Module

Asynchronous Programming

Table of contents

- Introduction
- Basic Example
- Task-based Asynchronous Pattern (TAP)
- Example

Introduction

- The function calling is made in a synchronous way
  - When no other instruction has to be executed before the end of the execution of the function
  - But some functions need a lot of time
    - Web access
    - File access
    - Communication by e-mail
    - …

→ Asynchronous Method

Introduction

- The.Net API offers often two versions of a method
  - Synchronous method
    - The whole app must wait!
    - The most harmful visible consequence :
      - The UI Thread is blocked
      - No more possible for the user to interact with the app
  - Asynchronous method
    - It asks to the thread pool to make the operation and then returns immediately to the calling method
    - The app can continue with another work which does not depend on the method until the potentially blocking task is ended [See for more details : http://msdn.microsoft.com/en-us/library/h4732ks0.aspx]

Introduction

- Synchronous Method

```
private void Download_Click(object sender, EventArgs e)
{
                var client = new WebClient();
                string data = client.DownloadString("http://....");
                Data.Text = data;
}
```

UI is blocked!

Introduction

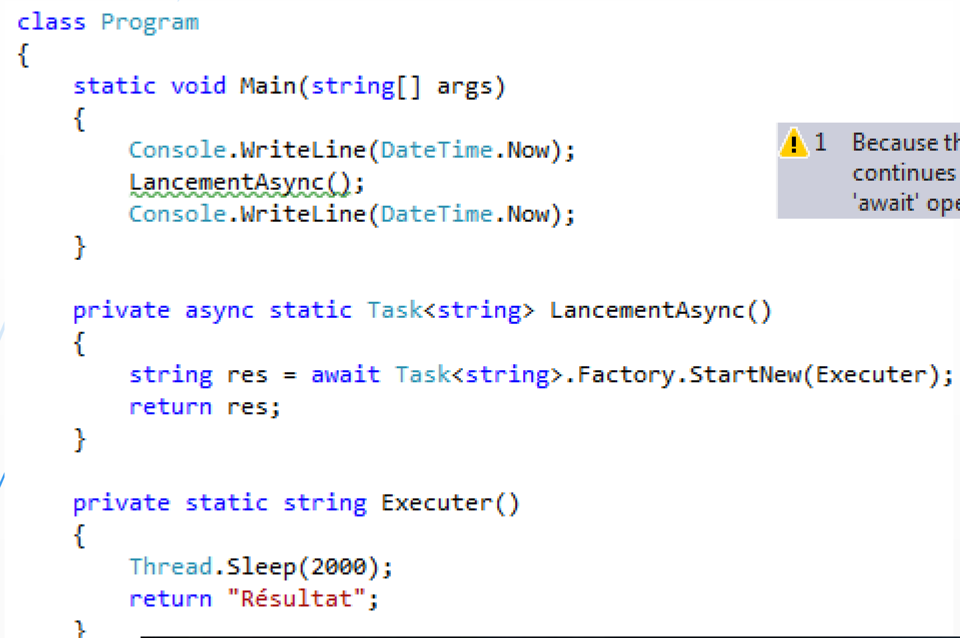
- Asynchronous Method in C#5 C#6

```
private async void Download_Click(object sender, EventArgs e)
 {
                var client = new WebClient();
                string data = await  client.DownloadStringTaskAsync("http://...");
                Data.Text = data;
}
```

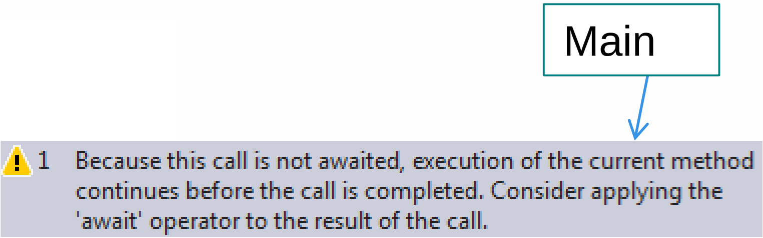
Introduction

- To synchronize asynchronous processes
  - An abstract mechanism called semaphore is used
- For the developer
  - A method is asynchronous if it is launched in parallel to the execution of the program
  - The program continues to run while waiting for the answer of the asynchronous method

