



Concepteur Développeur en Informatique

Développer des composants d'interface

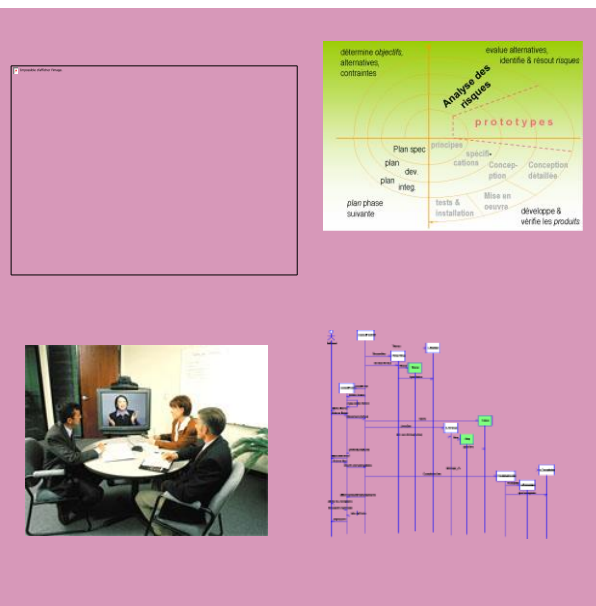
POO – Partie 7 Les exceptions

Accueil

Apprentissage

PAE

Evaluation



Localisation : U03-E03-S02

SOMMAIRE

1.	Introduction.....	3
2.	Exception	3
3.	Levée et propagation d'exception	5
4.	Un exemple de mise en œuvre de gestionnaires d'exceptions.	6
4.1.	Exemple 1 : Exception non gérée	6
4.2.	Exemple 2 : Exception gérée localement	7
4.1.	Exemple 2 : Exception gérée globalement.....	9
4.2.	Assurer une information correcte des développeurs	9
4.3.	Quel est le meilleur choix entre local et global ?.....	10
5.	Créer ses propres exceptions.....	12
5.1.	Utiliser des ressources externes	13

1. Introduction

Ce document introduit des techniques importantes de la programmation objet relatives à la gestion des erreurs.

Une application peut être rendue instable en fonction de différents facteurs qui sont notamment liés à la survenance d'erreurs lors de l'exécution de cette dernière.

Ces erreurs peuvent être le fait de l'utilisateur qui aura introduit des données erronées ou réalisé de fausses manœuvres.

Des erreurs peuvent également survenir en cas d'absence d'un élément nécessaire au bon fonctionnement de l'application comme un fichier ou un service indisponible.

Votre composant logiciel doit donc se défendre contre de telles situations potentielles. Cette pratique est connue sous le vocable de **programmation défensive**.

La programmation défensive se veut complémentaire de la programmation **préventive** qui veut que l'on intervienne en prévention de l'erreur : contrôle des informations communiquées par l'utilisateur avant de les introduire dans un calcul par exemple.

Il n'est pas souhaitable d'opposer ces deux méthodes : il convient de les utiliser en complémentarité.

La programmation défensive permettra de protéger votre code et d'assurer une meilleure **robustesse** à votre application, améliorant ainsi l'un des principaux critères de qualité d'une application.

La programmation défensive en objet s'appuie sur la **gestion des exceptions**.

2. Exception

Une exception est l'apparition, en cours d'exécution d'un programme, d'une erreur qui mène, si rien n'est fait pour l'intercepter, à l'arrêt du programme.

L'exception est déclenchée lorsqu'une situation inattendue survient.

L'exception est créée et se propage dans l'application jusqu'au point de traitement de celle-ci ou jusqu'à l'interruption du programme !

Les exceptions sont classées par nature d'anomalies. Les exceptions arithmétiques comme la division par 0, les exceptions relatives à la manipulation de fichiers, les exceptions qui surviennent lorsque de violation des règles de gestion, ... erreurs sont rencontrés lors de la manipulation d'objets métiers, un accès à un élément de tableau hors bornes ...

Les exceptions permettent de **séparer le code applicatif du code de traitements des erreurs** pouvant survenir.

La détection des erreurs et la gestion de leurs traitements associés sont donc mises en œuvre grâce à des mécanismes particuliers qui reposent sur les exceptions.

Il existe deux types d'exception : les exceptions générées par un programme en exécution (Runtime) et les exceptions générées par le Common Language Runtime.

