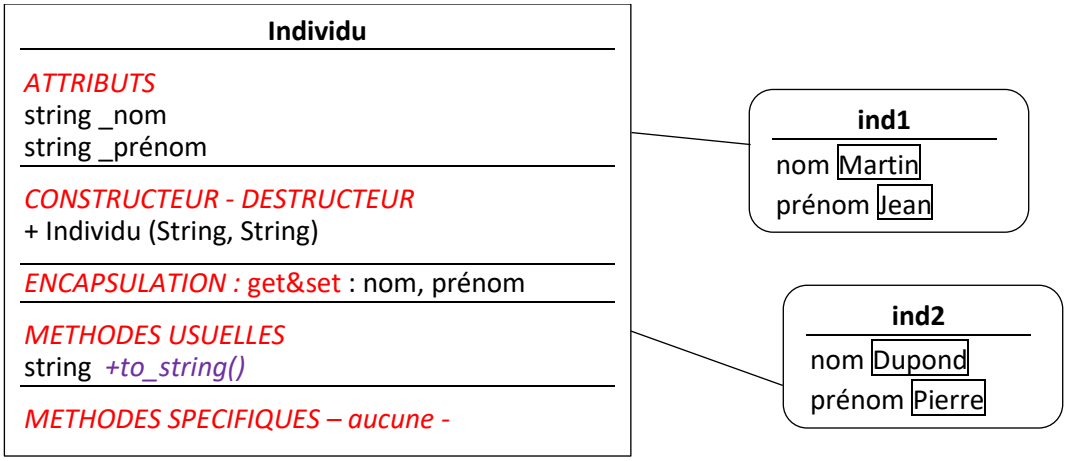
# Quelques rappels de POO, UML et C++

## REFERENCES

https://www.cplusplus.com/
https://openclassrooms.com/
https://cpp.developpez.com/cours/cpp/
http://manual.gromacs.org/2020
https://www.doxygen.nl/manual/docblocks.html

## CLASSE – OBJET – ATTRIBUT – METHODE

**Exemple de représentation UML**




**Plusieurs implémentations sont possibles en C++**

- 1 seul fichier : `main.cpp`
- 3 fichiers : `main.cpp` plus un fichier `.h` et un fichier `.cpp`

Ces deux implémentations possibles sont présentées ci-après et ont le même comportement, à savoir :

```
Création d'un objet individu : nom = Martin et prenom = Jean
Création d'un objet individu : nom =  et prenom =

Le premier individu est : nom (Martin) - prenom (Jean)
nom (Dupond) - prenom (Pierre)
```

# Quelques rappels de POO, UML et C++

**Exemple de codage avec un unique fichier** `main.cpp`

```
/**
 * @file main.cpp
 * @brief Rappel de syntaxe C++ : classe, objet, attributs, méthodes
 * @author Lopistéguy
 * @version 0.1
 * @date jj/mm/aaa
 */
```

```
#include <iostream>   /** Ne pas oublier : d'inclure les ressources et   fichier.h   nécessaires
using namespace std; /**                    et de définir l'espace de référence, si nécessaire également
```

```
/** CLASSE Indvidu **/
class Individu {
   // ATTRIBUTS
   public:
      string   nom;
      string   prenom;

   // ENCAPSULATION : nom, prenom
   public:
      void   setNom (string n) {  nom = n;          }
      string getNom ()         {  return (nom);  }
      void   setPrenom (string p) {  prenom = p;      }
      string getPrenom ()          {  return prenom; }

   // CONSTRUCTEUR
   public:
/*    // Version 1 avec liste d'initialisation
     Individu (string n="", string p="") : nom(n), prenom(p) {
        // code supplémentaire si nécessaire, ou bien rien
        cout << "création d'un objet individu : nom = " << getNom() << " et prenom = " << getPrenom() << endl;
      };
*/
      // Ou bien Version 2 sans liste d'initialisation mais appel des setters
      Individu (string n="", string p="") {
         setNom (n);
         setPrenom(p);
         cout << "création d'un objet individu : nom = " << getNom() << " et prenom = " << getPrenom()
              << endl;
      };

   // METHODES USUELLES : to_string
   public:
      string to_string (string message="") {
         return message + "nom (" + getNom() + ") - prenom (" + getPrenom() + ")";
      }

   // METHODES SPECIFIQUES : aucune

   };    // FIN DE : classe Individu
```

```
/** Programme principal **/
int main () {
   // Initialisation des attributs de l'objet ind1 lors de sa création
   Individu ind1 ("Martin", "Jean");

   // Création de l'objet ind 2 puis initialisation de ses attributs avec les setters
   Individu ind2;
   ind2.setNom ("Dupond");
   ind2.setPrenom ("Pierre");

   cout << ind1.to_string ("Le premier individu est : ") << endl;
   cout << ind2.to_string() << endl;
}
```