Basic Example

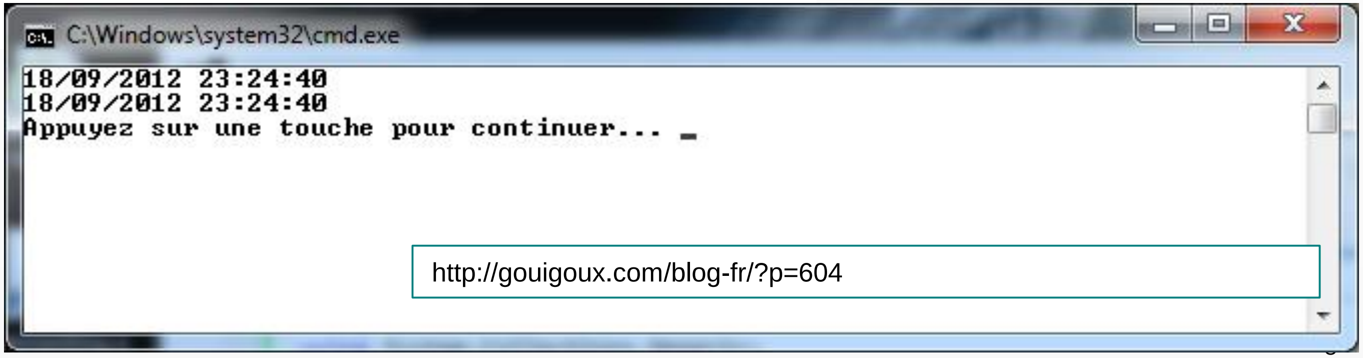


```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(DateTime.Now);
        LancementAsync();
        Console.WriteLine(DateTime.Now);
    }

    private async static Task<string> LancementAsync()
    {
        string res = await Task<string>.Factory.StartNew(Executer);
        return res;
    }

    private static string Executer()
    {
        Thread.Sleep(2000);
        return "Résultat";
    }
}
```



Main

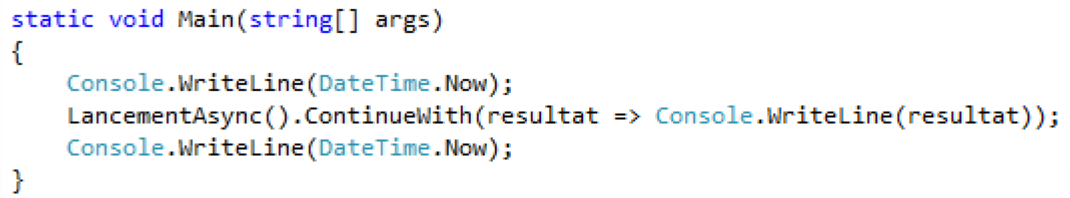
⚠ 1 Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the 'await' operator to the result of the call.



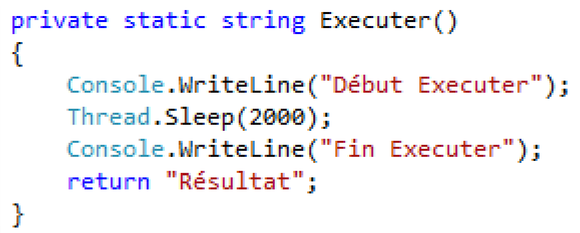
```
18/09/2012 23:24:40
18/09/2012 23:24:40
Appuyez sur une touche pour continuer... _
```

