Utilisation des exceptions

Le déclenchement d'une exception permet d'interrompre le flux normal d'exécution de l'application tout en fournissant des informations sur l'erreur qu'elle représente. Sans intervention du développeur pour la gérer, elle se propage automatiquement en remontant la pile des appels jusqu'au point d'entrée de l'application où elle peut causer sa fermeture complète.

1. Création et déclenchement d'exceptions

Les exceptions sont représentées par des objets héritant de la classe `System.Exception`. Elles peuvent être générées (levées) à tout endroit de l'application.

a. La classe Exception

La classe `System.Exception` est la classe de base de toutes les exceptions. Ses membres lui permettent de contenir de nombreuses informations relatives à l'erreur en cours de propagation. La plupart de ces membres sont valorisés automatiquement lorsqu'une exception est levée. Parmi ceux-ci, on trouve notamment la propriété `StackTrace` qui contient des informations particulièrement intéressantes pour localiser la source de l'exception.

L'instanciation d'un objet de type `Exception` peut être faite à l'aide de trois constructeurs :

```
public Exception();
public Exception(string message);
public Exception(string message, Exception innerException);
```

Le paramètre `message` correspond à un libellé personnalisé permettant de comprendre l'erreur lorsqu'elle est traitée loin de sa source. On peut par la suite retrouver ce libellé en consultant la propriété `Message` de l'exception.

Le paramètre `innerException` permet quant à lui d'encapsuler une exception que l'on traite dans une nouvelle exception que l'on génère.

b. Le mot-clé throw

Instancier un objet de type `Exception` ne suffit pas à lever l'exception. Pour ce faire, il est nécessaire d'exécuter une instruction `throw` en fournissant l'objet `Exception` à lever.

```
public void CompterJusqua(int nombre)
{
    if (nombre < 0)
    {
       string message = string.Format("Impossible de compter
de 0 à {0}", nombre);
       Exception erreur = new Exception(message);
       throw erreur;
    }

    //Ce code ne sera exécuté que si aucune exception n'a été levée
    for (int i = 0; i <nombre; i++)
    {
```

```
        Console.WriteLine(i);
    }
}
```

Si la valeur du paramètre `nombre` est inférieure à 0, une exception est levée, interrompant le flux d'exécution normal du programme. Elle pourra être gérée dans la méthode appelant la procédure `CompterJusqua`, ou plus loin dans la pile des appels.

c. Exceptions spécialisées

Dans certains cas, il peut être intéressant de lever des exceptions spécifiques à un type de comportement. Ceci peut permettre d'identifier très rapidement des erreurs simples à corriger mais peu évidentes à identifier, de localiser la zone de l'application à partir de laquelle l'exception a été levée, ou encore de fournir des informations particulières pour traiter l'erreur.

Le framework .NET fournit un certain nombre de types d'exceptions adaptés à des circonstances particulières. Le tableau ci-après liste quelques-uns de ces types.

`NotImplementedException`

Lancée par défaut dans tout squelette de code généré par Visual Studio. Signifie qu'une portion de code n'a pas encore été implémentée.

`StackOverflowException`

Lancée par le CLR lorsque le nombre d'éléments dans la pile d'appels devient trop important. Se produit typiquement lors d'un appel récursif.

`SqlException`

Lancée par les classes d'accès aux bases de données SQL Server, notamment lorsqu'une requête SQL est incorrecte.

`DivideByZeroException`

Lancée lorsqu'un calcul impliquant une division par zéro est effectué.

`OutOfMemoryException`

Lancée lorsque l'application a besoin d'un espace mémoire que le système n'est pas en mesure d'allouer.

2. Gérer les exceptions

Le déclenchement d'une exception pouvant causer la fin de l'exécution d'une application, il est nécessaire pour le développeur de savoir les traiter convenablement.

En effet, il est souvent souhaitable que l'exécution d'une application puisse se poursuivre en cas d'erreur, en dégradant éventuellement les fonctionnalités si les circonstances l'exigent.

a. La structure try ... catch

Le seul moyen d'intercepter une exception en C# est d'utiliser la structure `try ... catch`. Cette structure est constituée de deux parties :

- Un bloc `try` qui permet de définir un jeu d'instructions "à risque".
- Un ou plusieurs blocs `catch` permettant de spécifier le code à exécuter en cas d'exception.

