



# БІБЛІОТЕКА TPL (TASK PARALLELISM LIBRARY)

Питання 11.3.

# Бібліотека TPL (Task Parallelism Library)

---

- Задачі – абстракції .NET, які забезпечують блоки асинхронності, як проміси в JavaScript.
  - У початкових версіях .NET доводилося покладатися лише на потоки, які були створені безпосередньо або за допомогою класу `ThreadPool`.
  - Клас `ThreadPool` забезпечував керований рівень абстракції над потоками, але розробники все ще поклалися на клас `Thread` для кращого контролю.
  - Це вимагало написання великої кількості коду, який було важко підтримувати.
  - Клас `Thread` також був некерованим (`unmanaged`).
- Задача – це обгортка над потоком, яка створюється через `ThreadPool`.
  - Задачі надають такі функції, як очікування, скасування та продовження, і вони виконуються після завершення задачі.

# Характеристики задач

---

- Задачі виконуються TaskScheduler, і планувальник за умовчанням просто працює на ThreadPool.
- Ми можемо повернути значення із задач.
- Задачі дають вам знати, коли вони закінчатся, на відміну від ThreadPool або потоків.
- Задачі можна продовжувати за допомогою конструкції ContinueWith ().
- Можна очікувати на виконання задач, викликаючи Task.Wait().
  - Це блокує викликаючий потік, поки очікування не закінчиться.
- Задачі роблять код більш читабельним у порівнянні з потоками чи ThreadPool-об'єктами.
  - Вони також проклали шлях до впровадження конструкції асинхронного програмування в C # 5.0
- Ми можемо встановити стосунки «батьки-діти», коли одна задача починається з іншої.
- Ми можемо поширювати винятки дочірніх задач на батьківські.
- Задачу можна скасувати за допомогою класу CancellationToken.

# Створення та запуск задач

---

- Існує багато способів створення та запуску задач за допомогою бібліотеки TPL:
  - Клас `System.Threading.Tasks.Task`
  - Метод `System.Threading.Tasks.Task.Factory.StartNew`
  - Метод `System.Threading.Tasks.Task.Run`
  - `System.Threading.Tasks.Task.Delay`
  - `System.Threading.Tasks.Task.Yield`
  - Метод `System.Threading.Tasks.Task.FromResult<T>`
  - `System.Threading.Tasks.Task.FromException` та `Task.FromException<T>`
  - `System.Threading.Tasks.Task.FromCancelled` та `Task.FromCancelled<T>`

# Клас System.Threading.Tasks.Task

---

- Спосіб виконання роботи асинхронно (як і потік ThreadPool), заснований на Task-Based Asynchronous Pattern (TAP).
  - Неузагальнений клас Task не повертає результатів, тому в разі потреби в поверненні використовуйте Task<T>.
  - Задачі, що створюються за допомогою класу Task, не плануються для виконання, поки не викликано метод Start().
  - Викликати метод Start() можна лише для тих задач, які раніше не запускались.
  - Для перезапуску виконаної задачі потрібно створити нову і для неї викликати метод Start().
- Створення задач за допомогою лямбда-виразів: викликаємо конструктор Task та передаємо в нього лямбда-вираз з методом для виконання:

```
Task task = new Task (() => PrintNumber10Times ());  
task.Start();
```

- Створення задач за допомогою Action-делегата:

```
Task task = new Task (new Action (PrintNumber10Times));  
task.Start();
```

- Створення задач за допомогою делегата:

```
Task task = new Task (delegate {PrintNumber10Times ();});  
task.Start();
```

# Методи `System.Threading.Tasks.Task.Factory.StartNew` та `System.Threading.Tasks.Task.Run`

---

- Так задача створюється та планується для виконання всередині `ThreadPool`-об'єкта, а посилання на відповідний `Task`-об'єкт повертається викликаючій стороні.
- Створимо задачу за допомогою
  - лямбда-виразів: `Task.Factory.StartNew(() => PrintNumber10Times());`
  - `Action`-делегата: `Task.Factory.StartNew(new Action( PrintNumber10Times));`
  - делегата: `Task.Factory.StartNew(delegate { PrintNumber10Times(); });`
- Метод `Task.Run()` працює так же, як і метод `StartNew()` і повертає `ThreadPool`-потік.
- Створимо задачу за допомогою:
  - Лямбда-виразу: `Task.Run(() => PrintNumber10Times());`
  - `Action`-делегата: `Task.Run(new Action (PrintNumber10Times));`
  - Делегата: `Task.Run(delegate {PrintNumber10Times ();});`

