La programmation asynchrone

Il est de plus en plus nécessaire de développer des applications réactives et capables d'effectuer plusieurs tâches simultanément. Le framework .NET a apporté des solutions à ce problème depuis ses débuts avec les classes `Thread` ou `BackgroundWorker`, entre autres. Avec le framework .NET 4.0 sont arrivées la classe `Task` et sa contrepartie générique `Task<TResult>`. Ces types ont simplifié le travail du développeur en permettant de gérer simplement le lancement ou l'attente de l'exécution de blocs de code, mais en fournissant aussi le moyen d'exécuter plusieurs traitements asynchrones à la suite.

L'arrivée de C# 5 a encore simplifié la tâche du développeur avec l'intégration de l'asynchronisme directement dans le langage. En effet, les mots-clés `async` et `await` permettent d'écrire du code asynchrone de manière séquentielle, comme du code... synchrone !

1. Les objets Task

Les classes `Task` et `Task<TResult>` permettent l'exécution de code asynchrone en encapsulant l'utilisation de threads.

Fonctionnement des threads

Les threads sont des unités d'exécution qui peuvent travailler, selon l'architecture de la machine, en parallèle ou en pseudo-parallèle (chaque thread s'exécute pendant une petite durée, puis cède sa place à un autre thread, qui cédera de nouveau sa place à un autre thread, etc.). C'est le système d'exploitation qui décide du temps processeur alloué à chaque thread. Pour cette raison, il est impossible de prévoir exactement à quel moment sera exécutée une instruction placée dans un thread.

Création d'un objet Task

Avant toute utilisation, les objets `Task` doivent être instanciés. Pour cela, huit constructeurs sont à disposition, et ils ont tous comme premier paramètre un délégué (de type `Action` ou `Func`). Ce délégué correspond à la portion de code qui doit être exécutée dans un thread séparé. Le traitement est ensuite lancé à l'aide de la méthode `Start` de l'objet `Task`.

```
static void Main(string[] args)
{
    Console.WriteLine("Ce code est executé dans le thread
principal dont l'identifiant est {0}",
Thread.CurrentThread.ManagedThreadId);

    Action actionAExecuter = () => { Console.WriteLine("Ce code
est executé dans un thread séparé dont l'identifiant est {0}",
Thread.CurrentThread.ManagedThreadId); };

    Task task = new Task(actionAExecuter);
    Task task2 = new Task(actionAExecuter);

    task.Start();
    task2.Start();

    Console.ReadLine();
}
```

Le même code est ici exécuté deux fois, dans deux threads distincts. Le résultat de cette portion de code est le suivant :

```
Ce code est executé dans le thread principal dont l'identifiant
est 10
Ce code est executé dans un thread séparé dont l'identifiant
est 11
Ce code est executé dans un thread séparé dont l'identifiant
est 12
```

Lors d'une nouvelle exécution, le résultat ne sera pas forcément exactement le même. En effet, deux nouveaux threads seront créés et la valeur de la propriété `Thread.CurrentThread.ManagedThreadId` relative à chacun de ces threads sera probablement différente.

L'utilisation de la classe `Task<TResult>` permet pour sa part de récupérer un résultat de traitement fortement typé. Le code suivant additionne les nombres entiers de 1 à 10000 dans un thread séparé et renvoie le résultat. Celui-ci est affiché à l'aide de la propriété `Result` de la classe `Task <TResult>`.

```
static void Main(string[] args)
{
    Traitement();
    Console.ReadLine();
}

private static void Traitement()
{
    Task<long> tacheCalcul = new Task<long>(() =>
        {
            long result = 0;
            for (long i = 1; i <= 10000; i++)
                result += i;

            return result;
        });

    tacheCalcul.Start();
    Console.WriteLine(tacheCalcul.Result);
}
```

L'utilisation de cette propriété attend de manière synchrone la fin de l'exécution du traitement pour afficher son résultat. Afin de garder la notion d'asynchronisme, il est possible d'utiliser la méthode `ContinueWith`, qui crée une nouvelle tâche en lui donnant comme paramètre d'entrée la tâche précédente.

```
private static void Traitement()
{
    Task<long> tacheCalcul = new Task<long>(() =>
        {
            long result = 0;
            for (long i = 1; i <= 10000; i++)
```

```
                result += i;

            return result;
        });

    tacheCalcul.ContinueWith(tachePrecedente =>
Console.WriteLine(tachePrecedente.Result));

    tacheCalcul.Start();
}
```

2. Écrire du code asynchrone avec async et await

Le mot-clé `await` permet d'attendre la fin de l'exécution d'une `Task` démarrée. Pendant cette attente, l'application rend la main aux threads nécessitant du temps processeur, notamment pour maintenir l'interface graphique réactive. Le code placé à la suite d'une instruction utilisant ce mot-clé est exécuté après la fin de l'exécution du traitement asynchrone, comme s'il était placé dans une méthode `ContinueWith`.

La tâche devant être attendue peut être de type `Task` ou `Task<T>`. Ce dernier type doit être utilisé lorsque le code exécuté par la tâche renvoie une valeur. Si la valeur est de type `int`, alors la tâche doit être de type `Task<int>`.

```
//À la fin de l'exécution, la variable resultat contient 1
int resultat = await Task<int>.Run(() => return 1);
```

Toute méthode contenant une instruction utilisant `await` doit utiliser le mot-clé `async` dans sa déclaration. Un appel à une méthode marquée avec ce mot-clé peut potentiellement être effectué à l'aide du mot-clé `await`, mais pour cela, il faut modifier la signature de cette méthode asynchrone de manière à ce qu'elle renvoie un objet `Task` ou un `Task<T>`.

Le code suivant est une réécriture du code vu précédemment. La méthode `Traitement` ne renvoie ici pas d'objet `Task` car elle est appelée à partir de la méthode `Main` de l'application, or il est strictement interdit d'utiliser un mot-clé `async` ou `await` dans cette méthode.

```
private static async void Traitement()
{
    long resultat = await Task<long>.Run(() =>
        {
            long result = 0;
            for (long i = 1; i <= 10000; i++)
                result += i;

            return result;
        });

    Console.WriteLine(resultat);
}
```