Dodatek B
Elementy programowania współbieżnego

Do platformy .NET w wersji 4.0 dodana została biblioteka TPL (od ang. *Task Parallel Library*) oraz kolekcje umożliwiające pracę w różnych scenariuszach pojawiających się przy programowaniu współbieżnym. Całość popularnie nazywana jest *Parallel Extensions*. TPL nadbudowuje klasyczne wątki i pulę wątków, korzystając z nowej klasy `Task` (z ang. „zadanie”).

W szczegółach biblioteka TPL, mechanizmy synchronizacji, „kolekcje równoległe” oraz narzędzia pozwalające na debugowanie i analizowanie programów równoległych opisane zostały w książce *Programowanie równoległe i asynchroniczne w C# 5.0*, wydanej przez wydawnictwo Helion w grudniu 2013 roku. Tu chciałbym skupić się jedynie na najczęściej używanym jej elemencie — współbieżnej pętli `for` — oraz na słowach kluczowych C# `async` i `await`, a więc na definiowaniu metod wykonywanych asynchronicznie.

Równoległa pętla for

Załóżmy, że mamy zbiór stu liczb rzeczywistych, dla których musimy wykonać jakieś stosunkowo czasochłonne obliczenia. W naszym przykładzie będzie to obliczenie wartości funkcji $y = f(x) = \arcsin(\sin(x))$. Funkcja ta powinna z dokładnością numeryczną zwrócić wartość argumentu. I zrobi to, jednak trochę się przy tym namęczy — funkcje trygonometryczne są bowiem wymagające numerycznie. Powtórzmy te obliczenia kilkukrotnie, aby dodatkowo przedłużyć czas ich trwania. Listing B.1 prezentuje kod pliku *Program.cs* z aplikacji konsolowej, w której zaimplementowany został powyższy pomysł.

Listing B.1. Metoda zajmująca procesor

```
using System;


namespace ProgramowanieRównoległe
{
    class Program
    {
        static private double obliczenia(double argument)
        {
            for (int i = 0; i < 10; ++i)
            {
                argument = Math.Asin(Math.Sin(argument));
            }
            return argument;
        }
```

```
        static void Main(string[] args)
        {
            //przygotowania
            int rozmiar = 10000;
            Random r = new Random();
            double[] tablica = new double[rozmiar];
            for (int i = 0; i < tablica.Length; ++i) tablica[i] = r.NextDouble();

            //obliczenia sekwencyjne
            int liczbaPowtórzeń = 100;
            double[] wyniki = new double[tablica.Length];
            int start = System.Environment.TickCount;
            for (int powtórzenia = 0; powtórzenia < liczbaPowtórzeń; ++powtórzenia)
                for (int i = 0; i < tablica.Length; ++i)
                    wyniki[i] = obliczenia(tablica[i]);
            int stop = System.Environment.TickCount;
            Console.WriteLine("Obliczenia sekwencyjne trwały " +
                              (stop - start).ToString() + " ms.");

            /*
            //prezentacja wyników
            Console.WriteLine("Wyniki:");
            for (long i = 0; i < tablica.Length; ++i)
                Console.WriteLine(i + ". " + tablica[i] + " ?= " + wyniki[i]);
            */
        }
    }
}
```

Na tym samym listingu, w metodzie `Main`, widoczna jest pętla `for` wykonująca obliczenia na wcześniej przygotowanej tablicy liczb `double`. Wyniki nie są drukowane (kod w komentarzu) — tablica jest zbyt duża, żeby to miało sens (poza tym drukowanie w konsoli jest bardzo wolne i zdominowałoby czas obliczeń).

Powyższy kod zawiera dwie zagnieżdżone pętle `for`. Interesuje nas tylko ta wewnętrzna. Zewnętrzna jedynie powtarza obliczenia, co zwiększa wiarygodność pomiaru czasu, który realizujemy, wykorzystując własność `TickCount` ze statycznej klasy `Environment` (por. dodatek A). Pętlę wewnętrzną można bez większego wysiłku zrównoleglić dzięki klasie `Parallel` z przestrzeni nazw `System.Threading.Tasks`. Zrównoleglona wersja tej pętli (bez zmiany zewnętrznej pętli powtarzającej obliczenia) widoczna jest na listingu B.2. Dodajmy ją do metody `Main`.

