Kontekst synchronizacji

Wątek w platformie .NET może posiadać kontekst synchronizacji reprezentowany przez instancję klasy `SynchronizationContext` lub jej klasy potomnej. Jeżeli wątek posiada taki kontekst (co można sprawdzić odczytując statyczną własność `SynchronizationContext.Current`), możliwe jest przekazanie referencji do niego do innego wątku. Wówczas za pomocą metod tego obiektu `Send` lub `Post` możliwe jest uruchomienie wskazanej w ich argumencie metody lub wyrażenia Lambda w wątku, z którym kontekst synchronizacji jest związany. Brzmi to może dość zawile, ale w istocie nie jest bardziej skomplikowane niż użycie metod `Control.InvokeRequired` i `Control.Invoke`. Pokazuje to przykład widoczny na listingu 5.A. Jest to kod z pliku *Form1.cs* aplikacji Windows Forms, w której na formie umieszczony jest pasek postępu (`progressBar1`) i przycisk (`button1`). W aplikacji tego typu kontekst synchronizacji jest automatycznie tworzony dla wątków odpowiedzialnych za obsługę okien (wątków interfejsu). Jest to obiekt typu `WindowsFormsSynchronizationContext`, zdefiniowany w przestrzeni `System.Windows.Forms`, ale potomny względem klasy `SynchronizationContext` z przestrzeni nazw `System.Threading`.

Listing 5.A. Użycie kontekstu synchronizacji w aplikacji Windows Forms

```
using System;
...

using System.Threading;

namespace KontekstSynchronizacji_WinForm
{
    public partial class Form1 : Form
    {
        static int min = 0;
        static int max = 100;
        static int opoznienie = 100;

        public Form1()
        {
            InitializeComponent();
            resetujInterfejs(null, null);
        }

        private void resetujInterfejs(object sender, EventArgs e)
        {
            progressBar1.Minimum = min;
            progressBar1.Maximum = max;
            progressBar1.Value = min;
        }

        private void ustawWartoscPaskaPostepu(object parametr)
        {
            int nowaWartosc = (int)parametr;
            progressBar1.Value = nowaWartosc;
        }
```

```
        private void kontrolaPaskaPostepu(object parametr)
        {
            SynchronizationContext kontekst = parametr as SynchronizationContext;
            for (int i = min; i <= max; i++)
            {
                kontekst.Send(ustawWartoscPaskaPostepu, i);
                Thread.Sleep(opoznienie);
            }
            kontekst.Send((object niewykorzystany)=>{ button3.Text = "Koniec"; }, null);
        }

        private void button1_Click(object sender, EventArgs e)
        {
            SynchronizationContext kontekst = SynchronizationContext.Current;
            Thread t = new Thread(kontrolaPaskaPostepu);
            t.Start(kontekst);
        }
    }
}
```

Interesująca nas część przebiegu aplikacji rozpoczyna się wraz z kliknięciem przycisku, które uruchamia metodę zdarzeniową `button1_Click`. Metoda ta wykonywana jest rzecz jasna w wątku obsługującym interfejs. W tej metodzie tworzony jest wątek, który ma wykonać metodę `kontrolaPaskaPostepu`. Do tej metody przesyłany jest kontekst synchronizacji wątku interfejsu, czyli instancja klasy `WindowsFormsSynchronizationContext` odczytana ze statycznej własności `SynchronizationContext.Current`. Działająca w dodatkowym wątku metoda `kontrolaPaskaPostepu` wykonuje pętlę `for` z indeksem przebiegającym wartości od 0 do 100. W kolejnych iteracjach pętli zmieniana jest własność `Value` paska postępu. Ponieważ pętla ta nie jest wykonywana w wątku, w którym powstała kontrolka paska postępu, nie mogą bezpośrednio odwoływać się jej własności (`progressBar1.Value = i;`). Groziłoby to pojawieniem się wyjątku `InvalidOperationException`. Tu z pomocą przychodzi kontekst synchronizacji, którego metodą `Send` uruchomiamy metodę `ustawWartoscPaskaPostepu`. Metoda ta wykonywana będzie w wątku, z którego ów kontekst pochodzi, a więc w wątku interfejsu. W tej ostatniej metodzie można zatem bez obaw odczytywać i modyfikować kontrolki umieszczone na formie. Definiowanie osobnej metody `ustawWartoscPaskaPostepu` nie jest wcale konieczne. Równie dobrze możliwe byłoby wykorzystanie wyrażenia Lambda, którego sygnatura zgodna jest z delegatem `SendOrPostCallback` a więc przyjmuje jeden argument typu `object` i nie zwraca wartości: `kontekst.Send((object nowaWartosc) => { progressBar1.Value = (int)nowaWartosc; }, i);`. W tym przykładzie zależało mi jednak na wyraźnym odseparowaniu tej części kodu, która choć wywoływana w dodatkowym wątku, wykonywana będzie w wątku wskazanym przez kontekst synchronizacji.

