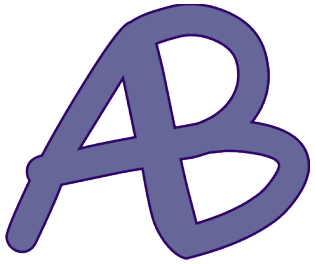


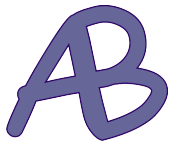
Wykład

Wątki – programowanie współbieżne

Wątki – programowanie współbieżne

2 cores

Wątki



Klasa BackgroundWorker umożliwia delegowanie pewnych operacji do wątków pracujących współbieżnie (równolegle) z główna aplikacją. Za jej pomocą pewne operacje mogą być wykonywane w tle.

Czasochłonne operacje mogą spowodować, że interfejs użytkownika zachowuje się tak, jakby przestał odpowiadać. Jeśli chcemy aby program nie był „zamrożony” do czasu zakończenia operacji możemy umieścić ją w obiekcie klasy BackgroundWorker.

Należy dodać:

```
using System.ComponentModel;
```

Wątki

Przygotowanie BackgroundWorker-a:

1. Utworzenie referencji do obiektu klasy BackgroundWorker. Tworzymy ją zwykle jako pole klasy reprezentującej aplikację (globalnie).

```
private BackgroundWorker worker;
```

„worker" to nazwa własna obiektu

2. Utworzenie instancji BackgroundWorker-a,

```
worker = new BackgroundWorker
{
   WorkerReportsProgress = true,
   WorkerSupportsCancellation = true
};
```

Opcjonalnie, możemy umożliwić wątkowi raportowanie postępów oraz umożliwić programowi głównemu przerwanie pracy wątku

Wątki

Klasa BackgroundWorker posiada metody:

- ✓ DoWork – w której umieszczamy operacje do wykonania w tle
- ✓ RunWorkerCompleted – która wywoływana jest po zakończeniu pracy wątku.
- ✓ ProgressChanged – którą wykorzystać można do raportowania postępów wątku do programu głównego (np. pasek postępu)

Uruchomienie wątku:

```
worker.RunWorkerAsync();
```

Wątki – metoda .DoWork

.DoWork – tu umieszczamy operacje do wykonania w tle.

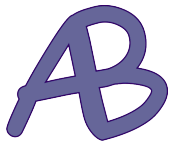
Krok 1: Dodanie metody .DoWork do obiektu klasy BackgroundWorkter

```
worker.DoWork += worker_DoWork;
```

Krok 2: zaimplementowanie metody

```
private void worker_DoWork(object? sender, DoWorkEventArgs e)
        {
              //kod wykonywany w tle
        }
```

Wątki – metoda .DoWork

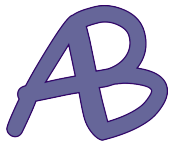


Metoda .DoWork zwraca wartość za pośrednictwem obiektu „e” (typu `DoWorkEventArgs`)

```
private void worker_DoWork(object? sender, DoWorkEventArgs e)
        {
            e.Result = 1000;
        }
```

Uwaga: z wnętrza metody DuWork nie ma możliwości sięgnięcia do kontrolek okna głównego programu.

Wątki – metoda .DoWork



Metoda DuWork może otrzymać parametry „zapakowane” w argument „e”

Uruchomienie wątku wraz z argumentem przekazanym w konstruktorze:

```
worker.RunWorkerAsync(10);
```

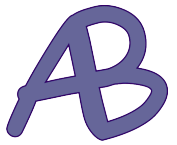
Odebranie argumentu w metodzie DoWork:

```
private void worker_DoWork(object? sender, DoWorkEventArgs e)
{
    int ile = (int)e.Argument;
}
```

Zabezpieczenie przed błędem w przypadku wywołania wątku bez argumentu:

```
int ile = 0;
if (e.Argument != null)
        ile= (int)e.Argument;
```

Wątki – metoda RunWorkerCompleted



Metoda RunWorkerCompleted wykonywana jest po zakończeniu pracy wątku. Jej zadaniem jest zwykle przekazanie wyniku pracy wątku do programu głównego. Ma ona bezpośredni dostęp do kontrolek programu głównego oraz jego zmiennych.

