Livret – 11

Le langage C# dans .Net

Exceptions, multi-threading.

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

SOMMAIRE

Les exceptions en

Plan général:

1. Les exceptions : syntaxe, rôle, classes

Rappelons au lecteur que la sécurité de fonctionnement d'une application peut être rendue instable par toute une série de facteurs :

Des problèmes liés au matériel : par exemple la perte subite d'une connexion à un port, un disque défectueux... Des actions imprévues de l'utilisateur, entraînant par exemple une division par zéro... Des débordements de stockage dans les structures de données...

Toutefois les faiblesses dans un logiciel pendant son exécution, peuvent survenir : lors des entrées-sorties, lors de calculs mathématiques interdits (comme la division par zéro), lors de fausses manoeuvres de la part de l'utilisateur, ou encore lorsque la connexion à un périphérique est inopinément interrompue, lors d'actions sur les données. Le logiciel doit donc se " ***défendre*** " contre de tels incidents potentiels, nous nommerons cette démarche la programmation défensive !

Programmation défensive
La ***programmation défensive*** est une attitude de pensée consistant à prévoir que le logiciel sera soumis à des défaillances dues à certains paramètres externes ou internes et donc à prévoir une réponse adaptée à chaque type de situation.

En programmation défensive il est possible de protéger directement le code à l'aide de la notion d'exception. L'objectif principal est d'améliorer la qualité de " ***robustesse*** " (définie par B.Meyer) d'un logiciel. L'utilisation des exceptions avec leur mécanisme intégré, autorise la construction rapide et efficace de logiciels robustes.

Rôle d'une exception
Une exception est chargée de signaler un comportement ***exceptionnel*** (mais prévu) d'une partie spécifique d'un logiciel. Dans les langages de programmation actuels, les exceptions font partie du langage lui-même. C'est le cas de C# qui intègre les exceptions comme une classe particulière: la classe **Exception**. Cette classe contient un nombre important de classes dérivées.

Comment agit une exception
Dès qu'une erreur se produit comme un manque de mémoire, un calcul impossible, un fichier inexistant, un transtypage non valide,..., un objet de la classe adéquate dérivée de la classe **Exception** est instancié. Nous dirons que le logiciel " ***déclenche une exception*** ".

1.1 Comment gérer une exception dans un programme

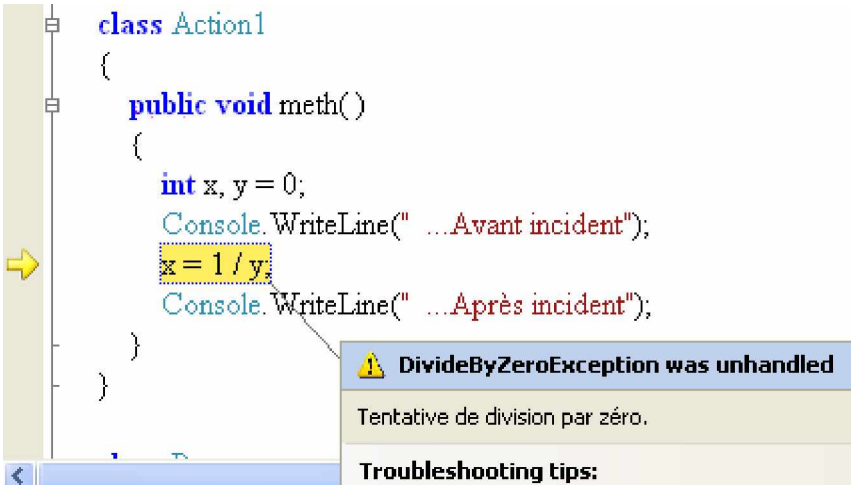
Programme sans gestion de l'exception

Soit un programme C# contenant un incident d'exécution (une division par zéro dans l'instruction x = 1/0;) dans la méthode meth() de la classe Action1, cette méthode est appelée dans la classe UseAction1 à travers un objet de classe Action1 :

```
class Action1 {
    public void meth()
```

```
    {
        int x, y =0;
        Console.WriteLine("  ...Avant incident");
        x = 1 / y;
        Console.WriteLine("  ...Après incident");
    }
}

class Program {
    static void Main(string[] args)
    {
        Action1 Obj = new Action1( );
        Console.WriteLine("Début du programme.");
        Obj.meth( );
        Console.WriteLine("Fin du programme.");
    }
}
```

Lors de l'exécution, après avoir affiché les chaînes "**Début du programme**" et " **...Avant incident**", le programme s'arrête et le CLR signale une erreur. Voici ci-dessous l'affichage obtenu dans Visual C# :



Sur la console l'affichage est le suivant :

> **Début du programme.**
> **...Avant incident.**
>
> *Le programme s'est arrêté à cet endroit et ne peut plus pouruivre son exécution.*

Que s'est-il passé ?

La méthode Main :
- a instancié un objet Obj de classe Action1,
- a affiché sur la console la phrase " Début du programme ",
- a invoqué la méthode meth() de l'objet Obj,
- a affiché sur la console la phrase " ...Avant incident",
- a exécuté l'instruction "x = 1/0;"

Dès que l'instruction "x = 1/0;" a été exécutée celle-ci a provoqué un incident. **En fait une exception de la classe DivideByZeroException a été "levée" (un objet de cette classe**

a été instancié) par le CLR, cette classe hérite de la classe ArithmeticException selon la hiérarchie d'héritage suivante de .Net Framework :

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

La classe mère de **toutes** les exceptions de .Net Framework est la classe **Exception**.

Le CLR a arrêté le programme immédiatement à cet endroit parce qu'elle n'a pas trouvé de code d'interception de cette exception qu'il a levée automatiquement :

```
class Action1 {
      public void meth()
      {
         int x, y =0;
         Console.WriteLine(" ...Avant incident");
         x = 1 / y;
         Console.WriteLine(" ...Après incident");
      }
}

class Program {
      static void Main(string[] args)
      {
         Action1 Obj = new Action1();
         Console.WriteLine("Début du programme.");
         Obj.meth();
         Console.WriteLine("Fin du programme.");
      }
}
```

Sortie du bloc

Sortie du bloc

Nous allons voir comment intercepter (on dit aussi "attraper" - to catch) cette exception afin de faire réagir notre programme pour qu'il ne s'arrête pas brutalement.

Programme avec gestion de l'exception

C# possède une instruction qui permet d'intercepter des exceptions dérivant de la classe Exception :

try ... catch

On dénomme cette instruction : un gestionnaire d'exceptions. Syntaxe **minimale** d'un tel gestionnaire try … catch :

```
try {
```

<lignes de code à protéger>

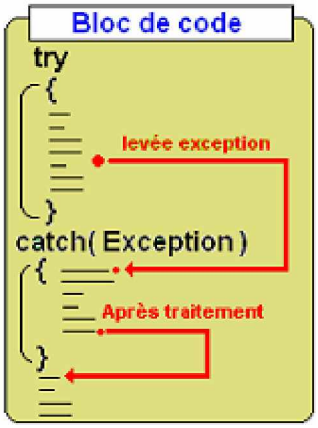
}catch (**UneException**) **{**

<lignes de code réagissant à l'exception UneException >

}

Le type **UneException** est obligatoirement une classe qui **hérite de la classe Exception**.

Schéma du fonctionnement d'un tel gestionnaire :



Le gestionnaire d'exception "déroute" l'exécution du programme vers le bloc d'interception catch qui traite l'exception (exécute le code contenu dans le bloc catch), puis renvoie et continue l'exécution du programme vers le code situé après le gestionnaire lui-même.

1.2 Principe de fonctionnement de l'interception

Dès qu'une **exception est levée** (instanciée), le CLR **stoppe immédiatement** l'exécution normale du programme à la **recherche d'un gestionnaire** d'exception susceptible d'intercepter (saisir) et de traiter cette exception. Cette recherche s'effectue à partir du **bloc englobant** et se poursuit sur les blocs plus englobants si aucun gestionnaire de **cette exception** n'a été trouvé.

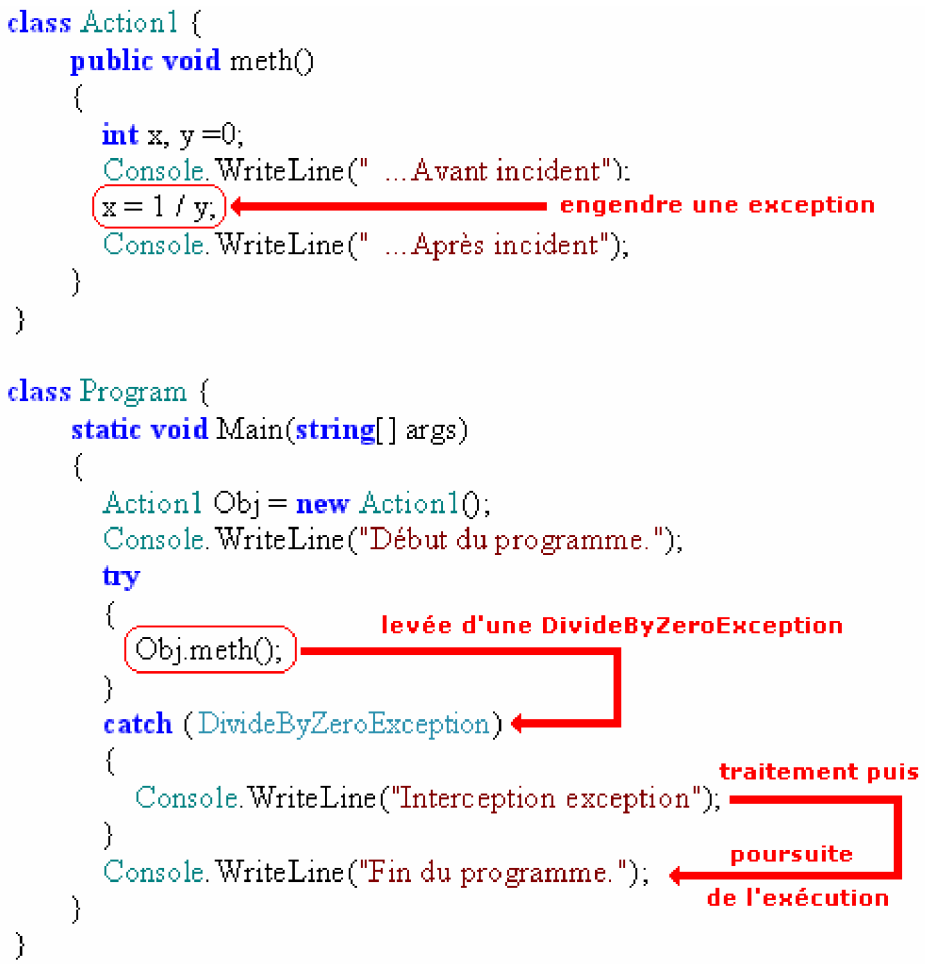
Soit le même programme C# que précédemment, contenant un incident d'exécution (une division par zéro dans l'instruction x = 1/0;). Cette fois nous allons gérer l'incident grâce à un gestionnaire d'exception **try..catch** dans le bloc englobant immédiatement supérieur**.**

Programme avec traitement de l'incident par try...catch :

```
class Action1 {
      public void meth()
      {
         int x, y =0;
         Console.WriteLine("  ...Avant incident");
         x = 1 / y;
         Console.WriteLine("  ...Après incident");
      }
 }
```

```
class Program {
      static void Main(string[] args)
      {
         Action1 Obj = new Action1();
         Console.WriteLine("Début du programme.");
         try
         {
            Obj.meth();
         }
         catch (DivideByZeroException)
         {
            Console.WriteLine("Interception exception");
         }
         Console.WriteLine("Fin du programme.");
      }
}
```

Figuration du déroulement de l'exécution de ce programme :



Ci-dessous l'affichage obtenu sur la console lors de l'exécution de ce programme :

Début du programme.
...Avant incident
Interception exception
Fin du programme.

- Nous remarquons que le CLR a donc bien exécuté le code d'interception situé dans le corps du "**catch** (DivideByZeroException){...}", il a poursuivi l'exécution normale après le gestionnaire.
- Le gestionnaire d'exception se situe dans la méthode Main (code englobant) qui appelle la méthode meth( ) qui lève l'exception.

Il est aussi possible d'atteindre l'objet d'exception qui a été instancié (ou levé) en déclarant un identificateur local au bloc catch du gestionnaire d'exception try … catch, cet objet est disponible dans tout le corps de la clause catch. Dans l'exemple qui suit, on déclare un objet Except de classe DivideByZeroException et l'on lit le contenu de deux de ses propriétés Message et Source :

```
try{
    Obj.meth();
}
catch ( DivideByZeroException Except ) {
   // accès aux membres publiques de l'objet Except :
  Console.WriteLine("Interception exception message : " + Except.Message);
  Console.WriteLine("Interception exception source : " + Except.Source);
  ….
}
```

2. Interception d'exceptions hiérarchisées

2.1 Interceptions de plusieurs exceptions

Dans un gestionnaire try...catch, il est en fait possible d'intercepter plusieurs types d'exceptions différentes et de les traiter.

Ci-après nous montrons la syntaxe d'un tel gestionnaire qui fonctionne comme un sélecteur ordonné, ce qui signifie qu'**une seule clause d'interception est exécutée**.

Dès qu'une exception intervient dans le < bloc de code à protéger>, le CLR scrute séquentiellement toutes les clauses **catch** de la première jusqu'à la nième. Si l'exception actuellement levée est d'un des types présents dans la liste des clauses le traitement associé est effectué, la scrutation est abandonnée et le programme poursuit son exécution après le gestionnaire.

```
try {
 < bloc de code à protéger>
}
catch ( TypeException1 E ) { <Traitement TypeException1 > }
catch ( TypeException2 E ) { <Traitement TypeException2 > }
  .....
 catch ( TypeExceptionk E ) { <Traitement TypeExceptionk > }
```

Où TypeException1, TypeException12, ... , TypeExceptionk sont des classes d'exceptions obligatoirement toutes **distinctes.**

Seule une seule clause **catch** (TypeException E) {...}est exécutée (celle qui correspond au bon type de l'objet d'exception instancié).

Exemple théorique :

Soit SystemException la classe des exceptions prédéfinies dans le nom d'espace System de .Net Framework, InvalidCastException, IndexOutOfRangeException, NullReferenceException et ArithmeticException sont des classes dérivant directement de la classe SystemException.

Supposons que la méthode meth() de la classe Action2 puisse lever quatre types différents d'exceptions: InvalidCastException, IndexOutOfRangeException, NullReferenceException et ArithmeticException.