# Метод System.Threading.Tasks.Task.Delay

---

- Можемо створити задачу, яка завершується після заданого інтервалу часу або може скасовуватись користувачем у будь-який момент за допомогою класу CancellationToken.
  - Раніше використовувався метод Thread.Sleep() для створення блокуючих конструкцій для очікування на інші задачі.
  - Проблема: метод використовує ресурси ЦП і працює синхронно.
  - Task.Delay() – краща альтернатива для очікування задач без задіявання ЦП, яка працює асинхронно (використовує системний годинник):

```
Console.WriteLine("What is the output of 20/2. We will show result in 2 seconds.");  
Task.Delay(2000);  
Console.WriteLine("After 2 seconds delay");  
Console.WriteLine("The output is 10");
```

# Метод `System.Threading.Tasks.Task.Yield`

---

- Ще один спосіб створення `await`-задачі.
  - Відповідна задача недоступна напрямку викликаючій стороні, проте використовується в деяких сценаріях, пов'язаних з асинхронним програмуванням та виконанням програми.
  - Це більше обіцянка, ніж задача: використовуючи `Task.Yield()`, можна змусити метод бути асинхронним та повертати управління ОС.
  - Коли решта методу виконується в часі пізніше, він все ще може працювати асинхронно. Цього можна досягти так:

```
await Task.Factory.StartNew(() => {},  
    CancellationToken.None,  
    TaskCreationOptions.None,  
    SynchronizationContext.Current != null?  
        TaskScheduler.FromCurrentSynchronizationContext() :  
        TaskScheduler.Current);
```

- Такий підхід може використовуватись для забезпечення респонсивності UI, час від часу передаючи управління UI-потoku всередині довготривалих задач.
  - Проте такий підхід не є переважним у додатках з UI.



# Метод System.Threading.Tasks.Task.Yield

Threads window showing 4 threads (Process ID: 2284):

ID	Managed ID	Category	Name	Location
11872	1	Main Thread	Main Thread	System.Console.dll!System.ConsolePal.WindowsConsoleStream.ReadFileNative
5752	4	Worker Thread	Worker Thread	<not available>
1664	6	Worker Thread	Worker Thread	Ch02.dll!Ch02_1CreatingAndStartingTasks.TaskYield
2300	0	Worker Thread	Worker Thread	<not available>

Code editor showing the implementation of Task.Yield() in Ch02\_1CreatingAndStartingTasks.cs:

```
28 TaskYield();
29 Console.ReadLine();
30 }
31
32 private async static void TaskYield()
33 {
34     for (int i = 0; i < 100000; i++)
35     {
36         Console.WriteLine(i);
37         if (i % 1000 == 0)
38             await Task.Yield();
39     }
40 }
```

Execution time for the yield call: ≤ 609ms elapsed

Кращі альтернативи: Application.DoEvents() у WinForms та Dispatcher.Yield (DispatcherPriority.ApplicationIdle) у WPF:

```
private async static void TaskYield()
{
    for (int i = 0; i < 100000; i++)
    {
        Console.WriteLine(i);
        if (i % 1000 == 0)
            await Task.Yield();
    }
}
```

- У разі консольних чи веб-додатків, коли запускаємо код і застосовуємо точку розриву на task's yield, все ще бачимо випадкові потоки з пулу, які перемикають контекст для виконання коду.

Якщо натиснути F5 та дозволити точці розриву отримати інше значення  $i$ , бачимо, що код став виконуватись іншим потоком з ID = 10244

The screenshot displays the Visual Studio IDE. At the top, the 'Threads' window shows a list of threads for Process ID: 2284 (5 threads). The thread with ID 10244 is highlighted, showing it is a Worker Thread executing 'Ch02.dll!Ch02\_1CreatingAndStartingTasks.TaskYield'. Below this, the '1CreatingAndStartingTasks.cs' file is open, showing the 'Ch02\_1CreatingAndStartingTasks' class. The code includes a 'TaskYield()' method that calls 'Console.ReadLine()'. A breakpoint is set at line 38, 'await Task.Yield();', which is currently highlighted. A tooltip indicates that 578ms elapsed at this point. The code also includes a loop from 0 to 100,000, writing each value to the console and yielding every 1,000 iterations.

