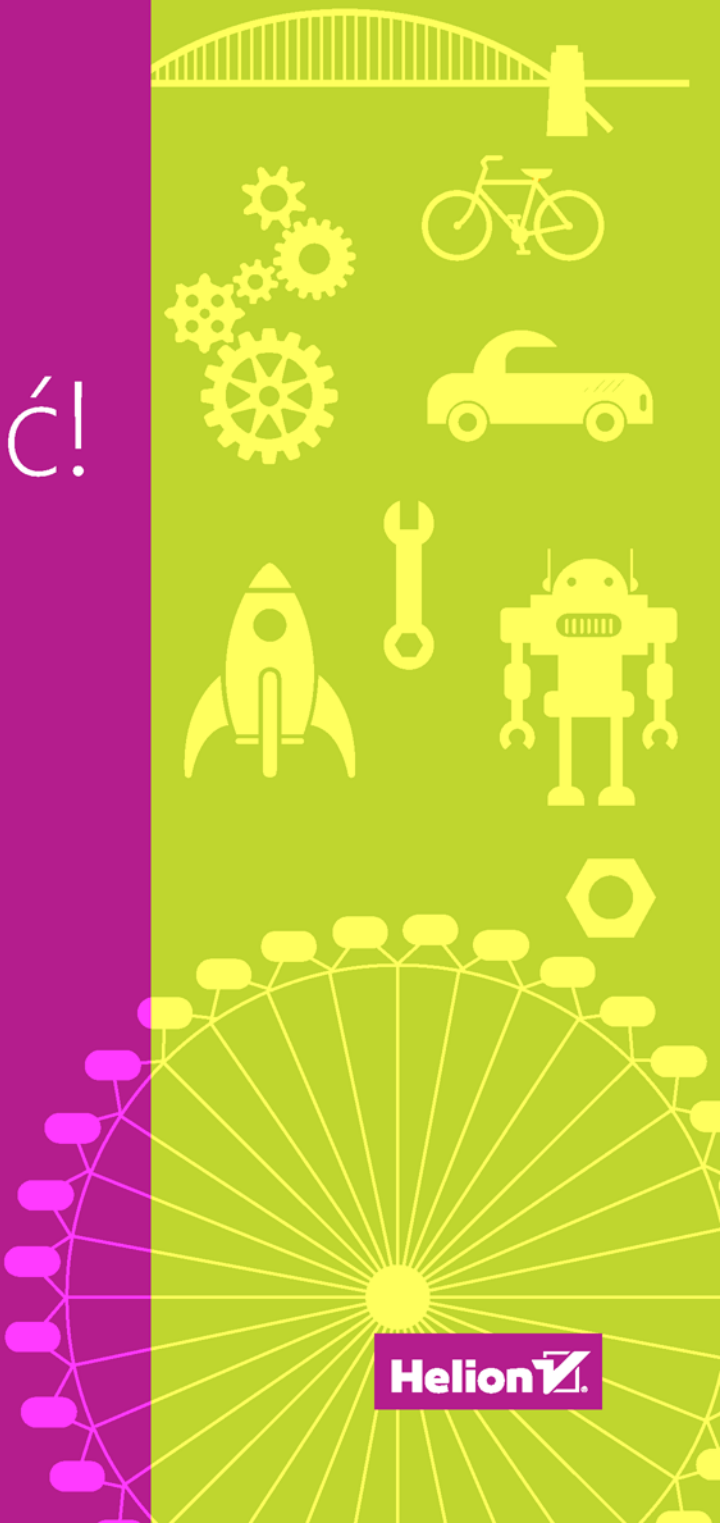
C#

Zacznij programować!

Rob Miles

Microsoft

Helion



Tytuł oryginału: Begin to Code with C#

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-6060-0

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/czaczp.zip

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

Printed in Poland.

Dla Mary

Spis treści w skrócie

Część I Podstawy programowania

Część II Programowanie zaawansowane

Część III Tworzenie gier

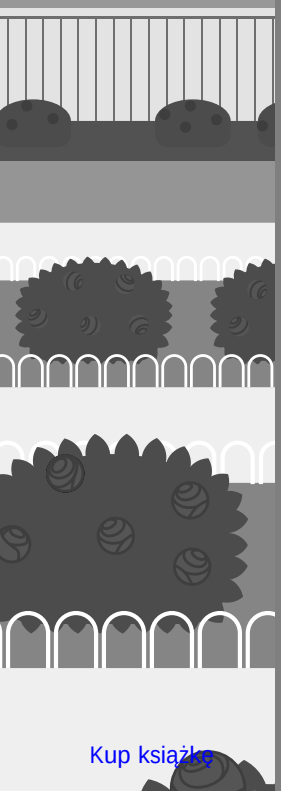
Część IV Tworzenie aplikacji (dodatek na WWW)

Spis treści

Część III Tworzenie gier

3.

Pisanie programów



Czego nauczysz się w tym rozdziale?

Ponieważ dowiedziałeś się już co nieco o komputerach, programach i programistach, możesz zacząć myśleć o pisaniu kodu programu.

W tym rozdziale przeanalizujesz dokładnie kilka programów C#, aby zrozumieć, jak działają. Nazywam te programy „aplikacjami Snaps" (czyli Snapami), ponieważ używają biblioteki Snaps, będącej prostym zbiorem zasobów programistycznych, które pomagają robić różne rzeczy „w mgnieniu oka" (ang. *in a snap*). Analizując, w jaki sposób te programy wykorzystują różne Snapy — dyskretne elementy funkcjonalności programistycznej lub inaczej mówiąc, zachowania dostarczane przez bibliotekę — poznasz podstawy programowania w języku C#. W międzyczasie będziesz uczyć się używania programu Visual Studio do tworzenia elementów kodu w rozwiązaniu *BeginToCodeWithCSharp* i zarządzania nimi, a także dowiesz się, co zrobić, gdy kompilator będzie narzekał, iż „według niego" dany program nie ma sensu.

Po lekturze tego rozdziału będziesz w stanie tworzyć programy zapewniające proste rozwiązania niektórych rzeczywistych problemów.

Struktura programu C#

Przyjrzyjmy się bliżej kilku aplikacjom Snaps, aby zrozumieć ich elementy i organizację tych elementów. Powitanie, które wyświetla się podczas uruchamiania rozwiązania *BeginToCode-WithCSharp*, nie jest skomplikowane, ale to dobre miejsce do rozpoczęcia naszej analizy. Zbadaliśmy pokrótce kod, który tworzy to doświadczenie, kiedy analizowaliśmy *MyProgram.cs* w rozdziale 2. Spójrz teraz na plik o nazwie *Ch03_01_WelcomeProgram.cs*. (Na wypadek, gdybyś zapomniałeś: użyj narzędzia *Solution Explorer* do nawigacji przez foldery rozdziałów rozwiązania i znalezienia tego pliku, a następnie wybierz plik, aby wyświetlić jego kod w oknie edytora).

Zwróć uwagę, że kod jest prawie dokładnie taki sam jak kod w pliku *MyProgram.cs*, więc ten program powinien dać nam to samo doświadczenie, prawda? Sprawdźmy to. Uruchom ponownie nasze rozwiązanie, wybierz *Chapter 03* na liście folderów oraz *Ch03_01_WelcomeProgram* na liście aplikacji Snaps, a następnie uruchom aplikację. Tak, otrzymujemy efekt taki jak poprzednio, co ma sens. Teraz przeanalizujmy ten program, aby dowiedzieć się, jak działa. Jego kod jest pokazany poniżej, a każda część programu została opatrzona objaśnieniem. Zbadamy te części linia po linii w kolejnych punktach tego podrozdziału.

```
using SnapsLibrary;

public class Ch03_01_WelcomeProgram
{
    public void StartProgram()
    {
        SnapsEngine.SetTitleString("Rozpoczynamy kodowanie z C#");
        SnapsEngine.DisplayString("Witaj w świecie aplikacji Snaps");
    }
}
```

Identyfikacja zasobów.

Rozpoczynanie definicji klasy.

Deklarowanie metody StartProgram.

Ustawianie tytułu i wyświetlanie wiadomości.

Identyfikacja zasobów

```
using SnapsLibrary;
```

Kompilator C# opisałem w rozdziale 2. Jest to program, który konwertuje wysokopoziomowy program C# (taki jak ten, który analizujemy) na kod maszynowy, możliwy do uruchomienia wewnątrz komputera. Gdy uruchamiasz kod napisany w języku C#, kompilator wbudowany w Visual Studio konwertuje dany program na kod maszynowy, aby można go było uruchomić na komputerze. Program C# może zawierać linie zwane **dyrektywami**, które przekazują instrukcje kompilatorowi. Ta pierwsza linia programu to dyrektywa `using`.