Le tableau suivant affiche quelques exceptions levées sur des opérations de base automatiquement levées par le CLR. Vous les avez sûrement déjà rencontrées ...

<xref:System.ArithmeticException>	Classe de base pour les exceptions qui se produisent pendant des opérations arithmétiques, telles que <xref:System.DivideByZeroException> et <xref:System.OverflowException>.
<xref:System.ArrayTypeMismatchException>	Levée quand un tableau ne peut pas stocker un élément donné, car le type réel de l'élément est incompatible avec le type réel du tableau.
<xref:System.DivideByZeroException>	Levée lors d'une tentative de division d'une valeur intégrale par zéro.
<xref:System.IndexOutOfRangeException>	Levée lors d'une tentative d'indexation d'un tableau à l'aide d'un index qui est inférieur à zéro ou en dehors des limites du tableau.
<xref:System.InvalidCastException>	Levée quand une conversion explicite d'un type de base en interface ou en un type dérivé échoue au moment de l'exécution.
<xref:System.NullReferenceException>	Levée quand vous essayez de référencer un objet dont la valeur est <code>null</code> .
<xref:System.OutOfMemoryException>	Levée quand une tentative d'allocation de mémoire à l'aide de l'opérateur <code>new</code> échoue. Cela indique que la mémoire disponible pour le Commun Language Runtime est épuisée.

La classe **Exception** est la **classe de base** des exceptions. Toutes les exceptions héritent donc de cette classe Exception.

La création de nouvelles exceptions doit respecter des règles très précises de normalisation.

La plupart des exceptions qui dérivent directement de la classe Exception n'ajoutent aucune fonctionnalité à la classe Exception. Mais elles existent pour préciser le sens, la signification, de l'erreur rencontrée.

Par exemple, la classe **InvalidCastException** dérive directement de **SystemException**.

Ainsi, le mécanisme des exceptions permet, à partir de la définition de types signifiants, de faciliter le travail du développeur.

Si vous développez une application qui crée de nouvelles exceptions, vous devez dériver ces exceptions de la classe Exception. Pour conserver une bonne cohérence d'ensemble, vos classes d'exception personnalisées devront d'ailleurs plutôt dériver d'**ApplicationException** ou d'une classe dérivée de celle-ci.

3. Levée et propagation d'exception

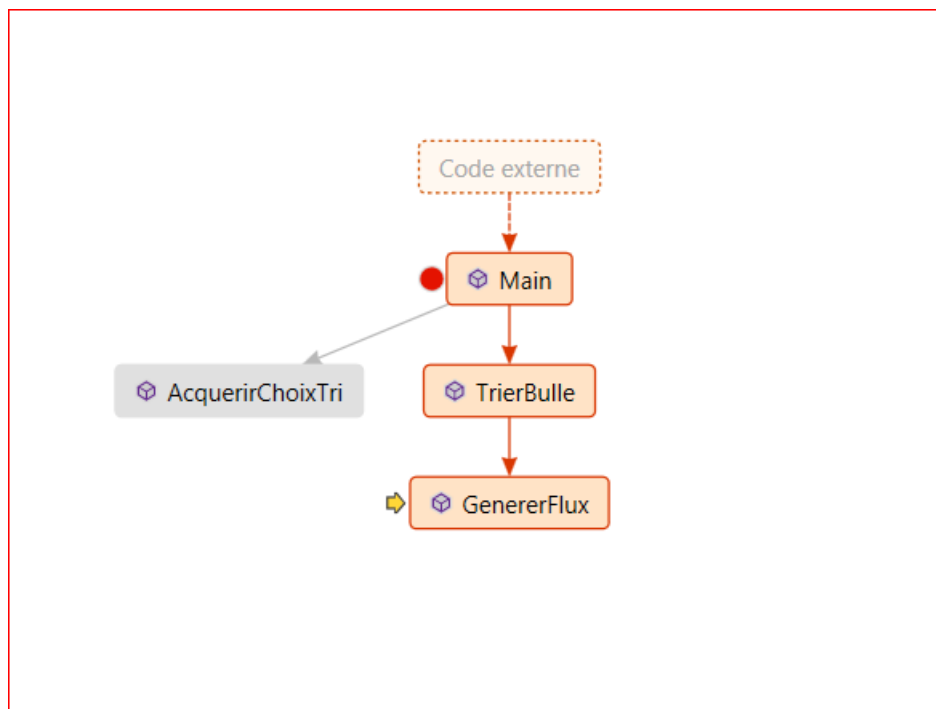
L'exception est créée et levée lorsqu'une erreur survient. Elle intervient en lien avec une opération qui se termine en erreur au sein d'une méthode.