Notre gestionnaire d'exceptions est programmé pour intercepter l'une de ces 4 catégories. Nous figurons ci-dessous les trois schémas d'exécution correspondant chacun à la levée (l'instanciation d'un objet) d'une exception de l'un des trois types et son interception :

```
class Action2    {
      public void meth( )
      {
         // une exception est levée .....
      }
   }

   class Program    {
      static void Main(string[] args)
      {
         Action2 Obj = new Action2();
         Console.WriteLine("Début du programme.");
         try
         {
            Obj.meth();
         }
         catch ( InvalidCastException E )
         {
            Console.WriteLine("Interception InvalidCastException");
         }
         catch ( IndexOutOfRangeException E )
         {
            Console.WriteLine("Interception IndexOutOfRangeException ");
         }
         catch ( NullReferenceException E )
         {
            Console.WriteLine("Interception NullReferenceException");
         }
         catch ( ArithmeticException E )
         {
            Console.WriteLine("Interception ArithmeticException");
         }
         Console.WriteLine("Fin du programme.");
      }
    }
```

Nous figurons ci-après deux schémas d'interception sur l'ensemble des quatre possibles d'une éventuelle exception qui serait levée dans la méthode meth( ) de l'objet Obj.

- **Schéma d'interception d'une IndexOutOfRangeException :**

```
static void Main(string[] args)
{
   Action2 Obj = new Action2();
   Console.WriteLine("Début du programme.");
   try
   {
      Obj.meth();
   }
   catch ( InvalidCastException E )
   {
      Console.WriteLine("Interception InvalidCastException");
   }
   catch ( IndexOutOfRangeException E )
   {
      Console.WriteLine("Interception IndexOutOfRangeException ");
   }
   catch ( NullReferenceException E )
   {
      Console.WriteLine("Interception NullReferenceException");
   }
   catch ( ArithmeticException E )
   {
      Console.WriteLine("Interception ArithmeticException");
   }
   Console.WriteLine("Fin du programme.");
}
```

levée d'une IndexOutOfRangeException

Traitement

poursuite de l'exécution après le bloc du gestionnaire

- **Schéma d'interception d'une ArithmeticException :**

```
static void Main(string[] args)
{
   Action2 Obj = new Action2();
   Console.WriteLine("Début du programme.");
   try
   {
      Obj.meth();
   }
   catch ( InvalidCastException E )
   {
      Console.WriteLine("Interception InvalidCastException");
   }
   catch ( IndexOutOfRangeException E )
   {
      Console.WriteLine("Interception IndexOutOfRangeException ");
   }
   catch ( NullReferenceException E )
   {
      Console.WriteLine("Interception NullReferenceException");
   }
   catch ( ArithmeticException E )
   {
      Console.WriteLine("Interception ArithmeticException");
   }
   Console.WriteLine("Fin du programme.");
}
```

levée d'une ArithmeticException

Traitement

poursuite de l'exécution après le bloc du gestionnaire

2.2 Ordre d'interception d'exceptions hiérarchisées

Dans un gestionnaire **try...catch** comprenant plusieurs clauses, la recherche de la clause catch contenant le traitement de la classe d'exception appropriée, s'effectue séquentiellement dans l'ordre d'écriture des lignes de code.

Soit le pseudo-code C# suivant :

```
try {
  < bloc de code à protéger générant un objet exception>
}
catch ( TypeException1 E ) { <Traitement TypeException1 > }
catch ( TypeException2 E ) { <Traitement TypeException2 > }
  .....
catch ( TypeExceptionk E ) { <Traitement TypeExceptionk > }
```

La recherche va s'effectuer comme si le programme contenait des **if...else if...** imbriqués :

```
if (<Objet exception> is TypeException1)  { <Traitement TypeException1 > }
else if (<Objet exception> is TypeException2)  { <Traitement TypeException2 > }
...
else if (<Objet exception> is TypeExceptionk)  { <Traitement TypeExceptionk > }
```

Les tests sont effectués sur l'appartenance de l'objet d'exception à une classe à l'aide de l'opérateur **is**.

> Signalons que l'opérateur **is** agit sur une classe et ses classes filles (sur une hiérarchie de classes), c'est à dire que tout objet de classe TypeExceptionX est aussi considéré comme un objet de classe parent au sens du test d'appartenance en particulier cet objet de classe TypeExceptionX est aussi considéré objet de classe Exception qui est la classe mère de toutes les exceptions C#.

Le test d'appartenance de classe dans la recherche d'une clause **catch** fonctionne d'une façon identique à l'opérateur **is** dans les if...else

On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs gestionnaires dans l'ordre inverse de la hiérarchie.

Exemple : Soit une hiérarchie d'exceptions de C#

```
System.Exception
  |__System.SystemException
        |__System.InvalidCastException
        |__System.IndexOutOfRangeException
        |__System.NullReferenceException
        |__System.ArithmeticException
```

Soit le modèle de gestionnaire d'interception déjà fourni plus haut :

```
try {   < bloc de code à protéger générant un objet exception> }
catch (InvalidCastException E ) { <Traitement InvalidCastException > }
catch ( IndexOutOfRangeException E ) { <Traitement IndexOutOfRangeException > }
catch (NullReferenceException E ) { <Traitement NullReferenceException > }
```

catch (ArithmeticException E) { <Traitement ArithmeticException > }

Supposons que nous souhaitions intercepter une cinquième classe d'exception, par exemple une DivideByZeroException**,** nous devons rajouter une clause :

catch (DivideByZeroException E) { <Traitement DivideByZeroException > }

Insérons cette clause en dernier dans la liste des clauses d'interception :

```
try
 {
    Obj.meth();
 }
 catch ( InvalidCastException E )
 {
    Console.WriteLine("Interception InvalidCastException");
 }
 catch ( IndexOutOfRangeException E )
 {
    Console.WriteLine("Interception IndexOutOfRangeException ");
 }
 catch ( NullReferenceException E )
 {
    Console.WriteLine("Interception NullReferenceException");
 }
 catch ( ArithmeticException E )
 {
    Console.WriteLine("Interception ArithmeticException");
 }
 catch (DivideByZeroException E )
 {
    Console.WriteLine("Interception DivideByZeroException ");
 }
 Console.WriteLine("Fin du programme.");
}
```

Nous lançons ensuite la compilation de cette classe et nous obtenons un message d'erreur :

Error 1 : Une clause catch précédente intercepte déjà toutes les expressions de this ou d'un super type ('System.ArithmeticException')

Le compilateur proteste à partir de la clause **catch** (DivideByZeroException E) en nous indiquant que l'exception est déjà interceptée.

Que s'est-il passé ?

Le fait de placer en premier la clause **catch** (ArithmeticException E) chargée d'intercepter les exceptions de classe ArithmeticException implique que n'importe quelle exception héritant de ArithmeticException comme par exemple DivideByZeroException, est considérée comme une ArithmeticException. Dans un tel cas cette DivideByZeroException est interceptée par la clause **catch** (ArithmeticException E) mais elle **n'est jamais** interceptée par la clause **catch** (DivideByZeroException E) .

Le seul endroit où le compilateur C# acceptera l'écriture de la **catch** (DivideByZeroException E) se situe dans cet exemple **avant la clause catch** (ArithmeticException E). Ci-dessous l'écriture d'un programme correct :

```
try
  {
    Obj.meth();
  }
  catch ( InvalidCastException E )
  {
    Console.WriteLine("Interception InvalidCastException");
  }
  catch ( IndexOutOfRangeException E )
  {
    Console.WriteLine("Interception IndexOutOfRangeException ");
  }
  catch ( NullReferenceException E )
  {
    Console.WriteLine("Interception NullReferenceException");
  }
  catch (DivideByZeroException E )
  {
    Console.WriteLine("Interception DivideByZeroException ");
  }
  catch ( ArithmeticException E )
  {
    Console.WriteLine("Interception ArithmeticException");
  }
  Console.WriteLine("Fin du programme.");
}
```

Dans ce cas la recherche séquentielle dans les clauses permettra le filtrage correct des classes filles puis ensuite le filtrage des classes mères.

On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs clauses dans l'ordre inverse de la hiérarchie.

3. Redéclenchement d'une exception mot-clef : throw

Il est possible de déclencher soi-même des exceptions en utilisant l'instruction **throw**, voir même de déclencher des exceptions personnalisées ou non. Une exception personnalisée est une classe héritant de la classe System.Exception définie par le développeur lui-même.

3.1 Déclenchement manuel d'une exception de classe déjà existante

Le CLR peut déclencher une exception automatiquement comme dans l'exemple de la levée d'une DivideByZeroException lors de l'exécution de l'instruction "x = 1/y ;".

Le CLR peut aussi lever (déclencher) une exception à votre demande suite à la rencontre d'une instruction **throw**. Le programme qui suit lance une ArithmeticException avec le message "Mauvais calcul !" dans la méthode **meth( )** et intercepte cette exception dans le bloc englobant **Main**. Le traitement de cette exception consiste à afficher le contenu du champ message de l'exception grâce à la propriété **Message** de l'exception :

```
class Action3
{
   public void meth()
   {
      int x = 0;
      Console.WriteLine("  ...Avant incident");
      if (x == 0)
         throw new ArithmeticException("Mauvais calcul !");
      Console.WriteLine("  ...Après incident");
   }
}

class Program {
   static void Main(string[] args)
   {
      Action3 Obj = new Action3();
      Console.WriteLine("Début du programme.");
      try
      {
         Obj.meth();
      }
      catch (ArithmeticException E)
      {
         Console.WriteLine("Interception ArithmeticException : "+E.Message);
      }
      Console.WriteLine("Fin du programme.");
   }
}
```

Résultats de l'exécution du programme précédent :

Début du programme.
...Avant incident
Interception ArithmeticException : Mauvais calcul !
Fin du programme.

3.2 Déclenchement manuel d'une exception personnalisée

Pour une exception personnalisée, le mode d'action est strictement identique, il vous faut seulement auparavant créer une nouvelle classe **héritant obligatoirement de la classe Exception** ou de n'importe laquelle de ses sous-classes.

Reprenons le programme précédent et créons une classe d'exception que nous nommerons ArithmeticExceptionPerso héritant de la classe des ArithmeticException puis exécutons ce programme :

```
class ArithmeticExceptionPerso : ArithmeticException
{
    public ArithmeticExceptionPerso(String s) : base(s)
    {
    }
}

class Action3
{
    public void meth()
    {
       int x = 0;
       Console.WriteLine("  ...Avant incident");
```

```
        if (x == 0)
            throw new ArithmeticExceptionPerso ("Mauvais calcul !");
        Console.WriteLine("  ...Après incident");
    }
 }

class Program {

    static void Main(string[] args)
    {
        Action3 Obj = new Action3();
        Console.WriteLine("Début du programme.");
        try
        {
            Obj.meth();
        }
        catch (ArithmeticExceptionPerso E)
        {
            Console.WriteLine("Interception ArithmeticExceptionPerso: "+E.Message);
        }
        Console.WriteLine("Fin du programme.");
    }
}
```

Résultats de l'exécution du programme précédent :

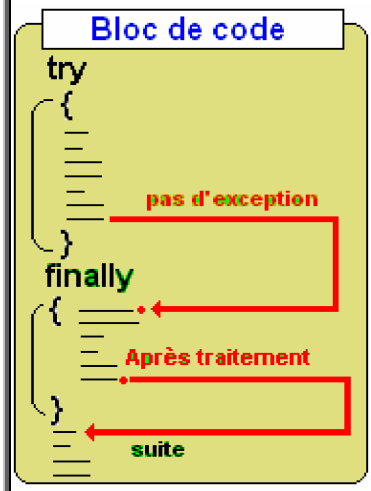
Début du programme.
...Avant incident
Interception ArithmeticExceptionPerso : Mauvais calcul !
Fin du programme.

L'exécution de ce programme est identique à celle du programme précédent, notre exception personnalisée fonctionne bien comme les exceptions prédéfinies de C#.

4. Clause finally

Supposons que nous soyons en présence d'un code contenant une éventuelle levée d'exception, mais supposons que quoiqu'il se passe nous désirions qu'un certain type d'action ait toujours lieu (comme par exemple fermer un fichier qui a été ouvert auparavant). Il existe en C# une **clause spécifique optionnelle** dans la syntaxe des gestionnaires d'exception permettant ce type de réaction du programme, c'est la clause **finally.** Voici en pseudo C#une syntaxe de cette clause :

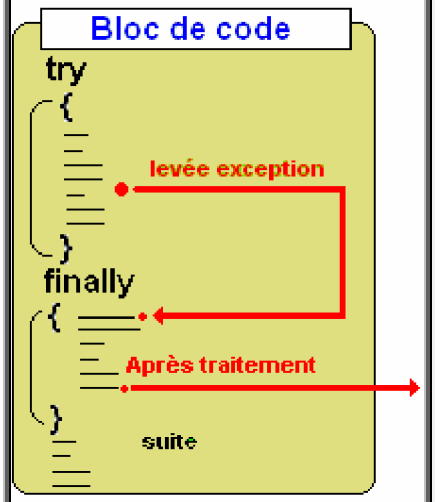
```
<Ouverture du fichier>
try {
    < action sur fichier>
}
finally {
   <fermeture du fichier>
}
.... suite
```



Fonctionnement sans incident

Si aucun incident ne se produit durant l'exécution du bloc < action sur fichier> :

l'exécution se poursuit à **l'intérieur du bloc finally par l'action** <fermeture du fichier> et **la suite** du code.



Fonctionnement si un incident se produit

Si un incident se produit durant l'exécution du bloc < action sur fichier> et qu'une exception est lancée:

malgré tout l'exécution se poursuivra à **l'intérieur du bloc finally par l'action** <fermeture du fichier>, puis **arrêtera** l'exécution du code dans le bloc.

La syntaxe C# autorise l'écriture d'une clause **finally** associée à plusieurs clauses **catch** :

```
try {
 <code à protéger>
}
catch (exception1 e ) { <traitement de l'exception1>  }
catch (exception2 e ) { <traitement de l'exception2>  }
...
finally {    <action toujours effectuée>  }
```

Remarque :

Si le code du bloc à protéger dans try...finally contient une instruction de rupture de séquence comme break, return ou continue, le code de la clause finally{...} est malgré tout exécuté

avant la rupture de séquence.

