysunek 2.10. Wkraczanie w ciało metody*