Модуль 2. Паралельне програмування з використанням бібліотеки TPL та PLINQ

Метод Dispose. Клас TaskFactory. Застосування лямбда-виразу у якості задачі. Створення продовження задачі. Повернення значення із задачі. Відміна задачі і обробка виключення AggregateException. Інші засоби організації задач.

Лекція 2. Клас TaskFactory

1. Метод Dispose ()

Метод Dispose() реалізується в класі Task, звільняючи ресурси, використовувані цим класом. Як правило, ресурси, пов' язані з класом Task, звільняються автоматично під час "Збирання сміття" (або після закінчення програми). Але якщо ці ресурси потрібно звільнити раніше, то для цього призначений метод Dispose ().

public void Dispose()

Це особливо важливо в тих програмах, де створюється багато задач.

Слід зазначити, що метод Dispose () можна викликати для окремої задачі тільки після її завершення. Отже, для з'ясування факту завершення окремої задачі, перш ніж викликати метод Dispose (), потрібно викликати метод Wait ().

Якщо спробувати викликати Dispose () для активної задачі, то буде згенероване виключення InvalidOperationException.

Для демонстрації можливостей цього методу метод Dispose () буде викликатися явним чином у всіх подальших прикладах програм лекції.

```
// Звільнити задачу  tsk.
tsk.Dispose();
```

2. Клас Task.Factory

Для спрощення створення і виклику задач, а також керування ними, у версії 4 .Net Framework з'явився новий клас Task.Factory. Це статичний клас, тобто містить тільки статичні методи. Основним методом є метод StartNew(), який створює задачу і одразу запускає її на виконання.

Метод StartNew() існує в багатьох формах. Нижче наведена найпростіша форма його оголошення:

public Task StartNew(Action action)

де action - точка входу у виконувану задачу. Спочатку в методі StartNew () автоматично створюється екземпляр об' єкту типу Task для дії, яка визначається параметром action, а потім планується запуск задачі на виконання. Отже, необхідність у виклику методу Start () тепер відпадає.

Наприклад, наступний виклик методу StartNew () в програмах, що розглядалися раніше, призведе до створення і запуску задачі tsk однією дією.

Task tsk = Task.Factory.StartNew(MyTask);

Після цього оператора відразу ж почне виконуватися метод MyTask().

Метод StartNew () виявляється ефективнішим в тих випадках, коли задача створюється і відразу ж запускається на виконання.

Приклад 2.1.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Lab_8
{
    class DemoTask
    {
        // Метод, який виконується як задача,
        static void MyTask()
        {
            Console.WriteLine("MyTask() №" + Task.CurrentId + " запущено");
            for (int count = 0; count < 10; count++)
            {
                Thread.Sleep(500);
                Console.WriteLine("В методі MyTask() #" + Task.CurrentId + ", рахунок
рівний" + count);
            }
            Console.WriteLine("MyTask №" + Task.CurrentId + " завершено");
        }
        static void Main()
        {
            Console.WriteLine("Основний потік запущений.");
            // Створити об'єкти для двох задач.
 //           Task tsk = new Task(MyTask);
//            Task tsk2 = new Task(MyTask);
            // Запустити задачі на виконання.
//            tsk.Start();
//            tsk2.Start();
            Task tsk = Task.Factory.StartNew(MyTask);
            Task tsk2 = Task.Factory.StartNew(MyTask);

            Console.WriteLine("Ідентифікатор задачі tsk: " + tsk.Id);
            Console.WriteLine("Ідентифікатор задачі tsk2: " + tsk2.Id);
            // Зберегти метод Main() активним до завершення решти задач
            for (int i = 0; i < 60; i++)
            {
                Console.Write(".");
                Thread.Sleep(100);
            }
            Console.WriteLine("Основний потік завершено.");
            Console.ReadKey();
        }
    }
}
```

Крім імені методу в задачу можна передати параметр, як в наступному прикладі.

