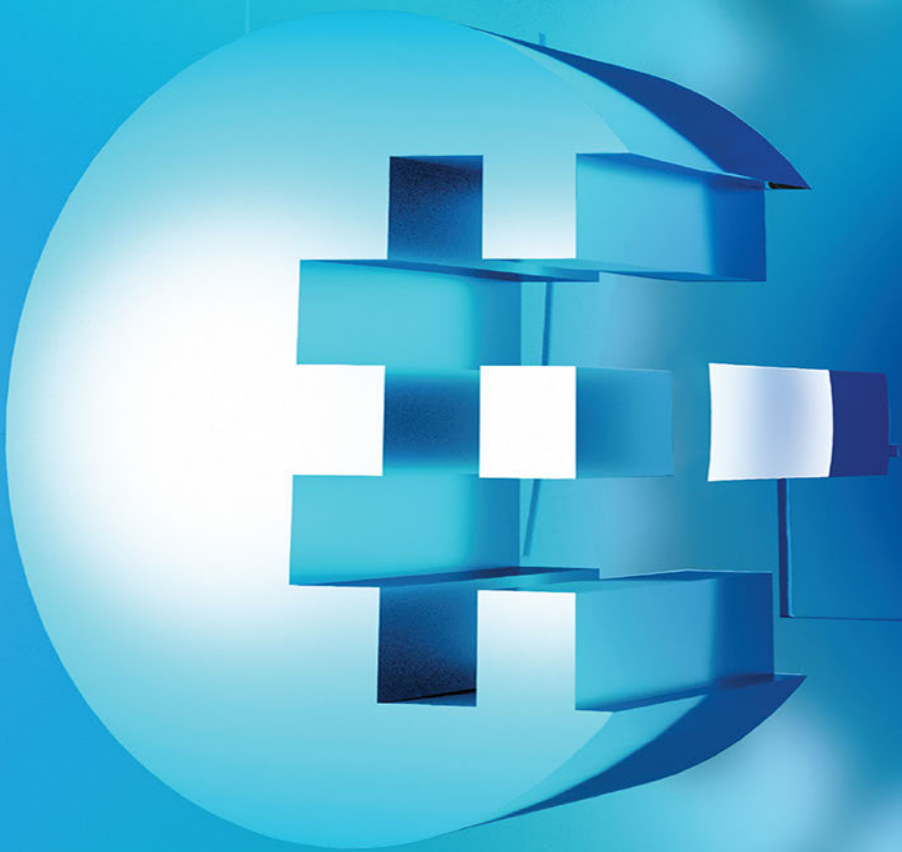
Helion

Jacek Matulewski



C#
Lekcje programowania

Praktyczna nauka programowania dla platform .NET i .NET Core

Redaktor prowadzący: Szymon Szwajger
Projekt okładki: Tomasz Wlaźlak

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/c6ntel
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-1102-2

Printed in Poland.

Spis treści

Dodatki 499

Wstęp

Celem niniejszej książki jest przede wszystkim nauka programowania. Wybór języka programowania, choć ważny, nie jest aż tak kluczowy, jak może się z początku wydawać. Wybrałem C# ze względu na jego nowoczesność, stosunkowo dużą popularność, ciągły rozwój, a także podobieństwo do innych ważnych obecnie języków z rodziny C, C++ i Java. Nie bez znaczenia jest także fakt, że ja osobiście bardzo ten język lubię. Należy jednak zaznaczyć, że wybór C# jako pierwszego języka nie jest standardowym posunięciem. C# jest bardzo wdzięcznym obiektem nauki, lecz zwykle nauka programowania zaczyna się od C i potem C++. Jest tak między innymi dlatego, że w C i C++ istnieją mechanizmy zarządzania pamięcią, których nie ma w C#. W tym języku, podobnie jak w siostrzanej Javie, jesteśmy zwolnieni z martwienia się o pamięć i groźbę wycieku pamięci (przynajmniej tak mówimy na pierwszych zajęciach — w rzeczywistości sprawa jest bardziej skomplikowana). Nie ma tym samym okazji, aby te mechanizmy poznać.

W pierwszych dwóch częściach książki znajduje się materiał, który jest przedmiotem wykładu i ćwiczeń „Programowanie I" dla studentów kognitywistyki Uniwersytetu Mikołaja Kopernika w Toruniu. Z kolei zawartość niektórych rozdziałów z dalszej części książki przedstawiam na kursie .NET dla studentów informatyki stosowanej.

Zakres zagadnień poruszanych w tej książce obejmuje: 1) język programowania C#, włączając posługiwanie się zmiennymi, metodami, tablicami i kolekcjami danych z platformy .NET Core, 2) podstawowe techniki programowania obiektowego, 3) różne formaty przechowywania i eksportu danych oraz 4) Visual Studio 2019 Community, czyli darmową wersję bardzo popularnego środowiska programistycznego firmy Microsoft. Ważnymi tematami, które nie zostały w tej książce szczegółowo omówione, są algorytmy i struktury danych. Te ostatnie są w C# i platformie .NET mniej istotne, skoro podstawowe struktury danych czekają gotowe do użycia. Podobnie zresztą jak podstawowe algorytmy sortowania i wyszukiwania. Wyjątkiem jest drzewo, którego nie ma w standardowych kolekcjach z platformy .NET i .NET Core i dlatego zostało szczegółowo omówione, zarówno w wersji podstawowej, jak i rozszerzonej z cyklami. Przedyskutowana została także standardowa metoda rekurencyjnego przechodzenia drzewa.

W pierwszych dwóch częściach wyraźnie oznaczone są fragmenty, które można pominąć, jeżeli ta książka jest pierwszą pozycją dotyczącą programowania, z jaką czytelnik się zapoznaje. W takiej sytuacji z pewnością konieczne będzie ponowne przeczytanie rozdziałów z pierwszej

i drugiej części, a wówczas można te fragmenty uzupełnić. Taki układ podyktowany jest chęcią przedstawienia spójnego opisu powiązanych ze sobą zagadnień, które z natury rzeczy mają różny stopień trudności. Większość rozdziałów z pierwszych dwóch części kończy się ćwiczeniami i zadaniami. W ćwiczeniach opisany jest kod, który czytelnik powinien czytać przy komputerze, przenosząc kod do Visual Studio i wykonując kolejne instrukcje. Jednak to zadania sprawią, że czytelnik stanie się programistą — nie mam wątpliwości, że dopiero samodzielne rozwiązywanie problemów powoduje, że omawiane w rozdziale zagadnienia stają się w pełni zrozumiałe.

