L'héritage

L'héritage permet de représenter et d'implémenter une relation de spécialisation entre une **classe de base** et une **classe dérivée**. Il permet donc la transmission des caractéristiques et du comportement du type de base vers le type dérivé, ainsi que leur modification.

1. Mise en œuvre

Pour déclarer une relation d'héritage entre deux classes, il est nécessaire de modifier la déclaration de celle que l'on veut définir comme étant la classe dérivée. Cette modification consiste en l'ajout du symbole ":" et du nom de la classe de base après la déclaration du type :

```
public class ClasseDeBase
{
    public int Identifiant;

    public void AfficherIdentifiant()
    {
          Console.WriteLine(Identifiant)
    }
}

public class ClasseDerivee : ClasseDeBase
{
}
```

À l'inverse d'autres langages comme le C++, C# n'autorise pas à faire dériver un type de plusieurs classes de base.

Une fois que cette relation d'héritage est mise en place, il est parfaitement possible d'écrire le code suivant :

```
ClasseDerivee monObjet = new ClasseDerivee();
monObjet.Identifiant = 42;
monObjet.AfficherIdentifiant();
//Affiche 42 dans la console
```

En effet, `ClasseDerivee` peut accéder aux membres `public`, `internal` et `protected` de sa classe de base, dont la variable `Identifiant` et la méthode `AfficherIdentifiant`.

2. Les mots-clés this et base

Le mot-clé `this` renvoie l'instance courante de la classe dans laquelle il est utilisé. Il permet par exemple de passer une référence de l'objet courant à un autre objet.

```
namespace ThisEtBase
{
    public class Voiture
    {
```

```
        public decimal Longueur;

        //Constructeur de la classe Voiture
        public Voiture()
        {
                //Les 2 lignes suivantes ont exactement le même
                //rôle puisque this renvoie l'objet courant

                Longueur = 4.2;
                this.Longueur = 4.2;

                //La ligne suivante écrit dans la console
                //le résultat de l'appel this.ToString()
                //("ThisEtBase.Voiture", dans ce cas)

                Console.WriteLine(this):
        }
    }
}
```

Le mot-clé `base` permet quant à lui de faire référence à l'objet courant, mais sous la forme de sa classe de base. Ce mot-clé n'est pas utilisable seul : il doit obligatoirement être utilisé pour faire référence à un membre de la classe de base. Il n'est donc pas possible d'écrire le code suivant :

```
Console.WriteLine(base);
```

Le compilateur C# nous renvoie en effet une erreur assez explicite :

En revanche, il est possible d'écrire ceci :

```
Console.WriteLine(base.ToString());
```

On pourrait s'attendre à ce que le résultat affiché dans la console soit le type de la classe de base (`System.Object`), ce qui n'est pas le cas puisque nous obtenons "ThisEtBase.Voiture". L'explication est très simple, et nous allons la détailler tout de suite.

Tout d'abord, le code de la méthode `ToString()` de la classe `System.Object` dont la classe `Voiture` hérite implicitement est le suivant :

```
public virtual string ToString()
{
    return GetType().ToString();
```

```
}
```

Le mot-clé `virtual` et son rôle sont traités à la section suivante : Redéfinition et masquage.

La méthode `GetType` appelée par cette méthode renvoie le type instancié, donc le type "réel" de l'objet, et non le type sous la forme duquel nous voyons l'objet. Le résultat de l'appel à `base.ToString()` est donc parfaitement logique.

3. Redéfinition et masquage

Dans une relation d'héritage, il peut être nécessaire de modifier un comportement pour l'adapter aux caractéristiques d'une classe dérivée. Deux manières de faire sont disponibles avec C# : la redéfinition et le masquage. Ces deux techniques correspondent à la réécriture d'une fonction dans une classe enfant, mais on les utilise dans des contextes différents.

a. Redéfinition de méthode

La redéfinition d'une méthode est applicable lorsque la méthode est définie dans la classe mère avec le mot-clé `virtual`. Ce mot-clé indique que la méthode a été pensée pour pouvoir être redéfinie.

Considérons une classe de base `Personne` définissant une méthode marquée comme `virtual`.

```
public class Personne
{
    public virtual void Conduire()
    {
            //Code pour conduire en respectant le code de la route
    }
}
```

La méthode `Conduire` est marquée comme `virtual` car certains types de personnes peuvent potentiellement exécuter la même action de manière différente. Une personne lambda doit conduire en respectant le code de la route, un pilote conduit de manière à effectuer son trajet le plus rapidement possible et un conducteur de bus doit s'arrêter fréquemment pour laisser monter ou descendre des passagers.