```
static void Main()
{
  var task = Task.Factory.StartNew (Greet, "Hello");
  task.Wait();  // Ожидаем завершения задачи.
}

static void Greet (object state) { Console.Write (state); }   // Hello
```

3. Застосування лямбда-виразу у якості задачі

Окрім використання звичайного іменованого методу у якості задачі, існує і інший підхід: вказати лямбда-вираз як окрему задачу.

Примітка

Лямбда-вирази є особливою формою *анонімних функцій*. Тому вони можуть виконуватися як окремі задачі. Лямбда-вирази особливо корисні в тих випадках, коли єдиним призначенням методу є рішення одноразової задачі. Лямбда-вирази можуть складати окрему задачу або ж викликати інші методи.

У наведеному нижче прикладі програми демонструється застосування лямбда-виразу у якості задачі. У цій програмі код методу MyTask() з попередніх прикладів програм перетворюється в лямбда-вираз.

Приклад 2.2.

// Застосувати лямбда-вираз у якості задачі

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Lab_8
{
    class DemoTaskFactory
    {
        static void Main()
        {
            Console.WriteLine("Основний потік запущено");
            // Далі лямбда-вираз використовується для визначення задачі
            Task tsk = Task.Factory.StartNew(() =>
            {
                Console.WriteLine("Задача запущена");
                for (int count = 0; count < 10; count++)
                {
                    Thread.Sleep(500);
                    Console.WriteLine("Підрахунок в задачі рівний " + count);
                }
                Console.WriteLine("Задача завершена");
            });

            // Чекати завершення задачі tsk.
            tsk.Wait();
```

```
            // Звільнити задачу tsk.
            tsk.Dispose();
            Console.WriteLine("Основний потік завершено");
            Console.ReadKey();
        }
    }
}
```

Окрім застосування лямбда-виразу для опису задачі, зверніть також увагу на те, що метод tsk.Dispose () викликається тільки після повернення з методу tsk.Wait (). Отже, як видно з прикладу, метод Dispose () можна викликати тільки після закінчення задачі.

Для того, щоб переконатися в цьому, спробуйте поставити виклик методу tsk.Dispose () в програмі перед викликом методу tsk.Wait (). Ви відразу ж помітите, що це призведе до виняткової ситуації.

4. Створення продовження задачі

Продовження - це одна задача, яка автоматично починається після завершення іншої задачі. Створити продовження можна, зокрема, за допомогою методу ContinueWith (), визначеного в класі Task.

Нижче наведена проста форма його оголошення:

public Task ContinueWith (Action<Task> дія_продовження)

де дія_продовження означає задачу, яка буде запущена на виконання після закінчення поточної задачі. У делегата Action є один параметр типу Task. Варіант делегата Action, який використаний в цьому методі, має наступний вигляд.

public delegate void Action<int>(T obj)

У цьому випадку узагальнений параметр **T** означає клас Task.

Приклад 2.3. Демонстрація продовження задачі

```
//  Метод, що виконується як задача
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Lab_8
{
    class DemoTask
    {
        static void MyTask()
        {
            Console.WriteLine("MyTask() запущена");
            for (int count = 0; count < 5; count++)
            {
                Thread.Sleep(500);
                Console.WriteLine("У методі MyTaskO підрахунок рівний " + count);
                Console.WriteLine("MyTask завершена");
            }
        }
        //Метод, що виконується як продовження задачі
        static void ContTask(Task t)
```

```
        {
            Console.WriteLine("Продовження запущене");
            for (int count = 0; count < 5; count++)
            {
                Thread.Sleep(500);
                Console.WriteLine("У продовженні підрахунок рівний " + count);
            }
            Console.WriteLine("Продовження завершене");
        }
        static void Main()
        {
            Console.WriteLine("Основний потік запущено");
            // Сконструювати об'єкт першої задачі
            Task tsk = new Task(MyTask);
            //А тепер створити продовження задачі
            Task taskCont = tsk.ContinueWith(ContTask);
            // Почати послідовність задач
            tsk.Start();
            // Чекати завершення продовження.
            taskCont.Wait();
            tsk.Dispose();
            taskCont.Dispose();
            Console.WriteLine("Основний потік завершений");
        }
    }
}
```