Warto wspomnieć też o tym, że sporo materiału z pierwszych dwóch części znalazło się w dwudziestu dwóch filmach udostępnionych studentom w serwisie YouTube w ramach zajęć zdalnych wymuszonych przez epidemię COVID-19 wiosną 2020 roku. Filmy są publicznie dostępne pod adresem *https://bit.ly/jacekmatulewski_programowanie*.

Chciałbym gorąco podziękować osobom, które zechciały przetestować pierwsze wersje rozdziałów z pierwszej i drugiej części książki. Są to między innymi Patryk Mruk, Sylwia Ziółkowska i Bartosz Matulewski. Na szczególne podziękowania zasługują także Rafał Linowiecki, który zweryfikował rozdział 19. i dodatek C, oraz Tomasz Wlaźlak z Wydziału Sztuk Pięknych UMK, który zgodził się zaprojektować okładkę.

Część I.

Podstawy programowania

Rozdział 1.

Pierwszy kontakt ze środowiskiem Visual Studio 2019 i językiem C#

Zanim zaczniemy, dwie uwagi. Pierwsza dotyczy środowiska programistycznego. Uważam, że najwygodniejszym rozwiązaniem w trakcie nauki języka C# i później, do tworzenia oprogramowania, jest zainstalowanie darmowego środowiska Visual Studio Community 2019, które można pobrać ze strony *https://visualstudio.microsoft.com/pl/vs/community/*. Korzystając z tego środowiska, wykonałem wszystkie zrzuty ekranu, jakie umieściłem w tej książce. Osoby, które nie chcą go instalować albo nie chcą tego robić od razu, mają jednak alternatywę, która — choć nie jest doskonała — umożliwia przetestowanie większości aplikacji konsolowych opisanych w tej książce: można użyć środowiska on-line dostępnego na stronie *https://dotnetfiddle.net/*.

Druga uwaga dotyczy sposobu czytania książki. W większości rozdziałów — choć nie we wszystkich — wprowadziłem podział na część „teoretyczną", w której opisuję nowe zagadnienia, oraz część zatytułowaną „Ćwiczenia", którą należy wykonywać przy komputerze, przepisując kolejne listingi i próbując kompilować powstające w ten sposób projekty prostych aplikacji konsolowych. Na końcu większości rozdziałów są pytania weryfikujące najważniejsze wiadomości z rozdziału oraz zadania, które mają utrwalać zdobyte umiejętności przez ich praktyczne ćwiczenie.

Dwa początkowe rozdziały są jednak nieco inne. Pierwszy ma charakter tutorialu, który szybko wprowadza do zagadnień związanych z językiem C# i środowiskiem Visual Studio. Przygotujemy w nim pierwszą aplikację konsolową, choć mam świadomość, że przy opisywaniu kolejnych kroków będę używał nieznanych konstrukcji języka C# i wielu słów, które mogą na razie być dla czytelnika niejasne (np. „metoda", „metoda statyczna", „klasa", „własność"). Wszystko to wyjaśnię w kolejnych rozdziałach, gdzie język C# zostanie opisany bardziej systematycznie. Ten rozdział ma być trochę jak bezpieczny skok do basenu w rękawkach. Chodzi o to, żeby oswoić się ze środowiskiem i poznać podstawowe elementy, a nie żeby od razu wszystko rozumieć. Dlatego wykonując kolejne polecenia z tego rozdziału, nie należy się martwić, gdy coś będzie niezrozumiałe lub nie będzie w pełni jasne, co i po co robimy.

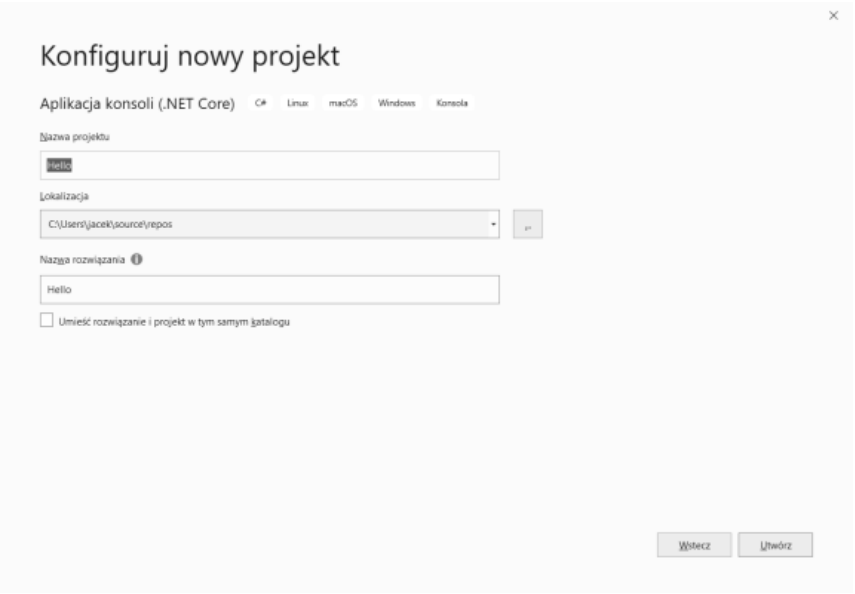
Projekt aplikacji konsolowej

Stwórzmy zatem projekt aplikacji konsolowej.

1. Po uruchomieniu Visual Studio 2019 pojawi się okno pozwalające m.in. na utworzenie nowego projektu (rysunek 1.1, lewy).

RYSUNEK 1.1. *Tworzenie aplikacji na podstawie szablonu Aplikacja konsolowa (.NET Core)*

2. Kliknijmy przycisk *Utwórz nowy projekt* z prawej strony okna. Pojawi się kolejny krok kreatora (rysunek 1.1, prawy), w którym zaznaczmy szablon *Aplikacja konsoli (.NET Core)* i kliknijmy przycisk *Dalej*.
3. W następnym oknie kreatora (rysunek 1.2) nazwijmy projekt „Hello". Pozostawmy bez zmian jego miejsce zapisu na dysku (*Lokalizacja*) i kliknijmy *Utwórz*.