Nous avons vu lors des définitions des itérations **while**, **for** et de l'instruction **continue**, que l'équivalence suivante entre un **for** et un **while** valide dans le cas général, était mise en défaut si le corps d'instruction contenait un **continue** (instruction forçant l'arrêt d'un tours de boucle et relançant l'itération suivante) :

*Equivalence incorrecte si Instr contient un **continue** :*

for (**Expr1** ; **Expr2** ; **Expr3**) Instr	**Expr1** ; **while** (**Expr2**) { Instr ; **Expr3** }

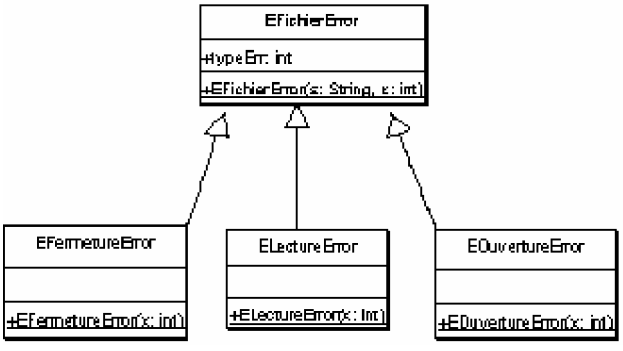
E*quivalence correcte même si Instr contient un **continue** :*

for (**Expr1** ; **Expr2** ; **Expr3**) Instr	**Expr1** ;
	while (**Expr2**)
	{
	try {
	Instr ;
	} **finally** { **Expr3**
	}
	}

5. Un exemple de traitement d'exceptions sur des fichiers

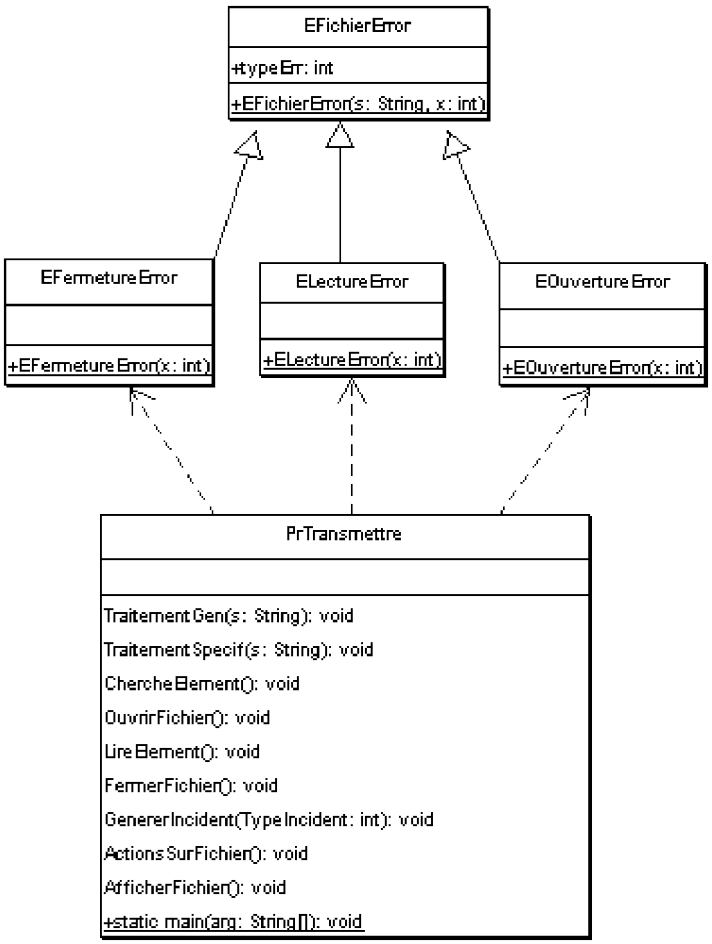
Créons une hiérarchie d'exceptions permettant de signaler des incidents sur des manipulations de fichiers.

Pour cela on distingue essentiellement trois catégories d'incidents qui sont représentés par trois classes :



Enoncé : Nous nous proposons de mettre en oeuvre les concepts précédents sur un exemple simulant un traitement de fichier. L'application est composée d'un bloc principal <programme> qui appelle une suite de blocs imbriqués.

Les classes en jeu et les blocs de programmes (méthodes) acteurs dans le traitement des exceptions sont figurées dans les diagrammes UML de droite :



<**programme**>

…<**ActionsSurFichier**>

…..<**AfficheFichier**>

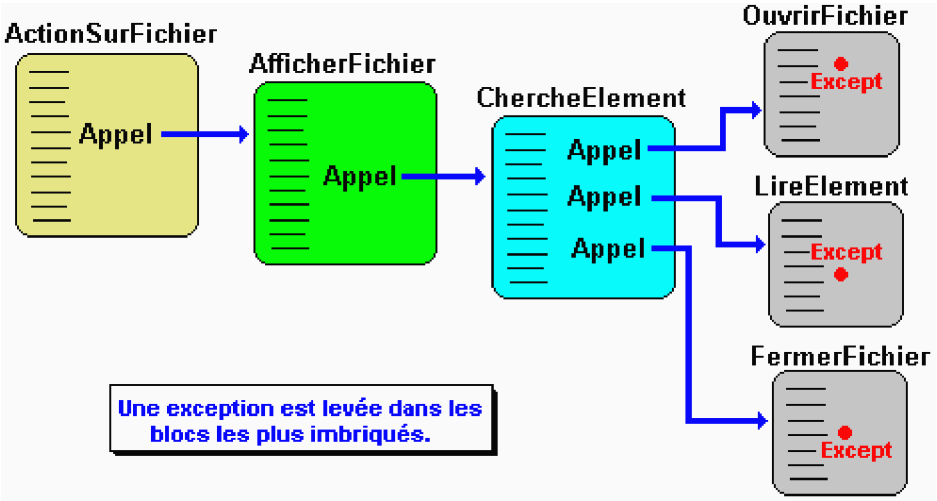
……..<**ChercheElement**>

.....<***OuvrirFichier***> **--> exception**

.....<***LireElement***> **--> exception**

…….....<***FermerFichier***> **--> exception**

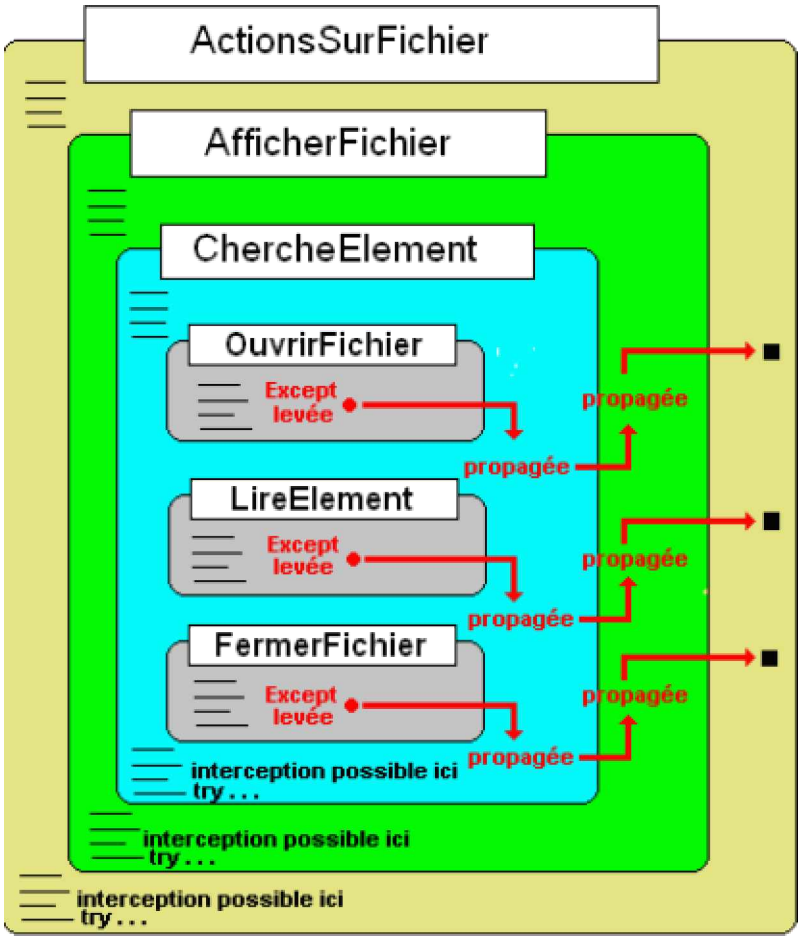
Les trois blocs du dernier niveau les plus internes <***OuvrirFichier***>, <***LireElement***> et <***FermerFichier***> peuvent lancer chacun une exception selon le schéma ci-après :



- ***La démarche***

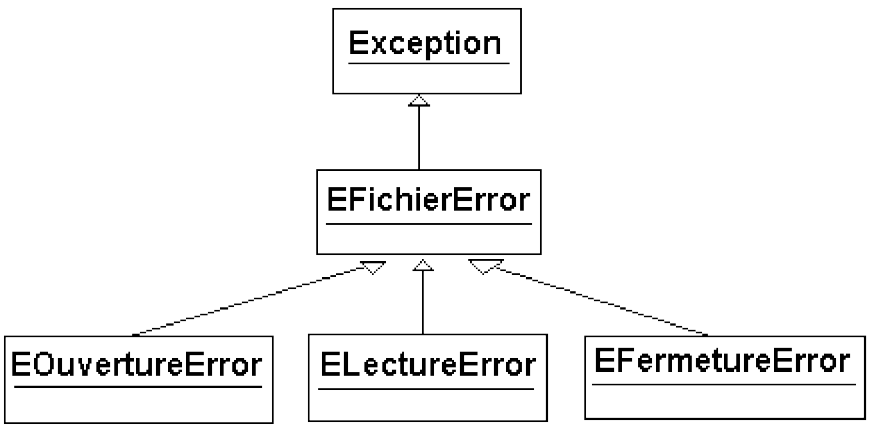
Les éventuelles exceptions lancées par les blocs <**OuvrirFichier**>,<**LireElement**> et <**FermerFichier**> doivent pouvoir se **propager** aux blocs de niveaux englobant afin d'être

interceptables à n'importe quel niveau.



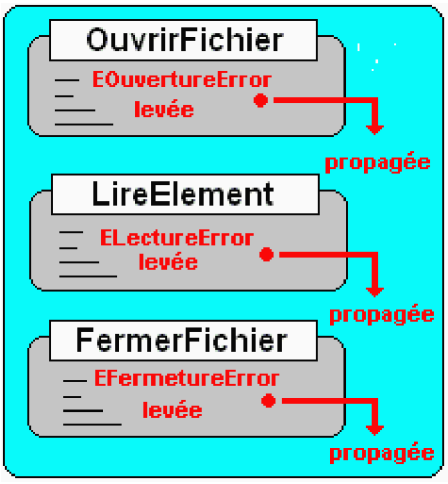
- ***Les classes d'exception***

On propose de créer une classe générale d'exception **EFichierError** héritant de la classe des **Exception**, puis 3 classes d'exception héritant de cette classe EFichierError :



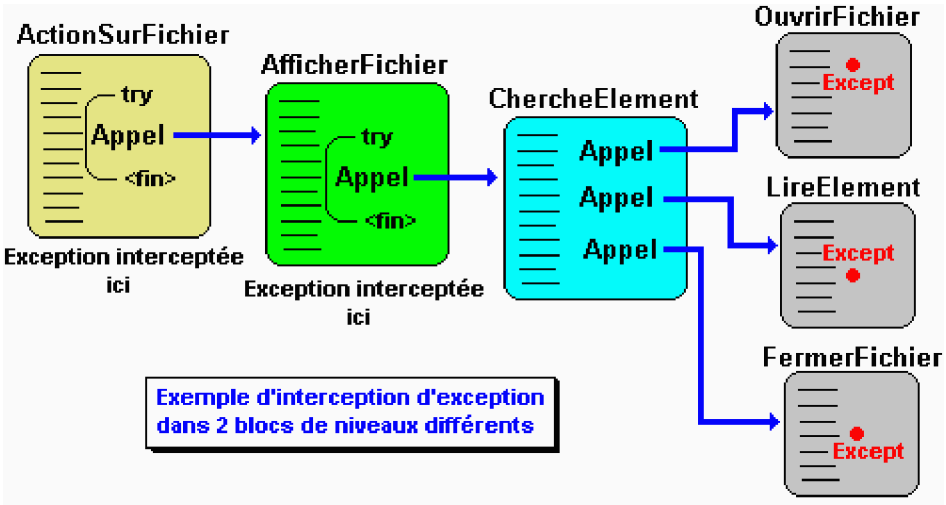
- ***Les blocs lançant éventuellement une exception***

Chaque bloc le plus interne peut lancer (lever) une exception de classe différente et la propage au niveau supérieur :



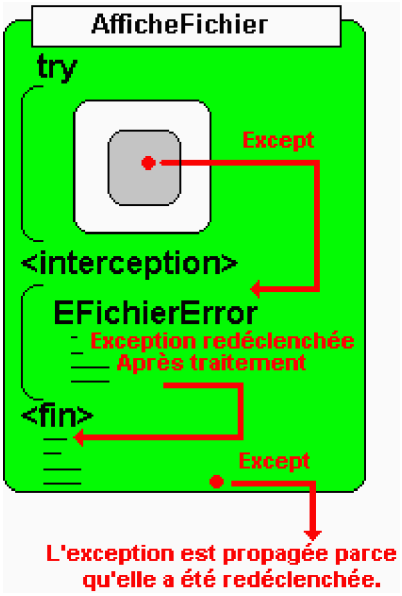
- ***Les blocs interceptant les exceptions***

Nous proposons par exemple d'intercepter les exceptions dans les deux blocs <**ActionsSurFichier**> et <**AfficheFichier**> :



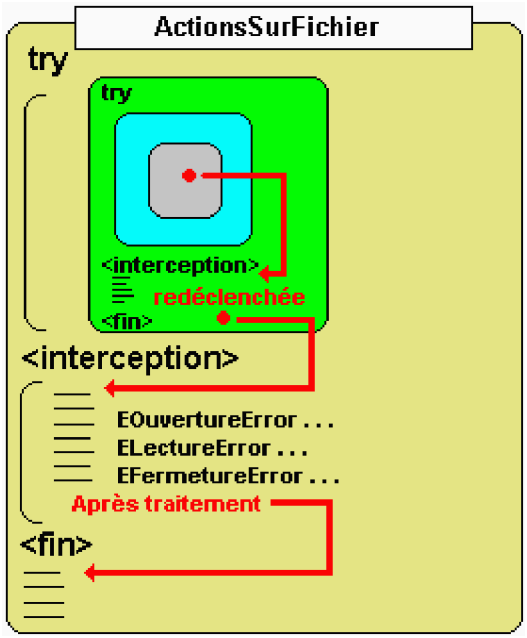
Le bloc <AfficherFichier>

Ce bloc interceptera une exception de type EFichierError, puis la redéclenchera après traitement :



Le bloc <ActionsSurFichier>

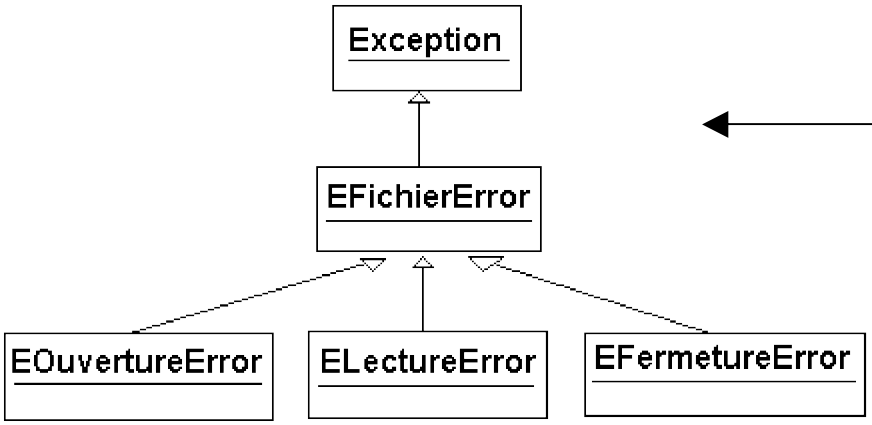
Ce bloc interceptera une exception de l'un des trois types EOuvertureError, ELectureError ou EFermetureError :