Як видно з цього прикладу, друга задача не починається до тих пір, поки не завершиться перша. Зверніть увагу на те, що в методі Main () довелося чекати закінчення тільки продовження задачі. Справа у тому, що метод MyTask () як задача завершується ще до початку методу ContTask як продовження задачі. Отже, чекати завершення методу MyTask () немає ніякої потреби.

Іноді продовження потрібно створити після завершення декількох задач. В цьому випадку слід використовувати метод

ContinueWhenAll

Приклад 2.4

Демонструє створення продовження після завершення трьох задач.

В цьому прикладі створюється три методи обчислення суми чисел `SumMas1(), SumMas2(), SumMas3()`. В продовженні задачі `ContSum()` обчислюється загальна сума.

В методі Main створюються три задачі для виклику цих методів і запускаються на виконання. Результати призначаються об’єктам `tsk1, tsk2, tsk3.` Потім з них формується масив

```
Task[] listTask = new Task[] { tsk1, tsk2, tsk3 };
```

Далі викликається продовження задач.

```
taskF.ContinueWhenAll(listTask, (p) => ContSum(), CancellationToken.None);
```

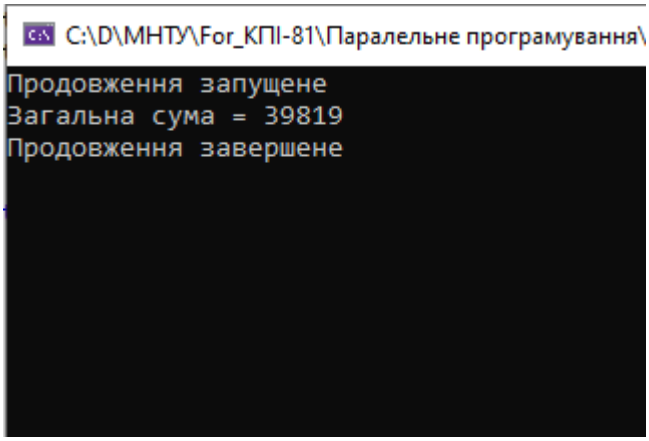
```
using System;
using System.Threading.Tasks;
using System.Threading;
namespace Lab8_2
{
    class Program
    {
        static long SumMas1()
        {
            long[] mas = new long[100];
            long sum = 0;
            Random rand = new Random();
            for (int i = 0; i < mas.Length; i++)
            {
                mas[i] = rand.Next(0, 50);
                sum += mas[i];
            }
            return sum;
        }
        static long SumMas2()
        {
            long[] mas = new long[100];
            long sum = 0;
            Random rand = new Random();
            for (int i = 0; i < mas.Length; i++)
            {
                mas[i] = rand.Next(51, 200);
                sum += mas[i];
            }
            return sum;
        }
        static long SumMas3()
        {
            long[] mas = new long[100];
            long sum = 0;
            Random rand = new Random();
            for (int i = 0; i < mas.Length; i++)
            {
                mas[i] = rand.Next(200, 300);
                sum += mas[i];
            }
            return sum;
        }
        static void ContSum()
        {
            Console.WriteLine("Продовження запущене");
            long sumAll = partSum1 + partSum2 + partSum3;
            Console.WriteLine("Загальна сума = {0}", sumAll);
            Console.WriteLine("Продовження завершене");
        }
        static long partSum1;
        static long partSum2;
        static long partSum3;
        static void Main(string[] args)
        {
            CancellationTokenSource cts = new CancellationTokenSource();
            var taskF = new TaskFactory();
            Task<long> tsk1 = Task.Factory.StartNew(SumMas1);
            Task<long> tsk2 = Task.Factory.StartNew(SumMas2);
            Task<long> tsk3 = Task.Factory.StartNew(SumMas3);
            Task[] listTask = new Task[] { tsk1, tsk2, tsk3 };
            partSum1 = tsk1.Result;
```

```
            partSum2 = tsk2.Result;
            partSum3 = tsk3.Result;

            taskF.ContinueWhenAll(listTask, (p) => ContSum(), CancellationToken.None);

            Console.ReadKey();
        }
    }
    }
```

