Gabriel T. St-Hilaire

Interface et Généricité

Concept orienté objet

420-3A4-ST
Structure de données et algorithmie avancée

Exemple : Comment régler ce problème ??

```
public class Maison
{
    public void DessinerMaison() {
        Console.WriteLine("⌂");
    }
}
```

```
var liste = new LinkedList<Forme>();
liste.AddLast(new Cercle(10, 20));
liste.AddLast(new CercleIncroyable(10, 20));
liste.AddLast(new Maison());

foreach (var dessinable in liste)
{
    dessinable.Dessiner();
}
```

Interface

Interface : Ensemble des **moyens qui permettent la connexion et l'échange d'information** entre les différents dispositifs ou systèmes, entre le matériel et un système, ou entre un système et son utilisateur.

- Office québécois de la langue française, 2020

Interface

Dans notre cas, on parlera du concept appliqué au paradigme orienté objet.

Une interface est donc un **ensemble de règles** que différentes classes doivent suivre.

Ceci nous permet de créer des objets différents tout en **étant sûrs qu'ils ont certaines fonctionnalités en commun**.

Exemple : interface.ipynb

```
interface IDessinable
{
    void Dessiner();
}
```

```
public abstract class Forme : IDessinable
{
    protected int PositionX { get; }
    protected int PositionY { get; }

    protected Forme(int positionX, int positionY)
    {
        PositionX = positionX;
        PositionY = positionY;
    }

    public abstract void Dessiner();
}
```

Exemple : interface.ipynb

```
public class Cercle : Forme
{
    public Cercle(int positionX,
        int positionY) : base(positionX, positionY)
    {
    }

    public override void Dessiner()
    {
        Console.WriteLine("○ à la position " +
                          PositionX + ", " + PositionY);
    }
}
```

```
public class CercleIncroyable : Cercle
{
    public CercleIncroyable(int positionX,
        int positionY) : base(positionX, positionY)
    {
    }

    public override void Dessiner()
    {
        Console.WriteLine("◍ à la position " +
                          PositionX + ", " + PositionY);
    }
}
```

Exemple : interface.ipynb

```
public class Maison : IDessinable
{
    public void Dessiner()
    {
        Console.WriteLine("⌂");
    }
}
```

```
var liste = new LinkedList<IDessinable>();
liste.AddLast(new Cercle(10, 20));
liste.AddLast(new CercleIncroyable(10, 20));
liste.AddLast(new Maison());

foreach (var dessinable in liste)
{
    dessinable.Dessiner();
}
```

Abstraction VS Interface

Une précision importante concernant le mot « interface » : il y a le « **concept** » d'une interface et il y a le « **mot clé** » `interface`.

On peut programmer une interface sans le mot clé `interface` … !

On peut dire qu'une interface est **ce qui permet l'échange d'information entre deux entités**.
C'est le contrat.

Une classe abstraite est donc une interface conceptuellement. De l'autre côté, une interface est une classe abstraite où tout est abstrait et public.

Abstraction VS Interface

Habituellement, on utilisera la **classe abstraite** quand il y a une **relation** entre les classes. Lorsqu'on peut dire « est une sorte de ».

Et on utilisera les **interfaces** quand on souhaite établir une relation entre des classes qui ne sont pas liés conceptuellement. Si on souhaite spécifier un **comportement commun** par exemple.

Interfaces standards

C# définit déjà plusieurs interfaces standards qu'on peut utiliser.

En travaillant avec les structures de données plus tard dans la session, on en verra quelques-unes.

Exemple : `ICollection<T>`

https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.icollection-1?view=net-7.0

Implémentation multiple

Contrairement à l'héritage, il est possible d'implémenter plusieurs interfaces.

