C#. Ćwiczenia

Autor: Marcin Lis
ISBN: 83-7361-128-2
Format: B5, stron: 166

Język C# został opracowany w firmie Microsoft i wywodzi się z rodziny C/C++, choć zawiera również wiele elementów znanych programistom Javy, jak na przykład mechanizmy automatycznego odzyskiwanie pamięci. Łączy więc w sobie wszystko to, co najlepsze w Javie i C++, a dodatkowo pozwala na wygodne korzystanie z klas wchodzących w skład platformy .NET.

Dzięki książce „C#. Ćwiczenia" nauczysz się programować w C# niezależnie od tego, czy znasz już C++ lub Javę. Kilkadziesiąt ćwiczeń pozwoli Ci poznać język C# od podstaw po zagadnienia zaawansowane. Zaletą książki jest zwięzły i przystępny opis prezentowanych zagadnień i nastawienie na praktykę programistyczną, a nie na rozważania teoretyczne.

Poznasz:

- Środowisko uruchomieniowe C# i Visual Studio
- Zmienne, operatory i typy danych
- Instrukcje C#
- Podstawy programowania obiektowego, tworzenie klas, metod i konstruktorów
- Użycie tablic w C#
- Obsługę błędów za pomocą wyjątków
- Interfejsy i rzutowanie
- Obsługę zdarzeń
- Korzystanie z komponentów interfejsu użytkownika Windows

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

Spis treści

Rozdział 3.

Instrukcje

Instrukcje warunkowe

Instrukcja if...else

Bardzo często w programie zachodzi potrzeba sprawdzenia jakiegoś warunku i, w zależności od tego, czy jest on prawdziwy czy fałszywy, dalsze wykonywanie różnych instrukcji. Do takiego sprawdzania służy właśnie instrukcja warunkowa `if…else`. Ma ona ogólną postać:

```
if (wyrażenie warunkowe){
  Instrukcje do wykonania, jeżeli warunek jest prawdziwy
}
else{
  Instrukcje do wykonania, jeżeli warunek jest fałszywy
}
```

Spróbujmy zatem wykorzystać taką instrukcję do sprawdzenia, czy zmienna całkowita jest mniejsza od zera.

Ćwiczenie 3.1.

Wykorzystaj instrukcję warunkową if... else do stwierdzenia, czy wartość zmiennej arytmetycznej jest mniejsza od zera. Wyświetl odpowiedni komunikat na ekranie.

```
using System;

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

```
    {
        int zmienna = -5;
        if (zmienna < 0){
            Console.WriteLine("Zmienna jest mniejsza od zera.");
        }
        else{
            Console.WriteLine("Zmienna nie jest mniejsza od zera.");
        }
    }
}
```

Spróbujmy teraz czegoś nieco bardziej skomplikowanego. Zajmijmy się klasycznym przykładem liczenia pierwiastków równania kwadratowego. Przypomnijmy, że jeśli mamy równanie w postaci:

$$A * x \wedge 2 + B * x + C = 0,$$

aby obliczyć jego rozwiązanie liczymy tzw. deltę (⊿), która równa jest:

$$B \wedge 2 - 4 * A * C.$$

Jeżeli delta jest większa od zera, mamy dwa pierwiastki:

$$x1 = (-B + \sqrt{\Delta})/2 * A$$

$$x2 = (-B - \sqrt{\Delta})/2 * A.$$

Jeżeli delta jest równa zero, istnieje tylko jedno rozwiązanie, a mianowicie:

$$x = -B/2 * A.$$

W przypadku trzecim, jeżeli delta jest mniejsza od zera, równanie takie nie ma rozwiązań w zbiorze liczb rzeczywistych. Skoro jest tutaj tyle warunków do sprawdzenia, jest to doskonały przykład do potrenowania zastosowania instrukcji `if…else`. Przy tym, aby nie komplikować sprawy, nie będziemy się w tej chwili zajmować wczytywaniem parametrów równania z klawiatury, ale podamy je bezpośrednio w kodzie.

Przed przystąpieniem do realizacji tego zadania musimy się tylko jeszcze dowiedzieć, w jaki sposób uzyskać pierwiastek z danej liczby? Na szczęście nie jest to wcale skomplikowane, wystarczy skorzystać z instrukcji `Math.Sqrt()`. Aby zatem dowiedzieć się, jaki jest pierwiastek kwadratowy z liczby 4, należy napisać:

```
Math.Sqrt(4);
```

Oczywiście zamiast liczby możemy też podać w takim wywołaniu zmienną, a wynik działania wypisać na ekranie np.:

```
int pierwszaLiczba = 4;
int drugaLiczba = Math.Sqrt(pierwszaLiczba);
Console.WriteLine(drugaLiczba);
```

Ćwiczenie 3.2.

Wykorzystaj operacje arytmetyczne oraz instrukcje if...else do obliczenia pierwiastków równania kwadratowego o parametrach podanych bezpośrednio w kodzie programu. Wyniki wyświetl na ekranie (rysunek 3.1).

```
using System;

class Pierwiastek
{
  public static void Main(string[] args)
  {
        int parametrA = 1, parametrB = -5, parametrC = 4;

        Console.WriteLine("Parametry równania:\n");
        Console.WriteLine("A: " + parametrA + " B: " + parametrB +
        ➥" C: " + parametrC + "\n");

        if (parametrA == 0){
              Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");
        }
        else{
              double delta = parametrB * parametrB - 4 * parametrA * parametrC;
              if (delta < 0){
                    Console.WriteLine("Delta < 0.");
                    Console.WriteLine("To równanie nie ma rozwiązania
                    ➥w zbiorze liczb rzeczywistych");
              }
              else{
                    double wynik;
                    if (delta == 0){
                          wynik = - parametrB / 2 * parametrA;
                          Console.WriteLine("Rozwiązanie: x = " + wynik);
                    }
                    else{
                          wynik = (- parametrB + Math.Sqrt(delta)) /
                          ➥2 * parametrA;
                          Console.Write("Rozwiązanie: x1 = " + wynik);
                          wynik = (- parametrB - Math.Sqrt(delta)) /
                          ➥2 * parametrA;
                          Console.WriteLine(", x2 = " + wynik);
                    }
              }
        }
  }
}
```

Rysunek 3.1. *Wynik działania programu obliczającego pierwiastki równania kwadratowego*

Instrukcja if...else if

Jak pokazało nam ćwiczenie 3.2, instrukcję warunkową można zagnieżdżać, to znaczy po jednym `if` może występować kolejne, po nim następne itd. Jednakże taka budowa kodu powoduje, że przy wielu zagnieżdżeniach staje się on bardzo nieczytelny. Aby tego uniknąć, możemy wykorzystać instrukcję `if…else if`. Zamiast tworzyć niewygodną konstrukcję, taką jak przedstawiona poniżej:

