



Cégep de Saint-Hyacinthe  
Département d'informatique

Programmation orientée objet

420-2DP-HY

**(3-3-3)**

## **Initiation aux exceptions #1**

(Version 1.2)

# **2 heures**

Préparé par

**Martin Lalancette**

Comprendre les éléments suivants:

- Les exceptions
- Instruction *try*
- Instruction *catch*
- Instruction *finally*
- *throw*

## Table des matières

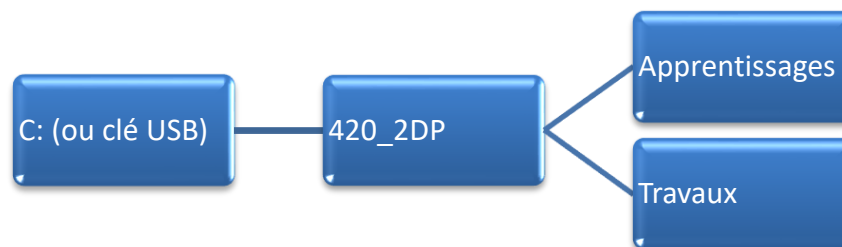
Introduction.....	3
Préparation de base .....	3
Les exceptions .....	3
Pourquoi les gérer? .....	4
Hiérarchie des exceptions .....	4
L'instruction <i>try</i> .....	5
L'instruction <i>catch</i> .....	5
L'instruction <i>finally</i> .....	7
<i>throw</i> .....	9
Bibliographie.....	12

## Introduction

Cette séquence a pour but de vous initier aux notions de base nécessaires à l'introduction de la programmation. Nous commencerons par énoncer les éléments théoriques appuyés d'exemples simples et faciles à reproduire. Afin d'axer l'attention sur la compréhension de ces notions, il y aura des exercices à faire tout au long de cette séquence. **La séquence portera essentiellement sur les exceptions.**

## Préparation de base

Pour bien suivre les instructions qui vont être mentionnées tout au long des séquences d'apprentissage, une préparation de base s'impose. Il est important de créer un répertoire de travail (sur votre C: ou clé USB). Voici une suggestion d'arborescence:



**Exercice 1. :** S'assurer d'avoir créé l'arborescence ici haut mentionnée sur votre P ou votre clé USB.

## Les exceptions

Le langage C# (comme en C++) peut gérer les situations inattendues, anormales ou exceptionnelles qui surviennent pendant l'exécution d'un programme. Nous les appelons « [exceptions](#) ». Les exceptions peuvent être générées par :

- le common language runtime (CLR),
- des bibliothèques tierces

- dans le code d'application à l'aide du mot clé **throw**. Donc vous pouvez vous-même établir des circonstances spéciales dans la définition de vos classes.

### Pourquoi les gérer?

Si aucun gestionnaire d'exceptions pour une exception donnée n'est présent, le programme cesse de s'exécuter (parfois anormalement, car il y a perte de contrôle) avec un message d'erreur. La compréhension de cette notion est nécessaire, car beaucoup de classes de .NET lancent des exceptions. Exemple :

```
int iDividende = 5;  
int iDiviseur = 0;  
float fResultat = 0;  
  
fResultat = iDividende / iDiviseur; // Division par zéro possible???
```

**Exercice 2. :** Créer la solution portant le nom de ce document et un projet de type Console (MVVM) portant le nom de cet exercice. Ajouter les lignes de code de l'exemple précédent dans le Main. Attendre les instructions du professeur.

Les exceptions **non interceptées** sont traitées par un **gestionnaire d'exceptions générique** fourni par le système.

### Hiérarchie des exceptions

La classe [Exception](#) est la classe **de base des exceptions**. Plusieurs classes d'exceptions héritent directement d'elle, parmi lesquelles **ApplicationException** et **SystemException**. Ces deux classes forment la base de la quasi-totalité des exceptions runtime. Les objets **Exception** contiennent des informations détaillées à propos de l'erreur, y compris l'état de la pile des appels et une description du texte de l'erreur.