Dla porównania zestawmy powyższy przykład z listingiem 5.B, na którym widoczny jest analogiczny kod, ale korzystający z metod `Control.InvokeRequired` i `Control.Invoke`. Zwróćmy uwagę na podobieństwa obu podejść. W obu przypadkach wątek dodatkowy musi mieć referencję do obiektu związanego z wątkiem interfejsu. Jest to obiekt kontekstu synchronizacji w pierwszym przypadku, a referencja do dowolnej kontrolki w drugim. Teoretycznie rzecz biorąc mechanizm oparty na kontekście synchronizacji mógłby być użyty dla dowolnych dwóch wątków, z których żaden nie musi być wątkiem okna. Można w ten sposób na przykład rozdzielić dwa moduły aplikacji z dwoma niezależnymi wątkami, zachowując ich pełną niezależność względem siebie. W praktyce samodzielne utworzenie kontekstu synchronizacji nie jest jednak łatwe – zwykle wykorzystywane są tylko gotowe konteksty dostarczone wraz z bibliotekami kontrolek, a więc `WindowsFormsSynchronizationContext` w przypadku Windows Forms, `DispatcherSynchronizationContext` w przypadku WPF i `AspNetSynchronizationContext` w przypadku ASP.NET. Konteksty synchronizacji dostarczone wraz z platformą .NET pozwalają na wyraźne rozdzielenie warstwy interfejsu (widoku) ze zdefiniowanymi w niej synchronicznymi metodami modyfikującymi interfejs i warstwy logiki, która nie musi zawierać żadnych bezpośrednich odniesień do interfejsu i działających w nim kontrolek. Wystarczy jej wiedza o kontekście synchronizacji wątku interfejsu. Zaletą użycia kontekstu

synchronizacji jest prostota kodu, który z niego korzysta. Wystarczy porównać metody `ustawWartoscPaskaPostepu` z listingu 5.A i 5.B[1].

Listing 5.B. Przykład analogiczny do listingu 5.A, ale synchronizacja przeprowadzana za pomocą Control.Invoke

```
using System;
...

using System.Threading;

namespace Invoke_WinForm
{
    public partial class Form1 : Form
    {
        static int min = 0;
        static int max = 100;
        static int opoznienie = 100;



...

        public static void ustawWartoscPaskaPostepu(ProgressBar progressBar,
                                                    int nowaWartosc)
        {
            if (progressBar.InvokeRequired)
            {
                progressBar.Invoke(
                    new new Action<ProgressBar,int>(ustawWartoscPaskaPostepu),
                    new object[] { progressBar, nowaWartosc });
            }
            else
            {
                if(progressBar!=null) progressBar.Value = nowaWartosc;
            }
        }

        private void kontrolaPaskaPostepu(object parametr)
        {
            for (int i = min; i <= max; i++)
            {
                ustawWartoscPaskaPostepu_Invoke(progressBar1, i);
                Thread.Sleep(opoznienie);
            }
            button1.Invoke(new Action(() => { button1.Text = "Koniec"; }));
```

