Najlepszy podręcznik poświęcony C#!

Programowanie C# 5.0

Tworzenie aplikacji Windows 8, internetowych oraz biurowych w .NET 4.5 Framework

HELION

O'REILLY®

Ian Griffiths

Tytuł oryginału: Programming C# 5.0

Tłumaczenie: Piotr Rajca

ISBN: 978-83-246-6984-4

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/csh5pr
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/csh5pr.zip

Printed in Poland.

Spis treści

ROZDZIAŁ 18.

Asynchroniczne cechy języka

Podstawową nowością wprowadzoną w C# 5.0 jest wsparcie języka dla stosowania i implementacji metod asynchronicznych. Metody asynchroniczne są niejednokrotnie najbardziej wydajnym sposobem korzystania z niektórych usług. Na przykład większość operacji wejścia-wyjścia jest wykonywana asynchronicznie przez jądro systemu operacyjnego, gdyż większość urządzeń peryferyjnych, takich jak kontrolery dysków lub karty sieciowe, jest w stanie wykonywać większość operacji autonomicznie. Wymagają użycia procesora wyłącznie podczas rozpoczynania i zakańczania operacji.

Choć wiele usług dostarczanych przez system Windows ma w rzeczywistości asynchroniczny charakter, to jednak programiści często decydują się na korzystanie z nich przy użyciu metod synchronicznych (czyli takich, które kończą się przed wykonaniem tego, co miały zrobić). Jednak takie postępowanie jest marnowaniem zasobów, gdyż powoduje ono zablokowanie wątku aż do momentu zakończenia operacji wejścia-wyjścia. W systemie Windows wątki są cennym zasobem, dlatego też uzyskuje on najwyższą wydajność, gdy liczba działających w nim wątków systemowych jest stosunkowo niewielka. W idealnym przypadku liczba wątków systemowych będzie odpowiadać liczbie wątków sprzętowych, lecz jest to przypadek optymalny, wyłącznie jeśli możemy zagwarantować, że wątki będą blokowane tylko w sytuacjach, gdy nie mają żadnych innych prac do wykonania. (Różnice pomiędzy wątkami systemowymi oraz wątkami sprzętowymi zostały wyjaśnione w rozdziale 17.) Im więcej wątków będzie blokowanych w wywołaniach metod synchronicznych, tym więcej będziemy potrzebowali wątków do obsługi obciążenia, a to z kolei prowadzi do ograniczenia wydajności. Dlatego też w kodzie, w którym wydajność działania odgrywa bardzo dużą rolę, metody asynchroniczne są użyteczne, gdyż zamiast marnować zasoby poprzez zmuszanie wątku do oczekiwania na zakończenie operacji wejścia-wyjścia, wątek może zainicjować taką operację, a następnie w międzyczasie zająć się czymś innym.

Jednak problem z metodami asynchronicznymi polega na tym, że ich stosowanie jest znacząco bardziej złożone od korzystania z metod synchronicznych, zwłaszcza kiedy w grę wchodzi koordynacja wielu powiązanych ze sobą operacji oraz obsługa błędów. To właśnie z tego powodu programiści bardzo często wybierają mniej wydajne, synchroniczne rozwiązania. Jednak nowe, asynchroniczne możliwości języka C# 5.0 pozwalają na tworzenie kodu, który może korzystać z wydajnych, asynchronicznych API, zachowując przy tym jednocześnie znaczną część prostoty cechującej kod używający prostszych rozwiązań synchronicznych.

Nowe możliwości języka przydają się także w niektórych przypadkach, gdy głównym celem zapewnienia wydajności działania nie jest maksymalizacja przepustowości. W przypadku kodu aplikacji klienckich bardzo ważnym zagadnieniem jest unikanie blokowania wątku obsługi interfejsu użytkownika, a jednym z rozwiązań jest stosowanie metod asynchronicznych. Wsparcie dla kodu asynchronicznego, jakie zapewnia C#, jest w stanie obsługiwać problemy związane z powinowactwem do wątku, co w ogromnym stopniu ułatwia tworzenie kodu obsługi interfejsu użytkownika zapewniającego błyskawiczną reakcję na poczynania użytkownika aplikacji.

Nowe słowa kluczowe: async oraz await

C# udostępnia wsparcie dla programowania asynchronicznego, wprowadzając dwa słowa kluczowe: `async` oraz `await`. Pierwsze z nich nie jest przeznaczone do samodzielnego użycia. Umieszcza się je natomiast w deklaracjach metod, a jego zadaniem jest poinformowanie kompilatora, że w metodzie będą używane możliwości asynchroniczne. Jeśli słowo to nie zostanie umieszczone w deklaracji metody, to nie będzie jej można używać wraz ze słowem kluczowym `await`. Jest to prawdopodobnie nieco nadmiarowe — kompilator zgłasza błąd, jeśli spróbujemy użyć słowa kluczowego `await` bez `async`, czyli najwyraźniej jest w stanie określić, czy ciało metody próbuje korzystać z możliwości asynchronicznych. A zatem dlaczego musimy jawnie deklarować asynchroniczność metody? Otóż wynika to z dwóch powodów. Przede wszystkim, jak się niebawem przekonasz, te możliwości drastycznie zmieniają zachowanie kodu generowanego przez kompilator, dlatego też stanowi to wyraźny sygnał informujący wszystkie osoby przeglądające kod, że metoda działa w sposób asynchroniczny. A poza tym słowo `await` nie zawsze było słowem kluczowym języka C#, zatem wcześniej nic nie stało na przeszkodzie, by używać go jako identyfikatora. Być może firma Microsoft mogła zaprojektować gramatykę słowa `await` w taki sposób, by było ono traktowane jako słowo kluczowe wyłącznie w bardzo specyficznych kontekstach, dzięki czemu we wszystkich innych przypadkach mogłoby wciąż być traktowane jako zwyczajny identyfikator. Niemniej jednak twórcy języka C# zdecydowali się zastosować nieco bardziej ogólne podejście: otóż słowa `await` nie można używać jako identyfikatora wewnątrz metod, w których deklaracji zastosowano modyfikator `async`, natomiast we wszystkich pozostałych miejscach kodu może ono służyć za identyfikator.

Słowo kluczowe `async` nie zmienia sygnatury metody. Determinuje ono sposób kompilacji metody, a nie jej używania.

A zatem modyfikator `async` jedynie deklaruje chęć używania słowa kluczowego `await`. (Choć nie wolno nam używać słowa kluczowego `await` bez `async`, to jednak nie jest błędem umieszczanie modyfikatora `async` w deklaracji metody, która nie wykorzystuje słowa kluczowego `await`. Niemniej jednak takie rozwiązanie nie ma żadnego sensu, dlatego też jeśli wystąpi, kompilator wygeneruje ostrzeżenie). Listing 18.1 przedstawia dosyć typowy przykład metody asynchronicznej. Używa ona klasy `HttpClient`[1], by poprosić jedynie o nagłówki konkretnego zasobu (używając w tym celu standardowego żądania `HEAD`, które istnieje w protokole HTTP

[1] Została ona tutaj zastosowana zamiast prostszej klasy `WebClient`, której używaliśmy w przykładach przedstawianych w poprzednich rozdziałach, gdyż zapewnia większą kontrolę nad szczegółami wykorzystania protokołu HTTP.

właśnie do tego celu). Uzyskane wynik są następnie wyświetlane w polu tekstowym stanowiącym element interfejsu użytkownika aplikacji — metoda ta stanowi fragment kodu ukrytego, obsługującego interfejs użytkownika aplikacji, który zawiera pole `TextBox` o nazwie `headerListTextBox`.

Listing 18.1. Stosowanie słów kluczowych async i await podczas pobierania nagłówków HTTP

```
private async void FetchAndShowHeaders(string url)
{
    using (var w = new HttpClient())
    {
        var req = new HttpRequestMessage(HttpMethod.Head, url);
        HttpResponseMessage response =
            await w.SendAsync(req, HttpCompletionOption.ResponseHeadersRead);

        var headerStrings =
            from header in response.Headers
            select header.Key + ": " + string.Join(",", header.Value);

        string headerList = string.Join(Environment.NewLine, headerStrings);
        headerListTextBox.Text = headerList;
    }
}
```

Powyższy kod zawiera jedno wyrażenie używające słowa kluczowego `await`, które zostało wyróżnione pogrubioną czcionką. Słowo to jest używane w wyrażeniach, które mogą być wykonywane przez dłuższy czas, zanim zwrócą wynik; oznacza ono, że dalsza część metody nie powinna być wykonana, dopóki operacja się nie zakończy. Wygląda to zatem jak zwyczajny, blokujący kod synchroniczny, jednak różnica polega na tym, że słowo kluczowe `await` nie powoduje zablokowania wątku.

Gdybyśmy chcieli zablokować wątek i poczekać na wyniki, to nic nie stoi na przeszkodzie, by to zrobić. Metoda `SendAsync` klasy `HttpClient` zwraca obiekt `Task<HttpResponseMessage>`, więc można by zastąpić wyrażenie z listingu 18.1 używające słowa kluczowego `await` wyrażeniem przedstawiony na listingu 18.2. Pobiera ono wartość właściwości `Result` zadania, a zgodnie z tym, czego dowiedzieliśmy się w rozdziale 17., jeśli zadanie nie zostało zakończone, to próba odczytu tej właściwości spowoduje zablokowanie wątku do czasu wygenerowania wyników (bądź do momentu, gdy zadanie zakończy się niepowodzeniem, lecz w takim przypadku wyrażenie zgłosi wyjątek).

