Les interfaces

Pour manipuler différents types d'objets possédant des fonctionnalités similaires, il est possible d'utiliser un contrat définissant les données et les comportements communs à ces types. En C#, on appelle ce contrat une interface.

Une interface est un type qui possède uniquement des membres publics. Ces membres ne possèdent pas d'implémentation : ce sont les types qui respecteront ce contrat qui implémenteront chacun des membres.

Les interfaces représentent une couche d'abstraction particulièrement utile pour rendre une application modulaire, puisqu'elles permettent d'utiliser n'importe quelle implémentation pour un même contrat.

1. Création

Les interfaces sont déclarées d'une manière similaire aux classes, les différences étant les suivantes :

- Il faut utiliser le mot-clé `interface` au lieu de `class`.
- Seuls les membres destinés à être visibles publiquement doivent être définis dans l'interface, et aucun modificateur de visibilité n'est autorisé sur ces membres.
- Aucun des membres de l'interface ne peut avoir d'implémentation.

La syntaxe générale pour la création d'une interface est la suivante :

```
<modificateur d'accès> interface <nom> [: interfaces de base ]
{

}
```

Par convention, le nom des interfaces en C# commence par un I (i majuscule).

Considérons le cas de la lecture de fichiers audio. Il est tout à fait possible de lire des fichiers aux formats MP3, WAV, OGG, MWA, et bien d'autres encore. La lecture de chacun de ces formats nécessitant un décodage particulier, il semble pertinent de créer une classe pour chacun des types de fichiers supportés par une application : `LecteurAudioMp3`, `LecteurAudioOgg`, etc.

Il est fort probable que ces classes soient très similaires et doivent être utilisées dans les mêmes circonstances, ce qui fait donc de cet ensemble un candidat parfait à la création d'une interface commune aux différents types.

Une interface possible pour ces différentes classes pourrait être la suivante :

```
public interface ILecteurAudio
{
    bool LectureEnCours { get; }

    bool EstEnPause { get; }

    void Lire(string cheminFichier);

    void Pause();

    void ReprendreLecture();
```

```
    void Stop();
}
```

`LectureCours` et `EstEnPause` sont deux propriétés pour lesquelles on indique que les classes implémentant l'interface doivent fournir au minimum la lecture publique. Toutes les méthodes définies dans l'interface seront elles aussi présentes dans toute classe implémentant l'interface.

2. Utilisation

Pour utiliser une interface, il faut tout d'abord déclarer la relation entre classe et interface. Pour ceci, on utilise la même syntaxe que pour l'héritage :

```
<modificateur d'accès> class <nom> [:interface1, interface2...]
{
}
```

La déclaration de la classe `LecteurMp3` serait donc la suivante :

```
public class LecteurMp3 : ILecteurAudio
{
}
```

Visual Studio indique que la compilation du code de l'interface et de la classe, à ce stade, produit très exactement six erreurs.