Voici comment gérer ces exceptions afin que notre programme se comporte correctement. Tout d'abord, il existe trois mots clés à comprendre : **try**, **catch** et **finally**.

## L'instruction *try*

Il faut utiliser cette instruction afin d'indiquer au compilateur quelles sont les lignes de code à surveiller, c.-à-d. celles susceptibles de générer des exceptions en les entourant d'accollades ({ et }). **Exemple** : Division par zéro.

```
try
{
    int iDividende = 5;
    int iDiviseur = 0;
    float fResultat = 0;

    fResultat = iDividende / iDiviseur; // Division par zéro possible???
}
...
```

## L'instruction *catch*

Cette instruction sert à définir la ou les actions à faire lorsqu'une exception survient sur une des lignes définies dans le **try**. Une fois qu'une exception est levée, elle se propage jusqu'en haut de pile des appels, jusqu'à ce qu'une instruction catch soit trouvée pour l'exception. À noter : **les lignes de code suivant celle qui génère l'exception ne seront pas exécutées**. C'est normal, car l'exécution sera déplacée dans le bloc d'instructions sous le **catch**. Cette instruction a besoin d'un objet pour recevoir le détail de l'exception. En C#, le mot clé **catch** est utilisé pour définir un gestionnaire d'exceptions. Cette instruction peut jouer un rôle de filtre en fonction des types d'exceptions définies.

**Exemple** : La division par zéro (suite).

```
try
{
    int iDividende = 0;
    int iDiviseur = 0;
    float fResultat = 0;

    iDividende = int.Parse(Console.ReadLine());
    iDiviseur = int.Parse(Console.ReadLine());

    fResultat = iDividende / iDiviseur; // Division par zéro possible???
}
catch (Exception e)
{
    Console.WriteLine("Erreur: " + e.Message);
}
```

Dans cet exemple, le *catch* permet d'intercepter toutes les possibilités d'exceptions pouvant survenir lors de l'exécution du programme.

**Exercice 3. :** Modifier le code afin d'ajouter les fonctionnalités du *try...catch*.

**Exercice 4. :** Créer une classe **Exercices** et copier les lignes de code du bouton précédent dans une méthode nommée **Diviser**(int iDividende, int iDiviseur). Trouver une façon de retirer le **Console.WriteLine** mais d'informer le parent. Poser les questions à l'utilisateur dans le Main pour alimenter les paramètres d'appel de la méthode.

Cependant, nous ne pouvons que définir les mêmes actions à faire. C'est pourquoi, qu'il est de bonnes pratiques (« Best Practice ») d'intercepter de façon distinguée ces exceptions. Plusieurs instructions *catch* peuvent être définies avec le *try*. Cela permet de filtrer le type d'exception pour effectuer des instructions plus précises. Tout d'abord, il faut comprendre l'ordre dans lequel les *catch* seront appelés. La première instruction *catch* qui peut gérer l'exception est exécutée ; toutes les instructions *catch* suivantes, même compatibles, sont ignorées. Exemple :

```
try
{
    int iDividende = 0;
    int iDiviseur = 0;
    float fResultat = 0;

    iDividende = int.Parse(Console.ReadLine());
    iDiviseur = int.Parse(Console.ReadLine());

    fResultat = iDividende / iDiviseur; // Division par zéro possible???
}
catch (DivideByZeroException)
{
    Console.WriteLine("Division par zéro!!!");
}
catch (Exception e)
{
    Console.WriteLine("Erreur: " + e.Message);
}
```

Ne sera pas appelée, car le 1er catch l'a intercepté

Priorité

## L'instruction *finally*

Le code dans un bloc **finally** est exécuté même si une exception est levée, permettant ainsi à un programme de **libérer des ressources** ou d'effectuer d'autres opérations. L'instruction **finally** sert à **garantir l'exécution d'un bloc d'instructions** de code quelle que soit la méthode de sortie du bloc try précédent. À noter : **ce bloc est toujours exécuté qu'il y ait ou non une exception de générée. Attention, si une exception cause la fermeture immédiate de l'application, le code se trouvant dans le *finally* ne sera pas nécessairement exécuté.** Cela peut dépendre de la configuration du poste.

