Module 318 Programmation orientée objets

CFC d'informaticien

Nicolas Wanner
Centre de **F**ormation **P**rofessionnel **T**echnique
École d'informatique

0 Table des matières

1 Les classes et les objets

A la fin de cette fiche, vous serez capable de faire la différence entre une classe et un objet. Vous serez également capable de créer des classes en C# à partir d'un diagramme UML et d'instancier des objets à partir d'une classe et de les utiliser dans vos programmes.

1.1 Quelques définitions

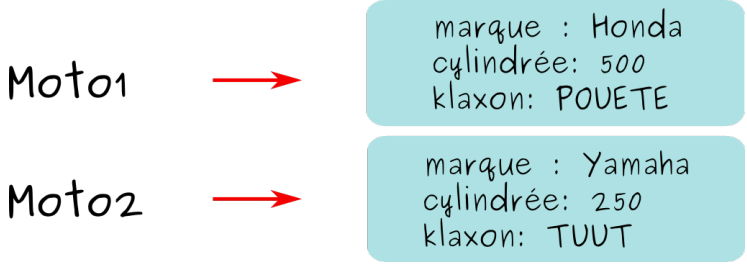
La POO ou la **P**rogrammation **O**rientée **O**bjets est une façon d'organiser ses programmes. En *POO*, le programmeur va chercher à découper son programme en objets qui communiquent entre eux. Alors qu'avec une approche de ***programmation structurée***, le développeur cherche à découper son programme en fonctions (traitements) qui s'échangent des données.

Le principal avantage de la *POO* est que le cerveau est habitué à voir des objets et à les manipuler il est donc plus naturel pour un développeur de concevoir des applications qui mettent en œuvre des objets qui interagissent que de cascades de fonctions.

Pour illustrer le paragraphe précédant, je vous propose de trouver l'intrus dans l'image suivante :

Vous aurez bien compris que votre cerveau va naturellement faire des groupes d'objets de mêmes natures et laisser le Gremlins tout seul. Ce qui nous amène vers la prochaine définitions.

Un objet est une entité (un espace mémoire) qui contient des **attributs** qui sont les données de l'objet et des **méthodes** qui sont des actions que l'objet sait faire.



Une classe est un moule ou un schéma qui va permettre de fabriquer des objets lors de l'exécution d'un programme.

1.2 Les avantages de la POO

La *POO* est une méthode de programmation qui permet aux développeurs de découper les applications en objets ce qui est une approche assez naturelle.

Si la découpe est bien faite, les objets sont relativement indépendants les uns des autres ce qui va simplifier la programmation ainsi que la lecture du code résultant.

De plus, il sera aussi plus facile de faire évoluer les programmes car le fonctionnement interne d'un objet n'a pas d'importance. Seul les interfaces le sont.

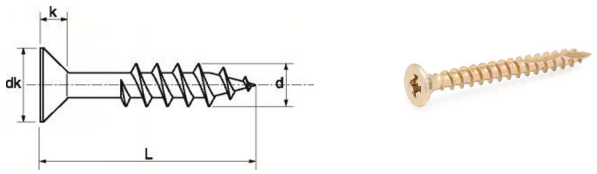
1.3 Créer et utiliser un objet

Pour créer un objet en mémoire, le C# utilise un modèle qui s'appelle une **classe** dans le jargon informatique.

Cette classe définit les **attributs**, c'est à dire les caractéristiques des objets (comme la couleur d'une voiture, le poids de la postière ou le QI du patron, ...)

Elle définit également les **méthodes**, c'est à dire les actions que peuvent faire les objets (Démarrer pour la voiture, distribuer un colis pour la postière ou payer le salaire des employés pour un patron).

La classe L'objet



Lorsque vous concevez une classe, vous devez garder à l'esprit qu'une classe est un groupe de "choses" qui vont ensemble. Ne faites pas de classes pour les choses que vous ne savez pas où mettre...

Si vous n'arrivez pas à donner un nom à une classe, c'est probablement que votre façon de délimiter un objet n'est pas optimale.