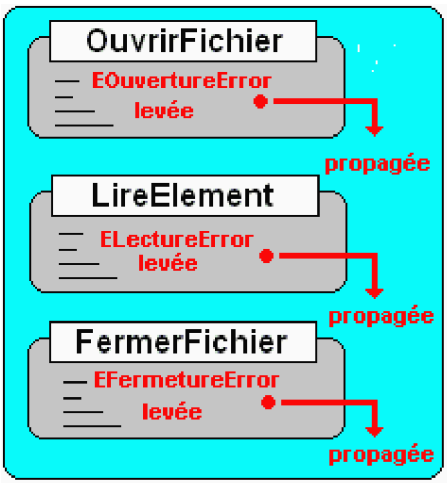
6. Une solution de l'exemple précédent en C#

```
using System ;
namespace PrTests
{
 /* pseudo-Traitement d'un fichier à plusieurs niveaux,
  * avec exception et relance d'exception
  */
 class EFichierError : Exception {
   public int typeErr ;
   public EFichierError ( String s, int x ): base ( s ) {
       typeErr = x ;
   }
```

```
}

class EOuvertureError : EFichierError {
  public EOuvertureError ( int x ): base ("Impossible d'ouvrir le fichier !" ,x ) { }
}

class ELectureError : EFichierError{
  public ELectureError ( int x ): base ("Impossible de lire le fichier !" ,x ) { }
}

class EFermetureError : EFichierError{
  public EFermetureError ( int x ): base ("Impossible de fermer le fichier !" ,x ) {
  }
}
```

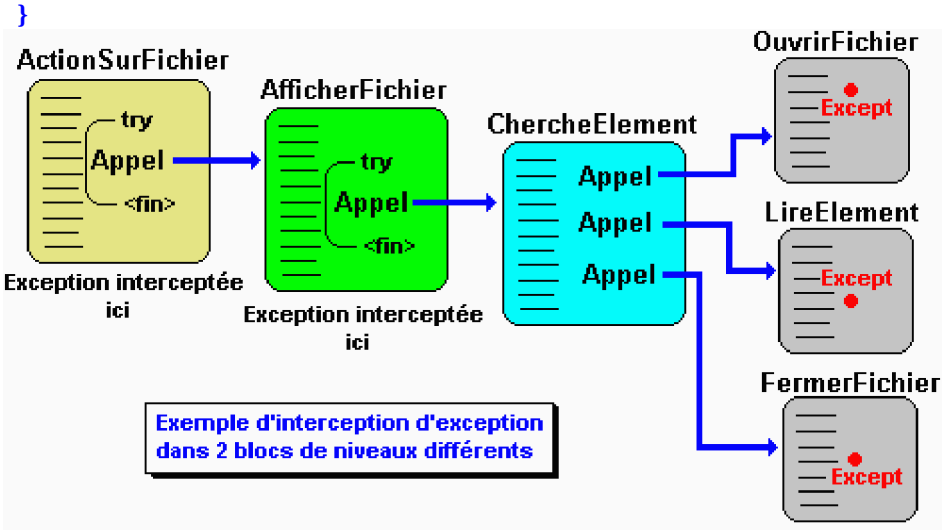



```
//-------------------------------------------------------------------------------------------------------------------

public class PrTransmettre {

  void TraitementGen ( String s ) {
    System .Console.WriteLine
("traitement general de l'erreur: " + s );
  }

  void TraitementSpecif ( String s ) {
    System .Console.WriteLine
("traitement specifique de l'erreur: " + s );
  }

  void OuvrirFichier ( ) {
    System .Console.WriteLine (" >> Action ouverture...");
    GenererIncident ( 1 );
    System .Console.WriteLine (" >> Fin ouverture.");
  }

void LireElement ( ) {
    System .Console.WriteLine (" >> Action lecture...");
    GenererIncident ( 2 );
    System .Console.WriteLine (" >> Fin lecture.");
  }

void FermerFichier ( ) {
    System .Console.WriteLine (" >> Action fermeture...");
    GenererIncident ( 3 );
    System .Console.WriteLine (" >> Fin fermeture.");
```



```
}
```

```
void ChercheElement ( ) {
   OuvrirFichier ( );
   LireElement ( );
   FermerFichier ( );
}
```

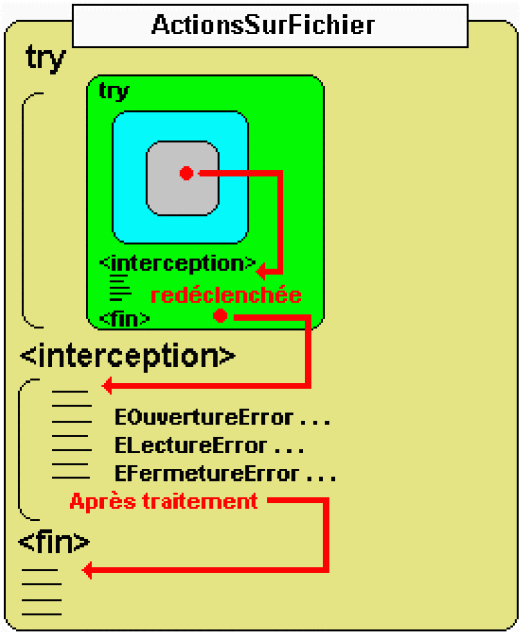



```
//-------------------------------------------------------------------------------------
```

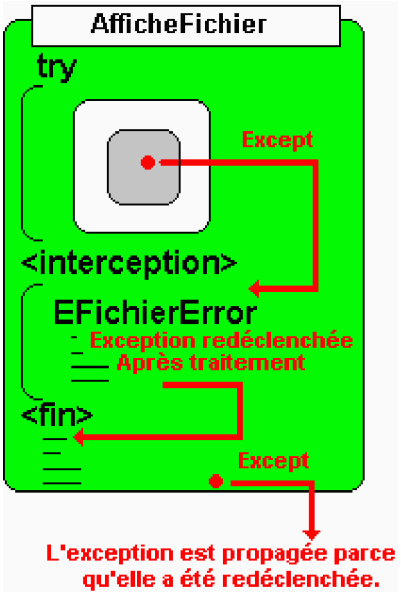
```
 void GenererIncident ( int TypeIncident )  {
  int n ;
  Random nbr = new Random ( );
 switch ( TypeIncident )
 {
  case 1 : n = nbr.Next ( ) % 4 ;
   if ( n == 0 )
      throw new EOuvertureError ( TypeIncident );
  break;
  case 2 : n = nbr.Next ( ) % 3 ;
   if ( n == 0 )
      throw new ELectureError ( TypeIncident );
  break;
  case 3 : n = nbr.Next ( ) % 2 ;
   if ( n == 0 )
      throw new EFermetureError ( TypeIncident );
  break;
 }
}
```

```
//-----------------------------------------------------
```

```
void  ActionsSurFichier ( ) {
  System .Console.WriteLine ("Debut du travail sur le fichier.");

 try
   {
     System .Console.WriteLine (".....");
     AfficherFichier ( );
   }


      catch( EOuvertureError E )
      {
        TraitementSpecif ( E.Message );
      }

      catch( ELectureError E )
      {
        TraitementSpecif ( E.Message );
      }

      catch( EFermetureError E )
      {
        TraitementSpecif ( E.Message );
      }
    System.Console.WriteLine ("Fin du travail sur le fichier.");
}
```




```
//-------------------------------------------------------------------
void  AfficherFichier ( )  {

  try    {
    ChercheElement ( );
   }
   catch( EFichierError E )   {
    TraitementGen ( E.Message );
     throw  E ;
   }
  }

  public static void  Main ( string [ ] arg ) {
     PrTransmettre Obj  =  new  PrTransmettre ( );
    try    {
      Obj.ActionsSurFichier ( );
     }
     catch( EFichierError E )   {
      System .Console.WriteLine ( " Autre  type  d'Erreur
  générale Fichier !");
     }
      System .Console.ReadLine ( );
   }
  }
 }
```



Processus et multi-threading

Plan général:

1. Rappels

1.1 La multiprogrammation
1.2 Multitâche et processus
1.3 Multi-threading et processus

2. C# autorise l'utilisation des processus et des threads

2.1 Les processus avec C#
2.2 Comment exécuter une application à partir d'une autre application
2.3 Comment atteindre un processus déjà lancé
2.4 Comment arrêter un processus

3. C# et les threads

3.1 Comment créer un Thread
3.2 Comment endormir, arrêter ou interrompre un Thread
3.3 Exclusion mutuelle, concurrence, section critique, et synchronisation
3.4 Section critique en C# : lock
3.5 Section critique en C# : Monitor
3.6 Synchronisation commune aux processus et aux threads

1. Rappels

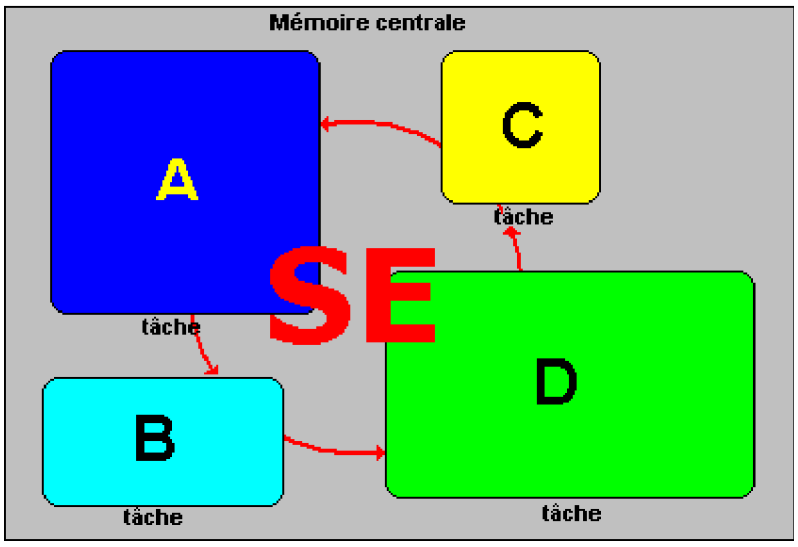
1.1 La multiprogrammation

Nous savons que les ordinateurs fondés sur les principes d'une machine de Von Neumann, sont des machines séquentielles donc n'exécutant qu'une seule tâche à la fois. Toutefois, le gaspillage de temps engendré par cette manière d'utiliser un ordinateur (le processeur central passe l'écrasante majorité de son temps à attendre) a très vite été endigué par l'invention de systèmes d'exploitations de multiprogrammation ou multitâches, permettant l'exécution "simultanée" de plusieurs tâches.

Dans un tel système, les différentes tâches sont exécutées sur une machine disposant d'**un seul processeur**, en apparence **en même temps** ou encore en **parallèle,** en réalité elles sont exécutées séquentiellement chacune à leur tour, ceci ayant lieu tellement vite pour notre conscience que nous avons l'impression que les programmes s'exécutent simultanément. Rappelons ici qu'une tâche est une application comme un traitement de texte, un navigateur Internet, un jeu,... ou d'autres programmes spécifiques au système d'exploitation que celui-ci exécute.

1.2 Multitâche et processus

Le noyau du système d'exploitation SE, conserve en permanence le contrôle du temps d'exécution en distribuant cycliquement des tranches de temps (time-slicing) à chacune des applications A, B, C et D figurées ci-dessous. Dans cette éventualité, une application représente dans le système un **processus** :

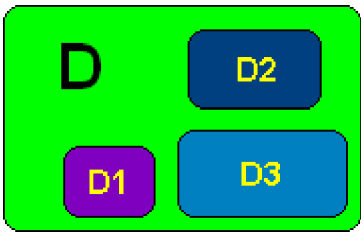


Rappelons la définition des **processus** donnée par A.Tanenbaum: un programme qui s'exécute et qui possède **son propre espace mémoire** : ses registres, ses piles, ses variables et son propre processeur virtuel (simulé en multiprogrammation par la commutation entre processus effectuée par le processeur unique).

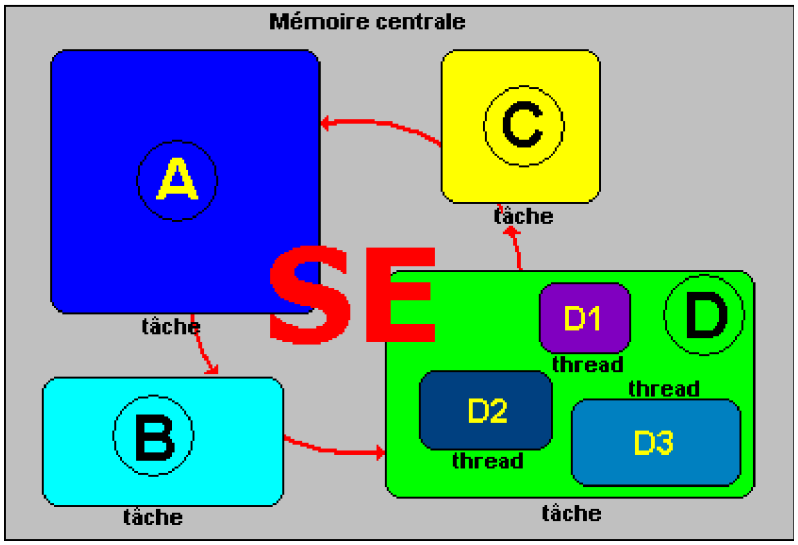
Thread
En fait, chaque processus peut lui-même fonctionner comme le système d'exploitation en lançant des sous-tâches internes au processus et par là même reproduire le fonctionnement

de la multiprogrammation. Ces sous-tâches sont nommées "flux d'exécution" ou **Threads.**

Ci-dessous nous supposons que l'application D exécute en même temps les 3 Threads D1, D2 et D3 :

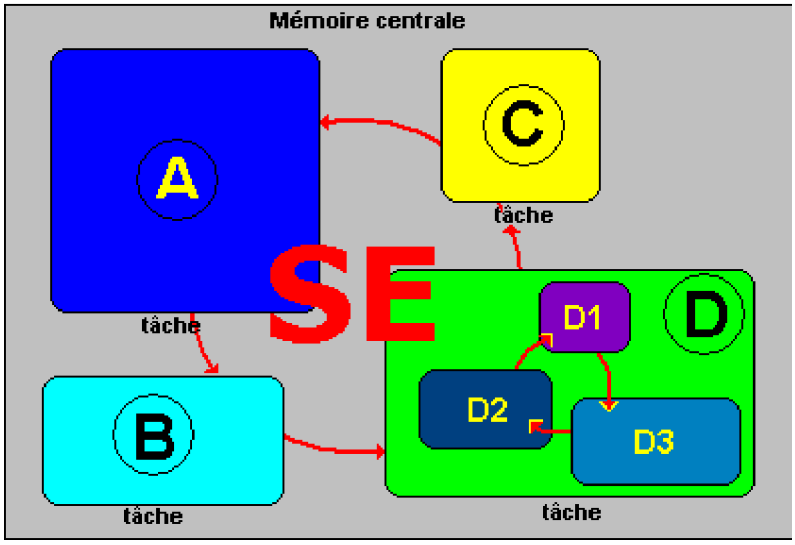


Reprenons l'exemple d'exécution précédent, dans lequel 4 processus s'exécutent "en même temps" et incluons notre processus D possédant 3 flux d'exécutions (threads) :



La commutation entre les threads d'un processus fonctionne de la même façon que la commutation entre les processus, chaque thread se voit alloué cycliquement, lorsque le processus D est exécuté une petite tranche de temps.