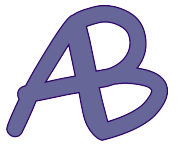
Krok 1: Dodanie metody .RunWorkerCompleted do obiektu klasy BackgroundWorkter:

```
worker.RunWorkerCompleted += worker_RunWorkerCompleted;
```

Krok 2: zaimplementowanie metody:

```
private void worker_RunWorkerCompleted(object? sender,
RunWorkerCompletedEventArgs e)
 {
        //kod wykonywany po zakończeniu pracy wątku
 }
```

Wątki – metoda RunWorkerCompleted

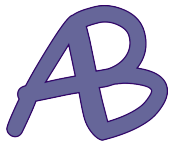


Metoda RunWorkerCompleted otrzymuje wynik pracy metody .DoWork za pośrednictwem parametru „e” (obiekt klasy `DoWorkEventArgs`) .

Jeżeli wątek zakończył pracę prawidłowo Wynik powinien znajdować się w e.Result

```
private void worker_RunWorkerCompleted(object? sender,
RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
    {
        // akcja, jeżeli pracę wątku przerwano
    }
    else if (e.Error != null)
    {
        // akcja, jeżeli praca wątku zakończyła się błędem
    }
    else
    {
        // akcja, jeżeli wątek zakończył się "normalnie"
        // wynik pracy wątku znajduje się w: e.Result
    }
}
```

Wątki – metoda ProgressChanged



ProgressChanged – metoda, którą wykorzystać można do raportowania postępów wątku do programu głównego (pasek postępu)

Metoda ProgressChanged musi być wywoływana cykliczne wewnątrz metody DoWork() z parametrem mówiącym o postępie wątku.

```
backgroundWorker1.ReportProgress (i);
```

Z wewnątrz metody ProgressChanged możemy sięgnąć do kontrolek procesu głównego.

Np.:

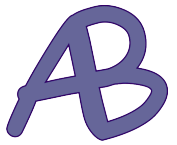
```
progressBar1.PerformStep();
```

lub

```
progressBar1.Value = e.ProgressPercentage;
```

Przerwanie pracy wątku

Wątki - Przerwanie pracy wątku



Przerywanie wątku

Do przerwania pracy wątku służy metoda CancelAsync()

Np.: `worker.CancelAsync();`

UWAGA: nie wymusza ona bezwarunkowego przerwania wątku. Stanowi tylko informację, że wątek powinien zakończyć pracę. Sposób jej zakończenia należy zdefiniować wewnątrz wątku.

Wątki

Oprogramowanie przerwania wątku:

W metodzie DoWork dodajemy reakcję na własność CancellationPending (*true* oznacza żądanie przerwania wątku)

```
if (worker.CancellationPending)
   {
        e.Cancel = true;
        return;
   }
```

W przypadku wykrycia żądania przerwania wątku:

1. ustawiamy pole Cancel zdarzenia „e” na true (potwierdzamy zamknięcie)
2. przerywany pracę metody DoWork (polecenie break)

Wątki

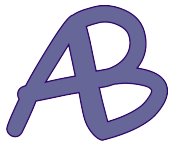
Wykrywanie sposobu w jaki wątek zakończył pracę (czy został przerwany):

W metodzie RunWorkerComleted, która kończy pracę wątku.

```
private void worker_RunWorkerCompleted(object? sender,
RunWorkerCompletedEventArgs e)
  {
            if (e.Cancelled)
            {
                // akcja, jeżeli pracę wątku przerwano
            }
            else if (e.Error != null)
            {
                // akcja, jeżeli praca wątku zakończyła się błędem
            }
            else
            {
                // akcja, jeżeli wątek zakończył się "normalnie"
                // wynik pracy wątku znajduje się w: e.Result
            }
        }
```

Jeżeli pole Canceled zdarzenia „e" jest równe true (wątek został zamknięty) reagujemy na ten fakt – np. wypisując komunikat o błędzie. W przeciwnym razie kończmy wątek normalnie

Wątki



Polecenie przydatne przy testowaniu działania programów oddelegowujących zadania do wątków.