```
if (warunek1){
  Instrukcje1
}
else{
  if (warunek2){
        instrukcje2
  }
  else{
        if (warunek3){
              instrukcje3
        }
        else{
              instrukcje4
        }
  }
}
```

Całość możemy zapisać dużo prościej i czytelniej w następującej postaci:

```
if (warunek1){
  Instrukcje 1
}
else if (warunek2){
  instrukcje 2
}
else if (warunek3){
  instrukcje 3
}
else{
  instrukcje 4
}
```

Ćwiczenie 3.3.

Zmodyfikuj program napisany w ćwiczeniu 3.2 tak, aby wykorzystywał on instrukcję if... else if.

```
using System;

class Pierwiastek
{
  public static void Main(string[] args)
  {
        int parametrA = 1, parametrB = -5, parametrC = 4;

        Console.WriteLine("Parametry równania:\n");
        Console.WriteLine("A: " + parametrA + " B: " + parametrB +
        ➥" C: " + parametrC + "\n");
        if (parametrA == 0){
              Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");
```

```
        }
        else{
                double delta = parametrB * parametrB - 4 * parametrA * parametrC;
                double wynik;

                if (delta < 0){
                        Console.WriteLine("Delta < 0.");
                        Console.WriteLine("To równanie nie ma rozwiązania
                        ➥w zbiorze liczb rzeczywistych");
                }
                else if (delta == 0){
                        wynik = - parametrB / 2 * parametrA;
                        Console.WriteLine("Rozwiązanie: x = " + wynik);
                }
                else{
                        wynik = (- parametrB + Math.Sqrt(delta)) / 2 * parametrA;
                        Console.Write("Rozwiązanie: x1 = " + wynik);
                        wynik = (- parametrB - Math.Sqrt(delta)) / 2 * parametrA;
                        Console.WriteLine(", x2 = " + wynik);
                }
        }
  }
}
```

Instrukcja switch

Jeżeli mamy do sprawdzenia wiele warunków, instrukcja `switch` pozwoli nam wygodnie zastąpić ciągi instrukcji `if…else if`. Jeśli mamy następujący fragment kodu:

```
if (a == 1){
  instrukcje1
}
else if (a == 2){
  instrukcje2
}
else if (a == 3){
  instrukcje3
}
else{
  instrukcje4
}
```

możemy zastąpić poniższym:

```
switch (a){
  case 1:
        instrukcje1;
        break;
  case 2:
        instrukcje2;
        break;
  case 3:
        instrukcje3;
        break;
  default:
        instrukcje4;
        break;
}
```

Sprawdzamy tu po kolei, czy a nie jest przypadkiem równe jeden, potem dwa i w końcu trzy. Jeżeli tak, wykonywane są instrukcje po odpowiedniej klauzuli case. Jeżeli a nie jest równe żadnej z wymienionych liczb, wykonywane są instrukcje po słowie default. Instrukcja break powoduje wyjście z bloku switch. Czyli, jeśli a będzie równe jeden, zostaną wykonane instrukcje instrukcje1, jeśli a będzie równe dwa, zostaną wykonane instrukcje instrukcje2, jeśli a będzie równe trzy, zostaną wykonane instrukcje instrukcje3. W przypadku gdyby a nie było równe ani jeden, ani dwa, ani trzy, zostaną wykonane instrukcje instrukcje4.

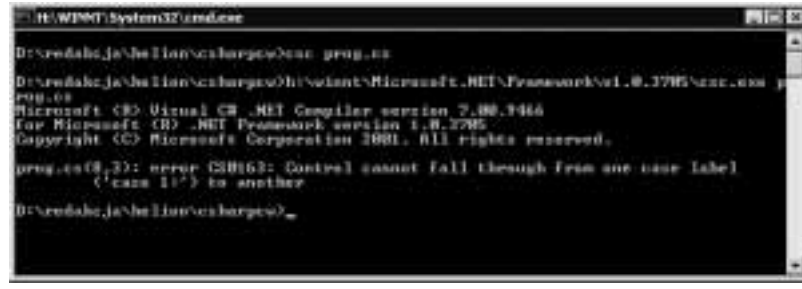
Ćwiczenie 3.4.

Zadeklaruj zmienną typu całkowitego i przypisz jej dowolną wartość. Korzystając z instrukcji switch, sprawdź, czy wartość ta równa jest 1 lub 2. Wyświetl odpowiedni komunikat na ekranie.

```
using System;

class Pierwiastek
{
  public static void Main(string[] args)
  {
        int a = 1;
        switch (a){
              case 1:
                    Console.WriteLine("a = 1");
                    break;
              case 2:
                    Console.WriteLine("a = 2");
                    break;
              default:
                    Console.WriteLine("Zmienna a nie jest równa ani 1, ani 2.");
                    break;
        }
  }
}
```

Warto w tym miejscu zauważyć, że w przypadku C#, odmiennie niż w językach takich jak C, C++ czy Java, nie możemy pominąć instrukcji break, powodującej wyjście z instrukcji switch. Próba taka skończy się błędem kompilacji (rysunek 3.2). Ściślej rzecz biorąc, nie musi być to dokładnie instrukcja break, ale dowolna instrukcja, w wyniku której zostanie opuszczony blok switch. Dokładniejsze wyjaśnienie i przykład takiej konstrukcji znajduje się przy opisie instrukcji goto.



Rysunek 3.2. *Próba pominięcia instrukcji break kończy się błędem kompilacji*

Instrukcja goto

Instrukcja `goto`, czyli instrukcja umożliwiająca skok do określonego miejsca w programie, od dawna jest uznawana za niebezpieczną i powodująca wiele problemów. Niemniej w C# jest ona obecna, choć zaleca się jej stosowanie jedynie w przypadku bloków `switch…case` oraz wychodzenia z zagnieżdżonych pętli, w których to sytuacjach jest faktycznie użyteczna. Używamy jej w postaci