Listing B.2. Przykład zrównoleglonej pętli for

```
//obliczenia równoległe
start = System.Environment.TickCount;
for (int powtórzenia = 0; powtórzenia < liczbaPowtórzeń; ++powtórzenia)
{
    Parallel.For(0, tablica.Length, (int i) => wyniki[i] = obliczenia(tablica[i]));
}
stop = System.Environment.TickCount;
```

```
Console.WriteLine("Obliczenia równoległe trwały " + (stop - start).ToString() + " ms.");
```

Użyta do zrównoleglenia metoda `Parallel.For` jest dość intuicyjna. Jej dwa pierwsze argumenty określają zakres indeksu pętli. W naszym przypadku jest on równy [0,10000). Wobec tego do metody podanej w trzecim argumencie przekazywane są liczby od 0 do 9999. Trzeci argument jest natomiast delegacją, do której można przypisać metodę (lub jak w naszym przypadku wyrażenie lambda). Jej polecenia wykonywane są w każdej iteracji pętli. Aby porównywanie czasów miało sens, powinna się tam znaleźć zawartość oryginalnej pętli. Argumentem wyrażenia lambda jest bieżący indeks pętli.

Proces zrównoleglania kodu może być oczywiście tak prosty, jak pokazałem powyżej, ale należy z góry uprzedzić, że są to rzadkie przypadki. Często w ogóle zrównoleglenie nie jest możliwe albo wymaga synchronizacji. Tak jest na przykład w pętli `for` z listingu 5.7, obliczającej silnię podanej w argumencie liczby, w której kolejne iteracje zależą od poprzednich — muszą być wobec tego wykonywane sekwencyjnie. Metoda `Parallel.For` automatycznie synchronizuje zadania po zakończeniu wszystkich iteracji, dlatego nie ma niebezpieczeństwa zamazania danych w ramach kolejnych jej powtórzeń (zewnętrzna pętla). Jeżeli jednak zajdzie taka potrzeba, synchronizację można zrealizować za pomocą operatora `lock`.

Warto przestrzec też, że nie należy się spodziewać, iż dzięki użyciu równoległej pętli `for` nasze obliczenia przyspieszą tyle razy, ile rdzeni procesora mamy do dyspozycji. Tworzenie i usuwanie zadań również zajmuje nieco czasu. Eksperymentując z rozmiarem tablicy i liczbą obliczanych sinusów, można sprawdzić, że zrównoleglanie opłaca się tym bardziej, im dłuższe są obliczenia wykonywane w ramach jednego zadania. Dla krótkich zadań użycie równoległej pętli może wręcz wydłużyć całkowity czas obliczeń. Na komputerze z jednym procesorem dwurdzeniowym uzyskane przeze mnie uśrednione przyspieszenie dla powyższych parametrów było równe 66,4%. Z kolei w przypadku ośmiu rdzeni czas obliczeń równoległych spadł do zaledwie 34,8%.

Przerywanie pętli

Podobnie jak w klasycznej pętli `for` również w przypadku wersji równoległej możemy w każdej chwili przerwać jej działanie. Nie robi się tego jednak instrukcją `break`. Służy do tego klasa `ParallelLoopState`, której instancję należy przesłać jako drugi argument metody lub wyrażenia lambda wykonywanego w każdej iteracji. Klasa ta udostępnia dwie metody: `Break` i `Stop`. Metoda `Break` pozwala na wcześniejsze zakończenie bieżącej iteracji bez uruchamiania następnych, a `Stop` nie tylko natychmiast kończy bieżący wątek, ale również podnosi flagę `IsStopped`, która powinna zostać sprawdzona we wszystkich uruchomionych wcześniej iteracjach i która jest sygnałem do ich natychmiastowego zakończenia (to leży jednak w gestii programisty przygotowującego kod wykonywany w każdej iteracji). Listing B.3 pokazuje przykład, w którym pętla jest przerywana, jeżeli wylosowana zostanie liczba 0.