Jako programista często będziesz chciał korzystać z gotowych elementów oprogramowania, podobnie jak kucharz czasami używa gotowego ciasta. Gotowe programy C# są spakowane jako biblioteki komponentów, które można dodawać do rozwiązania Visual Studio. Jak już wspomniałem, przykładem takiej biblioteki jest biblioteka Snaps, którą przygotowałem, aby ułatwić Ci rozpoczęcie nauki programowania. Dyrektywa `using` określa tutaj bibliotekę Snaps jako zasób, który został dodany do naszego rozwiązania, i jak zobaczysz za chwilę, ten program użyje pewnego elementu z tej biblioteki, a konkretnie klasy `SnapsEngine`. Ta dyrektywa `using` instruuje kompilator: „Jeśli wspominam o czymś, czego wcześniej nie widziałeś, zajrzyj do `SnapsLibrary` i sprawdź, czy tam tego nie ma". To trochę tak, jakby powiedzieć do naszego kucharza: „Jeśli potrzebujesz użyć ciasta, zajrzyj do lodówki". Pierwsze programy, które napiszemy, używają tylko elementów z biblioteki `SnapsLibrary`. Później będziemy tworzyć programy używające innych bibliotek.

ANALIZA KODU

Używanie dyrektywy using

W niektórych ramkach „Analiza kodu", takich jak ta, nie będziesz musiał analizować żadnego konkretnego kodu, aby zastanowić się nad pytaniami związanymi z kodem.

Pytanie: Czy dyrektywa `using` w rzeczywistości pobiera bibliotekę, której program chce użyć?

Odpowiedź: Nie. Może się to wydawać dezorientujące, ale dyrektywa `using` mówi kompilatorowi, gdzie szukać elementów, które można wykorzystać w programie. Zasoby dostępne dla programu są konfigurowane w projekcie Visual Studio. Możemy zmienić dyrektywę `using`, aby nakłonić kompilator do używania kodu z innych miejsc. To jakby powiedzieć kucharzowi: „Jeśli chcesz użyć ciasta, sprawdź przy umywalce", żeby użył zasobu z innej lokalizacji.

Pytanie: Jeśli dodam wiele dyrektyw `using`, czy zwiększy to rozmiar programu?

Odpowiedź: Nie. Ta dyrektywa wskazuje jedynie kompilatorowi, gdzie szukać elementów programu. Nie dodaje niczego do samego programu.

Rozpoczynanie definicji klasy

```
public class Ch03_01_WelcomeProgram
```

C# można nazwać językiem programowania **obiektowego**. To dlatego, że we wszechświecie C# wszystko jest **obiektem**. Obiekty w programie C# mogą być bardzo proste, jak pojedyncza liczba, lub złożone, jak cała gra wideo. Obiekt może zawierać inne obiekty. Wszystko, co jest zawarte w innym obiekcie, jest nazywane **składową** tego obiektu.

Projekt obiektu możemy wyrazić poprzez definicję **klasy** C#. Definicja klasy C# może opisywać składowe reprezentujące dane (wartości, które obiekt może przechowywać) oraz składowe reprezentujące zachowania (działania, które obiekt może wykonywać). Kiedy projektujesz obiekt, piszesz kod C#, który określa te dwie rzeczy. Ta linia programu informuje kompilator, że wyrażamy projekt klasy (`class`) o nazwie `Ch03 _ 01 _ WelcomeProgram`.

Więcej o klasach i obiektach dowiesz się w dalszej części książki.

ANALIZA KODU

Klasy i obiekty

Pytanie: Czy definicja klasy jest jedynym sposobem definiowania obiektu?

Odpowiedź: Nie. Istnieją inne rodzaje obiektów C#, które zobaczysz później.

Pytanie: Czy zdefiniowanie klasy faktycznie tworzy obiekt?

Odpowiedź: Nie. Klasę należy traktować jako plany lub projekt obiektu, podobne do planów domku na drzewie. Posiadanie planów domku na drzewie nie daje Ci w rzeczywistości gotowego domku, a definicja klasy nie daje obiektu.

Pytanie: Czy wszystkie klasy muszą zawierać dane i zachowania?

Odpowiedź: Nie. Niektóre klasy zawierają tylko składowe reprezentujące dane, a inne zawierają tylko składowe reprezentujące zachowanie. Przykładowo biblioteka Math, której jeszcze nie poznałeś, zawiera klasy potrafiące wykonywać funkcje matematyczne.

Pytanie: Kiedy ten program tak naprawdę tworzy obiekt na podstawie klasy `Ch03 _ 01 _ WelcomeProgram`?

Odpowiedź: Dzieje się to automatycznie. Sekwencja wygląda następująco: użytkownik uruchamia aplikację *BeginToCodeWithCSharp*, a następnie wybiera program *Ch03_01_WelcomeProgram* i uruchamia go. Aplikacja *BeginToCodeWithCSharp* tworzy obiekt na podstawie klasy `Ch03 _ 01 _ WelomeProgram`, a następnie uruchamia zachowanie `StartProgram` wewnątrz tego obiektu.

Deklarowanie metody StartProgram

```
public void StartProgram()
```

Zachowania w obiekcie są wyrażane w formie **metod**. Metoda to opisujący jakieś zachowanie fragment kodu C#, któremu nadano nazwę. Program może uruchomić kod z metody, podając

po prostu jej nazwę — nazywa się to **wywoływaniem metody**. Na początku będziesz wywoływał metody, które zostały już napisane (przeze mnie), ale później utworzysz własne.

Pojedyncza klasa tego programu, czyli `Ch03 _ 01 _ WelcomeProgram`, ma tylko jedno zachowanie, którym jest metoda o nazwie `StartProgram`. Początek metody `StartProgram` jest oznaczony przez deklarację `public void StartProgram()`. (**Modyfikator** metody `public` i typ zwracany `void` informują nas o naturze tej metody, ale nie musimy się w tej chwili wdawać w te szczegóły). Metoda `StartProgram` jest wyjątkowa. Jest to punkt wejścia dla aplikacji Snaps. Innymi słowy, aby uruchomić aplikację Snaps, należy wywołać metodę `StartProgram`.

Klasa tego programu nie zawiera żadnych składowych reprezentujących dane, ale później zaprojektujemy kilka obiektów, które będą je (dane) zawierały.

ANALIZA KODU

Deklarowanie metod w klasach

Pytanie: Jaka jest różnica między zachowaniem a metodą?

Odpowiedź: Zachowanie to akcja, którą może wykonywać obiekt. Metoda jest rzeczywistym kodem C#, który zapewnia to zachowanie.

Pytanie: Czy klasa może zawierać więcej niż jedną metodę?

Odpowiedź: Tak. To programista decyduje, ile zachowań powinna zapewniać klasa, i to on pisze metodę dla każdego z nich. Program demonstracyjny, któremu się przyglądamy, ma tylko jedno zachowanie: ma uruchomić demo. Później będziemy tworzyć klasy z wieloma metodami.

Pytanie: Jak używa się metody `StartProgram`?

Odpowiedź: `StartProgram` jest specjalną metodą pod tym względem, że definiuje punkt początkowy dla wszystkich aplikacji Snaps. Podczas pracy w środowisku Snaps dostarczonym przez bibliotekę Snaps, zawsze w celu uruchamiania programów będziemy wywoływać metodę `StartProgram`.

Ustawianie tytułu i wyświetlanie wiadomości

```
SnapsEngine.SetTitleString("Rozpoczynamy kodowanie z C#");
SnapsEngine.DisplayString("Witaj w świecie aplikacji Snaps");
```

Pierwsza z tych dwóch linii kodu to pierwsza **instrukcja** C# w metodzie `StartProgram`. Instrukcje to części programu, które wykonują zadania. Instrukcja może wywoływać metodę, podejmować decyzję lub manipulować jakimiś danymi. Instrukcje są przechowywane wewnątrz metod i są wykonywane, gdy metoda jest używana. Metoda `StartProgram` zawiera tylko dwie instrukcje; większe programy będą ich zawierać znacznie więcej. Te dwie instrukcje w metodzie `StartProgram` wywołują tak naprawdę inne metody.

Poszczególne instrukcje z metody są wykonywane po kolei, zaczynając od pierwszej i sukcesywnie przechodząc do kolejnych. Istnieje kilka typów instrukcji, których można używać; poznasz je wszystkie w trakcie nauki języka C#. Znak średnika (`;`) wskazuje koniec każdej instrukcji.

