Programowanie asynchroniczne. Operator await i modyfikator async

Język C# 5.0 wyposażony został w nowy operator `await`, ułatwiający synchronizację dodatkowych zadań uruchomionych przez użytkownika. Poniżej zaprezentuję prosty przykład jego użycia, który chyba wyjaśni jego działanie. Działnie tego operatora związane jest ściśle z biblioteką TPL (od ang. *Task Parallel Library*) i jej sztandarową klasą `Task`, które zostaną omówione w kolejnych rozdziałach. Jednak podobnie jak w przypadku opisanej niżej pętli równoległej `Parallel.For`, tak i w przypadku operatora `await`, dogłębna znajomość biblioteki TPL nie jest konieczna.

Spójrzmy na przykład widoczny na listingu 1.1, który przedstawia metodę zdarzeniową przycisku. Zdefiniowana jest w niej przykładową akcja pobierająca obiekt typu `object` a zwracająca liczbę całkowitą `long`. Referencję do niej zapisuję w zmiennej `akcja` i uruchamiam ją (synchronicznie). Czynność owa wprowadza jednosekundowe opóźnienie metodą `Thread.Sleep` (należy zadeklarować użycie przestrzeni nazw `System.Threading`[1]), które oczywiście opóźnia wykonywanie całej metody zdarzeniowej po kliknięciu przycisku. W efekcie na jedną sekundę aplikacja zamiera.

Listing 1.1. Synchroniczne wykonywanie kodu zawartego w akcji

```
private void button1_Click(object sender, EventArgs e)
{
    Func<object, long> akcja =
        (object argument) =>
        {
            msgBox("Akcja: Początek, argument: " + argument.ToString());
            Thread.Sleep(1000); //opóźnienie
            msgBox("Akcja: Koniec");
            return DateTime.Now.Ticks;
        };

    msgBox("button1_Click: Początek");
    msgBox("Wynik: "+akcja("synchronicznie"));
    msgBox("button1_Click: Koniec");
}

void msgBox(string komunikat)
{
    string taskID = Task.CurrentId.HasValue ? Task.CurrentId.ToString() : "UI";
    MessageBox.Show("! " + komunikat + " (" + taskID + ")");
}
```

W metodzie przedstawionej na listingu 1.2 ta sama akcja wykonywana jest asynchronicznie w osobnym wątku utworzonym przez platformę .NET na potrzeby zdefiniowanego przez nas zadania (instancja klasy `Task` z TPL). Synchronizacja następuje w momencie odczytania wartości `zadanie.Result`, czyli tak naprawdę wartości zwracanej przez czynność `akcja`. Jej sekcja `get` czeka ze zwróceniem wartości aż do zakończenia akcji wykonywanej przez zadanie wstrzymując do tego czasu wątek, w którym wykonywana jest metoda `button1_Click`. Jest to zatem typowy punkt synchronizacji, choć trochę ukryty. Zwróćmy uwagę, że po

[1] Alternatywnie moglibyśmy użyć instrukcji `await Task.Delay(1000);`, ale wówczas musielibyśmy oznaczyć wyrażenie lambda jako async, a wówczas należałoby referencję do niego zapisać w zmiennej typu `Func<object, Task<long>>`.

instrukcji `zadanie.Start()`, a przed odczytaniem własności `zadanie.Result` mogą być wykonywane dowolne czynności, o ile są niezależne od wartości zwróconej przez zadanie.

Listing 1.2. Użycie zadania do asynchronicznego wykonania kodu

```
private void button1_Click(object sender, EventArgs e)
{
    Func<object, long> akcja =
        (object argument) =>
        {
            msgBox("Akcja: Początek, argument: " + argument.ToString());
            Thread.Sleep(1000); //opóźnienie
            msgBox("Akcja: Koniec");
            return DateTime.Now.Ticks;
        };

    Task<long> zadanie = new Task<long>(akcja, "zadanie");
    zadanie.Start();
    msgBox("Akcja została uruchomiona");
    if (zadanie.Status != TaskStatus.Running &&
        zadanie.Status!=TaskStatus.RanToCompletion)
            msgBox("Zadanie nie zostało uruchomione");
    else msgBox("Wynik: "+zadanie.Result);
    msgBox("button1_Click: Koniec");
}
```

Nie jest konieczne, aby instrukcja odczytania własności `Result` znajdowała się w tej samej metodzie, co uruchomienie 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. Przykład takiej metody widoczny jest na listingu 1.3. Jeżeli używamy angielskich nazw metod, jest zwyczajem, aby metoda tworząca i uruchamiająca zadanie miały przyrostek `..Async`.