ID	Managed ID	Category	Name	Location
11872	1	Main Thread	Main Thread	System.Console.dll!System.ConsolePal.WindowsConsoleStream.ReadFileNative
1664	6	Worker Thread	Worker Thread	<not available>
15296	0	Worker Thread	Worker Thread	<not available>
10244	5	Worker Thread	Worker Thread	Ch02.dll!Ch02_1CreatingAndStartingTasks.TaskYield
12116	0	Worker Thread	Worker Thread	<not available>

```
28 TaskYield();
29 Console.ReadLine();
30 }
31
32 private async static void TaskYield()
33 {
34     for (int i = 0; i < 100000; i++)
35     {
36         Console.WriteLine(i);
37         if (i % 1000 == 0)
38             await Task.Yield();
39     }
40 }
```

# Метод System.Threading.Tasks.Task.FromResult<T>

---

```
static void Main(string[] args)
{
    StaticTaskFromResultUsingLambda();
}
private static void StaticTaskFromResultUsingLambda()
{
    Task<int> resultTask = Task.FromResult<int>( Sum(10));
    Console.WriteLine(resultTask.Result);
}
private static int Sum (int n)
{
    int sum=0;
    for (int i = 0; i < 10; i++)
    {
        sum += i;
    }
    return sum;
}
```

- Цей підхід представлено в .NET framework 4.5.
  - Можемо повертати завершену задачу.
  - З коду видно, що синхронний метод Sum() повертатиме результати в асинхронній манері, застосовуючи клас Task.FromResult<int>.
  - Даний підхід часто використовується в TDD для мокінгу асинхронних методів, а також всередині асинхронних методів для повернення значень за умовчанням при галуженні (умовах).

# Методи `System.Threading.Tasks.Task.FromException` та `System.Threading.Tasks.Task.FromException<T>`

---

- Створюють задачі, які завершаться заздалегідь визначеним винятком.
  - Використовуються для викидання винятків з асинхронних задач, а також у TDD.

```
return Task.FromException<long>(
    new FileNotFoundException("Invalid File name.));
```

- Тут ми огортаємо `FileNotFoundException` в задачу та повертаємо її викликаючій стороні.

# Методи `System.Threading.Tasks.Task.FromCanceled` і `System.Threading.Tasks.Task.FromCanceled<T>`

---

- Використовуються для створення задач, які завершуються в результаті скасування за допомогою токена (cancellation token):
  - ```
CancellationTokenSource source = new CancellationTokenSource();  
var token = source.Token;  
source.Cancel();  
Task task = Task.FromCanceled(token);  
Task<int> canceledTask = Task.FromCanceled<int>(token);
```
- У коді ми створили токен за допомогою класу `CancellationTokenSource`.
  - Далі створюється задача з цього токена.
  - Важливо: токен потрібно скасувати до того, як використовувати його в методі `Task.FromCanceled()`.
  - Такий підхід корисний, якщо потрібно повернути значення з асинхронних методів, а також при TDD.

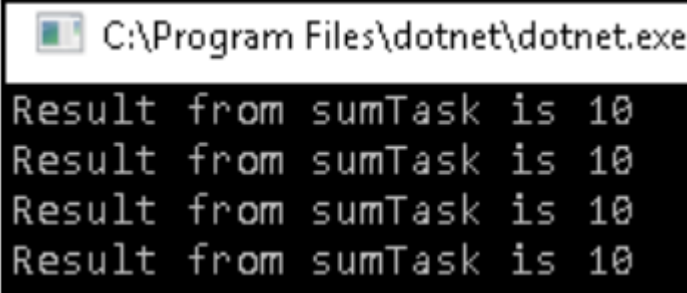
```

using System;
using System.Threading.Tasks;
namespace Ch02
{
    class _2GettingResultFromTasks
    {
        static void Main(string[] args)
        {
            GetResultsFromTasks();
            Console.ReadLine();
        }
        private static void GetResultsFromTasks()
        {
            var sumTaskViaTaskOfInt = new Task<int>(() => Sum(5));
            sumTaskViaTaskOfInt.Start();
            Console.WriteLine($"Result from sumTask is
                {sumTaskViaTaskOfInt.Result}");
            var sumTaskViaFactory = Task.Factory.StartNew<int>(() =>
                Sum(5));
            Console.WriteLine($"Result from sumTask is
                {sumTaskViaFactory.Result}");
            var sumTaskViaTaskRun = Task.Run<int>(() => Sum(5));
            Console.WriteLine($"Result from sumTask is
                {sumTaskViaTaskRun.Result}");
            var sumTaskViaTaskResult = Task.FromResult<int>(Sum(5));
            Console.WriteLine($"Result from sumTask is
                {sumTaskViaTaskResult.Result}");
        }
        private static int Sum(int n)
        {
            int sum = 0;
            for (int i = 0; i < n; i++)
            {
                sum += i;
            }
            return sum;
        }
    }
}