Ta pierwsza instrukcja ustawia tytuł naszego programu na *Rozpoczynamy kodowanie z C#*. W tym celu używa klasy `SnapsEngine`. Klasa `SnapsEngine` jest częścią biblioteki Snaps — zasobu, który zidentyfikowaliśmy w pierwszej linii tego programu — i zapewnia wiele zachowań, które możemy wykorzystać w naszych programach. Możesz potraktować `SnapsEngine` jako swoistego „lokaja", który potrafi robić różne rzeczy dla pisanych przez Ciebie programów.

Każde zachowanie `SnapsEngine` jest dostarczane jako metoda C#, którą nasze programy mogą wywoływać. W tym przykładzie możesz zobaczyć, jak używać metody `SetTitleString` z klasy `SnapsEngine`. Metodzie `SetTitleString` trzeba przekazać łańcuch znaków, który ma być użyty jako tytuł programu, podobnie jak poleceniu „Podaj mi drinka" wydawanemu lokajowi towarzyszy rodzaj napoju, jakiego sobie życzymy. Łańcuch znaków w języku C# jest podawany w nawiasach po nazwie metody, którą wywołujemy. Informacje dodawane do wywołania metody są nazywane **argumentem** metody.

Jeśli chodzi o sam łańcuch znaków, podwójne cudzysłowy (`"`) w instrukcji oznaczają jego początek i koniec — łańcuch znaków rozpoczyna się zaraz po pierwszym podwójnym cudzysłowie i kończy się bezpośrednio przed drugim. W języku C# jest to konwencją, że gdy chcesz określić tekst w postaci łańcucha znaków, umieszczasz go w cudzysłowach w taki właśnie sposób. Jeśli w łańcuchu znaków dodamy spacje — na przykład `" Witamy w aplikacjach Snaps "` — również zostaną one wyświetlone w tytule programu (chociaż użytkownik może ich nie zauważyć).

Druga instrukcja działa tak samo jak pierwsza. Wywołuje metodę z klasy `SnapsEngine`, która wyświetla łańcuch znaków jako komunikat na ekranie aplikacji Snaps (zamiast ustawiania łańcucha znaków jako tytułu na ekranie). Kiedy zobaczyłeś nazwę metody `DisplayString`, spodziewałeś się zobaczyć cudzysłowy i łańcuch znaków w nawiasach po nazwie metody? Bardzo dobrze!

ANALIZA KODU

Wywoływanie metod z klas

Pytanie: Gdzie zadeklarowano metodę `SetTitleString`?

> **Odpowiedź:** Metoda `SetTitleString` jest zadeklarowana w klasie `SnapsEngine` dokładnie w taki sam sposób, jak metoda `StartProgram` jest zadeklarowana w klasie `Ch03 _ 01 _ WelcomeProgram`. Później dowiesz się, jak tworzyć własne metody w klasach.

Pytanie: Co się stanie, jeśli nie podam metodzie `SetTitleString` łańcucha znaków, na którym będzie mogła pracować?

> **Odpowiedź:** Projekt metody `SetTitleString` określa, że gdy jest ona wywoływana, należy dostarczyć do niej łańcuch znaków. Jeśli program nie zapewni łańcucha znaków jako argumentu wywołania metody, kompilator będzie narzekał, że ten program jest nieprawidłowy.

Pytanie: Dlaczego łańcuch znaków dostarczany do metody `SetTitleString` musi być umieszczony w nawiasach? Kompilator z pewnością jest w stanie określić, że łańcuch znaków, który ma zostać wyświetlony, rozpoczyna się za znakiem podwójnego cudzysłowu.

> **Odpowiedź:** Nawiasy musimy stosować po to, żeby wskazać kompilatorowi początek i koniec listy argumentów dostarczanych do metody. Do metody `SetTitleString` dostarczany jest tylko jeden element, ale inne metody mogą mieć ich wiele. Jeśli spojrzysz w tekst programu, odkryjesz, że metoda `StartProgram` została określona w taki sposób, aby akceptować „pustą" listę argumentów, która oznacza, że nie działa ona na żadnych elementach. Projektanci języka C# użyli różnych znaków, aby zdefiniować granice różnych elementów programu (*ograniczyć* je). Jak widzieliśmy, łańcuchy znaków są *ograniczane* przez podwójne cudzysłowy. Listy argumentów są ograniczone przez otwierające i zamykające nawiasy okrągłe: `(` i `)`. Zawartość klasy i ciało metody są ograniczane przez nawiasy klamrowe: `{` i `}`. Jak można się spodziewać, kompilator jest bardzo skrupulatny w upewnianiu się, że użycie tych ograniczników „ma sens" i odrzuci każdy program, który będzie miał niedopasowane ograniczniki.

Te dwie instrukcje, które właśnie przeanalizowaliśmy (ustawiające tytuł ekranu i wyświetlające wiadomość), możesz potraktować jako „ładunek" przykładowego programu. Reszta kodu otaczającego te instrukcje zapewnia strukturę dla tych akcji. Aby napisać większy program, wystarczy zreplikować tę strukturę i dodać więcej instrukcji. Skoro już wiesz, z czego składa się prosty program, możesz zacząć tworzyć własne programy, używając aplikacji Snaps jako punktu wyjścia. Możesz na przykład utworzyć program wyświetlający dwie wiadomości, zamiast tytułu i wiadomości, tak jak zrobiłem to w programie *Ch03_02_MoreStatements.cs*:

```
using SnapsLibrary;
```

```
public class Ch03_02_MoreStatements
{
    public void StartProgram()
    {
        SnapsEngine.DisplayString("Witaj, świecie");
        SnapsEngine.DisplayString("Żegnajcie, kurczaki");
    }
}
```

Pierwsza instrukcja.

Druga instrukcja.

W tym programie metoda `SetTitleString` nie jest wywoływana, a te dwie instrukcje wywołują metodę `DisplayString`, żeby program wyświetlał najpierw jedną wiadomość, a potem drugą. W swoim programie możesz umieścić bardzo dużą liczbę instrukcji. Mógłbyś napisać na przykład program wyświetlający całą przemowę gettysburską (lub jakiś inny długi tekst) po jednym łańcuchu znaków na raz, dodając po prostu więcej instrukcji. Należy jedynie pamiętać, że gdy program zostanie uruchomiony, poszczególne instrukcje będą wykonywane po kolei. Powyższy program zawsze najpierw wyświetli komunikat *Witaj, świecie*, a dopiero po nim *Żegnajcie, kurczaki*.

Kiedy więc wyświetlasz łańcuch znaków za pomocą metody `DisplayString`, zastępuje ona łańcuch znaków, który był wyświetlany przez poprzednie wywołanie `DisplayString`, jeśli takie miało miejsce. Właśnie dlatego w naszym przykładzie *Witaj, świecie* jest zastępowane przez *Żegnajcie, kurczaki*. Później dowiesz się, jak wyświetlać na ekranie wiele linii tekstu. Jeśli chcesz, metody `DisplayString` możesz także użyć do wyświetlania bardzo długich wiadomości; tekst jest wtedy automatycznie zawijany, jeśli wychodzi poza krawędź ekranu. Gdy spróbujesz wyświetlić jakąś niezwykle długą wiadomość, przekonasz się, że będzie się ona rozciągać aż poza dolną krawędź ekranu, a użytkownik nie będzie w stanie odczytać jej w całości.

Dodatkowe Snapy

Co jakiś czas będę przedstawiał kolejne Snapy — zachowania dostarczane przez bibliotekę Snaps — którymi możesz się pobawić. Możesz ich używać w swoich programach w taki sam sposób, w jaki przeanalizowane przez nas programy używają `DisplayString`.

SpeakString

Możesz tworzyć programy, które wypowiadają tekst zamiast go wyświetlać. Oto przykład:

```
using SnapsLibrary;

class Ch03_03_Speaking
{
    public void StartProgram()
    {
        SnapsEngine.SpeakString("Cześć. Jestem Twoim przyjaznym komputerem.");
    }
}
```

Metoda `SpeakString` jest używana w taki sam sposób, co metoda `DisplayString`, ale powoduje, że komputer wypowiada podany tekst zamiast wyświetlać go na ekranie. Jest to przydatna metoda, ponieważ ułatwia tworzenie programów, które mogą rozmawiać z użytkownikiem.