Listing 18.2. Blokujący odpowiednik wyrażenia ze słowem kluczowym await

```
HttpResponseMessage response =
    w.SendAsync(req, HttpCompletionOption.ResponseHeadersRead).Result;
```

Choć wyrażenie `await` zastosowane w kodzie z listingu 18.1 robi coś, co jest pozornie podobne do powyższej instrukcji, to jednak działa zupełnie inaczej. Jeśli wynik zadania nie będzie dostępny od razu, to niezależnie od tego, co sugeruje jego nazwa, słowo kluczowe `await` sprawi, że wątek będzie czekał. Zamiast tego spowoduje ono zakończenie wykonywanej metody. Można użyć debugera, by przekonać się, że wywołanie metody `FetchAdnShowHeaders` kończy się natychmiast. Na przykład: jeśli wywołamy tę metodę w procedurze obsługi kliknięcia przycisku, przedstawionej na listingu 18.3, to możemy ustawić jeden punkt przerwania w wierszu zawierającym wywołanie `Debug.WriteLine`, oraz drugi w kodzie z listingu 18.1, w wierszu zawierającym instrukcję aktualizującą wartość właściwości `headerListTextBox.Text`.

Listing 18.3. Wywoływanie metody asynchronicznej

```
private void fetchHeadersButton_Click(object sender, RoutedEventArgs e)
{
    FetchAndShowHeaders("http://helion.pl/");
    Debug.WriteLine("Wywołanie metody zostało zakończone.");
}
```

Jeśli uruchomimy taki program w debugerze, przekonamy się, że najpierw zatrzymamy się w punkcie przerwania umieszczonym w wierszu z listingu 18.3, a dopiero później w punkcie przerwania z listingu 18.1. Innymi słowy, fragment kodu z listingu 18.1 umieszczony za wyrażeniem ze słowem kluczowym `await` zostaje wykonany *po* tym, gdy sterowanie zostanie przekazane z metody do kodu, który ją wywołał. Najwyraźniej kompilator jakoś zmienia dalszą część metody w taki sposób, aby została wykonana przy użyciu wywołania zwrotnego, realizowanego po zakończeniu operacji asynchronicznej.

Debuger Visual Studio stosuje różne sztuczki podczas debugowania metod asynchronicznych, aby zapewnić nam możliwość analizowania ich krok po kroku jak normalnych metod. Zazwyczaj jest to całkiem przydatne, jednak czasami ukrywa prawdziwy przebieg realizacji programu. Opisany powyżej przykład został uważnie zaprojektowany w taki sposób, aby przekreślić starania Visual Studio i pokazać faktyczny sposób realizacji kodu.

Warto zauważyć, że kod z listingu 18.1 oczekuje, że będzie wykonywany w wątku obsługi interfejsu użytkownika, gdyż pod koniec metody modyfikuje wartość właściwości `Text` pola tekstowego. Metody asynchroniczne nie dają gwarancji, że powiadomienia o zakończeniu operacji będą generowane w tym samym wątku, w którym operacja została rozpoczęta — w większości przypadków będą one generowane w innych wątkach. Pomimo to kod z listingu 18.1 działa zgodnie z zamierzeniami. Oznacza to, że słowo kluczowe `await` nie tylko spowodowało przeniesienie połowy kodu metody do wywołania zwrotnego, lecz także zadbało za nas o prawidłową obsługę powinowactwa do wątku.

To wszystko wyraźnie pokazuje, że użycie słowa kluczowego `await` zmusza kompilator do przeprowadzenia drastycznych zmian w naszym kodzie. W C# 4.0, chcąc użyć tego asynchronicznego API, a następnie zaktualizować interfejs użytkownika, konieczne było zastosowanie kodu podobnego do tego z listingu 18.4. Wykorzystuje on technikę opisaną w rozdziale 17.: przygotowuje kontynuację dla zadania zwróconego przez metodę `SendAsync`, wykorzystując przy tym obiekt `TaskScheduler`, by zapewnić, że kod kontynuacji zostanie wykonany w wątku obsługi interfejsu użytkownika.

Listing 18.4. Samodzielne tworzenie odpowiednika metody asynchronicznej

```
private void OldSchoolFetchHeaders(string url)
{
    var w = new HttpClient();
    var req = new HttpRequestMessage(HttpMethod.Head, url);

    var uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
    w.SendAsync(req, HttpCompletionOption.ResponseHeadersRead)
        .ContinueWith(sendTask =>
        {
            try
            {
                HttpResponseMessage response = sendTask.Result;
```

```
                var headerStrings =
                    from header in response.Headers
                    select header.Key + ": " + string.Join(",", header.Value);

                string headerList =
                    string.Join(Environment.NewLine, headerStrings);
                headerListTextBox.Text = headerList;
            }
            finally
            {
                w.Dispose();
            }
        },
        uiScheduler);
}
```

Jest to przykład bardzo dobrego, bezpośredniego wykorzystania TPL i zapewnia podobne efekty jak kod z listingu 18.1, choć nie stanowi on dokładnej reprezentacji sposobu, w jaki kompilator C# przekształca kod. Jak dowiesz się z dalszej części rozdziału, słowo kluczowe `await` używa wzorca, który jest obsługiwany przez klasy `Task` oraz `Task<T>`, lecz który ich nie wymaga. Dodatkowo gwarantuje ono wygenerowanie kodu, który obsługuje wcześniejsze zakończenie (czyli sytuacje, gdy zadanie zostanie wykonane, zanim będziemy gotowi rozpocząć oczekiwanie na jego zakończenie) znacznie bardziej efektywnie niż kod z listingu 18.4. Jednak zanim poznasz wszelkie szczegóły tego, co robi kompilator, warto się dowiedzieć, jakie problemy kompilator za nas rozwiązuje — a to najlepiej zrobić, pokazując kod, który musielibyśmy napisać w C# 4.0.

Nasz aktualny przykład jest całkiem prosty, gdyż realizuje tylko jedną asynchroniczną operację; jednak oprócz dwóch opisanych wcześniej czynności — czyli utworzenia jakiegoś wywołania zwrotnego obsługującego zakończenie oraz zapewnienia, że zostanie ono wykonane w odpowiednim wątku — musimy także zadbać o odpowiednią obsługę instrukcji `using` zastosowanej w kodzie z listingu 18.1. Kod z listingu 18.4 nie może używać instrukcji `using`, gdyż obiekt `HttpClient` chcemy zwolnić dopiero w momencie, gdy nie będzie już nam potrzebny. Wywołanie metody `Dispose` tuż przed zakończeniem metody zewnętrznej nie zda egzaminu, gdyż musimy mieć możliwość użycia obiektu w kodzie kontynuacji, a to zazwyczaj nastąpi trochę po zakończeniu metody. A zatem musimy utworzyć obiekt w jednej metodzie (zewnętrznej) i zwolnić go w innej (wewnętrznej). A ponieważ sami wywołujemy przy tym metodę `Dispose`, zatem sami musimy zadbać o obsługę wyjątków. Dlatego też konieczne było umieszczenie całego kodu przeniesionego do metody zwrotnej w bloku `try` i wywołanie metody `Dispose` w bloku `finally`. (W rzeczywistości zastosowane rozwiązanie nie jest kompletne i niezawodne — gdyby konstruktor klasy `HttpRequestMessage` lub metoda pobierająca mechanizm szeregowania zadań, co jest raczej mało prawdopodobne, zgłosiły wyjątek, to używany obiekt `HttpClient` nie zostałby prawidłowo zwolniony. Innymi słowy, nasz kod obsługuje jedynie tę sytuację, gdy problemy pojawią się w samej operacji sieciowej).

Kod z listingu 18.4 używa mechanizmu szeregowania zadań, by wykonać kontynuację przy wykorzystaniu obiektu `SynchronizationContext`, aktywnego w momencie rozpoczynania operacji. Dzięki temu zapewniamy, że wywołanie zwrotne zostanie wykonane w wątku umożliwiającym aktualizację interfejsu użytkownika. Choć to w zupełności wystarcza do zapewnienia poprawnego działania naszego przykładu, to jednak słowo kluczowe `await` robi dla nas nieco więcej.

Konteksty wykonania i synchronizacji

Jeśli realizacja kodu dociera do wyrażenia zawierającego słowo kluczowe `await` oraz operację, której wykonanie nie zakończyło się od razu, to wygenerowany przez kompilator kod reprezentujący `await` zapewni pobranie aktualnego kontekstu wykonania. (Może się zdarzyć, że nie będzie to wymagało wielkiego zachodu — jeśli nie jest to pierwszy blok `await` w danej metodzie oraz jeśli używany kontekst nie został zmieniony, to będzie on już pobrany). Po zakończeniu operacji asynchronicznej dalsza część kodu metody zostanie wykonana przy wykorzystaniu kontekstu wykonania[2].

Zgodnie z informacjami podanymi w rozdziale 17., kontekst wykonania obsługuje pewne kontekstowe informacje o bezpieczeństwie oraz lokalny stan wątku, które muszą być przekazywane, gdy jedna metoda wywołuje drugą (i to nawet jeśli robi to bezpośrednio). Niemniej jednak istnieje jeszcze inny rodzaj kontekstu, który może nas interesować, a zwłaszcza jeśli tworzymy kod obsługi interfejsu użytkownika; chodzi o kontekst synchronizacji.

Choć wszystkie wyrażenia `await` pobierają kontekst wykonania, to decyzja o tym, czy wraz z nim należy pobrać także kontekst synchronizacji, zależy od typu, na który oczekujemy. Jeśli oczekujemy na daną typu `Task`, to domyślnie kontekst synchronizacji także zostanie pobrany. Zadania nie są jedynymi obiektami, na jakie można oczekiwać, informacje dotyczące sposobu, w jaki można dostosować typy do obsługi słowa kluczowego `await`, zostały podane w dalszej części rozdziału, w punkcie pt. „Wzorzec słowa kluczowego await".