Le code suivant vous montre la syntaxe utilisée par le C# pour définir une classe :

```
1  class Moto {
2    public string marque;          // Marque de la moto
3    public float  cylindree;       // Volume des cylindres
4    public string klaxon;          // Bruit du klaxon
5
6    // Produire un son pour impressionner les piétons
7    public void   Klaxonner() {
8       console.WriteLine( this.klaxon );
9    }
10 }
```

Une lecture attentive de la classe nous indique qu'il y a trois attributs :

1. La `marque` qui est une variable de type string.

2. La `cylindree` qui est une variable de type float.
3. Le `klaxon` qui est une variable de type string.

La méthode `Klaxonner()` est aussi définie dans l'exemple. Pour accéder aux attributs de la classe dans une méthode, il faut utiliser le mot clé `this` qui indique que l'attribut appartient à l'objet actuel.

Nous verrons plus tard ce que signifie le mot clé `public` qui utilisé pour définir les attributs et les méthodes de cet exemple.

Dans le cadre scolaire, écrivez tous les identifiants en **Camel Case** et respectez les règles suivantes :

1. Le nom des classes commence par une majuscule.
2. Le nom des attributs commence par une minuscule.
3. Le nom des méthodes commence par une majuscule.
4. Placez une seule classe par fichier.

Maintenant que vous avez décrit votre classe, vous allez l'utiliser pour créer des objets de même nature (type). Le C# utilise, comme beaucoup d'autres langages de programmation, la commande `new` pour fabriquer un objet à partir d'une classe.

```
1 Moto moto1 = new Moto();
```

Dans l'exemple précédent, le programme défini une variable `moto1` du type `Moto` et grâce à la commande `new` va réserver et initialiser de la mémoire pour une moto.

L'extrait de code suivant vous montre comment utiliser votre nouvelle moto :

```
1 // Donner les valeurs
2 moto1.marque    = "Honda";
3 moto1.cylindree = 500;
4 moto1.klaxon    = "POUEEEETE !!!";
5
6 // Faire peur à un piéton
7 moto1.Klaxonner();
```

1.4 Le constructeur

Dans la section précédente, nous avons vu comment créer un objet à partir d'une classe en utilisant la commande `new`. Cela se dit aussi, instancier un objet de classe X. Nous avons également vu comment accéder à ses attributs et méthodes.

Lors de l'instanciation d'un nouveal objet, il est possible de lui donner des arguments qui vont être utilisés pour l'initialiser automatiquement.

Mais pour que cela fonctionne, il faut définir une méthode particulière qui s'appelle le **constructeur** et qui est appelée automatiquement par la commande `new` pour initialiser les attributs avec des valeurs sensées.

L'extrait de code montre comment la nouvelle classe classe `Moto` avec un constructeur :

```
class Moto {
  public string marque;          // Marque de la moto
  public float  cylindree;       // Volume des cylindres
  public string klaxon;          // Bruit du klaxon

  // Le constructeur
  public Moto( string fab, float cyl, string klax="Tuuut") {
     this.marque    = fab;
     this.cylindree = cyl;
     this.klaxon    = klax;
  }

  // Produire un son pour impressionner les piétons
  public void   Klaxonner() {
     console.WriteLine( this.klaxon );
  }
}
```

La méthode qui est utilisée pour construire la classe (le constructeur) doit toujours porter le même nom que la classe, `Moto` dans l'exemple. Le constructeur n'a pas de type de retour (ne pas mettre void devant) et il peut prendre autant d'argument que nécessaire.

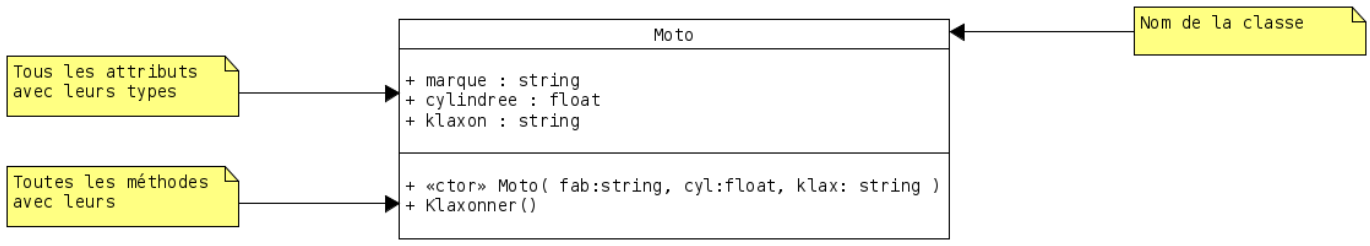
Dans l'exemple ci-dessus, le constructeur prend 3 arguments : `fab`, `cyl` et `klax`. Pour l'argument `klax` une valeur par défaut a été spécifiée. Cela veut dire que lors de l'instanciation de l'objet seul les 2 premiers paramètres sont obligatoires.

```
Moto honda  = new Moto("Honda",  500, "BRRRR");
Moto harley = new Moto("Harley", 1500);

honda.Klaxonner()   // produit -> BRRRRR
harley.Klaxonner()  // produit -> Tuuut
```