Listing 1.3. Wzór metody wykonującej jakąś czynność asynchronicznie

```
Task<long> DoSomethingAsync(object argument)
{
    Func<object, long> akcja =
        (object _argument) =>
        {
            msgBox("Akcja: Początek, argument: " + _argument.ToString());
            Thread.Sleep(1000); //opóźnienie
            msgBox("Akcja: Koniec");
            return DateTime.Now.Ticks;
        };

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

protected void button1_Click(object sender, EventArgs e)
```

```
{
    msgBox("button1_Click: Początek");
    Task<long> zadanie = DoSomethingAsync("zadanie-metoda");
    msgBox("Akcja została uruchomiona");
    if (zadanie.Status != TaskStatus.Running &&
        zadanie.Status!=TaskStatus.RanToCompletion)
            msgBox("Zadanie nie zostało uruchomione");
    else msgBox("Wynik: " + zadanie.Result);
    msgBox("button1_Click: Koniec");
}
```

Po tym wprowadzeniu możemy przejść do omówienia zasadniczego tematu. Wraz z wersjami 4.0 i 4.5 w platformie .NET (oraz w platformie Windows Runtime) pojawiło się wiele metod podobnych do przedstawionej powyżej metody `DoSomethingAsync` (ale oczywiście w odróżnieniu od niej robiących coś pożytecznego). Metody te wykonują asynchronicznie różnego typu długotrwałe czynności. Znajdziemy je w klasie `HttpClient`, w klasach odpowiedzialnych za obsługę plików (`StorageFile`, `StremWriter`, `StreamReader`, `XmlReader`), w klasach odpowiedzialnych za kodowanie i dekodowanie obrazów, czy w klasach WCF. Asynchroniczność jest wręcz standardem w aplikacjach Windows 8 z interfejsem Modern UI. I właśnie aby ich użycie było (prawie) tak proste, jak metod synchronicznych wprowadzony został w C# 5.0 (co odpowiada platformie .NET 4.5) 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`. Prezentuje to listing 1.4.

Listing 1.4. Przykład użycia modyfikatora async i modyfikatora await

```
protected async void button1_Click(object sender, EventArgs e)
{
    msgBox("button1_Click: Początek");
    Task<long> zadanie = DoSomethingAsync("async/await");
    msgBox("Akcja została uruchomiona");
    long wynik = await zadanie;
    msgBox("Wynik: " + wynik);
    msgBox("button1_Click: Koniec");
}
```

Operator `await` zwraca parametr użyty w klasie parametrycznej `Task<>`. Zatem w przypadku zadania typu `Task<long>` będzie to zmienna typu `long`. Jeżeli użyta została wersje nieparametryczna klasy `Task`, operator zwraca `void` i służy jedynie do synchronizacji (nie przekazuje wyniku; nieparametryczna klasa `Task` nie ma także własności `Result`).

Metody oznaczone modyfikatorem `async` nazywane są w angielskiej dokumentacji MSDN *async method*. Może to jednak wprowadzać pewne zamieszanie. Z powodu tej nazwy metody z modyfikatorem `async` (w naszym przypadku metoda `Button1_Click`) utożsamiane są bowiem z metodami wykonującymi asynchronicznie jakieś czynności (a taką w naszym przypadku jest `DoSomethingAsync`). Osobom poznającym dopiero temat często wydaje się, że aby metoda wykonywana była asynchronicznie wystarczy dodać do jej sygnatury modyfikator `async`. To nie jest prawda!

Możemy wywołać metodę `DoSomethingAsync` w taki sposób, że umieścimy ją bezpośrednio za operatorem `await` np. `long wynik = await DoSomethingAsync("async/await");`. Jaki to ma sens? Wykonywanie metody `button1_Click`, w której znajduje się to wywołanie zostanie wstrzymane aż do momentu zakończenia metody `DoSomethingAsync`, więc efekt jaki zobaczymy na ekranie będzie identyczny, jak w przypadku synchronicznym (listing 1.1). Różnica jest jednak zasadnicza, i to jest zasadnicza nowość, bowiem instrukcja zawierająca operator `await` nie blokuje wątku, w którym wywołana została metoda `button1_Click`. Kompilator zawiesza wywołanie metody `button1_Click` przechodząc do kolejnych czynności w miejscu jej wywołania aż do momentu zakończenia uruchomionego zadania. W momencie, gdy to nastąpi, wątek wraca do