Czasami mogą się zdarzyć sytuacje, w których nie będziemy chcieli używać kontekstu synchronizacji. Jeśli chcemy wykonać jakąś operację asynchroniczną, rozpoczynając ją w wątku obsługi interfejsu użytkownika, a jednocześnie nie ma konieczności dalszego pozostawania w tym wątku, to planowanie wykonania wszystkich kontynuacji przy użyciu kontekstu synchronizacji będzie jedynie niepotrzebnym obciążeniem. Jeśli operacja asynchroniczna jest reprezentowana przez obiekt `Task` lub `Task<T>`, to używając zdefiniowanej w tych klasach metody `ConfigureAwait` możemy zadeklarować, że nie chcemy używać kontekstu synchronizacji. W takim przypadku zwracana jest nieznacznie zmieniona reprezentacja operacji asynchronicznej, a jeśli jej użyjemy w wyrażeniu `await` zamiast oryginalnego zadania, to bieżący kontekst synchronizacji zostanie zignorowany (oczywiście o ile w ogóle będzie dostępny). (Nie można natomiast zrezygnować z wykorzystania kontekstu wykonania). Listing 18.5 pokazuje, jak można korzystać z metody `ConfigureAwait`.

Listing 18.5. Stosowanie metody ConfigureAwait

```
private async void OnFetchButtonClick(object sender, RoutedEventArgs e)
{
    using (var w = new HttpClient())
    using (Stream f = File.Create(fileTextBox.Text))
    {
        Task<Stream> getStreamTask = w.GetStreamAsync(urlTextBox.Text);
        Stream getStream = await getStreamTask.ConfigureAwait(false);

        Task copyTask = getStream.CopyToAsync(f);
        await copyTask.ConfigureAwait(false);
    }
}
```

[2] Okazuje się, że to samo dzieje się w przykładzie z listingu 18.4, gdyż TPL pobiera kontekst wykonywania za nas.

Powyższy kod reprezentuje procedurę obsługi kliknięć przycisku, dlatego też jest wykonywany w wątku obsługi interfejsu użytkownika. Pobiera on wartości właściwości `Text` kilku pól tekstowych, a następnie wykonuje pewną operację asynchroniczną — pobiera zawartość adresu URL i kopiuje pobrane dane do pliku. Po pobraniu zawartości dwóch właściwości `Text` powyższy kod nie używa już żadnych elementów interfejsu użytkownika, a zatem jeśli wykonanie operacji asynchronicznej trochę zajmuje, to nie będzie miało żadnego znaczenia, że jej pozostała część zostanie wykonana w innym wątku. Poprzez przekazanie wartości `false` w wywołaniu metody `ConfigureAwait` oraz poczekanie na zwróconą przez nie wartość informujemy TPL, że do zakończenia operacji może zostać wykorzystany dowolny wątek, przy czym w tym przypadku będzie to najprawdopodobniej jeden z wątków dostępnych w puli. Dzięki temu operacja będzie mogła zostać wykonana szybciej i bardziej efektywnie, gdyż nie będzie musiała bez potrzeby korzystać z wątku obsługi interfejsu użytkownika po każdym słowie kluczowym `await`.

Kod przedstawiony na listingu 18.1 zawiera tylko jedno wyrażenie ze słowem kluczowym `await`, lecz nawet ten kod trudno jest odtworzyć, wykorzystując klasyczny model programowania z użyciem TPL. Przykład z listingu 18.5 zawiera dwa takie wyrażenia, a odtworzenie sposobu jego działania bez pomocy `await` wymagałoby użycia dosyć rozbudowanego kodu, gdyż wyjątki mogłyby być zgłaszane przed pierwszym wyrażeniem `await`, po drugim wyrażeniu oraz pomiędzy nimi; oprócz tego w każdym z tych przypadków (jak również w sytuacji, gdy nie zostały zgłoszone żadne wyjątki) musielibyśmy zadbać o wywołanie metody `Dispose` w celu zwolnienia używanych obiektów `HttpClient` oraz `Stream`. Niemniej jednak sytuacja staje się znacząco bardziej skomplikowana, kiedy w grę zaczyna dodatkowo wchodzić sterowanie przepływem.

Wykonywanie wielu operacji i pętli

Załóżmy, że zamiast pobierać nagłówki lub kopiować zawartość odpowiedzi HTTP do pliku, chcemy tę zawartość przetworzyć. Jeśli jest ona bardzo duża, to pobranie jej jest operacją, która może wymagać wykonania wielu czasochłonnych kroków. Przykład przedstawiony na listingu 18.6 pobiera całą stronę WWW wiersz po wierszu.

Listing 18.6. Wykonywanie wielu operacji asynchronicznych

```
private async void FetchAndShowBody(string url)
{
    using (var w = new HttpClient())
    {
        Stream body = await w.GetStreamAsync(url);
        using (var bodyTextReader = new StreamReader(body))
        {
            while (!bodyTextReader.EndOfStream)
            {
                string line = await bodyTextReader.ReadLineAsync();
                headerListTextBox.AppendText(line);
                headerListTextBox.AppendText(Environment.NewLine);
                await Task.Delay(TimeSpan.FromMilliseconds(10));
            }
        }
    }
}
```

W powyższym kodzie zostały użyte trzy wyrażenia `await`. Pierwsze z nich powoduje wykonanie żądania HTTP GET, a operacja ta zakończy się w momencie odebrania pierwszej części odpowiedzi, choć w tym momencie odpowiedź może jeszcze nie być kompletna — może zawierać jeszcze kilka megabajtów danych, które trzeba będzie jeszcze przekazać. Powyższy przykład zakłada, że zawartość odpowiedzi będzie tekstowa, dlatego też przekazuje zwrócony obiekt `Stream` jako argument wywołania konstruktora strumienia `StreamReader`, który udostępnia bajty stanowiące zawartość strumienia jako tekst[3]. Następnie przykład używa metody `ReadLineAsync`, by wiersz po wierszu odczytywać zawartość odpowiedzi. Ponieważ dane są przesyłane fragmentami, zatem odczytanie pierwszego wiersza tekstu może trochę zająć, jednak kilka kolejnych wywołań metody zostanie zapewne wykonanych momentalnie, gdyż każdy odebrany pakiet sieciowy zazwyczaj zawiera więcej wierszy. Jeśli jednak nasz kod może odczytywać dane szybciej, niż są przesyłane siecią, to w końcu odczyta wszystkie wiersze, które były dostępne w pierwszym pakiecie, i pewnie minie trochę czasu, zanim pojawią się kolejne. Dlatego też wywołania metody `ReadLineAsync` będą zwracały zarówno zadania, których wykonanie zajmuje więcej czasu, jak i takie, które zostaną zakończone błyskawicznie. Trzecią operacją asynchroniczną jest wywołanie metody `Task.Delay`. W powyższym przykładzie została ona użyta po to, by nieco zwolnić odczyt danych i aby kolejne wiersze tekstu pojawiały się w interfejsie użytkownika stopniowo. Metoda `Task.Delay` zwraca obiekt `Task`, który zostanie zakończony po upływie określonego czasu; stanowi ona zatem asynchroniczny odpowiednik metody `Thread.Sleep`. (Metoda `Thread.Sleep` blokuje wątek, w którym została wywołana, natomiast wyrażenie `await Task.Delay` wprowadza opóźnienie bez blokowania wątku).

W powyższym przykładzie każde z wyrażeń `await` zostało umieszczone w odrębnej instrukcji; takie rozwiązanie nie jest jednak konieczne. Nic nie stoi na przeszkodzie, by użyć wyrażenia o następującej postaci: `(await t1) + (await t2)`. (W razie potrzeby można pominąć nawiasy, gdyż operator `await` ma wyższy priorytet niż operator dodawania, ja jednak preferuję wizualny porządek i hierarchię, jaką one zapewniają).

Nie przedstawię tu pełnego odpowiednika kodu z listingu 18.6, który należałoby napisać w języku C# 4.0, gdyż jest on zbyt duży. Ograniczę się jedynie do przedstawienia kilku problemów. Przede wszystkim w powyższym kodzie używamy pętli, wewnątrz której zostały umieszczone dwa wyrażenia `await`. Odtworzenie analogicznego kodu z użyciem obiektów `Task` i wywołań zwrotnych oznaczałoby konieczność stworzenia własnego odpowiednika pętli, gdyż jej zawartość musi zostać rozdzielona na trzy metody: pierwsza z nich rozpoczynałaby działanie pętli (byłaby ona metodą zagnieżdżoną, działającą jako kontynuacja metody `GetStreamAsync`), a pozostałe dwie byłyby wywołaniami zwrotnymi obsługującymi zakończenie operacji `ReadLineAsync` oraz `Task.Delay`. Takie rozwiązanie można by zaimplementować, tworząc metodę zagnieżdżoną służącą do rozpoczynania kolejnych iteracji i wywołując ją z dwóch miejsc: w miejscu, w którym chcemy rozpocząć działanie pętli, oraz w kontynuacji zadania `Task.Delay` w celu rozpoczęcia kolejnej iteracji pętli. Ta technika została zaprezentowana na listingu 18.7, choć przedstawia on tylko jeden aspekt działań, które wykonuje za nas kompilator — nie jest on kompletnym odpowiednikiem kodu z listingu 18.6.

[3] Precyzyjnie rzecz ujmując, powinniśmy sprawdzić nagłówki odpowiedzi HTTP, by określić użyty sposób kodowania i w odpowiedni sposób skonfigurować obiekt `StreamReader`. Jednak w tym przykładzie pozwalamy, by obiekt strumienia sam określił sposób kodowania, co na potrzeby przykładu powinno działać wystarczająco dobrze.