Listing B.3. Przerywanie pętli równoległej

```
static void Main(string[] args)
{
    Random r = new Random();
    long suma = 0;
    long licznik = 0;
    string s = "";

    //iteracje zostaną wykonane tylko dla liczb parzystych
    //pętla zostanie przerwana wcześniej, jeżeli wylosowana liczba jest równa 0
    Parallel.For(0, 10000, (int i, ParallelLoopState stanPetli) =>
    {
        int liczba = r.Next(7); // losowanie liczby oczek na kostce
        if (liczba == 0)
        {
            s += "Stop:";
```

```
                stanPetli.Stop();
            }
            if (stanPetli.IsStopped) return;
            if (liczba % 2 == 0)
            {
                s += liczba.ToString() + "; ";
                obliczenia(liczba);
                suma += liczba;
                licznik++;
            }
            else s += "[" + liczba.ToString() + "]; ";
        });

        Console.WriteLine(
            "Wylosowane liczby: " + s +
            "\nLiczba pasujących liczb: " + licznik +
            "\nSuma: " + suma +
            "\nŚrednia: " + (suma / (double)licznik).ToString());
    }
```

Programowanie asynchroniczne. Modyfikator async i operator await

Programowanie asynchroniczne to niezwykle istotny element aplikacji Universal Windows Platform (UWP), czyli aplikacji dla Windows 10, które mogą być dystrybuowane w sklepie Microsoft Store. Jednak mechanizm ten może być z powodzeniem używany również w „zwykłych” aplikacjach dla platform .NET i .NET Core. Pojawia się chociażby w Entity Framework od wersji 6.0 (rozdział 27.).

Metody wykonywane asynchronicznie można oczywiście również definiować samodzielnie. W dalszej części rozdziału postaram się opisać mechanizm asynchroniczności właśnie w taki sposób, aby umożliwić czytelnikowi używanie go we własnych projektach.

Język C# od wersji 5.0 został wyposażony w nowy operator `await`, ułatwiający synchronizację dodatkowych zadań uruchomionych przez użytkownika. Poniżej zaprezentuję prosty przykład, który chyba najlepiej wyjaśni jego działanie. Nie omówiłem wprawdzie zadań (mam na myśli bibliotekę TPL i jej sztandarową klasę `Task`), jednak — podobnie jak w przypadku opisanej wyżej pętli równoległej `Parallel.For` — dogłębna znajomość biblioteki TPL nie jest konieczna, by używać operatora `await`. Spójrzmy na listing B.4. Przedstawia on nową metodę `Main`, która definiuje przykładowe wyrażenie lambda i zapisuje referencję do niego w zmiennej `akcja`, następnie synchronicznie je wykonuje. Wyrażenie lambda wprowadza półsekundowe opóźnienie uzyskiwane dzięki użyciu metody `Thread.Sleep`, które opóźnia wyświetlenie informacji o zakończeniu jego działania.

Listing B.4. Synchroniczne wykonywanie kodu zawartego w akcji

```
    static void Main(string[] args)
    {
        //czynność
        Func<object, long> akcja =
            (object argument) =>
            {
                Console.WriteLine("Początek działania akcji - " + argument.ToString());
```

```
                System.Threading.Thread.Sleep(500); //opóźnienie 0.5s
                Console.WriteLine("Koniec działania akcji - " + argument.ToString());
                return DateTime.Now.Ticks;
            };

        long wynik = akcja("synchronicznie");
        Console.WriteLine("Synchronicznie: " + wynik.ToString());
    }
```

W listingu B.5 ta sama akcja wykonywana jest asynchronicznie w osobnym wątku utworzonym na potrzeby zdefiniowanego przez nas zadania. Synchronizacja następuje w momencie odczytania wartości zwracanej przez wyrażenie lambda, tj. w momencie odczytania właściwości `Result`. Jej sekcja `get` czeka ze zwróceniem wartości aż do zakończenia zadania i tym samym wstrzymuje wątek, w którym wykonywana jest metoda `Main`. Jest to zatem punkt synchronizacji. Zwróćmy też uwagę na to, że po instrukcji `zadanie.Start`, a przed odczytaniem właściwości `Result` mogą być wykonywane dowolne inne czynności, o ile są niezależne od wartości zwróconej przez zadanie.

