Introduction à la programmation orientée objet : du C au C++

Cours N°4

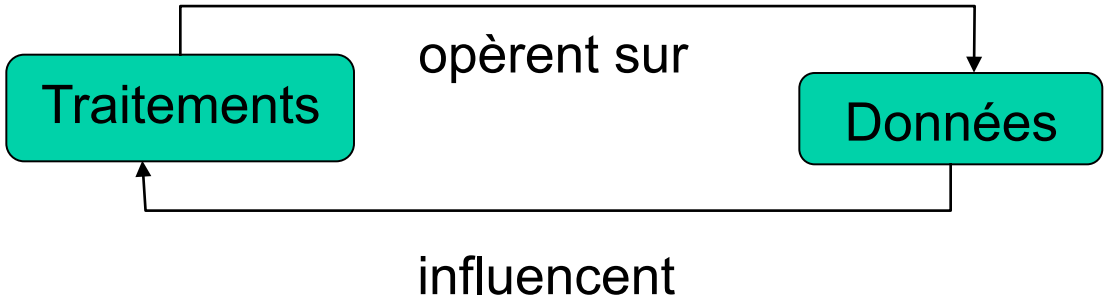
Abdelhak-Djamel Seriai
Maître de conférences
seriai@lirmm.fr
http://www2.lirmm.fr/~seriai/

Basé partiellement sur le cours de J. Sam Haroud

Le paradigme Orienté Objets
Les concepts

Programmation impérative/procédurale

- Dans les programmes que vous avez écrits jusqu'à maintenant, les notions de variables/types de **données** et de **traitement** de ces données étaient séparées :



- L'approche fonctionnelle n'est pas adaptée au développement d'applications qui évoluent sans cesse et dont la complexité croit continuellement
  - Les fonctions ne sont pas les éléments les plus stables
  - On peut pas décrire toujours un système par une fonction principale
  - Difficulté de la réutilisation

Les concepts objet

- Un peu d'histoire
  - 1967 : Simula, 1er langage de programmation à implémenter le concept de type abstrait à l'aide de classes
  - 1976 : Smalltalk implémente les concepts fondateurs de l'approche objet (encapsulation, agrégation, héritage) à l'aide de :
    - classes
    - associations entre classes
    - hiérarchies de classes
    - messages entre objets
  - 1980 : le 1er compilateur C++. C++ est normalisé par l'ANSI et de nombreux langages orientés objets académiques ont étayés les concepts objets : Eiffel, Objective C, Loops, java, …

Les concepts objet

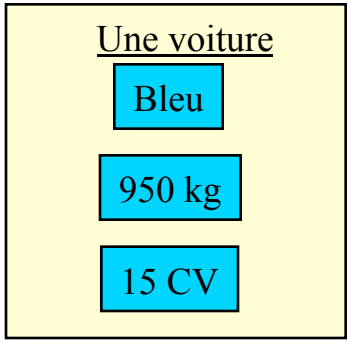
- Objet
  - Définition
    - Objet du monde réel :
      - Perception fondée sur le concept de masse
        - » grains de sable, les étoiles
    - Objet informatique :
      - Est une unité atomique formée de l'union d'un état et d'un comportement
      - Définit une représentation abstraite d'une entité du monde réel ou virtuel, dans le but de la piloter ou de la simuler
        - » Grain de sable, étoile
        - » Compte en banque, police d'assurance, équation mathématiques, etc.
    - Les objets du monde réel et du monde informatique naissent, vivent et meurent

Les concepts objet

- Objet
  - Caractéristique fondamentales d'un objet (informatique)