ANALIZA KODU

Wypowiadanie i wyświetlanie tekstu

Przeanalizujmy pewien kod i spróbujmy zrozumieć, dlaczego nie robi tego, co powinien. Załóżmy, że ten program napisał Twój młodszy brat. Chciał, żeby program wyświetlał wiadomość *Komputer działa*, a następnie mówił *Komputer działa*, ale narzeka, że komunikat pojawia się na ekranie dopiero, gdy komputer skończy mówić.

```
using SnapsLibrary;
class Ch03_04_DoubleOutput
{
    public void StartProgram()
    {
        SnapsEngine.SpeakString("Komputer działa");
        SnapsEngine.DisplayString("Komputer działa");
    }
}
```

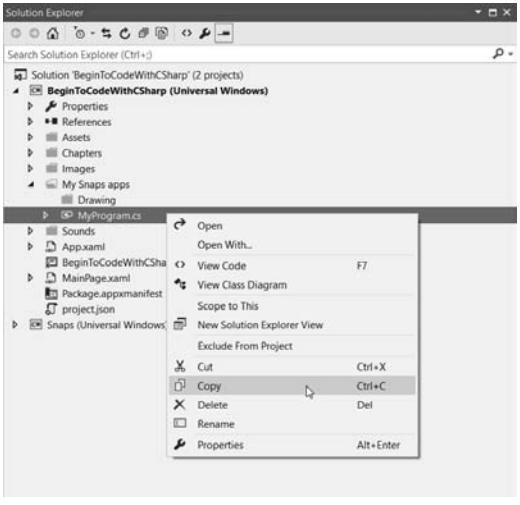
Pytanie: Dlaczego komunikat pojawia się na ekranie dopiero wtedy, gdy komputer skończy czytać tekst?

Odpowiedź: Kiedy próbujesz ustalić, co robi program, często przydaje się przejść przez instrukcje po kolei w taki sposób, w jaki wykonałby je komputer. Komputer czyta tekst zanim komunikat zostanie wyświetlony, ponieważ ściśle przestrzega kolejności instrukcji. Metoda `DisplayString` nie jest uruchamiana dopóki swojego działania nie zakończy metoda `SpeakString`. Ten problem można rozwiązać poprzez odwrócenie kolejności instrukcji. Zostało to zrobione w programie *Ch03_05_DoubleOutputFixed.cs*, który możesz przejrzeć w Visual Studio.

Tworzenie nowych plików programu

Programowanie jest bardzo kreatywne, a podczas lektury tej książki będziesz tworzyć własne programy. Mam nadzieję, że będziesz miał własne pomysły na programy i będziesz je budował równolegle z tymi, które proponuję. Każdy nowy program, który utworzysz, będzie nową aplikacją Snaps, którą inni uczniowie będą mogli analizować lub wykorzystywać.

Nową aplikację Snaps możesz utworzyć używając jako punktu wyjścia programu *MyProgram.cs*. Zacznij (jak zawsze) od otwarcia pliku rozwiązania *BeginToCodeWithCSharp*, a następnie użyj narzędzia *Solution Explorer*, aby znaleźć ten plik w folderze *My Snaps apps* w projekcie *BeginToCodeWithCSharp*. Kliknij prawym przyciskiem myszy plik w oknie *Solution Explorera*, aby otworzyć menu kontekstowe, a następnie wybierz *Copy* (kopiuj), jak pokazano na rysunku 3.1.



Rysunek 3.1. Kopiowanie programu

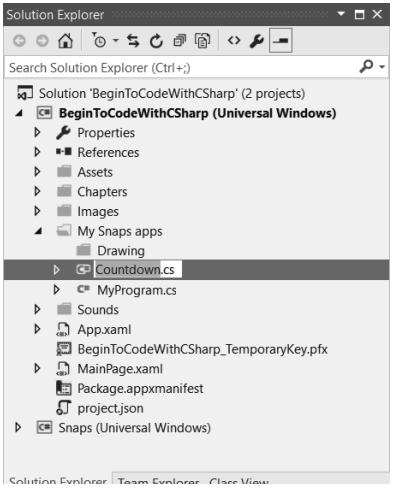
Teraz wklej tę kopię do folderu *My Snaps apps*, klikając folder prawym przyciskiem myszy i wybierając *Paste* (wklej), jak pokazano na rysunku 3.2.

Rysunek 3.2. Wklejanie programu

Na rysunku 3.3 widać tę kopię o nazwie *MyProgram — Copy.cs*, utworzoną w folderze *My Snaps apps*.

Rysunek 3.3. Skopiowany program pojawił się w folderze

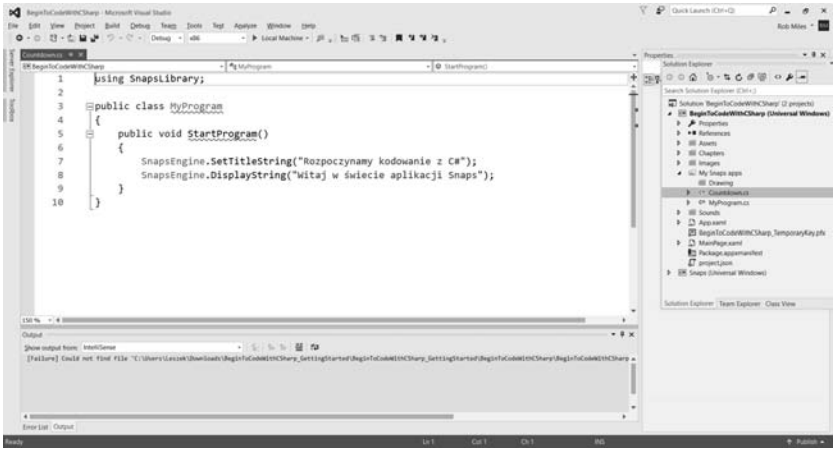
Zmieńmy nazwę tego nowego pliku, żeby odzwierciedlała funkcjonalność nowej aplikacji Snaps, którą zamierzasz zbudować. W oknie *Solution Explorer* kliknij prawym przyciskiem myszy skopiowany plik (ten, który zawiera w nazwie słowo *Copy*), aby ponownie otworzyć menu kontekstowe, po czym wybierz polecenie *Rename* (zmień nazwę). (Nie pokażę tego kroku, ponieważ jestem pewien, że doskonale wiesz, co robić!). Teraz możesz wprowadzić nową nazwę swojej aplikacji, jak pokazano na rysunku 3.4.



Rysunek 3.4. Wprowadzanie nowej nazwy

Zmień nazwę programu na **`Countdown`** (odliczanie). Uważaj, aby nie usunąć *.cs* na końcu nazwy. Jeżeli usuniesz tę część nazwy pliku, program Visual Studio nie będzie wiedział, że dany plik jest programem C# i nie zadziała poprawnie podczas próby uruchomienia programu. Po zakończeniu wprowadzania nazwy naciśnij *Enter*. Masz teraz kopię oryginalnego programu w pliku o nazwie *Countdown.cs*. Wkrótce zrozumiesz, dlaczego wybrałem taką nazwę.

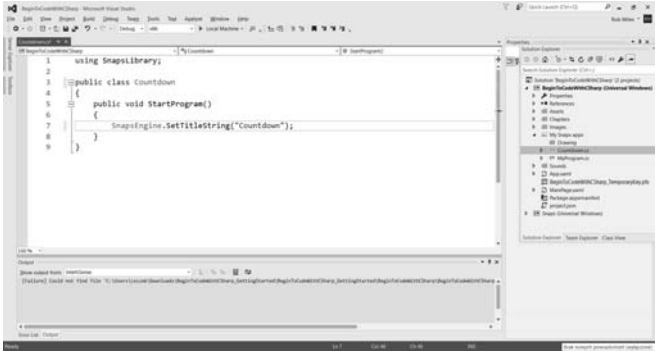
Następną rzeczą, jaką musimy zrobić, jest zmiana nazwy klasy, która zawiera nasz program. Kliknij plik *Countdown.cs* w oknie *Solution Explorer*, aby jego kod pojawił się w oknie edytora, jak pokazano na rysunku 3.5.



Rysunek 3.5. Otwarcie pliku Countdown.cs w edytorze programu Visual Studio

Patrząc na rysunek 3.5 można zauważyć, że Visual Studio próbuje nam coś powiedzieć. Falowane czerwone linie wskazują, że Visual Studio uznaje pewne elementy kodu programu za błędne. W tym przypadku środowisko jest niezadowolone, ponieważ nasze rozwiązanie *BeginToCode-WithCSharp* zawiera dwie wersje klasy `MyProgram` — oryginał w pliku *MyProgram.cs* i kolejną wersję w pliku *Countdown.cs*. Możemy rozwiązać ten problem przez nadanie klasie nowej nazwy.