Listing B.5. Użycie zadania do asynchronicznego wykonania kodu

```
    static void Main(string[] args)
    {
        //czynność
        Func<object, long> akcja =
            (object argument) =>
            {
                Console.WriteLine("Początek działania akcji - " + argument.ToString());
                System.Threading.Thread.Sleep(500); //opóźnienie 0.5s
                Console.WriteLine("Koniec działania akcji - " + argument.ToString());
                return DateTime.Now.Ticks;
            };

        ~~long wynik = akcja("synchronicznie");~~
        ~~Console.WriteLine("Synchronicznie: " + wynik.ToString());~~

        //w osobnym zadaniu
        Task<long> zadanie = new Task<long>(akcja, "zadanie");
        zadanie.Start();
        Console.WriteLine("Akcja została uruchomiona");
        //właściwość Result czeka ze zwróceniem wartości, aż zadanie zostanie zakończone
        //(synchronizacja)
        long wynik = zadanie.Result;
        Console.WriteLine("Zadanie: " + wynik.ToString());
    }
```

Ponadto nie jest konieczne, aby instrukcja odczytania właściwości `Result` znajdowała się w tej samej metodzie, co instrukcja uruchomienia zadania — należy tylko do miejsca jej odczytania przekazać referencję do zadania (w naszym przypadku zmienną typu `Task<long>`). Zwykle referencję tę przekazuje się jako wartość zwracaną przez metodę uruchamiającą zadanie. Zgodne z konwencją metody tworzące i uruchamiające zadania powinny zawierać w nazwie przyrostek `..Async` (listing B.6).

Listing B.6. Wzór metody wykonującej jakąś czynność asynchronicznie

```
static Task<long> ZróbCośAsync(object argument)
{
    //czynność, która będzie wykonywana asynchronicznie
    Func<object, long> akcja =
        (object _argument) =>
        {
            Console.WriteLine("Początek działania akcji - " + _argument.ToString());
            System.Threading.Thread.Sleep(500); //opóźnienie 0.5s
            Console.WriteLine("Koniec działania akcji - " + _argument.ToString());
            return DateTime.Now.Ticks;
        };

    Task<long> zadanie = new Task<long>(akcja, argument);
    zadanie.Start();
    return zadanie;
}

static void Main(string[] args)
{
    Task<long> zadanie2 = ZróbCośAsync("zadanie-metoda");
    Console.WriteLine("Akcja została uruchomiona (metoda)");
    long wynik = zadanie2.Result;
    Console.WriteLine("Zadanie-metoda: " + wynik.ToString());
}
```

Wraz z wersjami 4.0 i 4.5 w platformie .NET (oraz w platformie Windows Runtime, a potem Universal Windows Platform, a obecnie także w platformie .NET Core) pojawiło się wiele metod, które wykonują długotrwałe czynności asynchronicznie. Znajdziemy je w klasie `HttpClient`, w klasach odpowiedzialnych za obsługę plików (`StorageFile`, `StreamWriter`, `StreamReader`, `XmlReader`), w klasach odpowiedzialnych za kodowanie i dekodowanie obrazów czy też w klasach WCF. Asynchroniczność jest wręcz standardem w aplikacjach UWP dla systemu Windows 10. I właśnie aby ich użycie było (prawie) tak proste jak zastosowanie metod synchronicznych, w C# 5.0 (co odpowiada platformie .NET 4.5) został wprowadzony operator `await`. Ułatwia on synchronizację dodatkowego zadania tworzonego przez te metody. Należy jednak pamiętać, że metodę, w której chcemy użyć operatora `await`, musimy oznaczyć modyfikatorem `async`. A ponieważ modyfikatora takiego nie można dodać do metody wejściowej `Main`, stworzyłem dodatkową, wywoływaną z niej metodę `ProgramowanieAsynchroniczne`. Prezentuje to listing B.7.

