C# : Task

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

1. Introduction
2. `Task:async/await`
   - Sans méthode
   - Avec méthode
3. `Task<T>`
4. `WhenAll`
   - Sans résultat
   - Avec résultat

Plan

5. `WhenAny`
6. Parallélisme
   - `Run`
   - `Wait`
7. `FromResult`
8. Annulation avec `CancellationToken`

Plan

9 Progression et reporting

10 Gestion des exceptions

11 Cycle de vie d'une Task

12 `ValueTask<T>`

13 Bonnes pratiques (synthèse)

14 Exercices

C#

Pourquoi l'asynchronisme ?

- Accès réseau, disque, base de données, API...
- Opérations lentes
- Exécution synchrone $\Rightarrow$ blocage

C#

Pourquoi l'asynchronisme ?

- Accès réseau, disque, base de données, API...
- Opérations lentes
- Exécution synchrone $\Rightarrow$ blocage

Objectif

- Ne pas bloquer l'**UI**
- Libérer les threads serveur

C#

Pourquoi les Tasks ?

- **Réactivité :** Ne pas bloquer l'interface utilisateur (UI).
- **Performance :** Utiliser efficacement les processeurs multi-cœurs.
- **Abstraction :** Plus simple et moins coûteux que de gérer des `Threads` manuellement.

C#

Pourquoi les Tasks ?

- **Réactivité :** Ne pas bloquer l'interface utilisateur (UI).
- **Performance :** Utiliser efficacement les processeurs multi-cœurs.
- **Abstraction :** Plus simple et moins coûteux que de gérer des `Threads` manuellement.

Définition

- `Task` : représente une opération asynchrone **qui se termine plus tard** (équivalent des `Promise` en **JavaScript**).
- `Task<T>` : opération asynchrone qui **retourne une valeur**.
- `async/await` : syntaxe qui rend l'asynchrone **lisible** et **composable**.

C#

Ne pas confondre **Thread** et **Task**.

C#

Le Thread

- La plus petite unité d'exécution qu'un système d'exploitation peut planifier et exécuter.
- Chaque thread possède sa propre pile d'appels (stack), mais partage la mémoire avec les autres threads du même programme
- Créer un thread manuellement est une opération "coûteuse" en termes de ressources (mémoire et temps processeur).

C#

Le Thread

- La plus petite unité d'exécution qu'un système d'exploitation peut planifier et exécuter.
- Chaque thread possède sa propre pile d'appels (stack), mais partage la mémoire avec les autres threads du même programme
- Créer un thread manuellement est une opération "coûteuse" en termes de ressources (mémoire et temps processeur).

Le ThreadPool (Bassin de threads)

- Une collection de threads gérés par l'environnement d'exécution (le **CLR** en **.NET**).
- Au lieu de créer et détruire un thread pour chaque petite tâche, le programme pioche un thread déjà existant dans un "bassin" (pool), l'utilise, puis le remet à disposition une fois le travail fini.

C#

Qui les gère ?

- L'**OS** gère les threads physiques et décide quel thread a accès au processeur (ordonnancement).
- Le **CLR** gère le **ThreadPool**. Il ajuste dynamiquement le nombre de threads disponibles selon la charge de travail pour optimiser les performances

C#

Qui les gère ?

- L'**OS** gère les threads physiques et décide quel thread a accès au processeur (ordonnancement).
- Le **CLR** gère le **ThreadPool**. Il ajuste dynamiquement le nombre de threads disponibles selon la charge de travail pour optimiser les performances

Pourquoi utiliser des **Tasks** plutôt que des **Threads** ?

- Une **Task** ne signifie pas forcément un nouveau **thread**.
- Elle est gérée par le **Task Scheduler**, qui décide si la tâche doit être exécutée sur le **ThreadPool** (via `Task.Run` pour du calcul intensif) ou si elle peut simplement "attendre" sans consommer de thread (via await pour de l'I/O réseau ou disque).

C#

Thread vs Task

Caractéristique	Thread (Bas niveau)	Task (Haut niveau)
Modèle	1 thread = 1 ressource OS.	Modèle de promesse.
Ressources	Coûteux (mémoire, stack).	Léger, optimisé par le CLR.
Gestion	Manuelle (Start, Join, Abort).	Automatisée (async/await).
Retour	Pas de retour de valeur direct.	Retourne `Task<T>` facilement.
Usage	Rarement utilisé directement.	Standard moderne en .NET.