Le partage et la répartition du temps sont effectués **uniquement** par le système d'exploitation :



1.3 Multithreading et processus

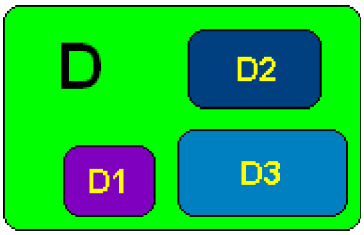
Définition :

La majorité des systèmes d'exploitation (Windows, Linux, Solaris, MacOs,...) supportent l'utilisation d'application contenant des threads, l'on désigne cette fonctionnalité sous le nom de **Multi-threading.**

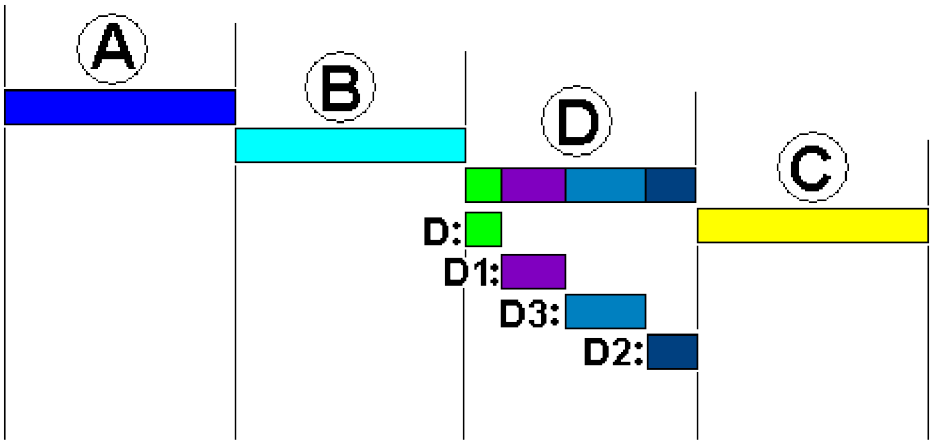
Différences entre threads et processus :

- Communication entre threads **plus rapide** que la communication entre processus,
- Les threads partagent un **même espace de mémoire** (de travail) entre eux,
- Les processus ont chacun un **espace mémoire personnel**.

Dans l'exemple précédent, figurons les processus A, B, C et le processus D avec ses threads dans un graphique représentant une tranche de temps d'exécution allouée par le système et supposée être la même pour chaque processus.

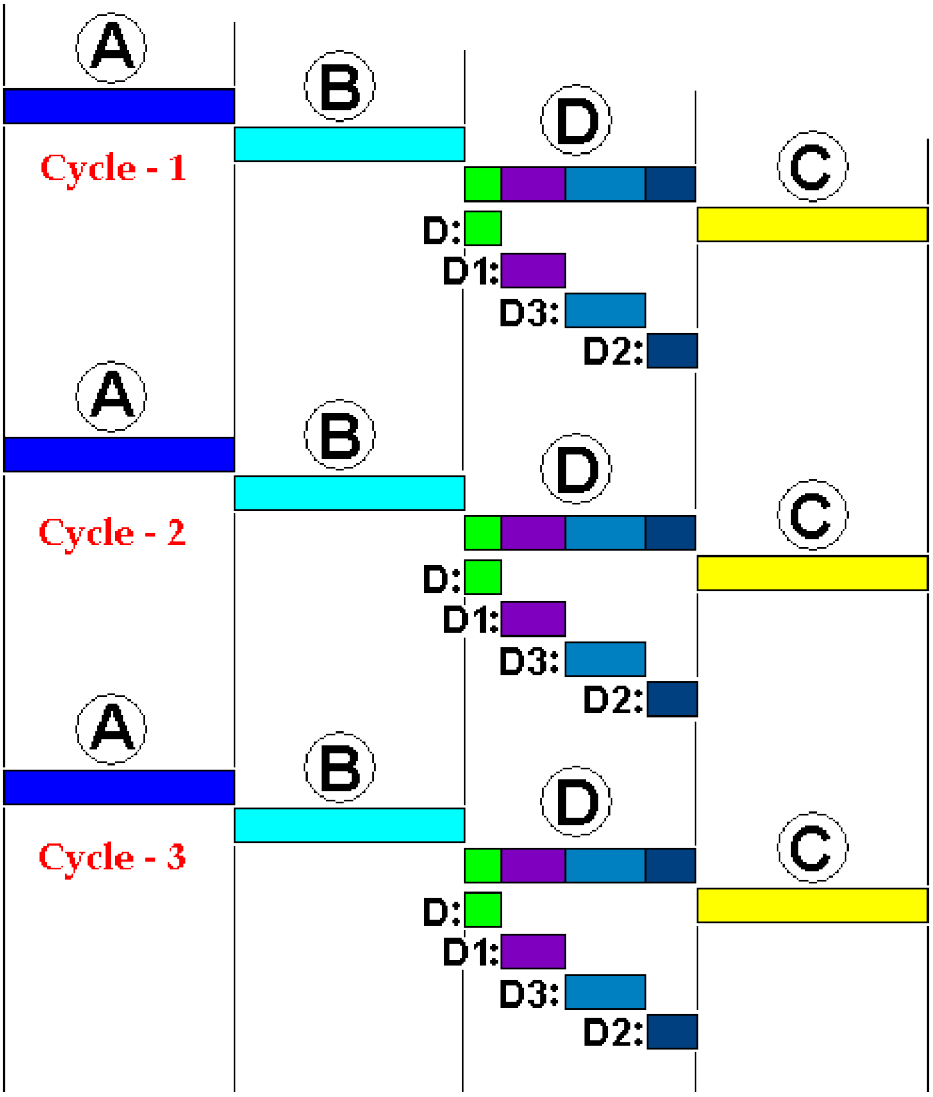


Le système ayant alloué le même temps d''exécution à chaque processus, lorsque par exemple le tour vient au processus D de s'exécuter dans sa tranche de temps, il exécutera une petite sous-tranche pour D1, pour D2, pour D3 et attendra le prochain cycle. Ci-dessous un cycle d'exécution :



Tranches de temps allouées pendant l'exécution

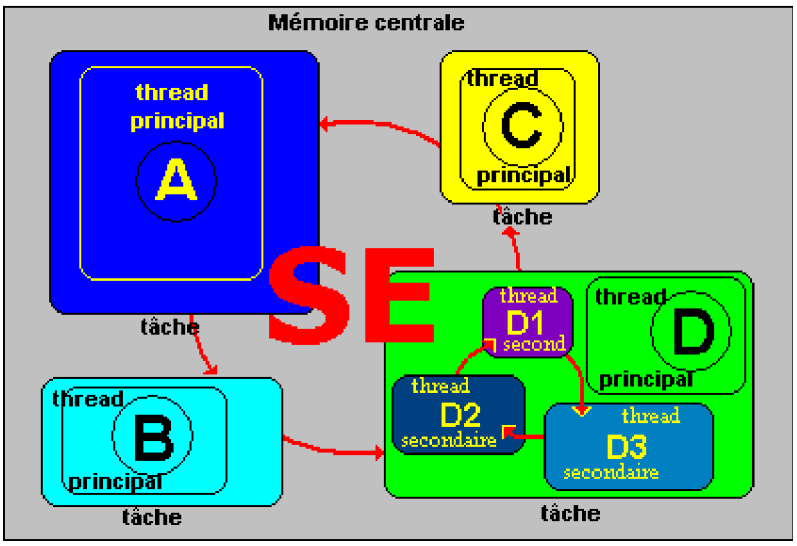
Voici sous les mêmes hypothèses de temps égal d'exécution alloué à chaque processus, le comportement de l'exécution sur 3 cycles consécutifs :



Le langages C# dispose de classes permettant d'écrire et d'utiliser des threads dans vos applications.

2. C# autorise l'utilisation des processus et des threads

Lorsqu'un programme C# s'exécute en dehors d'une programmation de multi-threading, le processus associé comporte automatiquement un thread appelé **thread principal**. Un autre thread utilisé dans une application s'appelle un thread secondaire. Supposons que les quatre applications (ou tâches) précédentes A, B, C et D soient toutes des applications C#, et que D soit celle qui comporte trois threads secondaires D1, D2 et D3 "parallèlement" exécutés :



2.1 Les processus avec C#

Il existe dans .Net Framework une classe nommée Process dans l'espace de nom System.Diagnostics, permettant d'accéder à des processus locaux à la machine ou distants et de les manipuler (démarrer, surveiller, stopper, …). Cette classe est bien documentée par la bibliothèque MSDN de Microsoft, nous allons dans ce paragraphe montrer comment l'utiliser à partir d'un cas pratique souvent rencontré par le débutant : comment lancer une autre application à partir d'une application déjà en cours d'exécution.

2.2 Comment exécuter une application à partir d'une autre

1°) Instancier un objet de classe Process :

```
Process AppliAexecuter = new Process( );
```

2°) Paramétrer les informations de lancement de l'application à exécuter :
Parmi les nombreuses propriétés de la classe Process la propriété StartInfo (**public** ProcessStartInfo StartInfo {**get**; **set**;}) est incontournable, car elle permet ce paramétrage. Supposons que notre application se nomme "Autreappli.exe" qu'elle soit située sur le disque C: dans le dossier "Travail", qu'elle nécessite au démarrage comme paramètre le nom d'un fichier contenant des données, par exemple le fichier "donnees.txt" situé dans le dossier "C:\infos". La propriété StartInfo s'utilise alors comme suit afin de préparer le lancement de l'application :

```
AppliAexecuter.StartInfo.FileName ="c:\\Travail\\Autreappli.exe";
AppliAexecuter.StartInfo.UseShellExecute = false;
AppliAexecuter.StartInfo.RedirectStandardOutput = false;
Appliexec.StartInfo.Arguments ="c:\\infos\\donnees.txt";
```

Remarques :

- UseShellExecute = **false**: permet de lancer directement l'application sans avoir à utiliser l'interface shell du système d'exploitation.

- RedirectStandardOutput = **false**: la sortie standard du processus reste dirigée vers l'écran par défaut.

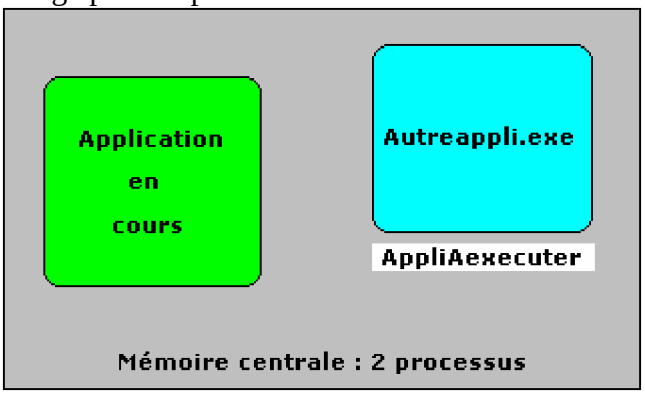
3°) Démarrer le processus par exemple par la surcharge d'instance de la méthode Start( ) de la classe Process :

```
AppliAexecuter.Start( );
```

Code récapitulatif de la méthode Main pour lancer un nouveau processus dans une application en cours d'exécution :

```
public static void Main( ) {
        Process AppliAexecuter = new Process( );
        AppliAexecuter.StartInfo.FileName ="c:\\Travail\\Autreappli.exe";
        AppliAexecuter.StartInfo.UseShellExecute = false;
        AppliAexecuter.StartInfo.RedirectStandardOutput = false;
        Appliexec.StartInfo.Arguments ="c:\\infos\\donnees.txt";
        AppliAexecuter.Start( );
}
```

Image produit par ce code dans la mémoire centrale :



2.3 Comment atteindre un processus déjà lancé

Atteindre le processus courant de l'application appelante :

```
Process Courant = Process.GetCurrentProcess();
```

Atteindre toutes les instances de processus d'une application nommée "Autreappli.exe" lancée plusieurs fois grâce à son nom sur l'ordinateur local :

```
Process [] localProcess = Process.GetProcessesByName("Autreappli");
```

Atteindre toutes les instances de processus d'une application nommée "Autreappli.exe" lancée plusieurs fois grâce à son nom sur un ordinateur distant de nom "ComputerDistant" :

```
Process [] localProcess =
Process.GetProcessesByName("Autreappli","ComputerDistant");
```

Atteindre toutes les instances de processus d'une application nommée "Autreappli.exe" lancée plusieurs fois grâce à son nom sur un ordinateur distant d'adresse IP connue par exemple "101.22.34.18":

```
Process [] localProcess =
Process.GetProcessesByName("Autreappli","101.22.34.18");
```

Atteindre un processus grâce à l'identificateur unique sur un ordinateur local de ce processus par exemple 9875 :

```
Process localProcess = Process.GetProcessById(9875);
```

Atteindre un processus grâce à l'identificateur unique de ce processus par exemple 9875 sur un ordinateur distant de nom "ComputerDistant" :

```
Process localProcess = Process.GetProcessById(9875, "ComputerDistant");
```

Atteindre un processus grâce à l'identificateur unique de ce processus par exemple 9875 sur un ordinateur distant d'adresse IP connue par exemple "101.22.34.18" :

```
Process localProcess = Process.GetProcessById(9875, "101.22.34.18");
```

2.4 Comment arrêter un processus

*Seuls les processus **locaux** peuvent être stoppés dans .Net Framework, les processus distants ne peuvent qu'être surveillés.*

Si le processus est une application fenêtrée (application possédant une interface IHM)

La méthode "**public bool** CloseMainWindow( )" est la méthode à employer

```
Process AppliAexecuter = new Process( );
….
AppliAexecuter.CloseMainWindow( );
```

Cette méthode renvoie un booléen qui indique :

- **True** si la fermeture a été correctement envoyée et le processus est stoppé.
- **False** si le processus n'est pas stoppé soit parcequ'il y a eu un incident, soit parce que le processus n'était une application fenêtrée (application console).

Si le processus n'est pas une application fenêtrée (application console)

La méthode "**public void** Kill( )" est la méthode à employer

```
Process AppliAexecuter = new Process( );
….
AppliAexecuter.Kill( );
```

Ou bien :

```
if( !AppliAexecuter.CloseMainWindow( ) )
        AppliAexecuter.Kill( );
```

3. C# et les threads

Comme l'avons déjà signalé tous les systèmes d'exploitation modernes permettent la programmation en multi-threading, le .Net Framework contient une classe réservée à cet usage dans l'espace de noms System.Threading : la classe non héritable Thread qui implémente l'interface _Thread.

```
public sealed class Thread: _Thread
```

Cette classe Thread sert à créer, contrôler, et modifier les priorités de threads.

