# TP POO C++ : l'héritage



## Sommaire



**Les objectifs de ce tp sont de découvrir la programmation orientée objet en C++.**
On désire réaliser un programme C++ permettant d'écrire facilement des histoires de Western.
*Dans nos histoires, nous aurons des brigands, des cowboys, des shérifs, des barmen et des dames en détresses ...* (à partir d'une idée de Laurent Provot)

# Rappels POO

## Notion d'objets

La programmation orientée objet consiste à **définir des objets logiciels et à les faire interagir entre eux**.

Concrètement, un **objet** est une **structure de données** (**ses attributs** c.-à-d. des variables) qui définit son **état** et une **interface** (**ses méthodes** c.-à-d. des fonctions) qui définit son **comportement**.

Un objet est créé à partir d'un **modèle** appelé **classe**. Chaque objet créé à partir de cette classe est une **instance** de la classe en question.

## Notion de classe

Une classe **déclare des propriétés communes** à un ensemble d'objets.

Une classe représentera donc une **catégorie d'objets**.

Elle apparaît comme un **type** ou un *moule* à partir duquel il sera possible de créer des objets.

## Notion de visibilité

Le C++ permet de préciser le **type d'accès des membres** (attributs et méthodes) d'un objet. Cette opération s'effectue au sein des classes de ces objets :

- **public** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privé** (*private*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et non par ceux d'une autre classe.

## Notion d'encapsulation

L'encapsulation est l'idée de **protéger les variables contenues dans un objet et de ne proposer que des méthodes pour les manipuler**.

L'objet est ainsi vu de l'extérieur comme une "boîte noire" possédant certaines propriétés et ayant un comportement spécifié.

# Travail demandé

## Western C++

On désire réaliser des programmes orientés objet en C++ qui **raconteront des histoires dans lesquelles un humain arrive, se présente et boit un coup** :


```
(Lucky Luke) -- Bonjour, je suis Lucky Luke et j'aime le coca-cola
(Lucky Luke) -- Ah ! un bon verre de coca-cola ! GLOUPS !
```

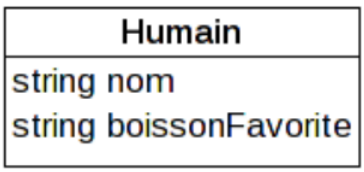
**Et une histoire dans laquelle un cowboy rencontre une dame coquette** :

```
(Lucky Luke) -- Bonjour, je suis le vaillant Lucky
Luke et j'aime le coca-cola
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une
jolie robe blanche
(Jenny) -- Regardez ma nouvelle robe verte !
(Lucky Luke) -- Ah ! un bon verre de coca-cola !
GLOUPS !
(Jenny) -- Ah ! un bon verre de lait ! GLOUPS !
```

## Les humains

Les intervenants de nos histoires sont tous des humains. Notre **humain** est caractérisé par son **nom** et sa **boisson favorite**. La boisson favorite d'un humain est, **par défaut**, de l'eau.

On modélise donc une **classe `Humain`** :

| UML | C++ |
|---|---|
|  | |



```
class Humain
{
   private:
      string nom;
      string boissonFavorite;
};
```

Toutes les variables (attributs) de la classe `Humain` seront privées.

Pour créer des objets à partir de cette classe, il faudra ... **un constructeur**. On le **déclare** de la manière suivante :

```
class Humain
{
   public:
      // Constructeur :
      Humain(const string nom="inconnu", const string boissonFavorite="eau"); // de l'eau
          par défaut

   private:
      string nom;
      string boissonFavorite;
};
```

*humain.h*

Le constructeur porte toujours le même nom que la classe.

Il faut maintenant **définir** ce constructeur afin qu'il **initialise toutes les variables de l'objet au moment de sa création** :

```
#include "humain.h"

// Constructeur
Humain::Humain(const string nom/*="inconnu"*/, const string boissonFavorite/*="eau"*/)
   : nom(nom), boissonFavorite(boissonFavorite)
{
}
```

*humain.cpp*

Le constructeur est appelé automatiquement au moment de la création d'un objet.

