C# .NET

Marc Chevaldonné
marc.chevaldonne@u-clermont1.fr
http://marc.chevaldonne.free.fr/ens_rech/Csharp_XML_GI_2010_2011.html

Année scolaire 2010 - 2011

C# avancé

Les événements

- objectif : simplifier le pattern broadcaster/subscriber tout en apportant plus de sécurité

- le pattern broadcaster/subscriber avec les délégués est dangereux car un subscriber peut modifier l'écoute d'un autre subscriber
  - => solution : les événements
  - => tutoriel 1 : introduction aux événements
  - cf. exemple 70

- il est conseillé de suivre le pattern standard utilisé dans la plateforme .NET et par tous les codeurs .NET (dignes de ce nom)
  - => tutoriel 2 : standard event pattern
  - cf. exemple 71

- très utilisés dans les interfaces graphiques fenêtrées
  - cf. exemple 72

C# avancé

Les exceptions

- de nombreuses exceptions sont déjà gérées par le système
  - index out of range
  - null reference type
  - ...
  - cf. exemple 73
- rappel : on peut en éviter de nombreuses à l'aide de TryParse à la place de Parse
  - cf. exemple 74
    -

C# avancé

Les exceptions

- les instructions : cf. exemple 75
  - try :
    - si le code dans le bloc try génère une exception => bloc catch
    - sinon on continue
  - catch :
    - indique quel type d'exception il faut traîter
  - finally :
    - toujours exécuté (après le try (et les catch), exception ou non)

- rethrow d'exceptions (cf. exemple 76)

- on peut créer sa propre classe d'exception (cf. exemple 77)

C# avancé

Directives préprocesseur

- fournir des informations additionnelles au compilateur à propos de régions dans le code
- les plus connues sont les directives conditionnelles
    - pour inclure (exclure) des régions de code dans (de) la compilation
- VisualStudio permet d'entrer les symboles de compilation conditionnelle dans les propriétés de projet

```
#define DEBUG
class UneClasse
{
   int a;
   void UneMéthode()
   {
      #if DEBUG
      Console.WriteLine(«Test : a = {0}», a);
      #endif
   }
   ...
}
```

C# avancé

Directives préprocesseur

- Quelques directives préprocesseur

#define symbole	définit le symbole
#undef symbole	défait la définition du symbole
#if symbole [operator symbole2]...	test entre symbole et symbole2 opérateurs : == != && || suivi éventuellement de #else #elif #endif
#endif	termine une directive conditionnelle
#else	code exécuté après un #endif
#elif symbole [operator symbole2]...	combine #else et #if branches
#region nom	marque le début d'une région
#end nom	marque la fin d'une région

C# avancé

Directives préprocesseur - Attribut conditionnel

- L'attribut `Conditional` indique au compilateur d'ignorer tous les appels d'une classe ou d'une méthode si le symbole n'est pas défini
- le compilateur entoure implicitement les appels de la classe ou de la méthode avec des directives `#if`

```
[Conditional («TEST»)]
static void MéthodeDeDebug(string message_debug)
{
    ...
}
```

- cf. exemple 78

C# avancé

classes Debug et Trace

- Debug et Trace sont des classes statiques (très similaires) qui permettent la gestion de méthodes classiques pour le debug :
  - logging
  - assertion

- Debug est utilisée en mode ... debug
  - toutes ses méthodes sont définies avec l'attribut [Conditional(«DEBUG»)]
- Trace est utilisée en modes debug et release
  - toutes ses méthodes sont définies avec l'attribut [Conditional(«TRACE»)]

- Par défaut, VisualStudio définit :
  - les symboles DEBUG et TRACE en mode debug
  - le symbole TRACE en mode release

C# avancé

classes Debug et Trace

- méthodes de Debug et Trace (cf. exemple 78)
  - Write
  - WriteLine
  - WriteIf
- méthodes de Trace (cf. exemple 78)
  - TraceInformation
  - TraceWarning
  - TraceError
- méthodes de Debug et Trace pour les assertions (cf. exemple 78)
  - Fail
  - Assert
- TraceListener (cf. exemple 78)
  - fichier
  - XML
  - Console...

C# avancé

bien debugger...

- Quelques conseils pour améliorer le debug :

- + réécrire ToString
- + utiliser l'attribut conditionnel DEBUG : ne faire des affichages Console qu'à travers une méthode avec l'attribut conditionnel DEBUG ou à travers la classe Trace (+ TraceListener)
- + utiliser les classes Debug et Trace (en particulier Write, WriteIf, Assert et Fail)

- - éviter d'écrire soi-même les directives préprocesseurs #if...
- --- ne jamais faire d'affichage Console depuis ses classes en dehors des règles précédentes !!!

Disposal et Ramasse-miettes

Garbage Collector : comment ça marche ?

- Le Garbage Collector (GC) contrôle la mémoire managée pour vous (memory leaks, pointeurs, ...)
- Vous (les développeurs de classes et les clients de ces classes) êtes responsables du nettoyage des ressources non managées :
  - fichiers
  - connections aux bases de données
  - COM objects
  - certains objets graphiques
  - ...

- Le GC détecte si un objet est accessible en traversant l'arbre de l'application depuis la racine (détecte ainsi les références circulaires et les relations complexes entre objets)
- Tout ce qui ne peut pas être atteint depuis l'application est «mort» et sera libéré par le GC

Disposal et Ramasse-miettes

Garbage Collector : comment ça marche ?

