

Programmation Orientée Objet

Chapitre : 1

Plan du document

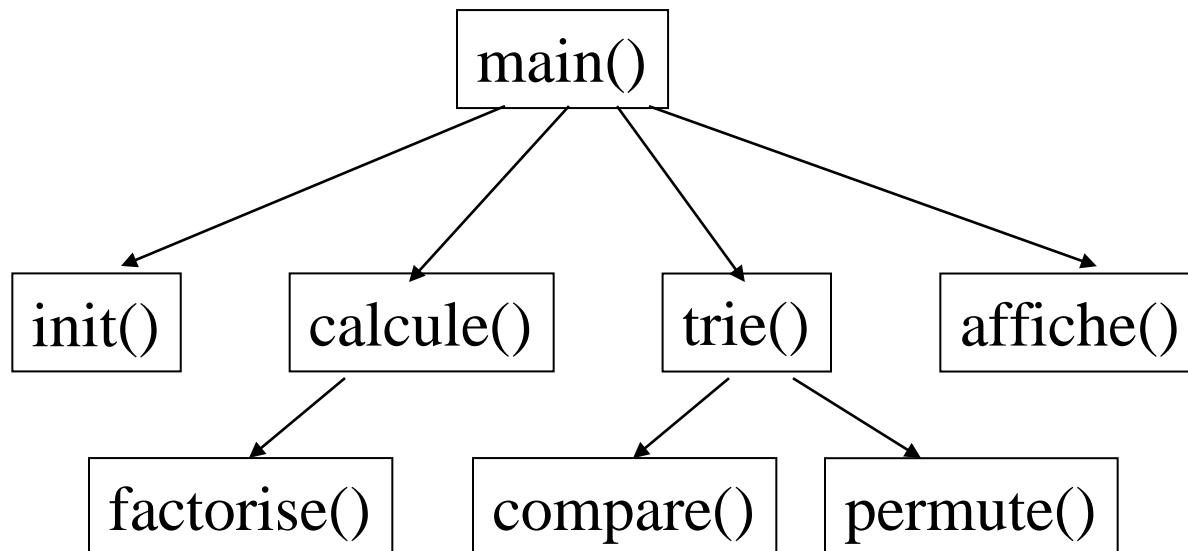
- De la programmation classique à la POO
- Avantages de la POO
- Exemples d'objets
- Objet logiciel
 - État
 - Comportement
 - Identité
- Concept de classe
 - Catégorie
 - Classe
 - Instance
 - Constructeur
- Héritage
- Association

Programmation << classique >>

- Objectif : écrire une séquence d'instructions de telle façon qu'un ordinateur produise une sortie appropriée pour une certaine entrée
- Ingrédients :
 - décomposition fonctionnelle
 - variables
 - passages de paramètres
 - tests
 - boucles
 - structures

Décomposition fonctionnelle

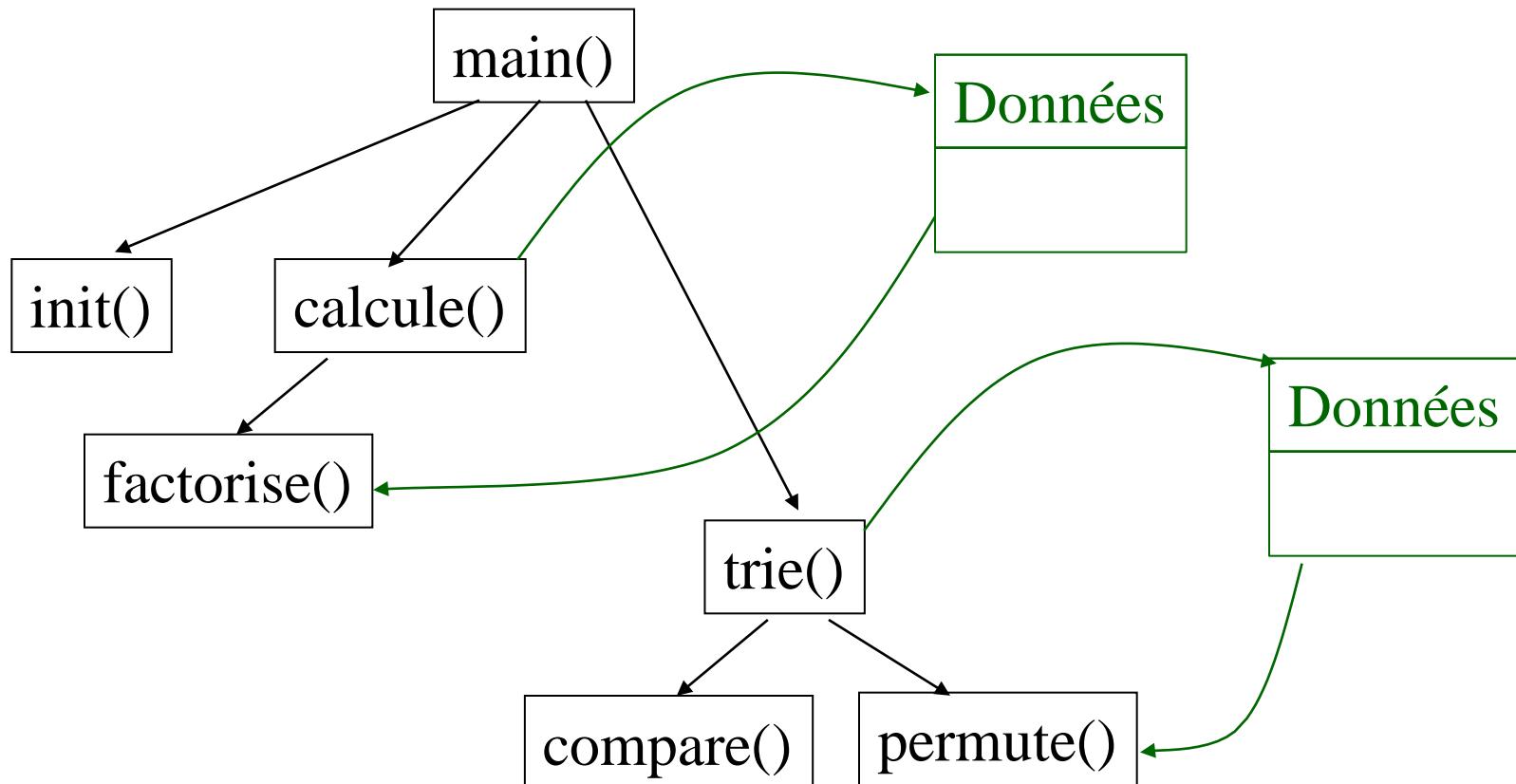
- Le programme est une fonction principale
- int main(argc, argv[])
- Cette fonction appelle des sous-fonctions
- Qui en appellent d'autres...