La syntaxe générale d'utilisation de cette structure est la suivante :

```
try
{
    //Jeu d'instructions à risque
}
catch (<type d'exception> <nom de variable>)
{
    //Instructions permettant de traiter les exceptions
}
[ autre(s) bloc(s) catch ... ]
```

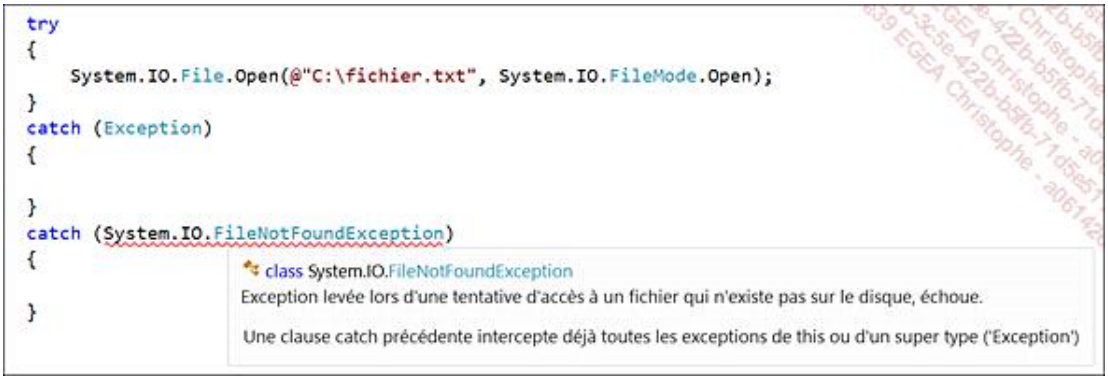
Lorsqu'une des instructions placées dans le bloc `try {}` déclenche une exception, le système examine chacun des blocs `catch {}` et exécute le code contenu dans le **premier bloc catch** dont le type d'exception correspond à l'erreur levée.

Dans ce cas, une fois l'erreur traitée dans le bloc `catch {}`, la méthode continue son exécution à partir de la ligne suivant la fin de l'instruction `try ... catch`.

L'appel de méthodes asynchrones (à l'aide du mot-clé `async`) n'est possible dans les blocs `catch` qu'à partir de la version 6 de C#.

Lorsque plusieurs types d'exceptions doivent être gérés, il est important de réfléchir à l'ordre dans lequel les blocs sont placés. En effet, si l'on a deux blocs gérant les types `Exception` et `FileNotFoundException` placés dans cet ordre, le bloc exécuté dans le cas d'une `FileNotFoundException` sera le bloc… gérant le type `Exception` !

Visual Studio est d'ailleurs tout à fait capable de nous prévenir de ce problème :



Ceci s'explique aisément par le fait que le type `FileNotFoundException` hérite du type `Exception`. L'exception levée est donc bien du type `Exception` et est donc gérée par le premier bloc.

Pour éviter ce problème, il faut placer les blocs correspondant aux types les plus spécialisés d'abord, et finir par

ceux des types les moins spécialisés.

```
try
{
    System.IO.File.Open(@"C:\fichier.txt", System.IO.FileMode.Open);
}
catch (System.IO.FileNotFoundException)
{

}
catch (Exception)
{

}
```

b. Les filtres d'exception

Dans certains cas, l'exécution d'un traitement relatif à une exception ne doit être déclenchée que si certaines conditions sont rencontrées. Pour cela, la solution la plus évidente est l'insertion de tests dans un bloc `catch`.

```
try
{
    //Génération d'un nombre aléatoire
    var random = new Random();
    int nombre = random.Next();

    Console.WriteLine("Au hasard : {0}", nombre);

    //Si ce nombre est pair
    if (nombre % 2 == 0)
    {
        throw new Exception("Le nombre est pair");
    }
    //Sinon
    else
    {
        throw new Exception("Le nombre est impair");
    }
}
catch (Exception ex)
{
    if (ex.Message.Contains(" pair"))
        Console.WriteLine("Traitement de l'exception : le nombre est pair");
    else
        Console.WriteLine("Traitement de l'exception : le nombre n'est pas pair");
}
```