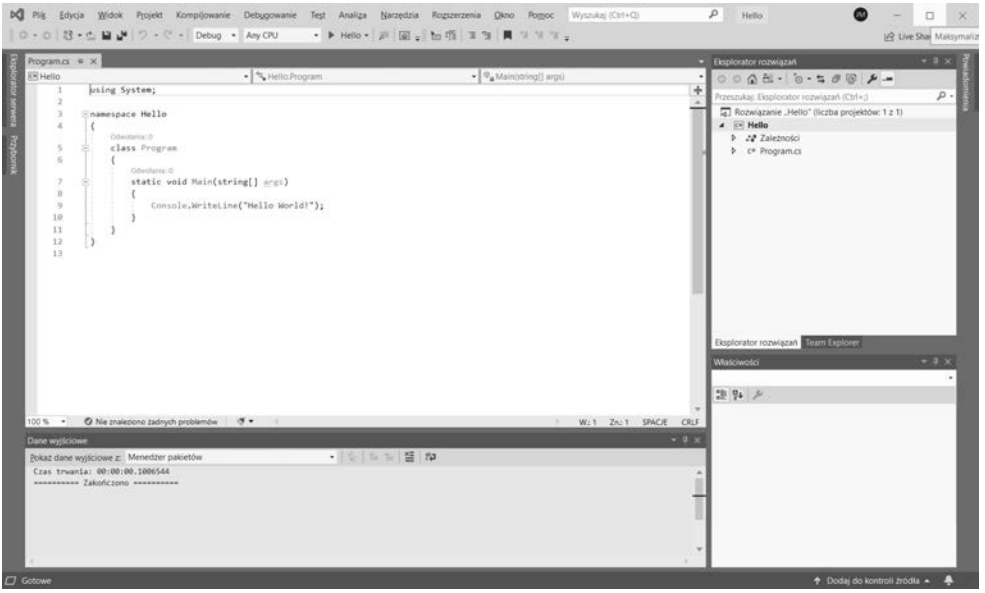
RYSUNEK 1.2. *Wybór nazwy i miejsca zapisania projektu*

4. Po kliknięciu przycisku *Utwórz* pojawi się zasadnicze okno Visual Studio (rysunek 1.3).

Stworzyliśmy projekt dla platformy .NET Core, która dostępna jest zarówno pod systemem Windows, jak i Linux i macOS. To oznacza, że aplikacja jest przenaszalna. Gdybyśmy jednak utworzyli projekt dla tradycyjnej platformy .NET, kod programu byłby taki sam (zarówno ten stworzony automatycznie, jak i ten, który napiszemy w dalszej części rozdziału), ale program mógłby być uruchamiany tylko w systemie Windows.

Pomimo że w książce opisane są projekty dla platformy .NET Core, zdobyte umiejętności mogą też zostać wykorzystane w przypadku projektów dla tradycyjnej platformy .NET.

Projekt został zapisany w katalogu *C:\Users\jacek\source\repos\Hello* (oczywiście nazwa użytkownika będzie inna), w którym znajduje się plik rozwiązania *Hello.sln*. Pozostałe pliki projektu, w tym plik z kodem źródłowym, zostały umieszczone w podkatalogu projektu *Hello*. Są to: plik projektu *Hello.csproj* oraz plik z kodem źródłowym *Program.cs*, który powinien się również pojawić w edytorze kodu Visual Studio (rysunek 1.3). W tym pliku znajduje się na razie cały kod aplikacji (listing 1.1).



RYSUNEK 1.3. *Visual Studio 2019 z otwartym projektem dla platformy .NET Core*

W jego pierwszej linii zadeklarowane jest użycie przestrzeni nazw `System`, która zawiera podstawowe typy (liczby całkowite, rzeczywiste, łańcuchy/tekst itp.) oraz klasę `Console`, służącą m.in. do wyświetlania napisów w konsoli. Do deklaracji użycia przestrzeni nazw służy polecenie złożone ze słowa kluczowego `using` i następującej po nim nazwy przestrzeni nazw. Dzięki temu użycie klas znajdujących się w obrębie zadeklarowanych przestrzeni nie będzie wymagało jawnego wymieniania przestrzeni nazw przed nazwą klasy, np. zamiast odwoływać się do pełnej nazwy klasy `System.Console`, możemy poprzestać na `Console`. Przestrzenie nazw mogą tworzyć hierarchiczną strukturę. Zadeklarowanie korzystania z przestrzeni nazw nie pociąga za sobą automatycznego dostępu do jej „podprzestrzeni" lub „nadprzestrzeni". Każdą trzeba deklarować osobno. Poleceń `using` może być na początku pliku więcej — osobne dla każdej użytej przestrzeni.

Za deklaracją używanych w pliku *Program.cs* przestrzeni nazw słowo kluczowe `namespace` otwiera naszą własną przestrzeń nazw o nazwie `Hello`. W niej znajduje się klasa `Program`, zawierająca jedną statyczną metodę o nazwie `Main`. Metoda to zbiór poleceń wykonywanych po kolei — w tej metodzie znajduje się jednak tylko jedno polecenie, które wyświetla napis o treści „Hello World!".

LISTING 1.1. *Kod utworzony przez Visual C#*

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
```

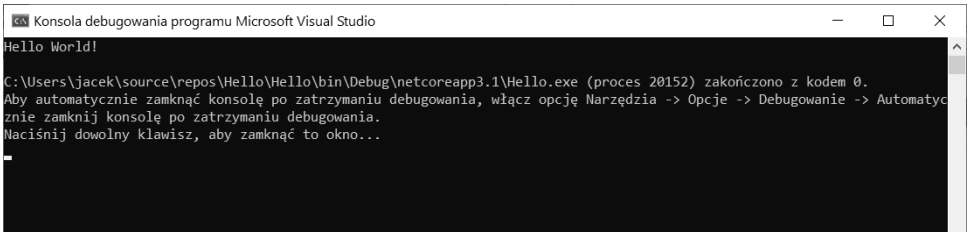
```
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

W edytorze kodu nad każdą metodą i klasą widoczna jest liczba ich użyć w pozostałej części kodu (odwołania do tych metod i klas). W tej chwili nad metodą `Main` powinno być widoczne „Odwołania: 0", ale gdy odwołań będzie więcej, można kliknąć tę informację, aby zobaczyć miejsca w kodzie, w których się one znajdują.

Zwróćmy też uwagę na małe znaki minus przy słowach kluczowych `namespace`, `class` i `static`. Pozwalają one na „zwinięcie" do jednej linii bloku poleceń objętych nawiasami klamrowymi {}. Ułatwia to edycję kodu, szczególnie wtedy, gdy już się bardzo rozrośnie.

C# to język, którego kod jest podobny do C++ i Javy. Większość słów kluczowych pochodzi wprost z języka C++. To oznacza, że osoby, które choć trochę znają któryś z tych języków, nie powinny mieć problemu z poznaniem C# oraz że po zapoznaniu się z C# przejście do Javy i do C++ też powinno być stosunkowo łatwe.