```
goto etykieta;
```

gdzie `etykieta` jest zdefiniowanym miejscem w programie. Nie możemy jednak w ten sposób *wskoczyć* do bloku instrukcji, tzn. nie jest możliwa konstrukcja:

```
for(int j = 0; j < 500; j++){
  if(j == 100) goto label1;
}
for(int i = 0; i < 1000; i++){
  label1:;
}
```

Nic jednak nie stoi na przeszkodzie, aby zdefiniować etykietę przed lub za druga pętlą, pisząc:

```
for(int j = 0; j < 500; j++){
  if(j == 100) goto label1;
}
label1:
for(int i = 0; i < 1000; i++){

}
```

Taki kod jest już jak najbardziej poprawny.

Ćwiczenie 3.5.

Napisz przykładowy program, w którym instrukcja goto jest wykorzystywana do wyjścia z zagnieżdżonej pętli for.

```
using System;

public
class main
{
  public static void Main()
  {
        for(int j = 0; j < 500; j++){
              for(int i = 0; i < 1000; i++){
                    if((j == 100) && (i > 200)){
                          goto label1;
                    }
              }
        }
        return;
        label1:
        Console.WriteLine("Pętla została przerwana!");
  }
}
```

Wykorzystujemy tutaj dwie pętle, zewnętrzną, gdzie zmienną iteracyjną jest `j`, oraz wewnętrzną, gdzie zmienną iteracyjną jest `i`. Wewnątrz drugiej pętli znajduje się warunek, który należy odczytywać następująco: jeżeli `j` równe jest 100 oraz `i` jest większe od 200, idź do miejsca w kodzie oznaczonego etykietą `label1`. Zatem po osiągnięciu warunku na ekranie zostanie wyświetlony napis `Pętla została przerwana` oraz aplikacja zakończy działanie.

Zauważmy, że przed etykietą znajduje się instrukcja `return` powodująca zakończenie działania funkcji `Main`. Gdyby jej nie było, napis o przerwaniu pętli wyświetlany byłby zawsze, niezależnie od spełnienia warunku wewnątrz pętli. Oczywiście w tym przypadku warunek zawsze zostanie spełniony, także `return` nie jest niezbędne, ale już w *prawdziwej* aplikacji zapomnienie o tym drobnym z pozoru fakcie mogłoby przysporzyć nam wielu problemów.

Spróbujmy teraz wykorzystać instrukcję `goto` w drugim z zalecanych przypadków jej użycia, to znaczy w bloku `switch…case`.

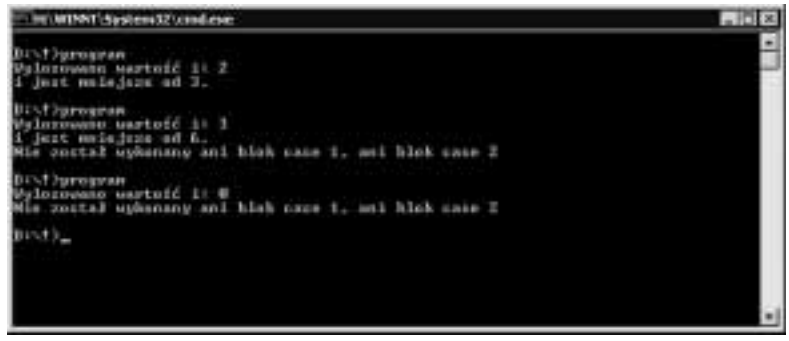
Ćwiczenie 3.6.

Napisz przykładowy kod, w którym instrukcja goto jest wykorzystywana w połączeniu z blokiem switch… case.

```
using System;

public
class main
{
  public static void Main()
  {
        Random r = new Random();
        int i = r.Next(5);
        Console.WriteLine("Wylosowano wartość i: " + i);
        switch(i){
              case 1 :
                    Console.WriteLine("i jest mniejsze od 3.");
                    break;
              case 2 :
                    goto case 1;
              case 3 :
                    Console.WriteLine("i jest mniejsze od 6.");
                    goto default;
              case 4 :
                    goto case 3;
              case 5 :
                    goto case 3;
              default :
                    Console.WriteLine("Nie został wykonany ani blok case 1,
                    ➥ani blok case 2");
                    break;
        }
  }
}
```

Przykładowe wyniki działania tej prostej aplikacji widoczne są na rysunku 3.3. Zmiennej `i` typu `int` przypisujemy losową liczbę całkowitą z zakresu `0 - 5`. Odpowiada za to linia `int i = r.Next(5);`, gdzie `r` jest zmienną wskazującą na obiekt klasy `Random`. Klasa ta służy właśnie do generowania losowych, a dokładniej pseudolosowych, liczb.



Rysunek 3.3. *Przykładowe wyniki działania programu z ćwiczenia 3.6*

Następnie w bloku `switch…case` rozpatrujemy następujące sytuacje:

- `i` jest równe `0` — wyświetlamy napis `Nie został wykonany ani blok case 1, ani blok case 2`
- `i` jest równe `1` lub `2` — wyświetlamy napis `i jest mniejsze od 3.`
- `i` jest równe `3`, `4` lub `5` — wyświetlamy napis `i jest mniejsze od 6.` oraz `Nie został wykonany ani blok case 1, ani blok case 2`

Pętle

Pętle w językach programowania pozwalają na wykonywanie powtarzających się czynności. Dokładnie w ten sam sposób działają one również w C#. Jeśli chcemy, na przykład, wypisać na ekranie 10 razy napis `C#`, to możemy zrobić to, pisząc 10 razy `Console.WriteLine("C#");`. Jeżeli jednak chcielibyśmy mieć, na przykład, 100 takich napisów, to, pomijając oczywiście sensowność takiej czynności, byłby to już problem. Na szczęście z pomocą przychodzą nam właśnie pętle.

Pętla for

Pętla typu `for` ma następująca składnię:

```
for (wyrażenie początkowe; wyrażenie warunkowe; wyrażenie modyfikujące){
  instrukcje do wykonania
}
```

`Wyrażenie początkowe` jest stosowane do zainicjalizowania zmiennej używanej jako licznik ilości wykonań pętli. `Wyrażenie warunkowe` określa warunek, jaki musi być spełniony, aby dokonać kolejnego przejścia w pętli, `wyrażenie modyfikujące` używane jest zwykle do modyfikacji zmiennej będącej licznikiem.

Ćwiczenie 3.7.

Wykorzystując pętlę typu for, napisz program wyświetlający na ekranie 100 razy napis C#.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        for (int i = 1; i <= 100; i++){
              Console.Write("C#");
        }
  }
}
```

Wynik działania tego prostego programu widoczny jest na rysunku 3.4.