```

## Отримання результатів від завершених задач

- Щоб повернути значення з задач, TPL постачає узагальнений варіант раніше розглянутих класів:
  - `Task<T>`
  - `Task.Factory.StartNew<T>`
  - `Task.Run<T>`
- Коли задача завершується, слід мати можливість одержати результати з неї, отримуючи доступ до властивості `Task.Result`.
  - У коді створимо кілька задач та спробуємо повернути значення від них при завершенні:



```

C:\Program Files\dotnet\dotnet.exe
Result from sumTask is 10
Result from sumTask is 10
Result from sumTask is 10
Result from sumTask is 10

```

# Скасування задач

---

- .NET Framework постачає 2 класи з підтримкою скасування:
  - CancellationTokenSource: відповідає за створення токенів та передачу запиту на скасування для всіх токенів, створених джерелом (source).
  - CancellationToken: використовується слухачами (listeners) для моніторингу поточного стану запиту
- Для створення задач, які можна скасовувати, потрібно виконати наступні кроки:
  - 1. Створити екземпляр класу System.Threading.CancellationTokenSource, який потім надає System.Threading.CancellationToken за допомогою властивості Token.  
**CancellationTokenSource tokenSource = new CancellationTokenSource();  
CancellationToken token = tokenSource.Token;**
  - 2. Передати токен при створенні задачі.  
**var sumTaskViaTaskOfInt = new Task<int>(() => Sum(5), token);  
var sumTaskViaFactory = Task.Factory.StartNew<int>(() => Sum(5), token);  
var sumTaskViaTaskRun = Task.Run<int>(() => Sum(5), token);**
  - 3. При потребі викликати метод Cancel() об'єкту CancellationTokenSource.

# Скасування задач

---

- За допомогою TPL можемо викликати метод `Cancel()`, який буде джерелом токена.
  - У свою чергу, він встановлює властивість `IsCancellationRequested` для токена.
  - Методу в основі, який виконується задачею, слід спостерігати за цією властивістю та елегантно завершувати роботу, якщо її значення задане.
- Існує кілька способів стеження за тим, чи джерело токена подало запит на скасування:
  - Опитування статусу токена за допомогою властивості `IsCancellationRequested`
  - Реєстрація методу зворотного виклику (callback) для скасування запиту
- Опитування статусу токена корисне в випадках, що включають рекурсивні методи чи методи з довготривалою обчислювальною логікою в циклах.
  - У нашому методі чи циклах записується код, який опитує `IsCancellationRequested` через певні оптимальні інтервали.
  - Якщо вона задана, то перериває цикл, викликаючи метод `ThrowIfCancellationRequested` класу токена.



# Скасування задач

---

```
private static void CancelTaskViaPoll()
{
    CancellationTokensSource cancellationTokensSource =
        new CancellationTokensSource();
    CancellationToken token = cancellationTokensSource.Token;
    var sumTaskViaTaskOfInt = new Task(() =>
        LongRunningSum(token), token);
    sumTaskViaTaskOfInt.Start();
    //Wait for user to press key to cancel task
    Console.ReadLine();
    cancellationTokensSource.Cancel();
}

private static void LongRunningSum(CancellationToken token)
{
    for (int i = 0; i < 1000; i++)
    {
        //Simulate long running operation
        Task.Delay(100);
        if (token.IsCancellationRequested)
            token.ThrowIfCancellationRequested();
    }
}
```

- Було створено токен за допомогою класу `CancellationTokensSource`.
  - Потім створено задачу, передавши токен.
  - Задача виконує довготривалий метод `LongRunningSum`, який продовжує опитувати властивість `IsCancellationRequested` токена.
  - Він викидає виняток, якщо користувач викликав `cancellationTokensSource.Cancel()` до завершення методу.
- Опитування практично не накладає затрат продуктивності та може використовуватись відповідно до ваших вимог.
  - Зокрема, коли маємо повний контроль над роботою, яка виконується задачею, such as if it's core logic that you wrote yourself.