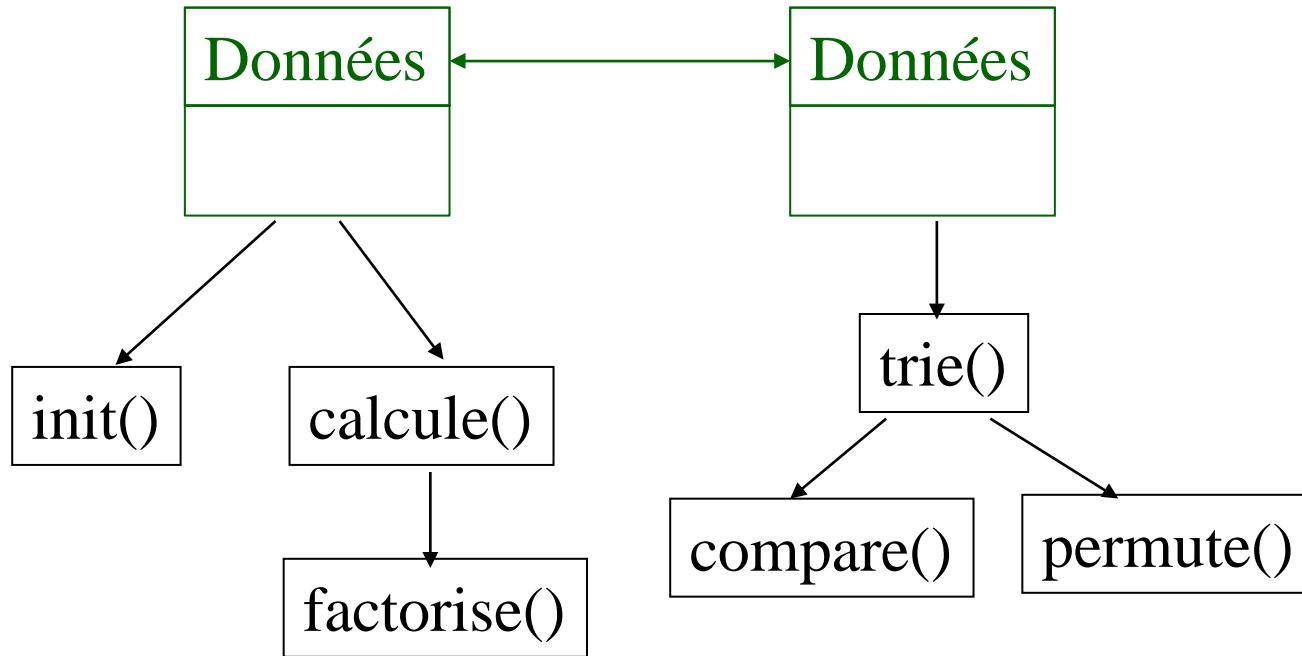
Programmation orientée objets

- Tous les ingrédients de la programmation procédurale classique sont présents
- Mais la décomposition fonctionnelle n'est pas assez structurante pour des programmes très complexes
- Structurer le programme autour des **objets manipulés**
- Les traitements sont associés aux objets
- Séparation plus claire entre données et traitements

Décomposition fonctionnelle classique



Décomposition orientée objets



Avantages de la programmation objet

Habituellement, un programme c'est une suite d'instructions. L'ordinateur est très bête et il faut tout lui détailler. Exemple programme d'ouverture de porte

Mettre la main sur la poignée de la porte
Tourner la poignée

Pousser la porte

Mettre le doigt sur l'interrupteur

Appuyer sur l'interrupteur pour allumer
l'ampoule

Tout se passe très bien. Mais qu'est-ce qui se passe par exemple si on met une porte automatique ?
Le programme sera incapable de trouver la poignée et d'ouvrir la porte !

En programmation objet, on associe aux objets des actions (aussi appelées *méthodes*).

Par exemple, à l'objet **porte** on peut associer la méthode **ouvrir**.

De même pour l'ampoule on pourrait associer une méthode **allumer**, **éteindre**, etc.

Le programme devient plus simple:

```
porte.ouvrir  
ampoule.allumer
```

On a plus besoin de savoir comment la portes'ouvre.

On se contente de l'ouvrir.

Pour indiquer qu'on applique la méthode (ouvrir) sur l'objet (porte), on note souvent **objet.méthode** (ici : porte.ouvrir).

Bien sûr il faut détailler ce que fait la méthode **ouvrir** de **porte** et la méthode **allumer** de **lumière**. On ne va pas détailler dans le programme ce que fait la méthode ouvrir, mais on va le détailler **dans l'objet lui-même**.

C'est normal, puisque la méthode ouvrir ne s'applique qu'à la porte, pas à la lumière:

porte.ouvrir:

Mettre la main sur la poignée
Tourner la poignée
Pousser la porte

On peut changer la porte en mettant une porte automatique.
On peut aussi l'ouvrir (même si la porte elle-même ne s'ouvre pas de la même façon):

porte.ouvrir:

Se placer devant la porte

Attendre que la porte soit complètement ouverte

Mais votre programme pourra l'ouvrir sans rien changer:

```
porte.ouvrir  
ampoule.allumer
```

Le programme principal : il est inchangé malgré le changement de porte

La programmation objet a plusieurs intérêts, entre autres:

Vous pouvez utiliser des objets sans savoir comment ils sont programmés derrière (c'est le cas notre ouverture de porte).

Les objets peuvent être modifiés sans avoir à modifier votre programme (c'est aussi le cas ici).

Les objets sont facilement réutilisables dans de nouveaux programmes.

Les langages objets offrent des mécanismes pour permettre ce genre de programmation.

Qu'est-ce qu'un objet ?