Rysunek 3.4.
Wynik działania pętli for z ćwiczenia 3.7

Zmienna `i` to tzw. *zmienna iteracyjna*, której na początku przypisujemy wartość `1` (`int i = 1`). Następnie, w każdym przebiegu pętli jest ona zwiększana o jeden (`i++`) oraz wykonywana jest instrukcja `Console.WriteLine("C# ");` Wszystko trwa tak długo, aż `i` osiągnie wartość 100 (`i <= 100`).

Wyrażenie modyfikujące jest zwykle używane do modyfikacji zmiennej iteracyjnej. Takiej modyfikacji możemy jednak dokonać również wewnątrz pętli. Struktura tego typu wygląda następująco:

```
for (wyrażenie początkowe; wyrażenie warunkowe;){
  instrukcje do wykonania
  wyrażenie modyfikujące
}
```

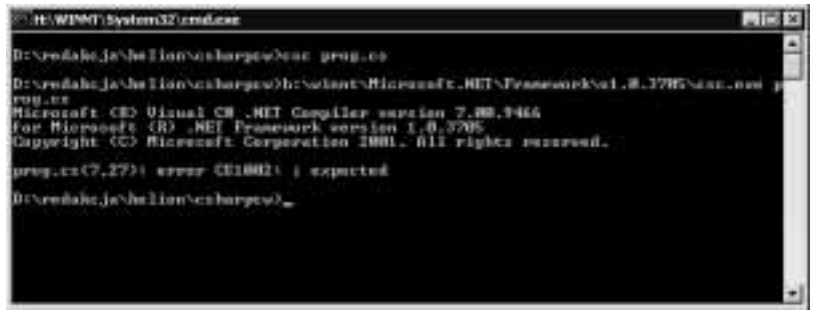
Ćwiczenie 3.8.

Zmodyfikuj pętle typu for z ćwiczenia 3.7 tak, aby wyrażenie modyfikujące znalazło się w bloku instrukcji for.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        for (int i = 1; i <= 100;){
              Console.Write("C# ");
              i++;
         }
  }
}
```

Zwracam uwagę, że mimo iż wyrażenie modyfikujące jest teraz wewnątrz pętli, średnik znajdujący się po `i <= 100` jest niezbędny! Jeśli o nim zapomnimy, kompilator zgłosi błąd (rysunek 3.5).



Rysunek 3.5. *Pominięcie średnika w pętli for powoduje błąd kompilacji*

Kolejna ciekawą możliwością jest połączenie wyrażenia warunkowego i modyfikującego. Konkretnie należy spowodować, aby wyrażenie warunkowe było jednocześnie wyrażeniem modyfikującym. Tworzenie takich konstrukcji umożliwiają nam, na przykład, operatory inkrementacji i dekrementacji.

Ćwiczenie 3.9.

Napisz taką pętlę typu for, aby wyrażenie warunkowe było jednocześnie wyrażeniem modyfikującym.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        for (int i = 1; i++ <= 100;){
              Console.Write("C# ");
        }
  }
}
```

W podobny sposób jak w poprzednich przykładach możemy się w pozbyć również wyrażenia początkowego. Należy je przenieść przed pętlę. Schematyczna konstrukcja wygląda następująco:

```
wyrażenie początkowe
for (; wyrażenie warunkowe;){
  instrukcje do wykonania
  wyrażenie modyfikujące
}
```

Ćwiczenie 3.10.

Zmodyfikuj pętle typu for w taki sposób, aby wyrażenie początkowe znalazło się przed pętlą, a wyrażenie modyfikujące wewnątrz pętli.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        int i = 1;
        for (; i <= 100;){
              Console.Write("C# ");
              i++;
        }
  }
}
```

Skoro zaszliśmy już tak daleko w pozbywaniu się wyrażeń sterujących, usuńmy również wyrażenie warunkowe. Jest to jak najbardziej możliwe!

Ćwiczenie 3.11.

Zmodyfikuj pętle typu for w taki sposób, aby wyrażenie początkowe znalazło się przed pętlą, natomiast wyrażenie modyfikujące i warunkowe wewnątrz pętli.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        int i = 1;
        for (; ;){
              Console.Write("C# ");
              if (i++ >= 100) break;
        }
  }
}
```

Jak można zauważyć, taka pętla też jest możliwa. Przypominam raz jeszcze, że średniki (tym razem już dwa) w nawisach po `for` są niezbędne! Warto też zwrócić uwagę na zmianę kierunku nierówności. Wcześniej sprawdzaliśmy, czy `i` jest mniejsze bądź równe 100, a teraz, czy jest większe bądź równe. Dzieje się tak dlatego, że we wcześniejszych ćwiczeniach sprawdzaliśmy, czy pętla ma się dalej wykonywać, natomiast w obecnym, czy ma się zakończyć.

W zasadzie, zamiast nierówności moglibyśmy napisać również `if (i++ == 100) break;`. Przy okazji nauczyliśmy się, w jaki sposób wykorzystuje się instrukcję `break` w przypadku pętli. Służy ona oczywiście do natychmiastowego przerwania wykonywania pętli.

Kolejna przydatna instrukcja, `continue`, powoduje rozpoczęcie kolejnej iteracji — w miejscu jej wystąpienia wykonywanie bieżącej iteracji jest przerywane i rozpoczyna się kolejny obieg. Najlepiej zobaczyć to na konkretnym przykładzie.