C#

Thread vs Task

Caractéristique	Thread (Bas niveau)	Task (Haut niveau)
Modèle	1 thread = 1 ressource OS.	Modèle de promesse.
Ressources	Coûteux (mémoire, stack).	Léger, optimisé par le CLR.
Gestion	Manuelle (Start, Join, Abort).	Automatisée (async/await).
Retour	Pas de retour de valeur direct.	Retourne `Task<T>` facilement.
Usage	Rarement utilisé directement.	Standard moderne en .NET.

async $\neq$ nouveau thread

C#

Exemple 1 : `Task.Delay` (le plus simple)

```
namespace ProjetTask
{
    class Program
    {
        static async Task Main()
        {
            Console.WriteLine("Début");

            // libère le thread pendant l'attente de 2 secondes
            await Task.Delay(2000);

            Console.WriteLine("Fin");
        }
    }
}
```

C#

Explications

- `await` ne bloque pas le thread, il suspend l'exécution de la méthode et rend la main au système.
- Signature de la méthode
  - `async` obligatoire pour les méthodes asynchrones.
  - `Task` (pas `void`)
- La méthode reprend après la fin de la tâche

C#

Exemple 2 : utiliser une méthode

```
namespace ProjetTask
{
    class Program
    {
        static async Task MethodeAsync()
        {
            Console.WriteLine("Début");
            await Task.Delay(2000);
            Console.WriteLine("Fin");
        }

        static async Task Main()
        {
            await MethodeAsync();
        }
    }
}
```

C#

Explications

- L'asynchronisme se propage
- `async` remonte dans la pile d'appels

Exemple 3 : `Task<T>` (retour de valeur)

```
namespace ProjetTask
{
    class Program
    {
        static async Task<int> CalculerAsync()
        {
            Console.WriteLine("Début");
            await Task.Delay(2000);
            return 42;
        }

        static async Task Main()
        {
            int resultat = await CalculerAsync();
            Console.WriteLine(resultat);
        }
    }
}
```

C#

Explications

- `Task<int>` : promesse d'un `int` futur.
- `await` récupère automatiquement la valeur une fois disponible.

C#

Exemple : `Task.WhenAll` (paralléliser l'attente)

```
namespace ProjetTask
{
    class Program
    {
        static async Task Main()
        {
            Console.WriteLine("Début");
            Task t1 = Task.Delay(3000);
            Task t2 = Task.Delay(5000);
            Task t3 = Task.Delay(2000);

            await Task.WhenAll(t1, t2, t3);
            Console.WriteLine("Tout est terminé");
        }
    }
}
```

C#

Explications

- Le programme précédent lance plusieurs tâches, puis attend **toutes**.
- `t1` démarre : un compte à rebours de 3 secondes commence.
- `t2` démarre immédiatement après : un compte à rebours de 5 secondes commence (pendant que t1 tourne encore).
- `t3` démarre immédiatement après : un compte à rebours de 2 secondes commence (pendant que t1 et t2 tournent encore).
- `await Task.WhenAll(t1, t2, t3)` : le programme fait une pause ici et attend que les trois chronomètres soient arrivés à zéro.

C#

Pour résumer

- Les tâches démarrent en parallèle
- Temps total = durée de la plus longue tâche

C#

Si l'exécution était séquentielle (l'une après l'autre), le programme prendrait `3 + 5 + 2 = 10` **secondes.**

```
await Task.Delay(3000); // Attend 3s
await Task.Delay(5000); // Puis attend 5s
await Task.Delay(2000); // Puis attend 2s
// total = 10 secondes
```

C#

Exemple : `WhenAll` avec résultats

```
namespace ProjetTask
{
    class Program
    {
        static async Task<int> CalculerValeur(int id, int delai)
        {
            await Task.Delay(delai);
            return id * 10;
        }
        static async Task Main()
        {

            Task<int> t1 = CalculerValeur(1, 3000);
            Task<int> t2 = CalculerValeur(2, 5000);
            Task<int> t3 = CalculerValeur(3, 2000);

            await Task.WhenAll(t1, t2, t3);

            int resultat1 = t1.Result;
            int resultat2 = t2.Result;
            int resultat3 = t3.Result;

            Console.WriteLine($"Résultats : {resultat1}, {resultat2}, {resultat3}");
        }
    }
}
```