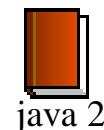
Objets de la vie courante



garfield



Rêve



007BEJ06



45BEJ91



123CDE91



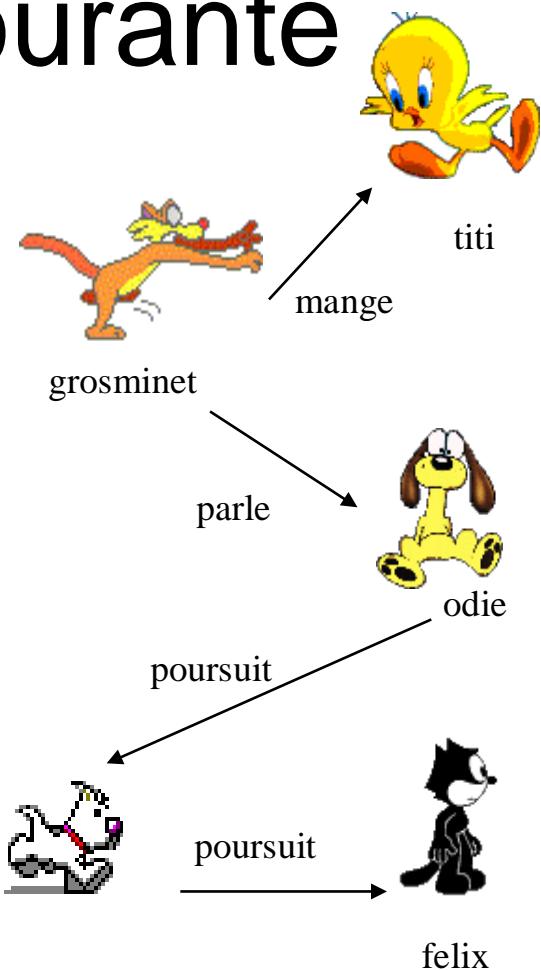
0102030405



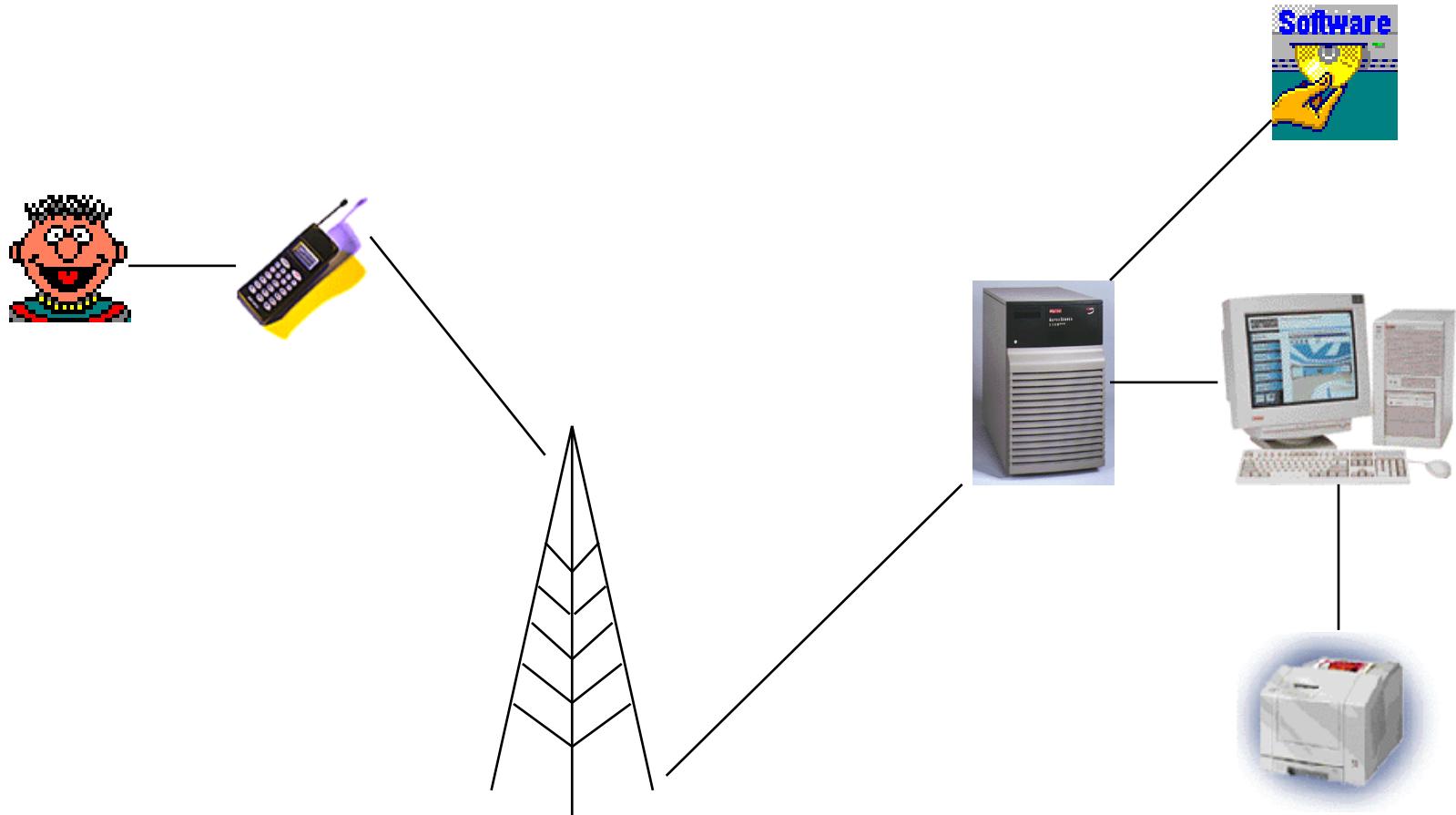
Dupond Dupont



0203040506



Les objets coopèrent



Objet boîte noire

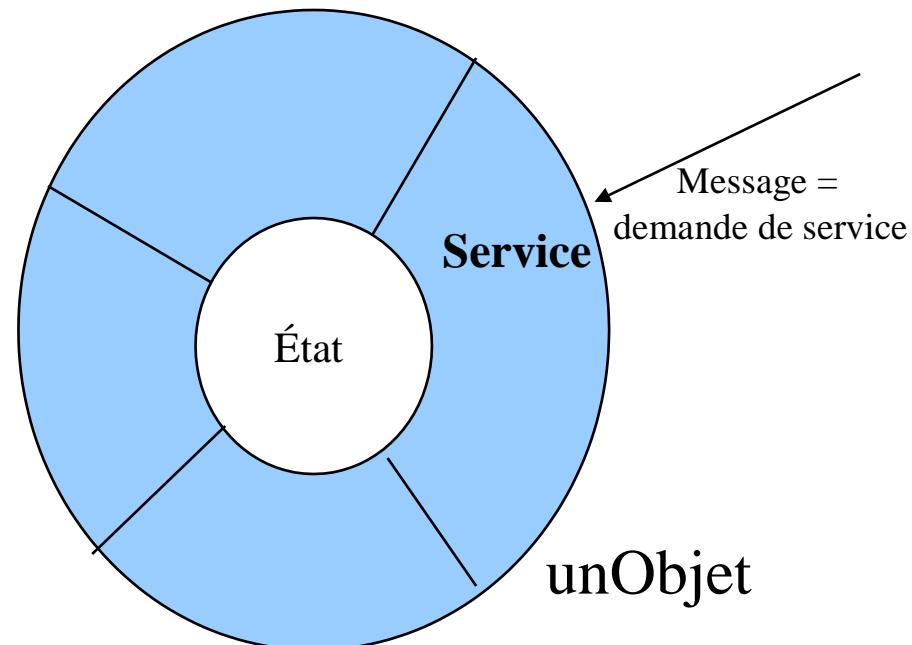


- Services rendus par l'objet :
 - Démarrer ;
 - Arrêter ;
 - Accélérer ;
 - Freiner ;
 - Climatiser ...
- Fonctionnement interne ???