Podczas uruchamiania programu platforma .NET Core wywoła statyczną metodę `Main` z klasy `Program`. Stanowi ona **punkt wejściowy** aplikacji (ang. *entry point*). To oznacza, że gdybyśmy nacisnęli *F5* lub *Ctrl+F5*, uruchamiając aplikację, kod tej metody zostanie wykonany i w oknie konsoli pojawi się napis „Hello World!" (rysunek 1.4). Warto spróbować. Oprócz właściwego napisu w konsoli pojawi się dodatkowa informacja od Visual Studio, która nie jest efektem uruchomienia samego programu i która nie pojawiłaby się, gdybyśmy program uruchamiali w normalny sposób, np. z Eksploratora plików systemu Windows.



RYSUNEK 1.4. *Efekt uruchomienia programu*

Skróty klawiszowe Visual Studio

Skróty klawiszowe *F5* i *Ctrl+F5* służą do uruchamiania projektu w środowisku Visual Studio. Przydatnych skrótów jest jednak w Visual Studio znacznie więcej. Kilka z nich zebranych jest w tabeli 1.1. Spora część dotyczy edycji tekstu — są więc takie same jak np. w notatniku Windows czy w standardowych edytorach tekstu. Są jednak też skróty klawiszowe specyficzne dla Visual Studio. Dwa przykłady to: *Ctrl*+spacja, który wyświetla podpowiedzi do kodu,

oraz *Ctrl+Shift*+spacja, który wyświetla informację o argumentach wywoływanej w kodzie metody. Zapewniam, że będą często używane. Równie często wykorzystywany jest skrót *Ctrl+.* (kropka), którego należy użyć, gdy fragment kodu podkreślony jest na czerwono (kursor edytora musi znajdować się w obrębie tego fragmentu). Dzięki temu skrótowi będziemy mogli na przykład dodać brakującą przestrzeń nazw, co jest częstą sytuacją, lub zaimplementować brakujące metody deklarowane przez interfejs. Dodatkowo w rozdziale 8. poznamy też spory zestaw skrótów klawiszowych używanych podczas debugowania.

TABELA 1.1. *Podstawowe klawisze skrótów edytora Visual C#*

Kombinacja klawiszy	Funkcja
Ctrl+F	Przeszukiwanie kodu
Ctrl+H	Przeszukiwanie z zastąpieniem
F3	Poszukiwanie następnego wystąpienia szukanego ciągu
Ctrl+J	Menu uzupełniania kodu
Ctrl+spacja	Menu uzupełniania kodu lub uzupełnienie, jeżeli sytuacja jest jednoznaczna
Ctrl+Shift+spacja	Informacja o argumentach metody
Ctrl+L	Usunięcie bieżącej linii
Ctrl+S	Zapisanie bieżącego pliku
Ctrl+Z	Cofnięcie ostatnich zmian w kodzie
Ctrl+A	Zaznaczenie kodu w całym pliku
Ctrl+X, *Ctrl+C*, *Ctrl+V*	Obsługa schowka
F7, Shift+F7	Przełączenie między edytorem a widokiem projektowania (w aplikacjach z interfejsem)
Ctrl+Shift+B lub *F6*	Budowanie całego projektu (klawisz *F6* może nie działać[1])
F5	Kompilacja i uruchomienie aplikacji w trybie debugowania
Ctrl+F5	Kompilacja i uruchomienie aplikacji bez debugowania
Ctrl+. lub *Alt+Enter*	Pokaż rozwiązanie problemu (kod z czerwonym podkreśleniem)

[1] Wówczas polecenie budowania projektów w rozwiązaniu można przypisać do klawisza *F6*. W tym celu należy z menu *Narzędzia* wybrać polecenie *Dostosuj...*. Pojawi się okno dialogowe, w którym klikamy przycisk *Klawiatura...* (na dole okna). Pojawi się okno opcji z wybraną pozycją *Środowisko*, *Klawiatura*. W polu tekstowym *Pokaż polecenia zawierające* należy wpisać „Kompilowanie". W liście poniżej widoczne wówczas będą tylko polecenia zawierające to słowo. Wśród nich należy znaleźć polecenie *Kompilowanie.Kompilujrozwiązanie*. Jeszcze niżej zobaczymy, że przypisana jest do niego kombinacja *Ctrl+Shift+B*. W polu *Naciśnij klawisze skrótu* naciśnijmy klawisz *F6* i kliknijmy *Przypisz*. Po powrocie do edytora ten klawisz również będzie uruchamiał kompilację projektu.

Podpowiadanie kodu (IntelliSense)

Zmodyfikujmy odrobinę kod metody `Main`, wstawiając między nazwę klasy `Console` i metodę `WriteLine` słowo `Out` oddzielone kropkami (listing 1.2). Kropka (operator dostępu) rozdziela przestrzenie nazw od nazw klas, a także nazwy klas od nazw metod.

LISTING 1.2. *Wyświetlanie napisu „Hello World!"*

```
static void Main(string[] args)
{
    Console.Out.WriteLine("Hello World!");
}
```

> Podobnie jak w C++ i Javie także i w C# wielkość liter ma znaczenie. `Console` i `console` to dwa różne słowa. I podobnie jak w tych językach nie ma znaczenia sposób ułożenia kodu. Oznacza to, że między słowa kluczowe, nazwy metod, klas itp. możemy w dowolny sposób wstawiać spacje i znaki końca linii. Nie możemy jedynie łamać łańcuchów, które powinny mieć cudzysłów zamykający w tej samej linii co rozpoczynający.

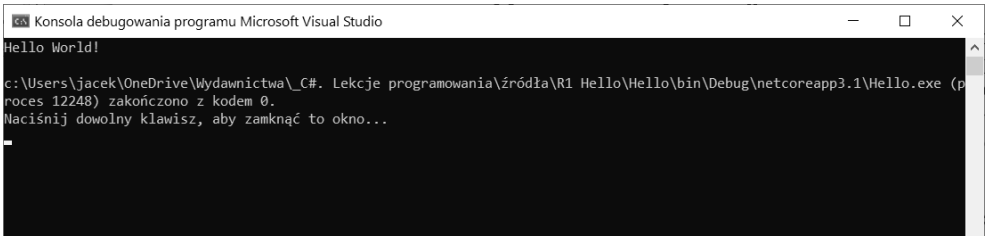
W edytorze po wstawieniu kropki po nazwie klasy `Console`, tj. operatora umożliwiającego dostęp do elementów składowych, pojawi się okno prezentujące wszystkie dostępne statyczne własności i metody tej klasy. Ta podpowiedź to element mechanizmu wglądu w kod o nazwie **IntelliSense**. Dzięki niemu pisanie kodu jest znacznie wygodniejsze. Wskazuje nam dostępne elementy, a jednocześnie pomaga w uniknięciu literówek. Wystarczy bowiem z pojawiającej się listy wybrać pozycję `Out`, zamiast wpisywać jej nazwę ręcznie. To samo stanie się, gdy postawimy kolejną kropkę po wpisaniu lub wybraniu `Out`. Wówczas pojawią się metody i własności dostępne w obiekcie `Out`. Jeżeli chcielibyśmy samodzielnie wpisać nazwę metody `WriteLine` (nie trzeba tego robić, bo już jest w kodzie), to gdy wpiszemy początek nazwy, a więc „wr" (nie dbając o wielkość liter) — automatycznie zostanie zaznaczona pierwsza pozycja, która odpowiada tej sekwencji. Wówczas wystarczy nacisnąć klawisz *Enter*, aby potwierdzić wybór, lub przesunąć wybraną pozycję na liście, korzystając z klawiszy ze strzałkami. Jeżeli niechcący zamkniemy okno IntelliSense, możemy je przywołać kombinacją klawiszy *Ctrl+J* lub *Ctrl*+spacja (por. tabela 1.1). IntelliSense pomoże nam również w przypadku argumentów metody `Console.WriteLine`. Warto z tej pomocy skorzystać.