On pourra alors créer nos propres humains :

| UML | C++ |
|---|---|
| lucky:Humain<br>boissonFavorite = coca-cola<br>nom = Luck Luke | `Humain lucky("Lucky Luke", "coca-cola");` |
| joe:Humain<br>boissonFavorite = whisky<br>nom = Joe Dalton | `Humain *joe = new Humain("Joe", "whisky");` |

Les objets `lucky` et `joe` sont des instances de la classe `Humain`. Un objet possède sa propre existence et un état qui lui est spécifique (c.-à-d. les valeurs de ses attributs).

Un humain pourra **parler**. On aura donc une **méthode** `parle(texte)` qui affiche :

`(nom de l'humain) -- texte`

| UML | C++ |
|---|---|
| Humain<br>string nom<br>string boissonFavorite<br>Humain(const string nom = "", const string boissonFavorite = "eau")<br>void parle(const string texte) | `class Humain`<br>`{`<br>`  private:`<br>`   string nom;`<br>`   string boissonFavorite;`<br>`  public:`<br>`   Humain(const string nom="", const string boissonFavorite="eau");`<br>`   void parle(const string texte);`<br>`};`<br><br>`void Humain::parle(const string texte)`<br>`{`<br>`   cout << "(" << nom << ") -- " << texte << endl;`<br>`}` |

On utilisera cette méthode dès que l'on voudra faire dire quelque chose par un humain :

```
Humain lucky("Lucky Luke", "coca-cola");

lucky.parle("Je parle !"); // appel de la méthode parle()
```

Ce qui donnera à l'exécution :

`(Lucky Luke) -- Je parle !`

Une méthode publique est un service rendu à l'utilisateur de l'objet.

Toutes les variables de la classe `Humain` étant **privées par respect du principe d'encapsulation**, on veut néanmoins pouvoir connaître sa boisson favorite et pouvoir modifier cette dernière.

Il faut donc créer deux **méthodes publiques** pour **accéder** à l'**attribut** `boissonFavorite` :

| UML | C++ |
|---|---|
| Humain<br>string nom<br>string boissonFavorite<br>Humain(const string nom = "", const string boissonFavorite = "eau")<br>string getBoissonFavorite()<br>void setBoissonFavorite(const string nouvelleBoissonFavorite)<br>void parle(const string texte) | `class Humain`<br>`{`<br>`  private:`<br>`   string nom;`<br>`   string boissonFavorite;`<br>`  public:`<br>`   Humain(const string nom="", const string boissonFavorite="eau");`<br>`   string getBoissonFavorite() const;`<br>`   void setBoissonFavorite(const string nouvelleBoissonFavorite);`<br>`   void parle(const string texte)`<br>`};`<br><br>`// Assesseur get`<br>`string Humain::getBoissonFavorite() const`<br>`{`<br>`  return boissonFavorite;`<br>`}`<br><br>`// Manipulateur set`<br>`void Humain::setBoissonFavorite(const string nouvelleBoissonFavorite)`<br>`{`<br>`  if(!nouvelleBoissonFavorite.empty())`<br>`   BoissonFavorite = nouvelleBoissonFavorite;`<br>`}` |

La méthode publique `getBoissonFavorite()` est un accesseur (*get*) et `setBoissonFavorite()` est un manipulateur (*set*) de l'attribut `boissonFavorite`. L'utilisateur de cet objet ne pourra pas lire ou modifier directement une propriété sans passer par un accesseur ou un manipulateur.

Un **humain** pourra également **se présenter** (il dit bonjour, son nom, et indique sa boisson favorite), et **boire** (il dira « Ah ! un bon verre de (sa boisson favorite) ! GLOUPS ! »). On veut aussi pouvoir connaître le **nom** d'un **humain** mais il n'est pas possible de le modifier.