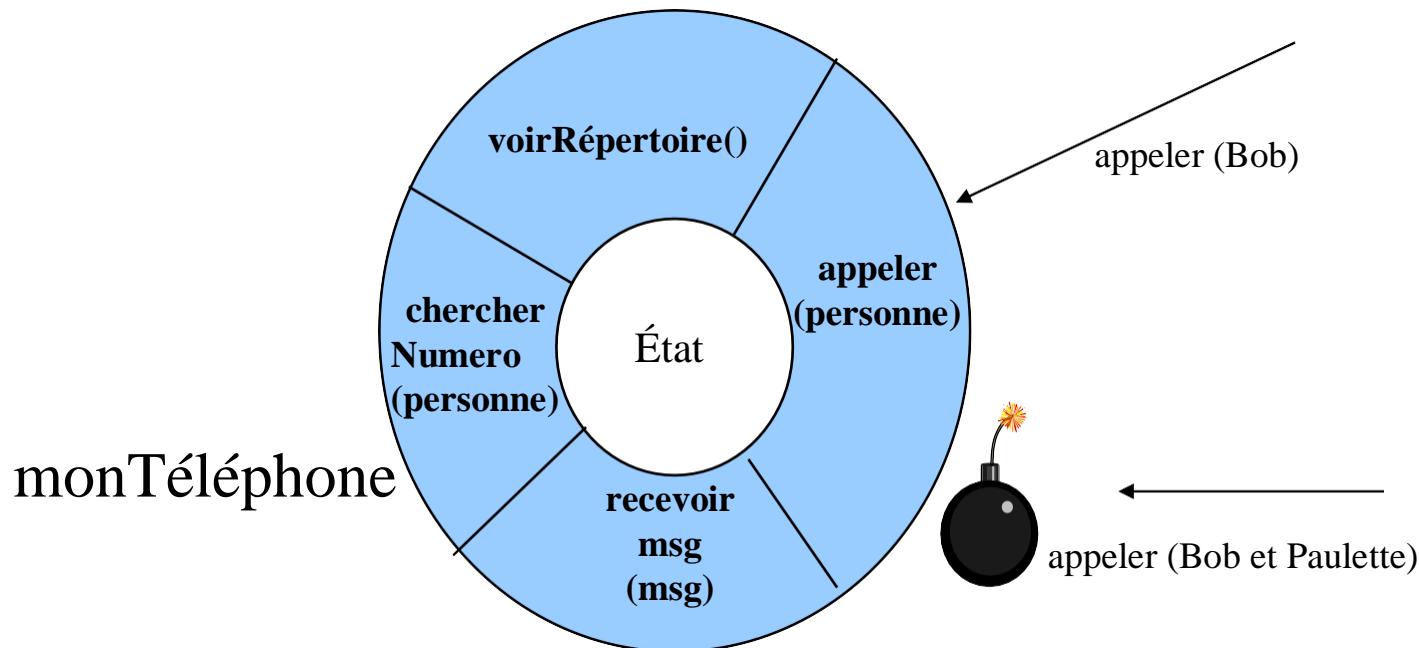
Objet logiciel

- Abstraction
 - Représentation abstraite des entités du monde réel ou virtuel dans le but de les piloter ou de les simuler
- Programme, logiciel

Objet =
État +
Comportement +
Identité



Téléphone portable



État d'un objet

- Attribut
 - Information qui qualifie l'objet qui le contient
 - Peut être une constante
 - État
 - Valeurs instantanées de tous les attributs d'un objet
 - Évolue *autour du temps*
 - Dépend de l'*histoire de l'objet*
- Identité d'un objet

