

Instrukcja

Zadanie 1. Obliczyć Symbolu Newtona dla zadanych parametrów N i K.

$$\binom{N}{K} = \frac{N \cdot (N-1) \cdots (N-K+1)}{1 \cdot 2 \cdots K}$$

Licznik i mianownik mają być względem siebie obliczane współbieżnie – nie używać *BackgroundWorker*. Zrealizować 3 różne implementacje wykorzystując:

- A. klasy `Task` i `Task<T>`; dodatkowo do przekazania stanu pomocna może być klasa [Tuple](#) (0.5 pkt)
- B. delegaty do asynchronicznego wywołania metod (1 pkt)
- C. metodę asynchroniczną "async-await" (1 pkt)

Zadanie 2. Obliczyć sekwencyjnie i-ty wyraz ciągu Fibonacciego. Obliczenia powinny odbywać się z wykorzystaniem klasy *BackgroundWorker*. Dodatkowo po każdym nowo obliczonym wyrazie aktualizować pasek postępu (*ProgressBar*). Spowolnić pętlę obliczającą kolejne wyrazy ciągu instrukcją `Thread.Sleep(20)`. (1 pkt)

Zadanie 3. Skompresować/zdekompresować pliki w wskazanym (poprzez `FolderBrowserDialog`) katalogu wykorzystując klasę [GZipStream](#). Każdy plik kompresować współbieżnie (używając zrównoleglonej pętli) do osobnego archiwum dodając rozszerzenie "gz" do nazwy pliku. (1 pkt)

Zadanie 4. Odwzorować nazwy domenowe z tablicy `hostNames` na adresy IP. Adresy odwzorowywać współbieżnie wykorzystując Parallel LINQ i [Dns.GetHostAddresses](#). (0.5 pkt)

```
string[] hostNames = { "www.microsoft.com", "www.apple.com",  
"www.google.com", "www.ibm.com", "cisco.netacad.net",  
"www.oracle.com", "www.nokia.com", "www.hp.com", "www.dell.com",  
"www.samsung.com", "www.toshiba.com", "www.siemens.com",  
"www.amazon.com", "www.sony.com", "www.canon.com", "www.alcatel-  
lucent.com", "www.acer.com", "www.motorola.com" };
```

1A. Do wykonania zadania 1A będą potrzebne klasy

System.Threading.Tasks.Task – zadanie, które nie zwraca wartości

System.Threading.Tasks.Task<TResult> - zadanie, które zwraca wartość

Przykładowy kod tworzenie nowego zadania z przekazywaniem parametrów:

```
Task<int> taskD = Task.Factory.StartNew<int>( //Func<object,int>
    (obj) => {
        int max = (int)obj;
        int sum = 0;
        for (int i = 0; i < max; i++) sum += i;
        return sum;
    },
    100 //state
);
Console.WriteLine(taskD.Result);
```

1B. Do wykonania zadania 1B będziemy potrzebowali:

```
class Program {
    public static int Add(int x, int y) { return x + y; }

    static void Main(string[] args) {
        Func<int, int, int> op = Add;
        IAsyncResult result;
        result = op.BeginInvoke(4, 3, null, null);
        while (result.IsCompleted == false) {
            // Perform additional processing here
            Console.Write(".");
        }
        int sum = op.EndInvoke(result);
        Console.WriteLine(sum);
    }
}

IAsyncResult BeginInvoke(..., AsyncCallback
callback, object state)
```

Funkcja ta rozpoczyna wywołanie asynchroniczne, nie czeka na zakończenie wywołania, zwraca jego wartość od razu.

EndInvoke (... , IAsyncResult)

Funkcja ta czeka, aż wywołanie asynchroniczne zostało ukończone a następnie zwraca jego wartość.

Przykładowy kod programu do użycia funkcji BeginInvoke oraz EndInvoke

1C. Do wykonania ostatniego podpunktu będziemy potrzebowali konstrukcji async-await


Każda metoda, która wykonuje zadanie asynchroniczne musi posiadać słowo kluczowe **async** w swojej deklaracji, aby poinformować kompilator, że chcemy użyć metody asynchronicznej w naszej funkcji.

```
private async void dumpWebPageAsync(string uri) {
```

Następnie, kiedy chcemy wywołać naszą metodę asynchroniczną musimy zastosować słowo kluczowe **await** przed jej wywołaniem. Jest ono używane do funkcji, które mogą wykonywać się dłużej i dlatego powinny zostać wykonane w osobnym wątku. Wskazuje także, że wykonywanie dalszej części instrukcji w danej funkcji powinno zostać wstrzymane do czasu zwrócenia wartości asynchronicznej funkcji.

Przykładowa funkcja async-await:

```
private async void dumpWebPageAsync(string uri) {  
    WebClient webClient = new WebClient();  
    string page = await webClient.DownloadStringTaskAsync(uri);  
    MessageBox.Show(page);  
}
```



Do wykonania zadania 2 będziemy potrzebowali klasy **BackgroundWorker**:

Aby użyć poprawnie tej klasy należy:

1. Stworzyć instancję klasy *BackgroundWorker*.
2. Dodać obsługę eventu *DoWork*.
Powinien on zawierać kod, który ma się wykonać na osobnym wątku. Użyć *BackgroundWorker.ReportProgress(int)* żeby raportować stan operacji. Co jakiś czas sprawdzać *CancellationPendingproperty* i jeżeli jest true to zakończyć zadanie.
3. Dodać obsługę eventu *ProgressChanged*.
Działa ona na głównym wątku UI.
4. Ustawić wartość *WorkerReportsProgress* na true.
5. Dodać obsługę eventu *RunWorkerCompleted*.
6. Wywołać *BackgroundWorker.RunWorkerAsync()* aby wywołać *DoWork*.
7. Opcjonalnie wywołać *BackgroundWorker.CancelAsync()* aby ustawić *CancellationPendingproperty* na true.

Przykładowy kod użycia *BackgroundWorker*:

```

BackgroundWorker bw = new BackgroundWorker();
bw.DoWork += ((object sender, DoWorkEventArgs args) => {
    BackgroundWorker worker = sender as BackgroundWorker;
    int n = (int)args.Argument; //Extract the argument
    int i;
    //Perform long running process
    for (i=0; i< n; i++) worker.ReportProgress(i+1);
    args.Result = i;
});
bw.ProgressChanged += ((object sender, ProgressChangedEventArgs args) => {
    //Update label in UI with progress
    label1.Text = args.ProgressPercentage.ToString();
});
bw.RunWorkerCompleted += ((object sender, RunWorkerCompletedEventArgs args) => {
    //Update the user interface
    MessageBox.Show("Result: " + args.Result);
});
bw.WorkerReportsProgress = true;
bw.RunWorkerAsync(100);

```


Przykładowy wygląd aplikacji:

K	<input type="text" value="5"/>	Tasks	792
N	<input type="text" value="12"/>	Delegates	792
		Async/Await	792

Fibonacci

i

GET



Kompresja

DNS

www.amazon.com =>
52.84.195.27
www.nokia.com =>
23.223.19.119
www.oracle.com =>
104.81.108.164
www.toshiba.com =>
174.35.79.136

Responsywność