Результат роботи



5. Повернення значення із задачі

Задача може повертати значення. Це дуже зручно з двох причин.

По-перше, це означає, що за допомогою задачі можна обчислити деякий результат. Таким чином підтримуються паралельні обчислення.

По-друге, процес, який викликається, буде заблокованим до тих пір, поки не буде отриманий результат. Це означає, що для організації очікування результату не потрібно ніякої *особливої синхронізації*.

Для того, щоб повернути результат із задачі, досить створити цю задачу, використовуючи узагальнену форму Task<TResult> класу Task. Нижче наведено два конструктори цієї форми класу Task:

public Task(Func<TResult> функція)

public Task(Func<Object, TResult> функція, Object стан)

де функція означає виконуваний делегат. Зверніть увагу на те, що делегат має бути типу Func, а не Action. Тип Func використовується саме в тих випадках, коли задача повертає результат. У першому конструкторі створюється задача без аргументів, а в іншому конструкторі - задача, що приймає аргумент типу Object, який передається як стан. Є також інші конструктори цього класу.

Також існують інші варіанти методу StartNew(), які доступні в узагальненій формі класу TaskFactory<TResult> і які підтримують повернення результату із задачі. Нижче наведені варіанти цього методу, які застосовуються паралельно з тільки що розглянутими конструкторами класу Task.

public Task<TResult> StartNew(Func<TResult> функція)

public Task<TResult> StartNew(Func<0bject, TResult> функція, Object стан)

У будь-якому разі значення, яке повертається задачею, виходить з властивості Result в класі Task, яка визначається таким чином.

public TResult Result { get; internal set; }

Аксесор set є внутрішнім для цієї властивості, і тому він доступний в зовнішньому коді тільки для читання. Отже, задача отримання результату блокує код до тих пір, поки результат не буде обчислений.

У наведеному нижче прикладі програми демонструється повернення задачею значень. У цій програмі створюються два методи. Перший з них, MyTask (), не приймає параметрів, а просто повертає логічне значення true типу bool. Другий метод, Sumlt(), приймає один параметр, який приводитися до типу int, і повертає суму зі значення, що передається як параметр.

Приклад 2.5.

```
// Повернути значення із задачі
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Lab_8
{
    class DemoTask
    {
        // Простий метод, що повертає результат і не приймає аргументів,
        static bool MyTask()
        {
            return true;
        }
        // Цей метод повертає суму з позитивного цілого значення,
        // яке йому передається як параметр
        static int Sumlt(object v)
        {
            int x = (int)v;
            int sum = 0;
            for (; x > 0; x--)
                sum += x;
            return sum;
        }
        static void Main()
        {
            Console.WriteLine("Основний потік запущений");
            // Сконструювати об'єкт першої задачі
            Task<bool> tsk = Task<bool>.Factory.StartNew(MyTask);
            Console.WriteLine("Результат після виконання задачі MyTask : " +
            tsk.Result);
            // // Сконструювати об'єкт другої задачі
            Task<int> tsk2 = Task<int>.Factory.StartNew(Sumlt, 3);
```

```
            Console.WriteLine("Результат після виконання задачі Sumlt : " +
            tsk2.Result);
            tsk.Dispose();
            tsk2.Dispose();
            Console.WriteLine("Основний потік завершений");
            Console.ReadKey();
        }
    }
}
```

6. Відміна задачі і обробка виключення AggregateException

У версії 4.0 .NET Framework впроваджена нова підсистема, що забезпечує відміну задачі. Ця нова підсистема грунтується на понятті *ознаки відміни*. Ознаки відміни підтримуються в класі Task, серед іншого, за допомогою методу *StartNew ()*.

