1 Module A.6 : Exceptions et diagnostics

1.1 Gestion des exceptions

L'exécution d'un programme peut mal se dérouler à cause d'une entrée anormale (ex. : Saisie d'un caractère qui n'est pas un entier) ou à l'échec d'un accès à un périphérique d'entrée/sortie. Dans cette situation, un programme lève une exception. Si une exception n'est pas gérée, elle provoque l'arrêt de l'exécution d'un programme (en mode Debug, une boîte de dialogue s'ouvre en indiquant l'exception puis arrête l'exécution). Dans la plupart des situations, nous ne voulons pas d'un tel comportement (l'erreur de saisie dans un formulaire internet fermerait le navigateur). Lors de l'écriture d'un programme, la gestion des exceptions est une étape importante à ne pas négliger car les choses peuvent mal se passer.

Il existe une manière de gérer les exceptions en C#. Cette manière repose sur trois mots clés, **try**, **catch**, **finally** qui sont délimiteurs de blocs de code.

— Bloc **try** : définition du bloc de code sujet à l'action d'erreurs et à leur interception. Il est toujours suivi par un bloc **catch**, **finally** ou les deux.

— Bloc **catch** : définition du bloc de code exécuté si une exception d'un certain type est levée dans le bloc **try**.

— Bloc **finally** : définition du bloc de code réalisé qu'une exception soit levée ou non (pratique pour les fermetures de flux de lecture/écriture dans un fichier, accès à une base de données). La seule exception à son exécution arrive en cas de boucle infinie et d'arrêt brutal du programme.

Exemple :

```
1  static int Calcul (int x)
2  {
3    return 10 / x;
4  }
5  ...
6  try
7  {
8    int resultat = Calcul(0);
9    Console.WriteLine(resultat);
10 }
11 catch (DivideByZeroException ex)
12 {
13   Console.WriteLine("x ne peut pas être égal à 0");
14 }
15 finally
16 {
17   Console.WriteLine("Je suis toujours exécuté");
18 }
```

Exceptions courantes :

Type d'exception	Type de base	Description
Exception	**Object**	Base de toutes les exceptions
SystemException	**Exception**	Base pour toutes les erreurs générées lors de l'exécution.
IndexOutOfRangeException	**SystemException**	Mauvaise indexation d'un tableau.
NullReferenceException	**SystemException**	Accès au contenu d'un objet null.
ArgumentException	**SystemException**	Base pour les erreurs d'arguments.
ArgumentNullException	**ArgumentException**	Argument null sur une fonction ne l'acceptant pas.
ArgumentOutOfRangeException	**ArgumentException**	Argument hors de la plage donnée de valeurs.

Lors de la levée d'une exception, si elle ne se trouve pas dans un bloc **try**, elle remonte la pile des appels de fonctions jusqu'à trouver un bloc **try**. Si aucun bloc **try** ne peut gérer l'exception, le programme s'arrête. Ce fonctionnement permet de séparer la gestion des exceptions de l'implémentation des règles fonctionnelles.

Plusieurs blocs **catch** peuvent suivre le même bloc **try** afin de traiter différents types d'exceptions. Lorsqu'une exception est levée, il y a parcours séquentiel des blocs jusqu'à trouver un bloc qui correspond à l'exception levée (exception même ou classe de base de l'exception). Un seul bloc **catch** est exécuté, si l'on veut que l'exception soit gérée dans un autre bloc **catch**, il faut utiliser le mot clé **throw** afin de lever la même exception. Les blocs **catch** doivent aller des exceptions les plus spécifiques aux exceptions plus générales. L'exception la plus générale est **Exception**.

Exemple :

```
1  try
2  {
3    string entree = Console.ReadLine();
4    byte b = byte.Parse(entree);
5  }
6  catch (IndexOutOfRangeException ex)
7  {
8    Console.WriteLine("Rentrez au moins un argument!");
9  }
10 catch (FormatException ex)
11 {
12   Console.WriteLine("Ce n'est pas un nombre!");
13 }
14 catch (OverflowException ex)
15 {
16   Console.WriteLine("Le nombre fourni dépasse la capacité du byte!");
17 }
```

Exception contient plusieurs propriétés importantes dont héritent chacun des types d'exception :

— **StackTrace** : **string** représentant l'ensemble des méthodes qui ont été appelées depuis la levée de l'exception jusqu'à au bloc **catch**.

— **Message** : **string** avec une description de l'erreur.

— **InnerException** : Exception interne qui a provoqué l'exception.

Un bloc **catch** a accès au contenu de l'exception qu'il a attrapé. Depuis C# 6, on peut rajouter une condition au bloc **catch** avec le mot clé **when**. Cela permet de séparer la réponse à une même exception dans des situations différentes.

Exemple :

```
1 catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
2 { ... }
3 catch (WebException ex) when (ex.Status == WebExceptionStatus.SendFailure)
4 { ... }
```

On peut lever soit même des exceptions à l'aide du mot clé **throw** dans notre code si un comportement est jugé non conforme.

Exemple :

```
1 static void AffichageAnnee(int annee)
2 {
3   if (annee < 0 || annee > 9999)
4     throw new ArgumentOutOfRangeException("L'année donnée est invalide : " +
          annee);
5
6   Console.WriteLine("L'an " + annee + " du calendrier Grégorien.");
7 }
```

1.2 Outils de débogage

1.2.1 Fondamentaux

Le débogage est une importante partie du processus de développement. Le langage C# a de nombreuses fonctionnalités pour fixer et résoudre les bugs.

