Débogage et gestion des erreurs

Le débogueur Visual Studio .NET est un puissant outil grâce auquel il est possible d'observer le comportement du programme au moment de l'exécution et de déterminer l'emplacement des erreurs. Il comprend des fonctionnalités qui sont intégrées dans les langages de programmation et leurs bibliothèques. Avec le débogueur, l'exécution du programme peut être arrêtée (suspendue) pour permettre au programmeur d'examiner le code, de l'exécuter pas à pas, et d'en évaluer les variables.

Il n'est pas question de faire ici un cours détaillé sur l'usage du débogueur, mais quelques fonctionnalités sont des plus utiles en cours de développement :

- Le fonctionnement en pas à pas
- Le point d'arrêt
- L'évaluation des variables en cours d'exécution
- L'insertion d'envois de messages

L'environnement Visual Studio .Net effectue l'analyse du code en cours de dactylographie. Pendant cette analyse, deux aides essentielles sont activées. La première, c'est l'aide en ligne qui informe le programmeur sur la syntaxe et les paramètres à fournir pour l'instruction commencée. La seconde, c'est le soulignement de la plupart des fautes de syntaxe, à la manière du correcteur orthographique du traitement de textes.

```
Private Sub BFin_GotFocus(ByVal sender As Object, ByVal e As System.EventArgs) Handles BFin.GotFocus

    MonBouton.

End Sub
```

Le soulignement bleu ondulé en cours de frappe avise le programmeur que son instruction est erronée ou incomplète. Dans cet exemple, le nom de l'objet est incorrect.

```
Private Sub BFin_GotFocus(ByVal sender As Object, ByVal e As System.EventArgs) Handles BFin.GotFocus

    BFin.
End Su
```

TabIndex
TabStop
Tag
Text
TextAlign

Le nom de l'objet a été corrigé, mais l'instruction est toujours incomplète. Un caractère reste souligné. Dès la frappe du point, s'affiche la liste des *vitamines* plausibles pour ce cas.

```
Private Sub BFin_GotFocus(ByVal sender As Object, ByVal e As System.EventArgs) Handles BFin.GotFocus

    BFin.Text
End
```

Text () As String
Obtient ou définit le texte associé à ce contrôle.

C'est une propriété de l'objet qui est choisie ici. Une aide contextuelle informe sur son type et sur son rôle. Mais l'instruction est toujours incomplète. Un caractère reste souligné.

```
Private Sub BFin_GotFocus(ByVal sender As Object, ByVal e As System.EventArgs) Handles BFin.GotFocus

    BFin.Text = Arrêt

End Sub
```

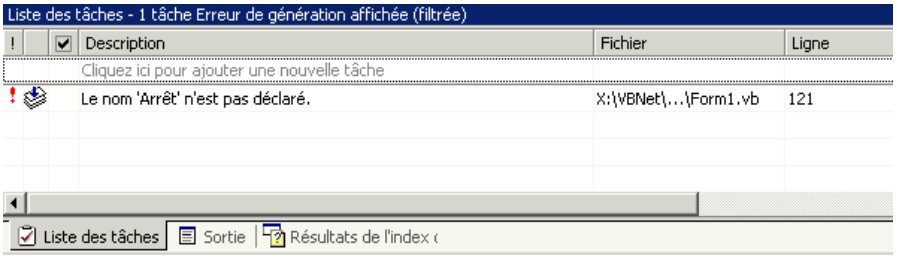
L'instruction est complétée, mais le programmeur a oublié qu'une valeur littérale alphanumérique s'encode entre guillemets. Il y a faute de syntaxe. Un soulignement subsiste.

```
Module UnModule
    Sub Main()
        Dim Nom As String
```

Variable locale inutilisée : 'Nom'.

A partir de la version 2005 de Visual Studio, l'environnement signale également des anomalies qui ne portent pas forcément préjudice au bon fonctionnement du programme. Ces *warnings* sont également soulignés d'un trait ondulé, mais celui-ci est de couleur verte.