1.5 Diagramme de classe

Pour échanger des idées sur les classes à implémenter dans un programme, il existe un formalisme graphique qui s'appelle le **diagramme de classe UML** et qui est le suivant :



1.5.1 Les attributs

Dans un diagramme de classe, la définition d'un attributs respecte la convention suivante :

En UML

```
+ variable1 : string
- variable2 : int
# variable3 : float
```

En C#

```
public    string  variable1;
private   int     variable2;
protected float   variable3;
```

1.5.2 Les méthodes

Dans un diagramme de classe, la définition d'une méthode respecte la convention suivante :

En UML

```
+ Bronzer( temps:int, saison:string ) :  int
```

Si aucun type de retour est spécifié dans l'UML, en C# il faut indiquer void

En C#

```
public int Bronzer( int temps, string saison  )
{
   // Le code de la méthode Bronzer
   return -1;
}
```

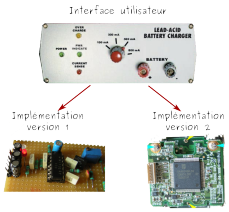
2 Encapsulation

Dans cette fiche, vous allez apprendre à limiter l'accès aux champs d'un objet afin de garantir qu'ils contiennent des valeurs valides

2.1 Limiter l'accès

Un des avantage de la programmation objet est de rendre le programme plus modulaire, avec des parties qui dépendent moins les unes des autres. Pour garantir cette indépendance, l'accès aux objets se fait uniquement par des interfaces clairement définies (les méthodes). L'accès aux attributs est généralement limité à un accès par les méthodes de l'objet lui-même.

Dans l'exemple ci-dessous, tiré du monde de l'électronique, vous avez une face avant qui est l'interface exportée par votre objet. Si vous ne décrivez pas son fonctionnement interne, vous pouvez changer l'électronique sans que l'utilisateur s'en aperçoive.



Par contre si vous attendez que l'utilisateur accède directement à la pin 3 du circuit sur veroboard quand vous changez votre électronique, l'utilisateur devra changer sa façon de travailler ! En termes informatiques, cela revient à modifier le code source de votre application ce qui est embêtant.

Voici un petit exemple de la façon de modifier la visibilité des éléments d'une classe en C#.

```
1 class Moto {
2   public    String Marque;    // Libre accès à tous
3   protected int    annee;     // Seulement pour la classe
4                               // et ses dérivés
5   private   float  cylindree; // Seulement pour la classe
6
7   // ...
8 }
```

Status	Description
public	Tout le monde peut accéder à cet attribut ou méthode
protected	Seul l'objet ou ses dérivés (descendants) peuvent accéder à cet attribut ou méthode
private	Seul l'objet en question peut accéder à cet attribut ou méthode

2.2 Comment accéder aux attributs protégés

Certains attributs protégés doivent quand même être accessibles de l'extérieur. Pour éviter que les utilisateurs n'inscrivent des valeurs erronées, vous devez lui mettre à disposition des méthodes qui vont valider les informations reçues ou mettre en forme les résultats pour la sortie.

Ces méthodes s'appellent en français des **accesseurs** et des **mutateurs**. De façon plus commune, ces méthodes s'appellent des **getter** et des **setter**. Dans la terminologie C#, le terme **propriété** est utilisé pour décrire les attributs accessible par des getter/setter.

```
class Joueur {
   private int energie;  // énergie vitale du joueur
                         // 0   :Le joueur est mort
                         // 100 : Le joueur est à fond

   // Lire l'énergie
   public int LireEnergie() {
      return this.energie;
   }

   // Écrire énergie (limiter entre 0 et 100 compris
   public void EcrireEnergie( int nouveau ) {
      this.energie = nouveau;
      if ( this.energie < 0 ) {
         this.energie = 0;
      }
      else if ( this.energie > 100 ) {
         this.energie = 100;
      }
   }

   // ...
}
```

Et voici comment utiliser les getter/setter créés à la main.

```
// ...
Joueur player1 = new Joueur();

player1.EcrireEnergie( 150 );  // corrige à 100

// ...

if ( player1.LireEnergie() == 0  ) {
   MessageBox.Show( "t'es mort!" );
}
// ...
```

Cette approche fonctionne parfaitement bien et est la seule disponible dans beaucoup de langages orientés objets. En C#, il existe une méthode automatique pour créer des accesseurs et des mutateurs qui seront ensuite cachés dans le signe =.