Na rysunku 3.6 widać, że zmieniłem nazwę klasy na `Countdown`, a ponadto zmieniłem działanie programu, modyfikując jedną instrukcję (tę, która wywołuje `SetTitleString`) i usuwając drugą (wywołującą `DisplayString`). Teraz program ustawia po prostu swój tytuł na *Countdown*. Oczywiście jako tytułu możesz użyć dowolnego łańcucha znaków, ale upewnij się, że z obu stron będzie on ograniczony znakami podwójnego cudzysłowu; w przeciwnym razie program się nie skompiluje.



Rysunek 3.6. Definiowanie klasy Countdown

Visual Studio jest już zadowolone, ponieważ usunęliśmy duplikat klasy `MyProgram`. Teraz powinieneś być w stanie uruchomić program za pomocą przycisku *Start Debugging* (zielonej strzałki).

ANALIZA KODU

Nazwy plików i nazwy klas

Rozwiązanie C# można rozdzielić na dużą liczbę osobnych plików programów. Warto zastanowić się, jak to działa.

Pytanie: Dlaczego musimy zmienić nazwę klasy, skoro zmieniliśmy już nazwę pliku?

> **Odpowiedź:** Aby odpowiedzieć na to pytanie, musisz zrozumieć różnicę między *logicznymi* i *fizycznymi* nazwami w programie. Nazwy plików przechowujących programy możesz potraktować jak nazwy fizyczne, ponieważ nazwa pliku jest powiązana z rzeczywistym plikiem, który jest przechowywany na komputerze. Jednak nazwy elementów w programie nie są powiązane z fizycznym plikiem zawierającym tekst programu. Istnieją w „logicznej" przestrzeni nazw, która jest definiowana przez programistę.
>
> Kiedy kompilator C# kompiluje program, odczytuje wszystkie pliki źródłowe i tworzy listę wszystkich elementów zdefiniowanych w programie. To jest logiczna przestrzeń nazw programu. Każdy z elementów w tej logicznej przestrzeni nazw musi mieć unikatową nazwę. Jeśli utworzymy dwa elementy o tej samej nazwie, kompilator będzie narzekał, i właśnie to wydarzyło się wcześniej, kiedy skopiowaliśmy plik *MyProgram.cs*. Po wykonaniu kopii były dwie klasy o nazwie `MyProgram`. Rozwiązaliśmy ten problem, zmieniając nazwę jednego z elementów i nadając mu nową, unikatową nazwę.

Pytanie: Czy nazwa pliku źródłowego programu (nazwa fizyczna) i nazwa klasy (nazwa logiczna) w tym pliku źródłowym muszą być zgodne?

> **Odpowiedź:** Nie. Często wygodnie jest dopasować te dwie nazwy, ponieważ może to ułatwić znajdowanie konkretnych elementów, ale kompilator języka C# tego nie wymusza.

Pytanie: Co by się stało, gdyby program zawierał już klasę o nazwie `Countdown` i dodalibyśmy kolejną?

> **Odpowiedź:** Prawdopodobnie wiesz, co by się stało. Kompilator zgłosiłby błąd, ponieważ nie lubi mieć dwóch elementów o tej samej nazwie.

Tak na marginesie, czy spodziewałeś się może, że aplikacja *Countdown* zostanie uruchomiona natychmiast po kliknięciu przycisku *Start Debugging*? Gdy aplikacja *BeginToCodeWithCSharp* jest uruchamiana po raz pierwszy, środowisko Snaps szuka klasy o nazwie `MyProgram`, a następnie wywołuje metodę `StartProgram` w tej klasie. Oznacza to, że przy każdym uruchomieniu aplikacja *BeginToCodeWithCSharp* uruchamia najpierw oryginalny program: *MyProgram.cs*. Dzięki temu możesz używać list *Folder* i *Aplikacje Snaps*, aby wybierać inne aplikacje, które chcesz uruchomić.

Ten proces kopiowania, wklejania i poprawiania możesz powtarzać za każdym razem, gdy będziesz chciał utworzyć nową aplikację i dodać ją do naszego środowiska Snaps. Ponieważ jednak wiesz już, że *MyProgram* to aplikacja, która jest uruchamiana automatycznie, gdy

uruchamiane jest środowisko Snaps, oto wskazówka, która może Ci to nieco ułatwić: zacznij od edycji zawartości pliku *MyProgram.cs*. Dzięki temu utworzony kod będzie uruchamiany bez konieczności wyszukiwania i wybierania tej nowej aplikacji w środowisku (tak jak musieliśmy zrobić, aby uruchomić aplikację *Countdown*).

Pamiętaj, że dopóki klasa w pliku *MyProgram.cs* nazywa się `MyProgram`, to właśnie ten program będzie uruchamiany w środowisku jako pierwszy. Gdy skończysz budować swoją nową aplikację w pliku *MyProgram.cs*, możesz skopiować i wkleić kod programu do nowego pliku źródłowego (nowego pliku *.cs*), nadać nowemu plikowi źródłowemu unikatową nazwę i zmienić nazwę nowej klasy programu, aby Visual Studio nie podkreślało kodu na czerwono i nie uniemożliwiało kompilacji programu. A na tym etapie, jeśli naprawdę będziesz chciał, aby aplikacja *MyProgram* działała tak, jak dotychczas (czyli ustawiała ten sam tytuł i wyświetlała tę samą wiadomość, które widzieliśmy w poprzednich rozdziałach), wiesz, jak wrócić do tego stanu.

Czy oczywiste jest już, dlaczego nazwałem ten plik źródłowy *MyProgram.cs*? Możesz go swobodnie wykorzystywać do budowania wielu programów!

ZRÓB TO SAM

Budowanie spikera odliczającego

To „Niech się dzieje" jest dość doniosłe. Stanowi bardzo ważny kamień milowy na Twojej drodze do oświecenia programistycznego. Do tej pory modyfikowałeś lub naprawiałeś istniejące programy, co jest świetnym sposobem na rozpoczęcie nauki programowania, ale w pewnym momencie będziesz musiał utworzyć własny program od podstaw. Ten czas właśnie nadszedł. Jeśli się nad tym zastanowić, nawet Bill Gates musiał od czegoś zacząć. Jestem jednak pewien, że jego pierwszy program nie był w stanie rozmawiać ze swoimi użytkownikami. Zmuszanie komputerów do mówienia było bardzo trudne w tamtych czasach, kiedy Bill Gates uczył się pisać kod, ale zapewne czuł tę samą ekscytację, którą Ty za chwilę poczujesz.

Po zbudowaniu tej aplikacji będziesz miał na koncie swój pierwszy program. Możesz uczynić go bardziej osobistym, korzystając z dowolnych komunikatów, a w następnym podrozdziale odkryjesz kolejne Snapy, których możesz użyć, aby uczynić swój program jeszcze bardziej interesującym.

Powinieneś już mieć „pustą" aplikację o nazwie *Countdown*. W tej chwili nie robi ona prawie nic — w stanie, w jakim ją ostatnio widziałeś, ustawia jedynie tytułowy łańcuch znaków — ale teraz napiszesz własne instrukcje, aby tchnąć w nią życie. Aby utworzyć swój program, możesz użyć metod `SpeakString`, `DisplayString` i `SetTitleString`, dostarczonych przez klasę `SnapsEngine`.

Musisz jedynie napisać program, który odlicza od 10 do 0. Wskazówka: Twój program będzie zawierać co najmniej 10 instrukcji. Rozbuduj tę podstawową wersję programu w taki sposób, aby podczas odliczania komputer wyświetlał jednocześnie liczby na ekranie. To powinno podwoić liczbę instrukcji w Twoim programie.