**Exemple de codage avec trois fichiers :** `main.cpp` **plus un fichier** `.h` **et un fichier** `.cpp`

```cpp
/**
 * @file main.cpp
 * @brief Rappel de syntaxe C++ : classe, objet, attributs, méthodes
 * @author Lopistéguy
 * @version 0.1
 * @date jj/mm/aaa
 */

#include "Individu.h"

int main () {
    // Initialisation des attributs de l'objet ind1 lors de sa création
    Individu ind1 ("Martin", "Jean");

    // Création de l'objet ind 2 puis initialisation de ses attributs avec les setters
    Individu ind2;
    ind2.setNom ("Dupond");
    ind2.setPrenom ("Pierre");

    cout << ind1.to_string ("le premier individu est : ") << endl;
    cout << ind2.to_string() << endl;
}
```

**main.cpp**

**Rappels :**

1. Contrairement à cet exemple simpliste, un `fichier.h` peut contenir la spécification de plusieurs classes (ici le fichier `Individu.h` contient uniquement la spécification de la classe `Individu`), et de plus, contenir aussi la spécification/signature de fonctions et procédures globales.
2. Il est admis que dans un `fichier.h`, la spécification d'une classe peut comporter le code de certaines méthodes, à savoir, par ordre de préférence :
    a. Le code de tous les setter&getter de la classe - ou bien aucun
    b. Le code de tous les constructeurs de la classe - ou bien aucun
    c. Le code de toutes les méthodes usuelles de la classe - ou bien aucune

# Quelques rappels de POO, UML et C++

```cpp
/**
 * @file individu.h
 * @brief Rappel de syntaxe C++ : classe, objet, attributs, méthodes
 * @author Lopistéguy
 * @version 0.1
 * @date jj/mm/aaa
 */

#ifndef OUTILS_H    // Pour ne pas inclure 2 fois la classe Indvidu.h
#define OUTILS_H

#include <iostream>
using namespace std;

/** CLASSE Indvidu **/
class Individu {
    // ATTRIBUTS
    public:
        string   nom;
        string   prenom;

    // ENCAPSULATION : nom, prenom
    public:
        void setNom (string n);
        string getNom ();
        void setPrenom (string p);
        string getPrenom ();

    // CONSTRUCTEUR
    public:
/*      // Version 1 avec liste d'initialisation
        Individu (string n="", string p="") : nom(n), prenom(p);
*/
        // Ou version 2 sans liste d'initialisation mais appel de setter
         Individu (string n="", string p="");

    // METHODES USUELLES : to_string
    public:
        string to_string (string message="");

    // METHODES SPECIFIQUES : aucune
};

#endif
```

**Individu.h**

```cpp
/**
 * @file Individu.cpp
 * @brief Rappel de syntaxe C++ : classe, objet, attributs, méthodes
 * @author Lopistéguy
 * @version 0.1
 * @date jj/mm/aaa
 */

#include "Individu.h"

// ENCAPSULATION : nom, prenom
void Individu::setNom (string n) { nom = n;      }
string Individu::getNom ()       { return (nom); }
void Individu::setPrenom (string p) { prenom = p;    }
string Individu::getPrenom ()       { return prenom; }

// CONSTRUCTEUR

/* // Version 1 avec liste d'initialisation
Individu::Individu (string n="", string p="") : nom(n), prenom(p) {
    // code supplémentaire si nécessaire
    cout << "creation d'un objet individu : nom = " << getNom()
         << " et prenom = " << getPrenom() << endl;
} */

// Ou bien version 2 sans liste d'initialisation mais appel des setter
Individu::Individu (string n, string p) {
    setNom (n);
    setPrenom(p);
    cout << "creation d'un objet individu : nom = " << getNom()
         << " et prenom = " << getPrenom() << endl;
}

// METHODES USUELLES : to_string

string Individu::to_string (string message) {
    return message + "nom (" + getNom() + ") - prenom ("
                   + getPrenom() + ")";
}
```