http://gouigoux.com/blog-fr/?p=604

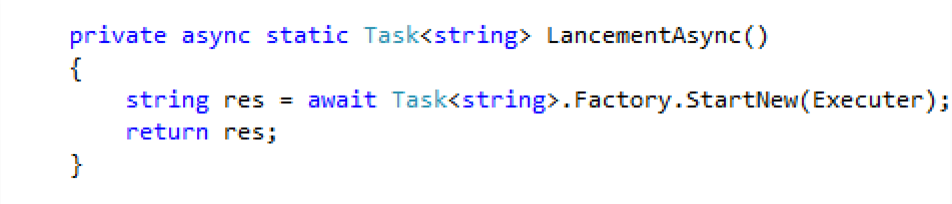
Basic Example



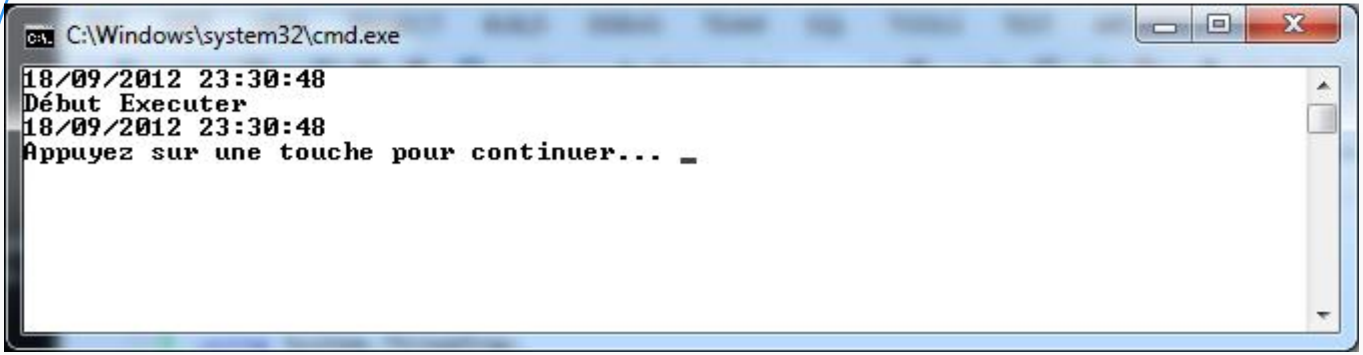
```
static void Main(string[] args)
{
    Console.WriteLine(DateTime.Now);
    LancementAsync().ContinueWith(resultat => Console.WriteLine(resultat));
    Console.WriteLine(DateTime.Now);
}
```



```
private static string Executer()
{
    Console.WriteLine("Début Executer");
    Thread.Sleep(2000);
    Console.WriteLine("Fin Executer");
    return "Résultat";
}
```

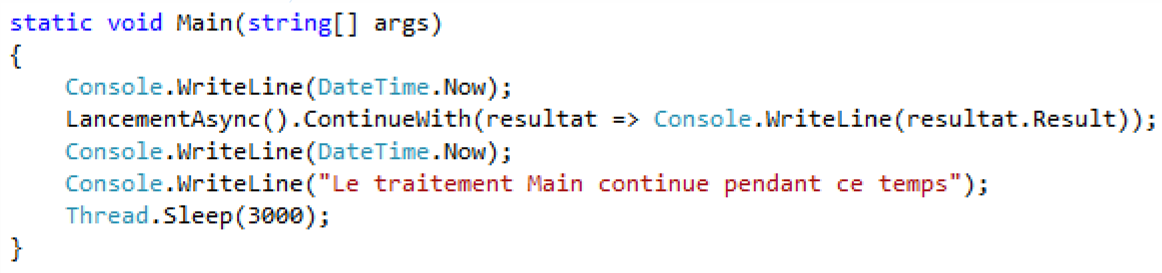


```
private async static Task<string> LancementAsync()
{
    string res = await Task<string>.Factory.StartNew(Executer);
    return res;
}
```

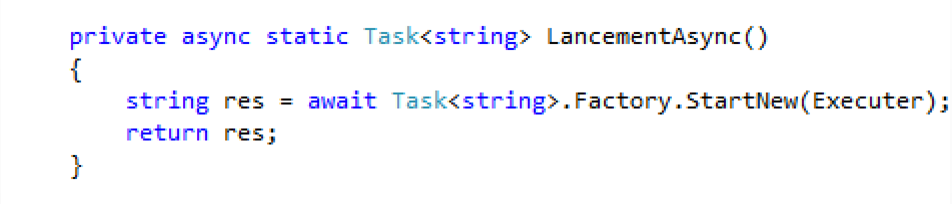


```
18/09/2012 23:30:48
Début Executer
18/09/2012 23:30:48
Appuyez sur une touche pour continuer... _
```