[1] Po więcej informacji warto zajrzeć na strony: *http://msdn.microsoft.com/en-us/magazine/gg598924.aspx*, *http://www.codeproject.com/Articles/31971/Understanding-SynchronizationContext-Part-I*.

```
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Thread t = new Thread(kontrolaPaskaPostepu_Invoke);
            t.Start(checkBox1.Checked);
        }
    }
}
```

Warto też zaznaczyć, że obok metod blokujących `SynchronizationContext.Send` i `Control.Invoke`, w obu mechanizmach mamy także do dyspozycji metody wykonywane asynchronicznie: `SynchronizationContext.Post` i `Control.BeginInvoke` (oraz `Control.EndInvoke`). One również uruchamiają zadania w wątku interfejsu, ale nie blokują wątku dodatkowego. W naszym przykładzie użycie metody `Post` nie ma jednak większego sensu. Spowodowałoby to szybkie przebiegnięcie pętli `for` (pomijając sztucznie wprowadzone do niej opóźnienie) i tym samym zakolejkowanie stu zadań do wykonania w wątku interfejsu, które byłyby wykonywane po kolei.

Metody `Post` i `Send` różnią się także sposobem obsługi błędów. W przypadku metody `Send` wyjątki zgłaszane w metodzie wykonywanej w wątku okna (w naszym przykładzie metoda `ustawWartoscPaskaPostepu`) są przekazywane do miejsca wywołania metody `Send`. Dzięki temu ma sens otoczenie wywołania tej metody konstrukcją `try..catch`. To bardzo wygodne rozwiązanie, które niestety nie zadziała w przypadku asynchronicznego wykonywania czynności zainicjowanych przez metodę `Post`. Wówczas wyjątki skazane są na brak obsłużenia i w efekcie najprawdopodobniej przerwą działanie wątku okna.

Komentarz [JM1]: A jak jest z obsługą błędów w Invoke i BeginInvoke?

Miłą zaletą mechanizmu synchronizacji wątku interfejsu opartego na kontekście synchronizacji jest to, że w zasadzie bez żadnych zmian można kod z listingu 5.A zastosować także w projektach WPF. Odtwórzmy w aplikacji WPF projekt aplikacji z paskiem postępu i przyciskiem. Przykładowy kod XAML widoczny jest na listingu 5.C.

Listing 5.C. Kod określający wygląd przykładowej aplikacji WPF

```
<Window x:Class="KontekstSynchronizacji_WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="117.335" Width="540.885">
    <Grid>
        <ProgressBar Name="progressBar1" HorizontalAlignment="Left" Height="29"
Margin="10,10,0,0" VerticalAlignment="Top" Width="498"/>
        <Button Name="button1" Content="Kontekst synchronizacji"
HorizontalAlignment="Left" Margin="10,45,0,0" VerticalAlignment="Top" Width="234"
RenderTransformOrigin="0.104,-0.768" Height="28" Click="Button_Click_1"/>
    </Grid>
</Window>
```

W metodzie zdarzeniowej przycisku `Button_Click_1` stwórzmy dodatkowy wątek, który będzie wykonywał metodę `kontrolaPaskaPostepu`. Wygląd tych dwóch metod nie różni się niczym zasadniczym od ich wersji z aplikacji Windows Forms poza faktem, że kontekst synchronizacji jest teraz instancją klasy `System.Windows.Threading.DispatcherSynchronizationContext`. Tak samo, jak w aplikacji Windows Forms wygląda również metoda `ustawWartoscPaskaPostepu` wywoływana w kontekście wątku interfejsu. Wszystkie trzy metody widoczne są na listingu 5.D.