### Exemple : Ouverture de fichiers

```
StreamReader fichierEntrant = null;
StreamWriter fichierSortant = null;
try
{
    fichierEntrant = new StreamReader("test.txt");
    fichierSortant = new StreamWriter("X:\\sortie.txt"); // Lecteur X n'existe pas

    // ...
}
catch (IOException e)
{
    Console.WriteLine("Fichier exception: " + e.Message);
}
finally
{
    // Permet de fermer des états et de libérer des ressources déjà ouvertes.
    fichierEntrant.Close();
    fichierEntrant = null;
}
```

En résumé, vous pouvez effectuer les combinaisons suivantes :

<p><b>Combinaison #1</b> : Situation assez courante.</p> <pre>try {     // Code à surveiller pour     // les exceptions possibles. } catch (Exception e) {     // Code pour gérer l'exception     // qui est levée. }</pre>	<p><b>Combinaison #2</b> : Les trois ...</p> <pre>try {     // Code à surveiller pour     // les exceptions possibles. } catch (Exception e) {     // Code pour gérer l'exception     // qui est levée. } finally {     // Code à exécuter exception ou pas. }</pre>
---	--

**Combinaison #3 : Les trois ... avec plusieurs *catch*...**

```
try
{
    // Code à surveiller pour les exceptions possibles.
}
catch (DivideByZeroException)
{
    // Code pour gérer l'exception qui est levée.
}
catch (Exception e)
{
    // Code pour gérer l'exception qui est levée.
}
finally
{
    // Code à exécuter exception ou pas.
}
```

En résumé, voici une liste de suggestions relatives aux **meilleures pratiques** pour la gestion des exceptions (MSDN, 2013):

- La méthode que vous choisissez dépend de la fréquence à laquelle l'événement se produit.
  - **Si l'événement est véritablement exceptionnel** et constitue une erreur (par exemple une fin de fichier inattendue), **l'utilisation de la gestion des exceptions est plus indiquée, car la quantité de code exécutée en situation normale est moindre.**
  - **Si l'événement se produit régulièrement**, l'utilisation de la méthode **par programmation pour rechercher les erreurs est plus appropriée (c.-à-d. la validation)**. Par exemple, vous pouvez vérifier par programmation la présence d'une condition qui risque probablement de se produire sans avoir recours à la gestion des exceptions. Dans d'autres situations, l'utilisation de la gestion des exceptions pour intercepter un cas d'erreur est appropriée. **N'oubliez pas que lorsqu'une exception se produit, la gestion de l'exception prendra plus de temps qu'une simple validation.**
- Utilisez des blocs try/finally autour du code pouvant potentiellement générer une exception et **centralisez vos instructions catch en un point unique**. De cette façon, l'instruction *try* génère l'exception, l'instruction *finally* ferme ou libère des ressources et l'instruction *catch* gère l'exception à partir d'un point central.



- Veuillez à toujours **classer les exceptions dans les blocs catch de la plus spécifique à la moins spécifique**. Cette technique permet de gérer l'exception spécifique avant qu'elle ne passe à un bloc catch plus général.