metody `button1_Click` i kontynuuje jej działanie[2]. W programie, na którym w tej chwili testujemy operator `await`, efektów tego jednak nie zobaczymy. Efekt będzie widoczny dopiero, gdy metodę `button1_Click` wywołamy z innej metody – niech będzie to metoda zdarzeniowa `button2_Click` związana z drugim przyciskiem. Zauważmy, że w serii instrukcji wywołanie metody oznaczonej modyfikatorem `async` nie musi się zakończyć przed wykonaniem następnej instrukcji – i w tym sensie jest ona asynchroniczna. Aby tak się stało musi w niej jednak zadziałać operator `await` czekający na wykonanie jakiegoś zadania (w naszym przykładzie metody `DoSomethingAsync`). W efekcie, w scenariuszu przedstawionym na listingu 1.5 metoda `button2_Click` zakończy się przed zakończeniem `button1_Click`.

Listing 1.5. Działanie modyfikatora async

```
private async void button1_Click(object sender, EventArgs e)
{
    msgBox("button1_Click: Początek");
    long wynik = await DoSomethingAsync("async/await");
    msgBox("Wynik: " + wynik.ToString());
    msgBox("button1_Click: Koniec");
}

private void button2_Click(object sender, EventArgs e)
{
    msgBox("button2_Click: Początek");
    button1_Click(null, null);
    msgBox("button2_Click: Koniec");
}
```

Ważna rzecz: samo użycie operatora `await` i modyfikatora `async` nie powoduje utworzenia nowych zadań lub wątków! Powoduje jedynie przekazanie na pewien czas sterowania z metody, w której znajduje się operator `await` i oznaczonej modyfikatorem `async` do metody, która ją wywołała i powrót w momencie ukończenia zadania, na które czeka `await`. Koszt jest zatem niewielki i rozwiązanie to może być z powodzeniem stosowane bez obawy o utratę wydajności. Ponadto, właśnie z uwagi na wydajność, operator `await` sprawdza czy w momencie, w którym dociera do niego sterowanie, metoda asynchroniczna nie jest już zakończona. Jeżeli tak, praca kontynuowana jest synchronicznie bez zbędnych skoków.

Metoda z modyfikatorem `async` może zwracać wartość `void` – tak, jak w przypadku przedstawionej wyżej metody zdarzeniowej `button1_Click`. Jednak w takim przypadku jej działanie nie może być żaden sposób synchronizowane. Po uruchomieniu nie mamy nad nią żadnej kontroli. W szczególności nie można użyć operatora `await`, ani metody `Wait` klasy `Task`, aby poczekać na jej zakończenie. Aby to było możliwe metoda z modyfikatorem `async` musi zwracać referencję `Task` lub `Task<>`. Wówczas możliwe jest użycie operatora `await`, za którym można zresztą ustawić dowolne wyrażenie o wartości `Task` lub `Task<>` (zmienne i własności tego typu oraz metody lub wyrażenia lambda zwracające wartość tego typu[3]). Przekazane zadanie umożliwia synchronizację. Ponadto użycie wersji parametrycznej umożliwia zwrócenie wartości przekazywanej potem przez operator `await`.

Sprawdźmy to tworząc odpowiednik metody `button1_Click` ze zmienioną sygnaturą (nie możemy tego zrobić z oryginałem, bo jest związany ze zdarzeniem `button1.Click`). Nowa metoda o nazwie

---

[2] Aby taki efekt uzyskać bez operatora `await`, należałoby użyć konstrukcji opartej na funkcjach zwrotnych (ang. *callback*). W efekcie kod stałby się raczej skomplikowany i przez to podatny na błędy. Warto też zauważyć, że `await` nie jest prostym odpowiednikiem metody `Task.Wait`, która po prostu zatrzymałaby bieżący wątek do momentu zakończenia zadania. W przypadku operaotra `await` nastąpi przekazanie sterowania do metody wywołującej i powrót w momencie zakończenia zadania.

[3] Gwoli prawy, należałoby to stwierdzenie uściślić, bowiem nie tylko zadania mogą być argumentem operatora `await`, a każdy typ, który zwraca metodę `GetAwaiter`. Więcej informacji dostępnych jest na stronie FAQ zespołu odpowiedzialnego za implementację mechanizmu `async`/`await` w platformie .NET (*http://blogs.msdn.com/b/pfxteam/archive/2012/04/12/10293335.aspx*).