C#

Explications

- Le programme précédent commence par lancer les tâches en parallèle.
- `t1` retourne 10 après 3 secondes.
- `t2` retourne 20 après 5 secondes.
- `t3` retourne 30 après 2 secondes.
- `await Task.WhenAll(t1, t2, t3)` : le programme fait une pause ici et attend que les trois chronomètres soient arrivés à zéro.
- Il récupères les résultats.
- Le programme aura duré 5 secondes au total.

C#

Schématisation

```
t1: |------3s------|
t2: |----------5s----------|
t3: |----2s----|
```

C#

`WhenAll` **retourne directement un** `T[]`

```
namespace ProjetTask
{
    class Program
    {
        static async Task<int> CalculerValeur(int id, int delai)
        {
            await Task.Delay(delai);
            return id * 10;
        }
        static async Task Main()
        {
            int[] res = await Task.WhenAll(
                CalculerValeur(1, 3000),
                CalculerValeur(2, 5000),
                CalculerValeur(3, 2000)
            );
            Console.WriteLine(string.Join(", ", res));
        }
    }
}
```

C#

Exemple : `WhenAny` (première tâche terminée)

```
namespace ProjetTask
{
    class Program
    {
        static async Task<string> RequeteAsync(string nom, int ms)
        {
            await Task.Delay(ms);
            return $"Réponse {nom}";
        }

        static async Task Main()
        {
            Console.WriteLine("Début");
            Task<string> a = RequeteAsync("A", 6000);
            Task<string> b = RequeteAsync("B", 2000);

            Task<string> first = await Task.WhenAny(a, b);
            Console.WriteLine(first.Result);
    }
}
```

C#

Remarques

- `WhenAny` retourne la première tâche terminée
- Les autres continuent à s'exécuter

C#

Remarques

- `async/await` sert à ne pas bloquer pendant l'attente.
- `Task.Run` sert à déplacer du calcul sur un autre thread (parallélisme).

C#

Remarques

- `async/await` sert à ne pas bloquer pendant l'attente.
- `Task.Run` sert à déplacer du calcul sur un autre thread (parallélisme).

Pour simplifier

- `async/await` ⇒ I/O-bound
- `Task.Run` ⇒ CPU-bound.

C#

Remarques

- `async/await` sert à ne pas bloquer pendant l'attente.
- `Task.Run` sert à déplacer du calcul sur un autre thread (parallélisme).

Pour simplifier

- `async/await` ⇒ I/O-bound
- `Task.Run` ⇒ CPU-bound.

Ne pas utiliser Task.Run pour des appels réseau

C#

Exemple

```
static async Task Main()
{
    Console.WriteLine("Début");
    Task t1 = Task.Run(() =>
    {
        Console.WriteLine("Tâche en cours d'exécution");
    });
    Console.WriteLine("Fin");

}
```

C#

Exemple

```
static async Task Main()
{
    Console.WriteLine("Début");
    Task t1 = Task.Run(() =>
    {
        Console.WriteLine("Tâche en cours d'exécution");
    });
    Console.WriteLine("Fin");

}
```

Résultat : le thread principal continue sans attendre la fin de la tâche.

```
Début
Fin
Tâche en cours d'exécution
```

C#

Exemple

```
static async Task Main()
{
    Console.WriteLine("Début");
    Task t1 = Task.Run(() =>
    {
        Console.WriteLine("Tâche en cours d'exécution");
    });
    Console.WriteLine("Fin");

}
```

Résultat : le thread principal continue sans attendre la fin de la tâche.

```
Début
Fin
Tâche en cours d'exécution
```

Ou

```
Début
Fin
```

C#

Pour attendre la fin de la tâche avant de poursuivre

```
static async Task Main()
{
    Console.WriteLine("Début");
    Task t1 = Task.Run(() =>
    {
        Console.WriteLine("Tâche en cours d'exécution");
    });
    t1.Wait();
    Console.WriteLine("Fin");
}
```

C#

Pour attendre la fin de la tâche avant de poursuivre

```
static async Task Main()
{
    Console.WriteLine("Début");
    Task t1 = Task.Run(() =>
    {
        Console.WriteLine("Tâche en cours d'exécution");
    });
    t1.Wait();
    Console.WriteLine("Fin");
}
```


Résultat

```
Début
Tâche en cours d'exécution
Fin
```

Remarque

`Wait` est bloquante et peut provoquer un deadlock.