Placez la souris sur le champ auquel vous désiré puis click-droit → Refactoriser, Encapsuler le champs

Une fois les getter / setter générés, voici le code que vous trouverez dans votre éditeur :

```
class Joueur {
   // L'attribut privé
   private int _energie;

   // La propriété (accès contrôlé)
   public int Energie {
      // Lire l'attribut
      get {
         return _energie;
      }

      // Ecrire dans l'attibut
      set {
         _energie = value;
         if (_energie > 100) {
            _energie= 100;
         }
         else if (_energie < 0) {
            _energie = 0;
         }
      }
   }
}
```

L'exemple ci-dessous montre comment utiliser les accesseurs et mutateurs créés "automatiquement" par Visual Studio.

```
// ...
Joueur player1 = new Joueur();

player1.Energie = 150;   // corrige à 100

// ...

if ( player1.Energie == 0  ) {
   MessageBox.Show( "t'es mort!" );
}
// ...
```

3 Surcharge et constructeurs

A la fin de cette fiche, vous serez capable de surcharger des méthodes pour faciliter l'utilisation de vos classes. Vous serez également capable de surcharger le constructeur et d'en désigné un comme constructeur par défaut.

3.1 Surcharge de méthodes

Dans la programmation orientée objet, il est possible d'avoir plusieurs méthodes qui portent le même nom mais qui attendent des paramètres différent en nombre et en type.

```
1  public void Afficher( int aVal )  {
2     // Définition pour afficher des nombres entiers
3  }
4
5  public void Afficher( double aVal )  {
6     // 1ère surcharge pour afficher des nbr à virgule
7  }
8
9  public void Afficher( String aVal )  {
10    //2ème surcharge pour afficher du texte
11 }
```

L'intérêt de la surcharge réside dans la possibilité de créer une méthode qui travaillera sur des données de types différents ou qui force l'utilisation de valeur par défaut si les paramètres sont manquant.

```
1  // L'energie du joueur est descendue d'une valeur par défaut
2  public void ChangerEnergie() {
3     this.ChangerEnergie( -5 );
4  }
5
6  // L'energie du joueur est modifiée d'après le paramètre
7  public void ChangerEnergie( int aDelta ) {
8     //...
9  }
```

Quand vous utilisez la surcharge, il est important de se rappeler que le comportement des méthodes surchargées doit être le même mais agir sur des types différents ou permettre l'utilisation de valeurs par défaut.
Si vous surchargez les méthodes pour avoir de comportements différents, vous allez rencontrer des problèmes de compréhension de votre code !

3.2 Surcharger le constructeur

Lorsque vous créez un nouvel objet avec le mot clé `new`

```
Elephant livreJungle = new Elephant();
```

le C# va automatiquement appeler une méthode qui s'appelle le **constructeur** et qui a pour objectif d'initialiser l'objet dans un état connu et valide.

Le constructeur est une méthode spéciale, il ne possède pas de valeur de retour (ne rien mettre entre le `public` et le nom du constructeur)

En général, quand vous créez un objet, vous aimeriez bien fixer des valeurs particulières à ses attributs. Sans constructeur, il faudrait créer l'objet puis assigner des valeurs à tous les attributs. Vous pouvez vous épargner ce travail en donnant des valeurs à votre constructeur.

```
class Elephant {
   public int age;
   public String nom;

   // Le constructeur sans paramètres
   public Elephant() {
      this.age = 32;
      this.nom = "colonel Hathi";
   }

   // Le constructeur avec paramètres
   public Elephant( int aAge, String aNom ) {
      this.age = aAge;
      this.nom = aNom
   }

   // ...
}
```

Dans l'exemple nous avons créé une surcharge du constructeur. Il est donc possible de créer des `Elephant` avec ou sans valeurs par défaut :

```
Elephant livreJungle = new Elephant( 32, "Colonel Hathi" );
Elephant anonyme     = new Elephant();
```