Listing 18.7. Niekompletna samodzielna implementacja pętli asynchronicznej

```
private void IncompleteOldSchoolFetchAndShowBody(string url)
{
    var w = new HttpClient();
    var uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
    w.GetStreamAsync(url).ContinueWith(getStreamTask =>
    {
        Stream body = getStreamTask.Result;
        var bodyTextReader = new StreamReader(body);

        Action startNextIteration = null;
        startNextIteration = () =>
        {
            if (!bodyTextReader.EndOfStream)
            {
                bodyTextReader.ReadLineAsync()
                    .ContinueWith(readLineTask =>
                {
                    string line = readLineTask.Result;

                    headerListTextBox.AppendText(line);
                    headerListTextBox.AppendText(Environment.NewLine);

                    Task.Delay(TimeSpan.FromMilliseconds(10))
                        .ContinueWith(delayTask =>
                            startNextIteration(), uiScheduler);
                },
                uiScheduler);
            }
        };
        startNextIteration();
    },
    uiScheduler);
}
```

Ten kod działa jako tako, jednak nawet nie podejmuje próby zwolnienia któregokolwiek z używanych zasobów. Występuje w nim kilka miejsc, w których potencjalnie może dojść do awarii, dlatego nie wystarczy umieść w kodzie jednej instrukcji `using` lub pary bloków `try/finally`, aby zabezpieczyć działanie kodu. A nawet bez tego dodatkowego utrudnienia działanie kodu ledwie można zrozumieć — wcale nie jest oczywiste, że próbuje on wykonać te same operacje co przykład z listingu 18.6. Po dodaniu odpowiedniej obsługi błędów ten kod byłby całkowicie niezrozumiały. W praktyce zapewne łatwiej by było zastosować całkowicie inne rozwiązanie, polegające na napisaniu klasy implementującej maszynę stanów i na tej podstawie określającej czynności, jakie ma wykonywać. Takie rozwiązanie zapewne ułatwiłoby napisanie prawidłowo działającego kodu, jednak wcale nie ułatwiłoby osobie analizującej kod zorientować się, że to, na co patrzy, jest w rzeczywistości niewiele więcej niż pętlą.

Nic zatem dziwnego, że tak wielu programistów preferuje stosowanie rozwiązań synchronicznych. Jednak C# 5.0 pozwala nam pisać kod asynchroniczny, który ma niemal taką samą strukturę co jego synchroniczny odpowiednik, bezboleśnie zapewniając nam przy tym wszystkie korzyści związane z większą wydajnością działania i sprawnym reagowaniem na poczynania użytkownika. Najprościej rzecz ujmując, właśnie te korzyści zapewniają nam słowa kluczowe `async` oraz `await`.

Każda metoda wykorzystująca słowo kluczowe `await` sama będzie wykonywana przez jakiś określony czas. A zatem oprócz korzystania z asynchronicznych API możemy uznać za stosowne, by stworzyć dla niej jakąś asynchroniczną reprezentację. Oba przedstawione słowa kluczowe pomagają nam to zrobić.

Zwracanie obiektu Task

Kompilator C# narzuca pewne ograniczenia na typy wartości wynikowych, które mogą zwracać metody oznaczone modyfikatorem `async`. Jak już się dowiedzieliśmy, mogą one zwracać `void`, jednak oprócz tego istnieją dwie inne możliwości: można zwracać instancję typu `Task` bądź typu `Task<T>`, gdzie `T` jest dowolnym typem. Dzięki temu kod wywołujący naszą asynchroniczną metodę może uzyskiwać informacje o statusie wykonywanych przez nią prac, a oprócz tego dysponuje możliwością dołączania do niej kontynuacji, a także pobierania wyniku (jeśli zwracany jest obiekt `Task<T>`). Oznacza to oczywiście, że jeśli nasza metoda jest wywoływana wewnątrz innej metody asynchronicznej (oznaczonej modyfikatorem `async`), to jej wynik będzie można pobrać, używając słowa kluczowego `await`.

Zwracanie zadań jest zazwyczaj bardziej preferowanym rozwiązaniem niż zwracanie typu `void`, gdyż w tym drugim przypadku kod wywołujący nie dysponuje tak naprawdę możliwością określenia, kiedy metoda została zakończona oraz czy należy zgłosić wyjątek. (Metody asynchroniczne mogą działać nawet po przekazaniu sterowania do kodu wywołującego — w końcu właśnie o to w nich chodzi — a zatem w momencie, kiedy nasza metoda zgłosi wyjątek, metody, która ją wywołała, może już w ogóle nie być na stosie). Zwracając obiekt `Task` lub `Task<T>`, zapewniamy kompilatorowi możliwość udostępniania wyjątków oraz w razie potrzeby zwracania wyników.

Oprócz ograniczenia nakazującego stosowanie modyfikatora `async` wyłącznie w metodach zwracających wynik typu `void`, `Task` bądź `Task<T>` nie można go także używać w metodzie stanowiącej punkt wejścia do programu, czyli w metodzie `Main`.

Zwrócenie zadania jest tak trywialnie proste, że nie ma żadnego powodu, by tego nie robić. Aby zmodyfikować metodę z listingu 18.6 tak, by zwracała zadanie, trzeba wprowadzić tylko jedną zmianę. Wystarczy zmienić typ wartości wynikowej z `void` na `Task`, jak pokazuje listing 18.8 — reszta kodu może pozostać bez zmian.

Listing 18.8. Zwracanie zadania

```
private async Task FetchAndShowBody(string url)
... jak wcześniej
```

Kompilator automatycznie generuje kod wymagany do utworzenia obiektu `Task` i w zależności od tego, czy metoda zwróci wynik, czy zgłosi wyjątek, ustawia jego status na zakończony lub zakończony niepowodzeniem. Także zwracanie wyniku z zadania jest bardzo łatwe. Wystarczy użyć typu `Task<T>`, a w kodzie metody umieścić instrukcję `return`, jak gdyby zwracała ona wartość typu `T`. Przykład takiej metody został przedstawiony na listingu 18.9.

Listing 18.9. Zwracanie zadania Task<T>

```
public static async Task<string> GetServerHeader(string url)
{
    using (var w = new HttpClient())
    {
        var request = new HttpRequestMessage(HttpMethod.Head, url);
        HttpResponseMessage response =
            await w.SendAsync(request, HttpCompletionOption.ResponseHeadersRead);

        string result = null;
        IEnumerable<string> values;
```

```
            if (response.Headers.TryGetValues("Server", out values))
            {
                result = values.FirstOrDefault();
            }
            return result;
        }
    }
```

Powyższa metoda asynchronicznie pobiera nagłówki HTTP, tak samo jak przykład z listingu 18.1, jednak zamiast je wyświetlać, pobiera i zwraca wartość pierwszego nagłówka `Server:`. Jak widać, instrukcja `return` zwraca łańcuch znaków, choć zadeklarowanym typem wartości wynikowej metody jest `Task<string>`. Kompilator generuje kod, który kończy wykonywanie zadania i używa zwróconego łańcucha znaków jako wyniku. W przypadku użycia typu `Task` lub `Task<T>` wygenerowany kod zwraca zadanie bardzo podobne do tego, które można uzyskać, używając klasy `TaskCompletionSource<T>`, opisanej w rozdziale 17.

Choć słowo kluczowe `await` może operować na dowolnej metodzie asynchronicznej pasującej do określonego wzorca (opisanego w dalszej części rozdziału), to jednak C# nie zapewnia równie wielkiej elastyczności, jeśli chodzi o możliwości implementacji metod asynchronicznych. Jedynymi typami, jakie mogą zwracać metody z modyfikatorem `async` są: `Task`, `Task<T>` oraz `void`.

Jednak zwracanie zadań ma pewną wadę. Otóż kod wywołujący nie ma obowiązku robić czegokolwiek z tak zwróconym zadaniem, zatem nasza metoda może być równie łatwa w użyciu co metoda zwracająca typ `void`, a jednocześnie ma tę zaletę, że udostępnia zadanie, które kod wywołujący może wykorzystać. Chyba jedynym powodem zwracania typu `void` mogłoby być narzucenie przez kod zewnętrzny konieczności użycia metody o określonej sygnaturze. Na przykład większość procedur obsługi zdarzeń musi używać typu `void`. Jednak oprócz sytuacji, gdy jesteśmy do tego zmuszeni, stosowanie w metodach asynchronicznych typu `void` nie jest zalecane.

Stosowanie async w metodach zagnieżdżonych

W przykładach przedstawionych do tej pory używaliśmy słowa kluczowego `async` tylko w zwyczajnych metodach. Jednak można je także stosować w metodach zagnieżdżonych — zarówno metodach anonimowych, jak i w wyrażeniach lambda. Na przykład: jeśli piszemy program, który tworzy elementy interfejsu użytkownika programowo, wygodnym rozwiązaniem może być dołączanie procedur obsługi zdarzeń w formie wyrażeń lambda, możemy się przy tym zdecydować, by niektóre z nich zostały zaimplementowane jako asynchroniczne, jak pokazano na listingu 18.10.

Listing 18.10. Asynchroniczne wyrażenie lambda

```
okButton.Click += async (s, e) =>
{
    using (var w = new HttpClient())
    {
        infoTextBlock.Text = await w.GetStringAsync(uriTextBox.Text);
    }
};
```

Składnia asynchronicznej metody anonimowej jest bardzo podobna, jak widać w przykładzie przedstawionym na listingu 18.11.

Listing 18.11. Asynchroniczna metoda anonimowa

```
okButton.Click += async delegate (object s, RoutedEventArgs e)
{
    using (var w = new HttpClient())
    {
        infoTextBlock.Text = await w.GetStringAsync(uriTextBox.Text);
    }
};
```

Żeby wszystko było jasne — powyższy kod nie ma nic wspólnego z asynchronicznym wywoływaniem delegatów, czyli techniką, o której wspominałem w rozdziale 9., służącą do korzystania z puli wątków i popularną, zanim metody anonimowe i TPL stały się lepszą alternatywą. Asynchroniczne wywoływanie delegatów jest rozwiązaniem, na które może się zdecydować kod korzystający z delegatu — jednak w takim przypadku asynchroniczność nie jest ani cechą delegatu, ani metody, która go wywołuje. Jest to jedynie rozwiązanie zastosowane przez kod używający delegatu. Jednak zastosowanie modyfikatora `async` w metodzie anonimowej lub wyrażeniu lambda pozwala nam na korzystanie wewnątrz nich ze słowa kluczowego `await`, zmieniając w ten sposób kod metody generowany przez kompilator.

