La programmation orientée objet (POO) en C++
Première partie

Thierry Vaira

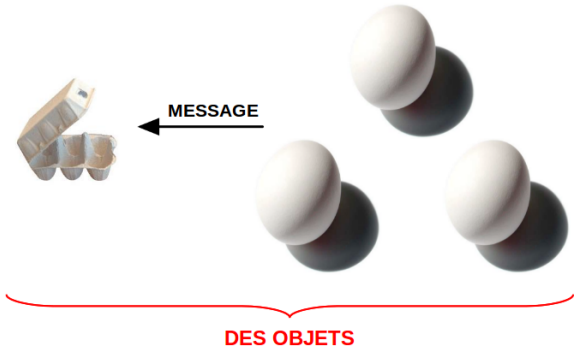
BTS SN Option IR

v.1.1

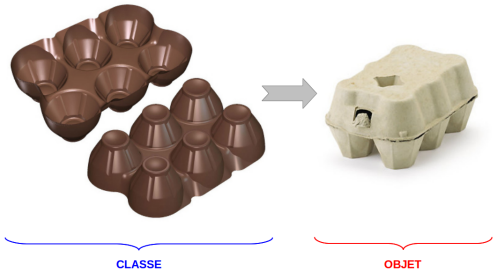
Sommaire

Qu'est ce que la Programmation Orientée Objet ?

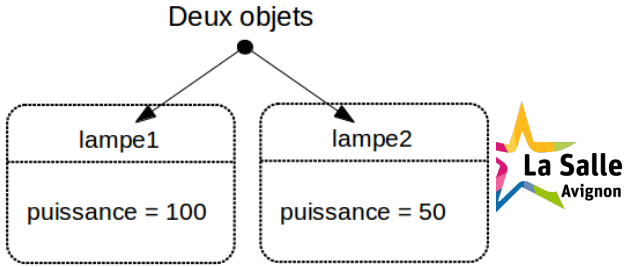
- La programmation objet consiste à **définir des objets logiciels** et **à les faire interagir entre eux**.
  - ➠ On parle d'échange de **messages**



- Un objet est créé à partir d'un **modèle** (« moule ») appelé **classe**.
  - ➠ On parle d'**instanciation**

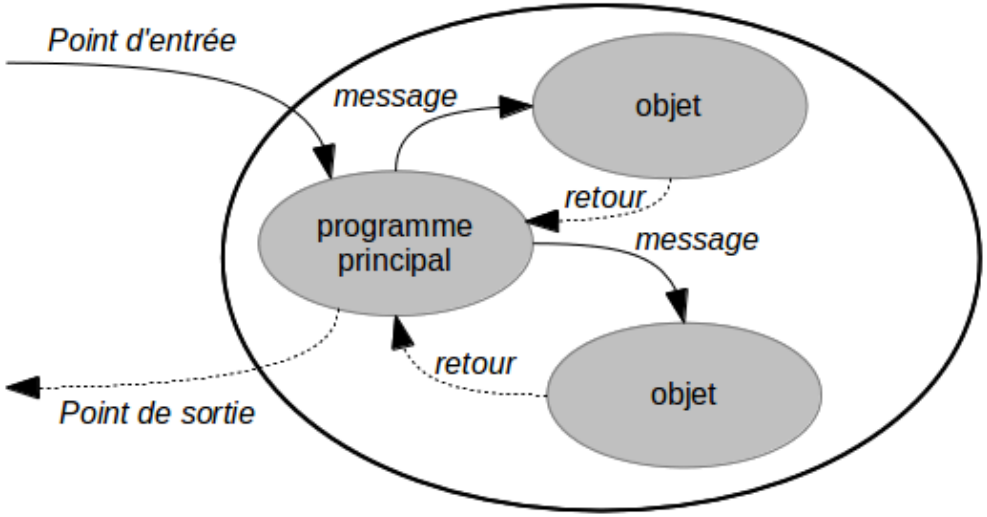


- Un objet posséde une **identité** qui permet de distinguer un objet d'un autre objet.
  - ➠ On parle de **son nom, son adresse mémoire**



Qu'est ce qu'un programme en P.O.O. ?

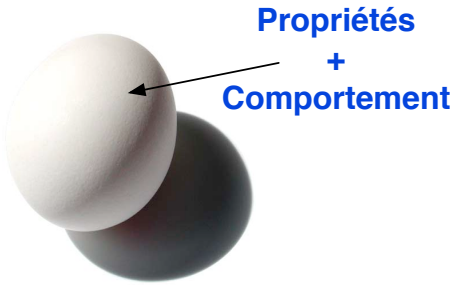
- Un programme Orienté Objet est écrit à partir d'un langage Orienté Objet. Les langages O.O. les plus utilisés sont : `C++`, `C#`, `Java`, `PHP`, `Python`, ...
- En C++, un programme Orienté Objet est vu comme un **ensemble d'objets interagissant entre eux en s'échangeant des messages**.