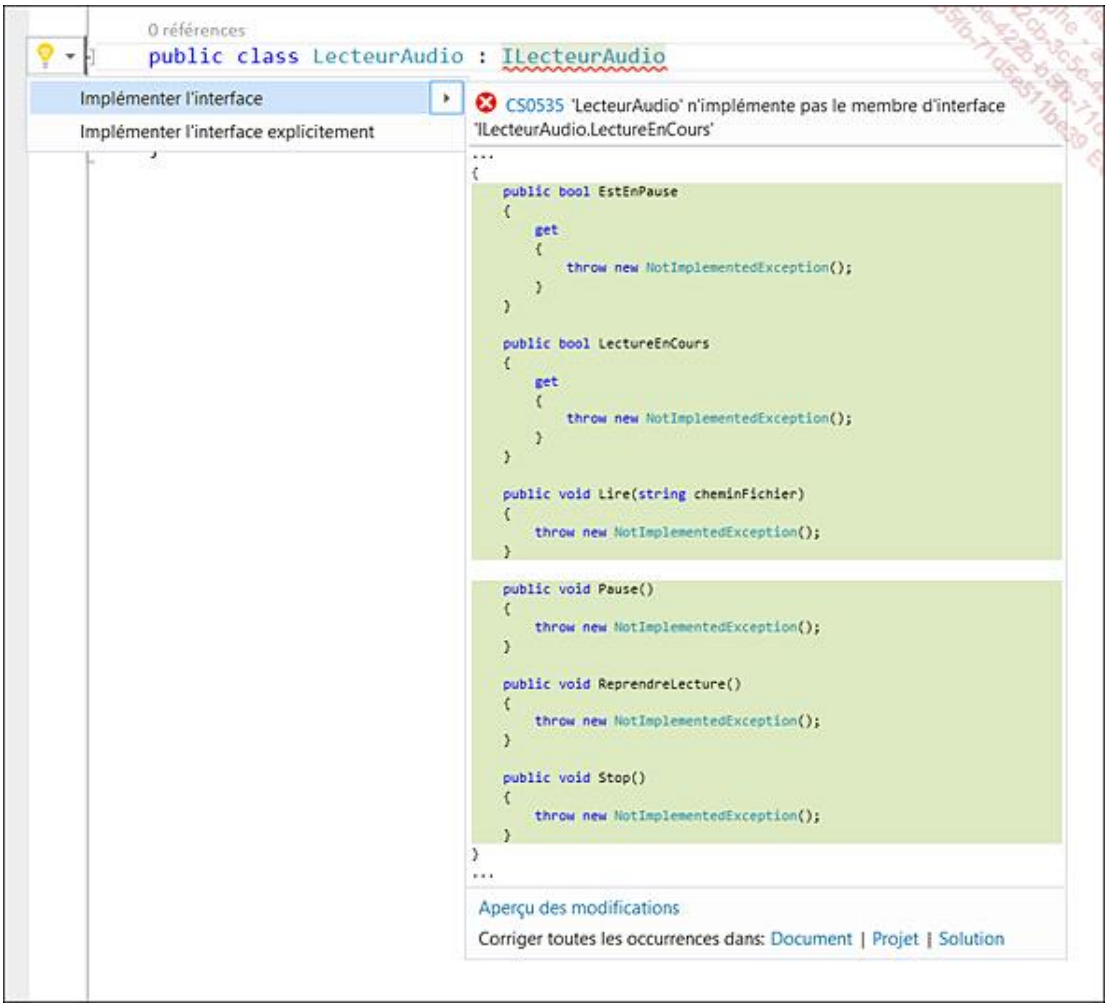
Le compilateur C# indique très clairement que les membres d'interface n'ont pas été implémentés dans la classe `LecteurMp3`. Il est possible de les implémenter de deux manières : implicitement ou explicitement.

a. Implémentation implicite

Dans la grande majorité des cas, les interfaces sont implémentées de manière implicite. Ceci signifie simplement que chacun des membres définis dans l'interface existe dans les classes implémentant l'interface.

Chacun des membres doit être implémenté avec une visibilité publique et, dans le cas de méthodes, avec la même signature que celle définie par l'interface.

L’outil de refactorisation de Visual Studio peut simplifier cet exercice en créant un squelette d’implémentation implicite. Il offre également la possibilité de prévisualiser les modifications correspondant à la génération de ce squelette.



Utiliser cet outil génère le code suivant :

```
class LecteurMp3 : ILecteurAudio
{
    public bool LectureEnCours
    {
        get { throw new NotImplementedException(); }
    }

    public bool EstEnPause
    {
        get { throw new NotImplementedException(); }
```

```
    }

    public void Lire(string cheminFichier)
    {
        throw new NotImplementedException();
    }

    public void Pause()
    {
        throw new NotImplementedException();
    }

    public void ReprendreLecture()
    {
        throw new NotImplementedException();
    }

    public void Stop()
    {
        throw new NotImplementedException();
    }
}
```

Une fois cette opération effectuée, chacun des membres de l'interface `ILecteurAudio` possède une implémentation (non fonctionnelle en l'état) dans la classe `LecteurMp3`.

b. Implémentation explicite

Il peut parfois arriver qu'une classe implémente plusieurs interfaces dont certains membres partagent la même signature. Dans ce cas, implémenter les interfaces implicitement amène à partager la même implémentation pour toutes les interfaces concernées.

Il peut pourtant être nécessaire que la classe comporte une implémentation de membre par interface. L'implémentation explicite est une réponse à cette problématique particulière.

Considérons les deux interfaces suivantes :

```
public interface ILecteurAudio
{
    void Lire(string cheminFichier);
}
```

```
public interface ILecteurVideo
{
    void Lire(string cheminFichier);
}
```

Pour implémenter ces deux interfaces dans une classe `LecteurMultimedia`, il faut faire précéder le nom de chacun des membres spécifiques par le nom de l'interface associée et l'opérateur point.

```
public class LecteurMultimedia : ILecteurAudio, ILecteurVideo
{
    void ILecteurAudio.Lire(string cheminFichier)
    {
           //Démarrage de la lecture audio
    }

    void ILecteurVideo.Lire(string cheminFichier)
    {
           //Préparation de l'affichage du flux vidéo
           //Démarrage de la lecture audio & vidéo
    }
}
```

Visual Studio permet aussi de générer ces implémentations explicites via son outil de refactorisation, avec prévisualisation du code généré :