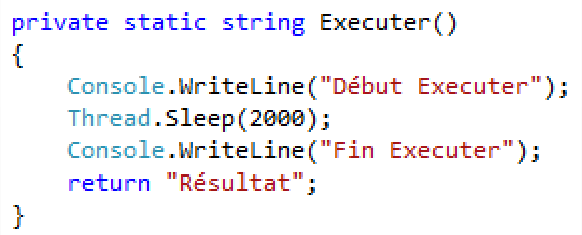
Basic Example



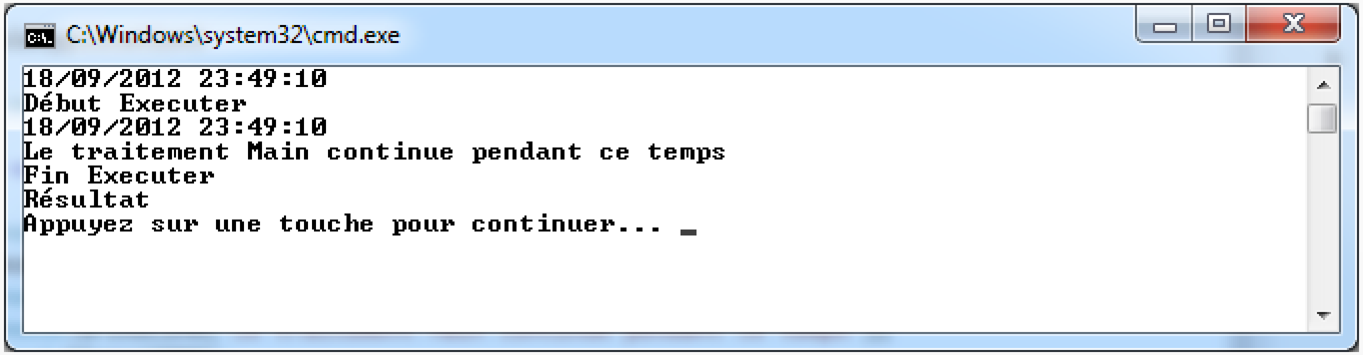
```
static void Main(string[] args)
{
    Console.WriteLine(DateTime.Now);
    LancementAsync().ContinueWith(resultat => Console.WriteLine(resultat.Result));
    Console.WriteLine(DateTime.Now);
    Console.WriteLine("Le traitement Main continue pendant ce temps");
    Thread.Sleep(3000);
}
```



```
    private async static Task<string> LancementAsync()
    {
        string res = await Task<string>.Factory.StartNew(Executer);
        return res;
    }
```



```
private static string Executer()
{
    Console.WriteLine("Début Executer");
    Thread.Sleep(2000);
    Console.WriteLine("Fin Executer");
    return "Résultat";
}
```



```
18/09/2012 23:49:10
Début Executer
18/09/2012 23:49:10
Le traitement Main continue pendant ce temps
Fin Executer
Résultat
Appuyez sur une touche pour continuer... _
```

Task-based Asynchronous Pattern

- For any operation that could potentially take more than 50ms to be completed
  - Task-based Asynchronous Pattern (TAP)
    - TAP uses a single method to represent the initiation and the completion of an asynchronous operation
  - Other patterns
    - Event-based Async Pattern
    - IAsyncResult Async Pattern
    - See http://msdn.microsoft.com/en-us/library/ms734701.aspx

Task-based Asynchronous Pattern

- Recommended asynchronous design pattern for new development
- Many new APIs in Windows Phone 8 will offer only a Task-based API
- Words
  - **await / async**
  - **Task**
    - Class
    - Represents an unity of asynchronous execution which can be synchronized with other tasks
    - A task can have a value of return or not
      - Generic definition Task<TResult > or void (better : Task)

Task-based Asynchronous Pattern

- Example

```
private async void SearchMoviesAsync(int year)
{
    …
    lstTitles.Items.Clear();
    while (true)
    {
        var movies = await  SearchMoviesBatchAsync(catalog, year, count, pageSize);
        if (movies.Length == 0)
            break;
        foreach (var title in movies)
        {
            lstTitles.Items.Add(title.Name);
        }
        count += movies.Length;
    }
}
 private async Task<Title[]> SearchMoviesBatchAsync(NetflixCatalog catalog, int year, int count, int pageSize)
{
    var query = from title in catalog.Titles where title.ReleaseYear == year orderby title.Name select title;
    return await  query.Skip(count).Take(pageSize).ToArrayAsync();
}
```

Task-based Asynchronous Pattern

- When SearchMoviesAsync is called,
  - It begins normally … until await
  - Two scenarios
    - The call to SearchMoviesBatchAsync ends in a synchronous way, in which case the execution continues normally
    - Or it runs in a asynchronous way
      - The control is returned to the method which calls SearchMoviesAsync (for example, btnSearch_Click)
      - When the call to SearchMoviesBatchAsync ends, the execution of SearchMoviesAsync continues