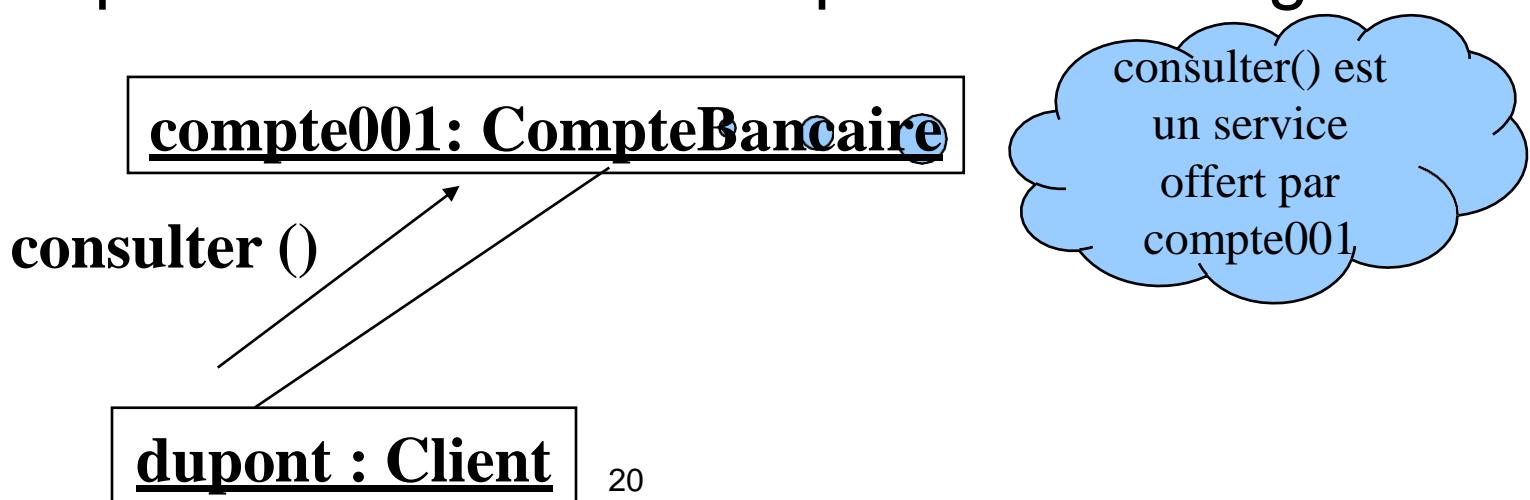
compte001 : CompteBancaire

 - solde
 - DEBITAUTORISE

attribut constant

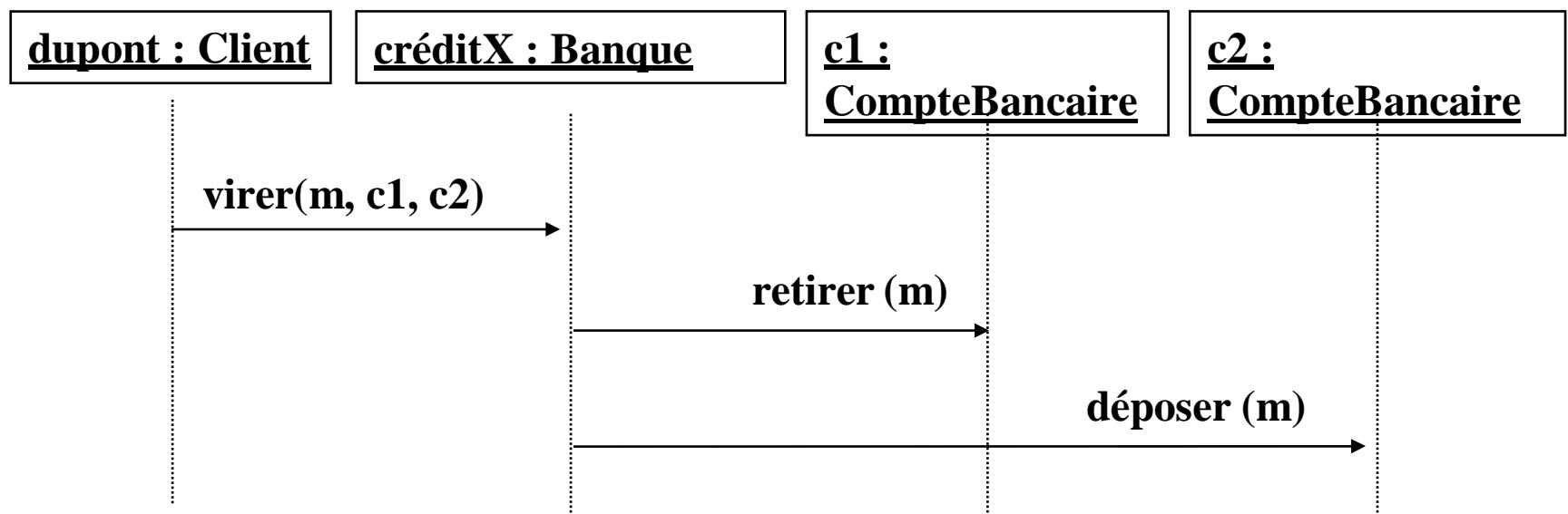
Comportement d'un objet

- Décrit les actions et les réactions d'un objet
 - Compétences d'un objet
- Service = opération = méthode
 - Comportement déclenché par un message



Les objets communiquent

- Permet de reconstituer une fonction par une mise en collaboration d'un groupe d'objets : envois de messages successifs



Identité d'un objet

- Caractérise son existence propre
- Indépendant du concept d'état
- Permet de désigner tout objet de façon non ambiguë

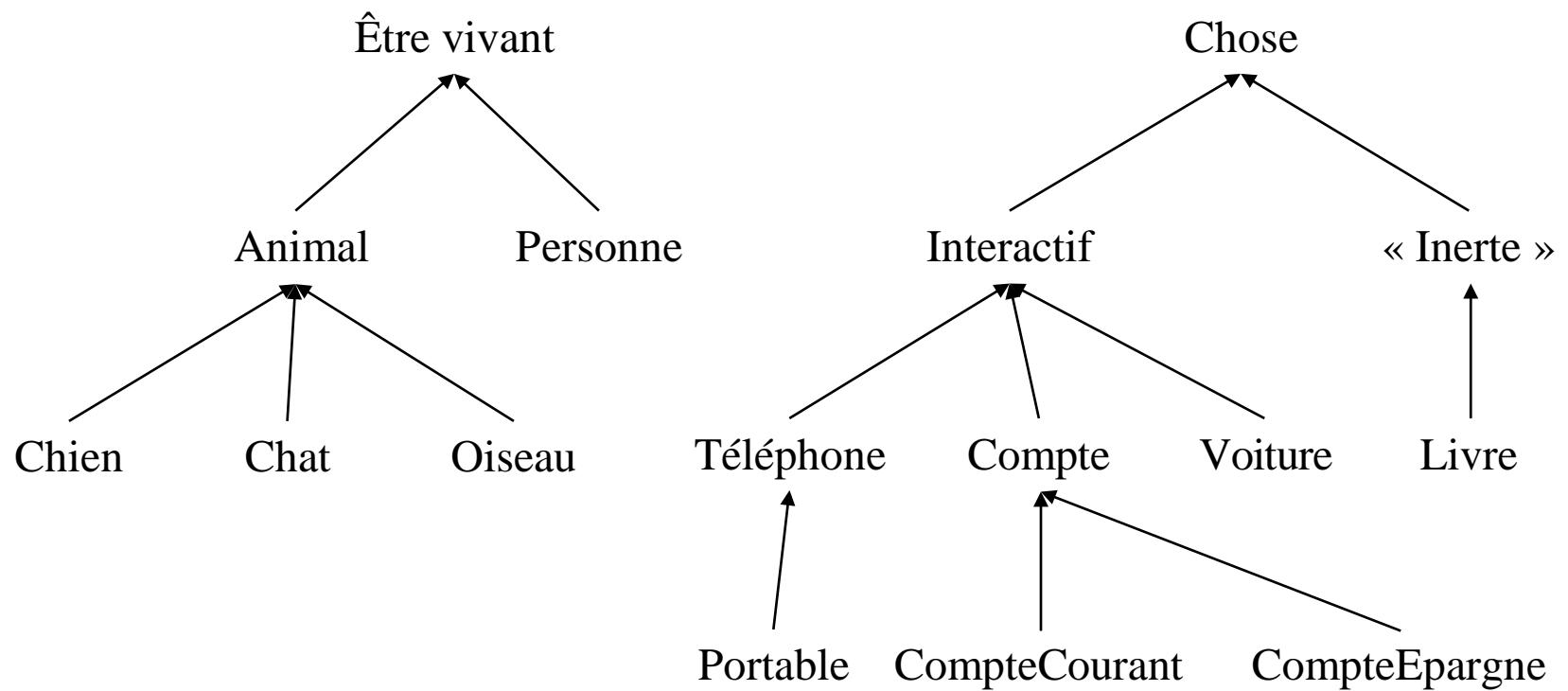
compte001 :CompteBancaire

1000

compte112 :CompteBancaire

1000

Catégoriser les objets (1)