L'exception se propage au sein de la pile des appels.

Dès qu'une exception survient, l'exécution normale du programme est interrompue et un gestionnaire d'exceptions est recherché d'abord dans le bloc d'instruction courant puis, s'il est absent, dans le bloc de la fonction appelante et à défaut, dans la fonction suivante dans la pile des appels.

En dernier lieu, arrivée à la première fonction, l'erreur s'affiche et le programme s'arrête si aucun gestionnaire d'exception n'a été trouvé.

Le schéma ci dessous représente la pile des appels des fonctions invoquées pour trier un tableau d'entiers réalisé lors d'un précédent exercice.



Si une exception survient dans la méthode GenererFlux, elle peut être interceptée :

- Localement dans cette fonction,
- Dans la fonction TrierBulle,
- Dans la fonction Main

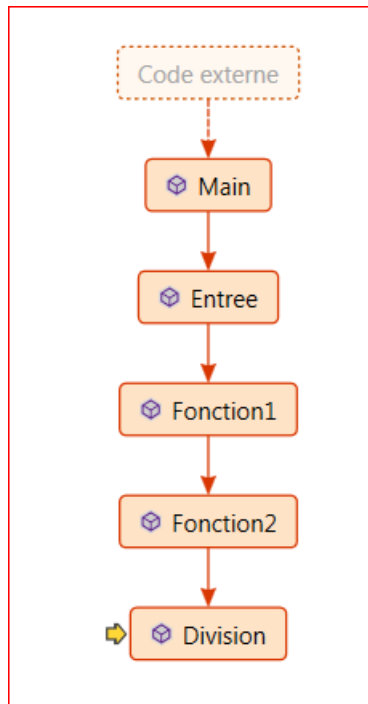
Il existe donc **deux stratégies pour traiter une exception**.

Soit l'exception est gérée localement au code qui la détecte, soit la méthode où l'exception est levée n'est pas candidate à son traitement et elle transmet alors cette exception à sa fonction appelante qui, à son tour, va devoir faire un choix : la traiter ou de nouveau la transmettre à sa fonction appelante.

Une exception se propage donc dans la pile des appels depuis la dernière fonction appelée jusqu'à la fonction appelante d'origine. Si l'exception n'est pas gérée dans la première fonction dans la pile des appels, le programme s'arrête et l'exception est affichée.

4. Un exemple de mise en œuvre de gestionnaires d'exceptions

Les exemples à suivre reposeront sur la pile des appels suivante.



Pile des appels de fonctions.

La fonction Main invoque la fonction Entree qui invoque à son tour la fonction1 Jusqu'à Division où l'erreur se produit.

4.1. Exemple 1 : Exception non gérée

Dans ce premier exemple, aucun gestionnaire d'exception n'a été mis en place.

```

class Exemple1
{
    1 référence
    internal static void Entree()
    {
        Fonction1();
        Console.ReadLine();
    }

    1 référence
    private static void Fonction1()
    {
        Fonction2();
    }

    1 référence
    private static void Fonction2()
    {
        int resultat = Division(10, 0);
    }

    1 référence
    private static int Division(int p1, int p2)
    {
        return p1 / p2;
    }
}
        
```

! L'exception DivideByZeroException n'a pas été gérée

Une exception non gérée du type 'System.DivideByZeroException' s'est produite dans GestionExceptions.exe

Informations supplémentaires : Tentative de division par zéro.

Conseils de débogage :

Assurez-vous que la valeur du dénominateur n'est pas égale à zéro avant d'effectuer une opération de division.

[Obtenir une aide d'ordre général pour cette exception.](#)

[Rechercher de l'aide en ligne complémentaire...](#)

Paramètres d'exception :

☐ Pause lorsque ce type d'exception est levé

Actions :

[Afficher les détails...](#)

[Copier le détail de l'exception dans le Presse-papiers](#)

[Ouvrir les paramètres d'exception](#)

Le programme s'arrête avec une erreur de type **System.DivideByZeroException**

4.2. Exemple 2 : Exception gérée localement

Nous allons ici mettre en place un mécanisme de gestion de l'exception en lieu et place où elle se produit.