Błędy kompilacji

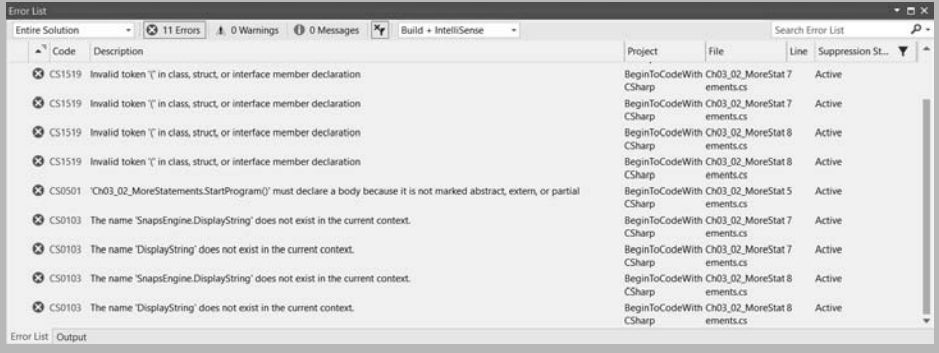
Zanim program będzie mógł zostać uruchomiony, musi najpierw zostać sprawdzony przez kompilator. Możesz potraktować ten proces trochę jak kontrole techniczne samolotów wykonywane przed lotem. Najpierw kapitan musi zrobić obchód samolotu, policzyć skrzydła, sprawdzić, czy we wszystkich oponach jest powietrze i upewnić się, że samolotem można bezpiecznie lecieć. Analogicznie kompilator przeprowadza wstępne sprawdzanie programu zanim będzie mógł go uruchomić. Jeśli program nie przestrzega reguł języka C#, kompilator wygeneruje błędy, które musisz naprawić Ty, jako programista.

Niestety, kompilator jest dużo bardziej drobiazgowy w kwestii błędów niż ludzie. Mogę na przykład podejść do kogoś i zapytać: „Co Ty robić?". Otrzymam odpowiedź, chociaż pytanie, które zadałem, jest sformułowane nieprawidłowo po polsku. Jeśli jednak spróbuję skompilować poniższy program, otrzymam błędy:

```
using SnapsLibrary;

public class BadBrackets
{
    public void StartProgram()
    (
        SnapsEngine.SpeakString("Witaj, świecie");
        SnapsEngine.SpeakString("Żegnajcie, kurczaki");
    )
}
```

Ten kod wygląda bardzo podobnie do znanego nam programu, ale istnieją dwa drobne błędy w tekście. Zła wiadomość jest taka, że generują one 11 bardzo mylących komunikatów o błędzie, jak pokazano na rysunku.



Najtrudniejsze w tym stanie rzeczy jest to, że żaden z tych komunikatów tak w rzeczywistości nie mówi Ci, co zrobiłeś źle (a niektóre z nich wyglądają naprawdę strasznie). Kompilator to bardzo sprytny program, ale nie jest wystarczająco mądry, żeby powiedzieć: „Użyłeś nawiasów okrągłych tam, gdzie

powinieneś był użyć nawiasów klamrowych". Zaktualizuj ten kod, aby instrukcje były poprzedzone otwierającym nawiasem klamrowym ({) i zakończone zamykającym nawiasem klamrowym (}), a program zostanie uruchomiony. Kiedy oznaczasz początek i koniec określonych części programu, zawsze musisz używać nawiasów klamrowych. Nawiasy okrągłe są używane do czegoś innego.

Z PUNKTU WIDZENIA PROGRAMISTY

Dobry programista musi być w stanie radzić sobie ze szczegółami

Ludzie niesamowicie dobrze radzą sobie z hałasem. Potrafimy wyłowić nasze imię z dobiegającej nas z oddali rozmowy i rozpoznać swoją matkę w morzu twarzy. Programy komputerowe muszą bardzo ciężko pracować, aby wydobyć znaczenie z danych. Jeden drobny, niepoprawny szczegół w programie może zdezorientować kompilator. Oznacza to, że aby zostać świetnym programistą, trzeba nauczyć się bardzo szczegółowo badać różne rzeczy, w niektórych przypadkach znak po znaku, żeby odkryć, co jest z nimi nie tak.

Najlepszym sposobem radzenia sobie z takimi błędami jest oczywiście wystrzeganie się ich. Ponieważ jednak jesteśmy ludźmi, jest to niemożliwe. Oto moje wskazówki dotyczące radzenia sobie z błędami kompilacji:

1. Zacznij od programu, który *kompiluje się* lub pomyślnie uruchamia. (Pamiętaj, że kompilatory pobierają kod wysokiego poziomu i generują kod maszynowy, który umożliwia komputerowi wykonanie żądanych przez nas działań. Dlatego mówimy, że program, który uruchamia się pomyślnie bez błędów, kompiluje się). Visual Studio udostępnia kreatory oprogramowania umożliwiające tworzenie programów, które nie robią zbyt wiele, ale się kompilują.
2. Często kompiluj (w programie Visual Studio za pomocą przycisku *Start Debugging*). Jeśli liczba zmian wykonanych od czasu ostatniej udanej kompilacji jest niewielka, możesz wyizolować możliwość wystąpienia błędu do kilku miejsc w kodzie.
3. Szukaj trzech klasycznych błędów kompilacji:
   a. Czegoś brakuje — nie umieszczono na przykład średnika na końcu instrukcji.
   b. Użyto niewłaściwego znaku — użyto na przykład znaku `[` zamiast `{`.
   c. Zastosowano niepoprawną pisownię — napisano na przykład „startProgram" zamiast „StartProgram". W świecie języka C# ma znaczenie, czy używasz wielkich, czy małych liter.
4. Nie oczekuj, że błąd występuje tam, gdzie wykrył go kompilator. Niektóre błędy (na przykład brakujący nawias klamrowy) mogą zostać wykryte wiele linii dalej od miejsca, w którym faktycznie zostały popełnione.
5. Użyj do pomocy mechanizmu zaznaczania tekstu kolorami. Słowa będące częścią języka C# są wyświetlane na niebiesko. Łańcuchy znaków są zaznaczane na czerwono. Jeśli jakieś słowo nie jest zaznaczone kolorem, którym Twoim zdaniem powinno być zaznaczone, być może wpisałeś je niepoprawnie.
6. Napraw wszystkie błędy, które jesteś w stanie dostrzec, a następnie ponownie skompiluj kod. Czasami kompilator staje się zdezorientowany i zgłasza błędy w liniach, które mają sens. Po naprawieniu wszystkich dostrzegalnych błędów, skompiluj program ponownie i sprawdź, czy działa.

7. Używaj poleceń *Cofnij* i *Ponów*. Visual Studio zawiera bardzo potężny edytor z przyciskami *Cofnij* (*Ctrl+Z*) i *Ponów* (*Ctrl+Y*), których możesz użyć do nawigowania wstecz i do przodu poprzez zmiany wprowadzone w kodzie. Aby dowiedzieć się, gdzie są błędy, możesz skorzystać z tych poleceń i falowanych czerwonych linii, które Visual Studio stosuje do podświetlania błędów.

ANALIZA KODU

Znajdowanie błędów kompilacji

Ten program generuje podczas kompilacji 20 komunikatów o błędzie. Sprawdź, czy potrafisz znaleźć wszystkie błędy.

```
using SnapsLibrary;


public Class MyProgram
{
    public void StartProgram()
    {
        SnapsEngine.SetTitleString("Rozpoczynamy kodowanie z C#");
        SnapsEngine.DisplayString(Witaj w świecie aplikacji Snaps");
    }
}
```

Oto one:

```
using SnapsLibrary;

public Class MyProgram
{
    public void StartProgram()
    {
        SnapsEngine.SetTitleString("Rozpoczynamy kodowanie z C#");


        SnapsEngine.DisplayString(Witaj w świecie aplikacji Snaps");
    }
}
```

Brak podwójnego cudzysłowu przez słowem Witaj.

Słowo class powinno zaczynać się od małej litery.

Jeśli naprawisz te dwa błędy, program pomyślnie się skompiluje.

Dodatkowe Snapy

Pod koniec wybranych rozdziałów będę przedstawiał dodatkowe Snapy, którymi możesz się pobawić. Możesz używać ich w swoich programach w taki sposób, w jaki wcześniej używałeś Snapa `SpeakString`.

Delay

Być może będziesz chciał wstrzymać działanie programu na jakiś czas za pomocą Snapa `Delay`:

```
using SnapsLibrary;

class Ch03_06_TenSecondTimer
{
    public void StartProgram()
    {
        SnapsEngine.DisplayString("Start");
        SnapsEngine.Delay(10);
        SnapsEngine.DisplayString("Koniec");
    }
}
```

Wstrzymanie na 10 sekund.

Ten program wyświetla komunikat *Start*, wstrzymuje działanie na 10 sekund, a następnie wyświetla komunikat *Koniec*. Metoda `Delay` różni się od metody `DisplayString` pod względem dostarczanych do niej danych. Metodzie `DisplayString` przekazujemy łańcuch znaków, który program ma wyświetlić. Natomiast metodzie `Delay` przekazujemy liczbę sekund, przez którą program ma pauzować. Może to być liczba ułamkowa, jeśli chcesz, aby program zatrzymał się na mniej niż sekundę:

```
SnapsEngine.SpeakString("Tik");
SnapsEngine.Delay(0.5);
SnapsEngine.SpeakString("Tak");
```

Wstrzymuje program na pół sekundy.

