Notes sur le module : *C# Avancé.*

Armel Pitelet

1er février 2022

Table des matières

1 Code Synchrone et Asynchrone **1**

2 Les Events **2**

1 Code Synchrone et Asynchrone

Remarque : *Sur les fenêtre wpf.*
Il y a un processus qui tourne sur le Thread UI qui s'occupe de gérer la partie visible d'une fenêtre.

Remarque : *Pour faire de l'asynchrone.*
On peut passer par la classe Thread (depreciated) qui est la pour des raisons historique pour déclarer une action à faire dans un nouveaux Thread.

Mot clef : **async**.
Authorise l'utilisation du **await**. Voir https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/concepts/async/.

Mot clef : **Task**.
Une task représente une action à effectuer de manière asynchrone (en générale mais pas obligatoirement). On ne peut pas mettre à jour un autre Thread que celui en cours dans une Task. (voir remarque sur le `Dispacher`).

Mot clef : **await**.
Tant que la ligne marquée await n'a pas finie son éxcécution, on attent avant de passer à la suite de la fonction.

Note :
On peut afficher les Threads en cours dans visual studio.

Remarque : *Sur la classe `Dispacher`.*
Elle permet de faire discuter les Threads entre eux via `Dispacher.Invoke(une lambda qui fait un truc)`.

Note :
Si on a le choix, c'est plus performant d'utiliser de l'asynchrone que du synchrone.

Remarque : *Pour faire des fonctions asynchrone.*

Il faut utiliser le mot clef `async` et renvoyer void ou bien renvoyer une `Task<UnType>`. Il vaut mieux renvoyer une Task car elle à des propriétés que l'on peut vouloir récupérer (comme la Task est elle toujours en cours d'exécution ou pas). Le `return` de la fonction n'a cependant pas à renvoyer lui même une `Task` (à partir du moment ou on a un `await` quelque part dans la fonction).

Remarque : *Sur `Task.FromResult()`.*

Permet de créer une `Task` à partir de ce que l'on met dans le `Result(un_truc)`. Lorsque l'évènement est levé, la méthode abonner est appelée. On se désabonne ensuite via un $Event- = UnTruc$.

Remarque : *Sur la notion d'Asynchrone.*

Faire de l'asynchrone avec `async await` revient à écrire du code de manière synchrone (le code après un await est excécuté aprés la fin de la Task marqué en await) mais ce qui est éxcécuté en asynchrone est exécutée sur un Thread à part qui ne bloque pas l'exécution des autres Threads en cours.

Note :

On peut rendre une tache synchrone en utilisant `Task.GetAwaiter().GetResult()`. C'est complètement inutile sauf si on ne peut absolument pas renvoyer une `Task` (comme sur du code Legacy par exemple)

Remarque : *Sur la gestion de plusieurs Task en même temps.*

Task.WhenAll(task1,task2), WaitAll(task1,task2) permettent d'attendre que les Task données soient terminées. `WhenAny` permet d'éxcécuter ce qui suit une fois qu'au moins une des Task est terminée

Remarque : *Divers trucs sur les Tasks en générale.*

`Task.FromResult(résultat)` permet de renvoyer un résultat (synchrone ou non). Est équivalent à passer par le constructeur de Task (`new Task()`). `Task.FromException(new Exception())` permet de créer une Task à partir d'une exception.

Remarque : *Sur la gestion des exceptions dans les Task.*

Les exceptions (pasque c'est bien fait) peuvent remonter dans le Thread principale. A partir du moment ou la Task est asynchrone et lève une exception tt se passe (pour nous) comme si on était en synchrone, il n'y a pas de précautions particulières à prendre.

Remarque : *Sur le type d'exception qui remonte des Task.*

Ces Exceptions sont typé de la même manière que les classiques.

Remarque : *Sur l'`AggregateException`.*

Type particulier d'exception qui peut popper dans certain contexts (ex : Parallel) à la place de l'excpetion que l'on attend. C'est une exception qui aggrège (wrap) un ensemble d'exceptions et qui contient d'autres exceptions. Ce type d'Exception arrive quand on est dans d'autres Thread que le principale. N'arrive pas si l'on fait bien les async await mais peut arriver facilement dans les systèmes multi-threadés et/ou parallélisés.

2 Les Events

Remarque : *Sur les Events.*

Dans visual Studio un symbol d'éclaire à coté d'un nom de variable imndique un évènement. Un évènement se déclare avec le mot clef `event` puis un type délégué Nom de l'event. Lorsque l'on

fait un `Evenement += UnTruc`. On abonne la méthode `UnTruc` à l'évènement `Evenement`. La desinscription est obligatoire à un moment lors du programme. Attention, si l'on passe des fonction anonymes, il n'existe aucun moyen de se désabonner !

Remarque : *Sur les `Loaded` et `Unloaded` dans WPF.*
Il n'est pas nécessaire de se désabonner à ces évènement la puisqu'ilms sont détruits en même temps que la fenêtre.

Note :
En Angular, le système d'abonnement est identique. Cas particulier, les requêtes Http ne nécessite pas de désabonnement.

Note :
En générale on passe plus par du `async await` que du `EventHandler` pour faire de l'asynchrone. Mais c'est encore très présent en Wpf.

Remarque : *Sur le passage d'argument aux Events.*
Il est nécessaire de créer des classes qui hérite de `EventArgs` pour passer des arguments aux évènements.

Remarque : *Sur le `InitializeComponent` en WPF.*
Cette commande vient en fait transformer le xaml en une vrai classe .cs (d'ou le `partial` devant les deux types de classes)

Note :
Les EventQQChose sont en faite des classes de **délégué**
Note :
Les `Event` sont des propriétés sans Getter Setter !

Remarque : *Sur les méthodes qui s'abonne aux event.*
Il vaut mieux toujours vérifier que l'event existe (c'est à dire qu'il ait au moins un abonnement $\rightarrow$ d'où les `if(MonEvent != null)`).

3 Divers