Nous avons alors plusieurs possibilités qui permettent de gérer avec précision ou non les exceptions qui surviennent.

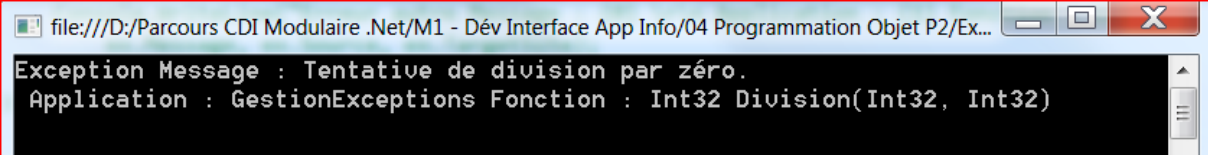
Nous entourons donc l'opération de division dans un bloc Try/Catch.

Dans ce premier exemple, nous traitons toutes les exceptions de la même manière : il existe un seul bloc Catch, dont les opérations seront exécutées quelle que soit la nature de l'exception. Pourquoi ?

```
private static int Division(int p1, int p2)
{
    try
    {
        return p1 / p2;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Exception Message : {0} \r\n Application : {1} Fonction : {2}",
            ex.Message, ex.Source, ex.TargetSite);
        return 0;
    }
}
```

Au sein du bloc catch est précisé le type d'exception intercepté. Ici, il s'agit du type le plus généraliste qui soit. Les exceptions sont définies par héritage d'un type parent plus général.

Le programme ne s'arrête plus car l'exception a été gérée.



Mais nous pouvons traiter plus finement l'exception.

Observons la hiérarchie d'héritage du type System.DivideByZeroException :

▲ Hiérarchie d'héritage

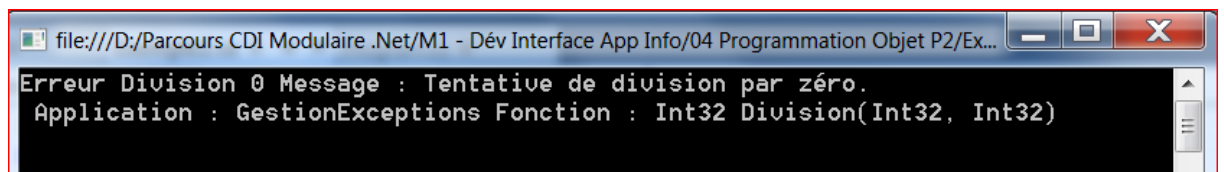
```
System.Object
System.Exception
System.SystemException
System.ArithmeticException
System.DivideByZeroException
```

Dans l'illustration qui suit, j'ai mis en place un gestionnaire qui traite les exceptions en fonction de leur type.

Il est nécessaire d'organiser les blocs catch depuis les erreurs du type le plus spécifique, ici `DividedByZeroException`, vers la gestion de l'erreur du type le plus général `System.Exception`.

```
private static int Division(int p1, int p2)
{
    try
    {
        return p1 / p2;
    }
    catch (DivideByZeroException ex)
    {
        Console.WriteLine("Erreur Division 0 Message : {0} \r\n Application : {1} Fonction : {2}",
            ex.Message, ex.Source, ex.TargetSite);
        return 0;
    }
    catch (ArithmeticException ex)
    {
        Console.WriteLine("Erreur arithmétique autre que div 0 Message : {0} \r\n Application : {1} Fonction : {2}",
            ex.Message, ex.Source, ex.TargetSite);
        return 0;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Erreur autre Message : {0} \r\n Application : {1} Fonction : {2}",
            ex.Message, ex.Source, ex.TargetSite);
        return 0;
    }
}
```

Le résultat obtenu est sans équivoque :