Catégoriser les objets (2)

- Constituer des groupes d'objets ayant les mêmes attributs (pas les mêmes valeurs !)
 - tous les animaux ont un poids, une taille, ...
- Prévoir leur comportement
 - on sait se servir d'un téléphone, quel qu'il soit
- Faire évoluer tous les objets d'un groupe simultanément
 - ajouter un attribut e-mail aux coordonnées des personnes dans un carnet d'adresse

Classe

- Description d'une famille d'objets
 - Mêmes attributs
 - Même méthodes
- Spécification
- Réalisation (implantation)
 - Décrit comment la spécification est réalisée
- Générateur d'objets = moule

}

Classe CompteBancaire

CompteBancaire

CompteBancaire
solde
déposer() retirer()

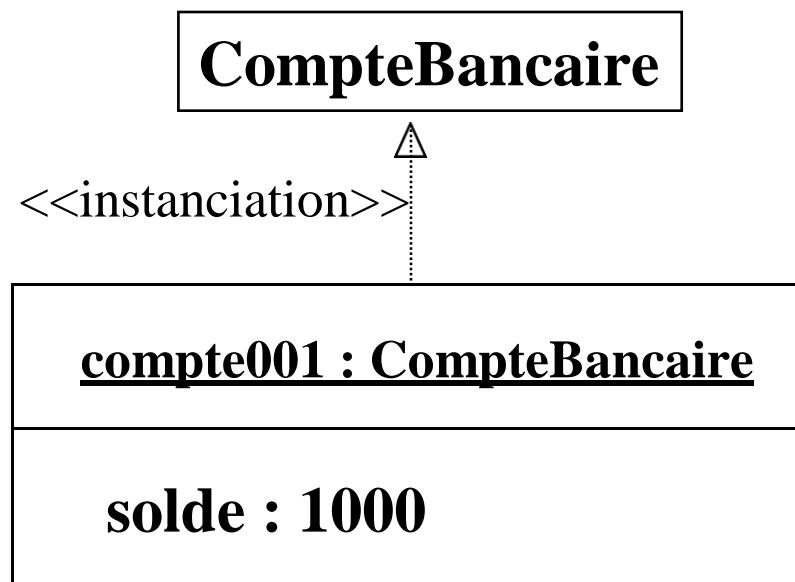
Nom de la classe

attributs

opérations

Instance

- Chaque objet appartient à une classe
- Relation d'instanciation « instance de »