- Le GC fonctionne dans un thread spécifique
- Le GC réarrange le tas à chaque passage pour n'occuper que des blocs mémoire contigus
- 2 outils pour aider le développeur à libérer la mémoire non managée :
  - les finaliseurs :
    - un mécanisme assurant qu'un objet pourra toujours libérer les ressources non managées
    - inconvénient : coût en termes de performance
  - l'interface IDisposable :
    - méthode plus légère pour libérer la mémoire non managée et plus performante
    - inconvénient : le client doit penser à l'appeler (donc pas sûr à 100%)

Disposal et Ramasse-miettes

Garbage Collector : comment ça marche ?

- Les finaliseurs sont appelés par le GC après avoir été «mis à la poubelle»
  - environ la même syntaxe que les destructeurs en C++
  - c'est le moyen le plus sûr de nettoyer la mémoire non managée, mais :
  - on ne sait pas quand... contrairement au C++ (ctor / destor)
  - quand le GC veut nettoyer un objet et découvre qu'il a un finaliseur, il ne libére pas tout de suite :
    - le GC place tous les objets qui ont un finaliseur dans une queue
    - les finaliseurs sont appelés dans un autre thread et pendant ce temps le GC continue son travaille
    - c'est seulement au cycle suivant que le GC libère donc ces objets ayant un finaliseur (qui restent donc plus longtemps)

Disposal et Ramasse-miettes

Garbage Collector : comment ça marche ?

- En fait, les objets ayant un finaliseur ne restent pas qu'un cycle de plus...
- Le GC utilise un système de générations :
  - génération 0 : objets créés depuis la dernière opération du GC
  - génération 1 : objets ayant survécu à une opération du GC
  - génération 2 : objets ayant survécu à deux opérations ou plus du GC
- permet de gérer différemment les variables locales des références plus durables :
  - les variables locales sont généralement de génération 0
  - les membres et les variables globales entrent rapidement en génération 1, et éventuellement en génération 2

- Pour améliorer ses performances :
  - le GC examine à chaque cycle les objets de génération 0,
  - 1 cycle sur 10, il examine les objets de génération 0 ou 1,
  - 1 cycle sur 100, il examine tous les objets

- Un objet ayant un finaliseur peut donc rester 9 cycles de plus qu'un objet sans finaliseur, ou pire : 90 cycles de plus !

Disposal et Ramasse-miettes

Garbage Collector : comment ça marche ?

- Les finaliseurs restent indispensables car ils sont le seul moyen sûr de libérer la mémoire non managée...
- ... mais la solution de l'implémentation de IDisposable peut permettre de n'avoir recours aux finaliseurs que dans des cas précis, et d'éviter le coût en performances

Disposal et Ramasse-miettes

Garbage Collector : comment ça marche ?

- L'interface IDisposable ne possède qu'une méthode : void Dispose( )
- L'appel de Dispose libère les ressources managées et non managées puis préviens le GC qu'il n'est pas nécessaire d'appeler le finaliseur
- En conséquence :
  - si le client appelle Dispose, les ressources sont libérées plus rapidement car le finaliseur n'est pas utilisé
  - si le client n'appelle pas Dispose, les ressources seront libérées (mais beaucoup plus tard) lors de l'appel du finaliseur
  - Conseils :
    - si vous êtes client, appelez Dispose sur toutes les classes implémentant IDisposable lorsque vous n'avez plus besoin des instances
    - si vous êtes développeur d'une classe avec des ressources non managées, implémentez l'interface IDisposable

Disposal et Ramasse-miettes

Garbage Collector : comment ça marche ?

- Comment utiliser une classe implémentant IDisposable ?

- L'appel de la méthode Dispose permet de libérer les ressources non managées, mais s'il y a une méthode entre l'instanciation de la classe et l'appel de Dispose qui lance une exception, Dispose ne sera pas appelée et c'est donc le finaliseur qui le sera
- => mauvaise méthode

- L'instanciation et l'utilisation de la classe implémentant IDisposable se fait dans un bloc try, et l'appel de Dispose dans le bloc finally associé
- => Dispose est ainsi toujours appelée => bonne méthode

- Puisque c'est une bonne méthode, .NET propose l'utilisation de l'instance de la classe implémentant IDisposable dans un bloc using
- => équivalent à try/finally + Dispose
- => sauf qu'on n'a même pas besoin d'appeler Dispose (automatique)

Disposal et Ramasse-miettes

Garbage Collector : comment ça marche ?

- Comment implémenter correctement une classe avec des ressources non managées ?

- Il existe un pattern qu'il est préférable de respecter : classe de base

```
public class LaClasseDeBase : IDisposable
{
   private bool disposed = false;
   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }
   protected virtual void Dispose(bool isDisposing)
   {
      if(disposed) { return; }
      if(isDisposing) { /* libérer les ressources managées ici*/ }
      //libérer les ressources non managées ici
      disposed = true;
   }
}
```

Disposal et Ramasse-miettes

Garbage Collector : comment ça marche ?

- Comment implémenter correctement une classe avec des ressources non managées ?

- Il existe un pattern qu'il est préférable de respecter : classe fille

```
public class LaClasseDérivée : LaClasseDeBase
{
   private bool disposedClasseDérivée = false;

   protected virtual void Dispose(bool isDisposing)
   {
      if(disp disposedClasseDérivée osed) { return; }
      if(isDisposing) { /* libérer les ressources managées ici*/ }
      //libérer les ressources non managées ici

      //on laisse le soin à la classe de base d'appeler
      //GC.SuppressFinalizer
      base.Dispose(isDisposing);

      disposed = true;
   }
}
```