Ознака відміни є екземпляром об'єкту типу CancellationToken, тобто структури, визначеної в просторі імен System.Threading. У структурі CancellationToken визначено декілька властивостей і методів, але ми скористаємося двома з них. По-перше, це доступна тільки для читання властивість *IsCancellationRequested*, яка оголошується таким чином.

public bool IsCancellationRequested { get; }

Вона повертає логічне значення true, якщо відміна задачі була викликана для ознаки, а інакше - логічне значення false.

І по-друге, це метод ThrowlfCancellationRequested(), який оголошується таким чином.

public void ThrowlfCancellationRequested()

Якщо ознака відміни, для якої викликається цей метод, отримала запит на відміну, то в цьому методі генерується виключення *OperationCanceledException*.

Інакше ніяких дій не виконуються. У відміняючому коді можна організувати відстежування згаданого виключення, щоб переконатися в тому, що відміна завдання дійсно сталася. Як правило, з цією метою спочатку перехоплюється виключення AggregateException, а потім його внутрішнє виключення аналізується за допомогою властивості InnerException або InnerExceptions.

Факт відміни задачі може бути перевірений самими різними способами. У наведеній нижче програмі демонструється відміна задачі. У ній застосовується опитування для контролю стану ознаки відміни. Метод **ThrowlfCancellationRequested()** викликається після входу в метод MyTask (). Це дає можливість завершити задачу, якщо вона була відмінена, ще до її запуску. Усередині циклу перевіряється властивість **IsCancellationRequested**. Якщо ця властивість містить логічне значення true, а воно встановлюється після виклику методу Cancel () для екземпляра

джерела ознак відміни, то на екран виводиться повідомлення про відміну і далі викликається метод ThrowlfCancellationRequested () для відміни задачі.

Приклад 2.6.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
namespace Lab_7
{
    class DemoTask
    {
        // Метод, що виконується як задача,
        static void MyTask(Object ct)
        {
            CancellationToken cancelTok = (CancellationToken)ct;
            // Перевірити, чи скасована задача, перш ніж запускати її.
            cancelTok.ThrowIfCancellationRequested();
            Console.WriteLine("MyTask() запущена");
            for (int count = 0; count < 10; count++)
            {
                // для відстежування відміни задачі застосовується опитування
                if (cancelTok.IsCancellationRequested)
                {
                    Console.WriteLine("Отриманий запит на відміну задачі");
                    cancelTok.ThrowIfCancellationRequested();
                }
                Thread.Sleep(500);
                Console.WriteLine("У методі MyTask() підрахунок рівний " + count);
            }
            Console.WriteLine("MyTask завершена");
        }
        static void Main()
        {
            Console.WriteLine("Основний потік запущений");
            // Створити об'єкт джерела ознак відміни
            CancellationTokenSource cancelTokSrc = new CancellationTokenSource();
            // Запустити задачу, передавши ознаку відміни їй самій і делегату
            Task tsk = Task.Factory.StartNew(MyTask,
cancelTokSrc.Token,cancelTokSrc.Token);
            // Дати задачі можливість виконуватися аж до її відміни.
            Thread.Sleep(2500);
            try
            {
                // Відмінити задачу
                cancelTokSrc.Cancel();
                // Припинити виконання методу Main() доти,
                // поки не завершиться задача tsk.
                tsk.Wait();
            }
            catch (AggregateException exc)
            {
                if (tsk.IsCanceled)
                    Console.WriteLine("Задача tsk відміненап\n");
                //Для перегляду виключення зняти коментарі з наступного рядка коду
                Console.WriteLine(exc);
            }
            finally
            {
                tsk.Dispose();
                cancelTokSrc.Dispose();
                Console.WriteLine("Основний потік завершений");
```