> Mechanizm IntelliSense, podpowiadający składowe obiektów, może pracować w dwóch trybach. W pierwszym ogranicza się do pokazania listy pól, metod i własności. W drugim zaznacza jedną z nich. W tym drugim przypadku wystarczy nacisnąć klawisz *Enter*, aby kod został uzupełniony, podczas gdy w pierwszym spowoduje to tylko wstawienie znaku nowej linii do kodu. Do przełączania między tymi trybami służy kombinacja klawiszy *Ctrl+Alt*+spacja.

Strumienie wyjścia i wejścia

Po naciśnięciu kombinacji klawiszy *Ctrl+F5* projekt zostanie skompilowany i uruchomiony (bez debugowania), a na ekranie pojawi się znany już czarny ekran konsoli, na którym ponownie zobaczymy napis „Hello World!" (rysunek 1.5). Dodanie słowa `Out` nic nie zmieniło. Chciałem je jednak przedstawić. Statyczna własność `Out` zwraca obiekt typu `System.IO.TextWriter` (klasa `TextWriter` w przestrzeni nazw `System.IO`), który umie m.in. wyświetlać napisy w konsoli. Statyczna metoda `Console.WriteLine`, której wcześniej używaliśmy, jest „skrótem" do metody `Console.Out.WriteLine`.

Słowo „statyczny" zostanie szczegółowo wyjaśnione w drugiej części książki. W skrócie oznacza ono, że metoda statyczna może być wywoływana na rzecz klasy — nie musimy tworzyć jej instancji.



RYSUNEK 1.5. *Zawartość strumienia wyjścia w oknie konsoli w przypadku uruchomienia bez debugowania*

Do metody `Main` dodajmy jeszcze jedno polecenie (listing 1.3), które wymusi na aplikacji, aby ta po wyświetleniu napisu „Hello World!" wstrzymała działanie aż do naciśnięcia przez użytkownika klawisza *Enter*.

LISTING 1.3. *Metoda Main z poleceniem sczytującym z klawiatury linię, czyli ciąg znaków zatwierdzony klawiszem Enter*

```
static void Main(string[] args)
{
   Console.Out.WriteLine("Hello World!");
   Console.In.ReadLine();
}
```

Będę wymiennie stosował terminy „ciąg znaków" i „łańcuch", choć częściej ten drugi. W obu przypadkach chodzi o obiekty typu `string`.

Każde polecenie w C# musi kończyć się znakiem średnika `;`. Poszczególne elementy poleceń można rozmieścić w wielu liniach. Nie dotyczy to jednak słów kluczowych, które muszą być w całości.

Metoda `Main` zawiera w tej chwili dwa polecenia. Pierwsze uruchamia metodę `Console.Out.WriteLine`, która umieszcza w standardowym strumieniu wyjścia (`Console.Out`) napis podany w argumencie, w naszym przypadku „Hello World!”, dodając na końcu znak końca linii. Istnieje też podobnie działająca metoda `Write`, która tego znaku nie dodaje. Obie metody mają odpowiedniki w klasie `Console`, więc — jak już wiemy — zadziałałaby również instrukcja `Console.WriteLine` (bez `Out`). Podobnie jest w przypadku metody `Console.In.ReadLine`, która pobiera tekst ze standardowego strumienia wejścia (reprezentowanego przez `Console.In`), czyli z klawiatury. Możemy ją też uruchomić za pomocą metody wywołanej bezpośrednio na rzecz klasy `Console`, tj. `Console.ReadLine`. Metoda ta czeka na wpisanie przez użytkownika ciągu znaków potwierdzonego naciśnięciem klawisza *Enter*. Oprócz metody `Console.ReadLine`, odczytującej cały łańcuch, mamy do dyspozycji także metodę `Read`, odczytującą kolejny znak ze strumienia wejściowego (wprowadzonego i potwierdzonego klawiszem *Enter*), oraz metodę `Console.ReadKey`, zwracającą naciskany w danej chwili klawisz (ta ostatnia nadaje się do tworzenia interaktywnych menu sterowanych klawiaturą lub gier działających w konsoli).

Oprócz standardowych strumieni wejścia i wyjścia jest również strumień błędów (`Console.Error`). Działa analogicznie jak strumień wyjścia, drukując napisy w konsoli.

Wszystkiego po trochu

Wstrzymanie pracy aplikacji nie jest głównym zadaniem metody `ReadLine`. Służy ona przede wszystkim do odbierania od użytkownika informacji w postaci łańcucha (ciągu znaków). Aby zaprezentować jej prawdziwe zastosowanie, użyjemy jej do pobrania od użytkownika imienia. W tym celu umieszczamy w metodzie `Main` polecenia widoczne na listingu 1.4.

LISTING 1.4. *Rozbudowana wersja metody Main*

```
static void Main(string[] args)
{
    Console.WriteLine("Jak Ci na imię?");
    Console.Write("Napisz tutaj swoje imię: ");
    string imię = Console.ReadLine();
    if (imię.Length == 0)
    {
        Console.Error.WriteLine("\n\n\t*** Błąd: nie podano imienia!\n\n");
        return;
    }
    bool niewiasta = imię.ToLower()[imię.Length - 1] == 'a';
    if (imię == "Kuba" || imię == "Barnaba") niewiasta = false;
    Console.WriteLine(
        "Niech zgadnę, jesteś " + (niewiasta ? "dziewczyną" : "chłopakiem") + "!");

    Console.WriteLine("Naciśnij Enter...");
    Console.Read();
}
```

Przepisując powyższy kod, warto używać mechanizmu podpowiedzi IntelliSense. To powinno jak najszybciej wejść nam w krew. Należy również zwrócić uwagę na pojedyncze i podwójne cudzysłowy — nie mogą być stosowane zamiennie. Trzeba także uważać, aby nie zrobić literówki w słowach kluczowych i nazwach typów. Kompilator wprawdzie to zauważy i będzie starał się pomóc, wyświetlając komunikaty w trakcie kompilacji, a edytor będzie wskazywał problematyczne miejsca przez podkreślanie ich na czerwono już w trakcie wpisywania kodu, lecz najlepszym sposobem na uniknięcie literówek jest korzystanie z IntelliSense.