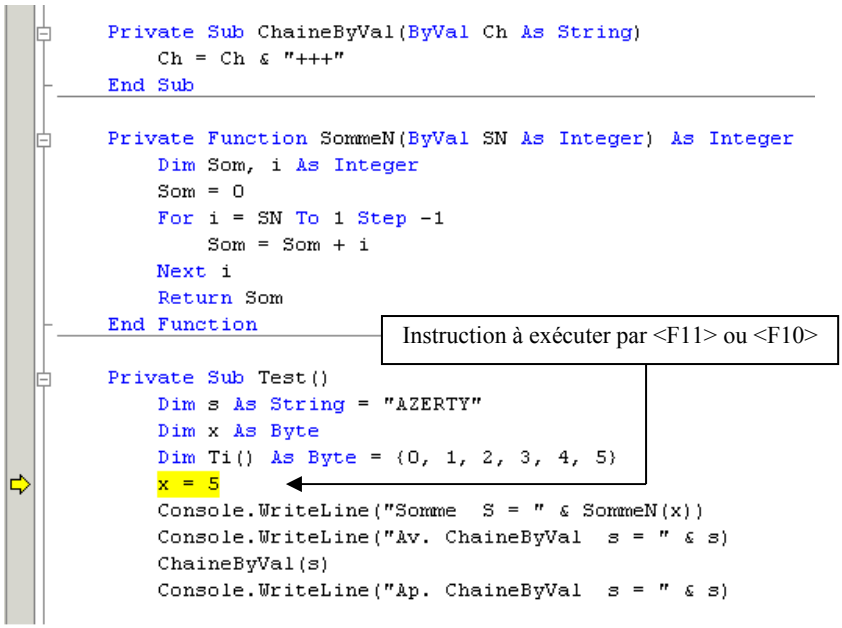
Si le programmeur lance l'exécution avec son code erroné, il est averti que des erreurs sont détectées. Il a alors le choix d'arrêter l'exécution ou la poursuivre. Dans ce dernier cas, les lignes fautives sont ignorées. S'il préfère corriger son code, la fenêtre *Liste des tâches* l'informe sur le type et l'endroit de l'erreur. Un double clic sur une ligne ainsi présentée renvoie directement le programmeur là où une correction s'impose. Les messages d'erreurs disparaissent de la liste au fur et à mesure des corrections, sans qu'il soit nécessaire de recompiler.



Mais il n'y pas que des fautes de syntaxe dans un programme en cours de développement. Lorsque le programme est exécuté, le programmeur a l'occasion de constater que le déroulement ne s'effectue pas comme prévu, que les résultats ne sont pas ceux attendus. Il y a des erreurs logiques. C'est ici que les fonctionnalités du débogueur énumérées ci-dessus s'avérent très utiles.

Le pas à pas

C'est l'exécution ligne par ligne du code, le programmeur activant l'exécution de la ligne présentée quand il le souhaite selon un des deux modes possibles.
Le pas à pas détaillé se commande par la touche <F11> et le pas à pas principal par <F10>. En mode détaillé, l'exécution se poursuit ligne par ligne jusqu'à l'intérieur des fonctions et procédures. Lorsque l'exécution est sur le point d'entrer dans un sous-programme qu'il n'est pas opportun d'étudier ainsi, le mode principal en effectue l'exécution sans y entraîner le programmeur.