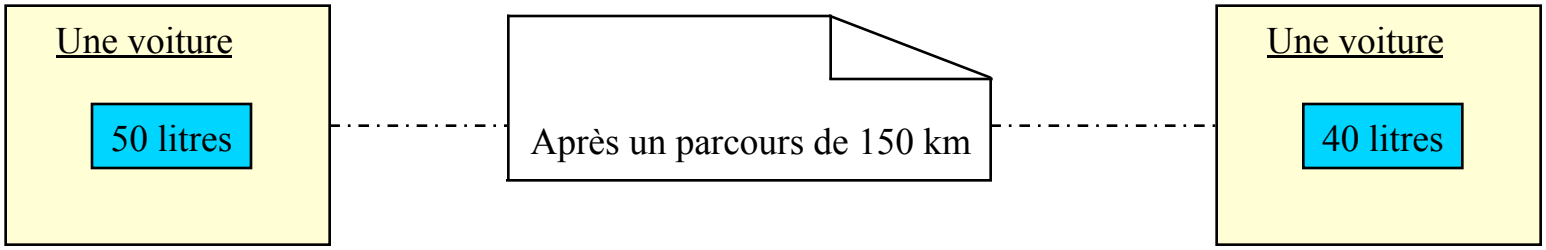
Objet = état + Comportement + Identité

    - état
      - Regroupe les valeurs instantanées de tous les attributs d'un objet :
        - » Un attribut est une information qui qualifie l'objet qui le contient
        - » Chaque attribut peut prendre une valeur dans un domaine de définition donné
      - Exemple : Un objet voiture regroupe les valeurs des attributs couleur, masse et puissance fiscale



Les concepts objet

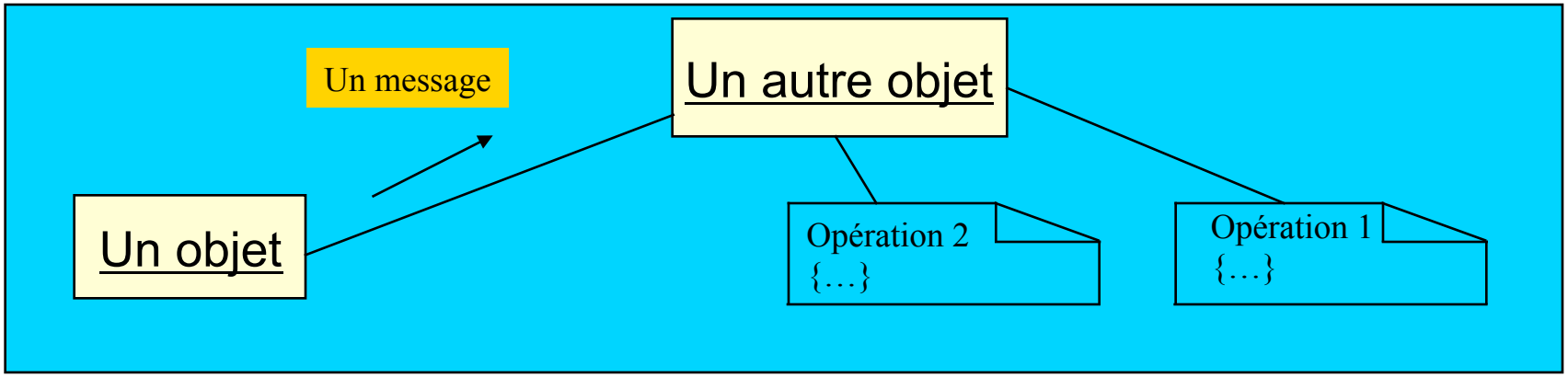
- Objet
  - Caractéristique fondamentales d'un objet (informatique)
    - état
      - L'état d'un objet à un instant donné, correspond à une sélection de valeurs, parmi toutes les valeurs possibles des différent attributs
      - L'état évolue au cours du temps, il est la conséquence de ses comportements passés
        - » Une voiture roule, la quantité de carburant diminue, les pneus s'usent, etc.



      - Certaines composantes de l'état peuvent être constantes
        - La marque de la voiture, pays de la construction de la voiture

Les concepts objet

- Objet
  - Caractéristique fondamentales d'un objet (informatique)
    - Le comportement
      - Regroupe toutes les compétences d'un objet et décrit les actions et les réactions de cet objet
      - Chaque atome (partie) de comportement est appelé opération
        - » Les opérations d'un objet sont déclenchées suite à une stimulation externe, représentée sous la forme d'un message envoyé par un autre objet
        - » L'état et le comportement sont liés