W kodzie widocznym na listingu 1.4 pojawiło się wiele nowych elementów, których znaczenie wyjaśni się w kolejnych rozdziałach. Zdefiniowane zostały dwie zmienne: `imię` typu `string` oraz `niewiasta` typu `bool`. Zmienne można traktować podobnie jak w matematyce — jak symbole mogące mieć jakieś wartości. Zmienna typu `string` może przechowywać teksty (łańcuchy), a zmienna typu `bool` wartości logiczne, tj. prawdę (wartość `true`) lub fałsz (`false`). Z punktu widzenia programisty zmienną można utożsamiać z miejscem w pamięci, które przechowuje wartość. Zmienna umożliwia zarówno odczytanie tej wartości, jak i jej zmianę.

Trzecie polecenie metody `Main` zapisuje do zmiennej `imię` łańcuch odebrany z klawiatury. Do zmiany wartości zmiennej (lub jej inicjacji, gdy zmiana następuje tuż po uruchomieniu, jak w tym przypadku) służy operator przypisania `=` (pojedynczy znak równości).

Drugą nowością jest słowo kluczowe `if`. To instrukcja warunkowa, której konstrukcja jest następująca: `if (`***`warunek`***`)` ***`polecenie`***`;`. Polecenie po nawiasie jest wykonywane tylko, jeżeli warunek jest spełniony, tj. wyrażenie ***`warunek`*** w nawiasach okrągłych ma wartość równą `true`. Weźmy na przykład instrukcję:

```
if (imię.Length == 0)
{
    Console.Error.WriteLine("\n\n\t*** Błąd: nie podano imienia!\n\n");
    return;
}
```

Jeżeli łańcuch przechowywany przez zmienną `imię` będzie miał zerową długość, pojawi się komunikat błędu i działanie metody `Main` — a tym samym i programu — zostanie zakończone. Odpowiada za to polecenie `return`, które kończy działanie metody. W komunikacie błędu wewnątrz cudzysłowów widoczne są sekwencje znaków `\n` i `\t`. Pierwszy to znak końca linii — dalsza część łańcucha będzie wyświetlana od nowej linii, natomiast drugi to znak tabulacji, który przesuwa dalszą część łańcucha o kilka miejsc w prawo.

Zwróćmy uwagę, że w warunku pojawia się operator porównania `==`, który zwraca wartość typu `bool` tj. wartość logiczną równą `true` lub `false`. W C# operator `=` służy do zmiany wartości zmiennej, a operator `==` do jej porównania z inną wartością lub zmienną. Nie należy ich mylić.

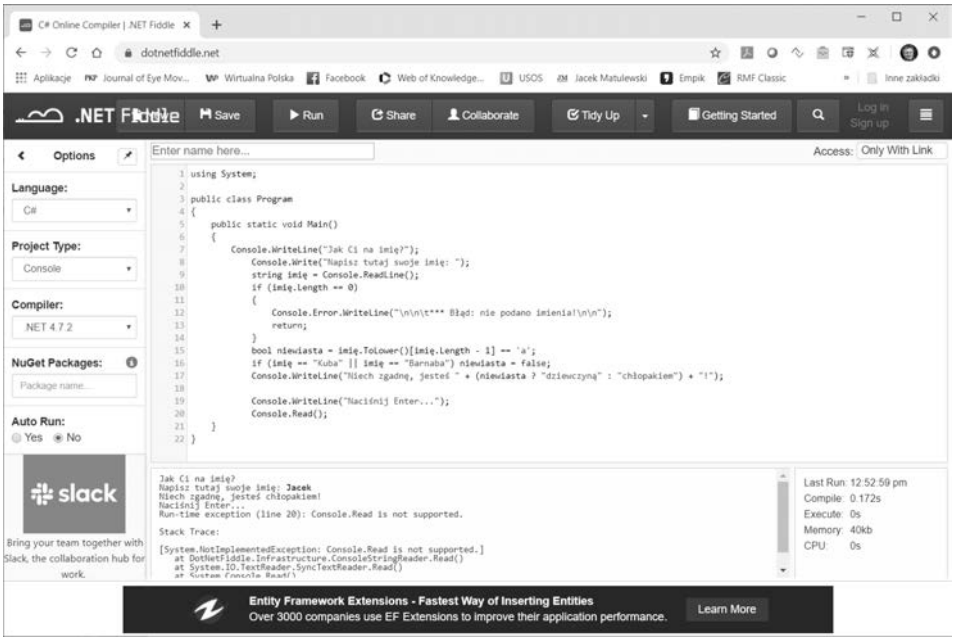
Instrukcję `if` można rozszerzyć o polecenie, które jest wykonywane, jeżeli warunek nie jest spełniony, np.

```
if (imię.Length == 0)
{
```

```
    Console.Error.WriteLine("\n\n\t*** Błąd: nie podano imienia!\n\n");
    return;
}
else
{
    Console.Error.WriteLine("Wpisano łańcuch o długości " + imię.Length);
}
```