Ćwiczenie 3.12.

Napisz program wyświetlający na ekranie liczby od 1 do 20, które nie są podzielne przez 2. Skorzystaj z pętli for i instrukcji continue.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        for (int i = 1; i <= 20; i++){
              if (i % 2 == 0)
                    continue;
              Console.WriteLine(i);
        }
  }
}
```

Wynik działania takiej aplikacji widoczny jest na rysunku 3.6. Przypominam, że % to operator dzielenia modulo — dostarcza on resztę z dzielenia. W każdym zatem przebiegu sprawdzamy, czy `i` modulo `2` nie jest przypadkiem równe `0`. Jeśli jest (zatem `i` jest podzielne przez 2), przerywamy bieżącą iterację instrukcją `continue`. W przeciwnym przypadku (`i` modulo `2` jest różne od zera) wykonujemy instrukcję `Console.WriteLine(i)`. Ostatecznie na ekranie otrzymamy wszystkie liczby od jeden do dwadzieścia, które nie są podzielne przez dwa.

Rysunek 3.6. *Program wyświetlający liczby od 1 do 20 niepodzielne przez 2*

Ten sam program można by oczywiście napisać bez użycia instrukcji `continue`. Jak tego dokonać, pokaże nam kolejne ćwiczenie.

Ćwiczenie 3.13.

Zmodyfikuj kod z ćwiczenia 3.12 tak, aby nie było konieczności użycia instrukcji continue.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        for (int i = 1; i <= 20; i++){
              if (i % 2 != 0){
                    Console.WriteLine(i);
              }
        }
  }
}
```

Pętla while

O ile pętla typu `for` służyła raczej do wykonywania z góry znanej ilości operacji, w przypadku pętli `while` zwykle nie jest ona znana. Nie jest to oczywiście obligatoryjne. Tak naprawdę pętlę `while` można napisać tak, by była dokładnym funkcjonalnym odpowiednikiem pętli `for`, a pętle `for` tak, by była odpowiednikiem pętli `while`. Ogólna konstrukcja pętli typu `while` jest następująca:

```
while (wyrażenie warunkowe){
  instrukcje;
}
```

Instrukcje są wykonywane tak długo, dopóki wyrażenie warunkowe jest prawdziwe. Oznacza to, że gdzieś w pętli musi nastąpić modyfikacja warunku, bądź też instrukcja `break`. Inaczej pętla będzie się wykonywała w nieskończoność!

Ćwiczenie 3.14.

Używając pętli typu while, napisz program wyświetlający na ekranie 100 razy napis C#.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        int i = 1;
        while (i <= 100){
              Console.Write("C# ");
              i++;
        }

  }
}
```

Ćwiczenie 3.15.

Zmodyfikuj kod z ćwiczenia 3.14 tak, aby wyrażenie warunkowe zmieniało jednocześnie wartość zmiennej i.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        int i = 1;
        while (i++ <= 100){
              Console.Write("C# ");
        }

  }
}
```

Ćwiczenie 3.16.

Korzystając z pętli while, napisz program wyświetlający na ekranie liczby od 1 do 20 niepodzielne przez 2.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        int i = 1;
        while (i <= 20){
              if (i % 2 != 0){
                    Console.WriteLine(i);
              }
              i++;
        }
  }
}
```

Zauważmy, że w tym przypadku nie możemy skorzystać z konstrukcji takiej jak w poprzednim ćwiczeniu. Tym razem zmienna iteracyjna `i` musi być modyfikowana we wnętrzu pętli. Dzieje się tak dlatego, że wewnątrz korzystamy z jej wartości. Gdybyśmy zatem napisali pętlę tę w postaci `while (i++ <= 20)`, otrzymalibyśmy na ekranie liczby od 2 do 21 niepodzielne przez 2, co byłoby niezgodne z założeniami programu.

Pętla do while

Oprócz przedstawionych dotychczas istnieje jeszcze jedna odmiana pętli. Mianowicie `do…while`. Jej konstrukcja jest następująca:

```
do{
  instrukcje;
}
while(warunek);
```

Zapis ten należy rozumieć jako: wykonuj `instrukcje`, dopóki `warunek` jest prawdziwy. Spróbujmy zatem wykonać zadanie przedstawione ćwiczeniu 3.14, ale korzystając z pętli typu `do…while`.

Ćwiczenie 3.17.

Korzystając z pętli do... while, napisz program wyświetlający na ekranie 100 razy napis C#.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        int i = 1;
        do{
              Console.Write("C# ");
```

```
        }
        while (i++ != 10);
    }
}
```

Wydawać by się mogło, że to przecież to samo, co zwykła pętla `while`. Wydaje się wręcz, że to po prostu odwrócona pętla `while`. Jest jednak pewna różnica powodująca, że `while` i `do…while` nie są dokładnymi odpowiednikami. Otóż w przypadku pętli `do… while` instrukcje wykonane będą co najmniej jeden raz, nawet jeśli warunek jest na pewno fałszywy. Dzieje się tak oczywiście dlatego, że sprawdzenie warunku zakończenia pętli odbywa się dopiero po jej pierwszym przebiegu.

Ćwiczenie 3.18.

Zmodyfikuj kod z ćwiczenia 3.17 w taki sposób, aby wyrażenie warunkowe na pewno było fałszywe. Zaobserwuj wyniki działania programu.

```
using System;

class main
{
  public static void Main(string[] args)
  {
        int i = 101;
        do{
              Console.Write("C# ");
        }
        while (i++ <= 100);
  }
}
```

Tym razem, mimo że warunek był fałszywy (początkowa wartość zmiennej `i` to 101, które na pewno jest większe niż użyte w wyrażeniu warunkowym 100), na ekranie pojawi się jeden napis `C#`. Jeszcze dobitniej fakt wykonania jednego przebiegu pętli niezależnie od prawdziwości warunku można pokazać, stosując konstrukcję:

```
do{
      Console.Write("C# ");
}
while (false);
```

W tym przypadku już na pierwszy rzut oka widać, że warunek jest fałszywy. Prawda?