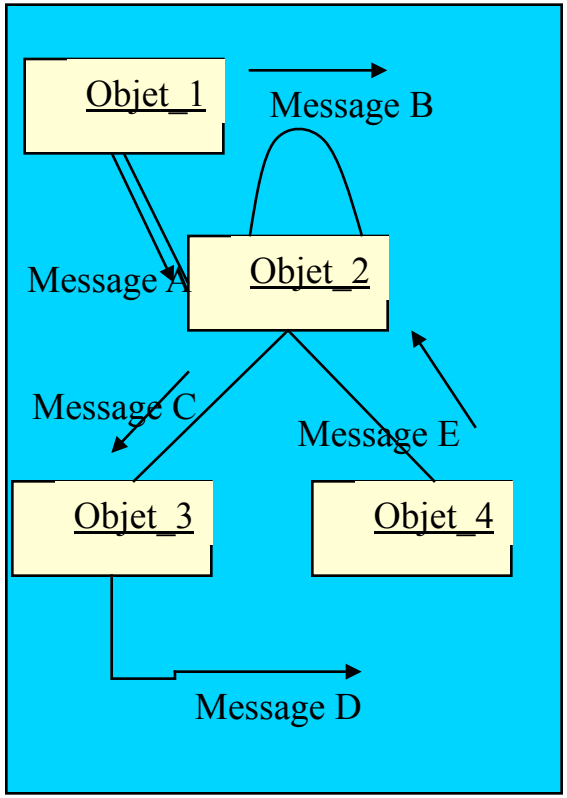


Les concepts objet

- Objet
  - Caractéristique fondamentales d'un objet
    - L'identité
      - Chaque objet possède une identité qui caractérise son existence propre
      - L'identité permet de distinguer tout objet de façon non ambiguë, indépendamment de son état
      - Permet de distinguer deux objets dont toutes les valeurs d'attributs sont identiques
        - » deux pommes de la même couleur, du même poids et de la même taille sont deux objets distincts.
        - » Deux véhicules de la même marque, de la même série et ayant exactement les mêmes options sont aussi deux objets distincts.

Les concepts objet

- Objet
  - Communication entre objets : le concept de message
    - Les programmes à objets peuvent être vus comme des sociétés d'objets qui travaillent en synergie afin de réaliser les fonctions de l'application
    - Le comportement global d'une application repose sur la communication entre les objets qui la composent
    - L'unité de communication entre objets s'appelle message
    - Il existe cinq catégories principales de massages
      - Les constructeurs qui créent des objets,
      - Les destructeurs qui détruisent des objets,
      - Les sélecteurs qui renvoient tout ou partie de l'état de l'objet,
      - Les modificateurs qui changent tout ou partie de l'état d'un objet
      - Les itérateurs qui visitent l'état d'un objet ou le contenu d'une structure de données qui contient plusieurs objets.