Dans une classe `Pilote` héritant de `Personne`, il faudrait donc redéfinir la méthode `Conduire` pour tenir compte de cette différence. Pour cela, il faut réécrire la méthode avec son code spécifique dans la classe `Pilote`, et la marquer avec le mot-clé `override`.

```
public class Pilote : Personne
{
    public override void Conduire()
    {
            //Code pour conduire le plus vite possible
    }
}
```

La méthode `override` doit avoir la même signature et la même visibilité que la méthode de la classe de base.

b. Masquage de méthode

Le masquage de méthode est similaire à la redéfinition, mais ses effets sont différents (ceux-ci seront étudiés à la section suivante : Différences entre redéfinition et masquage). Il permet de réécrire une méthode dans une classe dérivée, mais pour le masquage, on utilise le mot-clé `new` au lieu du mot-clé `override`.

```
public class Pilote : Personne
{
    public new void Conduire()
    {
           //Code pour conduire le plus vite possible
    }
}
```

Il n'est pas nécessaire que la méthode soit marquée comme `virtual` par la classe de base pour la masquer.

c. Différences entre redéfinition et masquage

Bien que ces deux notions paraissent identiques, elles sont légèrement différentes dans leur effet.

La classe `Personne` définit deux méthodes, `Conduire_Override` et `Conduire_New`. La classe `Pilote` hérite de `Personne` et redéfinit la première méthode, tandis qu'elle va masquer la seconde.

```
public class Personne
{
    public virtual void Conduire_Override()
    {
           Console.WriteLine("Je conduis en respectant le code de
la route");
    }

    public void Conduire_New()
    {
           Console.WriteLine("Je conduis en respectant le code de
la route");
    }

}

public class Pilote : Personne
{
    public override void Conduire_Override()
    {
           Console.WriteLine("Je conduis le plus vite possible
pour gagner");
    }

    public new void Conduire_New()
    {
```

```
            Console.WriteLine("Je conduis le plus vite possible
pour gagner");
    }
}
```

Redéfinition

Exécutez le code suivant :

```
Personne personne = new Personne();
personne.Conduire_Override();

Pilote pilote = new Pilote();
pilote.Conduire_Override();

Personne personnePilote = new Pilote();
personnePilote.Conduire_Override();
```

Le résultat de cette exécution est le suivant :

```
Je conduis en respectant le code de la route
Je conduis le plus vite possible pour gagner
Je conduis le plus vite possible pour gagner
```

Masquage

Exécutez maintenant le code suivant :

```
Personne personne = new Personne();
personne.Conduire_New();

Pilote pilote = new Pilote();
pilote.Conduire_New();

Personne personnePilote = new Pilote();
personnePilote.Conduire_New();
```

Le résultat de cette exécution est le suivant :

```
Je conduis en respectant le code de la route
Je conduis le plus vite possible pour gagner
Je conduis en respectant le code de la route
```

On constate une différence au niveau de la troisième opération. La clé est dans cette instruction :

```
Personne personnePilote = new Pilote();
```

La méthode `Personne.Conduire_Override` est marquée comme `virtual`. La présence de ce mot-clé déclenche le processus suivant :

- Au moment de son exécution, le CLR inspecte le type concret de la variable `personnePilote`. Ici, ce type est `Pilote`.
- Il recherche la méthode substituée `Conduire_Override` dans le type concret `Pilote`.

À ce stade, deux possibilités :

- La méthode est trouvée, et il l'exécute.
- La méthode n'est pas trouvée et dans ce cas le CLR recommence en considérant que le type concret est la classe mère du type concret, et ce jusqu'à trouver la méthode cherchée.

Lorsque l'on utilise l'opérateur `new` sur une méthode, on coupe l'arbre de recherche.

Si la méthode `Personne.Conduire_New` est marquée comme `virtual`, on exécute la recherche en commençant par la classe mère du premier type masquant la méthode. Ici, la classe `Personne`.

Si la méthode n'était pas marquée comme `virtual` au départ, on exécute simplement la méthode de la classe manipulée, ici `Personne.Conduire_New`.

Ces concepts peuvent paraître compliqués à appréhender à première vue, mais les manipuler rend la chose plus simple. N'hésitez pas à écrire du code avec des relations d'héritage pour mieux l'assimiler.

4. Imposer ou interdire l'héritage

Dans certains cas, il peut être intéressant de placer certaines contraintes sur une classe afin de limiter les conditions de son utilisation, notamment d'imposer ou d'interdire l'héritage de cette classe. En C#, il est possible de le faire à l'aide de deux mots-clés : `abstract` et `sealed`.

Les classes abstraites

Les classes abstraites sont des types définis comme non instanciables. Elles ne sont donc utilisables qu'au travers de classes dérivées.

Ces types sont utilisés pour centraliser des éléments de code communs à plusieurs types ou pour fournir des fonctionnalités de base devant être complétées dans un type dérivé. Ils peuvent contenir tous les éléments de code d'une classe "normale" : méthodes, propriétés, variables membres...

Les classes abstraites peuvent aussi contenir des méthodes abstraites, c'est-à-dire dont l'implémentation n'est pas fournie. Il est alors nécessaire que chaque type dérivé fournisse sa propre implémentation de la méthode.