Cette méthode fonctionne bien mais elle n'est pas très pratique car il faut dupliquer passablement de code. C'est pourquoi, il est possible de désigner un constructeur par défaut. Il s'agit normalement du constructeur qui a le plus de paramètres.

```
class Elephant {
   public int age;
   public String nom;

   // Le constructeur sans paramètres
   // Ce constructeur fait appel au constructeur avec 2 paramètres.
   public Elephant()  :this( 0, "Inconnu" ) {
      // Doit rester vide
   }

   // Le constructeur avec le nom
   // Celui-ci prend 1 paramètre et donne le travail au suivant
```

```
13      public Elephant( String aNom ) this( 0, aNom ) {
14          // Doit rester vide
15      }
16
17      // Le constructeur avec paramètres
18      // Notre constructeur délégué
19      public Elephant( int aAge, String aNom ) {
20          this.age = aAge;
21          this.nom = aNom;
22      }
23      // ...
24  }
```

4 La patron Modèle - Vue

Un patron de conception (design pattern) est une façon d'organiser les programmes ou une partie de programme pour résoudre un problème particulier.

Il existe une grande quantité de design pattern qui sont expliqué en long et en large dans des livres. Un des patron de conception les plus en vogue est le patron Modèle - Vue - Contrôleur ou MVC. Il est utilisé pour découpler les données de l'affichage.

Dans le cadre de ce module, nous allons nous limiter au pattern "Model - View" (MV) qui est simple à comprendre mais relativement efficace dans la réutilisation de code.

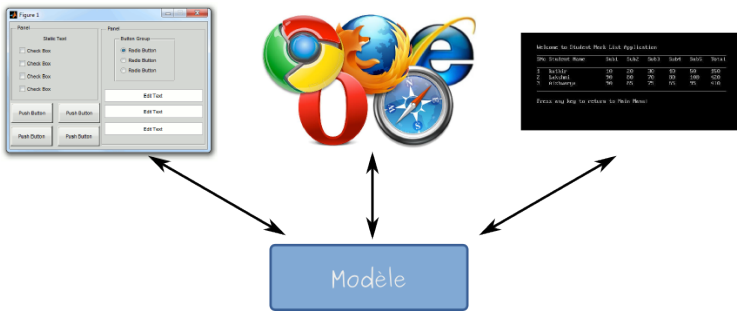
4.1 Quel est sont utilité ?

Un programme fait rapidement ressemble à l'image précédente. C'est un sac de noeuds dans lequel il est difficile de s'y retrouver et d'en réutiliser une partie dans un autre programme.

Le but du pattern MV est de couper votre application en 2 blocs.

La vue	La vue est la partie du programme qui se charge de communiquer avec l'utilisateur. Elle doit donc proposer une interface graphique ou une application en mode console.
Le modèle	Le modèle est la partie du programme qui se charge d'enregistrer les données et de les traiter (filtrer, calculer, trier, ...)

Grâce à ce pattern, les programmes deviennent plus lisibles car il y a une séparation entre les données et l'interface utilisateur. Vous aurez aussi l'avantage de pouvoir utiliser le même modèle pour des applications en mode console, en mode graphique ou même pour une interface web. L'illustration suivante montre schématiquement ce principe.



4.2 Créer le modèle

Pour commencer, il faut concevoir une classe qui serve de ***modèle***. Dans cette classe, il faut placer tout ce qui est nécessaire pour accéder aux données.

Il ne faut par contre pas trouver de commandes qui accèdent à la console (Console.WriteLine, Console.ReadLine, ...) ou à des éléments graphiques (tbxSaisie.Text, LblAffichage.Text, ...)

Voici un exemple de modèle pour enregistrer des sommes d'argent :

```
class ModeleArgent {
   private double valeurCHF;
   private double tauxChange;

   public ModeleArgent(change) {
      valeurCHF = 0;
      tauxChange = change;
   }

   public EcrireCHF(double valeur) {
      valeurCHF = valeur;
   }

   public double LireCHF() {
      return valeurCHF;
   }

   public EcrireEUR(double valeur) {
      valeurCHF = valeur / valeur;
   }

   public double LireEUR() {
      return valeurCHF * valeur;
   }

}
```

4.3 Créer une vue

La vue est responsable de présenter les données du modèle à l'utilisateur et de permettre à l'utilisateur de saisir des données.