Les concepts objet

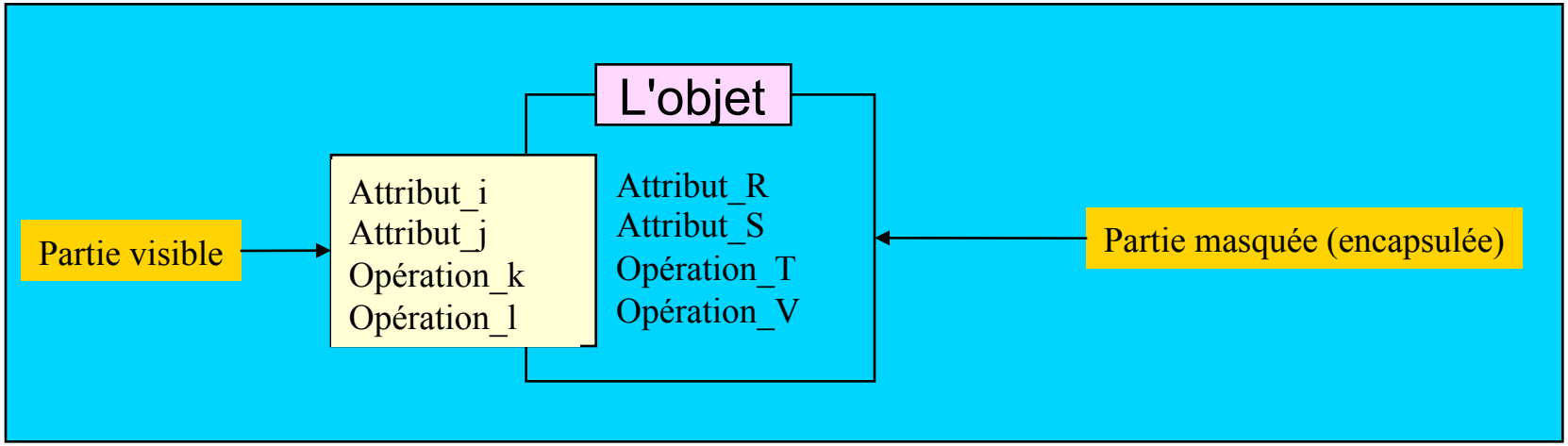
- Classe
  - Définition
    - Une classe décrit une abstraction d'objets ayant
      - Des propriétés similaires
      - Un comportement commun
      - Des relations identiques avec les autres objets
      - Une sémantique commune
    - Par exemple Fichier (resp. Paragraphe, resp. Véhicule) est la classe de tous les fichiers (resp. paragraphes, resp. véhicules)
    - Une classe a trois fonctions:
      - Sert de ``patron'' (*template*) à objets : elle définit la structure générale des objets qu'elle crée en indiquant quelles sont les *variables d'instance* ;
      - Sert de ``conteneur'' d'objets : contient et donc donne l'accès à l'ensemble des objets qu'elle a créés;
      - Sert de ``réceptacle'' des *méthodes* que ses objets peuvent utiliser puisque tous les objets d'une classe utilisent les mêmes *méthodes* , il serait inutile de les dupliquer dans ces objets eux-mêmes.

Les concepts objet

- Classe
  - Caractéristiques d'une classe
    - Un objet créé par (on dit également appartenant à) une classe sera appelé une *instance de cette classe* ce qui justifie le terme ``*variables d'instances* ''
    - les valeurs des *variables d'instances* sont propres à chacune de ces instances et les caractérisent
    - Les généralités sont contenues dans la classe et les particularité sont contenues dans les objets
    - Les objets sont construits à partir de la classe, par un processus appelé instanciation : tout objet est une instance de classe
  - Nous distinguons deux types de classes
    - Classe concrète : peut être instaciée
    - Classe abstraite : est une classe qui ne donne pas directement des objets.

Les concepts objet

- Encapsulation
  - Consiste à masquer les détails d'implémentation d'un objet, en définissant une interface.
    - Est la séparation entre les propriétés externes, visibles des autres objets, et les aspects internes, propres aux choix d'implantation d'un objet.
  - L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.

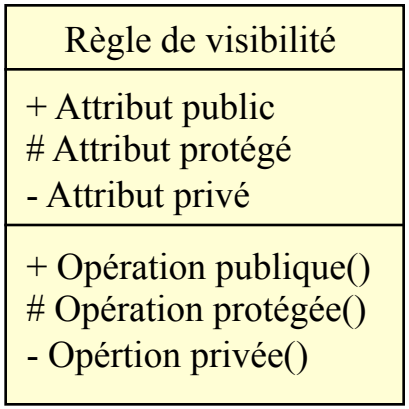


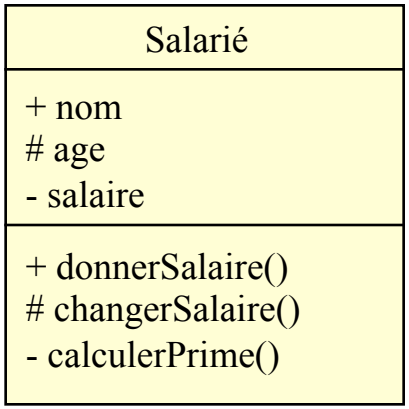
Les concepts objet

- Encapsulation
  - Présente un double avantage
    - facilite l'évolution d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface
      - Les utilisateurs d'une abstraction ne dépendent pas de la réalisation de l'abstraction mais seulement de sa spécification : ce qui réduit le couplage dans les modèles
    - garantit l'intégrité des données, car elle permet d'interdire l'accès direct aux attributs des objets (utilisation d'accesseurs)
      - Les données encapsulées dans les objets sont protégées des accès intempestifs