**Question 1.** Compléter les fichiers `humain.cpp` et `humain.h` fournis afin d'assurer l'exécution du programme de test ci-dessous.

```cpp
#include <iostream>

using namespace std;

#include "humain.h"

/* TP POO C++ : tous humains ... */

int main()
{
   Humain lucky("Lucky Luke", "coca-cola");

   cout << "Une histoire sur " << lucky.getNom() << endl;
   lucky.sePresente();
   lucky.boit();

   Humain *joe = new Humain("Joe");
   cout << "Une histoire sur " << joe->getNom() << endl ;
   joe->setBoissonFavorite("whisky");
   joe->sePresente();
   joe->boit();

   return 0;
}
```

*histoire-1.cpp*

```
Une histoire sur Lucky Luke
(Lucky Luke) -- Bonjour, je suis Lucky Luke et j'aime le coca-cola
(Lucky Luke) -- Ah ! un bon verre de coca-cola ! GLOUPS !
Une histoire sur Joe
(Joe) -- Bonjour, je suis Joe et j'aime le whisky
(Joe) -- Ah ! un bon verre de whisky ! GLOUPS !
```

## Notion d'héritage

L'**héritage** est un **concept fondamental de la programmation orientée objet**. Elle se nomme ainsi car le principe est en quelque sorte le même que celui d'un arbre généalogique. Ce principe est fondé sur des **classes « filles »** qui héritent des caractéristiques des **classses « mères »**.

L'héritage permet **d'ajouter des propriétés à une classe existante pour en obtenir une nouvelle plus précise**. Il permet donc **la spécialisation ou la dérivation de types**.


B **hérite** de A : "un B **est** un A avec des choses en plus". Toutes les instances de B sont aussi des instances de A.

On dit aussi que B dérive de A. A est une généralisation de B et B est une spécialisation de A.

## Propriétés de l'héritage

L'héritage est **une relation entre classes** qui a les propriétés suivantes :
- si B hérite de A et si C hérite de B alors C hérite de A
- une classe ne peut hériter d'elle-même
- si A hérite de B, B n'hérite pas de A
- il n'est pas possible que B hérite de A, C hérite de B et que A hérite de C
- le C++ permet à une classe C d'hériter des propriétés des classes A et B (héritage multiple)

## Notion de visibilité

*Rappels : Le C++ permet de préciser le **type d'accès des membres** (attributs et méthodes) d'un objet. Cette opération s'effectue au sein des classes de ces objets.*

Il faut maintenant tenir compte de la situation d'héritage :

- **public** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privé** (*private*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et **non par ceux d'une autre classe même dérivée**.
- **protégé** (*protected*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et **par ceux d'une classe dérivée**.

## Notion de redéfinition

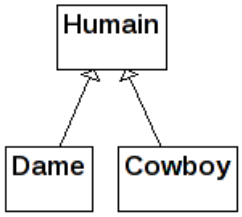
En utilisant l'héritage, il est possible **d'ajouter des caractéristiques**, **d'utiliser les caractéristiques héritées** et de **redéfinir les méthodes héritées**.

Généralement, cette **redéfinition (*overriding*)** se fait par **surcharge** et permet de modifier le comportement hérité.

## Exemple détaillé : des dames coquettes

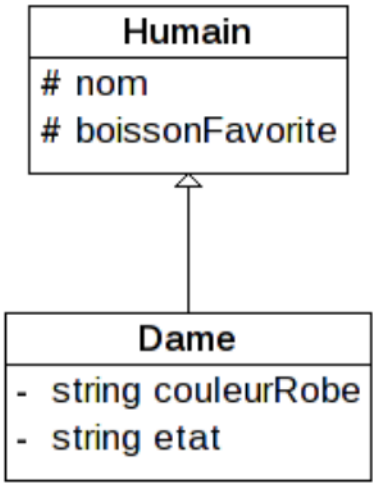
Les **dames** et les **cowboys** sont tous des **humains**. Ils ont tous un nom et peuvent tous se présenter. Par contre, il y a certaines différences entre ces **deux classes d'humains**.

On va donc réaliser la modélisation suivante en utilisant l'héritage :




Une **dame** est caractérisée par la **couleur de sa robe** (une chaîne de caractères), et par son **état** (libre ou captive).