Wzorzec słowa kluczowego await

Większość metod asynchronicznych, których będziemy używać wraz ze słowem kluczowym `await`, zwraca jakieś zadania TPL. Niemniej jednak C# wcale tego nie wymaga. Kompilator pozwala na stosowanie ze słowem kluczowym `await` dowolnych obiektów, implementujących określony wzorzec. Choć klasy `Task` i `Task<T>`, obsługują ten wzorzec, to jednak sposób jego działania oznacza, że kompilator będzie używał zadań w nieco inny sposób, niż to robimy, korzystając z biblioteki TPL bezpośrednio — po części właśnie z tego powodu napisałem wcześniej, że kod wykorzystujący zadania i stanowiący odpowiednik kodu używającego słowa kluczowego `await` nie stanowi dokładnego odpowiednika kodu generowanego przez kompilator. W tym podrozdziale wyjaśnię, jak kompilator używa zadań oraz innych typów, które mogą być stosowane wraz ze słowem kluczowym `await`.

W dalszej części tego podrozdziału stworzymy własną implementację wzorca słowa kluczowego `await`, aby pokazać, czego oczekuje kompilator. (Tak się składa, że Visual Basic rozpoznaje i obsługuje dokładnie ten sam wzorzec). Listing 18.12 przedstawia metodę asynchroniczną o nazwie `UseCustomAsync`, która korzysta z naszej asynchronicznej implementacji. Metod ta zapisuje wynik wyrażenia `await` w zmiennej typu `string`, a zatem najwyraźniej oczekuje, że nasza asynchroniczna operacja zwróci łańcuch znaków. Wywołuje ona metodę `CustomAsync`, zwracającą tę implementację wzorca. Jak widać, nie jest to wcale `Task<string>`.

Listing 18.12. Wywoływanie niestandardowej implementacji typu współpracującego z await

```
static async Task UseCustomAsync()
{
    string result = await CustomAsync();
    Console.WriteLine(result);
}

public static MyAwaitableType CustomAsync()
{
    return new MyAwaitableType();
}
```

Kompilator oczekuje, że typ operandu słowa kluczowego `await` będzie udostępniał metodę o nazwie `GetAwaiter`. Może to być zwyczajna metoda składowa bądź metoda rozszerzenia. (A zatem definiując odpowiednią metodę rozszerzenia, można sprawić, że słowo kluczowe `await` będzie współpracowało z typem, który sam z siebie go nie obsługuje). Metoda ta musi zwracać obiekt lub wartość zapewniającą trzy możliwości.

Przede wszystkim musi udostępniać właściwość typu `bool` o nazwie `IsCompleted`, którą kod wygenerowany przez kompilator do obsługi słowa kluczowego `await` będzie sprawdzał w celu określenia, czy operacja już się zakończyła. W przypadku gdy operacja została już zakończona, przygotowywanie wywołania zwrotnego byłoby stratą czasu. A zatem kod obsługujący słowo kluczowe `await` nie będzie tworzyć niepotrzebnego delegatu, jeśli właściwość `IsCompleted` zwróci wartość `true`, a zamiast tego od razu wykona dalszą część metody.

Oprócz tego kompilator wymaga jakiegoś sposobu pobrania wyniku, kiedy operacja zostanie już zakończona. Dlatego też obiekt lub wartość zwracana przez metodę `GetAwaiter` musi udostępniać metodę `GetResult`. Typ wyniku zwracanego przez tę metodę definiuje typ wyniku operacji — a zatem będzie to typ całego wyrażenia `await`. W przykładzie z listingu 18.12 wynik wyrażenia `await` jest zapisywany w zmiennej typu `string`, a zatem wynik zwracany przez metodę `GetResult` obiektu zwróconego przez metodę `GetAwaiter` klasy `MyAwaitableType` musi być typu `string` (bądź jakiegoś innego typu, który niejawnie można skonwertować na `string`).

I w końcu ostatnią możliwością, której potrzebuje kompilator, jest dostarczenie metody zwrotnej. Jeśli właściwość `IsCompleted` zwróci wartość `false`, informując tym samym, że operacja jeszcze się nie zakończyła, to kod wygenerowany przez kompilator do obsługi słowa kluczowego `await` wygeneruje delegat, który wykona pozostałą część kodu metody. (Przypomina to nieco przekazywanie delegatu do metody `ContinueWith` zadania). Kompilator wymaga w tym celu nie metody, lecz całego interfejsu. Musimy zatem zaimplementować interfejs `INotifyCompletion`, lecz oprócz niego istnieje jeszcze jeden interfejs, `ICiriticalNotifyCompletion`, którego implementacja jest zalecana, o ile tylko jest to możliwe. Oba te interfejsy są podobne: każdy z nich definiuje jedną metodę (`OnCompleted` oraz `UnsafeOnCompleted`), która pobiera jeden delegat typu `Action`, który klasa implementująca interfejs musi wywołać w momencie zakończenia operacji. Oba te interfejsy oraz ich metody różnią się tym, że pierwszy z nich wymaga od klasy implementującej przekazania kontekstu wykonania do metody docelowej, natomiast w przypadku drugiego interfejsu nie jest to konieczne. Kompilator C# zawsze przekazuje kontekst wykonania za nas, a zatem jeśli metoda `UnsafeOnCompleted` będzie dostępna, to kompilator wywoła ją, by uniknąć dwukrotnego przekazywania kontekstu. (Jeśli kompilator wywoła metodę `OnCompleted`, to także obiekt zwrócony przez metodę `GetAwaiter` przekaże kontekst wykonania). Niemniej jednak skorzystanie z metody `UnsafeOnCompleted` może być niemożliwe ze względów bezpieczeństwa. Ponieważ metoda ta nie przekazuje kontekstu wykonania, zatem kod niedysponujący pełnym zaufaniem nie może jej wywoływać, gdyż pozwalałoby to na ominięcie pewnych mechanizmów zabezpieczeń. Metoda `UnsafeOnCompleted` jest oznaczona atrybutem `SecurityCriticalAttribute`, co oznacza, że może ją wywoływać tylko kod dysponujący pełnym zaufaniem. A zatem metoda `OnCompleted` jest potrzebna, by także kod, który nie dysponuje pełnym zaufaniem, mógł korzystać z obiektu zwracanego przez metodę `GetAwaiter`.

Listing 18.13 przedstawia minimalną, nadającą się do użycia implementację wzorca słowa kluczowego `await`. Przedstawiony kod jest jednak bardzo uproszczony, gdyż zawsze kończy się synchronicznie, a zatem jego metoda `OnCompleted` nic nie robi. W rzeczywistości, jeśli nasza przykładowa klasa zostanie użyta w taki sposób, w jaki wzorzec `await` ma być używany,

to jej metoda `OnCompleted` w ogóle nie zostanie wywołana — właśnie dlatego zgłasza wyjątek. Niemniej jednak choć przedstawiony przykład jest nierealistycznie prosty, to jednak całkiem dobrze pokazuje sposób działania słowa kluczowego `await`.

Listing 18.13. Wyjątkowo prosta implementacja wzorca słowa kluczowego await

```
public class MyAwaitableType
{
    public MinimalAwaiter GetAwaiter()
    {
        return new MinimalAwaiter();
    }

    public class MinimalAwaiter : INotifyCompletion
    {
        public bool IsCompleted { get { return true; } }

        public string GetResult()
        {
            return "Oto wynik!";
        }

        public void OnCompleted(Action continuation)
        {
            throw new NotImplementedException();
        }
    }
}
```

Po przedstawieniu tego kodu możemy już zobaczyć, jak działa przykład z listingu 18.12. Wywoła on metodę `GetAwaiter` instancji typu `MyAwaitableType` zwróconej przez metodę `CustomAsync`. Następnie sprawdzi wartość właściwości `IsCompleted` uzyskanego obiektu i jeśli okaże się, że ma ona wartość `true` (co też się stanie), to bezzwłocznie zostanie wykonana reszta metody. Kompilator nie wie o tym, że właściwość `IsCompleted` zawsze ma wartość `true`, dlatego też wygeneruje kod pozwalający na prawidłowe obsłużenie przypadku, gdyby właściwość ta przyjęła wartość `false`. Kod ten utworzy delegat, który kiedy zostanie wywołany, wykona pozostałą część metody, po czym przekaże ten delegat do metody `OnComplete`. (Nasz przykładowy kod nie implementuje metody `UnsafeOnCompleted`, zatem zostanie użyta metoda `OnCompleted`). Kod, który wykonuje te wszystkie operacje, został przedstawiony na listingu 18.14.

Listing 18.14. Bardzo ogólne przybliżenie działania słowa kluczowego await

```
static void ManualUseCustomAsync()
{
    var awaiter = CustomAsync().GetAwaiter();
    if (awaiter.IsCompleted)
    {
        TheRest(awaiter);
    }
    else
    {
        awaiter.OnCompleted(() => TheRest(awaiter));
    }
}

private static void TheRest(MyAwaitableType.MinimalAwaiter awaiter)
{
    string result = awaiter.GetResult();
    Console.WriteLine(result);
}
```

Metoda została podzielona na dwie części, gdyż kompilator unika tworzenia delegatu, jeśli właściwość `IsCompleted` przyjmie wartość `true`, a my chcemy, by nasz kod działał podobnie. Niemniej jednak nie jest to dokładnie to samo, co robi kompilator C# — potrafi on także uniknąć tworzenia dodatkowych metod dla poszczególnych instrukcji `await`, choć oznacza to, że generowany przez niego kod jest znacząco bardziej skomplikowany. W rzeczywistości w przypadku metod zawierających tylko jedno słowo kluczowe `await` generowany przez kompilator narzut jest znacząco większy do tego z listingu 18.14. Niemniej jednak wraz ze wzrostem liczby używanych wyrażeń `await` ta dodatkowa złożoność zaczyna się opłacać, gdyż kompilator nie musi dodawać kolejnych metod. Listing 18.15 przedstawia kod, który w nieco większym stopniu przypomina to, co faktycznie generuje kompilator.