Les concepts objet

- Encapsulation
  - Il est possible d'assouplir le degré d'encapsulation au profit de certaines classes utilisatrices bien particulières
    - En définissant des niveaux de visibilité
  - Les trois niveaux distincts d'encapsulation couramment retenus sont:
    - Niveau privé : c'est le niveau le plus fort; la partie privée de la classe est totalement opaque
    - Niveau publique : ceci revient à se passer de la notion d'encapsulation et de rendre visibles les attributs pour toutes les classes
    - *Niveau protégé : c'est le niveau intermédiaire ; les attributs placés dans la partie protégée sont visibles par les classes dérivées de la classe fournisseur. Pour toutes les autres classes, les attributs restent invisibles*





Programmation orientée objet en C++

Les classes en C++

- En C++ une classe se déclare par le mot-clé class.
  - Exemple :

```
class Rectangle {
...
    };
```

- La déclaration d'une instance d'une classe se fait de la façon similaire à la déclaration d'une variable classique :
  - nom_classe nom_instance ;
  - Exemple :
    - Rectangle rect1; déclare une instance rect1 de la classe Rectangle.

Déclaration des attributs

- La syntaxe de la déclaration des attributs est la même que celle des champs d'une structure.
  - type nom_attribut ;

- Exemple:
  - les attributs hauteur et largeur, de type double, de la classe Rectangle pourront être déclarés par :

```
class Rectangle {
        double hauteur;
        double largeur;
};
```

Accès aux attributs

- L'accès aux valeurs des attributs d'une instance de nom nom_instance se fait comme pour accéder aux champs d'une structure :

```
nom_instance.nom_attribut
```

  - Exemple: lavaleur de l'attribut hauteur d'une instance rect1 de la classe Rectangle sera référencée par l'expression :

```
rect1.hauteur
```

Déclaration des méthodes

- La syntaxe de la définition des méthodes d'une classe est la syntaxe normale de définition des fonctions :

```
type_retour nom_methode (type_arg1 nom_arg1, ...) {
        // corps de la méthode
        ...
}
```

  - on a **pas besoin de passer les attributs** de la classe comme arguments aux méthodes de cette classe
    - Les attributs d'une classe constituent des variables directement accessibles dans toutes les méthodes de la classes (*i.e.* des variables globales à la classe). Il n'est donc **pas nécessaire de les passer comme arguments des méthodes**.

```
class Rectangle {
        ...
double surface() {
return (hauteur * largeur);
}
 };
```

Actions et Prédicats

- En C++ on peut distinguer les méthodes qui modifient l'état de l'objet (« **actions** ») de celles qui ne changent rien à l'objet (« **prédicats** »).
  - On peut pour cela ajouter le mot const **après** la liste des arguments de la méthode :

```
type_retour nom_methode (type_arg1 nom_arg1, ...) const
```

  - Exemple :

```
class Rectangle { ...
      double surface() const { return (hauteur * largeur);
}
... };
```

Accès aux méthodes

- L'appel aux méthodes définies pour une instance de nom nom_instance se fait à l'aide d'expressions de la forme :

```
nom_instance.nom_methode(val_arg1,...)
```

  - Exemple :