On peut ajouter des points d'arrêts où le programme va s'arrêter en mode débogage. Une fenêtre permet de voir les différentes variables existantes au niveau du point d'arrêt et leurs contenus.

1.2.2 Les directives de précompilation

On peut changer le comportement du compilateur avec des directives qui commencent par #. Par exemple, #if, #else if, #else et #define. La dernière directive permet de déclarer des symboles qui seront testés par les trois premières directives.

1.3 Outils de diagnostics

Les outils courants pour le débogage d'un programme ont été présentés dans la section précédente. Il est important de corriger les erreurs, mais il est sans doute fort sage de prévenir les erreurs balisant le code à exécuter. On présente dans cette section les outils de diagnostics standards utilisés en C#.

1.3.1 Trace d'exécution avec Debug et Trace

Ces classes sont utiles pour tracer l'exécution d'un programme et fournissent des méthodes pour générer une trace lorsque le programme s'exécute. Les méthodes WriteLine et WriteLineIf permettent d'écrire sans ou avec condition. Les méthodes Indent et Unindent permettent de rajouter ou d'enlever un cran d'indentation.

Exemple :

```
1  Debug.WriteLine ("Démarrage du programme");
2  Debug.Indent ();
3  Debug.WriteLine ("A l'intérieur d'une fonction");
4  Debug.Unindent ();
5  Debug.WriteLine ("A l'extérieur d'une fonction");
6  string client = "Rob";
7  Debug.WriteLineIf (string.IsNullOrEmpty (client), "Le nom est vide");
8
9  // Le résultat ;
10 // Démarrage du programme
11 //   A l'intérieur d'une fonction
12 // A l'extérieur d'une fonction
```

Ces deux classes remplissent les mêmes fonctions, mais dans deux modes de compilation différents :

— Mode Debug : lorsque l'on développe et corrige sur notre poste. Les points d'arrêts sont pris en compte (**Debug**).

— Mode Release : lorsque le programme tourne en production (**Trace**).

Debug et **Trace** permettent de facilement tester des assertions. Une assertion est une affirmation que l'on pense être vraie. On peut ajouter du code pour tester l'assertion à l'aide de la méthode Assert. Si l'assertion est fausse, une boîte de dialogue est ouverte pour signaler un problème. On peut choisir d'arrêter le programme ou le continuer. Si l'assertion est vraie, rien ne se passe.

Exemple :

```
1  string nomClient = "Rob";
2  Debug.Assert (!string.IsNullOrWhiteSpace (nomClient)); // Réussite.
3
4  nomClient = "";
5  Debug.Assert (!string.IsNullOrWhiteSpace (nomClient)); // Echec.
```

On peut moduler l'importance du message avec les méthodes TraceInformation, TraceWarning et TraceError.

Par défaut, les traces s'écrivent dans la console qui est la sortie standard. On peut bien évidemment changer cela. Un champ de **Trace** est Listeners. On peut ajouter des écouteurs de trace (listeners) et en supprimer. Par exemple, on peut écrire la trace dans un fichier avec**TextWriterTraceListener**.

Exemple :

```
1  Stream fichier = File.Create("fichierTexte.txt");
2  TraceListener textListener = new TextWriterTraceListener (fichier);
3  Trace.Listeners.Add (textListener);
```

```
// Les traces s'affichent à présent dans le fichier fichierTexte.txt.

Trace.TraceInformation ("C'est un message d'information");
Trace.TraceWarning ("C'est un message d'avertissement");
Trace.TraceError ("C'est un message d'erreur");

Trace.Listeners.Remove (textListeners);
// Les traces ne s'écriront plus dans le fichier fichierTexte.txt.
```

Il existe également des outils plus poussés fournis par les classes **TraceSource** et **TraceSwitch**. Ces outils dépassent largement le cadre de cette formation.

1.3.2 Mesurer le temps d'exécution

La classe **Stopwatch** permet de mesurer le temps d'exécution dans un programme de manière extrêmement fine. La résolution est inférieure à la microseconde. La durée mesurée est fournie notamment avec la propriété ElapsedMilliseconds.

Le tableau suivant fournit une liste de méthodes accessibles pour **Stopwatch**.

Méthodes	**Explications**
void Start ()	Démarre ou reprend la mesure du temps écoulé pour un intervalle.
void Reset ()	Arrête la mesure d'intervalle de temps et remet la durée à zéro.
void Stop ()	Cesse de mesurer la durée pour un intervalle.
Stopwatch StartNew ()	Initialise une nouvelle instance de **Stopwatch**, initialise le temps écoulé à 0 et commence à mesurer le temps écoulé.

Les instances de **Stopwatch** sont utiles pour comparer le temps d'exécution de méthodes.

Exemple :

```
// Création, initialisation et début mesure d'un intervalle de temps.
Stopwatch s = Stopwatch.StartNew();
MethodeBrute();
s.Stop();  // Arrêt mesure.
long tempsBrut = s.ElapsedMilliseconds;

s.Reset(); // Remise à zéro.
s.Start(); // Lancement mesure.
MethodeOptimisee();
s.Stop();
long tempsOptimise = s.ElapsedMilliseconds;
```