Nous pouvons donc réaliser une vue sous la forme d'un programme Console. Cela donne le code suivant :

```
class Program {
   static void Main(string[] args) {
      ModeleArgent  monModele = new ModeleArgent (1.1);
      string lire;
      double valeur;

      Console.Write("La somme en CHF  :");
      lire = Console.ReadLine();
      valeur = Convert.ToDouble(lire);
      monModele.sommeCHF = valeur;

```

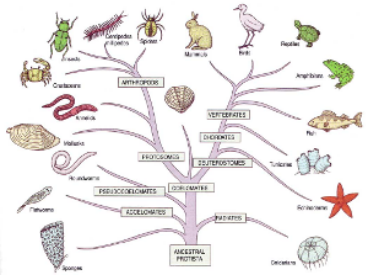
```
12          Console.WriteLine("{0} en CHF --> {1} EUR", valeur, monModele.LireEuro
                ());
13
14          Console.ReadLine();
15      }
16  }
```

Pour que cela fonctionne, il faut bien sûr que le modèle et la vue soient dans le même projet et que dans la vue, et qu'un objet de classe Modèle soit instancier dans la vue.

5 Composition et héritage

A la fin de cette fiche, vous serez capables de

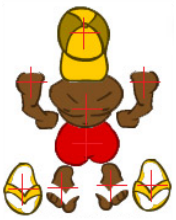
1. Créer des objets en combinant plusieurs objets plus simples.
2. Créer des objets qui héritent des caractéristiques d'autres objets



5.1 Créer un objet composite

En programmation orientée objet, vous pouvez étendre les possibilités d'une classe en lui ajoutant un objet comme attribut. Dans l'image ci-dessous, le personnage est **composé** de pieds, de tongues, de bras, d'un caleçon, d'un torse et d'une casquette

http://coronalabs.com/



http://coronalabs.com/

Si vous devez créer cet objet Personnage, il faudrait le composer à partir d'autres objets. L'image ci-dessous, montre comment créer un tel objet.

```
class Personnage {
   // Droite
   public Pied      dPied ;
   public Tongue    dTongue ;
   public Bras      dBras ;

   // Gauche
   public Pied      gPied
   public Tongue    gTongue ;
   public Bras      gBras ;

   // Reste
   public Calcon    calcon ;
   public Corp      corp ;
   public Tete      tete ;
   public Casquette casquette ;

}
```

5.2 L'héritage

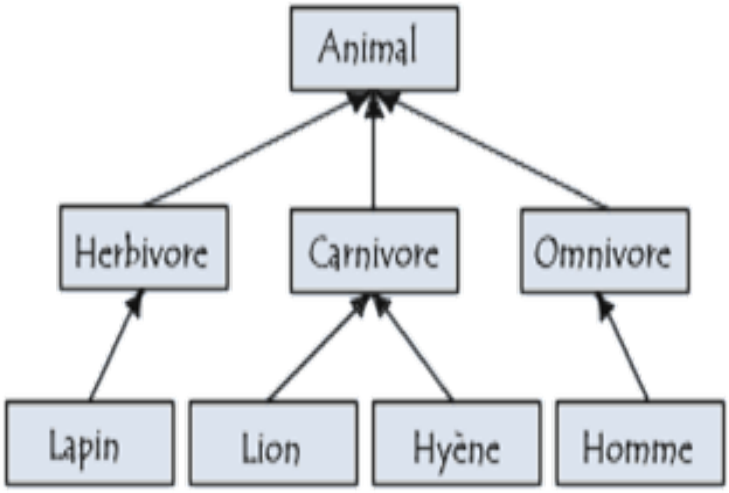
Une des notions les plus importantes en POO est l'héritage. L'héritage permet à une classe de recevoir tous les attributs et toutes les méthodes de la classe parent dont elle hérite.

Qu'est-ce ?

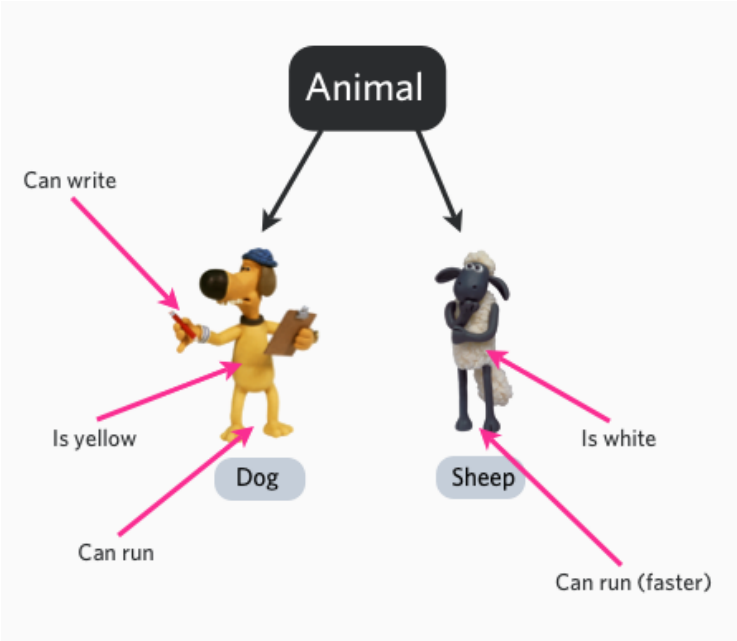
- Une ***classe fille*** est construite sur la base d'une ***classe parent***.
- Toutes les caractéristiques du parent se retrouvent dans la fille.