Listing 18.15. Nieco lepsze przybliżenie sposobu działania słowa kluczowego await

```
private class ManualUseCustomAsyncState
{
    private int state;
    private MyAwaitableType.MinimalAwaiter awaiter;

    public void MoveNext()
    {
        if (state == 0)
        {
            awaiter = CustomAsync().GetAwaiter();
            if (!awaiter.IsCompleted)
            {
                state = 1;
                awaiter.OnCompleted(MoveNext);
                return;
            }
        }
        string result = awaiter.GetResult();
        Console.WriteLine(result);
    }
}

static void ManualUseCustomAsync()
{
    var s = new ManualUseCustomAsyncState();
    s.MoveNext();
}
```

Powyższy kod i tak jest prostszy do tego, który kompilator generuje w rzeczywistości, jednak pokazuje ogólną strategię działania: kompilator generuje zagnieżdżony typ działający jako maszyna stanów. Definiuje on pole (`state`) przechowujące informacje o tym, do którego miejsca dotarła realizacja metody, oraz pola reprezentujące zmienne lokalne metody. (W powyższym przykładzie jest to jedynie zmienna `awaiter`). Kiedy operacja asynchroniczna nie zostaje zablokowana (czyli gdy właściwość `IsCompleted` natychmiast zwróci wartość `true`), od razu można wykonać następną część kodu. Jednak w przypadku, gdy wykonanie operacji wymaga nieco czasu, aktualizowana jest wartość zmiennej `state` w celu zapamiętania aktualnego stanu, po czym wywoływana jest metoda `OnCompleted` obiektu zwróconego przez metodę `GetAwaiter`. Należy zwrócić uwagę, że metodą, którą chcemy wywołać po zakończeniu operacji, jest aktualnie wykonywana metoda, czyli `MoveNext`. Takie rozwiązanie jest stosowane zawsze, niezależnie od ilości zastosowanych słów kluczowych `await` — każde wywołanie zwrotne wykonywane po zakończeniu operacji asynchronicznej powoduje wywołanie tej samej metody — po prostu klasa pamięta, dokąd dotarła realizacja metody, i wznawia jej działanie od tego miejsca.

Nie pokażę tu faktycznego kodu generowanego przez kompilator. Jest on skrajnie nieczytelny, gdyż zawiera mnóstwo całkowicie *niewymawialnych* identyfikatorów. (Zapewne pamiętasz z rozdziału 3., że kiedy kompilator C# musi wygenerować identyfikatory, które nie mogą kolidować z naszym kodem ani być w nim widoczne, to tworzy nazwy, które są prawidłowe, lecz które język C# uznaje za niedozwolone; to właśnie takie nazwy są określane jako *niewymawialne*, ang. *unspeakable*). Co więcej, kod generowany przez kompilator używa różnych klas pomocniczych należących do przestrzeni nazw `System.Runtime.CompilerServices`, które są przeznaczone wyłącznie do użycia w metodach asynchronicznych i służą do zarządzania takimi aspektami ich działania jak określanie, które interfejsy są dostępne w danym obiekcie, oraz obsługa przekazywania kontekstu wykonania. Co więcej, jeśli metoda zwraca zadanie, to używane są dodatkowe klasy pomocnicze, które to zadanie tworzą i aktualizują. Niemniej jednak jeśli chodzi o możliwość zrozumienia natury związku pomiędzy typem współpracującym ze słowem kluczowym `await` oraz kodem generowanym przez kompilator w celu obsługi tego słowa kluczowego, to kod z listingu 18.15 jest stosunkowo dobrym przybliżeniem.

Obsługa błędów

Słowo kluczowe `await` obsługuje wyjątki tak, jak byśmy sobie tego życzyli: jeśli wykonanie operacji asynchronicznej zakończy się niepowodzeniem, to wyjątek zostanie zgłoszony przez wyrażenie `await` realizujące tę operację. Ogólna zasada, zgodnie z którą kod asynchroniczny może mieć taką samą strukturę co zwyczajny kod synchroniczny, obowiązuje także w obliczu zgłaszanych wyjątków, kompilator robi wszystko co niezbędne, by zapewnić taką możliwość.

Listing 18.16 przedstawia dwie asynchroniczne operacje, z których jedna jest wykonywana wewnątrz pętli. Przypomina to nieco przykład z listingu 18.6. Poniższy przykład wykonuje jednak nieco inne operacje na pobieranych danych, jednak co jest najważniejsze — zwraca zadanie. Dzięki temu istnieje miejsce, do którego mogą trafić informacje o błędzie, w przypadku gdyby wykonanie operacji się nie udało.

Listing 18.16. Kilka potencjalnych źródeł niepowodzenia

```
private static async Task<string> FindLongestLineAsync(string url)
{
    using (var w = new HttpClient())
    {
        Stream body = await w.GetStreamAsync(url);
        using (var bodyTextReader = new StreamReader(body))
        {
            string longestLine = string.Empty;
            while (!bodyTextReader.EndOfStream)
            {
                string line = await bodyTextReader.ReadLineAsync();
                if (longestLine.Length > line.Length)
                {
                    longestLine = line;
                }
            }
            return longestLine;
        }
    }
}
```

Obsługa wyjątków jest potencjalnie sporym wyzwaniem dla operacji asynchronicznych, gdyż w momencie wystąpienia problemu metoda, która zapoczątkowała wykonywanie operacji, zazwyczaj będzie już zakończona. Przedstawiona w powyższym przykładzie metoda `FindLongestLineAsync` zazwyczaj kończy działanie w momencie wykonania pierwszego wyrażenia `await`. (Może się zdarzyć, że będzie inaczej — jeśli analizowany zasób będzie dostępny w lokalnej pamięci podręcznej HTTP, to operacja asynchroniczna może się natychmiast zakończyć sukcesem. Jednak zazwyczaj jej wykonanie zajmie nieco czasu, a to oznacza, że metoda zostanie zakończona). Załóżmy, że wykonanie operacji zakończy się pomyślnie i zacznie być wykonywana dalsza część metody, jednak gdzieś wewnątrz pętli pobierającej zawartość odpowiedzi zostanie przerwane połączenie sieciowe. W efekcie jedna z operacji rozpoczętych przez wywołanie metody `ReadLineAsync` zakończy się niepowodzeniem.

Wyjątek dla tej operacji zostanie zgłoszony przez wyrażenie `await`. Wewnątrz tej metody nie ma żadnego kodu obsługującego wyjątki, co zatem powinno stać się potem? Zazwyczaj oczekiwalibyśmy, że wyjątek zacznie być przekazywany w górę stosu wywołań, powstaje jednak pytanie, co znajduje się na stosie powyżej tej metody? Niemal na pewno nie będzie to ten sam kod, który ją wywołał — pamiętamy zapewne, że metoda zazwyczaj jest kończona w chwili, gdy dotrze do pierwszego wyrażenia `await`, a zatem na tym etapie nasz kod zazwyczaj będzie działał jako efekt wywołania zwrotnego wykonanego przez obiekt zwrócony przez metodę `GetAwaiter` na potrzeby zadania zwróconego przez metodę `ReadLineAsync`. Istnieje pewne prawdopodobieństwo, że nasz kod będzie jeszcze wykonywany w tym samym wątku z puli, a kod znajdujący się na stosie bezpośrednio nad nim będzie elementem obiektu zwróconego przez metodę `GetAwaiter`. Jednak ten kod nie będzie wiedział, co należy zrobić z wyjątkiem.

Jednak wyjątek nie jest przekazywany w górę stosu. Kiedy wyjątek zgłoszony w metodzie asynchronicznej zwracającej zadanie nie zostanie obsłużony, jest on przechwytywany przez kod wygenerowany przez kompilator, który następnie zmienia stan zwracanego zadania, informując, że jego wykonanie zakończyło się niepowodzeniem. Jeśli kod wywołujący metodę `FindLongestLineAsync` korzysta bezpośrednio z możliwości TPL, to będzie on w stanie wykryć fakt zgłoszenie wyjątku, sprawdzając stan zadania, oraz pobrać obiekt wyjątku, korzystając z właściwości `Exception` zadania. Ewentualnie można także wywołać metodę `Wait` lub odczytać wartość właściwości `Result` zadania, a każda z tych operacji spowoduje zgłoszenie wyjątku `AggregatedException` zawierającego obiekt oryginalnego wyjątku. Jeśli jednak kod wywołujący metodę `FindLongesLineAsync` użyje zwróconego obiektu zadania wraz ze słowem kluczowym `await`, to wyjątek zostanie ponownie zgłoszony w tym wyrażeniu. Z punktu widzenia kodu wywołującego wygląda to tak, jak gdyby wyjątek został zgłoszony w normalny sposób, co też pokazuje przykład z listingu 18.17.

Listing 18.17. Obsługa wyjątków zgłaszanych w wyrażeniu await

```
try
{
    string longest = await FindLongestLineAsync("http://192.168.22.1/");
    Console.WriteLine("Najdłuższy wiersz: " + longest);
}
catch (HttpRequestException x)
{
    Console.WriteLine("Błąd podczas pobierania strony: " + x.Message);
}
```

Powyższy kod jest niemal zwodniczo prosty. Pamiętajmy, że kompilator przeprowadza bardzo głęboką restrukturyzację naszego kodu wokół każdego wystąpienia słowa kluczowego `await` oraz że wykonanie kodu, który z pozoru może się wydawać jedną metodą, w rzeczywistości może wymagać wykonania kilku wywołań. Dlatego też zachowanie semantyki nawet tak prostego kodu obsługi błędów (lub podobnych konstrukcji, takich jak instrukcja `using`) jak ten z powyższego przykładu nie jest zadaniem trywialnym. Jeśli kiedykolwiek próbowałeś napisać analogiczny kod obsługi błędów w operacjach asynchronicznych bez korzystania z pomocy kompilatora, to zapewne docenisz, jak bardzo pomaga w tym C#.