La syntaxe de déclaration d'une classe abstraite nécessite d'ajouter le mot-clé `abstract` avant le nom de la classe.

```
<modificateur d'accès> abstract <nom de classe>
{
```

```
}
```

Ci-dessous un exemple de classe abstraite possédant une méthode abstraite, et dont une classe hérite :

```
public abstract class ClasseAbstraite
{
    public abstract void MethodeAbstraite();
}

public class ClasseDerivee : ClasseAbstraite
{
    public override void MethodeAbstraite()
    {
           //Implémentation de la méthode
    }
}
```

Les classes scellées

Les classes scellées fonctionnent à l'inverse des classes abstraites : elles sont forcément instanciables (elles ne peuvent pas être définies comme `abstract`), mais il est impossible d'en hériter. Pour cette raison, il est impossible de définir un de ses membres comme `abstract` ou `virtual`, puisqu'il serait de toute façon impossible de lui fournir une implémentation dans une classe dérivée.

La syntaxe de déclaration d'une classe scellée nécessite d'ajouter le mot-clé `sealed` avant le nom de la classe.

```
<modificateur d'accès> sealed <nom de classe>
{
}
```

5. Le transtypage

Le transtypage est un des points fondamentaux de la programmation orientée objet avec C#, car il est une des clés pour la bonne mise en œuvre du polymorphisme. C'est en effet lui qui permet de visualiser un objet sous plusieurs formes.

En fonction du contexte d'utilisation, il peut être implicite ou explicite. Les conversions implicites sont effectuées automatiquement lorsqu'elles sont nécessaires. Elles ne peuvent être effectuées que lorsque deux conditions sont réunies :

- Les types source et destination sont compatibles : les deux types sont liés par une relation d'héritage, ou une opération de conversion implicite vers le type destination a été définie dans le type source.
- La conversion d'un type vers l'autre n'entraîne aucune perte de données.

Des transtypages implicites peuvent être effectués, entre autres, d'un type numérique `short` vers `int`, car aucune perte de données ne sera occasionnée, ou d'un type dérivé vers un type de base, car l'objet du type dérivé est forcément un objet du type de base.

```
short valeurShort = 12;
```

```
int valeurInt = valeur


//Le type Pilote hérite de Personne
Pilote pilote = new Pilote();
Personne personne = pilote;
```

En revanche, l'écriture des lignes de code suivantes cause une erreur de compilation :

```
int valeurInt = 42;
short valeurShort = valeurInt;
```

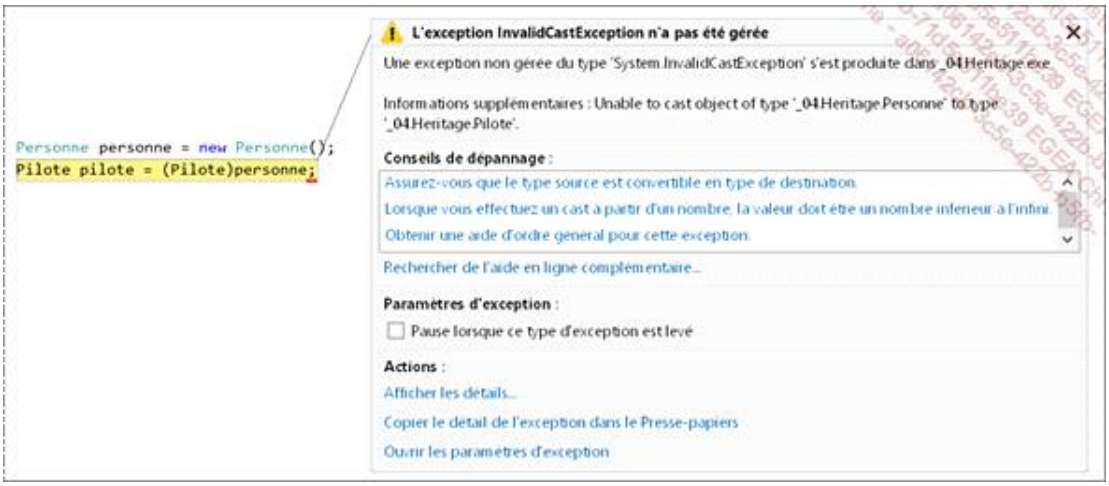
En effet, même si l'on sait que la valeur 42 peut parfaitement être contenue dans un `short`, le compilateur détecte une possible perte de données sur la conversion du type `int` (4 octets) vers `short` (2 octets). Il est ici requis d'utiliser un **cast (transtypage explicite)** afin de valider l'intention du développeur.

L'opérateur de transtypage est représenté par un couple de parenthèses entre lesquelles est précisé le type de destination. Dans le cas mentionné ci-dessus, la correction est la suivante :

```
int valeurInt = 42;
short valeurShort = (short)valeurInt;
```

Le cas est légèrement plus compliqué pour transtyper de la classe `Personne` vers la classe `Pilote`. Il faut en effet que l'objet de type `Personne` contienne déjà un objet de type `Pilote` pour pouvoir retourner vers ce type.

Le code suivant génère une erreur d'exécution :



Pour éliminer cette erreur, il faudrait avoir le code suivant :

```
Personne personne = new Pilote();
Pilote pilote = (Pilote)personne;
```