3.1 Comment créer un Thread

Le C# 2.0 propose quatre surcharges du constructeur de Thread, toutes utilisent la notion de **delegate** pour préciser le code à exécuter dans le thread, nous examinons celle qui est la plus utilisée depuis la version 1.0.

```
public Thread ( ThreadStart start );
```

Le paramètre start de type ThreadStart est un objet **delegate** sans paramètre qui pointe sur (fait référence à) la méthode à appeler à chaque exécution du thread ainsi créé :

```
public delegate void ThreadStart ( );
```

Vous devez donc écrire une méthode de classe ou d'instance ayant la même signature que le delegate **void** ThreadStart ( ), puis créer l'objet **delegate** qui pointera vers cette méthode. Vous pouvez nommer cette méthode du nom que vous voulez, pour rester dans le style Java nous la dénommerons **run( )**. Ci-dessous un pseudo-code C# de création d'un thread à partir d'un délégué pointant sur une méthode de classe :

```
public class ChargerDonnees    {
  public static void run( ){  ….  }
}

public class AppliPrincipale    {
  public static void Main( ){
    ThreadStart ChargerDelegate = new ThreadStart (ChargerDonnees.run);
     Thread thrdChargement = new Thread(ChargerDelegate);
  }
}
```

Notons qu'il est possible d'alléger le code d'instanciation du thread en créant un objet délégué anonyme qui est passé en paramètre au constructeur de Thread :

```
Thread thrdChargement = new Thread( new ThreadStart (ChargerDonnees.run) );
```

Dans l'exemple qui suit nous lançons trois threads en plus du thread principal automatiquement construit par le CLR pour chaque processus, l'un à partir d'une méthode **static** run0 de la classe Program et les deux autres à partir d'une méthode **static** run1 et d'une méthode d'instance run2 de la classe AfficherDonnees :

```
public class AfficherDonnees
   {
      public static void run1()
      {
         for (int i1 = 1; i1 < 100; i1++)
            System.Console.WriteLine(">>>  thread1 = " + i1);
      }
      public void run2()
      {
         for (int i2 = 1; i2 < 100; i2++)
            System.Console.WriteLine("***  thread2 = " + i2);
      }
   }
   public class Program
   {
      public static void run0()
      {
         for (int i0 = 1; i0 < 100; i0++)
            System.Console.WriteLine(".... thread0 = " + i0);
      }
      static void Main(string[] args)
      {
         Console.WriteLine("Lancement des threads :");
         AfficherDonnees obj = new AfficherDonnees();
         Thread thread0 = new Thread(new ThreadStart( run0 ));
         Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
         Thread thread2 = new Thread(new ThreadStart(obj.run2));
         thread0.Start();
         thread1.Start();
         thread2.Start();
         for (int i = 1; i < 100; i++)
            System.Console.WriteLine(" i = " + i);
         System.Console.WriteLine("fin de tous les threads.");
      }
   }
```

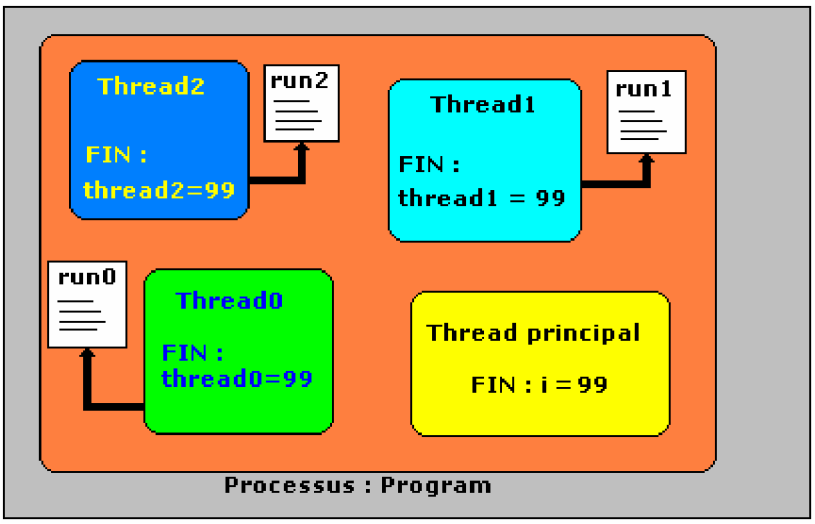
Résultats de l'exécution du programme précédent (dépendants de votre configuration machine+OS) :

```
Lancement des threads :
....    thread0 = 1
>>>  thread1 = 1
 i = 1
***   thread2 = 1
....    thread0 = 2
>>>  thread1 = 2
 i = 2
***  thread2 = 2
....   thread0 = 3
…….
i = 98
***   thread2 = 98
....    thread0 = 99
>>>  thread1 = 99
 i = 99
***   thread2 = 99
fin de tous les threads.
```

L'exécution précédente montre bien que chacun des thread0, thread1 et thread2 se voient allouer une tranche de temps pendant laquelle chacun d'eux exécute une partie de sa boucle **for** et imprime ses informations sur la console. Chaque thread se termine lorsque le code de la

méthode run vers laquelle il pointe a fini de s'exécuter.

Nous figurons ci-après l'image de la mémoire centrale pour le processus Program avec chacun des 3 threads instanciés pointant vers la méthode qu'il exécute :



3.2 Comment endormir, arrêter ou interrompre un Thread

La société Microsoft a déprécié depuis la version 2.0 du .Net Framework les méthodes Suspend() et Resume() qui permettaient d'effectuer la synchronisation entre les threads, toutefois elles n'ont pas été supprimées. En ce sens Microsoft adopte la même attitude que Sun pour Java, afin de ne pas encourager les développeurs à utiliser des outils qui se sont montrés sensibles aux blocages du type verrou mortel. Nous ne proposerons donc pas ces méthodes dangereuses au lecteur, mais plutôt des méthodes sûres.

Endormir : Méthode Sleep (…)

On parle d'endormir un thread pendant un certain temps **t**, lorsque l'exécution de ce thread est arrêtée pendant ce temps **t**, c'est à dire que le thread est retiré de la file d'attente de l'algorithme d'ordonnancement du système. A la fin du temps **t**, le thread est automatiquement "réveillé" par le système, c'est à dire qu'il est replacé dans la file d'attente de l'ordonnanceur et donc son exécution repart.

La méthode "**public static void** Sleep(**int** millisecondsTimeout)" sert à endormir un thread pendant un temps exprimé en millisecondes. Dans la méthode run0( ) de l'exemple précédent si nous rajoutons l'instruction "Thread.Sleep(2);", nous "ralentissons" l'exécution de la boucle, puisque à tous les tours de boucles nous bloquons le thread qui l'exécute pendant 2 ms.

```
public static void run0( )
{
   for (int i0 = 1; i0 < 100; i0++)
   {
      System.Console.WriteLine(".... thread0 = " + i0);
      Thread.Sleep(2);
   }
}
```

Résultats de l'exécution du programme précédent avec Thread.Sleep(2) dans la méthode run0( ) ***:***

Lancement des threads : **.... thread0 = 1** **>>> thread1 = 1** **i = 1** ***** thread2 = 1** **……** **i = 98** ***** thread2 = 98** **.... thread0 = 79** **>>> thread1 = 99** **i = 99** ***** thread2 = 99** **.... thread0 = 80** **fin de tous les threads.** **.... thread0 = 81** **.... thread0 = 82** **.... thread0 = 83** **.... thread0 = 84**	**.... thread0 = 85** **.... thread0 = 86** **.... thread0 = 87** **.... thread0 = 88** **.... thread0 = 89** **.... thread0 = 90** **.... thread0 = 91** **.... thread0 = 92** **.... thread0 = 93** **.... thread0 = 94** **.... thread0 = 95** **.... thread0 = 96** **.... thread0 = 97** **.... thread0 = 98** **.... thread0 = 99** ***(résultats dépendants de votre configuration machine+OS)***

Notons que cette exécution est semblable à la précédente, du moins au départ, car nous constatons vers la fin que le thread0 a pris un léger retard sur ses collègues puisque le thread1 et le thread2 se termine avec la valeur 99, le thread principal affiche la phrase "**fin de tous les threads**" alors que le thread0 n'a pas encore dépassé la valeur 80.

Les trois threads thread1, thread2 et le thread principal sont en fait terminés seul le thread0 continue sont exécution jusqu'à la valeur 99 qui clôt son activité.

Arrêter : Méthode Abort (…)

Dans l'exemple précédent le message "**fin de tous les threads**" n'est pas conforme à la réalité puisque le programme est bien arrivé à la fin du thread principal, mais thread0 continue son exécution. Il est possible de demander au système d'arrêter définitivement l'exécution d'un thread, cette demande est introduite par la méthode d'instance **Abort** de la classe Thread qui lance le processus d'arrêt du thread qui l'appelle.

Nous reprenons le programme précédent dans lequel nous lançons une demande d'arrêt du thread0 par l'instruction : thread0.Abort();

```
public class AfficherDonnees
  {
    public static void run1()
    {
      for (int i1 = 1; i1 < 100; i1++)
        System.Console.WriteLine(">>>  thread1 = " + i1);
    }
    public void run2()
    {
      for (int i2 = 1; i2 < 100; i2++)
        System.Console.WriteLine("***  thread2 = " + i2);
    }
  }
  public class Program
  {
    public static void run0()
```

```
    {
      for (int i0 = 1; i0 < 100; i0++)
         System.Console.WriteLine(".... thread0 = " + i0);
         Thread.Sleep(2);
    }
    static void Main(string[] args)
    {
      Console.WriteLine("Lancement des threads :");
      AfficherDonnees obj = new AfficherDonnees( );
      Thread thread0 = new Thread(new ThreadStart( run0 ));
      Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
      Thread thread2 = new Thread(new ThreadStart(obj.run2));
      thread0.Start( );
      thread1.Start( );
      thread2.Start( );
      for (int i = 1; i < 100; i++)
         System.Console.WriteLine(" i = " + i);
      thread0.Abort();
      System.Console.WriteLine("fin de tous les threads.");
    }
  }
```

Résultats de l'exécution du programme précédent avec Thread.Sleep(2) et thread0.Abort() ***:***

Lancement des threads :
.... thread0 = 1
>>> thread1 = 1
i = 1
***** thread2 = 1**
…….
i = 98
***** thread2 = 98**
.... thread0 = 79
>>> thread1 = 99
i = 99
***** thread2 = 99**
.... thread0 = 80
fin de tous les threads. ***(Résultats dépendants de votre configuration machine+OS)***

Le thread0 a bien été arrêté avant sa fin normale, avec comme dernière valeur 80 pour l'indice de la boucle **for**.

Attendre : Méthode Join (…)

Un thread peut se trouver dans des états d'exécution différents selon qu'il est actuellement en cours d'exécution, qu'il attend, qu'il est arrêté etc… Il existe en C# un type énuméré ThreadState qui liste toutes les valeurs possibles des états d'un thread :

public enum ThreadState { Running = 0, StopRequested = 1, SuspendRequested = 2, Background = 4, Unstarted = 8, Stopped = 16, WaitSleepJoin = 32, Suspended = 64, AbortRequested = 128, Aborted = 256 }

Dans la classe Thread, nous trouvons une propriété "**public** ThreadState ThreadState {**get**;}" en lecture seule qui fournit pour un thread donné son état d'exécution. En consultant cette propriété le développeur peut connaître l'état "en direct" du thread, notons que cet état peut varier au cours du temps.

On peut faire attendre la fin d'exécution complète d'un thread pour qu'un autre puisse continuer son exécution, cette attente est lancée par l'une des trois surcharges de la méthode d'instance **Join** de la classe Thread :

> **public void** Join();

Dans le programme précédent rappelons-nous que le thread0 prend du "retard" sur les autres car nous l'avons ralenti avec un Sleep(2). Au lieu de l'arrêter définitivement avant la dernière instruction de la méthode Main remplaçons l'instruction "thread0.Abort();"par l'instruction "thread0.Join();".

Que se passe-t-il : Le thread principal qui exécute la méthode main, invoque la méthode **Join** du thread0 avant de terminer son exécution, ce qui signifie que le thread principal bloque tant que le thread0 n'a pas fini complètement son exécution. Nous ajoutons au programme précédent une méthode **public static void** etatsThreads qui affiche l'état d'exécution du thread principal et des 3 threads instanciés :

```
public static void etatsThreads(Thread principal, Thread thrd1, Thread thrd2, Thread
thrd3 )
{
        System.Console.WriteLine(principal.Name + " : " + principal.ThreadState);
        System.Console.WriteLine(thrd1.Name + " : " + thrd1.ThreadState);
        System.Console.WriteLine(thrd2.Name + " : " + thrd2.ThreadState);
        System.Console.WriteLine(thrd3.Name + " : " + thrd3.ThreadState);
}

static void Main(string[] args)
{
        Console.WriteLine("Lancement des threads :");
        AfficherDonnees obj = new AfficherDonnees();
        Thread principal = Thread.CurrentThread;
        principal.Name = "principal";
        Thread thread0 = new Thread(new ThreadStart(run0));
        thread0.Name = "thread0";
        Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
        thread1.Name = "thread1";
        Thread thread2 = new Thread(new ThreadStart(obj.run2));
        thread2.Name = "thread2";
        etatsThreads(principal, thread0, thread1, thread2);
        thread0.Start();
        thread1.Start();
        thread2.Start();
        etatsThreads(principal, thread0, thread1, thread2);
        for (int i = 1; i < 100; i++)
        {
           System.Console.WriteLine(" i = " + i);
        }
        etatsThreads(principal, thread0, thread1, thread2);
        thread0.Join();
        etatsThreads(principal, thread0, thread1, thread2);
        System.Console.WriteLine("fin de tous les threads.");
    }
```

Nous obtenons une référence sur le thread principal par la propriété CurrentThread dans l'instruction : Thread principal = Thread.CurrentThread.

Résultats de l'exécution du programme précédent avec Thread.Sleep(2) et thread0.Join() :

Lancement des threads **principal : Running** **thread0 : Unstarted** **thread1 : Unstarted** **thread2 : Unstarted** **.... thread0 = 1** **>>> thread1 = 1** **principal : Running** ***** thread2 = 1** **>>> thread1 = 2** **.... thread0 = 2** **thread0 : WaitSleepJoin** ***** thread2 = 2** **>>> thread1 = 3** **.... thread0 = 3** **thread1 : WaitSleepJoin** ***** thread2 = 3** **>>> thread1 = 4** **thread2 : WaitSleepJoin** **.... thread0 = 4** ***** thread2 = 4** **>>> thread1 = 5** **i = 1** **…….**	**i = 99** **principal : Running** **.... thread0 = 85** **thread0 : WaitSleepJoin** **thread1 : Stopped** **.... thread0 = 86** **thread2 : Stopped** **.... thread0 = 87** **.... thread0 = 88** **.... thread0 = 89** **.... thread0 = 90** **.... thread0 = 91** **.... thread0 = 92** **.... thread0 = 93** **.... thread0 = 94** **.... thread0 = 95** **.... thread0 = 96** **.... thread0 = 97** **.... thread0 = 98** **.... thread0 = 99** **principal : Running** **thread0 : Stopped** **thread1 : Stopped** **thread2 : Stopped** **fin de tous les threads.**

(Résultats dépendants de votre configuration machine+OS)

Au début de l'exécution les états sont :

principal : Running
thread0 : Unstarted
thread1 : Unstarted
thread2 : Unstarted

Seul le thread principal est en état d'exécution, les 3 autres nonencore démarrés.