**Individu.cpp**

# Quelques rappels de POO, UML et C++

## TYPES

### Types entiers

- **int :** contient un entier de taille normale, positif ou négatif.
- **short int :** contient un entier de petite taille (16 bits), positif ou négatif.
- **long int :** contient un entier de grande taille (32 bits), positif ou négatif.
- **long long int :** contient un entier de plus grande taille (64 bits), positif ou négatif.
- **unsigned int :** contient un entier de taille normale, positif ou nul.
- **unsigned short int :** contient un entier de petite taille, positif ou nul.
- **unsigned long int :** contient un entier de grande taille (32 bits), positif ou nul.
- **unsigned long long int :** contient un entier de plus grande taille (64 bits), positif ou nul.

### Types réels

- **float :** simple précision
- **double :** double précision
- **long double :** précision étendue

### Autres types

- **char :** contient un caractère
- **bool :** contient true ou false
- **string :** n'est pas un type de base C++ ➔ #include <string> ou #include <iostream>

**Opérateur** `const` **:** l'élément reçoit une valeur unique qui ne change plus dans le code.

## ACCESSIBILITÉ : `private`, `protected`, `public`

**…des attributs et des méthodes** (i.e. des membres)

- Accès `public` : les membres sont accessibles par tous.
- Accès `private` : les membres ne sont accessibles que par la classe elle-même.
- Accès `protected` : les membres ne sont accessibles que par la classe elle-même et par toutes les classes dérivées.

**…de la classe héritée** (i.e. classe mère, superclasse) **ex.** `class Etudiant : public Individu`

- Héritage `public` : l'accès à la classe mère d'une classe peut être effectué partout dans le code.
- Héritage `private` : l'accès à la classe mère d'une classe est restreint uniquement à la classe fille.
- Héritage `protected` : l'accès à la classe mère d'une classe est restreint ou à la classe fille et à toute ses classes dérivées.

**Note** : Il est assez rare d'utiliser l'héritage privé ou protégé. Dans ces cas-là, on préfère généralement utiliser la composition, objets membres, qui est plus efficace.

## NOMENCLATURE

- Choisir des identifiants explicites (par exemple des verbes pour les booléens)
- Eviter les abréviations, elles sont source d'ambiguïté pour vous et la maintenance
- Pour les **constants** utiliser des **majuscules** avec le séparateur souligné '_'
- Pour les **autres identifiants** utiliser des **minuscules** avec la notation `camelCase`
- Les types tels que **`class`**, **`struct`**, **`typedef`** et **`enum`** commencent par une **majuscule**
- Les **variables**, **attributs** et **fonctions** (cf. méthodes) commencent par une **minuscule**
- L'encapsulation d'un attribut privé **`nom`** se fait avec **`setNom()`** ou **`Nom()`**, et **`getNom()`**

**AUTRES… convention pour préfixer des éléments**

- Le nom d'une **classe abstraite** est préfixé par `Abstract_`
- Une **variable globale** est préfixée par `g_`
- Une **constante globale** est préfixée par `c_`
- Un **membre `static`** attribut ou méthode d'une classe est préfixé par `s_`

## DOCUMENTATION/COMMENTAIRES (ex.*https://franckh.developpez.com/tutoriels/outils/doxygen/#LIII-E*)

- Il est souhaitable de se référer à une norme (ex. doxygen) pour adopter une nomenclature
- Les commentaires sont obligatoires pour les déclarations dans un **fichier.h**
- Les commentaires sont recommandés dans le code des définitions **fichier.cpp**

## FICHIERS .h .cpp

- Un **`fichier.h`** consacré aux déclarations contient :
  - les **`#include`** qui définissent des éléments utilisés (ex. #include <string> )
  - pas d'include de fichier.cpp
  - les **`#define`** de **macro** et de **constantes symboliques**

```
#define   plus(x,y)   ((x)+(y))
#define   VRAI        true
#define   FOREVER     for(;;)
```

  - les déclarations de types (cf. **`typedef`**, **`enum`**, **`struct`**, **`class`**)
  - les lignes de protection anti inclusions multiples (cf. **`#if`**, **`#ifdef`**, **`#ifndef`**, **`#else`**, **`#endif`**)
  - les prototypes de procédures (void) et de fonctions (return d'un type donné)
  - peut comporter du code (cf. **`get&set`**)
  - est inclue (cf. **`#include`**) dans des **`fichier.cpp`** et/ou **`fichier.h`**
- Un **`fichier.cpp`** contient :
  - les **`#include`** utiles
  - la fonction principale **`main()`** est dans un seul des **`fichier.cpp`** du projet
  - la définition de procédures et fonctions définies dans des classes de **`fichier.h`**