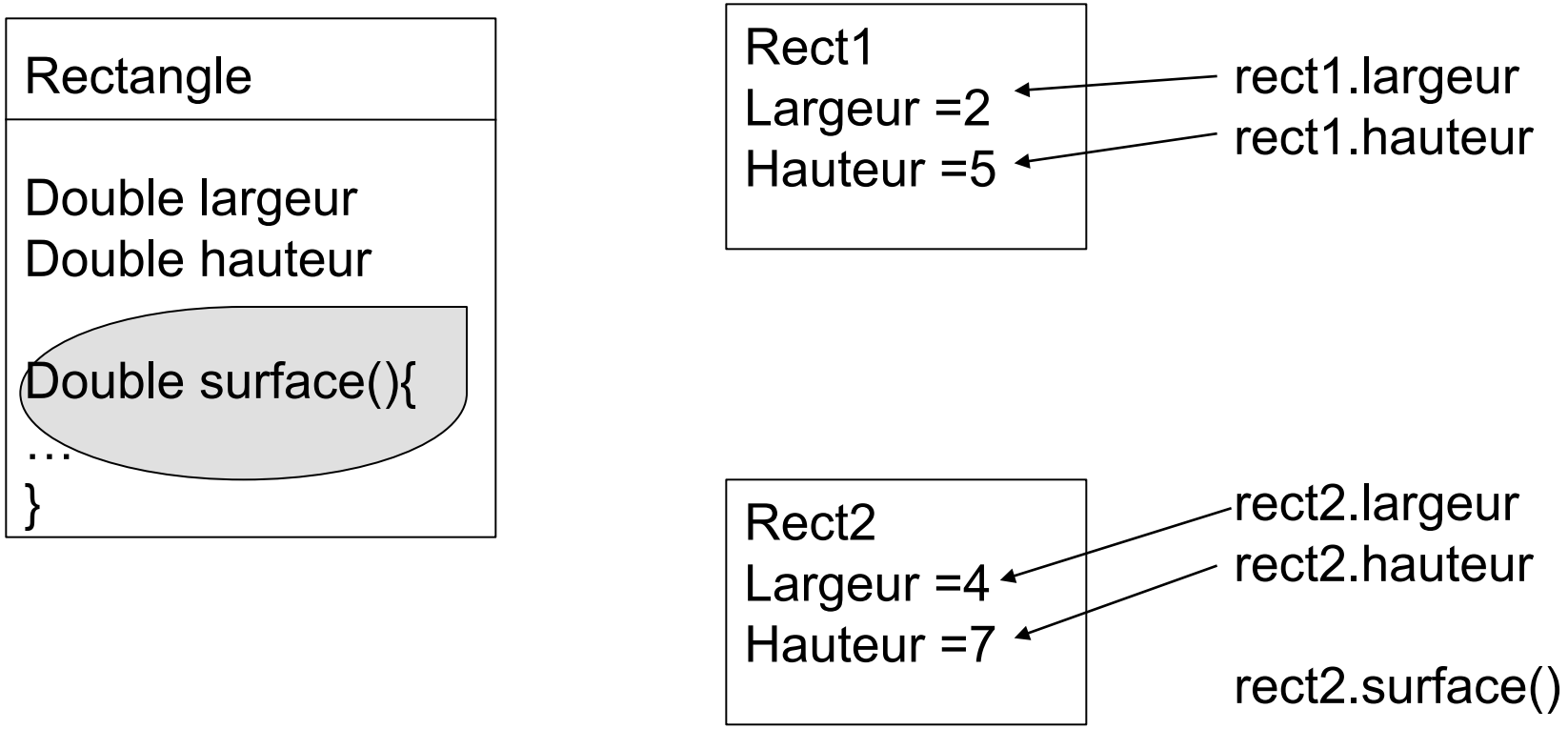
```
void surface() const;
```

    - définie pour la classe Rectangle peut être appelée pour une instance rect1 de cette classe par :

```
rect1.surface()
```

Accès aux attributs et méthodes

- Chaque instance a ses propres attributs : aucun risque de confusion d'une instance à une autre.



Encapsulation et interface

- Tout ce qu'il n'est pas nécessaire de connaître à l'extérieur d'un objet devrait être dans le corps de l'objet et identifié par le mot clé private :

```
class Rectangle {
      double surface() const { ... }
private:
      double hauteur; double largeur;
};
```

- Attribut d'instance **privée** = inaccessible depuis l'extérieur de la classe. C'est également valable pour les méthodes.
  - Erreur de compilation si référence à un(e) attribut/méthode d'instance privée :

```
Rectangle.cc:16: 'double Rectangle::hauteur' is private
```

  - Si aucun droit d'accès n'est précisé, c'est private par défaut.