Wprowadzanie danych

Wiemy, jak wyprowadzać dane na konsolę, czyli po prostu jak wyświetlać je na ekranie. Bardzo przydałaby się jednak możliwość ich wprowadzania do programu. Nie jest to bardzo skomplikowane zadanie, choć napotkamy pewne trudności związane z koniecznością dokonywania konwersji typów danych.

Argumenty wiersza poleceń

Przekazywanie danych do aplikacji jako argumentów w wierszu poleceń przy wywoływaniu programu jest dobrze znanym większości programistów sposobem. Taka możliwość występuje prawdopodobnie w większości popularnych języków programowania. Nie inaczej jest w C#, gdzie, aby z tych danych skorzystać, należy odpowiednio zadeklarować funkcję `Main`. Konkretnie deklaracja powinna wyglądać następująco:

```
public static void main(string[] args)
{
  //instrukcje
}
```

Jak widać, argumenty są przekazywane do aplikacji w postaci tablicy obiektów `string`. Ponieważ w C#, podobnie jak w Javie, tablice są obiektami, nie ma potrzeby dodatkowego przekazywania parametru określającego liczbę argumentów, jak ma to miejsce w C i C++. Jej rozmiar określany jest parametrem `Length` (bliższe informacje o tablicach znajdują się w rozdziale piątym).

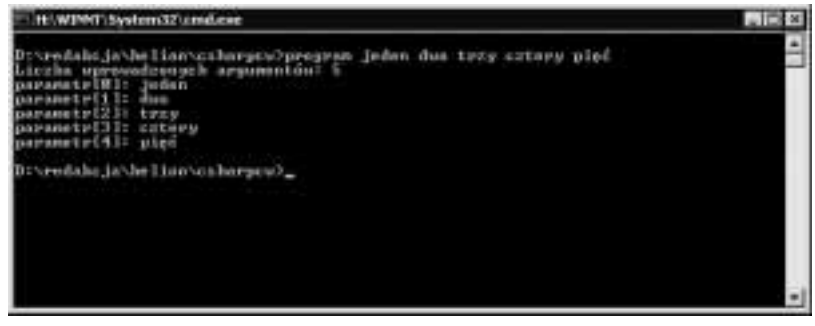
Ćwiczenie 3.19.

Wyświetl na ekranie wszystkie argumenty podane przez użytkownika w wierszu poleceń.

```
using System;

public
class main
{
  public static void Main(string[] args)
  {
        Console.WriteLine("Liczba wprowadzonych argumentów: {0}", args.Length);
        for(int i = 0; i < args.Length; i++){
              Console.WriteLine("parametr[{0}]: {1}", i, args[i]);
        }
  }
}
```

Dane są wyprowadzane na ekran przy użyciu typowej pętli `for`. Przykładowy efekt działania programu widoczny jest na rysunku 3.7.



Rysunek 3.7. *Program wyświetlający argumenty podane w wierszu poleceń*

Przy pracy z argumentami przekazywanymi z wiersza poleceń napotkamy na pewien problem. Załóżmy bowiem, że chcielibyśmy napisać program, który dodaje do siebie dwie liczby całkowite i wyświetla wynik tego działania na ekranie. Wydawać by się mogło, że najprostsze rozwiązanie wygląda tak jak na poniższym listingu:

```
using System;

public
class main
{
  public static void Main(string[] args)
  {
        if(args.Length < 2){
              Console.WriteLine("Należy podać dwa argumenty w wierszu poleceń!");
              return;
        }
        Console.WriteLine("Wynikiem działania jest: {0}", args[0] + args[1]);
  }
}
```

Oczywiście nie jest ono prawidłowe, gdyż dokonujemy tutaj operacji na dwóch ciągach znaków, a nie dwóch liczbach! Zatem wynikiem dodawania, na przykład, 12 i 8 będzie w powyższym programie 128, zamiast spodziewanego 20. Aby aplikacja działała poprawnie, należy najpierw dokonać konwersji argumentów z typu `string` do typu `int`, a dopiero potem wykonywać działanie. Konwersji takiej dokonamy przy wykorzystaniu statycznej metody `Parse` w postaci:

```
int zmienna = Int32.Parse("liczba");
```

Ćwiczenie 3.20.

Napisz program dokonujący dodawania dwóch liczb podanych jako parametry w wierszu poleceń.

```
using System;

public
class main
{
  public static void Main(string[] args)
  {
        int a, b;
        if(args.Length < 2){
              Console.WriteLine("Należy podać dwa argumenty w wierszu poleceń!");
              return;
        }
        try{
              a = Int32.Parse(args[0]);
              b = Int32.Parse(args[1]);
        }
        catch(Exception){
              Console.WriteLine("Jeden z argumentów nie jest poprawną liczbą!");
              return;
        }
        Console.WriteLine("Wynikiem działania jest: {0}", a + b);
  }
}
```

Konwersji z typu `string` do typu `int` dokonuje wspomniana już instrukcja `Int32.Parse`. Dodatkowo sprawdzamy również, czy konwersja ta zakończyła się sukcesem poprzez zastosowanie bloku `try…catch`. Dokładniejsze wytłumaczenie tej konstrukcji znajduje się w rozdziale szóstym, w którym opisane jest stosowanie wyjątków.

Przypomnijmy sobie teraz aplikację napisaną w ćwiczeniach 3.2 i 3.3. Obliczała ona pierwiastki równania kwadratowego, jednak argumenty tego równania były podawane bezpośrednio w kodzie. Za każdym razem, kiedy następowała konieczność ich zmiany, musieliśmy ponownie kompilować program. Na pewno nie było to zbyt wygodne. Teraz, kiedy wiemy już, w jaki sposób stosować argumenty, podając je w wierszu poleceń, i wiemy, w jaki sposób dokonać konwersji danych, możemy pokusić się o spore usprawnienie tamtych aplikacji.

Ćwiczenie 3.21.