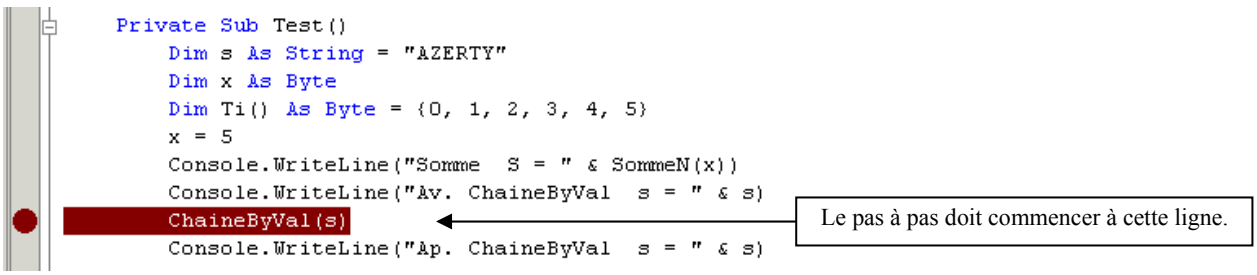


La ligne de code sur le point d'être exécutée est désignée par une flèche jaune dans la marge gauche de la feuille de code et la ligne de code elle-même est surlignée.
Pour que la ligne s'exécute, le programmeur doit presser <F11>.
Dans l'exemple ci-contre, la ligne suivant celle indiquée par la flèche fait appel à la fonction **`SommeN()`**. Le moment venu d'exécuter cette ligne, le programmeur doit presser <F11> s'il souhaite exécuter la fonction en pas à pas, et <F10> dans le cas contraire.

Au sein d'une même procédure, le programmeur peut provoquer la ré-exécution d'une ligne ou au contraire éviter qu'une ligne soit exécutée. Il lui suffit pour cela de glisser la flèche jaune jusqu'à la ligne de code souhaitée.