La classe doit alors respecter le contrat de l'ensemble des interfaces.

```
public class Chien : IDessinable, ICloneable
```

Exemple : interface.ipynb

```
public class Chien : IDessinable, ICloneable
{
    private string Nom { get; }

    public Chien(string nom)
    {
        Nom = nom;
    }

    public void Dessiner()
    {
        Console.WriteLine(Nom + " 🐐");
    }

    public object Clone()
    {
        return new Chien(Nom);
    }
}
```

```
var chien = new Chien("Fido");
chien.Dessiner();

var chienClone = chien.Clone();
((Chien)chienClone).Dessiner();
```

```
(chien is Chien).Display();
(chien is IDessinable).Display();
(chien is ICloneable).Display();
```

https://learn.microsoft.com/en-us/dotnet/api/System.ICloneable?view=net-7.0

Interface - Résumé

- Une interface est un ensemble de règles que différentes classes doivent suivre;
- L'interface est un supertype;
- Par convention, en C#, on débute le nom de l'interface par un « I » majuscule;
- Les méthodes ne sont pas « override » dans la sous-classe;
- Dans une interface, tous les membres et les méthodes sont publics et abstraites par défaut;
- L'implémentation multiple est possible.

Classes et méthodes génériques

Les génériques introduisent le concept de **paramètres de type** qui permettent de concevoir des classes et des méthodes qui **diffèrent la spécification jusqu'à ce que la classe ou la méthode soit déclarée et instanciée par le code**.

On peut appliquer ce concept sur les classes (et interfaces) et sur les méthodes.

Les génériques sont largement utilisés dans les structures de données qu'on utilisera cette session.

Méthodes génériques

Plutôt que d'ajouter plusieurs surcharges dans une classe pour couvrir différents types de paramètre, il peut parfois être utile d'utiliser les méthodes génériques.

À noter que d'avoir plusieurs surcharges dans une classe peut être souhaité. Il faut plutôt **définir à quel moment la définition de surcharge est inutile et s'il est plus approprié d'utiliser une méthode générique**.

Exemple : interface.ipynb

```
static void Swap(ref int lhs, ref int rhs)
{
    int temp;

    temp = lhs;
    lhs = rhs;
    rhs = temp;
}

static void Swap(ref string lhs, ref string rhs)
{
    string temp;

    temp = lhs;
    lhs = rhs;
    rhs = temp;
}

// Autres surcharges
```

```
var intA = 1;
var intB = 2;
Swap(ref intA, ref intB);
Console.WriteLine(intA + " " + intB);

var stringA = "1";
var stringB = "2";
Swap(ref stringA, ref stringB);
Console.WriteLine(stringA + " " + stringB);
```

Exemple : interface.ipynb

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;

    temp = lhs;
    lhs = rhs;
    rhs = temp;
}

var intA = 1;
var intB = 2;
Swap(ref intA, ref intB);
Console.WriteLine(intA + " " + intB);

var stringA = "1";
var stringB = "2";
Swap(ref stringA, ref stringB);
Console.WriteLine(stringA + " " + stringB);
```

Classes et interfaces génériques

Toujours dans l'objectif de permettre de paramétrer les types, il est également possible d'appliquer ce concept aux classes et aux interfaces.

Le type devient donc généralisé dans l'ensemble de l'instance.

L'utilisation la plus courante des classes génériques concerne les collections. Les opérations telles que l'ajout et la suppression d'éléments sont effectuées essentiellement de la même manière, quel que soit le type de données stockées.

Exemple : interface.ipynb

```
interface IDessinable
{
    void Dessiner();
}

interface ICloneable<T>
{
    T Clone();
}
```

```
public class Chien : IDessinable, ICloneable<Chien>
{
    private string Nom { get; }

    public Chien(string nom)
    {
        Nom = nom;
    }

    public void Dessiner()
    {
        Console.WriteLine(Nom + " 🦙");
    }

    public Chien Clone()
    {
        return new Chien(Nom);
    }
}
```

Exemple : interface.ipynb

```
var chien = new Chien("Fido");
chien.Dessiner();

var chienClone = chien.Clone();
chienClone.Dessiner();
```

Généricité - Résumé

- Le concept de généricité diffère la spécification du type jusqu'à ce que la classe ou la méthode soit déclarée et instanciée par le code;
- On peut appliquer ce concept sur les classes (et interfaces) et sur les méthodes;
- Pour les méthodes, il faut définir à quel moment la définition de surcharge est inutile et q'il est plus approprié d'utiliser une méthode générique;