Après invocation de la méthode **Start** de chaque thread, la tour revient au thread principal pour exécuter la boucle **for** (**int** i = 1; i < 100; i++), les 3 autres threads sont dans la file d'attente :

principal : Running
thread0 : WaitSleepJoin
thread1 : WaitSleepJoin
thread2 : WaitSleepJoin

Après la fin de l'exécution de la boucle **for** (**int** i = 1; i < 100; i++) du thread principal celui-ci est sur le point de s'arrêter, les thread1 et thread2 ont fini leur exécution, le thread0 continue son exécution car ralenti par le Sleep(2) à chaque tour de boucle :

principal : Running
thread0 : WaitSleepJoin
thread1 : Stopped
thread2 : Stopped

Après la terminaison du décompte de la boucle du thread0 jusqu'à la valeur 99, le thread0 se termine et le thread principal est sur le point de se terminer (il ne lui reste plus la dernière instruction d'affichage " System.Console.WriteLine("fin de tous les threads.")" à exécuter) :

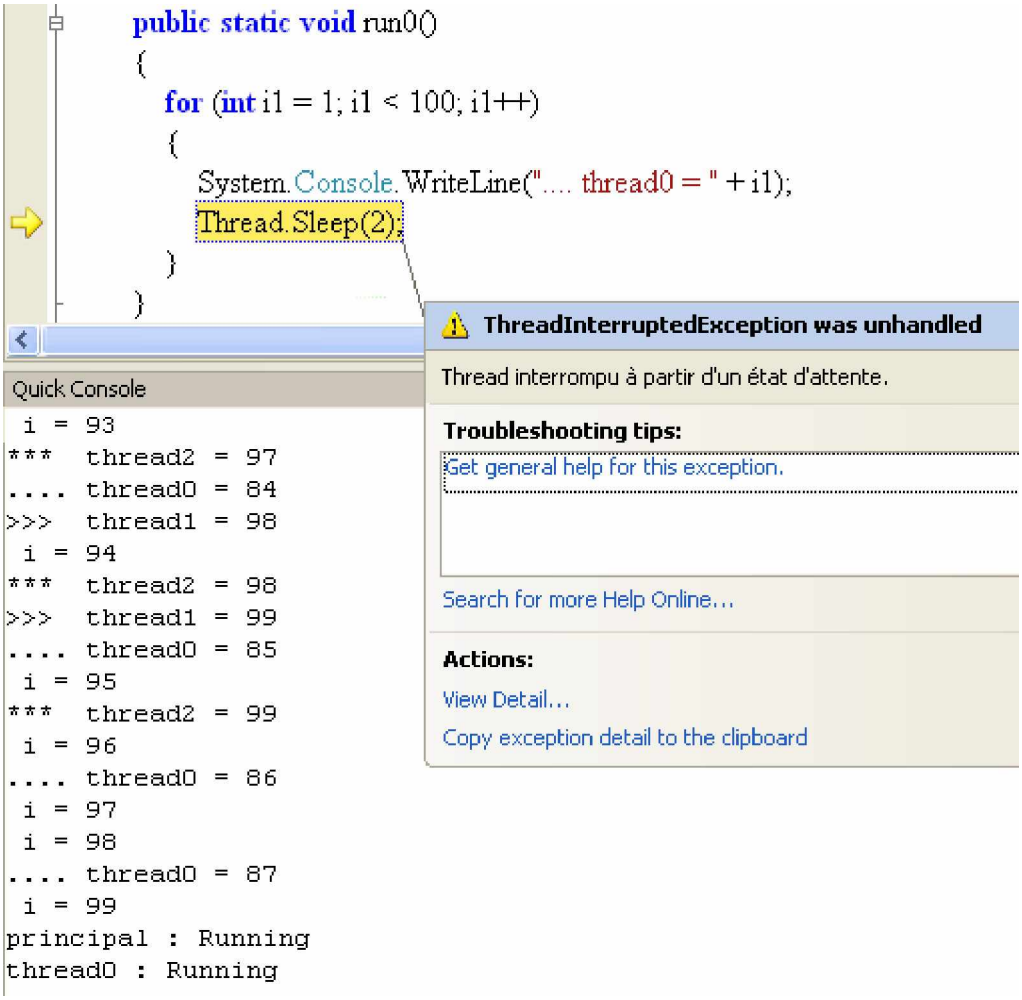
principal : Running

thread0 : Stopped
thread1 : Stopped
thread2 : Stopped
fin de tous les threads.

Interrompre-réveiller : Méthode Interrupt (…)

Si le thread est dans l'état **WaitSleepJoin** c'est à dire soit endormi (**Sleep**), soit dans la file d'attente attendant son tour (**Wait**), soit en attente de la fin d'un autre thread (**Join**), il est alors possible d'interrompre son état d'attente grâce à la méthode **Interrupt**. Cette méthode interrompt temporairement le thread qui l'invoque et lance une exception du type **ThreadInterruptedException**.

Dans le programme précédent nous ajoutons l'instruction "thread0.Interrupt()" dans la méthode Main, juste après la fin de la boucle **for** (**int** i = 1; i < 100; i++) {…}. Lors de l'exécution, dès que le thread0 se met en mode **WaitSleepJoin** par invocation de la méthode Sleep(2), il est interrompu :



Une exception **ThreadInterruptedException** a bien été lancée et peut être interceptée dans le thread principal.

Nous listons ci-dessous le code source de la méthode Main produisant l'interruption du thread0 figurée précédemment :

```
static void Main(string[] args)
{
        Console.WriteLine("Lancement des threads :");
        AfficherDonnees obj = new AfficherDonnees();
        Thread principal = Thread.CurrentThread;
        principal.Name = "principal";
        Thread thread0 = new Thread(new ThreadStart(run0));
        thread0.Name = "thread0";
        Thread thread1 = new Thread(new ThreadStart(AfficherDonnees.run1));
        thread1.Name = "thread1";
        Thread thread2 = new Thread(new ThreadStart(obj.run2));
        thread2.Name = "thread2";
        etatsThreads(principal, thread0, thread1, thread2);
        thread0.Start();
        thread1.Start();
        thread2.Start();
        etatsThreads(principal, thread0, thread1, thread2);
        for (int i = 1; i < 100; i++)
        {
           System.Console.WriteLine(" i = " + i);
        }
        thread0.Interrupt();
        etatsThreads(principal, thread0, thread1, thread2);
        thread0.Join();
        etatsThreads(principal, thread0, thread1, thread2);
        System.Console.WriteLine("fin de tous les threads.");
}
```

Attention : La rapidité du processeur, l'influence du CLR et la charge instantanée du système changent considérablement les résultats obtenus ! Il faut donc n'utiliser ces outils que pour du parallélisme réel et non pour du séquentiel (cf. la notion de synchronisation paragraphe suivant)

3.3 Exclusion mutuelle, concurrence, section critique, et synchronisation

ressource partagée

D'un point de vue général, plusieurs processus peuvent accéder en lecture et en écriture à un **même espace mémoire** par exemple : accéder à un spooler d'imprimante, réserver une page en mémoire centrale, etc…Cet espace mémoire partagé par plusieurs processus se dénomme **une ressource partagée**.

concurrence

Lorsque le résultat final après exécution des processus à l'intérieur sur une ressource partagée n'est pas déterministe, mais dépend de l'ordre dans lequel le système d'exploitation a procédé à l'exécution de chacun des processus, on dit que l'on est en situation de **concurrence.**

synchronisation

Lorsqu'une structure de données est accédée par des processus en situation de concurrence, il est impossible de prévoir le comportement des threads sur cette structure. Si l'on veut obtenir un comportement déterministe, il faut ordonner les exécutions des processus d'une manière séquentielle afin d'être sûr qu'**un seul** processus accède à toute la structure **jusqu'à la fin** de son exécution , puis laisse la main au processus suivant etc. Cet régulation des exécutions des processus s'appelle la **synchronisation**.

Exclusion mutuelle

> Lorsque plusieurs processus travaillent sur une ressource partagée, la synchronisation entre les divers processus sous-entend que cette ressource partagée est exclusivement à la disposition d'un processus pendant sa durée complète d'exécution. Le mécanisme qui permet à un seul processus de s'exécuter sur une ressource partagée à l'exclusion de tout autre, se dénomme l'**exclusion mutuelle**.

section critique

> Lorsqu'un bloc de lignes de code traite d'accès par threads synchronisés à une ressource partagée on dénomme ce bloc de code une **section critique**.

Tout ce que nous venons de voir sur les processus se reporte intégralement aux **threads** qui sont des processus légers.

Dans un système d'exploitation de multiprogrammation, ces situations de concurrence sont très courantes et depuis les années 60, les informaticiens ont mis en œuvre un arsenal de réponses d'exclusion mutuelle ; ces réponses sont fondées sur les notions de **verrous**, **sémaphores**, **mutex**, **moniteurs**.

Le développeur peut avoir besoin dans ses programmes de gérer des situations de concurrence comme par exemple dans un programme de réservation de place de train et d'édition de billet de transport voyageur. Le multi-threading et les outils de synchronisation que le langage de programmation fournira seront une aide très précieuse au développeur dans ce style de programmation.

Pour programmer de la synchronisation entre threads, le langage C# met à la disposition du développeur les notions de **verrous**, **sémaphores**, **mutex**, **moniteurs**.

3.4 Section critique en C# : lock

L'instruction lock (…) { …. }

Le mot clef **lock** détermine un bloc d'instructions en tant que **section critique**. Le verrouillage de cette section critique est obtenu par exclusion mutuelle sur un objet spécifique nommé verrou qui peut être dans deux états : soit disponible ou libre, soit vérouillé. Ci-dessous la syntaxe C# de l'instruction lock :

```
object verrou = new object();
lock ( verrou )
{
   ... lignes de code de la section critique
}
```

Lorsqu'un thread Th1 veut entrer dans la section critique délimitée par lock, nous sommes en face de deux possibilités selon que l'objet verrou est libre ou non :

- **Si l'objet verrou est libre** (c'est à dire qu'aucun autre thread n'exécute le code de la section critique) alors on dit que le thread Th1 acquiert le verrou, d'autre part il le verrouille pour tout autre thread. Dès que le thread Th1 finit d'exécuter la dernière instruction de la section critique, il libère le verrou qui devient disponible pour un autre

thread.

- **Si l'objet verrou n'est pas libre** et qu'un thread Th2 demande à acquérir ce verrou (veut entrer dans la section critique) pendant que Th1 est dans la section critique, le thread Th2 est alors mis dans la file d'attente associée à l'objet verrou par le CLR (le thread est donc bloqué en attente). Chaque nouveau thread demandant à acquérir ce verrou est rangé dans la file d'attente du verrou tant que ce dernier n'est pas libéré. Dès que le verrou devient libre le CLR autorise le thread en tête de file à acquérir le verrou et ce thread est retiré de la file d'attente des threads du verrou.
- **Si une exception est levée** dans la section critique, le verrou est automatiquement libéré par l'instruction lock pour ce thread.

Exemple C# de synchronisation avec lock :

1°) Partons d'un exemple où le fait qu'il n'y ait pas de synchronisation provoque un comportement erratique. Soit un tableau d'entiers "**int**[ ] datas = **new int**[50]" tous à la valeur 1, nous construisons trois threads modifiant ce tableau, le premier rajoute 1 à chaque cellule du tableau, le second multiplie par 2 le contenu de chaque cellule du tableau, le troisième soustrait 1 à chaque cellule du tableau.

Nous créons une classe ThreadModifierDonnees dont la vocation est de permettre à des délégués, à qui nous donnerons un nom lors de leur instanciation, de travailler sur le tableau d'entiers à travers une méthode run().

Dans cette classe ThreadModifierDonnees la méthode run( ) effectue une action différente selon le nom du délégué qui l'invoque (ajouter 1, multiplier par 2, soustraire 1).

Afin de souligner la concurrence entre les 3 threads nous déséquilibrons les temps alloués aux threads par un endormissement différent pour le thread1 (Thread.Sleep(1);) et pour le thread2 (Thread.Sleep(0) ;), le thread3 restant indemne de toute modification temporelle :

```
public class ThreadModifierDonnees
{
    private int[ ] donnees;
    private string nom;
    public ThreadModifierDonnees(int[ ] donnees, string nom)
    {
       this.donnees = donnees;
       this.nom = nom;
    }
    public void run( )
    {
       for (int i = 0; i < donnees.Length; i++)
       {
            if (nom == "modif1")
            {
               donnees[i] += 1;
               Thread.Sleep(1);
            }
            else
               if (nom == "modif2")
               {
                  donnees[i] *= 2;
                  Thread.Sleep(0);
               }
               else
                  if (nom == "modif3")
```

```
                    donnees[i] -= 1;
            }
        }
}

public class Program
{
    public static int[ ] datas = new int[50];

    public static void afficherDatas()
    {
        for (int i = 0; i < datas.Length; i++)
            Console.Write(datas[i]);
        Console.WriteLine();
    }
    public static void initDatas()
    {
        for (int i = 0; i < 50; i++)
            datas[i]=1;
    }
    static void Main(string[] args)
    {
        initDatas();
        ThreadModifierDonnees modif_1 = new ThreadModifierDonnees(datas,"modif1");
        ThreadModifierDonnees modif_2 = new ThreadModifierDonnees(datas,"modif2");
        ThreadModifierDonnees modif_3 = new ThreadModifierDonnees(datas, "modif3");
        afficherDatas();
        Thread thread1 = new Thread(new ThreadStart(modif_1.run));
        Thread thread2 = new Thread(new ThreadStart(modif_2.run));
        Thread thread3 = new Thread(new ThreadStart(modif_3.run));
        thread1.Start();
        thread2.Start();
        thread3.Start();
        afficherDatas();
        System.Console.ReadLine();
    }
 }
```

La méthode initDatas() rempli de 1 le tableau datas, la méthode afficherDatas affiche le contenu des 50 cellules du tableau datas.