Wstrzymanie pracy wątku:

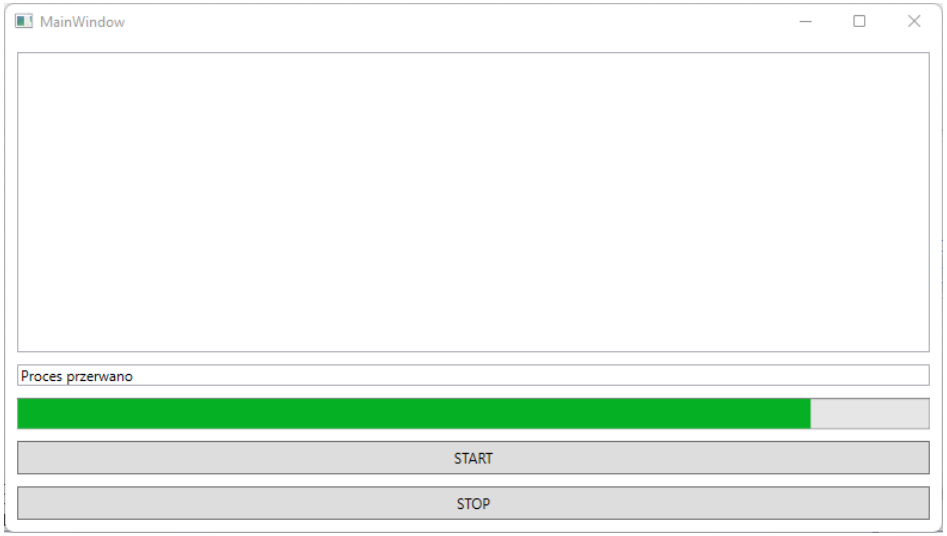
Thread.Sleep (czas);

czas – podany w milisekundach

Dodać należy przestrzeń nazw:

```
using System.Threading;
```

Przykład

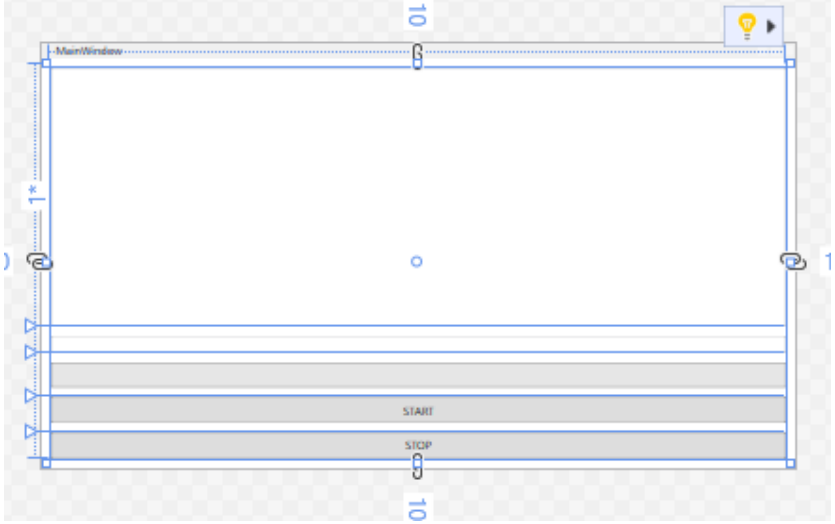


Celowo obliczeniuchłonna metoda generowania permutacji zbioru liczb z zakresu od 0 do n -1.
Liczby generowane są za pomocą losowania bez powtórzeń.

Algorytm generujący liczby działa w tle (w osobnym wątku)

Wątki

Przygotowanie layoutu aplikacji




```
<Grid Margin="10,10,10,10">
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <TextBox x:Name="textBox01" Padding="5,10" AcceptsReturn="True" TextWrapping="Wrap"
VerticalScrollBarVisibility="Auto"/>
        <TextBox x:Name="textBox02"  Grid.Row="1" Margin="0,10,0,0"></TextBox>
        <Border Grid.Row="2" BorderBrush="Black" BorderThickness="0.4" Margin="0,10,0,10">
            <ProgressBar  x:Name="progressBar01"  Height="25" Width="Auto" Value="0"/>
        </Border>
        <Button Grid.Row="3" Padding="30,5"  Margin="0,0,0,10"
Click="Button_Click">START</Button>
        <Button Grid.Row="4" Padding="30,5" Click="Button_Click_1">STOP</Button>
    </Grid>
```