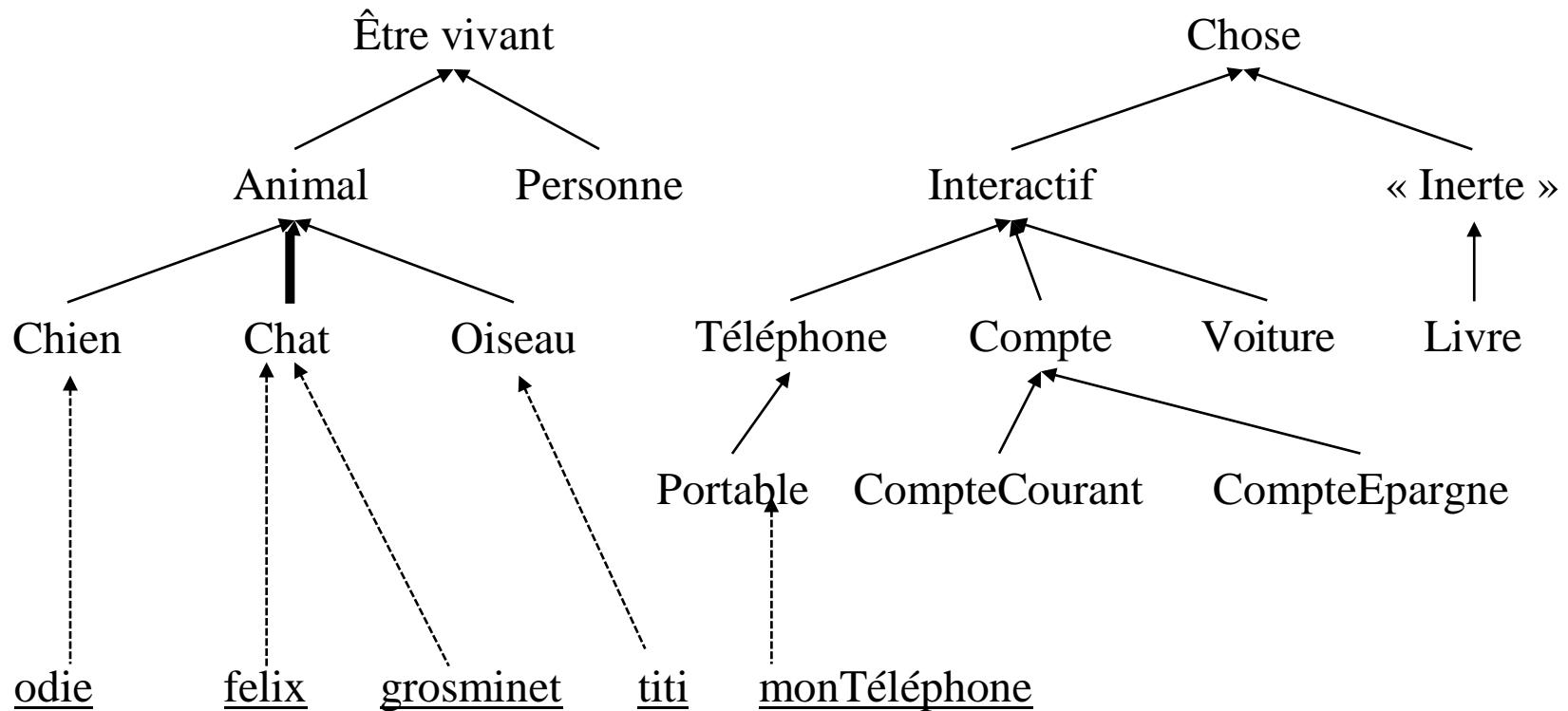


Classe

Relation d'instanciation

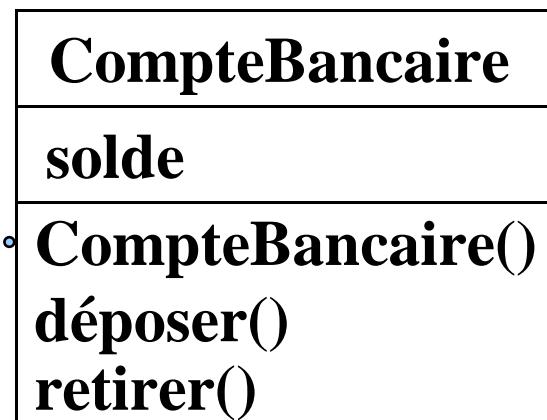
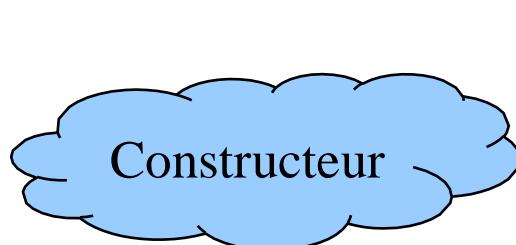
Instance

Les objets sont rattachés à leur classe



Constructeur

- Un objet doit être créé au sein d'une classe == moule
- Méthode particulière : constructeur
 - Construit l'objet avec ses attributs, ses méthodes
 - Initialise les valeurs des attributs
 - Nom du constructeur = Nom de la classe

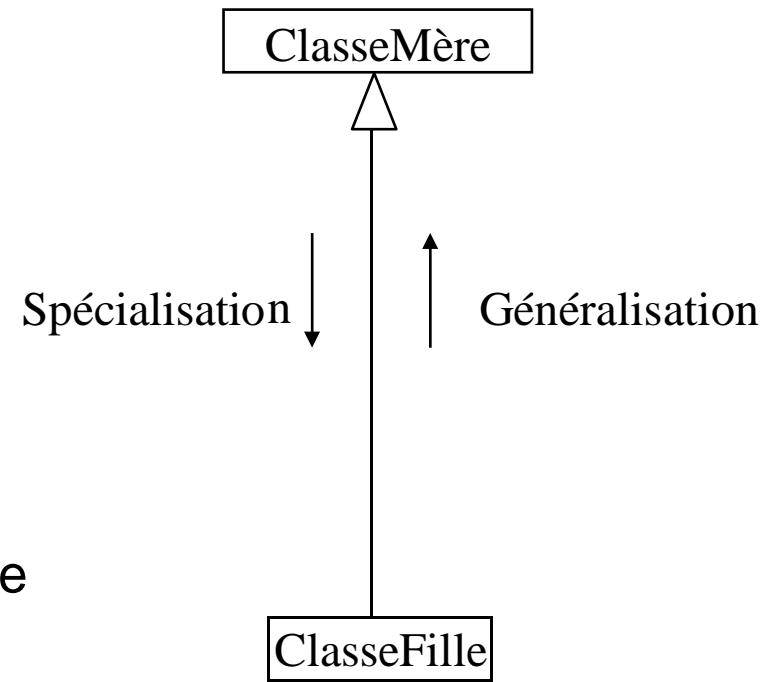


Nom de la classe
attributs
opérations

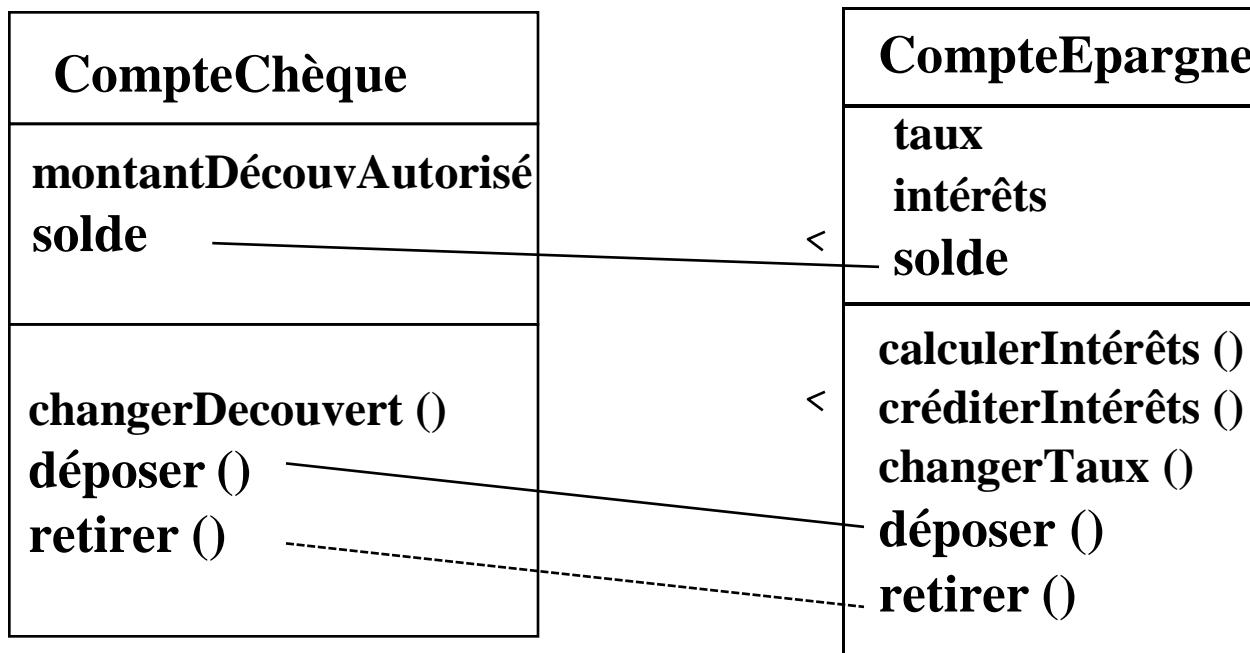
Héritage: les concepts

Généralisation et spécialisation

- Généralisation
 - Factoriser les éléments communs (attributs, opérations...) d'un ensemble de classes
 - Simplifier les diagrammes (regroupements sémantiques)
 - « est un », « est une sorte de »
- Spécialisation
 - Utiliser les caractéristiques d'une classe déjà identifiée et en spécifier de nouvelles.
- Hiérarchie de classe
 - Arborescence des héritages