Listing B.7. Przykład użycia modyfikatorów async i await

```
static async void ProgramowanieAsynchroniczne()
{
    Task<long> zadanie2 = ZróbCośAsync("zadanie-metoda");
    Console.WriteLine("Akcja została uruchomiona (metoda)");
    long wynik = await zadanie2;
    Console.WriteLine("Zadanie-metoda: " + wynik.ToString());
}

static void Main(string[] args)
{
```

```
            ProgramowanieAsynchroniczne();
            Console.Write("Naciśnij Enter..."); Console.ReadLine();
            Console.WriteLine("Koniec.");
        }
```

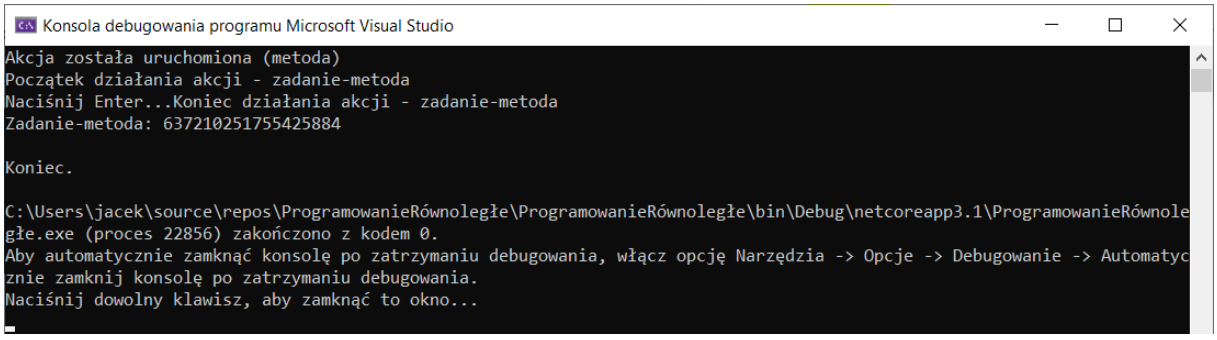
Operator `await` zwraca parametr typu `Task<>` (czyli `long` w przypadku `Task<long>`) lub `void`, jeżeli użyta została wersja nieparametryczna klasy `Task`.

W angielskiej dokumentacji MSDN metody oznaczone modyfikatorem `async` są nazywane *async methods*. Może to jednak wprowadzać pewne zamieszanie. Metody z modyfikatorem `async` (w naszym przypadku `ProgramowanieAsynchroniczne`) mylone są bowiem z metodami wykonującymi asynchronicznie jakieś czynności (w naszym przypadku metoda `ZróbCośAsync`). Osobom poznającym dopiero temat często wydaje się, że aby metoda była wykonywana asynchronicznie, wystarczy dodać do jej sygnatury modyfikator `async`. To nie jest prawda.

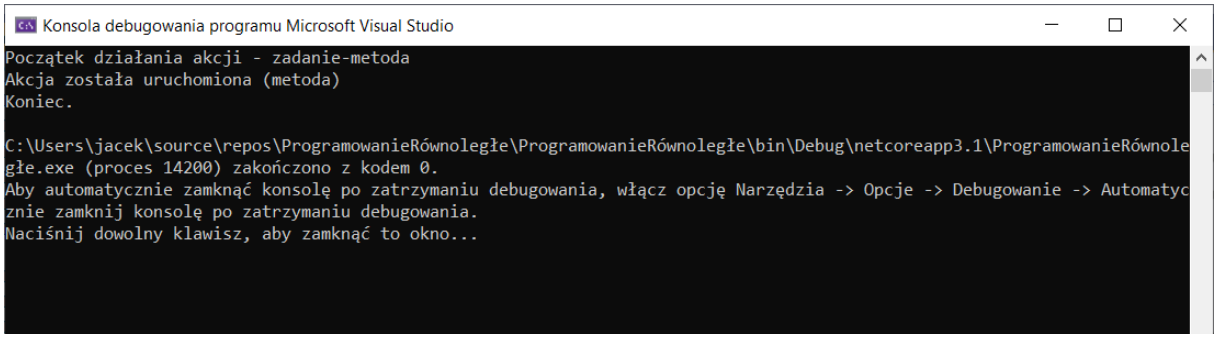
Możemy oczywiście wywołać metodę `ZróbCośAsync` w taki sposób, że umieścimy ją bezpośrednio za operatorem `await`, np. `long wynik = await ZróbCośAsync("async/await");`. Czy to ma sens? Wykonywanie metody `ProgramowanieAsynchroniczne`, w której znajduje się to wywołanie, zostanie wstrzymane aż do momentu zakończenia metody `ZróbCośAsync`, więc efekt, jaki zobaczymy na ekranie, będzie identyczny jak w przypadku synchronicznym (listing B.5). Różnica jest jednak zasadnicza, ponieważ instrukcja zawierająca operator `await` nie blokuje wątku, w którym została wywołana metoda `ProgramowanieAsynchroniczne`. Kompilator zawiesza jej wywołanie, przechodząc do kolejnych czynności aż do momentu zakończenia uruchomionego zadania. Gdy to nastąpi, wątek wraca do metody `ProgramowanieAsynchroniczne` i kontynuuje jej działanie.