Encapsulation et interface

- l'interface, qui est accessible de l'extérieur, se déclare avec le mot-clé public:

```
class Rectangle {
public:
      double surface() const { ... }
 private:
      ...
};
```

  - Dans la plupart des cas :
    - **Privé** :
      - Tous les attributs
      - La plupart des méthodes
    - **Publique** :
      - Quelques méthodes bien choisies (interface)

Méthodes «get» et «set»

- **« accesseurs » et « manipulateurs »**
  - Si on a besoin d'utiliser des attributs privés depuis l'extérieur de la classe
    - Si le programmeur *le juge utile*, il **inclut les méthodes publiques comme:**
    - Manipulateurs (« méthodes set ») :
      - Modification
      - Affectation de l'argument à une variable d'instance précise

```
void setHauteur(double h) { hauteur = h; }
void setLargeur(double L) { largeur = L; }
```

    - Accesseurs (« méthodes get ») :
      - Consultation
      - Retour de la valeur d'une variable d'instance précise

```
double getHauteur() const { return hauteur; }
double getLargeur() const { return largeur; }
```

La variable « this »

- « this » est un **pointeur sur l'instance courante**

```
void setHauteur(double hauteur) {
hauteur = hauteur; // ambiguïté
```

```
void setHauteur(double hauteur) {
this->hauteur = hauteur; //  ça marche
}
```

Un exemple complet de classe

```
#include <iostream>
using namespace std;
// définition de la classe
class Rectangle {
public:
// définition des méthodes
double surface() const {
return (hauteur * largeur); }

double getHauteur() const {
 return hauteur; }
double getLargeur() const {
return largeur; }

void setHauteur(double hauteur) {
this->hauteur = hauteur;}

void setLargeur(double largeur) {
this->largeur = largeur; }

private:
 // déclaration des attributs
double hauteur;
double largeur;
};
```

```
//utilisation de la classe
int main() {
Rectangle rect1;
Rectangle rect2;

double lu;
cout << "Quelle hauteur? ";
cin >> lu;
rect1.setHauteur(lu);

cout << "Quelle largeur? ";
cin >> lu;
rect1.setLargeur(lu);
rect2.setLargeur(lu*2);

cout << "surface = " << rect1.surface() << endl;

return 0;
}
```

initialisation des attributs

- Première solution : **affecter individuellement une valeur** à chaque attribut

```
Rectangle rect;
double lu;
cout << "Quelle hauteur? ";
cin >> lu;
rect.setHauteur(lu);
cout << "Quelle largeur? ";
cin >> lu;
rect.setLargeur(lu);
```

- Ceci est une *mauvaise solution* dans le cas général :
  - si le nombre d'attributs est élevé, ou si le nombre d'objets créés est grand (rectangles r[0], r[1], ..., r[99]) elle est fastidieuse et source d'erreurs (oublis)
  - elle implique que tous les attributs fassent partie de l'interface (public) ou soient assortis d'un manipulateur (set)

Initialisation des attributs

- Deuxième solution : définir une **méthode dédiée à l'initialisation** des attributs

```
class Rectangle {
private:
      double hauteur;
      double largeur;
public:
      void init(double h, double L) {
            hauteur = h;
            largeur = L;
      }
...};
```

  - C++ fait déjà ce travail d'initialisation en fournissant des méthodes particulières appelées **constructeurs**
    - Un constructeur réalise toutes les opérations requises en « début de vie » de l'objet.

Les constructeurs

- Un constructeur est une méthode :
  - invoquée *automatiquement* lors de la déclaration d'un objet (instanciation d'une classe)
  - assurant *l'initialisation des attributs*

- Syntaxe de base :

```
NomClasse(liste_arguments) {
/* initialisation des attributs utilisant liste_arguments */
}
```

  - Exemple (à améliorer par la suite) :

```
Rectangle(double h, double L) {
hauteur = h;
largeur = L;
}
```

Les constructeurs

- Les constructeurs sont donc des méthodes (presque) comme les autres
  - que l'on peut surcharger
    - Une classe peut donc avoir **plusieurs constructeurs**, pour peu que leur liste d'arguments soient différentes ;
  - aux arguments desquels on peut donner des valeurs par défaut

- Ce qu'il faut noter de particulier sur la syntaxe des constructeurs :
  - pas d'indication de type de retour (pas même void)
  - doit avoir le même nom que la classe pour laquelle ils sont définis

Initialisation par constructeur

- La *déclaration avec initialisation* d'un objet se fait comme pour une variable ordinaire.
- Syntaxe :

```
NomClasse instance(valarg1, ..., valargN);
```

  – où valarg1, ..., valargN sont les valeurs des arguments du constructeur.