Remarque

`Wait` est bloquante et peut provoquer un deadlock.

Deadlock

- Une situation où deux (ou plusieurs) entités s'attendent mutuellement de façon circulaire, si bien que plus aucune ne peut progresser.
- Chacun attend que l'autre libère une ressource, qui ne sera jamais libérée.

Remarque

`Wait` est bloquante et peut provoquer un deadlock.

Deadlock

- Une situation où deux (ou plusieurs) entités s'attendent mutuellement de façon circulaire, si bien que plus aucune ne peut progresser.
- Chacun attend que l'autre libère une ressource, qui ne sera jamais libérée.

Contexte typique

- Application UI (**WPF** / **WinForms**)
- **ASP.NET** classique (pas **Core**)

C#

Solution : utiliser `await`

```
static async Task Main()
{
    Console.WriteLine("Début");
    Task t1 = Task.Run(() =>
    {
        Console.WriteLine("Tâche en cours d'exécution");
    });
    await t1;
    Console.WriteLine("Fin");
}
```

C#

Solution : utiliser `await`

```
static async Task Main()
{
    Console.WriteLine("Début");
    Task t1 = Task.Run(() =>
    {
        Console.WriteLine("Tâche en cours d'exécution");
    });
    await t1;
    Console.WriteLine("Fin");
}
```


Résultat

```
Début
Tâche en cours d'exécution
Fin
```

C#

Problème : rigidité des interfaces

- On a une interface asynchrone avec une méthode retournant `Task<T>`.
- L'implémentation possède **déjà** la donnée (Cache, calcul simple, Mock).
- **Dilemme :** Comment retourner une valeur immédiate sans casser la signature `async` ?

C#

Problème : rigidité des interfaces

- On a une interface asynchrone avec une méthode retournant `Task<T>`.
- L'implémentation possède **déjà** la donnée (Cache, calcul simple, Mock).
- **Dilemme :** Comment retourner une valeur immédiate sans casser la signature `async` ?

Exemple

```
public interface ICalculService
{
    Task<int> CalculerAsync(int x);
}
```

C#

Solution : `Task.FromResult`

`Task.FromResult<T>` crée une tâche qui est **déjà terminée** avec le résultat fourni.

- **Retourner une valeur** au format `Task<T>` sans création de thread supplémentaire **ni** d'asynchronisme réel
- **Adaptation de signature** (compatibilité API) : pas une mise en async.
- Exécution purement synchrone.

C#

Exemple

```
public class CalculService : ICalculService
{
    public Task<int> CalculerAsync(int x)
    {
        return Task.FromResult(x * x);
    }
}
```

C#

Exemple

```
public class CalculService : ICalculService
{
    public Task<int> CalculerAsync(int x)
    {
        return Task.FromResult(x * x);
    }
}
```


Explication

- Pas besoin de `async/await` si le résultat est instantané.
- **Moins de surcoût** qu'une machine d'état `async`.

C#

FromResult vs async vs Task.Run

Approche	Coût	Thread	Quand l'utiliser ?
`Task.FromResult`	Faible	Aucun	Valeur immédiate / cache / mocks
`async Task<T>`	Moyen	Aucun	I/O async réel (`...Async`)
`Task.Run`	Élevé	Thread pool	CPU-bound (calcul lourd)

Pourquoi annuler ?

- L'utilisateur change d'avis (UI), timeout métier, arrêt serveur.
- Bonne pratique : proposer un `CancellationToken` sur les méthodes longues.