Ostatnią rzeczą, na którą chciałbym zwrócić uwagę czytelnika, jest polecenie `bool niewiasta = imię.ToLower()[imię.Length - 1] == 'a';`, a konkretnie użyte w nim nawiasy kwadratowe []. Ale po kolei. To polecenie to deklaracja zmiennej `niewiasta` typu `bool`. Wartością tej zmiennej staje się wynik wyrażenia `imię.ToLower()[imię.Length - 1] == 'a'`. Co tu mamy? Po pierwsze wywołanie metody `imię.ToLower()` zwróci wpisany przez użytkownika łańcuch, zapisany w zmiennej `imię`, ale wszystkie jego znaki zmienione są na małe litery. Jeżeli więc wpisany został łańcuch „Jacek", wynikiem użycia metody `ToLower` będzie „jacek". Natomiast nawiasy kwadratowe zdefiniowane są dla łańcuchów w taki sposób, że zwracają literę (obiekt typu `char`) o podanym w nawiasach indeksie (nie pozwalają na jej zmianę). Jaki to indeks? Własność `imię.Length` zwraca długość łańcucha. W przypadku tekstu „jacek" zwróci wartość 5. Jednak indeks pierwszej litery równy jest 0, a nie 1. To oznacza, że łańcuch „jacek" nie ma litery o indeksie 5. Ostatnia litera ma indeks 4. Właśnie dlatego wartość w nawiasach jest zmniejszona o 1. Oznacza to, że wyrażenie `imię.ToLower()[imię.Length - 1]` to ostatnia litera wprowadzonego łańcucha (z uwzględnieniem konwersji do małych liter). Dla łańcucha „Jacek" będzie to „k". Jednak to jeszcze nie koniec, bowiem w wyrażeniu będącym warunkiem instrukcji `if` ta litera porównywana jest do litery „a". Zwróćmy uwagę, że litera ta podawana jest w pojedynczych cudzysłowach. Literał `'a'` to nie łańcuch, a jeden znak typu `char`, czyli takiego samego, jak ostatnia litera, którą odczytaliśmy z wprowadzonego łańcucha. Całe wyrażenie będzie wobec tego równe `true`, jeżeli ostatnia litera imienia będzie równa „a", a `false` — jeżeli nie. W ten sposób odgadujemy, czy wprowadzone imię jest żeńskie, czy męskie (rysunki 1.6 i 1.7). To oczywiście pozwala rozpoznać tylko polskie imiona, a i to z wyjątkami. Ze względu na te wyjątki po omawianym wyrażeniu jest kolejna instrukcja warunkowa `if`, która sprawdza, czy wprowadzone imiona to przypadkiem nie Barnaba lub Kuba. Pomijam inne wyjątki, jak np. Bonawentura, Dyzma i Kosma, a także żeńskie imiona, które nie kończą się na „a", np. Beatrycze, Miriam czy Noemi.

RYSUNEK 1.6. *Interakcja z użytkownikiem w tradycyjnym stylu z lat 80. ubiegłego wieku*



RYSUNEK 1.7. *Ta sama aplikacja uruchomiona w środowisku on-line. Proszę zwrócić uwagę, że klasa Program i metoda Main muszą mieć modyfikator public (w środowisku Visual Studio nie jest to konieczne). Pojawia się również błąd przy uruchomieniu metody Console.Read na końcu kodu*

Pytania

1. Jakie skróty klawiszowe służą do kompilacji i uruchomienia programu? Czym różni się działanie skrótów *F5* i *Ctrl+F5*?
2. Jakie strumienie są dostępne w klasie `Console`? Do czego służą?
3. Do czego służy metoda `Console.WriteLine`, a do czego `Console.ReadLine`?
4. Czym różnią się operatory = i ==? Jakiego typu wartości zwraca drugi z nich?
5. Do czego służy instrukcja `if..else`?
6. Jak nazywa się pierwsza metoda wywoływana po uruchomieniu programu?

Rozdział 2.

.NET Framework, .NET Core i .NET Standard

W 2002 roku Microsoft opublikował pierwszą wersję platformy .NET (ang. *.NET Framework*) i dedykowanego jej języka C#. Były one odpowiedzią na język Java i jego wirtualną maszynę, rozwijane wówczas przez firmę Sun. To była rewolucja w programowaniu aplikacji — w obu platformach celem kompilacji nie był konkretny typ komputera z jego specyficzną architekturą, ale wirtualne środowisko uruchomieniowe. W przypadku platformy .NET środowisko to nazywa się CLR (ang. *Common Language Runtime*). Takie podejście pozwoliło na większą przenaszalność wytworzonego oprogramowania (choć oficjalne wydania platformy były dostępne tylko dla systemów Windows), a tym samym zmniejszyło koszty jego wytwarzania. Równocześnie utrzymano wysoką wydajność dzięki technologii podwójnej kompilacji (o tym poniżej).

Platforma .NET była rozwijana aż do wersji o numerze 4.8, którą opublikowano w kwietniu 2019 roku. W tym momencie jej młodsza siostra, **platforma .NET Core**, która jest rozwijana dopiero od 2016 roku, ale w 2019 miała już wersję 3.0, uzyskała wystarczającą dojrzałość, aby przejąć pałeczkę sztandarowej platformy Windows[1]. Platforma .NET Core jest wolna i otwarta — rozwijana jest przez społeczność działającą pod kierunkiem Microsoft, a jej kod źródłowy jest publicznie dostępny. Od wersji 3.0 .NET Core pozwala na tworzenie tzw. aplikacji desktopowych, tj. aplikacji z graficznym interfejsem użytkownika (okienkowych), z wykorzystaniem zarówno kontrolek WPF, jak i Windows Forms. Platforma .NET Core ma ponadto tę ogromną przewagę nad starszą .NET Framework, że pozwala na tworzenie oprogramowania nie tylko dla systemu Windows, ale również dla Linux i macOS. Mówiąc o wieloplatformowości, należy pamiętać także o platformie UWP. Stanowi ona część wielkiego, ale niestety niezbyt udanego projektu „Windows na każdym ekranie", jest bowiem dostępna zarówno dla systemu Windows 10 (korzystają z niej aplikacje pobierane ze sklepu on-line Microsoft Store), jak i na konsole Xbox oraz systemy mobilne Windows 10 Mobile. Problem w tym, że te ostatnie się nie przyjęły. Nie należy także zapominać o rozwijanej od 2004 roku platformie Mono, której celem było

[1] Słowo *core* w nazwie (z ang. „rdzeń") oznaczało początkowe przeznaczenie platformy .NET Core. Miała to być mniejsza wersja platformy .NET Framework, dostępna na różnych platformach sprzętowych i systemach operacyjnych. Odniosła sukces, w efekcie czego postanowiono odtworzyć pełną funkcjonalność .NET Framework. Nazwa jednak pozostała.

umożliwienie przenoszenia aplikacji .NET na systemy inne niż Windows. Straciła ona na znaczeniu po uruchomieniu projektu .NET Core, ale nadal jest ważnym elementem Xamarin. A Xamarin to jeszcze jedna platforma związana z Microsoft i językiem C#, która pozwala na przygotowywanie w języku C# aplikacji mobilnych dla systemów Android i iOS (także dla Windows 10 Mobile). Jej ważnym atutem jest możliwość przygotowania wspólnego „rdzenia" aplikacji na te platformy, co bardzo ułatwia utrzymywanie aplikacji mobilnych i zmniejsza koszty.