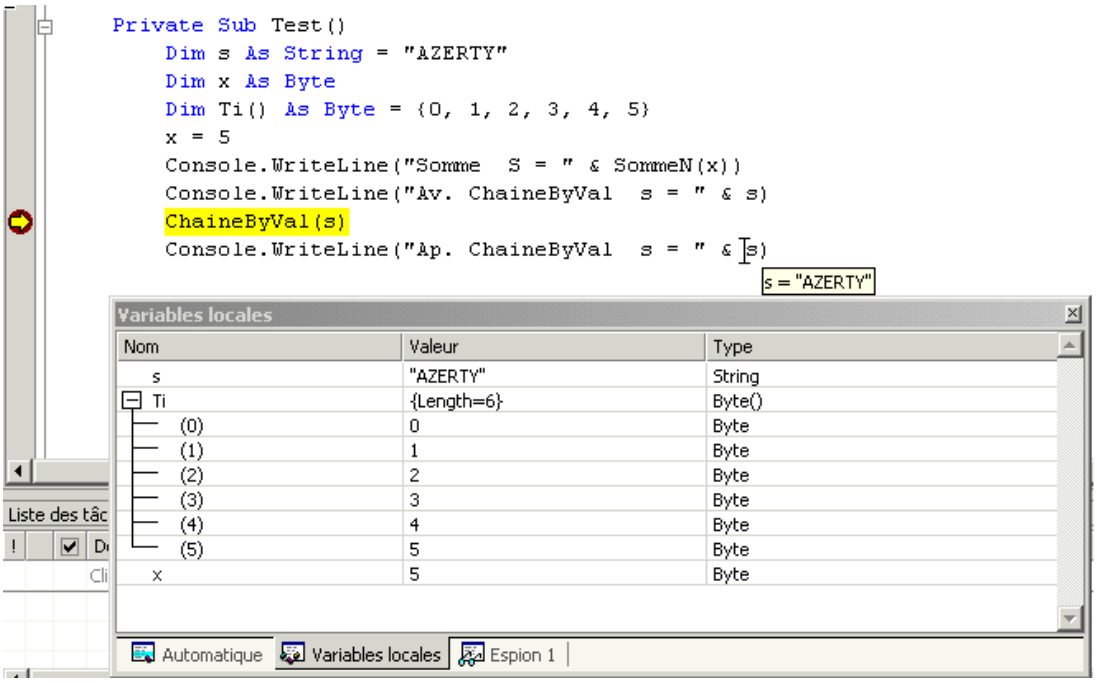
Le point d'arrêt

Comme il n'est pratiquement jamais utile d'exécuter tout le code en pas à pas, le programmeur place un point d'arrêt dans son code à l'endroit à partir duquel il souhaite étudier l'exécution en détail. L'exécution du programme se déroule ordinairement jusqu'à la première ligne marquée d'un point d'arrêt et se met alors automatiquement en pas à pas. Le programmeur peut poursuivre par <F11> ou <F10>. Quand le programmeur le souhaite, l'appui de la touche <F5> provoque l'abandon du pas à pas jusqu'au prochain point d'arrêt ou jusqu'à la fin du programme.
Un point d'arrêt est marqué par un clic dans la marge gauche de la feuille de code en regard de la ligne de code souhaitée pour le départ du pas à pas. Un point brun est placé dans la marge et la ligne de code est surlignée. Un point d'arrêt ne peut être fixé sur une déclaration, mais seulement sur une instruction valide.



L'évaluation des variables en cours d'exécution

Pendant l'exécution en pas à pas, il suffit d'amener le curseur de la souris sur une variable pour que s'affiche spontanément sa valeur. En outre, une fenêtre présentant divers onglets dont *Variables locales* et *Espion* est présente pendant le débogage. On accède en permanence aux valeurs de toutes les variables de la procédure étudiée par l'onglet *Variables locales*. L'onglet *Espion* permet la garde de variables explicitement désignées de n'importe quelle procédure. La mise en garde d'une variable s'effectue par un *copier* de la variable à partir du code et son *coller* dans cet onglet.



L'insertion d'envois de messages

L'insertion d'envois de messages n'est plus du ressort du débogueur, mais bien d'un jeu d'outils adéquats offerts par le langage.
Comme de tout temps, le programmeur peut placer des instructions de sortie avec les messages qu'il souhaite aux endroits qu'il veut tester. Mais cette méthode présente au moins deux inconvénients. D'une part ces messages, qui ne peuvent généralement pas subsister dans la version compilée définitive, doivent être retirés manuellement. D'autre part, les contenus des messages disparaissent aussitôt l'exécution terminée et ne supportent donc pas une deuxième lecture.

Les moyens offerts par VB.Net sont communs à tout Visual Studio .Net et ils sont livrés par les deux classes **`Trace`** et **`Debug`**. Ces deux classes sont identiques sauf que les méthodes de **`Trace`** subsistent dans les versions définitives tandis que celles de **`Debug`** ne sont pas intégrées dans les exécutables.

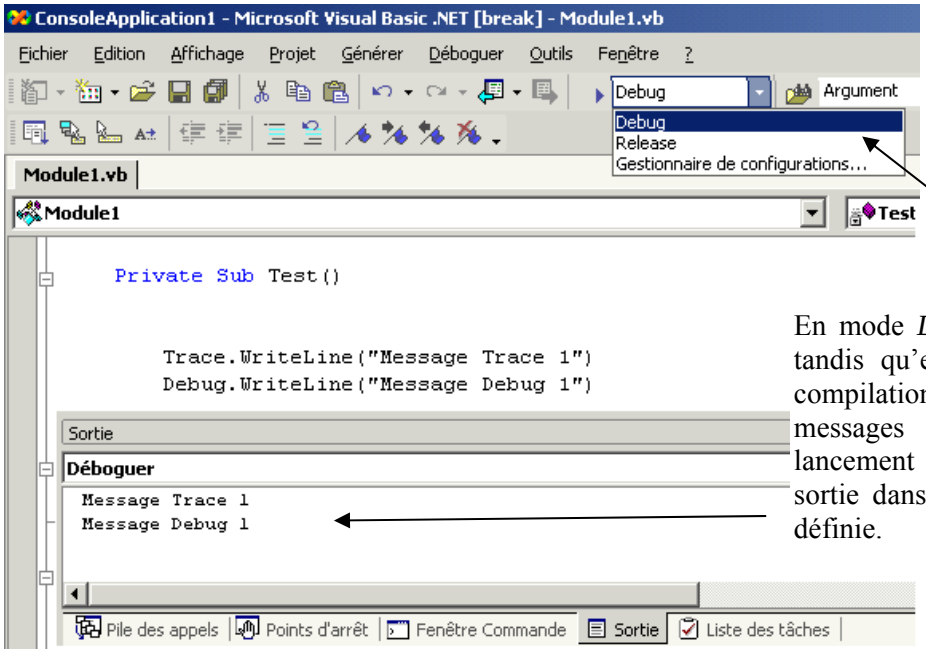
Ces classes envoyent leurs messages vers des *écouteurs* qui les redirigent vers des sorties désignées. Il y trois types d'écouteurs prédéfinis :