C#

Exemple : annulation

```
namespace ProjetTask
{
    class Program
    {
        static async Task TravailAnnulableAsync(CancellationToken ct)
        {
            for (int i = 1; i <= 10; i++)
            {
                ct.ThrowIfCancellationRequested();
                await Task.Delay(300, ct);
                Console.WriteLine($"étape {i}");
            }
        }
        static async Task Main()
        {
            using var cts = new CancellationTokenSource();
            cts.CancelAfter(1000);

            try
            {
                await TravailAnnulableAsync(cts.Token);
            }
            catch (OperationCanceledException)
            {
                Console.WriteLine("Annulé");
            }
        }
    }
}
```

C#

Reporter la progression

- Les tâches longues peuvent notifier un `IProgress<T>` (UI friendly).
- `Progress<T>` poste souvent sur le contexte de synchronisation (UI).

C#

Exemple : `IProgress<int>`

```
namespace ProjetTask
{
    class Program
    {
        static async Task TravailAvecProgressAsync(IProgress<int> progress)
        {
            for (int p = 0; p <= 100; p += 20)
            {
                await Task.Delay(200);
                progress?.Report(p);
            }
        }

        static async Task Main()
        {
            var progress = new Progress<int>(p => Console.WriteLine($"{p}%"));
            await TravailAvecProgressAsync(progress);
        }
    }
}
```

C#

Gestion des exceptions avec `async/await`

Contrairement aux Threads classiques, l'exception est capturée naturellement par le `await`

- L'exception est encapsulée dans la `Task`.
- Elle est relancée lors de l'appel à `await`, permettant l'usage d'un bloc `try-catch` standard.

`await` déballe l'exception de la Task

```
namespace ProjetTask
{
    class Program
    {
        static async Task MethodeRisqueeAsync()
        {
            Console.WriteLine("Avant exécution");
            throw new InvalidOperationException();
        }

        static async Task Main()
        {
            try
            {
                await MethodeRisqueeAsync();
            }
            catch (InvalidOperationException ex)
            {
                Console.WriteLine($"Exception capturée : {ex.Message}");
            }
        }
    }
}
```

`await` déballe l'exception de la Task

```
namespace ProjetTask
{
    class Program
    {
        static async Task MethodeRisqueeAsync()
        {
            Console.WriteLine("Avant exécution");
            throw new InvalidOperationException();
        }

        static async Task Main()
        {
            try
            {
                await MethodeRisqueeAsync();
            }
            catch (InvalidOperationException ex)
            {
                Console.WriteLine($"Exception capturée : {ex.Message}");
            }
        }
    }
}
```

Résultat

```
Avant exécution
Exception capturée : Operation is not valid due to the current state of the object.
```

C#

Cycle de vie d'une Task

Une `Task` représente une opération asynchrone qui évolue à travers différents états.

- **Created** / **WaitingForActivation** : Task instanciée mais n'a pas encore débuté.
- **Running** : l'opération est en cours d'exécution.
- **Final States** :
  - **RanToCompletion** : Succès total.
  - **Faulted** : Échec via une exception non gérée.
  - **Canceled** : Annulation via `CancellationToken`.

Optimisation : `ValueTask<T>`

Pour les scénarios haute performance où le résultat est souvent déjà disponible.

- **Problème** : `Task<T>` est un objet de classe (alloué sur le tas/heap), ce qui crée une pression sur le Garbage Collector (GC).
- **Solution** : `ValueTask<T>` est une structure (allouée sur la pile/stack).

C#

Bonnes pratiques essentielles

- Préférer les API **asynchrones natives** : `ReadAsync`, `GetAsync`, etc.
- Éviter `.Result` et `.Wait()` $\Rightarrow$ risques de blocage/deadlock.
  - `await` : libère le thread pendant l'attente (I/O) $\rightarrow$ le thread peut faire autre chose.
  - `.Wait()` / `.Result` : bloque le thread jusqu'à la fin $\rightarrow$ on revient à du synchrone.
- Propager l'asynchronisme : `async` « remonte » dans la pile d'appels.
- N'utiliser `Task.Run` que pour du **CPU-bound**.
- Signature : `Task`/`Task<T>` (éviter `async void` sauf événements).

C#

Pourquoi `await` est recommandé ?