L'exemple présenté ici ne comporte que deux cas très simples de traitements, mais ils peuvent parfois être nombreux. Les erreurs COM se présentent notamment sous la forme d'un seul type d'exception : `COMException`. C'est la propriété `HResult` de l'objet `COMException` qui porte l'information indiquant le type d'erreur qui s'est produit. Il est dans ce cas très facile de se retrouver avec un bloc `catch` très long, et, par conséquent, difficilement maintenable.

Les filtres d'exception offrent la possibilité de cloisonner le traitement de chacune de ces exceptions dans un bloc `catch` à l'aide du mot-clé `when` suivi d'un prédicat renvoyant une valeur booléenne.

L'exemple précédent peut être réécrit pour tirer parti de cette nouveauté de C# 6 de la manière suivante :

```
try
{
    //Génération d'un nombre aléatoire
    var random = new Random();
    int nombre = random.Next();

    Console.WriteLine("Au hasard : {0}", nombre);

    //Si ce nombre est pair
    if (nombre % 2 == 0)
    {
        throw new Exception("Le nombre est pair");
    }
    //Sinon
    else
    {
        throw new Exception("Le nombre est impair");
    }
}
catch (Exception ex) when (ex.Message.Contains(" pair"))
{
    Console.WriteLine("Traitement de l'exception : le nombre est pair");
}
catch (Exception ex) when (ex.Message.Contains("impair"))
{
    Console.WriteLine("Traitement de l'exception : le nombre n'est pas pair");
}
```

Il est bien évidemment possible de combiner l'utilisation de blocs `catch` "simples" et de blocs utilisant des filtres d'exception pour couvrir un maximum d'erreurs d'exécution. Il faut néanmoins prendre en compte le fait qu'un seul bloc `catch` peut être exécuté, qu'un filtre lui soit associé ou non.

c. Le bloc finally

Certains traitements doivent être effectués après l'exécution correcte d'une portion de code, mais aussi lorsqu'une exception est levée. Ces traitements peuvent correspondre notamment à une libération de ressource, à une notification de l'utilisateur ou à un enregistrement de données.

Pour cela, l'utilisation d'une structure `try ... catch` peut s'avérer fastidieuse puisqu'une duplication de code est nécessaire : au minimum une instruction identique à la fin des blocs `try` et `catch`. Cette situation n'étant pas optimale, l'utilisation d'un bloc `finally` est préconisée afin de simplifier l'écriture, la lecture et la maintenance du code.

Ce type de bloc est ajouté à la suite d'une structure `try ... catch` de la manière suivante :

```
try
{
    //...
}
catch (...)
{
    //...
```

```
}
finally
{
    //Traitements à effectuer systématiquement
}
```

Il est également possible d'ajouter un bloc `finally` après un bloc `try` seul. En effet, dans certains cas, il est pertinent de ne pas vouloir intercepter une exception mais de vouloir tout de même exécuter un jeu d'instructions.

La structure `using` fait notamment usage de cette technique : lors de la compilation, elle est transformée en une structure `try ... finally`. Le bloc `finally` contient alors un appel à la méthode `Dispose` de la variable utilisée dans le bloc `using` initial.

Considérons la portion de code suivante :

```
using (FileStream stream = new FileStream(@"C:\monfichier.txt",
FileMode.Open))
{
    //Traitements
}
```

Après compilation, l'assembly généré contient un code intermédiaire équivalent au code suivant :

```
FileStream stream;
try
{
    stream = new FileStream(@"C:\monfichier.txt", FileMode.Open);

    //Traitements
}
finally
{
    if (stream != null)
        stream.Dispose();
}
```

Ainsi, même en cas d'erreur d'exécution dans les traitements, le fichier ouvert est libéré.