„Zawieszenie działania metody `ProgramowaniaAsynchroniczne`” to obrazowe, ale mało precyzyjne sformułowanie. Tak naprawdę kompilator „tnie” tę metodę w miejscu wystąpienia operatora `await` i tę część, która ma być wykonana po zakończeniu zadania i odebraniu wyniku, umieszcza w metodzie zwrotnej (ang. *callback*). Co więcej, ta metoda zwrotna wcale nie będzie wykonywana w tym samym wątku co pierwsza część metody `ProgramowanieAsynchroniczne` i metoda `Main`; wykorzystany zostanie wątek, w którym pracowało zadanie tworzone przez metodę `ZróbCośAsync`.

Z tego wynika, że efekt działania operatora `await` zobaczymy dopiero, gdy metodę `ProgramowanieAsynchroniczne` wywołamy z innej metody, w której będą dodatkowe instrukcje wykonywane w czasie wstrzymania metody `ProgramowanieAsynchroniczne`. W naszym przykładzie wywołujemy ją z metody `Main`, która po wywołaniu metody `ProgramowanieAsynchroniczne` wstrzymuje działanie aż do naciśnięcia klawisza *Enter*. W serii instrukcji wywołanie metody oznaczonej modyfikatorem `async` nie musi się zakończyć przed wykonaniem następnej instrukcji — w tym sensie jest ona asynchroniczna. Aby tak się stało, musi w niej jednak zadziałać operator `await`, w naszym przykładzie czekający na wykonanie metody `ZróbCośAsync`. W efekcie, jeżeli w metodzie `Main` usuniemy ostatnie polecenia wymuszające oczekiwanie na naciśnięcie klawisza *Enter*, metoda ta zakończy się przed zakończeniem metody `ProgramowanieAsynchroniczne`, kończąc tym samym działanie całego programu i nie pozwalając metodzie `ZróbCośAsync` wyświetlić komunikatu o zakończeniu działania. Jeżeli polecenia te są obecne, instrukcje z metody `ZróbCośAsync` zostaną wykonane już po wyświetleniu przez metodę `Main` komunikatu o konieczności naciśnięcia klawisza *Enter*. Dowodzi tego rysunek B.1 (por. też diagram przy listingu B.8).



Materiał z książki Jacka Matulewskiego „C#. Lekcje programowania. Praktyczna nauka programowania dla platform .NET i .NET Core” dla studentów kursu .NET 2023/2024. Proszę o nieudostępnianie osobom spoza kursu.



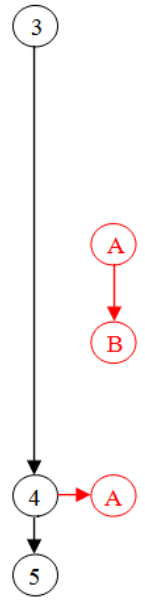
Rysunek B.1. Zadania utworzone w ten sposób nie blokują wątku głównego — działają jak wątki tła

Diagram przy listingu B.8 obrazuje przebieg programu. Liczby w okręgach oznaczają kolejne ważne punkty programu, m.in. te, w których następuje przepływ wątku z metody do metody (dwa okręgi z tą samą liczbą), lub moment uruchamiania zadania. Lewa kolumna odpowiada wątkowi głównemu (w nim wykonywana jest metoda `Main`), a prawa — wątkowi tworzonemu na potrzeby zadania, a potem wykorzystywanemu do dokończenia metody zawierającej operator `await`. Warto nad tym diagramem spędzić chwilę czasu z ołówkiem w ręku i postarać się zrozumieć, jak działa operator `await` i które fragmenty kodu czekają na zakończenie zadania, a które nie.