```
            }
            Console.ReadKey();
        }
    }
}
```

Як видно з цього прикладу, виконання методу MyTask () відміняється в методі Main () через лише дві секунди. Отже, в методі MyTask () виконуються чотири кроки циклу. Коли ж перехоплюється виключення AggregateException, перевіряється стан задачі. Якщо задача tsk скасована, що і повинне статися в цьому прикладі, то про це виводиться відповідне повідомлення. Слід зазначити, що коли повідомлення AggregateException генерується у відповідь на відміну задачі, то це ще не свідчить про помилку, а просто означає, що задача була скасована.

7. Інші засоби організації задач

Ми розглянули тільки деякі основні способи організації і виконання задач. Але є і інші корисні засоби. Зокрема, задачі можна робити вкладеними, коли одні задачі можуть створювати інші, або ж породженими, коли вкладені задачі виявляються тісно зв'язаними із задачею, що створює їх.

При створенні задачі є можливість вказати різні додаткові параметри, що впливають на особливості її виконання. Для цієї мети вказується екземпляр об'єкту типу **TaskCreationOptions** в конструкторі класу Task або ж у методі StartNew(). Крім того, в класі Task.Factory доступне ціле сімейство методів FromAsync(), що підтримують модель асинхронного програмування (APM - Asynchronous Programming Model).

В наступному прикладі демонструється керування залежними задачами. За допомогою Task.Factory створюється і планується 6 задач. Задача A буде виконана тільки після завершення задач B і C. Задача C буде виконана тільки після завершення задач F, E або D.

Приклад 2.7.

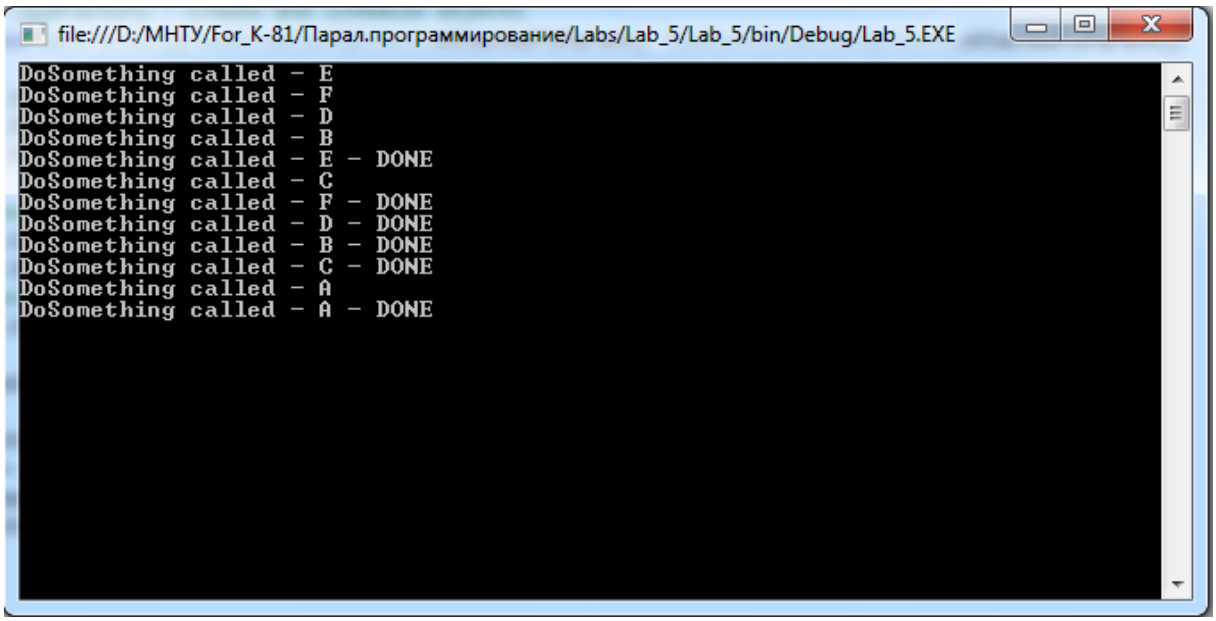
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
namespace Lab_7
{
    class Program
    {
        static void Main(string[] args)
        {
            UseTask demoTask = new UseTask();
            Console.ReadKey();
        }
    }

    class UseTask
    {
```

```
        public UseTask()
        {
            // Set the Task.Factory - Create and schedule objects
            var tf = new TaskFactory(TaskCreationOptions.AttachedToParent,
TaskContinuationOptions.AttachedToParent);

            var F = tf.StartNew(() => DoSomething('F'));
            var E = tf.StartNew(() => DoSomething('E'));
            var D = tf.StartNew(() => DoSomething('D'));

            // C will be started only when F, E or D has been completed
            var C = tf.ContinueWhenAny(new Task[] { F, E, D }, tasks =>
DoSomething('C'));
            var B = tf.StartNew(() => DoSomething('B'));
            // A will only happens once B and C are completed.
            var A = tf.ContinueWhenAll(new Task[] { B, C }, tasks =>
DoSomething('A'));
        }
        private void DoSomething(char @char)
        {
            Console.WriteLine("DoSomething called - {0}", @char);
            Random rd = new Random();
            Thread.Sleep(rd.Next(1000, 3000));
            Console.WriteLine("DoSomething called - {0} - DONE", @char);
        }
    }
}
```