✍ En C++, il existe toujours une fonction principale `main`. Certains langages O.O. (comme le Java) disposent d'une classe principale qui doit contenir une méthode `main`.

Qu'est ce qu'un objet ?

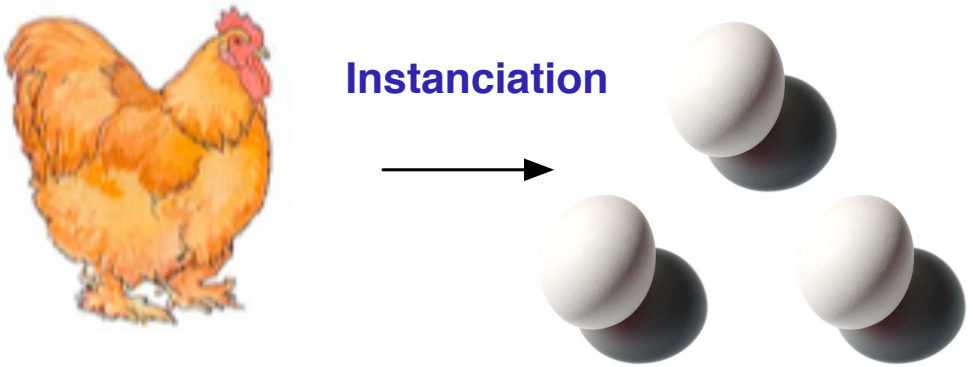
- Un **objet** est caractérisé par le rassemblement, au sein d'une même **unité** d'exécution, d'un ensemble de **propriétés** (constituant son **état**) et d'un ensemble d'**opérations** (constituant son **comportement**)
  - ➠ On parle d'**encapsulation**



- La notion de **propriété** est matérialisée par un **attribut**
  - ➠ Comparable à une « variable locale » à un objet, une donnée (information, caractéristique ou état) propre à l'objet
- La notion d'**opération** est matérialisée par une **méthode**
  - ➠ Comparable à une « fonction » de l'objet, une action pouvant être effectuée sur l'objet

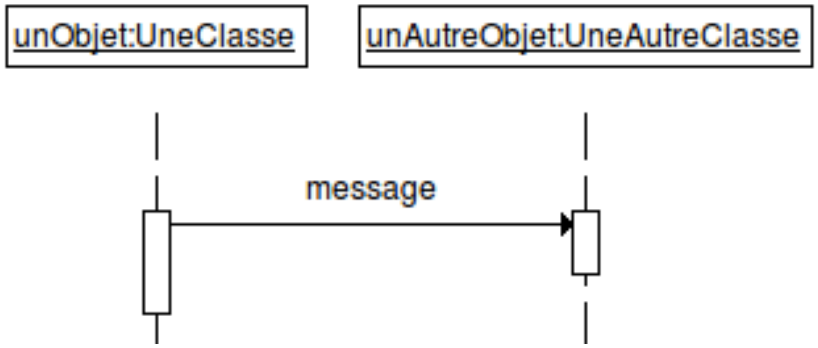
Qu'est ce qu'une classe ?

- Une **classe** définit un **type** qui peut être vue comme un « **moule** »ou une « **fabrique d'objets** »et qui comporte :
  - La **déclaration** des **attributs**
  - La **déclaration** et la **définition** des **méthodes**
- La « fabrication »d'un objet s'appelle l'**instanciation**
  - L'objet est appelé **instance de classe** et possède le type que définit sa classe
- Remarque : La classe n'existe qu'à la compilation, l'objet n'existe qu'à l'exécution



Comment interagir avec un objet ?

- Les **objets** interagissent entre eux en s'échangeant des **messages**.



- La réponse à la réception d'un message par un objet est appelée une **méthode**.
- Une méthode est donc la mise en oeuvre du message : elle décrit la réponse qui doit être donnée au message.
- Le **code des méthodes** contient généralement :
  - Des **accès en lecture/écriture** aux **attributs** d'un objet
  - Des **appels** (on parle d'**invocation**) aux **méthodes** d'un objet

Exemple d'une lampe

- Une lampe est **caractérisée par** :
  - Sa **puissance** (une **propriété** ➠ un **attribut**)
  - Le **fait qu'elle soit allumée ou éteinte** (un **état** ➠ un **attribut**)
- Au niveau **comportement**, les **actions possibles** sur une lampe sont donc :
  - L'**allumer** (une **méthode**)
  - L'**éteindre** (une autre **méthode**)