Listing 5.D. Użycie kontekstu synchronizacji w aplikacji WPF

```
private void ustawWartoscPaskaPostepu(object parametr)
{
    int nowaWartosc = (int)parametr;
    progressBar1.Value = nowaWartosc;
}
```

```
private void kontrolaPaskaPostepu(object parametr)
{
    SynchronizationContext kontekst = parametr as SynchronizationContext;

    for (int i = min; i <= max; i++)
    {
        kontekst.Send(ustawWartoscPaskaPostepu, i);
        Thread.Sleep(opoznienie);
    }
    kontekst.Send((object niewykorzystany) => { button1.Content = "Koniec"; }, null);
}

private void Button_Click_1(object sender, RoutedEventArgs e)
{
    SynchronizationContext kontekst = SynchronizationContext.Current;
    Thread t = new Thread(kontrolaPaskaPostepu);
    t.Start(kontekst);
}
```

Nieco różnic jest w przypadku aplikacji dla Windows 8 z interfejsem Modern UI. Po pierwsze mamy tam do czynienia z platformą WinRT, a nie zwykłą platformą .NET. Platforma ta nie wspiera tworzenia wątków ani za pomocą klasy `Thread`, ani za pomocą puli wątków. Możemy jednak odtworzyć aplikację używając zadań (zob. rozdział 6.). Wówczas przekonamy się, że z wątkiem interfejsu związany jest kontekst synchronizacji zaimplementowany w klasie `WinRTSynchronizationContext` z przestrzeni nazw `System.Threading`. Sposób jego przekazania i użycia jest w zasadzie podobny, jak w powyższych przykładach z tą istotną różnicą, że nie ma wsparcia dla metody `Send`, a jedynie dla asynchronicznej metody `Post`.

Groźba zagłodzenia wątku interfejsu

W przypadku aplikacji z interfejsem graficznym, w których dodatkowe wątki modyfikują wygląd interfejsu, należy pamiętać o wszystkich niebezpieczeństwach, jakie czyhają na nas w przypadku aplikacji wielowątkowych. W szczególności bardzo łatwo o zakleszczenie, w efekcie którego może dojść do zagłodzenia wątku interfejsu. Aby to pokazać wystarczy na końcu metody zdarzeniowej uruchamianej po kliknięciu przycisku z listingów 5.A, 5.B bądź 5.D umieścić wywołanie metody `Join` na rzecz tworzonych w nich wątków (instrukcja `t.Join()`). Efektem będzie trwałe „zastygnięcie” aplikacji tuż po kliknięciu przycisku. Dlaczego? Użycie metody `Control.Invoke` lub metody `SynchronizeContext.Send`, a więc metod, które wstrzymują bieżący wątek do czasu, gdy zakończy się wykonywanie działań w wątku interfejsu, należy traktować jak punkt synchronizacji. Prześledźmy działanie obu wątków na przykładzie kodu z listingu 5.D. Po kliknięciu przycisku przez użytkownika, w wątku interfejsu uruchamiana jest metoda `Button_Click_1`. W niej tworzony jest dodatkowy wątek, w którym zaczyna działać pętla `for`. W jej pierwszej iteracji wywoływana jest metoda `kontekst.Send`, która stara się zakolejkować czynność do wykonania przez wątek interfejsu i czeka na jej zakończenie. Ale wątek interfejsu nadal wykonuje metodę `Button_Click_1` nie mogąc jej opuścić ze względu na polecenie `t.Join`, które z kolei czeka na zakończenie dodatkowego wątku. A skoro tak, nie może wykonać czynności zleconych w metodzie `Send`. W konsekwencji oba wątki „stoją” nawzajem na siebie czekając. Zatrzymanie wątku interfejsu powoduje, że interfejs aplikacji nie odpowiada na czynności użytkownika.

Zadanie

1. Przygiotuj „wrapper” do kontrolki `ProgressBar`, w którym dostęp do własności jest bezpieczny ze względu na wątki.