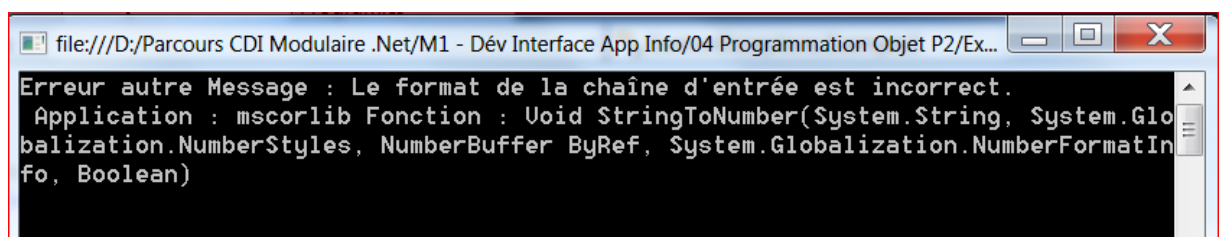
The screenshot shows a console window with the following text:

```
Erreur Division 0 Message : Tentative de division par zéro.
Application : GestionExceptions Fonction : Int32 Division(Int32, Int32)
```

Pour les besoins du test, je vais mettre sous surveillance une autre opération. Ici une opération de conversion. Dans un objectif de test uniquement car cette opération n'a aucune raison d'être ...

```
private static int Division(int p1, int p2)
{
    try
    {
        string s = "A";
        p2 = int.Parse(s);
        return p1 / p2;
    }
}
```

En observant le résultat obtenu, nous constatons que l'erreur a provoqué l'arrêt du programme et le branchement au bloc catch généraliste du fait de la nature de l'erreur.



The screenshot shows a console window with the following text:

```
Erreur autre Message : Le format de la chaîne d'entrée est incorrect.
Application : mscorlib Fonction : Void StringToNumber(System.String, System.Globalizati
on.NumberStyles, NumberBuffer ByRef, System.Globalization.NumberFormatInfo, Boolean)
```

L'exécution du programme se poursuit après le bloc try/catch

4.1. Exemple 2 : Exception gérée globalement

Ici, c'est dans la fonction appelante que les erreurs sont prises en charge.

Ici, l'erreur de conversion va provoquer l'arrêt de l'exécution de la fonction sur l'instruction Parse et déléguer la prise en charge de l'erreur au gestionnaire défini dans la fonction2.

L'opération de division ne sera pas réalisée.

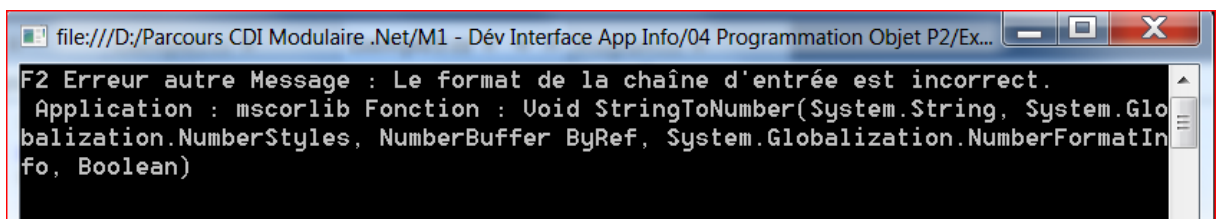
```

1 référence
private static void Fonction2()
{
    try
    {
        int resultat = Division(10, 0);
    }
    catch (DivideByZeroException ex)
    {
        Console.WriteLine("F2 Erreur Division 0 Message : {0} \r\n Application : {1} Fonction : {2}",
            ex.Message, ex.Source, ex.TargetSite);
    }
    catch (ArithmeticException ex)
    {
        Console.WriteLine("F2 Erreur arithmétique autre que div 0 Message : {0} \r\n Application : {1} Fonction : {2}",
            ex.Message, ex.Source, ex.TargetSite);
    }
    catch (Exception ex)
    {
        Console.WriteLine("F2 Erreur autre Message : {0} \r\n Application : {1} Fonction : {2}",
            ex.Message, ex.Source, ex.TargetSite);
    }
}

1 référence
private static int Division(int p1, int p2)
{
    string s = "A";
    p2 = int.Parse(s);
    return p1 / p2;
}

