Ce qu'il faut retenir sur la gestion des erreurs

En C#, lorsqu'il y a **un bug 🐞 dans notre programme** on appelle ça une **exception**. ⚠ Problème, quand cela se produit l'erreur est "levée" et la pile d'appels est renvoyée à l'utilisateur. De plus, le programme est désormais planté et il faudra le redémarrer pouvoir continuer à l'utiliser.

Cela présente plusieurs problèmes ⚠ :

1. SI il y a des exceptions, c'est que notre programme **n'est pas résilient**. Il faut trouver un moyen de parer à toutes les éventualités.
2. Notre programme est inutilisable. Il va falloir le relancer.
3. On risque **d'afficher des informations techniques** à l'utilisateur : la stacktrace. Mettez vous à la place de l'utilisateur final. **Il va sans doute préférer avoir un jolie message d'erreur** qu'un charabia technique incompréhensible 👀.

Qu'est ce que la "Stacktrace" ?

Une **pile d'appel**, (ou pile de trace, trace d'appels, etc) ça ressemble à ça :

```
Unhandled exception. System.FormatException: Input string was not in a correct format.
at System.Number.ThrowOverflowOrFormatException(ParsingStatus status, TypeCode type)
at System.Number.ParseInt32(ReadOnlySpan`1 value, NumberStyles styles, NumberFormatInfo info)
at System.Int32.Parse(String s)
at MyApp.Calculator.Compute(String[] input) in C:\Users\Sylvain\source\repos\MyApp\Calculator.cs:line 47
at MyApp.SuperCalculator.Compute(String[] input) in C:\Users\Sylvain\source\repos\MyApp\Calculator.cs:line 13
at MyApp.Program.Main(String[] args) in C:\Users\Sylvain\source\repos\MyApp\Program.cs:line 30
```

C'est la liste (la pile plus exactement) de toutes les instructions exécutées par le programme qui ont mené à l'erreur. **On doit donc la lire de haut en bas de l'instruction la plus récente à la plus ancienne.**

La première ligne étant l'erreur levée par le programme en question.

En général, lorsqu'on met une application en production (donc pas en test) comme un site internet codé en asp .Net Core (le framework web C# de Microsoft) il y a **des configurations par défaut qui empêchent de montrer la pile d'appels aux utilisateurs finaux. Très pratique !**

Comment empêcher de divulguer des erreurs aux utilisateurs finaux ?

On va utiliser les blocs try et catch comme dans l'exemple ci dessous :

```
try {
// Mon code.
} catch {
// Si mon code plante,
// ce code va être exécuté.
}
```

Si je mets un bloc try, je suis obligé de mettre un bloc catch. **Si une erreur se produit entre mes accolades try alors le code à l'intérieur du bloc catch va être exécuté.**

Evidemment, je ne suis pas obligé de mettre une seule ligne de code dans le bloc try. Je peux encapsuler 1000 lignes si ça me chante. Mais gare à la lisibilité ! 👀

De même que si une erreur se produit dans les méthodes appelées (et les méthodes des méthodes) le code à l'intérieur du bloc catch s'exécutera.

Si je le souhaite, je peux **mettre des parenthèses à l'instruction catch pour y passer un objet de type Exception** (ou comme nous allons le rappeler plus bas, les classes dérivées de la classe exceptions).

```
try {
// Mon code.
} catch (Exception myError) {
// Si mon code plante,
// ce code va être exécuté.
// et je peux accéder à l'erreur
// grace à ma variable myError
}
```

L'objet Exception sera **préremplie d'informations sur l'erreur** et elles seront accessibles grâce à la variable myError dans cet exemple (comme un paramètre de fonction).

Toutes les exceptions héritent directement ou indirectement de la classe Exception. Donc, grâce au polymorphisme, nous pouvons "catcher" des erreurs plus précises comme par exemple la IndexOutOfRangeException.