# Скасування задач

---

```
private static void DownloadFileWithoutToken()
{
    WebClient webClient = new WebClient();
    webClient.DownloadStringAsync(new
        Uri("http://www.google.com"));
    webClient.DownloadStringCompleted += (sender, e) =>
    {
        if (!e.Cancelled)
            Console.WriteLine("Download Complete.");
        else
            Console.WriteLine("Download Cancelled.");
    };
}
```

- Реєстрація запиту на скасування використовує делегат Callback, який викликається, коли запит скасування подається токеном.
  - Слід використовувати цей підхід з операціями, які блокуються так, щоб було неможливо перевірити значення CancellationToken звичайним способом.
- Як тільки викличемо метод DownloadStringAsync класу WebClient, the control leaves the user.
  - Хоч клас WebClient дозволяє скасувати задачу за допомогою методу webClient.CancelAsync(), ми не контролюємо, коли викликати його.

# Скасування задач

---

```
static void Main(string[] args)
{
    CancellationTokensSource cancellationTokensSource = new
        CancellationTokensSource();
    CancellationToken token = cancellationTokensSource.Token;
    DownloadFileWithToken(token);
    //Random delay before we cancel token
    Task.Delay(2000);
    cancellationTokensSource.Cancel();
    Console.ReadLine();
}

private static void DownloadFileWithToken(CancellationToken token)
{
    WebClient webClient = new WebClient();
    //Here we are registering callback delegate that will get called
    //as soon as user cancels token
    token.Register(() => webClient.CancelAsync());
    webClient.DownloadStringAsync(new
        Uri("http://www.google.com"));
    webClient.DownloadStringCompleted += (sender, e) => {
        //Wait for 3 seconds so we have enough time to cancel task
        Task.Delay(3000);
        if (!e.Cancelled)
            Console.WriteLine("Download Complete.");
        else
            Console.WriteLine("Download Cancelled.");
    };
}
```

- Даний код можна змінити так, щоб використати делегат Callback і мати більше контролю над скасуванням задачі.
  - У новій версії ми передали токен та підписались на cancellation callback за допомогою методу Register.
  - Як тільки користувач викликає метод cancellationTokensSource.Cancel(), він скасує операцію завантаження за допомогою webClient.CancelAsync().
  - CancellationTokensSource також добре працює з legacy ThreadPool.QueueUserWorkItem:
  - // створюємо джерело токена.  
CancellationTokensSource cts = new CancellationTokensSource();  
  
// передаємо токен у cancellable-операцію  
ThreadPool.QueueUserWorkItem(new WaitCallback(DoSomething), cts.Token);

# Очікування завершення запущених задач

---

- Раніше ми викликали властивість `Task.Result`, щоб отримати результат заведеної задачі.
  - Це блокує викликаючий потік, поки результат не стане доступним.
  - TPL надає інший спосіб очікування одного або кількох задач.
- Існує кілька відповідних API, доступних в TPL:
  - `Task.Wait`
  - `Task.WaitAll`
  - `Task.WaitAny`
  - `Task.WhenAll`
  - `Task.WhenAny`

# Очікування завершення запущених задач. Task.Wait

---

- Метод екземпляру, який можна застосувати для очікування однієї задачі.
  - Можемо задати максимальний період часу, для якого викликаюча сторона буде очікувати на завершення задачі до самоблокування з timeout-винятком.
  - Також можемо мати повний контроль над моніторингом подій that have been canceled by passing a cancellation token to the method.
  - Викликаючий метод блокуватиметься, поки потік завершиться, скасується або викине виняток:

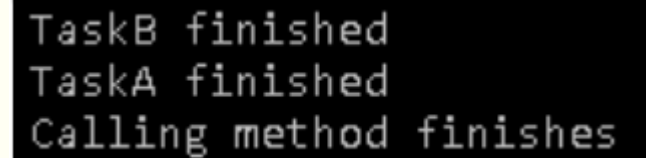
```
var task = Task.Factory.StartNew(() => Console.WriteLine("Inside Thread"));  
//Blocks the current thread until task finishes.  
task.Wait();
```
- Маємо 5 перевантажених версій методу Wait() :
  - **Wait()**: нескінченно очікує завершення задачі. Викликаючий потік блокується, поки дочірній потік не завершено.
  - **Wait(CancellationToken)**: очікує, поки задача завершить виконання for the task to finish execution indefinitely or when the cancellation token is canceled.
  - **Wait(int)**: Waits for the task to finish execution within a specified period of time, in milliseconds.
  - **Wait(TimeSpan)**: Waits for the task to finish execution within a specified time interval.
  - **Wait(int, CancellationToken)**: очікує for the task to finish execution within a specified period of time, in milliseconds, or when the cancellation token is canceled.