1. **`TextWriterTraceListener`** redirige la sortie vers une instance de la classe **`TextWriter`** ou tout autre type de la classe **`Stream`** (**`Console`** ou fichier)
2. **`EventLogTraceListener`** redirige la sortie vers un journal d'événements
3. **`DefaultTraceListener`** renvoie les messages vers les méthodes **`OutPutDebugString`** et **`Debugger.Log`**. Pendant la mise au point des codes, ces messages sont affichés dans la fenêtre *Sortie* de Visual Studio. C'est le seul écouteur par défaut des classes **`Trace`** et **`Debug`** car c'est le seul qui soit automatiquement inclus dans la collection **`Listeners`**.

Tous les écouteurs de la collection reçoivent les mêmes messages des méthodes de sorties. Simplement, chacun le redirige vers la sortie qui lui a été désignée, un flux pour **`TextWriterTraceListener`** et un journal d'événements pour **`EventLogTraceListener`**.

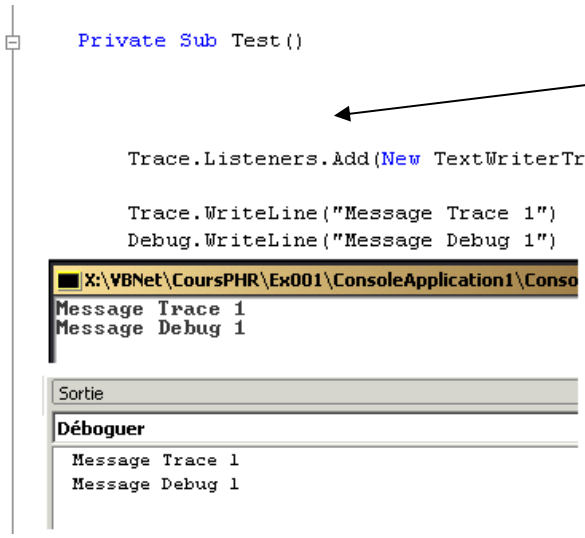
Les méthodes de sorties pour **`Trace`** et **`Debug`** sont **`Write`**, **`WriteLine`**, **`Fail`** qui émettent inconditionnellement leurs sorties, et **`WriteIf`**, **`WriteLineIf`**, **`Assert`** pour lesquelles les sorties sont soumises à conditions.

Quelques précisions s'imposent au sujet des classes **`Trace`** et **`Debug`**.



En mode *Debug*, tous les messages sont envoyés, tandis qu'en mode *Release*, qui est le mode de compilation des versions définitives, seuls les messages *Trace* subsistent. Bien entendu, le lancement du fichier exécutable ne produit aucune sortie dans ce cas, aucune redirection n'ayant été définie.

Les classes **`Trace`** et **`Debug`** partagent la même collection **`Listeners`**. Dès lors, l'ajout d'un écouteur à l'une ou l'autre de ces deux classes profitent à toutes les deux. Quand un seul écouteur doit être utilisé, tant en débogage qu'en exécution, il faut vider la collection avant d'y insérer l'écouteur souhaité.



Une ligne **`Trace.Listeners.Clear()`** avant l'ajout d'un écouteur vide la collection de ceux qui s'y trouve déjà, dont **`DefaultTraceListener`**.