Metody `Delay` możesz użyć na przykład po to, aby program wyglądał jakby się nad czymś „zastanawiał", albo żeby dać użytkownikowi czas na przeczytanie informacji wyświetlonych na ekranie.

SetTextColor

Ten Snap pozwala ustawić kolor tekstu w wiadomości, która ma być wyświetlona na ekranie:

```
using SnapsLibrary;

class Ch03_07_BlueText
{
    public void StartProgram()
    {
        SnapsEngine.SetTextColor(SnapsColor.Blue);
        SnapsEngine.DisplayString("Niebieski poniedziałek");
    }
}
```

Wbudowany kolor biblioteki Snaps, który reprezentuje niebieski.

Możesz również wywołać tę metodę, aby zmienić kolor tekstu już wyświetlonego na ekranie.

```
using SnapsLibrary;

class Ch03_08_DelayedBlueText
{
    public void StartProgram()
    {
        SnapsEngine.DisplayString("Niebieski poniedziałek");
        SnapsEngine.Delay(2);
        SnapsEngine.SetTextColor(SnapsColor.Blue);
    }
}
```

Ten program najpierw wyświetla wiadomość *Niebieski poniedziałek* w domyślnym kolorze. Po dwóch sekundach zmienia kolor tekstu na niebieski.

SetTitleColor

Ten Snap pozwala ustawić kolor tekstu w wiadomości tytułowej na ekranie:

```
using SnapsLibrary;

class Ch03_09_GreenSystemStarting
{
    public void StartProgram()
    {
        SnapsEngine.SetTitleColor(SnapsColor.Green);
```

```
        SnapsEngine.SetTitleString("Uruchamianie systemu");
    }
}
```

Ten program ustawia tekst tytułu na zielony, a następnie wyświetla komunikat *Uruchamianie systemu* jako tytuł strony. Ogólnie rzecz biorąc, najlepiej jest ustawiać kolor tytułów i wiadomości przed ich wyświetleniem; w przeciwnym razie w momencie wyświetlania będą „migotać", zmieniając kolor na żądany. Aby zobaczyć, co mam na myśli, odwróć kolejność instrukcji w programie *Ch03_09_GreenSystemStarting.cs*. Efekt ten został zminimalizowany w programie *Ch03_08_DelayedBlueText.cs* w wyniku zastosowanego opóźnienia.

SetBackgroundColor

Ten Snap pozwala ustawić kolor tła ekranu. Możesz użyć go do wskazania alarmów lub innych stanów programu.

```
using SnapsLibrary;

class Ch03_10_RedScreen
{
    public void StartProgram()
    {
        SnapsEngine.SetBackgroundColor(SnapsColor.Red);
    }
}
```

Tworzenie własnych kolorów

Biblioteka Snaps zawiera wiele wbudowanych kolorów, które możesz wykorzystać w swoich programach. Możesz zobaczyć te wartości `SnapsColor` w przykładach, którym się przyglądaliśmy: `SnapsColor.Blue`, `SnapsColor.Green` i `SnapsColor.Red`. Możesz jednak chcieć używać kolorów, których nie ma w bibliotece. Ja na przykład lubię kolor liliowy. Opisując kolor komputerowi musisz użyć liczb, ponieważ — jak wiemy — komputery tak naprawdę działają tylko z wartościami liczbowymi. Aby opisać konkretny kolor, możemy użyć trzech wartości: nasycenia czerwieni, nasycenia zieleni i nasycenia niebieskiego w tym kolorze. W przypadku biblioteki Snaps (i wielu innych platform komputerowych, w tym systemu Windows), każda z liczb opisujących poziom nasycenia koloru mieści się w zakresie od 0 do 255.

Poziomów nasycenia barw czerwonej, zielonej i niebieskiej w poszczególnych kolorach możesz poszukać w internecie. Okazuje się, że liliowy składa się z następujących wartości: 200 czerwonego, 162 zielonego i 200 niebieskiego. Oto sposób użycia tych wartości w Snapach związanych z kolorami:

```
using SnapsLibrary;

class Ch03_11_LilacScreen
{
    public void StartProgram()
    {
        SnapsEngine.SetBackgroundColor(red:200,green:162,blue:200);
    }
}
```

Nasycenie barw czerwonej, zielonej i niebieskiej, pozwalające uzyskać kolor liliowy.

Metodzie `SetBackbroundColor` można dostarczyć do działania jeden argument lub trzy. Może ona otrzymać jedną wartość `SnapsColor` lub wartości czerwonego, zielonego i niebieskiego. Każda z wartości nasycenia koloru jest identyfikowana przez nazwę, co ułatwia programistom określenie, które wartości są używane do określonych celów.

Po zaprojektowaniu metody programista musi zdecydować, ile informacji potrzebuje ta metoda, aby wykonywać swoją pracę i jaką formę powinny przyjmować te informacje. W przypadku `SetBackgroundColor` ta wersja metody musi otrzymać informacje o nasyceniu barw czerwonej, zielonej i niebieskiej. Elementy dostarczane do metody są podawane w formie listy, na której poszczególne pozycje są rozdzielone przecinkami. Jeśli pominiesz jakiś element lub podasz zbyt wiele elementów, kompilator zgłosi błąd podczas tworzenia programu.

```
SnapsEngine.SetBackgroundColor(red:255,green:255);
```

```
Error 1 No overload for method 'SetBackgroundColor' takes 2 arguments
```

Kompilatorowi nie podoba się ta instrukcja, ponieważ metoda `SetBackgroundColor` z biblioteki Snaps nie została utworzona (przeze mnie) w taki sposób, aby akceptować tylko dwa elementy.

CO MOGŁO PÓJŚĆ NIE TAK?

Złe schematy kolorów

Jeżeli napiszesz program, który wyświetla czerwony tekst na czerwonym tle, nie otrzymasz żadnych błędów, ale co powiedzą użytkownicy Twojego programu? Osobiście lubię używać domyślnych kolorów (czyli tych, które otrzymujesz po uruchomieniu programu). Jeśli chcesz pokazać swoją kreatywność, możesz wybrać inne kolory, ale powinieneś przetestować swój schemat kolorów na wielu różnych

urządzeniach, ponieważ niektóre maszyny mogą wyświetlać kolory znacznie lepiej niż inne. Powinieneś także uzgodnić proponowany schemat kolorów z klientem, jeśli go posiadasz, ponieważ kolory są jedną z rzeczy, o których klienci mają wyrobione zdanie. Ponadto różni ludzie z odmienną intensywnością dostrzegają poszczególne kolory. Nie zakładaj, że inni widzą kolory w taki sam sposób, jak Ty!

ZRÓB TO SAM

Budowanie minutnika

Możesz teraz wykorzystać swoje umiejętności programistyczne, aby utworzyć program do mierzenia czasu gotowania jajek. Używając metody `Delay` z biblioteki Snaps możesz wstrzymać działanie program na czas gotowania jajka, a następnie poinformować, że jajko jest gotowe. Moje testy wskazują, że w celu otrzymania idealnego jajka, należy gotować je przez pięć minut (czyli 300 sekund). Ten kod służy jako dobry punkt wyjścia — żeby zrobić minutnik, skopiuj go zamiast kopiować lub edytować plik *MyProgram.cs* (zamieszczony tu listing przedstawia tylko fragment niezbędnego kodu):

```
using SnapsLibrary;

class Ch03_12_EggTimerStart
{
    public void StartProgram()
    {
        SnapsEngine.SetTitleString("Minutnik");
        SnapsEngine.DisplayString("Pozostało 5 minut");
        SnapsEngine.Delay(60);
        SnapsEngine.DisplayString("Pozostały 4 minuty");
    }
}
```

Myślę, że to kolejny ważny kamień milowy dla Ciebie jako programisty. W przeciwieństwie do programu odliczającego, który utworzyłeś wcześniej, ten program ma wszystkie zadatki na dobry produkt. Twoja mama uzna ten program za przydatny. Sklep Windows ma kilka produktów, które działają jak timery i nie ma powodu, dla którego timer zrobiony przez Ciebie nie może stać się jednym z nich.

Możesz wprowadzić do timera dodatkowe funkcje, aby robił takie rzeczy, jak zmiana koloru ekranu, gdy jajko jest prawie gotowe, a nawet zapewniał 30-sekundowe ostrzeżenie przed upływem czasu — a pod koniec może nawet uruchamiać się odliczanie w stylu „dziesięć, dziewięć, osiem,...". Możesz także zrobić timer, który mówi i wyświetla ile czasu pozostało.