Słowo kluczowe `await` pobiera oryginalny wyjątek z obiektu `AggregatedException` i ponownie go zgłasza. To dzięki temu metody asynchroniczne mogą obsługiwać błędy w taki sam sposób jak zwyczajny kod synchroniczny.

Weryfikacja poprawności argumentów

Sposób, w jaki C# automatycznie zgłasza błędy za pośrednictwem obiektu zadania zwracanego przez naszą asynchroniczną metodę (i to bez względu na ilość używanych wywołań zwrotnych), ma jedną wadę. Powoduje on bowiem, że kod taki jak ten z listingu 18.18 nie robi tego, na czym mogłoby nam zależeć.

Listing 18.18. W jaki sposób nie należy sprawdzać poprawności argumentów

```
public async Task<string> FindLongestLineAsync(string url)
{
    if (url == null)
    {
        throw new ArgumentNullException("url");
    }
    ...
```

Wewnątrz metody asynchronicznej kompilator traktuje wszystkie wyjątki w taki sam sposób: żaden z nich nie może zostać przekazany w górę stosu, jakby się to stało w normalnej metodzie, i każdy zostanie zasygnalizowany poprzez odpowiednią zmianę stanu zwracanego zadania — zostanie ono oznaczone jako zakończone niepowodzeniem. Dotyczy to nawet tych wyjątków, które są zgłaszane przed wykonaniem słowa kluczowego `await`. W naszym przykładzie weryfikacja argumentów jest wykonywana, zanim metoda wykona jakiekolwiek inne operacje, zatem w tym przypadku nasz kod będzie wykonywany w tym samym wątku, w którym metoda została wywołana. Można by pomyśleć, że wyjątek zgłoszony w tym miejscu kodu zostanie przekazany bezpośrednio do kodu wywołującego. W rzeczywistości jednak zauważy on jedynie zwyczajne zakończenie metody zwracającej obiekt zadania, przy czym stan tego zadania będzie informował, że zakończyło się ono niepowodzeniem.

Jeśli metoda wywołująca od razu wykona zwrócone zadanie, używając do tego celu słowa kluczowego `await`, to nie będzie to miało większego znaczenia — wyjątek i tak zostanie przez nią zauważony. Jednak może się zdarzyć, że kod nie będzie chciał od razy wykonać zadania, a w takim przypadku wyjątek nie zostanie zauważony tak szybko. Ogólnie przyjęta konwencja związana ze stosowaniem i obsługą prostych wyjątków związanych z weryfikacją argumentów zaleca, by w przypadkach, gdy nie ma wątpliwości, że problem został spowodowany kodem przez kod wywołujący, wyjątek należy zgłosić natychmiast. A zatem w powyższym przykładzie naprawdę powinniśmy wymyślić coś innego.

Jeśli nie ma możliwości sprawdzenia poprawności argumentów bez wykonywania jakichś długotrwałych operacji, to chcąc napisać naprawdę asynchroniczną metodę, nie będziemy w stanie zastosować się do powyższej konwencji. W takim przypadku trzeba będzie podjąć decyzję, czy wolimy, by metoda została zablokowana do momentu, gdy będzie w stanie sprawdzić poprawność argumentów, czy też by wyjątki dotyczące argumentów metody były zgłaszane za pośrednictwem zadania, a nie bezpośrednio.

Standardowym rozwiązaniem jest napisanie normalnej metody, która sprawdzi poprawność argumentów przed wywołaniem metody asynchronicznej, wykonującej zamierzone operacje. (Okazuje się, że w podobny sposób należałoby postępować w przypadku przeprowadzania natychmiastowej wersyfikacji argumentów iteratora. Iteratory zostały opisane w rozdziale 5.). Listing 18.19 przedstawia właśnie taką metodę publiczną oraz początek samej metody asynchronicznej.

Listing 18.19. Weryfikacja argumentów metody asynchronicznej

```
public Task<string> FindLongestLineAsync(string url)
{
    if (url == null)
    {
        throw new ArgumentNullException("url");
    }
    return FindLongestLineCore(url);
}

private async Task<string> FindLongestLineCore(string url)
{
    ...
```

Ponieważ metoda publiczna nie została oznaczona jako asynchroniczna (nie dodano do niej modyfikatora `async`), zatem wszelkie zgłaszane przez nią wyjątki będą przekazywane bezpośrednio do kodu wywołującego. Natomiast wszelkie problemy, które nastąpią po rozpoczęciu prywatnej metody asynchronicznej, będą zgłaszane za pośrednictwem obiektu zadania.

Wyjątki pojedyncze oraz grupy wyjątków

Z rozdziału 17. można się było dowiedzieć, że TPL definiuje model pozwalający na raportowanie wielu błędów — właściwość `Exception` zadania zwraca obiekt `AggregatedException`. Nawet jeśli pojawi się tylko jeden problem, to i tak informacje o nim trzeba będzie pobierać z obiektu `AggregateException`. Niemniej jednak w razie stosowania słowa kluczowego `await` to pobranie wyjątku jest wykonywane automatycznie za nas — jak mieliśmy się okazję przekonać w przykładzie z listingu 18.17, pobiera on pierwszy wyjątek zapisany w tablicy `InnerExcecptions`, a następnie ponownie go zgłasza.

Takie rozwiązanie jest wygodne, jeśli w operacji może wystąpić tylko jeden problem — nie musimy bowiem pisać żadnego dodatkowego kodu, który by obsługiwał ten grupowy wyjątek i pobierał jego zawartość. (Jeśli korzystamy z zadania zwróconego przez metodę asynchroniczną, to nigdy nie będzie ono zawierać więcej niż jednego wyjątku). Jednak takie rozwiązanie przysparza problemów, kiedy posługujemy się złożonymi zadaniami, w których jednocześnie może się pojawić kilka wyjątków. Na przykład do metody `Task.WhenAll` przekazywana jest kolekcja zadań, a metoda ta zwraca jedno zadanie, które zostanie zakończone

wyłącznie w przypadku, gdy wszystkie zadania podrzędne zostaną prawidłowo zakończone. Jeśli któreś z nich zakończą się niepowodzeniem, to uzyskamy obiekt `AggregateException` zawierający wiele błędów. W razie użycia słowa kluczowego `await` do obsługi takiego zadania zgłosi ono wyłącznie pierwszy z wyjątków.

Standardowe mechanizmy TPL — metoda `Wait` oraz właściwość `Result` — udostępniają pełen zbiór błędów, jednak blokują wykonywanie wątku, jeśli zadanie jeszcze nie zostało zakończone. A co moglibyśmy zrobić, gdybyśmy chcieli skorzystać z wydajnego, asynchronicznego działania słowa kluczowego `await`, które wykonuje coś w wątkach, wyłącznie jeśli znajdzie się dla nich coś do zrobienia, a jednocześnie chcielibyśmy zauważać wszystkie wyjątki? Jedno z potencjalnych rozwiązań zostało przedstawione na listingu 18.20.

Listing 18.20. Zastosowanie słowa kluczowego await i metody Wait

```
static async Task CatchAll(Task[] ts)
{
    try
    {
        var t = Task.WhenAll(ts);
        await t.ContinueWith(
                    x => {},
                    TaskContinuationOptions.ExecuteSynchronously);
        t.Wait();
    }
    catch (AggregateException all)
    {
        Console.WriteLine(all);
    }
}
```

Powyższa metoda używa słowa kluczowego `await`, by skorzystać z wydajności asynchronicznych metod C#, jednak zamiast stosować je wraz z samym zadaniem złożonym, używa go wraz z zadaniem, do którego została dodana kontynuacja. Kontynuacja może zostać pomyślnie zakończona, jeśli zostanie zakończona poprzedzająca ją operacja, niezależnie od tego, czy zakończy się ona pomyślnie, czy też nie. Zastosowana kontynuacja jest pusta, więc wewnątrz niej nie mogą wystąpić żadne problemy, a to oznacza, że w tym miejscu nie zostaną zgłoszone żadne wyjątki. Natomiast jeśli wykonanie którejkolwiek z operacji zakończyło się niepowodzeniem, to wywołanie metody `Wait` spowoduje zgłoszenie wyjątku `AggregatedException` — dzięki temu blok `catch` będzie w stanie zauważyć wszystkie wyjątki. Co więcej, ponieważ metoda `Wait` jest wykonywana dopiero po zakończeniu realizacji wyrażenia `await`, zatem wiemy, że zadanie zostało zakończone, a zatem wywołanie to nie spowoduje zablokowania metody.

Jedną z wad takiego rozwiązania jest to, że tworzy ono dodatkowe zadanie tylko po to, byśmy mogli zaczekać bez narażania się na napotkanie wyjątku. W powyższym przykładzie kontynuacja została skonfigurowana w taki sposób, że jest wykonywana synchronicznie, dzięki czemu unikamy realizacji drugiego fragmentu kodu przy użyciu puli wątków; niemniej jednak i tak stanowi to marnowanie zasobów. Nieco bardziej zagmatwane, lecz jednocześnie bardziej wydajne rozwiązanie mogłoby polegać na użyciu słowa kluczowego `await` w standardowy sposób i napisaniu kodu obsługi wyjątków w taki sposób, by sprawdzał on, czy nie zostało zgłoszonych więcej wyjątków. Takie rozwiązanie przedstawia listing 18.21.