- Exemple :

```
Rectangle r1(18.0, 5.3); // invocation du constructeur
```

Construction des attributs

- Que se passe-t-il si les attributs sont eux-mêmes des objets ?
  - Exemple :

```
class RectangleColore {
private:
Rectangle rectangle;
Couleur couleur;
 //...
};
```

  - Le constructeur d'un rectangleColore devrait faire appel au constructeur de Rectangle, ce qui n'est pas possible directement. On peut alors imaginer passer par une instance anonyme intermédiaire :

```
RectangleColore(double h, double L, Couleur c) {
      rectangle = Rectangle(h, L);
      couleur = c;
}
```

    - Ce n'est pas la solution adoptée en C++.

Appel aux constructeurs des attributs

- Un constructeur devrait normalement contenir une section d'appel aux constructeurs des attributs.
  - Ceci est également valable pour l'initialisation des attributs de type de base.
  - Il s'agit de la « section deux-points » des constructeurs :

```
NomClass(liste_arguments)
// section d'initialisation par constructeurs
: attribut1(...), //appel au constructeur de attribut1
....
attributN(...)
{
// autres opérations
}
```

  - Exemple :

```
RectangleColore(double h, double L, Couleur c)
: rectangle(h, L), couleur(c)
{}
```

Constructeur par défaut

- Un constructeur par défaut est un constructeur qui **n'a pas d'argument** ou dont **tous** les arguments ont des **valeurs par défaut**.
  - Exemples :

```
// Le constructeur par defaut
Rectangle() : hauteur(1.0), largeur(2.0) {}

// 2ème constructeur
Rectangle(double c) : hauteur(c), largeur(2.0*c) {}

// 3ème constructeur
Rectangle(double h, double L) : hauteur(h), largeur(L) {}
```

Destructeur

- SI l'initialisation des attributs d'une instance implique la mobilisation de ressources : *fichiers, périphériques, portions de mémoire (pointeurs), etc.*
  - il est alors important de **libérer ces ressources** après usage

- Comme pour l'initialisation, C++ fourni une méthode appelée *destructeur* invoquée automatiquement en fin de vie de l'instance.
- La syntaxe de déclaration d'un destructeur pour une classe NomClasse est :

```
~NomClasse() {
// opérations (de libération)
}
```

  - Le destructeur d'une classe est une méthode *sans arguments*
    - Son nom est celui de la classe, précédé du signe ~ (tilda).
    - Si le destructeur n'est pas défini explicitement par le programmeur, le compilateur en génère automatiquement une version minimale.

Destructeur

- Supposons que l'on souhaite compter le nombre d'instances d'une classe actives à un moment donné dans un programme.
  - Utilisons comme compteur une variable globale de type entier :
    - le constructeur incrémente le compteur
    - le destructeur le décrémente

```
long compteur=0;

class Rectangle {
//...
Rectangle(): hauteur(0.0), largeur(0.0) { //constructeur
++compteur; }

~Rectangle() {
--compteur; } // destructeur

//...
```

    - on est obligé ici de **définir explicitement le destructeur**

Constructeur de copie

- C++ fournit également un moyen de créer la copie d'une instance : le *constructeur de copie*
- Ce constructeur permet d'initialiser une instance en utilisant les **attributs d'une autre instance** du même type.
  - Syntaxe

```
NomClasse(NomClasse const& obj)
:.... {...}
```

  - Exemple

```
Rectangle(Rectangle const& obj) :
hauteur(obj.hauteur), largeur(obj.largeur)
{}
```

Constructeur de copie

- L'invocation du constructeur de copie se fait par une instruction de la forme :

```
Rectangle r1(12.3, 24.5);
Rectangle r2(r1);
```

  - r1 et r2 sont deux *instances* **distinctes** mais ayant des mêmes valeurs pour leurs attributs (à ce moment là du programme).

- Un constructeur de copie est *automatiquement généré* par le compilateur s'il n'est pas explicitement défini

- Ce constructeur opère une initialisation *membre à membre* des attributs (si l'attribut est un objet le constructeur de cet objet est invoqué)

- Cependant, il est parfois nécessaire de redéfinir le constructeur de copie, en particulier lorsque certains attributs sont des *pointeurs*