Task-based Asynchronous Pattern

- Syntax
  - Async suffix after the operation name
    - Example : Get**Async** for a get operation
  - If the name already exists, use TaskAsync
    - Example : Get**TaskAsync**

Task-based Asynchronous Pattern

- All async methods must return
  - void, Task or Task<TResult>
  - Task<TResult>
    - For any function that returns a value
    - TResult is the type of the return value
  - void / Task
    - Not a good practice
    - There is only a case which justifies this structure

(see http: // msdn.microsoft.com/en-us/magazine/jj991 977.aspx)

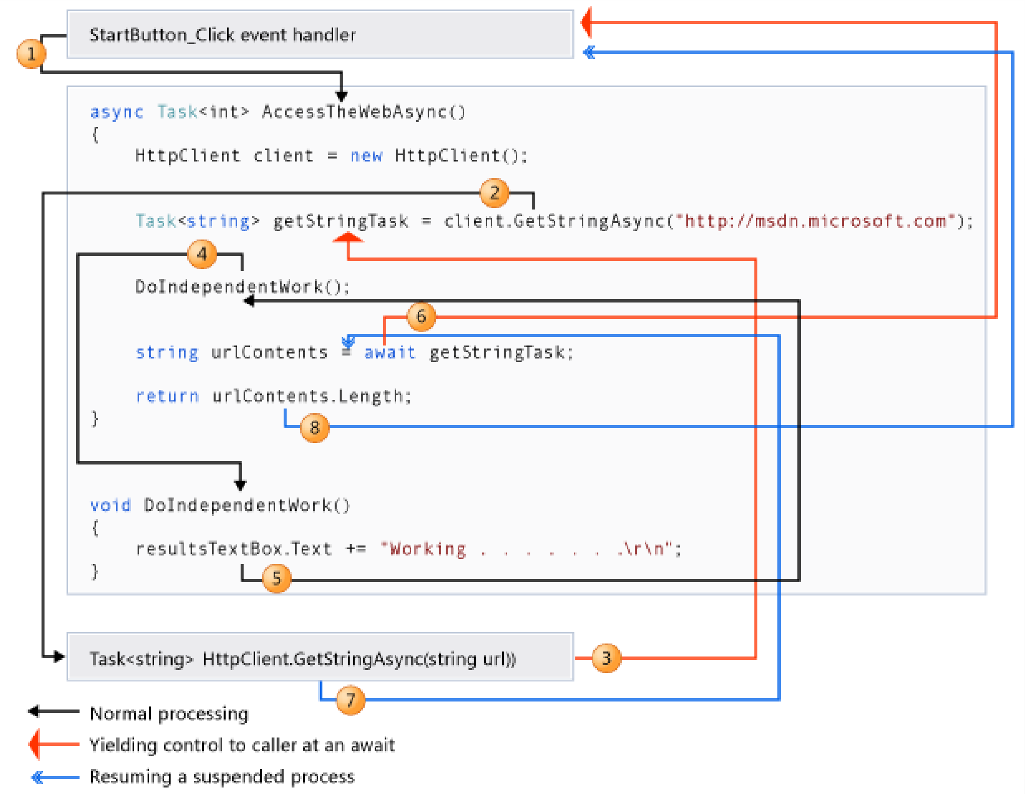
Task-based Asynchronous Pattern

- Error Handlings
  - Same as with synchronous code
  - Exceptions thrown while code is executing asynchronously are surfaced back on the calling thread
  - If an async method is called without the keyword await, if an exception is raised in this method, it will remain silent
    - The await keyboard (or .Wait()) is necessary to raise the errors

Task-based Asynchronous Pattern

- Remark
  - If the async method is called without await
    - The called method is still executed on background thread
    - The calling thread does not wait for the result
    - Only feasible for methods that return void or Task

Example


StartButton_Click event handler
1
2
4
6
8
5
Task<string> HttpClient.GetStringAsync(string url))
3
7
Normal processing
Yielding control to caller at an await
Resuming a suspended process

```
async Task<int> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();

    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    DoIndependentWork();

    string urlContents = await getStringTask;

    return urlContents.Length;
}

void DoIndependentWork()
{
    resultsTextBox.Text += "Working . . . . . . .\r\n";
}
```