`DoSomethingMoreAsync` widoczna jest na listingu 1.6[4]. Usunąłem argumenty, których i tak nie używaliśmy i zmieniłem zwracaną wartość z `void` na `Task`. Dzięki temu metoda ta nie jest już typu „wystrzel i zapomnij", a może być kontrolowana z miejsca uruchomienia (zob. widoczna również na listingu 1.6 metoda `button2_Click`). Zdziwienie może budzić jednak fakt, że za słowem kluczowym `return` w metodzie `DoSomethingMoreAsync` wcale nie ma instrukcji tworzącej zwracane przez tą metodę zadanie (tak naprawdę instrukcji `return` mogłoby wcale nie być). W metodach z modyfikatorem `async` i zwracających wartość `Task`, zadanie jest przypisywane przez kompilator. W ten sposób ułatwiona jest wielostopniowa obsługa metod asynchronicznych. Należy jednak pamiętać, że te metody nie tworzą nowych zadań, a jedynie je przekazują.

Listing 1.6. Metoda async zwracająca zadanie

```
private async Task DoSomethingMoreAsync()
{
    msgBox("DoSomethingMoreAsync: Początek");
    long wynik = await DoSomethingAsync("async/await");
    msgBox("DoSomethingMoreAsync: Wynik: " + wynik.ToString());
    msgBox("DoSomethingMoreAsync: Koniec");
    return;
}

private async void button2_Click(object sender, EventArgs e)
{
    msgBox("button2_Click: Początek");
    await DoSomethingMoreAsync();
    msgBox("button2_Click: Koniec");
}
```

A co w przypadku metod *async*, które miałyby zwracać wartość. Załóżmy, że metoda `DoSomethingMore` miałaby zwracać wartość typu `long` (np. wartość zmiennej `wynik`). W takim przypadku należy zmienić typ tej metody na `Task<long>`, a za słowem kluczowym `return` wstawić wartość typu `long`. Pokazuje to listing 1.7. Warto zapamiętać, choć to uproszczone stwierdzenie, że w metodach *async* operator `await` wyłuskuje z typu `Task<>` parametr, a słowo kluczowe `return` w metodach `async` zwracające wartość typu `Task<>` działa odwrotnie – otacza dowolne obiekty typem `Task<>`.

Listing 1.7. Metoda async zwracająca wartość long

```
private async Task<long> DoSomethingMoreAsync()
{
    msgBox("DoSomethingMoreAsync: Początek");
    long wynik = await DoSomethingAsync("async/await");
    msgBox("DoSomethingMoreAsync: Wynik: " + wynik.ToString());
    msgBox("DoSomethingMoreAsync: Koniec");
    return wynik;
}

private async void button2_Click(object sender, EventArgs e)
{
    msgBox("button2_Click: Początek");
    msgBox("button2_Click: Wynik: " + await DoSomethingMoreAsync());
    msgBox("button2_Click: Koniec");
}
```

[4] Zwróćmy uwagę na przyrostek „Async". W końcu jest to teraz metoda, która działa asynchronicznie, choć żadnego zadania nie tworzy.

I kolejna sprawa. Co w metodach *async* dzieje się w przypadku błędów? Nieobsłużone wyjątki zgłoszone w metodzie z modyfikatorem `async` i zwracające zadania (`Task` lub `Task<>`) są za pośrednictwem tych zadań przekazywane do metody wywołującej. Można zatem użyć normalnej konstrukcji `try..catch`, jak na listingu 1.8. Gorzej jest w przypadku metod *async* zwracających `void` (tych typu „wystrzel i zapomnij”, jak `button1_Click` z naszego przykładu). Wówczas wyjątek przekazywany jest do puli wątków kryjącej się za mechanizmem zadań i przechwytywanie wyjątków nic nie da.

Listing 1.8. Obsługa wyjątków zgłaszanych przez metody async

```
private async void button2_Click(object sender, EventArgs e)
{
    msgBox("button2_Click: Początek");
    try
    {
        msgBox("button2_Click: Wynik: " + await DoSomethingMoreAsync());
    }
    catch(Exception exc)
    {
        msgBox("button2_Click: Błąd!\n" + exc.Message);
    }
    msgBox("button2_Click: Koniec");
}
```