# Очікування завершення запущених задач. Task.WaitAll

---

- Статичний метод, який використовується для очікування багатьох задач.
  - Задачі передаються масивом у метод, а викликаюча сторона блокується, поки всі задачі не завершаться.
  - Цей метод також підтримує timeout і скасувальні токени. Приклади використання:

```
Task taskA = Task.Factory.StartNew(() => Console.WriteLine("TaskA finished"));
Task taskB = Task.Factory.StartNew(() => Console.WriteLine("TaskB finished"));
Task.WaitAll(taskA, taskB);
Console.WriteLine("Calling method finishes");
```

A terminal window with a black background and white text. It displays three lines of output: "TaskB finished", "TaskA finished", and "Calling method finishes". This demonstrates that Task.WaitAll blocks the calling method until all tasks (A and B) are completed, even if they finish in a different order than they were started.

```
TaskB finished
TaskA finished
Calling method finishes
```

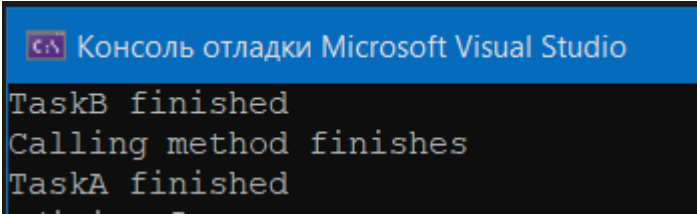
- Сценарій використання методу: коли потрібні дані з кількох джерел (we have one task for each source) і ми бажаємо скомбінувати дані від усіх задач так, щоб їх можна було відображати в UI.

# Очікування завершення запущених задач. Task.WaitAny

---

- Ще один статичний метод, визначений у класі Task.
  - Як і WaitAll, метод WaitAny застосовується для очікування кількох задач, проте викликаюча сторона буде розблокована, як тільки довільна з задач, переданих як масив у метод, завершує виконання.
  - Як і інші методи, WaitAny підтримує timeout та скасувальні токени.
  - Приклади:

```
Task taskA = Task.Factory.StartNew(() => Console.WriteLine("TaskA finished"));
Task taskB = Task.Factory.StartNew(() => Console.WriteLine("TaskB finished"));
Task.WaitAny(taskA, taskB);
Console.WriteLine("Calling method finishes");
```



```
Консоль отладки Microsoft Visual Studio
TaskB finished
Calling method finishes
TaskA finished
```

- Метод WaitAny блокує поточний потік.
  - Як тільки будь-яка із задач завершується, викликаючий потік розблокується.
  - Прикладом використання може бути ситуація, коли потрібні дані доступні з кількох різних джерел і мають бути отримані якомога швидше.
  - Here, we create tasks that make requests to different sources.
  - As soon as any of the tasks finish, we will unblock the calling thread and get the result from the finished task.

# Очікування завершення запущених задач. Task.WhenAll і Task.WhenAny

---

- Task.WhenAll – неблокуючий варіант методу WaitAll().
  - Повертає задачу, яка представляє очікуючу дію для всіх визначених задач.
  - На відміну від WaitAll(), який блокує викликаючий потік, WhenAll може be awaited всередині асинхронного методу, звільняючи викликаючий потік для виконання інших операцій.
- Task.WhenAny – неблокуючий варіант WaitAny().
  - Повертає задачу, яка інкапсулює очікуючу дію для єдиної underlying задачі.
  - Викликаючий потік може викликати await для неї всередині асинхронного методу.