Ici, **`DefaultTraceListener`** n'est plus le seul écouteur. Un **`TextWriterTraceListener`** a été ajouté et il dirige ses sorties vers la console. Un fichier ouvert aurait pu être désigné à la place de **`Console.Out`**.
Il résulte de cette modification que tous les messages sont maintenant envoyés à l'écran et dans la fenêtre *Sortie* en mode *Debug*. La version exécutable quant à elle n'enverra que les messages *Trace* à l'écran.

En ce qui concerne les méthodes de sorties des écouteurs, il faut encore savoir que la différence entre **`Write`** et **`WriteLine`** est la même que celle qui existe entre les méthodes de même nom qui servent à la sortie des données en application Console. La présence du suffixe **`If`** désigne seulement le mode conditionnel. Il existe encore des méthodes **`Fail`** et **`Assert`** qui ne présentent pas d'intérêts ici. Le lecteur peut se reporter à l'aide en ligne de son Visual Studio, et au site MSDN de Microsoft déjà référencé, pour compléter son information à ce sujet.

Voici l'essentiel ordinairement nécessaire au programmeur :

```
Dim FichierTrace As New TextWriterTraceListener("MonFichierMsg.txt")
Trace.Listeners.Clear()                                    ' Collection vidée
Trace.Listeners.Add(FichierTrace)                          ' Ce fichier à l'indice 0
…
Trace.Listeners(0).WriteLine(Date.Now & " MonMessage")     ' Date, heure, message
…
FichierTrace.Close()                                       ' Vider le buffer

Debug.WriteLine("Message inconditionnel")                  ' Trace ou Debug
Debug.WriteLineIf(Condition, "Message si condition réalisée")
```

La gestion des erreurs

Le débogage permet la mise au point d'une application mais ne la protège pas des erreurs qui surviennent en cours d'exploitation. Le traçage vers un fichier de l'exécution d'une version compilée peut informer le programmeur sur les circonstances des erreurs. Ces erreurs sont le plus souvent dues à une mauvaise utilisation de l'application. Le programmeur doit s'efforcer de prévoir toutes les maladresses dont peut être victime son application et l'en protéger.

Outre de programmer des contrôles de validité des données à traiter, le programmeur peut user d'un outil du langage qui permet la prévention et la récupération des erreurs sans programmation excessive.

Sans contrôle de validité des données,
ni récupération de l'erreur :

```
Private Sub UneProcedure()
Dim N1, N2, R As Integer
…
   R = N1 / N2
   Console.WriteLine(CType(R, String))
…
End Sub
```

L'affichage réalisé par cette procédure est le résultat de la division de **N1** par **N2** si tout va bien. Si **N2** vaut **0**, alors une fenêtre du **Just-In-Time Debugging** vient proposer un débogage dont l'utilisateur ne peut rien faire. Le programme est ensuite arrêté.
Si les variables étaient ici de type réel, la division ne produirait pas d'erreur, mais le résultat serait **+Infini** (si **N1** est positif). Ceci n'est guère satisfaisant, mais sans erreur. C'est au programmeur d'assurer le contrôle de validité des données.

Avec contrôle de validité des données,
sans récupération de l'erreur :

```
Private Sub UneProcedure()
Dim N1, N2, R As Integer
…
If N2 <> 0 then
  R = N1 / N2
  Console.WriteLine(CType(R, String))
Else
   ' … traitement approprié de l'erreur
End If
…
End Sub
```

Le programmeur, qui contrôle ici la validité de la variable **N2**, peut gérer l'erreur comme il le souhaite.
Soit ignorer l'appel de cette procédure par un **Exit Sub**, soit informer l'utilisateur par un message convivial, soit encore effectuer n'importe quel traitement qu'il juge utile.