```

Résultat :



```

file:///D:/Parcours CDI Modulaire .Net/M1 - Dév Interface App Info/04 Programmation Objet P2/Ex...
F2 Erreur autre Message : Le format de la chaîne d'entrée est incorrect.
Application : mscorlib Fonction : Void StringToNumber(System.String, System.Globali
balization.NumberStyles, NumberBuffer ByRef, System.Globalization.NumberFormatIn
fo, Boolean)

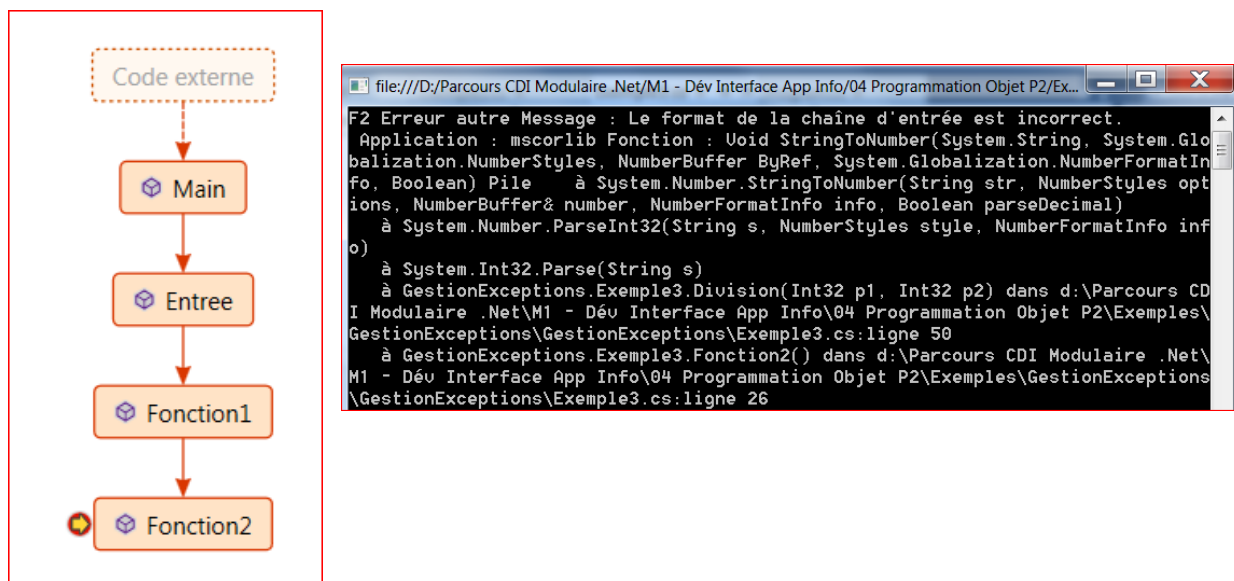
```

A noter : Gérer des exceptions de base du CLR n'est pas une bonne pratique en programmation. Il faut éviter qu'elles soient levées.

4.2. Assurer une information correcte des développeurs

Vous aurez peut-être remarqué que les messages affichés ne permettent pas au développeur de déterminer rapidement l'origine de l'exception, en particulier si le gestionnaire la prenant en charge est situé dans la pile à une position éloignée de l'appel de la fonction où est survenue l'erreur.

Pour une information plus complète vous devez récupérer l'information relative à la pile des appels présente dans la propriété **StackTrace** de l'exception.

Illustration à partir de l'exemple précédent :

La trace nous permet de déterminer que l'exception a eu lieu au sein de la fonction Division invoquée depuis la fonction 2 où elle a été traitée.

4.3. Quel est le meilleur choix entre local et global ?

Il s'agit le plus souvent d'un mixte entre deux.

Les gestionnaires globaux ne permettent pas de proposer aisément un processus de reprise après incident efficace. Ils permettent toutefois de s'assurer que le programme ne plante pas lamentablement, laissant l'utilisateur sans autre recours que de redémarrer l'application !

Dans l'exemple de la division par 0 ou d'une conversion de type, il nous faudrait plutôt mettre en place un gestionnaire local qui permettrait à la personne de fournir une valeur différente de 0 et convertible en entier.

Dans l'exemple qui suit, je vais utiliser les deux approches mixées.

Le traitement de la division par zéro sera pris en charge localement alors que le problème de conversion sera lui géré par la fonction appelante. Il s'agit ici d'un cas d'école qui devrait bien entendu être adapté pour des besoins réels. Ici, si aucune action n'est réalisée lors de l'erreur de conversion, il suffit d'ignorer l'exception en question.