- Ne bloque pas de thread $\Rightarrow$ meilleure réactivité / scalabilité
- Évite les deadlocks en contexte **UI** / **ASP.NET** classique
- S'intègre proprement avec `CancellationToken` et `IProgress`

C#

Recommandation selon le contexte

Contexte	**Bonne pratique**
Console	async / await
UI (WPF/WinForms)	await obligatoire
ASP.NET Core	await partout
Bibliothèque	await + ConfigureAwait(false)

C#

Piège : `async void`

```
async void ChargerAsync() { ... }
```

C#

Piège : `async void`

```
async void ChargerAsync() { ... }
```

Règle d'or

```
async Task ChargerAsync() { ... }
```

C#

Piège : `async void`

```
async void ChargerAsync() { ... }
```

Règle d'or

```
async Task ChargerAsync() { ... }
```


Pourquoi ?

- Exceptions non capturables
- Rupture de la chaîne `async`

C#

Piège : `Task.Run` mal utilisé

```
await Task.Run(() => httpClient.GetAsync(url));
```

C#

Piège : `Task.Run` mal utilisé

```
await Task.Run(() => httpClient.GetAsync(url));
```

Règle d'or

```
await httpClient.GetAsync(url);
```

C#

Piège : `Task.Run` mal utilisé

```
await Task.Run(() => httpClient.GetAsync(url));
```

Règle d'or

```
await httpClient.GetAsync(url);
```

Pourquoi ?

- `Task.Run` pour les calculs, pas pour les **I/O**
- Gaspillage des threads

Piège : le code suivant ne capture jamais l'exception

```
namespace ProjetTask
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                TraiterAsync();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }

        static async Task TraiterAsync()
        {
            await Task.Delay(1000);
            throw new Exception("Erreur !");
        }
    }
}
```

Il faut le remplacer par

```
namespace ProjetTask
{
    class Program
    {
        static async Task Main(string[] args)
        {
            try
            {
                await TraiterAsync();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }

        static async Task TraiterAsync()
        {
            await Task.Delay(1000);
            throw new Exception("Erreur !");
        }
    }
}
```

C#

Exercice 1 (bases)

1. Écrire `AttendreEtAfficherAsync(int ms)` qui fait un `Task.Delay(ms)` puis affiche `OK`.
2. Écrire `CalculerCarreAsync(int x)` qui attend 300ms puis retourne x^2.

C#

Exercice 2 (composition)

1. Lancer 3 appels `CalculerCarreAsync` et afficher les résultats avec `WhenAll`.
2. Écrire `PremierResultatAsync()` qui lance deux tâches et affiche la première terminée via `WhenAny`.

C#

Correction

```
static async Task<int> CalculerCarreAsync(int x)
{
    // simulation d'un calcul
    await Task.Delay(300);
    return x * x;
}

static async Task Exercice2_Composition()
{
    // lancer les taches en parallèle (elles partent sur le ThreadPool)
    Task<int> t1 = CalculerCarreAsync(5);
    Task<int> t2 = CalculerCarreAsync(10);
    Task<int> t3 = CalculerCarreAsync(8);

    // attendre que toutes soient terminees
    int[] resultats = await Task.WhenAll(t1, t2, t3);

    // affichage
    Console.WriteLine($"Resultats : {string.Join(", ", resultats)}");
}
```

C#

Exercice 3 (annulation et erreurs)

1. Ajouter un `CancellationToken` à un traitement en boucle et annuler après 1 seconde.
2. Simuler une exception dans une tâche, la capturer proprement autour du `await`.

Correction

```
static async Task Exercice3_ExceptionsEtAnnulation()
{
    using var cts = new CancellationTokenSource();
    cts.CancelAfter(1000); // annulation automatique après 1s

    try
    {
        await TraitementRisqueAsync(cts.Token);
    }
    catch (OperationCanceledException)
    {
        // capture l'annulation
        Console.WriteLine("Le traitement a depasse le delai imparti.");
    }
    catch (Exception ex)
    {
        // Capture une erreur simulée
        Console.WriteLine($"Une erreur est survenue : {ex.Message}");
    }
}

static async Task TraitementRisqueAsync(CancellationToken ct)
{
    for (int i = 0; i < 10; i++)
    {
        ct.ThrowIfCancellationRequested(); // vérifie l'annulation
        await Task.Delay(300, ct); // l'attente est aussi annulable
        if (i == 2) throw new Exception("échec critique du calcul !"); // simulation erreur

        Console.WriteLine($"Etape {i} terminée");
    }
}
```