Quel intérêt ?

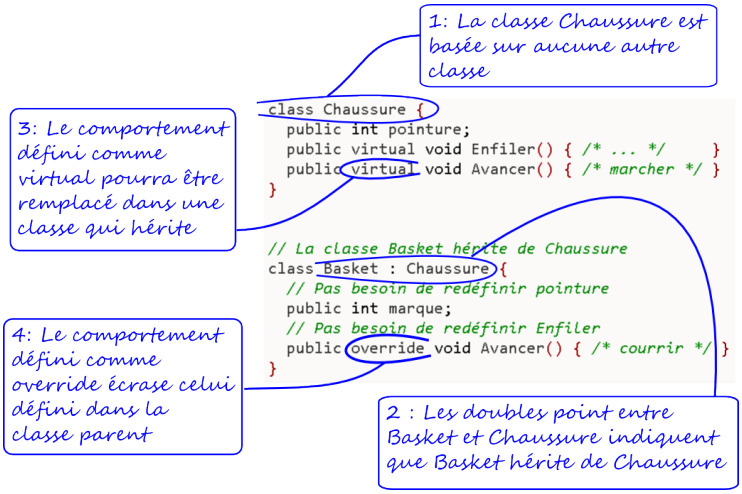
- Éviter les copier / coller.
- Réduit la quantité code.
- Limite les erreurs / tests.



L'héritage permet d'étendre une classe en lui ajoutant des méthodes et des attributs. Il permet également de remplacer le comportement défini dans la classe parent par un autre comportement plus adaptés,



L'exemple suivant montre comment la classe Basket hérite de la classe Chaussure puis remplace le comportement de la méthode Avancer. Elle montre aussi que la basket a des attributs supplémentaires.



Le trans-typage permet de faire passer une Basket pour une Chaussure de façon entièrement transparente. L'intérêt du trans-typage est montré dans l'exemple suivant. Nous avons une armoire de ***Chaussures*** mais il nous sera possible d'y ranger des ***Baskets***, des ***Ballerines*** ou tout autres héritiers de la classe Chaussure.

```
// ...

// Créer un tableau de chaussures
Chaussure[] armoire = new Chaussures[3];

// Dans ce tableau je peux ranger des baskets
armoire[0] = new Basket();

// Mais aussi des ballerines (un dérivé de chaussures)
armoire[1] = new Ballerine();

// ou toutes sortes de dérivés de chaussures

// ...
```

L'exemple suivant montre comment utiliser le trans-typage avec des listes à la place de tableaux.

```
// ...

// Créer une liste de chaussures
List<Chaussure> armoire = new List<Chaussure>();

// Dans cette liste je peux ranger des baskets
armoire.Add( new Basket() );

// Mais aussi des ballerines (un dérivé de chaussures)
armoire.Add( new Ballerine() );

// Je peux chercher dans mon armoire...
foreach( Chaussure chausse in armoire ) {
   chausse.Avancer();
}

// ...
```

6 Fichiers et sérialisation

6.1 Lecture de fichiers

Pour rendre possible la lecture de fichiers dans un programme, il faut ajouter la ligne suivante dans l'entête du fichier :

```
using System.IO;
```

L'extrait de code suivant montre comment lire toutes les lignes d'un fichier textuel

```
StreamReader fichier = null;          // Objet fichier
string       nom     = "fichier.txt"; // Nom du fichier
string       ligne   = "";            // Ligne lue

// Vérifier si le fichier existe
if ( File.Exists( nom ) )
{
   // Le fichier existe alors je peux l'ouvrir
   // en créant un objet de classe StreamReader
   fichier = new StreamReader( nom );

   // Tant que je n'ai pas atteint la fin du fichier ...
   while(fichier.EndOfStream == false)
   {
      // Je lis la ligne suivante
      ligne = fichier.ReadLine();
      // J'affiche la ligne lue
      // Mais je peux faire des choses plus intéressante avec
      Console.WriteLine( ligne );
   }

   // Fermer le fichier et libérer la variable
   fichier.Close();
   fichier = null;
}
```