Listing 18.21. Poszukiwanie dodatkowych wyjątków

```
static async Task CatchAll(Task[] ts)
{
    Task t = null;
    try
    {
        t = Task.WhenAll(ts);
        await t;
    }
    catch (Exception first)
    {
        Console.WriteLine(first);
        if (t != null && t.Exception.InnerExceptions.Count > 1)
        {
            Console.WriteLine("Znaleziono więcej wyjątków:");
            Console.WriteLine(t.Exception);
        }
    }
}
```

To rozwiązanie pozwala uniknąć tworzenia drugiego zadania, jednak jego wadą jest to, że wygląda nieco dziwnie.

Operacje równoległe i nieobsłużone wyjątki

Najprostszym sposobem używania słowa kluczowego `await` jest wykonywanie kolejnych operacji jedna po drugiej dokładnie w taki sam sposób, jaki robimy to w kodzie synchronicznym. Choć wydaje się, że działanie całkowicie sekwencyjne nie pozwala wykorzystywać całego potencjału kodu asynchronicznego, to jednak zapewnia możliwość znacznie wydajniejszego wykorzystania dostępnych wątków niż użycie analogicznego kodu synchronicznego, a dodatkowo w aplikacjach klienckich doskonale współpracuje z kodem obsługi interfejsu użytkownika. Jednak można pójść jeszcze dalej.

Istnieje możliwość jednoczesnego uruchomienia kilku różnych operacji. Można wywołać metodę asynchroniczną, a następnie zamiast od razu skorzystać ze słowa kluczowego `await`, można zapisać wynik w zmiennej i w podobny sposób uruchomić drugą operację asynchroniczną, po czym zaczekać na zakończenie obu. Choć takie rozwiązanie jest możliwe do wykonania, to jednak kryje ono w sobie pewną pułapkę na nieostrożnych programistów; przedstawia ją przykład z listingu 18.22.

Listing 18.22. W jaki sposób nie należy wykonywać wielu współbieżnych operacji

```
static async Task GetSeveral()
{
    using (var w = new HttpClient())
    {
        w.MaxResponseContentBufferSize = 2000000;

        Task<string> g1 = w.GetStringAsync("http://helion.pl/");
        Task<string> g2 =
            w.GetStringAsync("http://helion.pl/kategorie/programowanie/c-sharp");

        // BŁĄD!

        Console.WriteLine((await g1).Length);
        Console.WriteLine((await g2).Length);
    }
}
```

Powyższa metoda pobiera równocześnie zawartość dwóch stron WWW. Po uruchomieniu obu operacji metoda używa słowa kluczowego `await`, by pobrać ich wyniki i wyświetlić długości zwróconych łańcuchów znaków. Jeśli operacje zakończą się pomyślnie, to powyższy kod zadziała prawidłowo, jednak nie zapewnia on prawidłowej obsługi błędów. Jeśli wykonanie pierwszej operacji zakończy się niepowodzeniem, to powyższy kod nigdy nie wykona drugiego wyrażenia `await`. To oznacza, że jeśli także druga operacja zakończy się niepowodzeniem, to nie będzie kodu, który mógłby sprawdzić zgłoszone wyjątki. W końcu TPL wykryje, że wyjątki nie zostały zauważone, co spowoduje zgłoszenie wyjątku `UnobservedTaskException`, a on najprawdopodobniej doprowadzi do awarii programu. (Zagadnienia związane z obsługą niezaobserwowanych wyjątków zostały opisane w rozdziale 17.). Problem polega na tym, że takie sytuacje zdarzają się bardzo rzadko — konieczne jest bowiem, by obie operacje zakończyły się niepowodzeniem i to w bardzo krótkim odstępie czasu — a zatem bardzo łatwo będzie je przegapić podczas testowania aplikacji.

Takich problemów można uniknąć dzięki uważnej obsłudze błędów — na przykład można przechwytywać wszystkie wyjątki zgłaszane przez pierwsze wyrażenie `await` przed wykonaniem drugiego z nich. Ewentualnie można także skorzystać z metody `Task.WhenAll`, by poczekać na wyniki obu operacji wykonywanych w formie jednego zadania — w takim przypadku, gdyby nie udało się wykonać którejkolwiek z operacji, uzyskalibyśmy zadanie zakończone niepowodzeniem, z informacjami o błędach zapisanymi w obiekcie `AggregatedException`, dzięki czemu moglibyśmy sprawdzić wszystkie zgłoszone wyjątki. Oczywiście jak mogliśmy się przekonać, obsługa wielu błędów w przypadku korzystania ze słowa kluczowego `await` może być dosyć kłopotliwa. Jeśli jednak chcemy uruchamiać wiele asynchronicznych operacji i pozwolić, by wszystkie były wykonywane jednocześnie, to kod niezbędny do koordynacji uzyskiwanych wyników będzie bardziej złożony niż w przypadku wykonywania tych samych operacji sekwencyjnie. Niemniej jednak słowa kluczowe `await` oraz `async` i tak znacznie ułatwiają nam życie.

Podsumowanie

Operacje asynchroniczne nie blokują wątku, w którym zostały rozpoczęte; dzięki temu są bardziej wydajne od zwyczajnych metod synchronicznych, co ma szczególnie duże znaczenie na bardzo obciążonych komputerach. Ta cecha sprawia również, że z powodzeniem można z nich korzystać w aplikacjach klienckich, gdyż pozwalają na wykonywanie długotrwałych operacji bez obniżania szybkości reakcji interfejsu aplikacji na działania użytkownika. Jednak wadą operacji asynchronicznych zawsze była ich wysoka złożoność, dotyczy to w szczególności obsługi błędów w przypadku stosowania wielu powiązanych ze sobą operacji. W języku C# 5.0 wprowadzono słowo kluczowe `await`, które pozwala na pisanie kodu asynchronicznego w sposób bardzo zbliżony do zwyczajnego kodu synchronicznego. Sprawy się nieco komplikują, jeśli chcemy, by jedna metoda zarządzała kilkoma operacjami wykonywanymi równolegle, jednak nawet jeśli napiszemy tę metodę w taki sposób, że poszczególne operacje asynchroniczne będą wykonywane w ściśle określonej kolejności, to i tak uzyskamy korzyści. W przypadku aplikacji serwerowej tą korzyścią będzie znacznie bardziej wydajne wykorzystanie wątków, dzięki czemu taka aplikacja będzie w stanie obsłużyć większą liczbę jednocześnie działających użytkowników, gdyż każda z operacji będzie zużywać mniej zasobów. Natomiast w przypadku aplikacji klienckich tą korzyścią będzie działający sprawniej interfejs użytkownika.

Metody korzystające ze słowa kluczowego `await` muszą być oznaczone przy użyciu modyfikatora `async` i powinny zwracać wynik typu `Task` lub `Task<T>`. (C# pozwala także na zwracanie wyniku typu `void`, jednak zazwyczaj jest on stosowany wyłącznie w ostateczności, gdy nie ma innego wyboru). Kompilator zadba o to, by zadanie zostało zakończone pomyślnie, jeśli nasza metoda zostanie prawidłowo wykonana, oraz by zakończyło się niepowodzeniem, jeśli w trakcie wykonywania metody pojawią się jakiekolwiek problemy. Ponieważ słowo kluczowe `await` może operować na każdym obiekcie `Task` lub `Task<T>`, zatem ułatwia ono rozdzielenie logiki asynchronicznej na wiele metod, gdyż metoda nadrzędna może używać go do wykonywania metod podrzędnych. Zazwyczaj faktyczne operacje są wykonywane przez jakieś metody wykorzystujące zadania, jednak nie jest to regułą, gdyż słowo kluczowe `await` wymaga jedynie użycia określonego wzorca — można w nim podać dowolne wyrażenie pozwalające na wywołanie metody `GetAwaiter` w celu uzyskania obiektu odpowiedniego typu.

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

T

U

V

W

X

Z

Ź

Ż

Programowanie

O'REILLY®

C# 5.0. Programowanie

Tworzenie aplikacji Windows 8, internetowych oraz biurowych w .NET 4.5 Framework

W dzisiejszych czasach szczególną popularnością cieszą się języki programowania pozwalające na pisanie kodu łatwego do przenoszenia między platformami. Nikt nie ma czasu na pisanie kilku wersji jednej aplikacji. C# to uniwersalny język, w którym bez trudu dokonasz tego dzieła. Dzięki swojej elastyczności, wydajności oraz mocnemu wsparciu społeczności zdobył uznanie programistów. Taki wybór to strzał w dziesiątkę!

Ten rewelacyjny podręcznik jest Twoim kluczem do poznania wszystkich niuansów języka C# 5.0. Kolejne wydanie zostało zaktualizowane o wszystkie nowości w C#. Znajdziesz tu kompletny opis języka i platformy .NET. W trakcie lektury oprócz standardowych zagadnień będziesz mógł sprawdzić, jak tworzyć aplikacje dla systemu Windows 8 i interfejsu Metro. Ponadto błyskawicznie opanujesz detale związane z programowaniem obiektowym, dynamicznym i statycznym określaniem typów oraz językiem XAML. Książka ta jest uznanym kompendium wiedzy na temat języka C#. Musisz ją mieć!

Dzięki tej książce:

- przygotujesz interfejs użytkownika zgodny z duchem Windows 8
- wykorzystasz wielowątkowość w platformie .NET
- poznasz podstawy programowania obiektowego
- przekonasz się, jak LINQ może ułatwić Ci życie
- opanujesz język C#

Wykorzystaj potencjał języka C#!

Ian Griffiths jest autorem kursu WPF oraz instruktorem w firmie Pluralsight specjalizującej się w prowadzeniu kursów Microsoft .NET. Pracuje także jako niezależny konsultant. Jest współautorem książek *Windows Forms in a Nutshell*, *Mastering Visual Studio .NET* oraz *Programming WPF*, wydanych przez wydawnictwo O'Reilly.

ISBN 978-83-246-6984-4

Cena 129,00 zł

9 788324 669844