# Продовження виконання задач

---

- Працює схоже до промісів.
  - Для використання потрібно об'єднати кілька задач у ланцюг.
  - Друга задача розпочинатиметься, коли перша завершиться, і результат першої задачі чи винятки передаються дочірній задачі.
- Можемо з'єднати в ланцюг багато задач або створити селективний ланцюг продовження за допомогою методів з TPL:
  - `Task.ContinueWith`
  - `Task.Factory.ContinueWhenAll`
  - `Task.Factory.ContinueWhenAll<T>`
  - `Task.Factory.ContinueWhenAny`
  - `Task.Factory.ContinueWhenAny<T>`

# Продовження виконання задач за допомогою методу Task.ContinueWith()

---

```
var task = Task.Factory.StartNew<int>(() => GetData())  
    .ContinueWith((i) => GetMoreData(i.Result))  
    .ContinueWith((j) => DisplayData(j.Result));
```

- Управляти продовженням задач можливо, передаючи перелічення `System.Threading.Tasks.TaskContinuationOptions` як параметр з наступними опціями:
  - ***None***: значення за умовчанням. Задача-продовження запрацює, коли основна задача завершиться.
  - ***OnlyOnRanToCompletion***: задача-продовження (continuation task) запуститься після успішного завершення основної задачі (не скасована та без збоїв).
  - ***NotOnRanToCompletion***: задача-продовження запрацює, коли основна задача була скасована чи збійна.
  - ***OnlyOnFaulted***: задача-продовження запрацює, коли основна задача зазбоїла.
  - ***NotOnFaulted***: задача-продовження запрацює, коли основна задача не зазбоїла.
  - ***OnlyOnCancelled***: задача-продовження запрацює, коли основна задача була скасована.
  - ***NotOnCancelled***: задача-продовження запрацює, коли основна задача не була скасована.

# Продовження виконання задач за допомогою Task.Factory.ContinueWhenAll та Task.Factory.ContinueWhenAll<T>

---

```
private async static void ContinueWhenAll()
{
    int a = 2, b = 3;
    Task<int> taskA = Task.Factory.StartNew<int>(() => a * a);
    Task<int> taskB = Task.Factory.StartNew<int>(() => b * b);
    Task<int> taskC = Task.Factory.StartNew<int>(() => 2 * a * b);
    var sum = await Task.Factory.ContinueWhenAll<int>(new Task[]
    { taskA, taskB, taskC }, (tasks)
    => tasks.Sum(t => (t as Task<int>).Result));
    Console.WriteLine(sum);
}
```

- Можемо очікувати на кілька задач та з'єднувати їх за умови успішного завершення усіх.
  - У коді потрібно обчислити  $a*a + b*b + 2 * a * b$ .
  - Розбиваємо задачу на 3 частини:  $a*a$ ;  $b*b$ ;  $2*a*b$ .
  - Кожна з них виконується різними потоками: taskA, taskB, taskC.
  - Очікуємо на виконання всіх 3 задач та передаємо їх як перший параметр методу ContinueWhenAll().
  - Коли всі потоки завершать виконання, задіюється делегат-продовження, визначений у другому параметрі методу ContinueWhenAll().
  - Даний делегат додає результати виконання з усіх потоків та передає суму викликаючій стороні, яка виведе цю суму в консоль.

## Продовження виконання задач за допомогою Task.Factory.ContinueWhenAny та Task.Factory.ContinueWhenAny<T>

---

```
private static void ContinueWhenAny() {  
    int number = 13;  
    Task<bool> taskA = Task.Factory.StartNew<bool>(() => number / 2 != 0);  
    Task<bool> taskB = Task.Factory.StartNew<bool>(() => (number / 2) * 2 != number);  
    Task<bool> taskC = Task.Factory.StartNew<bool>(() => (number & 1) != 0);  
    Task.Factory.ContinueWhenAny<bool>(new Task<bool>[] { taskA, taskB, taskC },  
        (task) => {  
            Console.WriteLine((task as Task<bool>).Result);  
        }  
    );  
}
```

- У коді 3 різні частини логіки перевіряють, чи непарне число.
  - Нехай буде невідомо, яка з цих 3 частин буде швидшою.
  - Щоб обчислити результат, створимо 3 задачі з відповідною логікою та запустимо їх конкурентно.
  - Логічно отримати лише перший готовий результат, а решту відкинути.
  - Для цього використовується метод ContinueWhenAny().