Sans contrôle de validité des données,
avec récupération de l'erreur :

```
Private Sub UneProcedure()
Dim N1, N2, R As Integer
…
Try
  R = N1 / N2
Catch MonErr As Exception
   ' … traitement approprié de l'erreur
Finally
   ' … traitement à faire en tous cas
End Try
…
End Sub
```

Le programmeur, qui ne contrôle pas ici la validité de la variable **N2**, peut toutefois récupérer l'erreur et la gérer comme dans l'exemple précédent.
C'est le bloc de contrôle **Try … End Try** qui permet cette récupération.

Attention, la récupération d'une erreur sans sa gestion peut générer des résultats erronés. Dans l'exemple ci contre, aucun message ne vient perturber l'utilisateur et l'application ne s'arrête pas, mais la valeur affichée est erronée quand **N2** vaut **0**.

```
Private Sub UneProcedure()
Dim N1, N2, R As Integer
…
Try
  R = N1 / N2
Catch
  Console.WriteLine(CType(R, String))
Finally
   ' … traitement à faire en tous cas
End Try
…
End Sub
```

Les mots clés de cet outil de récupération des erreurs sont **Try**, **Catch**, **Finally**, **When**, **Exit Try** et **End Try**. L'emploi de **Exit Try** permet de ne pas exécuter les autres instructions du bloc **Try**. Le mot clé **Catch** permet la désignation éventuelle du type d'erreur à prendre en compte et le mot **When** permet d'en préciser la condition d'interception. Plusieurs instructions **Catch** peuvent se succéder et il convient dans ce cas de les classer par type d'erreur du plus précis vers le moins précis.

```
Private Sub UneProcedure()
Dim N1, N2, R As Integer
…
Try
      R = N1 / N2
      If N1 = 1 Then Exit Try              ' Il peut être opportun de ne pas exécuter
      ' sinon suite des traitements        ' la suite
Catch MonErr As DivideByZeroException      ' Les Catch, à partir du plus précis …
      ' … traitement approprié
Catch MonErr As ArithmeticException
      ' … traitement approprié
Catch MonErr As Exception When N1 = 1      ' Attention : pas de récupération si N1 <> 1
      ' … traitement approprié
Catch                                      ' … jusqu'au plus vague (Erreur indéterminée)
      ' … traitement approprié
Finally                                    ' Le bloc Finally est facultatif, mais s'il
      ' … traitement à faire en tous cas   ' est présent, ses traitements sont exécutés
End Try                                    ' même en cas d'erreurs, ou d'Exit Try, ou de
…                                          ' Return (quand Try est dans une fonction)
End Sub
```

Générer des erreurs

Une instruction particulière, **Throw**, permet au programmeur de générer les exceptions appropriées aux erreurs qu'il définit.

Voici son fonctionnement par un exemple extrait d'une application **MonApplication**.

Une procédure à risque, **Calcul**, est appelée à partir d'un bloc **Try … End Try** de la procédure **UneProcedure**. Le programmeur, détecte la situation d'erreur dans **Calcul** par un test de validité de l'opérande **Arg2** et génère lui-même une erreur avec un message adéquat à l'intention de la procédure appelante.
Dans **Calcul**, c'est la ligne **Throw New ApplicationException("MonErreur")** qui retourne l'erreur de type **ApplicationException** avec le message **MonErreur**.
Dans cet exemple, "***MonErreur dans MonApplication***" est la phrase affichée par le bloc **Try** de la procédure appelante quand sa variable **Terme2** vaut **0**.

```
Private Sub UneProcedure()
    Dim Resultat, Terme1, Terme2 As Integer
    ' … acquisition des valeurs pour Terme1 et Terme2
    Try
        Calcul(Resultat, Terme1, Terme2)
        MessageBox.Show(Resultat)
    Catch UneErreur As ApplicationException
        MessageBox.Show(UneErreur.Message & " dans " & UneErreur.Source)
    End Try
 End Sub

Private Sub Calcul(ByRef Rep As Integer, ByVal Arg1 As Integer,
                                                    ByVal Arg2 As Integer)
    If Arg2 = 0 Then
        Throw New ApplicationException("MonErreur")
    Else
        Rep = Arg1 / Arg2
    End If
End Sub
```