⚠Mais attention⚠ : si je mets seulement un catch avec une IndexOutOfRangeException et que j'ai une erreur d'un autre type, l'erreur ne sera pas attrapée.

Si **je choisis de catcher des erreurs bien précises,** j**e dois tout de même prévoir des cas plus généralistes**. Je peux, pour cela, ajouter plusieurs blocs catch.

```
try {
// Mon code.
} catch (IndexOutOfRangeException myError) {
// L'erreur IndexOutOfRangeException
// sera capturée à ce niveau.
}
catch (Exception myError) {
// Toutes les autres erreurs seront capturées ici
// parce que la classe Exception
// représentent l'erreur la plus "générale"
}
```

L'ordre des blocs est important ! Le code s'exécutant de haut en bas, je dois toujours **mettre en premier (vers le haut) les erreurs les plus spécifiques puis les erreurs les plus génériques (vers le bas).** Sinon certaines erreurs ne pourront jamais être "catchée" parce qu'une erreur plus vaste les aura capturé auparavant. **En pratique, le code ne compilera même pas !**

Et finally dans tout ça ?

Je peux ajouter un bloc finally. **Quelle que soit l'erreur déclenchée le bloc finally s'exécutera.**

```
try {
// Mon code.
}
catch (Exception myError) {
// une erreur
}
finally {
// Cette instruction s'exécutera quelle que soit l'erreur.
}
```

Et la commande throw ?

De manière simple, la commande throw déclenche une erreur. On dit aussi qu'elle permet de lever une exception. C'est à dire qu'elle affichera par exemple la stracktrace si l'erreur n'est pas attrapée.

Ce qui sous entend que je peux déclencher manuellement une erreur si je le souhaite.

```
static void MyAwesomeFunction(){
  throw new NotImplementedException();
}
```

Dans cet exemple, **j'ai voulu écrire une fonction mais je n'ai pas encore eu le temps de la coder.** Si un ou une collègue de travail essaie d'utiliser ma fonction, il y aura une erreur indiquant que la fonction n'est pas encore implémentée.

J'ai donc "levé" manuellement une erreur. Pour cela, j'ai écris un throw avec un nouvel objet de type exception, comme ici la "NotImplementedException()" **sans oublier auparavant d'utiliser le mot clef new car il s'agit d'un objet.**

Même dans un bloc catch, je peux aussi lever une exception avec la commande throw (si je ne mets rien après le throw, **c'est l'erreur courante qui sera relancée**). Cela peut avoir un intérêt si je veux quand même remonter l'erreur.

```
try {
// Mon code.
}
catch (Exception myError) {
// une erreur
throw; // je souhaite tout de même lever l'erreur
}
```

Et la commande when ?

Je peux aussi récupérer une erreur de manière conditionnelle en utilisant l'instruction when. Evidemment, la condition (dans les parenthèses du when) doit dépendre de l'objet exception capturée auparavant.

```
try {
// Mon code.
```

```
}
catch (Exception myError) when( myError = ... ){
// une erreur conditionnelle
}
```

Et en milieu professionnel ?

Voici quelques recommandations sur la gestion des erreurs en milieu professionnel :

- En général, o**n n'englobe pas la totalité du code** par des try et des catch.
- **On encapsule plutôt les centres névralgiques** des programmes (appels à des API, appels à des BDD, exécution de batch, ou d'opération d'écriture sur disque, etc).
- C'est une mauvaise pratique de faire un try et un catch **dans le but de masquer une erreur.** Un peu comme si vous laissez la poussière sous le tapis. Vous pouvez faire ça si vous "**loggez**" l'erreur dans une base de donnée par exemple afin qu'on puisse savoir qu'il y a une erreur à un endroit précis.
- **Il est rare d'utiliser plus d'un catch.** Ou alors c'est pour répondre à un besoin bien spécifique.
- **La commande when est rarement utilisée.** (Et je ne savais même pas qu'elle existait avant de réaliser ce cours... 😳).