## throw

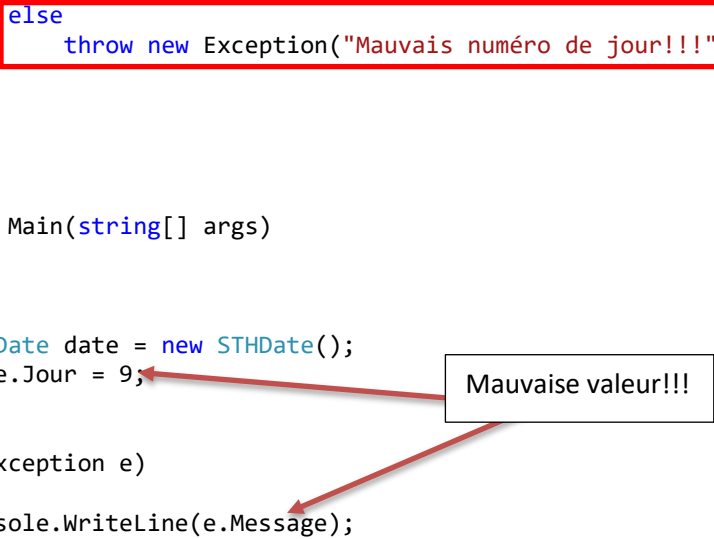
Les exceptions peuvent être générées explicitement par un programme à l'aide du mot clé **throw**. Donc si vous voulez volontairement générer une exception selon vos critères cette instruction sera très pratique.

**Exemple :** Générer une exception si le numéro du jour ne se trouve pas entre 1 et 7 concernant la propriété Jour de la classe STHDate.

```
public class STHDate
{
    private byte _iJour;

    public byte Jour
    {
        get
        {
            return _iJour;
        }
        set
        {
            if (value >= 1 && value <= 7)
                _iJour = value;
            else
                throw new Exception("Mauvais numéro de jour!!!");
        }
    }
}

...
static void Main(string[] args)
{
    try
    {
        STHDate date = new STHDate();
        date.Jour = 9;
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        Console.ReadKey();
    }
}
...
```



**Exercice 5. :** Ajouter un projet de type console, structurer selon MVVM et code l'exemple précédent.

**Exercice 6. :** Ajouter un nouveau projet de type console et structurer selon MVVM. Ajouter une classe **Personne** qui contient les propriétés suivantes :

- 1) **Nom** : Nom de la personne. Ne doit pas être vide (Générer une exception).
- 2) **Prénom** : Prénom de la personne. Ne doit pas être vide (Générer une exception).
- 3) **Age** : l'âge de la personne. Doit se situer entre 0 et 130. Générer une exception, lorsque non respecté.

Dans le programme principal, créer un objet **Personne** avec des informations de test. Ajouter un *try* afin d'attraper les exceptions si elles sont déclenchées et afficher un message d'erreur en conséquence.

## Créer vos propres classes d'exceptions

Vous avez la possibilité de créer vos propres classes d'exceptions afin d'en personnaliser la gestion. Pour ce faire, ces nouvelles classes doivent dériver de la classe de base **Exception**. Même si nous n'avons pas vu en détail l'héritage voici un exemple avec la classe **STHDate** qui implémentera une classe d'exception qui évite d'écrire du texte lors de l'envoi (throw) d'une exception afin de respecter l'encapsulation.

```

public class STHDateException : Exception
{
    #region Énumérations
    public enum États
    {
        MauvaisJour,
        MauvaisMois,
        MauvaiseAnnée
    }
    #endregion


    #region Propriétés
    public États État { get; set; }
    #endregion

    #region Constructeurs
    public STHDateException(États état)
    {
        État = état;
    }
    #endregion
}

public class STHDate
{
    private int _iJour;

    public int Jour
    {
        get { return _iJour; }
        set
        {
            if (value >= 1 && value <= 7)
                _iJour = value;
            else
                //throw new Exception("Mauvais numéro de jour!!!");
                throw new STHDateException(STHDateException.États.MauvaisJour);
        }
    }
}

```



**Exercice 7. :** Coder l'exemple suivant dans le projet contenant la classe **STHDate**.

**Exercice 8. :** Dans le projet contenant la classe **Personne**, ajouter la classe **PersonneException** en appliquant le même principe vu avec la classe **STHDateException**.

## Bibliographie

MSDN. (2013, 01 01). *Gestion et levée des exceptions*. Consulté le 04 07, 2013, sur MSDN:  
<http://msdn.microsoft.com/fr-fr/library/vstudio/5b2yeyab.aspx>