Napisz program obliczający pierwiastki równania kwadratowego, w którym parametry równania są wprowadzane w wierszu poleceń (rysunek 3.8).

```
using System;

class Pierwiastek
{
  public static void Main(string[] args)
  {
        int parametrA, parametrB, parametrC;

        if(args.Length < 3){
              Console.WriteLine("Wywołanie programu: program parametr1
              ➥parametr2 parametr3");
              return;
        }

        try{
              parametrA = Int32.Parse(args[0]);
              parametrB = Int32.Parse(args[1]);
              parametrC = Int32.Parse(args[2]);
        }
        catch(Exception){
              Console.WriteLine("Jeden z parametrów równania nie jest poprawną
              ➥liczbą całkowitą!");
              return;
        }

        Console.WriteLine("Wprowadzone parametry równania:\n");
        Console.WriteLine("A: " + parametrA + " B: " + parametrB + " C: "
        ➥+ parametrC + "\n");

        if (parametrA == 0){
              Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");
        }
        else{
              double delta = parametrB * parametrB - 4 * parametrA * parametrC;
              double wynik;

              if (delta < 0){
                    Console.WriteLine("Delta < 0.");
```

```
                        Console.WriteLine("To równanie nie ma rozwiązania w zbiorze
                        ➥liczb rzeczywistych");
                }
                else if (delta == 0){
                        wynik = - parametrB / 2 * parametrA;
                        Console.WriteLine("Rozwiązanie: x = " + wynik);
                }
                else{
                        wynik = (- parametrB + Math.Sqrt(delta)) / 2 * parametrA;
                        Console.Write("Rozwiązanie: x1 = " + wynik);
                        wynik = (- parametrB - Math.Sqrt(delta)) / 2 * parametrA;
                        Console.WriteLine(", x2 = " + wynik);
                }
        }
  }
}
```

Rysunek 3.8. *Równania kwadratowe o parametrach podanych w wierszu poleceń*

Instrukcja ReadLine

Wprowadzanie danych do programu w wierszu poleceń nie zawsze jest wygodne. Często chcielibyśmy wykonywać tę czynność już w trakcie działania programu. Pomoże nam w tym instrukcja `Console.ReadLine()`, która zwraca wprowadzoną przez użytkownika jedną linię tekstu (ciąg znaków zakończony znakiem końca linii).

Ćwiczenie 3.22.

Napisz program, który w pętli wczytuje kolejne wiersze tekstu wprowadzane przez użytkownika i wyświetla je na ekranie. Program powinien zakończyć działanie po odczytaniu ciągu znaków quit.

```
using System;

class Pierwiastek
{
  public static void Main()
  {
        string s;
        while(!(s = Console.ReadLine()).Equals("quit")){
                Console.WriteLine(s);
        }
  }
}
```

Warunek w pętli `while` zapisaliśmy w bardzo skondensowanej formie. Kolejność wykonywania instrukcji jest tu następująca:

1. Odczytanie linii znaków z konsoli (`Console.ReadLine()`).
2. Przypisanie odczytanej wartości do zmiennej s (`s=`).
3. Wywołanie metody `Equals` na rzecz obiektu wskazywanego przez s (`Equals("quit")`).

Należy w tym miejscu zwrócić również uwagę, że taki zapis może w pewnych sytuacjach spowodować błąd w działaniu aplikacji, konkretnie wygenerowanie wyjątku `NullReferenceException`. Dlaczego? Otóż może się zdarzyć, że metoda `ReadLine()` zamiast ciągu znaków zwróci wartość `null`. W takim wypadku wartością przypisaną do zmiennej `s` również będzie `null`. Skoro tak, nie będzie możliwe wykonanie metody `Equals`, stąd też wygenerowanie wyjątku.

Zapobiec takiej sytuacji można na dwa sposoby. Albo rozbijając warunek pętli `while` na klika oddzielnych instrukcji i sprawdzając, czy `s` nie jest równe `null`, albo przez odwrócenie kolejności wykonywania instrukcji, czyli zamiast pisać

```
(s = Console.ReadLine()).Equals("quit")
```

można zastosować konstrukcję

```
"quit".Equals(s = Console.ReadLine())
```

Ćwiczenie 3.23.

Popraw kod z ćwiczenia 3.22, usuwając warunek z pętli while, i przenieś go do wnętrza pętli.

```
using System;

class Pierwiastek
{
  public static void Main()
  {
        string s = "";
        while(true){
              s = Console.ReadLine();
              if(s != null && !s.Equals("quit")){
                    Console.WriteLine(s);
              }
              else{
                    break;
              }
        }
  }
}
```

Skoro potrafimy już wykorzystywać instrukcję `ReadLine()`, możemy poprawić program do obliczania pierwiastków równania kwadratowego tak, aby parametry były wprowadzane w trakcie działania programu. Będzie on teraz prosił użytkownika o podanie kolejnych liczb i podstawiał je pod zmienne `parametrA`, `parametrB` i `parametrC`.

Oczywiście przed przypisaniem danych do wspomnianych zmiennych musimy dokonać konwersji z typu `string` na typ `int`. Co jednak powinniśmy zrobić w sytuacji, kiedy użytkownik nie poda prawidłowej liczby, ale, na przykład, wpisze dowolną kombinację znaków? Najlepiej byłoby poprosić o ponowne wprowadzenie parametru. Jak tego dokonać? Najwygodniej będzie skorzystać z pętli `while` lub `do…while` i prosić użytkownika o wprowadzanie liczby tak długo, dopóki nie poda poprawnej.

Skoro jednak mamy trzy zmienne, a tym samym trzy parametry do wprowadzenia, nie ma sensu pisać trzech pętli wyglądających praktycznie tak samo. Lepiej utworzyć dodatkową funkcję, której zadaniem będzie właśnie dostarczenie prawidłowej liczby całkowitej wprowadzonej przez użytkownika.

Ćwiczenie 3.24.