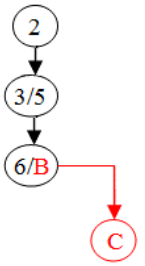
Listing B.8. Diagram przebiegu programu

```
Task<long> ZróbCośAsync(object argument)
{
   //czynność, która będzie wykonywana asynchronicznie
   Func<object, long> akcja =
      (object _argument) =>
      {
         Console.WriteLine("Początek działania akcji - " + _argument.ToString());
         System.Threading.Thread.Sleep(500); //opóźnienie 0.5s
         Console.WriteLine("Koniec działania akcji - " + _argument.ToString());
         return DateTime.Now.Ticks;
      }

   Task<long> zadanie = new Task<long>(akcja, argument);
   zadanie.Start();

   return zadanie;
}

static async void ProgramowanieAsynchroniczne()
{
   Task<long> zadanie2 = ZróbCośAsync("zadanie-metoda");
   Console.WriteLine("Akcja została uruchomiona (metoda)");
   long wynik = await zadanie2;

   Console.WriteLine("Zadanie-metoda: " + wynik.ToString());
}

static void Main(string[] args)
```

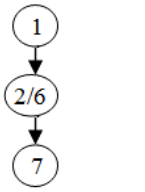
```
{
    ProgramowanieAsynchroniczne();
    Console.Write("Naciśnij Enter..."); Console.ReadLine();
    Console.WriteLine("Koniec.");
}
```




```
Task<long> ZróbCośAsync(object argument)
{
    //czynność, która będzie wykonywana asynchronicznie
    Func<object, long> akcja =
        (object _argument) =>
        {
            Console.WriteLine("Początek działania akcji - " + _argument.ToString());
            System.Threading.Thread.Sleep(500); //opóźnienie 0.5s
            Console.WriteLine("Koniec działania akcji - " + _argument.ToString());
            return DateTime.Now.Ticks;
        };

    Task<long> zadanie = new Task<long>(akcja, argument);
    zadanie.Start();

    return zadanie;
}
```




```
static async void ProgramowanieAsynchroniczne()
{
    Task<long> zadanie2 = ZróbCośAsync("zadanie-metoda");
    Console.WriteLine("Akcja została uruchomiona (metoda)");
    long wynik = await zadanie2;

    Console.WriteLine("Zadanie-metoda: " + wynik.ToString());
}
```




```
static void Main(string[] args)
{
    ProgramowanieAsynchroniczne();
    Console.Write("Naciśnij Enter..."); Console.ReadLine();
    Console.WriteLine("Koniec.");
}
```

* * *

Zgodnie z zapowiedzią to tylko najbardziej podstawowe wiadomości o głównych elementach TPL i o programowaniu asynchronicznym w C# (w wersji 5.0 i nowszych). Więcej informacji na ten temat, w szczególności o bardzo ważnym zagadnieniu synchronizacji wątków, znajdzie czytelnik we wspomnianej już książce *Programowanie równoległe i asynchroniczne w C# 5.0* (Helion, 2013).

Zadania

1. Napisz program przeszukujący tysiącelementową tablicę z losowymi wartościami całkowitymi w poszukiwaniu minimalnej i maksymalnej wartości. Przyspiesz działanie owego programu, korzystając z pętli `Parallel.For` lub `Parallel.ForEach`, i sprawdź, o ile skraca się działanie programu w zależności od liczby procesorów.
2. Napisz klasę `Czasomierz` z własnościami *auto-implemented* `Interwał` typu `int` i `Włączony` typu `bool`. Jeżeli własność `Włączony` równa jest `true`, co liczbę milisekund określoną przez własność `Interwał` wywołuj metodę przekazaną przez konstruktor klasy `Czasomierz` (jego argumentem powinna być delegacja).