Déclaration d'une classe en C++

- La déclaration de la classe `Lampe` se fait dans un fichier d'en-tête (*header*) `Lampe.h`

```
class Lampe
{
   private:
      // Attributs
      int puissance;
      bool estAllumee;

   public:
      // Méthodes
      void allumer();
      void eteindre();
};
```

Définition des méthodes d'une classe en C++

- La définition d'une classe revient à définir l'ensemble de ses méthodes. On doit faire précéder chaque méthode de l'opérateur de résolution de portée :: précédé du nom de la classe (ici Lampe) pour préciser au compilateur que ce sont des membres de cette classe.
- La définition des méthodes de la classe Lampe se fait dans un fichier source Lampe.cpp

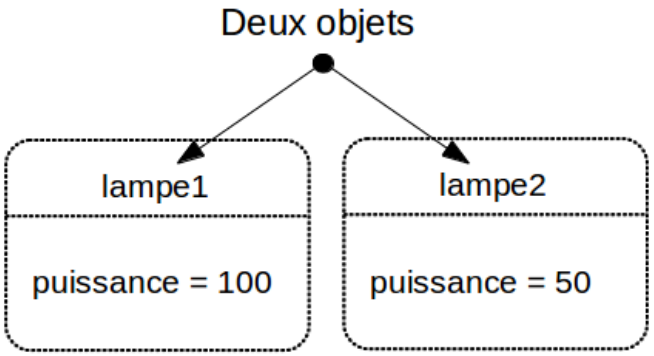
```
#include "Lampe.h"

void Lampe::allumer()
{
   this->estAllumee = true;
}

void Lampe::eteindre()
{
   this->estAllumee = false;
}
```

Instances de classe

- Chaque instance possède un **exemplaire propre** de **chaque attribut défini par la classe**
- Un objet possède sa propre existence et un état (c.-à-d. les valeurs de ses attributs) qui lui est propre (indépendant des autres objets)



- Le **code des méthodes** est **commun à toutes les instances** (il n'existe qu'une seule fois en mémoire)
- Le mot-clé `this` (auto pointeur), utilisé dans le code des méthodes d'une classe, fait **référence à l'instance (l'objet lui-même) sur laquelle est invoquée la méthode**

Instanciation d'un objet

- Instancier un objet revient à créer une variable d'un type classe. Une instance de classe est donc un objet.

```
Lampe l1; // instancie un objet l1 de type Lampe
l1.allumer(); // appel la méthode allumer() à partir de l'objet l1

Lampe *l2 = new Lampe(); // alloue dynamiquement une instance de type Lampe
// La variable l2 n'est pas un objet mais un pointeur sur un objet Lampe !
l2->eteindre(); // appel la méthode eteindre() à partir de l'objet l2
(*l2).eteindre(); // cette écriture est possible : je pointe sur l'objet puis j'
    appelle sa méthode
delete l2;
```

✍ On accède aux membres comme pour une structure : avec l'**opérateur d'accès .** ou l'**opérateur d'indirection ->** si on manipule une adresse.

Constructeur

- L'**instanciation d'un objet** est réalisée par l'**appel à un constructeur** (une méthode qui porte le même nom que la classe)
- D'un point de vue **système**, le constructeur a pour but d'**allouer un nouvel objet en mémoire**
- D'un point de vue **fonctionnel**, le constructeur a pour but de **créer un nouvel objet dans un état « utilisable »**
- Il doit donc simplement contenir les instructions d'**initialisation (des attributs) de l'objet**

```
class Lampe
{
   ...
   public:
      // Constructeur
      Lampe(int puissance);
};

// Définition du constructeur
Lampe::Lampe(int puissance)
{
   this->puissance = puissance;
   this->estAllumee = false;
}
```

Règles du constructeur

- Un constructeur est chargé d'**initialiser un objet de la classe**.
- Il est appelé **automatiquement au moment de la création** de l'objet.
- Un constructeur est une **méthode qui porte toujours le même nom que la classe**.
- Il existe quelques contraintes :
  - Il peut avoir des paramètres, et des valeurs par défaut.
  - Il peut y avoir plusieurs constructeurs pour une même classe (surcharge).
  - Il n'a jamais de type de retour.
- Il existe **implicitement** un **constructeur par défaut** :
  - Son rôle est de créer une instance non initialisée quand aucun autre constructeur fourni n'est applicable.
  - Il est **sans paramètre**.
  - Il est **fortement recommandé** d'écrire soi-même le constructeur par défaut