Ten projekt możesz również wykorzystać do tworzenia timerów, które mogą być przydatne w wielu innych sytuacjach. Oto cztery, jakie przychodzą mi do głowy:

- Twój najlepszy przyjaciel odkrył pasję do wywoływania własnych zdjęć i potrzebuje czasomierza, którego może używać w ciemni. Ten timer powinien po prostu co 5 sekund informować, ile czasu upłynęło.
- Wraz ze współpracownikami założyliście klub quizowy i chcecie kontrolować, ile czasu będzie miał każdy zespół na udzielanie odpowiedzi. Każda drużyna dostaje 10 sekund.
- Twój brat ma grę, w której każdy gracz musi użyć wykałaczki, aby w 30 sekund zjeść jak najwięcej ziaren fasolki po bretońsku (nie powiedziałem, że ta gra ma sens) i potrzebuje do tego timera.
- Twoja mama zaczęła ćwiczyć i potrzebuje czegoś do odmierzania czasu poszczególnych etapów treningu i informowania, jakie jest kolejne ćwiczenie. Ma do wykonania pięć ćwiczeń: bieg w miejscu, pompki, pajacyki, przysiady i krokodylki. Każde ćwiczenie powinno być wykonywane przez 30 sekund, po czym ma następować 10-sekundowy odpoczynek.

Spróbuj swoich sił w tworzeniu tych i innych czasomierzy, jakie przychodzą Ci do głowy. W następnym rozdziale dowiesz się, w jaki sposób program może uzyskać dane wejściowe od użytkownika, abyś mógł opracować jeszcze lepsze timery, które pozwalają użytkownikowi ustawiać czas.

Czego się nauczyłeś?

W tym rozdziale poznałeś lepiej środowisko Visual Studio, w którym tworzysz swoje programy. Dowiedziałeś się, że program napisany w języku C# jest wyrażany jako sekwencja instrukcji wykonywanych po uruchomieniu programu. Zobaczyłeś również, w jaki sposób pisany przeze mnie i przez Ciebie wysokopoziomowy kod C# jest kompilowany na niskopoziomowe instrukcje komputerowe przez program zwany kompilatorem. Czasami kod się kompilował i program uruchamiał się pomyślnie, a czasami kod nie kompilował się z powodu błędów.

Przekonałeś się, że kompilator zapewnia zgodność programu z regułami języka C#. Kompilator odrzuca programy, które nie zawierają poprawnych instrukcji. O ile człowiek toleruje brakującą lub niepoprawną interpunkcję w czytanym tekście, to kompilator odrzuci wszystko, co nie jest zgodne z zasadami języka programowania.

Programy, które do tej pory napisaliśmy, wykorzystują zestaw Snapów dostarczanych przez bibliotekę Snaps, umożliwiającą robienie takich rzeczy, jak wypowiadanie komunikatów, wyświetlanie kolorów lub czasowe wstrzymywanie wykonywania programu. Te komponenty są dostarczane jako metody, do których przekazywane są dane, wskazujące im, co mają robić. Do metody `SpeakString` przekazywany jest na przykład łańcuch znaków, który ma być wypowiadany przez komputer.

Oto kilka pytań dotyczących programów, instrukcji i kompilatorów, nad którymi warto się zastanowić.

Czy użytkownik programu musi mieć kopię Visual Studio, aby uruchomić ten program?

Nie. Visual Studio pozwala wygenerować plik programu, który użytkownicy będą mogli uruchamiać bez Visual Studio.

Czy muszę wiedzieć, jak działa każde polecenie Visual Studio?

Nie. Na początek wystarczy Ci znajomość zaledwie kilku przycisków. Kolejne funkcjonalności będziesz odkrywał w trakcie lektury książki.

Czy kompilator jest niekompetentny, ponieważ dezorientuje go nieprawidłowy kod programu?

Możesz pomyśleć, że kompilator jest trochę głupiutki, ponieważ czasami narzeka, gdy zobaczy niewłaściwy znak. Być może zachodzisz w głowę, dlaczego kompilator nie podstawi po prostu właściwego znaku i nie pójdzie dalej. Jak się jednak okazuje, istnieje bardzo dobry powód, dla którego kompilator tego nie robi. Gdyby kompilator wstawiał brakujące według niego rzeczy, przyjmowałby założenie dotyczące tego, co Ty jako programista próbowałeś osiągnąć. Przekonaliśmy się już, że takie założenia bywają niebezpieczne. Kompilator słusznie uznaje, że znacznie bezpieczniej jest nalegać, abyś wyrażał dokładnie i poprawnie, czego wymagasz od programu.

Czy dowolna metoda C# może przyjmować dowolną liczbę argumentów, które wskazują jej, co ma zrobić?

Nie. Każda metoda jest indywidualnie zaprogramowana do przyjmowania określonego zestawu informacji. Metoda `Delay` musi zostać poinformowana o tym, na ile sekund ma wstrzymać działanie programu. Metoda `SpeakString` potrzebuje łańcuchów znaków, które mają być wypowiadane przez komputer. Kompilator wie, jaki zestaw informacji ma akceptować dana metoda i tylko takie dane jej przekaże. Jeśli spróbujesz podać łańcuch znaków metodzie `Delay`, program się nie skompiluje.

Czy instrukcje w programie są zawsze wykonywane w tej kolejności, w jakiej zostały zapisane?

Tak. Program możesz potraktować jak opowieść, recepturę lub sekwencję instrukcji. Wykonywanie kroków w innej kolejności niż zostało to określone nie miałoby sensu.

Czy biblioteka Snaps jest częścią języka C#?

Nie. Biblioteka Snaps wraz z jej metodami została przygotowana, żeby pomóc Ci uczyć się programowania i tworzenia prostych aplikacji. Te metody nie są częścią języka C#, ale zostały napisane za jego pomocą. Inne biblioteki klas i metod dostarczane z językiem C# poznasz nieco dalej w tej książce.

C#. Bierz to, co najlepsze!

Pierwsze kroki w programowaniu bywają frustrujące. Wszystko wydaje się jasne i proste, ale tylko do pierwszych prób skompilowania kodu. Półki księgarń uginają się pod ciężarem podręczników programowania w różnych językach, jednak większość nie ułatwia nauki podstaw, które są absolutnie niezbędne każdemu koderowi. Spośród licznych języków programowania C# jest wart szczególnej uwagi. To dojrzały język, wyjątkowo wszechstronny i prosty w stosowaniu, dzięki czemu jest ulubionym narzędziem profesjonalistów. Ponadto z uwagi na inne właściwości stanowi idealny wybór dla początkujących koderów, gdyż ułatwia kształtowanie dobrych nawyków.

To książka dla każdego, kto chce osiągnąć programistyczne oświecenie! Jest napisana w innowacyjny sposób, przyjaznym i zrozumiałym językiem, aby każdy adept programowania zdobył wiedzę i umiejętności potrzebne do pisania dobrego, użytecznego kodu. Dzięki niej nauczysz się myśleć jak zawodowy programista. Poza przydatnymi informacjami znajdziesz tu szereg praktycznych ćwiczeń i przykładów kodu. Nie zabrakło również bardzo istotnych wskazówek dotyczących potencjalnych problemów i technik ich rozwiązywania. Tam, gdzie to potrzebne, dokładnie opisano, które elementy mogą zawieść i na co należy zwrócić uwagę. Podręcznik został przygotowany tak, aby programowanie było przyjemne, satysfakcjonujące i wyzwalało pełnię kreatywności!

Dzięki tej książce:

- przygotujesz wygodne i bezpłatne środowisko pracy
- dowiesz się, na czym dokładnie polega działanie kodu
- zdobędziesz solidne podstawy programowania i nauczysz się patrzeć z perspektywy programisty
- zrozumiesz i przyswoisz kluczowe pojęcia, takie jak klasy, interfejsy i metody
- będziesz tworzyć nowoczesne i wciągające gry
- nauczysz się technik testowania i debugowania kodu

Rob Miles

naucza programowania od ponad 25 lat. Wykłada programowanie w C# i inżynierię oprogramowania na Uniwersytecie w Hull. Zdobył tytuł MVP i regularnie zasiada w jury konkursu Microsoftu Imagine Cup. Chętnie dzieli się wiedzą i lubi inspirować inżynierów. Występuje na różnych konferencjach dla programistów i publikuje książki o kodowaniu. Hobbystycznie programuje i pisze... wiersze.

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

ISBN 978-83-283-6060-0

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 89,00 zł

Microsoft Press