Wątki

Przygotowanie BackroundWorkera

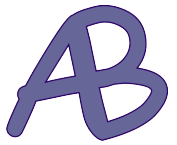
Tworzymy referencje do obiektu klasy `BackgroundWorker`

Tworzymy sam obiekt, a następnie dodajemy do niego 3 główne metody obsługi

```
private BackgroundWorker worker;
public MainWindow()
{
    InitializeComponent();

    worker = new BackgroundWorker
    {
        WorkerReportsProgress = true,
        WorkerSupportsCancellation = true
    };
    worker.DoWork += worker_DoWork;
    worker.ProgressChanged += worker_ProgressChanged;
    worker.RunWorkerCompleted += worker_RunWorkerCompleted;
}
```

Wątki



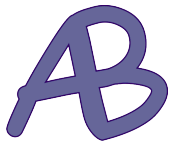
Metoda DoWork

Metoda DoWork – wykonywana w trakcie pracy wątku – jego główne zadania

```
private void worker_DoWork(object? sender, DoWorkEventArgs e)
{
        Random random = new Random();
        DateTime dateTime = DateTime.Now; //pobieramy czas systemowy
        int czas_start = dateTime.Hour*3600 + dateTime.Minute* 60+
        dateTime.Second;
           // czas startowy - liczba sekund, która upłyneła od początku doby
        int ile = 0, x;
        long licznik=0;
        bool powt;
        if (e.Argument != null) ile = (int)e.Argument;
        // sprawdzany, czy metoda wywołana została z argumentem (zakres)
        List<int> list = new List<int>(); //tworzymy pustą listę
```

Dodatkowo sprawdzamy tu czas wykonania obliczeń.

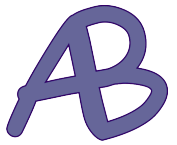
Wątki



Metoda DoWork – c.d.

```
for (int i = 0; i < ile; i++)
{
    if (worker.CancellationPending)
    // reakcja na polecenie przerwania wątku
    {
         e.Cancel = true;
         return;
    }
    //Thread.Sleep(1);
         // aby lepiej prześledzić działanie aplikacji
         // można obliczenia sztucznie spowolnić
    if (i%100==0) worker.ReportProgress((int)Math.Round(((double)list.Count / ile)
* 100));
         // raportujemy postęp – co 100 iteracji, żeby niepotrzebnie nie //
spowalniać programu
    do // losujemy liczbę bez powtórzeń i dodajemy do listy
    {
              powt = false;
              x = random.Next(ile+1);
              foreach (int element in list)
                 if ( element == x) powt = true;
              licznik++;
              if (!powt) list.Add(x);
    } while (powt);
}
```

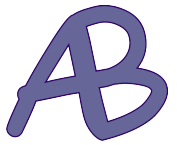
Wątki



Metoda DoWork – c.d.

```
  string wynik="";
  foreach (int i in list)
  wynik += i.ToString() + " ";
        // na podstawie listy generujemy łańcuch przygotowany do wyświetlenia
  dateTime = DateTime.Now;
        // ponownie pobieramy czas systemowy
  int czas = dateTime.Hour * 3600 + dateTime.Minute * 60 + dateTime.Second -
czas_start;
        // liczymy ile sekund upłyneło od czas_start
  e.Result = licznik.ToString()+" "+czas.ToString() +"\n"+ wynik;
        // wygenerowany łańcuch wstawiamy do obiektu e
}
```

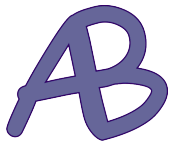
Wątki



Metoda DoWork – c.d.

```
  string wynik="";
  foreach (int i in list)
  wynik += i.ToString() + " ";
        // na podstawie listy generujemy łańcuch przygotowany do wyświetlenia
  dateTime = DateTime.Now;
        // ponownie pobieramy czas systemowy
  int czas = dateTime.Hour * 3600 + dateTime.Minute * 60 + dateTime.Second -
czas_start;
        // liczymy ile sekund upłyneło od czas_start
  e.Result = licznik.ToString()+" "+czas.ToString() +"\n"+ wynik;
        // wygenerowany łańcuch wstawiamy do obiektu e
}
```