Constructeur par défaut

```
class Lampe { ...
   public:
      Lampe(); // Déclaration du constructeur par défaut
      ...
};

// Définition du constructeur par défaut
Lampe::Lampe()
{
   this->puissance = 50;
   this->estAllumee = false;
}

// Exemples :
Lampe l1;// appel du constructeur par défaut : initialise puissance à 50
Lampe l2(75);// appel du constructeur : initialise puissance à 75

Lampe *l3 = new Lampe();// appel du constructeur par défaut : initialise
     puissance à 50
Lampe *l4 = new Lampe(100);// appel du constructeur : initialise puissance à 100
Lampe *l5;// pas d'appel du constructeur : aucun objet créé (pointeur non
     initialisé)
```

Tableaux d'objets

- Il est possible de conserver et de manipuler des objets `Lampes` dans un tableau.

```
// Un tableau de 10 objets Lampe
Lampe guirlande[10]; // Le constructeur par défaut est appelé 10 fois (pour
     chaque objet Lampe du tableau) !

for(int i = 0; i < 10; i++)
{
   guirlande[i].allumer();
   sleep(1); // attendre 1 s
   guirlande[i].eteindre();
}
```

Destructeur

Le **destructeur** est la **méthode membre appelée automatiquement lorsqu'une instance (objet) de classe cesse d'exister en mémoire** :

- Son rôle est de **libérer toutes les ressources qui ont été acquises depuis la construction** (typiquement libérer la mémoire qui a été allouée dynamiquement par cet objet).
- Un destructeur est une **méthode qui porte toujours le même nom que la classe, précédé de "~"**.
- Il existe quelques contraintes :
  - Il ne possède aucun paramètre.
  - Il n'y a qu'un et un seul destructeur.
  - Il n'a jamais de type de retour.

```
class Lampe
{
   ...
   public:
      ~Lampe();     // Destructeur
};

Lampe::~Lampe()    // Destructeur
{
   // rien à faire pour l'instant
}

// Exemple :
Lampe *l2 = new Lampe(100);

delete l2; // appel automatique du destructeur
```

Les objets constants

Les règles suivantes s'appliquent aux **objets constants** :

- On déclare un objet constant avec le modificateur **const**
- On ne pourra pas modifier l'**état (ses attributs)** d'un objet constant
- On ne peut appliquer que des **méthodes constantes** sur un objet constant
- Une **méthode constante** est tout simplement une méthode **qui ne modifie aucun des attributs de l'objet**

```
class Lampe { ...
   public:
      int getPuissance() const;
};
// Une méthode constante
int Lampe::getPuissance() const {
   return this->puissance;
}

const Lampe l3(100); // l3 est un objet
     constant

int p = l3.getPuissance(); // autorisé :
      getPuissance() est déclarée const

l3.allumer(); // interdit : allumer() n'
     est pas const (car elle modifie l'
     attribut estAllumee)
```

Accès aux membres (visibilité)

- En POO, il est possible de préciser le **type d'accès aux membres** (**attributs** et **méthodes**) d'un objet.
- Cette opération s'effectue et s'applique au niveau des classes :
  - Accès `public` : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
  - Accès `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. Seule une méthode membre de la même classe peut utiliser ce membre ; il est invisible de l'extérieur de la classe.
  - Accès `protected` : comme les membres privés, mais ils peuvent en plus être utilisés par une classe dérivée (notion d'**héritage**).

Controle d'accès aux membres

```
Lampe l1;

l1.estAllumee = true; // erreur d'accès : 'bool Lampe::estAllumee' is private

l1.allumer(); // pas d'erreur : 'void Lampe::allumer()' is public
```

✍ Une méthode peut évidemment être privée : c'est alors une méthode interne à l'objet (qui réalise un comportement à usage interne).

Règle 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**.
- En respectant ce principe, tous les **attributs** d'une classe seront donc **privés (`private`)**.
- On permettra un accès extérieur en lecture et/ou écriture via des méthodes appelées **accesseurs et mutateurs**.
- L'objet est ainsi vu de l'extérieur comme une « boîte noire » possédant certaines **propriétés** et ayant un **comportement** spécifié et sécurisé.
- C'est le **comportement** (ses méthodes publiques) d'un objet qui modifiera son **état** (c'est-à-dire la valeur de ses attributs privés). L'ensemble des méthodes publiques forme l'**interface**.

Les accesseurs et mutateurs

On ajoute souvent (mais pas obligatoirement) des **méthodes publiques** pour établir un accès contrôlé aux variables privées de la classe :

- `get()` (*getter* ou **accesseur**) : permet l'accès en lecture
- `set()` (*setter* ou **mutateur ou manipulateur**) : permet l'accès en écriture

```
class Lampe { ...
   public:
      int getPuissance() const; // un accesseur
      void setPuissance(int puissance); // un mutateur
};

int Lampe::getPuissance() const {
   return puissance;
}

void Lampe::setPuissance(int puissance) {
   this->puissance = puissance;
}
```