Pour intercepter une exception particulière et transmettre sa prise en charge à un gestionnaire plus global, il nous faut appeler la méthode **throw** sans préciser l'objet exception considéré. L'exception est alors **propagée de nouveau** pour être interceptée au niveau du **point d'appel précédent dans la pile**. Cette approche permet de ne pas modifier l'objet d'origine et l'historique des appels.

```
1 référence
private static int Division(int p1, int p2)
{
    try
    {
        string s = "A";
        p2 = int.Parse(s);
        return p1 / p2;
    }
    catch (DivideByZeroException ex)
    {
        Console.WriteLine("Erreur Division 0 Message : {0} \r\n Application : {1} Fonction : {2}",
            ex.Message, ex.Source, ex.TargetSite);
        return 0;
    }
    catch (FormatException ex)
    {
        // actions réalisées lorsque cette exception survient

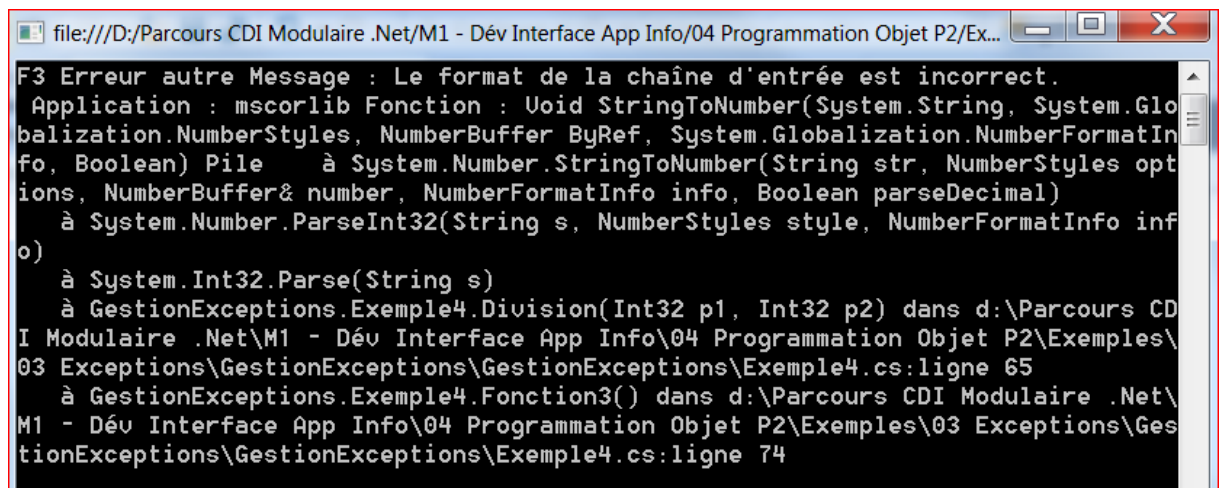
        throw;
    }
}
```

La fonction appelée

```
1 référence
private static void Fonction3()
{
    try
    {
        int resultat = Division(10, 0);
    }

    catch (Exception ex)
    {
        Console.WriteLine("F3 Erreur autre Message : {0} \r\n Application : {1} Fonction : {2} Pile {3} ",
            ex.Message, ex.Source, ex.TargetSite, ex.StackTrace);
    }
}
```

L'exception récupérée dans la fonction appelante fonction 3 :



```
file:///D:/Parcours CDI Modulaire .Net/M1 - Dév Interface App Info/04 Programmation Objet P2/Ex...
F3 Erreur autre Message : Le format de la chaîne d'entrée est incorrect.
Application : mscorlib Fonction : Void StringToNumber(System.String, System.Globali
zation.NumberStyles, NumberBuffer ByRef, System.Globalization.NumberFormatInfo, Boolean) Pile à System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
à System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
à System.Int32.Parse(String s)
à GestionExceptions.Exemple4.Division(Int32 p1, Int32 p2) dans d:\Parcours CDI Modulaire .Net\M1 - Dév Interface App Info\04 Programmation Objet P2\Exemples\03 Exceptions\GestionExceptions\GestionExceptions\Exemple4.cs:ligne 65
à GestionExceptions.Exemple4.Fonction3() dans d:\Parcours CDI Modulaire .Net\M1 - Dév Interface App Info\04 Programmation Objet P2\Exemples\03 Exceptions\GestionExceptions\GestionExceptions\Exemple4.cs:ligne 74
```

5. Créer ses propres exceptions

Il est important de créer ses propres exceptions au niveau du domaine (métier) afin de pouvoir les gérer au mieux lorsqu'un incident est à l'origine de la levée de l'une d'elles.