6.2 Écriture de fichiers

Pour rendre possible l'écriture de fichiers dans un programme, il faut ajouter la ligne suivante dans l'entête du fichier :

```
using System.IO;
```

L'extrait de code suivant montre comment écrire quelques lignes dans un fichier textuel

```
StreamWriter fichier = null;           // Objet fichier
string       nom     = "fichier.txt"; // Nom du fichier
bool         ajouter = true;          // Ajouter rue/false

// J'ouvre le fichier en écriture.
// Si ajouter est true alors j'ajoute à la fin du fichier
fichier = new StreamWriter( nom, ajouter  );


// Je fais appel à la méthode WriteLine pour ajouter
// du texte dans le fichier. Une nouvelle ligne est
// automatiquement créée
fichier.WriteLine( "Un peu de texte dans le fichier" );
fichier.WriteLine( "Et une autre ligne" );
fichier.WriteLine( "J'en ajoute autant que je veux" );
fichier.WriteLine( "..." );

// Fermez le fichier et libérer la variable
fichier.Close();
fichier = null;
```

6.3 Sérialiser et désérialiser un objet

Il peut être intéressant de sauver le contenu d'un objet dans un fichier pour reprendre le travail exactement où il avait été laissé. Cette opération s'appelle la sérialisation. La façon dont les données sont encodées dans le fichier dépend du sérialiseur. En C# le format est le XML.

Pour rendre possible la sérialisation d'objets dans un programme, il faut ajouter les lignes suivantes dans l'entête du fichier :

```
using System.IO;
using System.Xml.Serialization;
```

L'extrait de code suivante défini une classe **Exemple**. Notez la présence de l'instruction **`[serializable]`** placé avant la définition de la classe. Cela indique que la classe doit pouvoir être chargée et sauvée sur le disque.

```
[Serializable]
public class Exemple {
   public int    nbEntier  = 1;
   public double nbDecimal = 3.14;
   public bool   nbBool    = false;
   public string duTexte   = "hello";

   // Constructeur simple
   public Exemple()  :this(0,0,false,"") { }

   // Constructeur désigné
   public Exemple( int  aEntier, double aDecimal,
                   bool aBool,   string aTexte ) {
       this.nbEntier  = aEntier;
       this.nbDecimal = aDecimal;
       this.nbBool    = aBool;
       this.duTexte   = aTexte;
   }
}
```

Ensuite, pour enregistrer un objet de cette classe, il faut créer un objet à sérializer, créer un sérializeur et un fichier en écriture :

```
Exemple       monObj      = null;   // L'objet
XmlSerializer serialiseur = null;   // Le sérialiseur
StreamWriter  ecriture    = null;   // Fichier

// Je crée l'objet à sauver
monObj      = new Exemple( 5, 2.1, false, "hello" );
// Je créer le sauvteur
serialiseur = new XmlSerializer( typeof(MonObj) );
// Je crée le fichier de sauvegarde
ecriture    = new StreamWriter( "sauvegarde" );

// Je sauve mes données dans le fichier
serialiseur.Serialize( ecriture, monObj );

// Je ferme le fichier et le libère
ecriture.Close();
ecriture = null;
```

Pour recharger l'état d'un objet, l'opération est similaire voici un extrait de code qui réalise le travail :

```
Exemple       monObj      = null;   // L'objet
XmlSerializer serialiseur = null;   // Le désérialiseur
StreamReader  lecture     = null;   // Le fichier

// Je créer le sauvteur
serialiseur = new XmlSerializer( typeof(Exemple) );
// Je crée le fichier de sauvegarde
lecture     = new StreamReader( "sauvegarde" );

// Créer un objet ave les données chargées
monObj = (Exemple )serialiseur.Deserialize(lecture);

// Fermer et libérer le lecteur de fichier
lecture.Close();
lecture = null;
```

Il est important de noter que lors de l'appel à la méthode **`Deserialize`**, un nouvel objet est construit. De plus il faut forcer le type de la valeur retournée pour que la compilation se passe sans erreur. Forcer le type (type casting en anglais) se fait en mettant le type entre parenthèses.