UML




C++

```
class Humain
{
  public:
      Humain(const string nom="", const
string boissonFavorite="eau");

  protected:
      string nom;
      string boissonFavorite;
};

// La classe Dame hérite de la classe Humain
class Dame : public Humain
{
   public:
      Dame(const string nom="", const string
boissonFavorite="lait", const string
couleurRobe="blanche");

   private:
      string couleurRobe;
      string etat;
};

// Le constructeur Dame appelle le constructeur
Humain puis intialise ses propres attributs
Dame::Dame(const string nom, const string
boissonFavorite, const string couleurRobe) :
Humain(nom, boissonFavorite),
couleurRobe(couleurRobe), etat("libre")
{ }
```

Il est nécessaire d'indiquer une visibilité `protected` aux membres `nom` et `boissonFavorite` pour permettre aux objets de la classe `Dame` d'accéder à ces membres.

Elle peut également **changer de robe** (tout en s'écriant « Regardez ma nouvelle robe (couleur de la robe) ! »). On désire aussi changer le mode de présentation des dames car une dame ne pourra s'empêcher de parler de la couleur de sa robe. Et quand on demande son **nom** a une **dame**, elle devra répondre « Miss (son nom) ».

Il faut donc **redéfinir les méthodes hérités** `getNom()` et `sePresente()` pour obtenir le comportement suivant :

```
Dame jenny("Jenny");

jenny.sePresente();
```

devra donner :

```
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une jolie robe blanche
```

On déclare la classe `Dame` :

```
class Dame : public Humain
{
   public:
      // Constructeurs et Destructeur
      Dame(const string nom="", const string boissonFavorite="lait", const string
          couleurRobe="blanche");

      // Accesseurs
      string getNom() const; // surcharge
      string getEtat() const;

      // Services
      void sePresente() const; // surcharge
      void changeDeRobe(const string couleurRobe);

   private:
      string couleurRobe;
      string etat;
};
```

*dame.h*

On définit les méthodes de la classe `Dame` :

```
#include "dame.h"

// Constructeur
Dame::Dame(const string nom/*=""*/, const string boissonFavorite/*="lait"*/, const string
    couleurRobe/*="blanche"*/) : Humain(nom, boissonFavorite), couleurRobe(couleurRobe),
    etat("libre")
{
}

// Accesseurs
string Dame::getNom() const
{
```

```
   return "Miss " + nom;
}

string Dame::getEtat() const
{
   return etat;
}

// Services
void Dame::sePresente() const
{
   cout << "(" << nom << ") -- " << "Bonjour, je suis " << getNom() << " et j'ai une jolie
      robe " << couleurRobe << endl;
}

void Dame::changeDeRobe(const string couleurRobe)
{
   this->couleurRobe = couleurRobe;
   cout << "(" << nom << ") -- " << "Regardez ma nouvelle robe " << couleurRobe << " !" <<
      endl;
}
```

*dame.cpp*

## Des cowboys et des dames coquettes ...

Un **cowboy** est un **humain** qui est caractérisé par sa popularité (0 pour commencer) et un adjectif le caractérisant ("vaillant" par défaut). On désire aussi changer le mode de présentation des cowboys. Un cowboy dira ce que les autres disent de lui (son adjectif).

De même, on veut donner une boisson par défaut à chaque sous-classe d'humain : du lait pour les dames et du whisky pour les cowboys.

**Question 2.** Écrire les classes `Cowboy` et `Dame` afin d'assurer l'exécution du programme de test ci-dessous.

```
#include <iostream>

using namespace std;

#include "cowboy.h"
#include "dame.h"

/* TP POO C++ : des cowboys et des dames coquettes */

int main()
{
   Cowboy lucky("Lucky Luke");
   Dame jenny("Jenny");

   // 1. La rencontre ...
   lucky.sePresente();
   jenny.sePresente();
```

```
   // 2. Allons boire un coup ...
   jenny.changeDeRobe("verte");
   lucky.boit();
   jenny.boit();

   return 0;
}
```

*histoire-2.cpp*

```
Lucky Luke) -- Bonjour, je suis le vaillant Lucky Luke et j'aime le whisky
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une jolie robe blanche
(Jenny) -- Regardez ma nouvelle robe verte !
(Lucky Luke) -- Ah ! un bon verre de whisky ! GLOUPS !
(Jenny) -- Ah ! un bon verre de lait ! GLOUPS !
```