Задачі плануються на виконання екземпляром об'єкту класу TaskScheduler. Як правило, для цієї мети надається планувальник, використовуваний за умовчанням в середовищі .NET Framework. Але цей планувальник може бути налагоджений під конкретні потреби розробника. Крім того, допускається застосування спеціалізованих планувальників завдань.

Висновки

Для спрощення реалізації паралелізму задач в бібліотеці TPL (Task Parallel Library) введено нові класи, основні з яких – це клас **Task і Task.Factory**. На відміну від класу Task, в класі Task.Factory визначений

метод StartNew(), який створює задачу і одразу запускає її на виконання. Метод StartNew () виявляється ефективнішим в тих випадках, коли задача створюється і відразу ж запускається на виконання. Якщо логіка виконання паралельної програми вимагає певного порядку виконання задач, можна створити продовження задачі за допомогою методу ContinueWith (), визначеного в класі Task.

Задача може повертати значення. Таким чином, за допомогою задачі можна обчислити деякий результат. При цьому процес, який викликається, буде заблокованим до тих пір, поки не буде отриманий результат. Це означає, що для організації очікування результату не потрібно ніякої особливої синхронізації як це вимагається при використанні потоків

Контрольні запитання і завдання для самостійного виконання

1. Яке призначення методу Dispose()?
2. Що робить наступний оператор коду
Task tsk = Task.Factory.StartNew(MyTask);
3. Яке призначення класу Task.Factory?
4. Яка різниця між методами Parallel.Invoke () і Task.Factory.StartNew()?
5. Як передати в задачу, запущену методом Task.Factory.StartNew () параметр? Навести приклад.
6. Що таке лямбда-вирази? Коли їх доцільно використовувати у якості задачі?
7. Коли краще застосовувати клас Task, а не клас Task.Factory?
8. Який метод створює продовження задачі?
9. Що таке *продовження* задачі? Коли доцільно створювати продовження?
10. Що означає наступний оператор коду?

```
Task<int> tsk2 = Task<int>.Factory.StartNew(Sumlt, 3);
```

Лабораторна робота 2.