Taka mnogość platform może być przytłaczająca, a dodajmy do tego jeszcze Unity, czyli system tworzenia gier w języku C#, który jest wspierany przez Microsoft i również korzysta z platformy .NET. Dobrą stroną jest to, że oprogramowanie dla wszystkich tych platform można przygotowywać w jednym języku. Przydatna byłaby jednak także możliwość łatwego przenoszenia między nimi kodu źródłowego, a nawet gotowych skompilowanych bibliotek. Stąd pomysł utworzenia **.NET Standard** (tabela 2.1). To nie jest kolejna platforma. To specyfikacja platform z rodziny .NET, włączając w to Mono i UWP. Nie można tworzyć aplikacji zgodnych z .NET Standard, można natomiast tworzyć takie biblioteki. Najnowsza wersja tej specyfikacji to 2.1. Jest z nią zgodna .NET Core 3.0, ale nie będzie już zgodnej z nią platformy .NET Framework. Natomiast w przypadku UWP i Unity takie wersje dopiero są zapowiadane.

TABELA 2.1. *Zgodność standardu .NET Standard z poszczególnymi wersjami platform*[2]

.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1	nie ma i nie będzie
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
UWP (wersja Windows)	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	jeszcze nie przygotowano
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	jeszcze nie przygotowano

[2] Źródło: *https://docs.microsoft.com/pl-pl/dotnet/standard/net-standard.*

Warto powiedzieć też kilka słów o języku C# i jego historii. Został zaprojektowany i jest rozwijany przez grupę programistów Microsoftu pod kierunkiem Andersa Hejlsberga. Był on wcześniej odpowiedzialny za kompilator Turbo Pascala, środowisko Delphi i bibliotekę VCL w firmie Borland, a od 1996 roku, już w Microsofcie, za projekt J++, a potem C# i platformę .NET oraz TypeScript. Język C#, jeżeli brać pod uwagę jego słowa kluczowe i podstawową składnię, bez wątpienia należy do tej samej rodziny co języki C++ i Java. Jego dynamiczny rozwój spowodował jednak, że pojawiło się w nim ostatnio wiele nowych konstrukcji, wykraczających poza ramy ustalone przez standard języka C++. Najlepszymi przykładami są zapytania LINQ i używane w nich operatory lub wsparcie dla programowania asynchronicznego. Inne elementy języka były natomiast często wprowadzone w C# wcześniej niż w C++, czego dobrym przykładem są wyrażenia lambda. Rozpoczynając naukę języka C#, można więc bazować na znajomości innych języków, ale warto przygotować się również na nowości. Najnowsza wersja języka C# 8.0 implementowana jest w platformie .NET Core 3.0 (specyfikacja .NET Standard 2.1). Ostatnia wersja platformy .NET Framework 4.8 jest natomiast zgodna z C# 7.0.

Ważnym składnikiem technologii .NET Core jest środowisko uruchomieniowe **CoreCLR**, odpowiednik CLR w .NET Framework. Separuje ono aplikacje od warstwy systemu operacyjnego, co daje kilka istotnych korzyści. Przede wszystkim pozwala chronić stabilność systemu. Ponadto umożliwia wprowadzenie nowych mechanizmów, których nie oferuje sam system operacyjny. Najlepszym przykładem jest zarządzanie pamięcią. Korzyścią, którą chciałbym jednak podkreślić w szczególny sposób, jest wspomniana już wyżej przenaszalność. Platforma .NET Core jest dostępna nie tylko dla systemu Windows, ale również Linux i macOS.

Kod źródłowy C# aplikacji lub biblioteki nie jest kompilowany bezpośrednio do kodu maszynowego rozumianego przez procesor. Kompilacja jest dwustopniowa. W pierwszej fazie zostaje on skompilowany do kodu pośredniego, wspólnego dla wszystkich środowisk uruchomieniowych .NET Core, bez względu na system operacyjny i procesor. Ten kod jest czytelny dla środowiska uruchomieniowego CoreCLR. Druga faza to kompilacja kodu pośredniego przez CoreCLR za pomocą kompilatorów *just-in-time* (skrót JIT). Nie odbywa się ona bezpośrednio po pierwszej kompilacji, ani nawet na tym samym komputerze. Platforma .NET Core przeprowadza ją dopiero w momencie uruchamiania programu.

Nieco zamieszania może powodować fakt, że wynikiem pierwszej kompilacji kodu źródłowego C# aplikacji są pliki *.exe* (w .NET Core 3.0 i nowszych i we wszystkich wersjach tradycyjnej platformy .NET) lub *.dll* (do .NET Core 2.2). Mimo tego samego rozszerzenia i nagłówka rozpoczynającego się od „MZ" struktura tych plików jest zupełnie inna niż tradycyjnych plików wykonywalnych i bibliotek zawierających kod uruchamiany w platformie Win32 lub Win64. Zawierają one bowiem kod pośredni. Z punktu widzenia użytkownika, jeżeli tylko platforma .NET jest zainstalowana w systemie, nie ma to jednak większego znaczenia — po prostu uruchamia on program, klikając dwukrotnie plik *.exe* (od .NET Core 3.0). Jednak z punktu widzenia systemu operacyjnego różnica jest ogromna.

Pytania

1. Czym różni się .NET Framework od .NET Core? Sprawdź, jakie są ostatnie wersje tych platform.
2. Jakie są zalety środowiska uruchomieniowego, np. CoreCLR?
3. Czym jest .NET Standard?

Programistą być

Książka, którą trzymasz w ręku, to znakomity podręcznik do nauki programowania od podstaw. Autor, wieloletni wykładowca akademicki i nauczyciel studentów informatyki, proponuje naukę programowania w języku C#. To wybór nieprzypadkowy. C# jest nowoczesnym, obiektowym, stale rozwijanym i popularnym językiem, którym opiekuje się firma Microsoft. Dodatkowo — dzięki podobieństwu C# do innych często używanych języków z rodziny C, C++ i Java — wszystko, czego nauczysz się z podręcznika, możesz wykorzystać także do programowania w tych językach.

Zakres zagadnień omawianych w książce obejmuje:

- język programowania C# 6.0, w tym posługiwanie się zmiennymi, metodami, tablicami oraz kolekcjami danych z platform .NET i .NET Core
- podstawowe techniki programowania obiektowego
- różne formaty przechowywania i eksportu danych
- darmowe środowisko programistyczne Microsoft Visual Studio 2019 Community

Jeśli chcesz uczyć się programować od podstaw — to książka dla Ciebie. Jeśli znasz już podstawy, ale chcesz poznać C# oraz platformy .NET i .NET Core — to również dobry wybór.

Uwaga! Podręcznik zawiera ćwiczenia, w których krok po kroku opisano tworzenie aplikacji konsolowych w darmowym Visual Studio 2019 i omówiono ich kod, a także bogaty zestaw zadań do samodzielnego wykonania!

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-1102-2
9 788328 311022

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 119,00 zł