Quelques règles à respecter pour nos exceptions métier :

- Elles doivent hériter de la classe **ApplicationException** ou d'une classe dérivée de cette dernière
- Elles peuvent être sérialisées et doivent donc implémenter l'interface **ISerializable** ainsi qu'un constructeur avec en argument deux paramètres en entrée respectivement de type **SerializationInfo** et **StreamingContext**
- Elles implémentent un constructeur qui peut embarquer l'exception d'origine.

```

/// </summary>
[Serializable]
4 références
public class SalarieException : ApplicationException, ISerializable
{
    private string _idMessage = string.Empty;
    /// <summary> ...
    0 références
    public string IdMessage
    {
        get { return _idMessage; }
        set { _idMessage = value; }
    }
    /// <summary> ...
    0 références
    public SalarieException()
        : base()
    { }
    /// <summary> ...
    0 références
    public SalarieException(string IdMessage, string message)
        : base(message)
    { _idMessage = IdMessage; }
    /// <summary> ...
    0 références
    public SalarieException(string IdMessage, string message, Exception inner)
        : base(message, inner)
    { _idMessage = IdMessage; }

    /// <summary> ...
    0 références
    protected SalarieException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    { }
}

```

Nous verrons toutefois qu'il n'est pas nécessairement utile de créer de trop nombreuses exceptions. Et nous reviendrons d'ailleurs sur certains choix initiaux qui ont pu être retenus pour protéger les attributs de nos classes.

5.1. Utiliser des ressources externes

Les messages de vos exceptions doivent être de préférence stockés dans des fichiers de ressources externes.

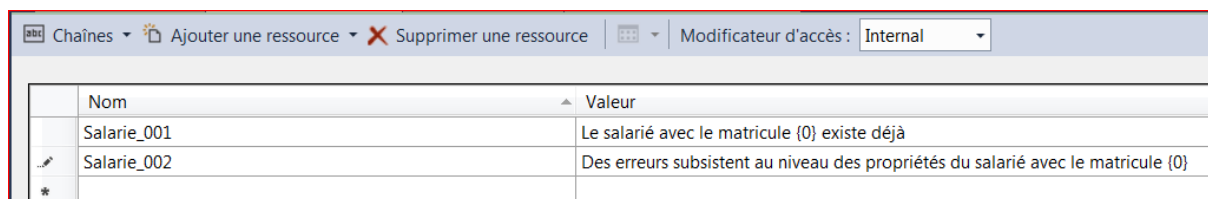
Pour ajouter un fichier de ressources, cliquez depuis l'explorateur de solutions après sélection d'un projet sur **ajouter nouvel élément** et choisissez un élément de **type fichier de ressources extension .resx**.

Cette démarche permet de mieux structurer nos messages et de faciliter la mise en place d'une version régionalisée de l'application (culture spécifique éventuellement dans une autre langue).

Il est utile d'affecter un **identifiant** à chacun de vos messages transmis lors de la levée de l'exception.

Choisissez une règle d'identification simple qui vous permettra d'administrer simplement ce fichier de messages. J'ai choisi ici la concaténation du type et d'un numéro d'ordre de 3 chiffres.

Cet identifiant sera affecté à la propriété **IdMessage** de l'exception permettant de tracer efficacement celle-ci ultérieurement.



	Nom	Valeur
	Salarie_001	Le salarié avec le matricule {0} existe déjà
	Salarie_002	Des erreurs subsistent au niveau des propriétés du salarié avec le matricule {0}

Levée de l'exception en tenant compte de la culture courante pour la mise en forme des valeurs régionalisées (chaînes, date, monétaires, séparateur décimal,...)

Les cultures sont accessibles depuis l'espace de noms **System.Globalization**.

```
throw new SalarieException(Messages.Salarie_001,
    string.Format(CultureInfo.CurrentCulture, Messages.Salarie_001, Matricule));
```

Vous pouvez réfléchir à la factorisation de certaines exceptions. Ainsi, nous pourrions ici mettre en œuvre une exception plus générique applicable à tout objet de notre système déjà présent. Cette solution ne présente toutefois pas que des avantages.