Après avoir lancé les 3 threads nous affichons afficherDatas affiche de nouveau le contenu des 50 cellules du tableau datas. En relançant plusieurs fois le programme nous obtenons des résultats différents (qui peuvent aussi différer d'une machine à l'autre) dûs au fait que les threads ne sont pas synchronisés sur la boucle **for** de la méthode **run** :

Exécution : ***(Résultats dépendants de votre configuration machine+OS)***
11111111111111111111111111111111111111111111111111
33222222211111111111111111111111111111111111111111

Autre exécution : ***(Résultats dépendants de votre configuration machine+OS)***
11111111111111111111111111111111111111111111111111
33222222222211111111111111111111111111111111111111

Autre exécution : ***(Résultats dépendants de votre configuration machine+OS)***
11111111111111111111111111111111111111111111111111
33222221111111111111111111111111111111111111111111
etc …

2°) A l'évidence, les résultats ne sont pas fiables puisqu'ils dépendent du temps et de la charge du système, donc ce programme n'est pas valide. Afin de remédier à cet inconvénient majeur nous utilisons un verouillage de la boucle for dans la méthode run et nous déclarons un verrou de type object dans la classe ThreadModifierDonnees. L'objet verrou est **static** afin que ce soit le même objet pour toutes les instances de telle sorte que les threads consulteront le même verrou :

```
public class ThreadModifierDonnees
{
    private int[] donnees;
    private string nom;
    private static object verrou = new object();

    public ThreadModifierDonnees(int[] donnees, string nom)
    {
        this.donnees = donnees;
        this.nom = nom;
    }
    public void run()
    {
        lock (verrou)
        {
            for (int i = 0; i < donnees.Length; i++)
            {
                if (nom == "modif1")
                {
                    donnees[i] += 1;
                    Thread.Sleep(1);
                }
                else
                    if (nom == "modif2")
                    {
                        donnees[i] *= 2;
                        Thread.Sleep(0);
                    }
                    else
                        if (nom == "modif3")
                        {
                            donnees[i] -= 1;
                        }
            }
        }
    }
}
```

Le code de la classe Program reste identique, et les résultats de l'exécution sont déterministes et dirigés uniquement par l'ordre séquentiel de lancement des 3 threads

1°) Pour l'ordre de lancement des threads suivant :

```
thread1.Start();
thread2.Start();
thread3.Start();
```

Résultats d'exécution pour l'ordre de lancement 1,2,3 :
11111111111111111111111111111111111111111111111111
33333333333333333333333333333333333333333333333333

2°) Pour l'ordre de lancement des threads suivant:

```
thread2.Start();
thread1.Start();
thread3.Start();
```

Résultats d'exécution pour l'ordre de lancement 2,1,3 :
1111111111111111111111111111111111111111111111111
2222222222222222222222222222222222222222222222222

3°) Pour l'ordre de lancement des threads suivant:

```
thread3.Start();
thread2.Start();
thread1.Start();
```

Résultats d'exécution pour l'ordre de lancement 3,2,1 :
1111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111

Si le thread prinicipal « va trop vite » on lui demande d'attendre la fin de l'exécution des 3 autres threads avant qu'il exécute la méthode afficherDatas()

```
thread1.Join(); // on attend que ce thread ait terminé la totalité de son exécution
thread2.Join(); // on attend que ce thread ait terminé la totalité de son exécution
thread3.Join(); // on attend que ce thread ait terminé la totalité de son exécution
afficherDatas();
```

3.5 Section critique en C# : Monitor

La classe Monitor

Cette classe fournit un moyen général de synchroniser les accès à un objet en C#. Elle est abstraite

> **public sealed abstract class** Monitor

Elle contient des méthodes permettant à un thread de travailler plus finement sur une section critique que l'instruction lock. Voici trois méthodes static de la classe Monitor permettant à un thread :

- D'acquérir un verrou : **public static void** Enter(object obj);
- De libérer un verrou : **public static void** Exit(object obj);
- D'essayer d'aquérir un verrou : **public static bool** TryEnter(object obj);

Les méthodes Enter et Exit permettent de délimiter une section critique :

```
object verrou = new object();
Monitor.Enter ( verrou ) ;
  ... lignes de code de la section critique
Monitor.Exit ( verrou ) ;
```

La méthode TryEnter permet à un thread de consulter l'état d'un verrou et dans l'éventualité où ce verrou est verrouillé, si le développeur le souhaite, de ne pas mettre immédiatement le thread dans la file d'attente du verrou, mais d'effectuer d'autres actions. Ci-dessous un pseudo-code C# dans lequel le thread qui exécute la méthode2() teste l'état du verrou avant de rentrer dans la section critique contenue dans la méthode1() ou bien effectue d'autres actions si cette section critique est occupée par un autre thread :

```
private object verrou = new object();

public void methode1()
{
  Monitor.Enter ( verrou ) ;
    ... lignes de code de la section critique
  Monitor.Exit ( verrou ) ;
}
…….
public void methode2()
{
  if (Monitor.TryEnter ( verrou )
    methode1( );
  else
    ...Autres actions
}
```

Dans le cas où une exception serait levée dans une section critique le verrou n'est pas automatiquement levé comme avec l'instruction **lock**(){…}, Microsoft conseille au développeur de protéger son code par un **try**…**finally**. Voici le code minimal permettant d'assurer cette protection semblablement à un lock :

```
object verrou = new object();
Monitor.Enter ( verrou ) ;
try
 {
  ... lignes de code de la section critique
 }
finally
 {
   Monitor.Exit ( verrou ) ;
 }
```

Dans l'exemple précédent de section critique mettant à jour les 50 cellules d'un tableau d'entiers, nous remplaçons l'instruction **lock** par un appel aux méthodes **static** *Enter* et *Exit* de la classe Monitor sans protection du code :

```
public class ThreadModifierDonnees
  {
     private int[] donnees;
     private string nom;
     private static object verrou = new object();
```

```
    public ThreadModifierDonnees(int[] donnees, string nom)
    {
       this.donnees = donnees;
       this.nom = nom;
    }
    public void run()
    {
          Monitor.Enter ( verrou ) ;
          for (int i = 0; i < donnees.Length; i++)
          {
             if (nom == "modif1")
             {
                donnees[i] += 1;
                Thread.Sleep(1);
             }
             else
                if (nom == "modif2")
                {
                   donnees[i] *= 2;
                   Thread.Sleep(0);
                }
                else
                   if (nom == "modif3")
                   {
                      donnees[i] -= 1;
                   }
          }
          Monitor.Exit ( verrou ) ;
    }
}
```

Même programme avec appel aux méthodes **static** *Enter* et *Exit* de la classe Monitor et protection du code par **try**…**finally** dans la méthode run( ):

```
    public void run()    {
      Monitor.Enter ( verrou ) ;
       try
       {
          for (int i = 0; i < donnees.Length; i++)    {
             if (nom == "modif1")
             {
                donnees[i] += 1;
                Thread.Sleep(1);
             }
             else
                if (nom == "modif2")
                {
                   donnees[i] *= 2;
                   Thread.Sleep(0);
                }
                else
                   if (nom == "modif3")
                   {
                      donnees[i] -= 1;
                   }
          }
       }
       finally   {  Monitor.Exit ( verrou ) ;   }
    }
```

3.6 Synchronisation commune aux processsus et aux threads

Il existe dans la version C# 2.0 deux classes permettant de travailler aussi bien avec des threads qu'avec des processus.

La classe Semaphore

> **public sealed class** Semaphore : WaitHandle

Semaphore est une classe dédiée à l'accès à une ressource partagée non pas par un seul thread mais par un nombre déterminé de threads lors de la création du sémaphore à travers son constructeur dont nous donnons une surcharge :

> **public** Semaphore (**int** initialCount, **int** maximumCount);
>
> initialCount = le nombre de thread autorisés à posséder le sémaphore en plus du thread principal, la valeur 0 indique que seul le thread principal possède le sépmaphore au départ.
>
> MaximumCount = le nombre maximal de thread autorisés à posséder le sémaphore.

Pour essayer de simuler une section critique accessible par un seul thread à la fois par la notion de verrou, il faut instancier un sémaphore avec un maximumCount = 1 et utiliser les méthodes *WaitOne* et *Release* de la classe Semaphore :

```
Semaphore verrou = new Semaphore (0, 1);

verrou.WaitOne ( );
... lignes de code de la section critique
verrou.Release ( );
```

Toutefois l'utilisation d'un sémaphore même avec *maximumCount = 1* ne permet pas d'ordonnancer les accès à la section critique, dans ce cas le sémaphore sert seulement à assurer qu'une section critique est accédée entièrement par un seul thread. Illustrons ce propos avec notre exemple de code partagé mettant à jour les 50 cellules d'un tableau d'entiers.

Nous créons un sémaphore **public static** dans la classe ThreadModifierDonnees que nous nommons verrou, nous supprimons les endormissements Sleep(…), nous instancions le sémaphore Semaphore verrou = **new** Semaphore (0, 1), enfin nous encadrons la section critique par les appels des méthodes *WaitOne* et *Release* :

```
public class ThreadModifierDonnees
  {
     private int[] donnees;
     private string nom;
     public static Semaphore verrou = new Semaphore ( 0, 1 );
```

```
    public ThreadModifierDonnees(int[] donnees, string nom)
    {
       this.donnees = donnees;
       this.nom = nom;
    }
    public void run()
    {
       verrou.WaitOne( );
          for (int i = 0; i < donnees.Length; i++)
          {
             if (nom == "modif1")
             {
                donnees[i] += 1;
                Thread.Sleep(1);
             }
             else
                if (nom == "modif2")
                {
                   donnees[i] *= 2;
                   Thread.Sleep(0);
                }
                else
                   if (nom == "modif3")
                   {
                      donnees[i] -= 1;
                   }
          }
       verrou.Release( );
    }
}
```

Dans la méthode Main de la classe Program, par construction du sémaphore c'est le thread principal qui possède le sémaphore verrou, on fait libérer ce sémaphore par le thread principal par l'instruction "ThreadModifierDonnees.verrou.Release(1)" qui a vocation à autoriser un des 3 threads thread1, thread2 ou thread3 à entrer dans la section critique :

```
  static void Main(string[] args)
  {
        initDatas();
        ThreadModifierDonnees modif_1 = new ThreadModifierDonnees(datas,"modif1");
        ThreadModifierDonnees modif_2 = new ThreadModifierDonnees(datas,"modif2");
        ThreadModifierDonnees modif_3 = new ThreadModifierDonnees(datas, "modif3");
        afficherDatas();
        Thread thread1 = new Thread(new ThreadStart(modif_1.run));
        Thread thread2 = new Thread(new ThreadStart(modif_2.run));
        Thread thread3 = new Thread(new ThreadStart(modif_3.run));
        thread1.Start();
        thread2.Start();
        thread3.Start();
        ThreadModifierDonnees.verrou.Release ( 1 );
        afficherDatas();
        System.Console.ReadLine();
    }
  }
```

Le résultat obtenu est identique à celui que nous avions obtenu avec lock ou Monitor :

Pour l'ordre de lancement des threads suivant :

```
thread1.Start();
thread2.Start();
thread3.Start();
```

Résultats d'exécution :
1111111111111111111111111111111111111111111111111
3333333333333333333333333333333333333333333333333

Il ne faut pas s'y méprendre, si l'on change l'ordre de lancement on a toujours l'accès garanti par le sémaphore à la section critique par un seul thread, mais l'ordre d'accès séquentiel n'est pas garanti :

1°) Pour l'ordre de lancement des threads suivant:

```
thread2.Start();
thread1.Start();
thread3.Start();
```

Résultats d'exécution :
1111111111111111111111111111111111111111111111111
3333333333333333333333333333333333333333333333333

2°) Pour l'ordre de lancement des threads suivant:

```
thread3.Start();
thread2.Start();
thread1.Start();
```

Résultats d'exécution :
1111111111111111111111111111111111111111111111111
2222222222222222222222222222222222222222222222222

La classe Mutex

> **public sealed class** Mutex : WaitHandle

Mutex est une classe dédiée à la synchronisation entre processus ou entre threads. Elle permet un accès exclusif à une ressource partagée en situation de concurrence : elle autorise un accès exclusif à cette ressource par un seul thread ou processus. Si un thread ou processus acquiert un mutex, l'autre thread ou processus qui veut acquérir ce mutex est interrompu jusqu'à ce que le premier thread ou processus libère le mutex. Elle fonctionne comme un sémaphore à un seul thread autorisé.

Les méthodes intéressantes pour gérer une section critique sont semblables à celle de la classe Semaphore, on peut considérer la classe Mutex comme une simplification de la classe Semaphore dans le cas d'un seul thread ou d'un seul processus qui accède à la section critique. Elle possède toutefois l'avantage sur la classe Semaphore de permettre de gérer l'**ordre d'accès séquentiel à la section critique.**

On utilise les méthodes *WaitOne* et *ReleaseMutex* de la classe Mutex:

```
Semaphore verrou = new Mutex ( );
```

```
verrou.WaitOne ( );
... lignes de code de la section critique
verrou.ReleaseMutex ( );
```

Ci-dessous le code de la classe ThreadModifierDonnees avec un mutex **public** que nous nommons verrou, nous avons aussi supprimé les endormissements Sleep(…), et nous encadrons la section critique par les appels des méthodes *WaitOne* et *ReleaseMutex* :

```
public class ThreadModifierDonnees
  {
     private int[] donnees;
     private string nom;
     public static Mutex verrou = new Mutex ( );

     public ThreadModifierDonnees(int[] donnees, string nom)
     {
        this.donnees = donnees;
        this.nom = nom;
     }
     public void run()
     {
        verrou.WaitOne( );
           for (int i = 0; i < donnees.Length; i++)
           {
              if (nom == "modif1")
              {
                 donnees[i] += 1;
                 Thread.Sleep(1);
              }
              else
                 if (nom == "modif2")
                 {
                    donnees[i] *= 2;
                    Thread.Sleep(0);
                 }
                 else
                    if (nom == "modif3")
                    {
                       donnees[i] -= 1;
                    }
           }
        verrou.ReleaseMutex( );
     }
  }
```

Nous retrouvons dans la méthode Main de la classe Program, un code identique à celui correspondant à une utilisation d'un **lock** ou d'un Monitor :

```
  static void Main(string[] args)
  {
        initDatas();
        ThreadModifierDonnees modif_1 = new ThreadModifierDonnees(datas,"modif1");
        ThreadModifierDonnees modif_2 = new ThreadModifierDonnees(datas,"modif2");
        ThreadModifierDonnees modif_3 = new ThreadModifierDonnees(datas, "modif3");
        afficherDatas();
        Thread thread1 = new Thread(new ThreadStart(modif_1.run));
```

```
        Thread thread2 = new Thread(new ThreadStart(modif_2.run));
        Thread thread3 = new Thread(new ThreadStart(modif_3.run));
        thread1.Start();
        thread2.Start();
        thread3.Start();
        afficherDatas();
        System.Console.ReadLine();
    }
  }
```

Nous donnons les résultats obtenus selon l'ordre de lancement des thread :

1°) Pour l'ordre de lancement des threads suivant :

```
thread1.Start();
thread2.Start();
thread3.Start();
```

Résultats d'exécution :
11111111111111111111111111111111111111111111111111
33333333333333333333333333333333333333333333333333

2°) Pour l'ordre de lancement des threads suivant:

```
thread2.Start();
thread1.Start();
thread3.Start();
```

Résultats d'exécution :
11111111111111111111111111111111111111111111111111
22222222222222222222222222222222222222222222222222

3°) Pour l'ordre de lancement des threads suivant:

```
thread3.Start();
thread2.Start();
thread1.Start();
```

Résultats d'exécution :
11111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111

Ces résultats sont identiques à ceux que l'on obtient en utilisant un **lock**.

En conclusion, l'instruction **lock** s'avère être la plus souple et la plus simple à utiliser pour définir une section critique et pour assurer une synchronisation efficace des threads sur cette section critique.