Napisz program obliczający pierwiastki równania kwadratowego, w którym parametry są wprowadzane w trakcie działania programu.

```
using System;

class Pierwiastek
{
  public static int pobierzLiczbe(string param)
  {
        int liczba = 0;
        bool sukces;
        do{
              Console.WriteLine("Proszę podać {0} parametr równania:", param);
              try{
                    liczba = Int32.Parse(Console.ReadLine());
                    sukces = true;
              }
              catch(Exception){
                    Console.WriteLine("Podany parametr nie jest prawidłową
                    ➥liczbą całkowitą!");
                    sukces = false;
              }
        }
        while(!sukces);
        return liczba;
  }
  public static void Main(string[] args)
  {
        int parametrA, parametrB, parametrC;

        parametrA = pobierzLiczbe("pierwszy");
        parametrB = pobierzLiczbe("drugi");
        parametrC = pobierzLiczbe("trzeci");

        Console.WriteLine("Wprowadzone parametry równania:\n");
        Console.WriteLine("A: " + parametrA + " B: " + parametrB + " C:
        ➥" + parametrC + "\n");

        if (parametrA == 0){
              Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");
        }
```

```
        else{
                double delta = parametrB * parametrB - 4 * parametrA * parametrC;
                double wynik;

                if (delta < 0){
                        Console.WriteLine("Delta < 0.");
                        Console.WriteLine("To równanie nie ma rozwiązania
                        ➥w zbiorze liczb rzeczywistych");
                }
                else if (delta == 0){
                        wynik = - parametrB / 2 * parametrA;
                        Console.WriteLine("Rozwiązanie: x = " + wynik);
                }
                else{
                        wynik = (- parametrB + Math.Sqrt(delta)) / 2 * parametrA;
                        Console.Write("Rozwiązanie: x1 = " + wynik);
                        wynik = (- parametrB - Math.Sqrt(delta)) / 2 * parametrA;
                        Console.WriteLine(", x2 = " + wynik);
                }
        }
    }
}
```

Na tym w zasadzie moglibyśmy zakończyć ćwiczenia z wprowadzania danych i rozwiązywania równań kwadratowych, ale spróbujmy wykonać jeszcze jeden przykład. Zauważmy bowiem, że wygodnie byłoby, aby nasz program umożliwiał zarówno podawanie parametrów w wierszu poleceń, jak i w trakcie swojego działania. Dzięki temu użytkownik mógłby wybrać bardziej wygodny dla niego sposób. Co więcej, jeśli przy podawaniu danych w wierszu poleceń pomyli się, aplikacja pozwoli na ponowne ich wprowadzenie.

Musimy zatem połączyć kod z ćwiczenia 3.21 z kodem z ćwiczenia 3.24. Dodatkowo powinniśmy wprowadzić jeszcze jedno usprawnienie. Do tej pory zakładaliśmy bowiem, że parametry równania muszą być liczbami rzeczywistymi. Nie ma jednak żadnego powodu, aby dalej utrzymywać takie ograniczenie. Pozwólmy, aby nasza aplikacja potrafiła również rozwiązywać równania, w których parametrami są liczby rzeczywiste.

Dodatkowymi zmianami będzie więc zmiana typu zmiennych `parametrA`, `parametrB` i `parametrC` z `int` na `double` oraz skorzystanie z metody `Parse` klasy `Double`.

Ćwiczenie 3.25.

Napisz program rozwiązujący równanie kwadratowe o zadanych parametrach, będących dowolnymi liczbami rzeczywistymi. Parametry mogą być wprowadzane zarówno w trakcie działania programu, jak i w wierszu poleceń.

```
using System;

class Pierwiastek
{
  public static double pobierzLiczbe(string param)
  {
        double liczba = 0;
        bool sukces;
        do{
```

```
                Console.WriteLine("Proszę podać {0} parametr równania:", param);
                try{
                        liczba = Double.Parse(Console.ReadLine());
                        sukces = true;
                }
                catch(Exception){
                      Console.WriteLine("Podany parametr nie jest prawidłową liczbą!");
                        sukces = false;
                }
        }
        while(!sukces);
        return liczba;
}
public static void Main(string[] args)
{
        bool liniaKomend = true;
        double parametrA = 0, parametrB = 0, parametrC = 0;

        if(args.Length < 3){
                liniaKomend = false;
        }
        if(liniaKomend){
                try{
                        parametrA = Double.Parse(args[0]);
                        parametrB = Double.Parse(args[1]);
                        parametrC = Double.Parse(args[2]);
                }
                catch(Exception){
                        Console.WriteLine("Jeden z wprowadzonych parametrów
                        ➥nie jest poprawną liczbą!");
                        liniaKomend = false;;
                }
       }

        if(!liniaKomend){
                parametrA = pobierzLiczbe("pierwszy");
                parametrB = pobierzLiczbe("drugi");
                parametrC = pobierzLiczbe("trzeci");
        }

        Console.WriteLine("Wprowadzone parametry równania:\n");
        Console.WriteLine("A: " + parametrA + " B: " + parametrB +
        ➥" C: " + parametrC + "\n");

        if (parametrA == 0){
                Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");
        }
        else{
                double delta = parametrB * parametrB - 4 * parametrA * parametrC;
                double wynik;

                if (delta < 0){
                        Console.WriteLine("Delta < 0.");
                        Console.WriteLine("To równanie nie ma rozwiązania
                        ➥w zbiorze liczb rzeczywistych");
                }
                else if (delta == 0){
                        wynik = - parametrB / 2 * parametrA;
                        Console.WriteLine("Rozwiązanie: x = " + wynik);
```

```
                }
                else{
                        wynik = (- parametrB + Math.Sqrt(delta)) / 2 * parametrA;
                        Console.Write("Rozwiązanie: x1 = " + wynik);
                        wynik = (- parametrB - Math.Sqrt(delta)) / 2 * parametrA;
                        Console.WriteLine(", x2 = " + wynik);
                }
        }
    }
}
```

Nasz program potrafi już skorzystać z liczb rzeczywistych (rysunek 3.9). Jeżeli w wierszu poleceń podane zostały trzy parametry i każdy z nich jest prawidłową liczbą całkowitą, zostaną one użyte do rozwiązania równania. Jeśli jednak parametrów jest mniej lub przy wprowadzaniu któregokolwiek z nich został popełniony błąd, aplikacja poprosi o ponowne ich wprowadzenie.

Rysunek 3.9. *Parametry równania mogą być podawane w postaci liczb rzeczywistych*

Należy zwrócić uwagę, że sposób wprowadzania liczb z przecinkiem zależy od ustawień regionalnych systemu! Konkretnie od tego, jaki symbol został ustalony jako separator dziesiętny. W przypadku ustawień polskich domyślnie jest to przecinek, przy ustawieniach angielskich — kropka.