# Батьківські та дочірні (вкладені) задачі

---

- Інший тип взаємодії між потоками – відношення «parent-child».
  - Дочірня задача створюється як вкладена задача в тіло батьківської.
  - Дочірню задачу можна створювати приєднаною (attached) або від'єднаною (detached).
  - Обидва види задач створюються всередині батьківської задачі.
  - За умовчанням створені задачі є від'єднаними, для їх приєднання слід присвоїти властивості `AttachedToParent` значення `true`.
- Прикріплену задачу можна розглядати в таких сценаріях:
  - Всі викинуті в дочірній задачі винятки повинні розповсюджуватись у батьківську задачу.
  - Статус батьківської задачі залежить від дочірньої задачі.
  - Батьківській задачі потрібно очікувати на завершення дочірньої задачі

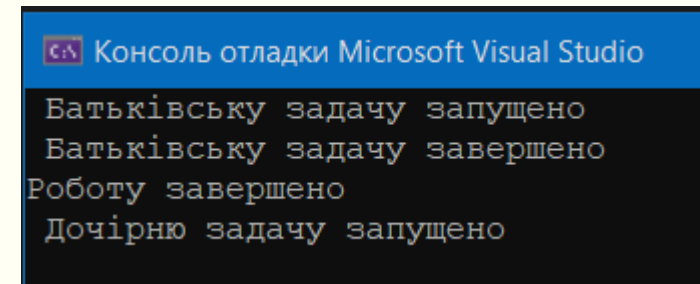
# Створення від'єднаної (detached) задачі

---

- За умовчанням, дочірня або вкладена задача створюється від'єднаною.
  - У кодї батьківська задача може не очікувати на завершення дочірньої та закінчує роботу першою:

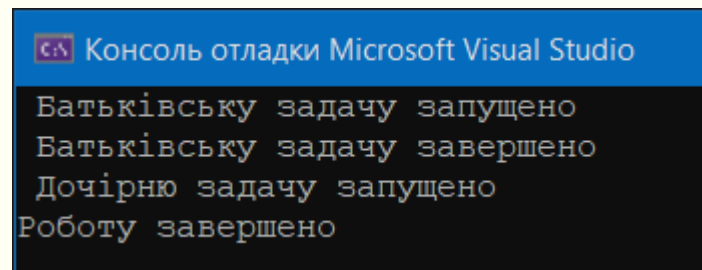
```
Task parentTask = Task.Factory.StartNew(() =>
{
    Console.WriteLine(" Батьківську задачу запущено");
    Task childTask = Task.Factory.StartNew(() => {
        Console.WriteLine(" Дочірню задачу запущено");
    });
    Console.WriteLine(" Батьківську задачу завершено");
});

Console.WriteLine("Роботу завершено");
```



```
Консоль отладки Microsoft Visual Studio
Батьківську задачу запущено
Батьківську задачу завершено
Роботу завершено
Дочірню задачу запущено
```

- Батьківська задача повинна завершуватись, тому викликаємо `parentTask.Wait()` перед останнім виводом:



```
Консоль отладки Microsoft Visual Studio
Батьківську задачу запущено
Батьківську задачу завершено
Дочірню задачу запущено
Роботу завершено
```

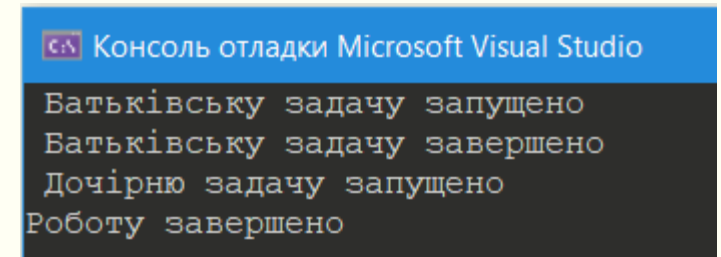
# Створення приєднаної (attached) задачі

---

- Приєднана задача створюється схоже до від'єднаної, проте зі встановленою властивістю `AttachedParent` значенням `true`.
  - Батьківська задача не завершує свою роботу, поки дочірня задача не завершила виконання.

```
Task parentTask = Task.Factory.StartNew(() =>
{
    Console.WriteLine(" Батьківську задачу запущено");
    Task childTask = Task.Factory.StartNew(() => {
        Console.WriteLine(" Дочірню задачу запущено");
    }, TaskCreationOptions.AttachedToParent
    );
    Console.WriteLine(" Батьківську задачу завершено");
});

// очікування завершення батьківської задачі
parentTask.Wait();
Console.WriteLine("Роботу завершено");
```



Консоль отладки Microsoft Visual Studio

```
Батьківську задачу запущено
Батьківську задачу завершено
Дочірню задачу запущено
Роботу завершено
```



---

# ДЯКУЮ ЗА УВАГУ!

**Наступна тема: Паралельне виконання коду**

---