**La séparation .h et .cpp donne de la souplesse pour architecturer une application**

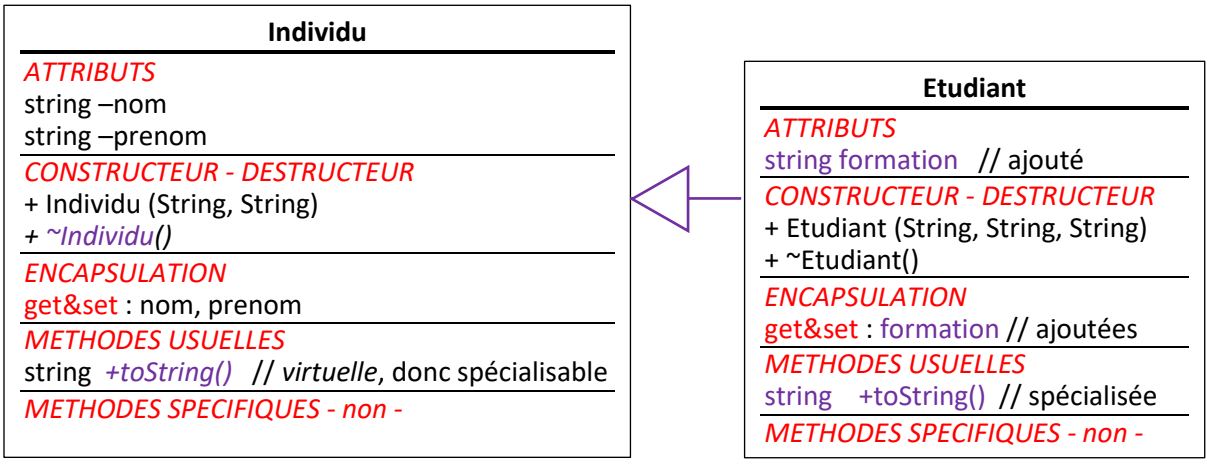
***Pourtant :***

- *On peut créer un fichier.cpp par classe et donc se passer de `fichier.h` (comme en Java)*
- *On peut définir plusieurs classes dans un `fichier.cpp`*

***Ces pratiques arrivent assez rapidement à des blocages de conception architecturales C++***

## HERITAGE – méthode virtuelle & polymorphisme – cf TDn°1

- Dans une classe fille, il est au moins possible de :
  - ajouter des attributs (ex. formation)
  - ajouter des méthodes (ex. get&set formation)
  - redéfinir une méthode définie dans la classe mère (ex. toString())




- Une méthode redéfinie (cf. spécialisée) dans une sous-classe est dite **méthode virtuelle** ; on la déclare avec le mot clé ***virtual*** dans la classe mère et elle est représentée en italique en UML.
- Lorsqu'une méthode virtuelle est redéfinie dans une classe fille, alors la méthode aura le comportement ainsi redéfini, pour tout objet de cette classe-là. La méthode est dite polymorphe - on parle de **polymorphisme**.
- Lorsqu'une classe définit une méthode *virtuelle*, le **destructeur**, s'il est défini, doit être obligatoirement ***virtuel***. Sinon on risque de n'appeler que le destructeur de la classe mère alors qu'il s'agit d'un objet de la classe fille.

**Note** : si la définition de la méthode est déportée dans un fichier .cpp, il n'est pas nécessaire de RE-préciser que la méthode est virtuelle dans le fichier .cpp.

## METHODE VIRTUELLE PURE ➔ CLASSE ABSTRAITE dite interface par abus de langage en C++

- Une **méthode est dite virtuelle pure** lorsqu'elle est déclarée **= 0**

```
// méthode virtuelle pure dans le .h
virtual double getVolume() = 0;
```

- Une classe ayant une méthode virtuelle pure est dite **classe abstraite**
- Une classe abstraite ne peut pas être instanciée. En effet un objet d'une classe abstraite serait dans l'incapacité d'exécuter les méthodes virtuelles pures.

**Une classe abstraite définit les méthodes que devront implémenter les sous-classes.**

- Une **classe concrète** est donc une classe dont toutes les méthodes sont implémentées.