Wątki



Metoda RunWorkerComlited

```
private void worker_RunWorkerCompleted(object? sender,
RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
    {
        textBox02.Text = "Proces przerwano";
    }
    else if (e.Error != null)
    {
        textBox02.Text = "Błąd obliczeń";
    }
    else
    {
        progressBar01.Value = 0;
        textBox01.Text = e.Result.ToString();
    }
}}
```

Wątki

Metoda ProgressChanged

```
private void worker_ProgressChanged(object? sender,
ProgressChangedEventArgs e)
{
      textBox02.Text = "Progres=" + e.ProgressPercentage;
      progressBar01.Value = e.ProgressPercentage;
}
```

sterowanie paskiem postępu

Zdarzenie to wywoływane jest w DoWork:
worker.ReportProgress();

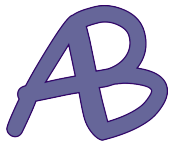
Wątki

Przerwanie wątku

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    worker.CancelAsync();
}
```

Wymaga oprogramowania w DoWork:

Wątki

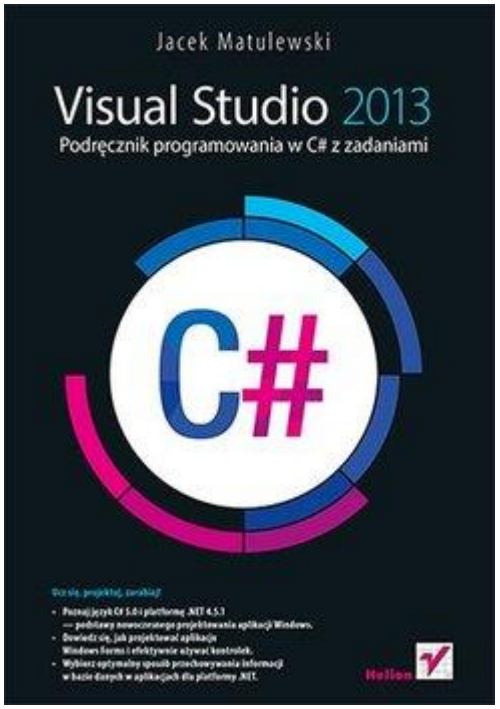


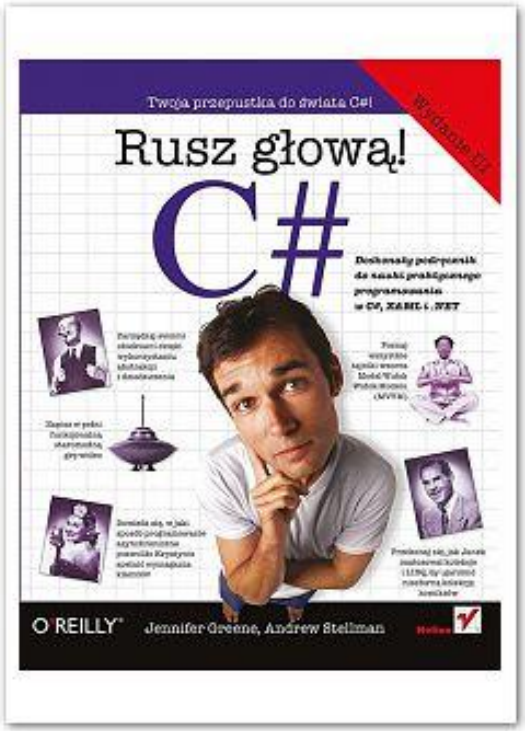
Uruchomienie wątku

```
private void Button_Click(object sender, RoutedEventArgs e)
{
      textBox01.Clear();
      int ile;
      if (!int.TryParse(textBox02.Text, out ile))
          ile = 0;
      worker.RunWorkerAsync(ile);
}
```

Przekazanie parametru w wywołaniu wątku jest opcjonalne

Literatura:





Użyte w tej prezentacji tabelki pochodzą z książki:Visual Studio 2013. Podręcznik programowania w C# z zadaniami
Autor: Matulewski Jacek, Helion