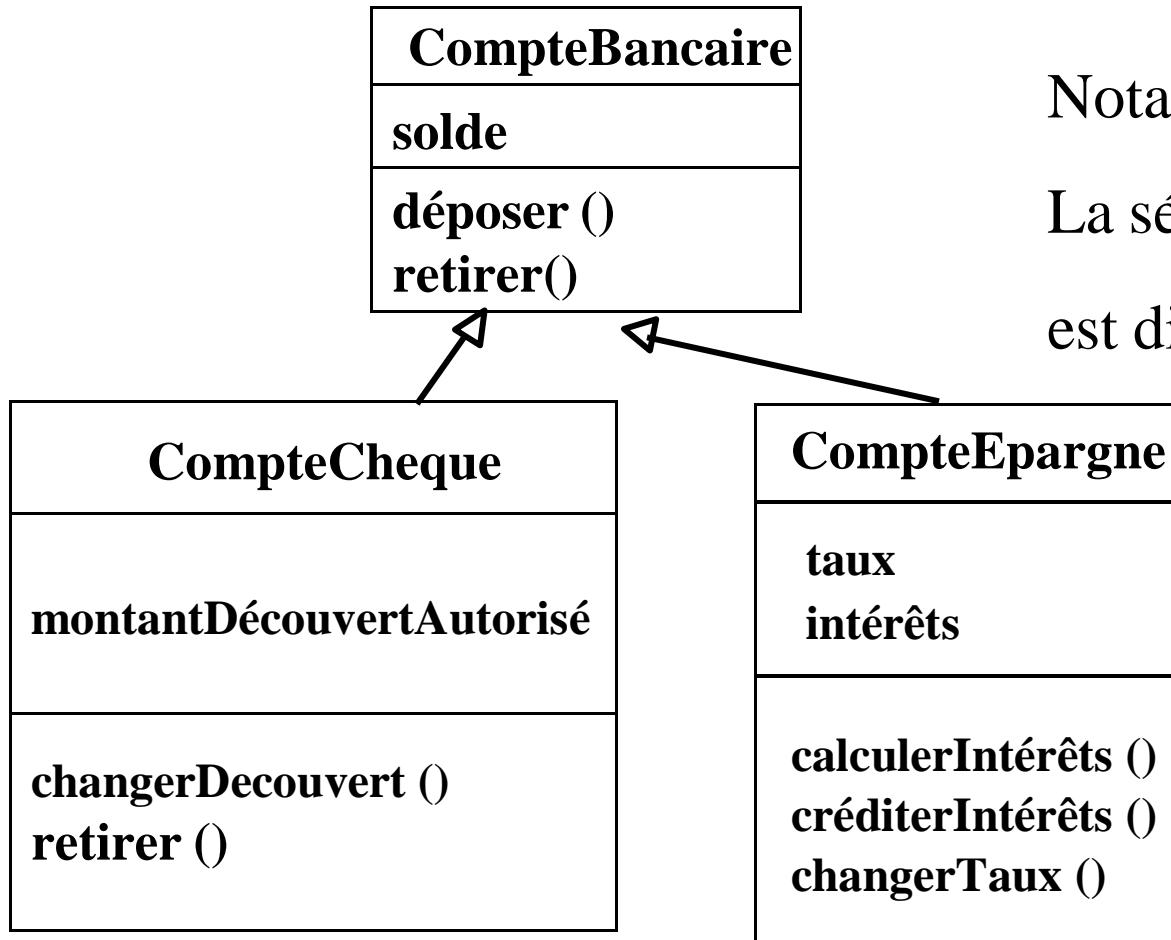


Généralisation



Constat: des caractéristiques communes

Généralisation

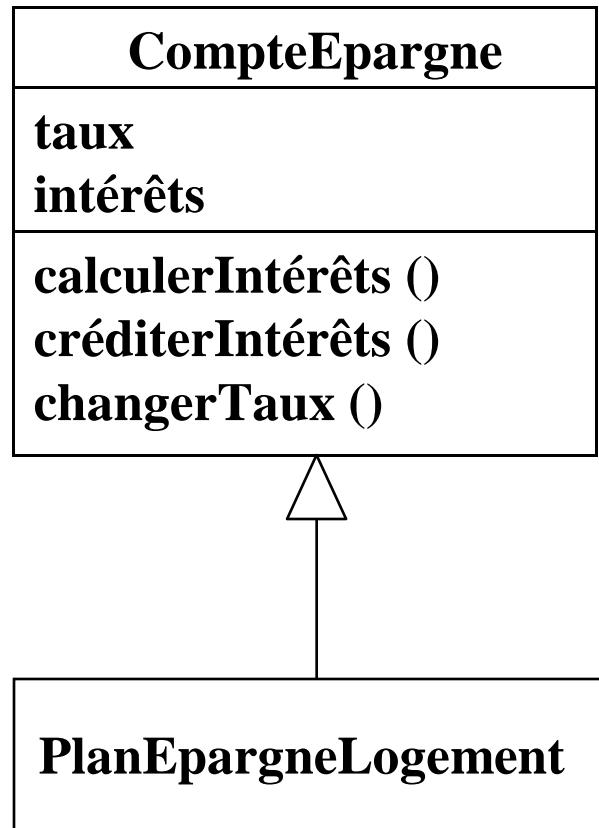
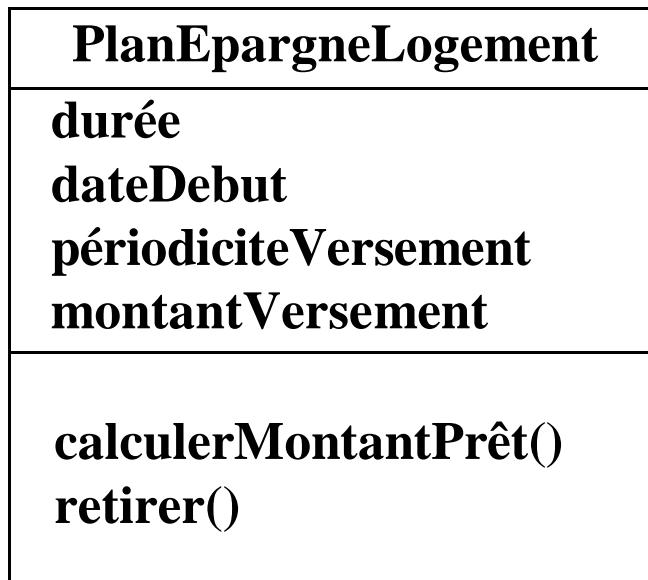


Nota :

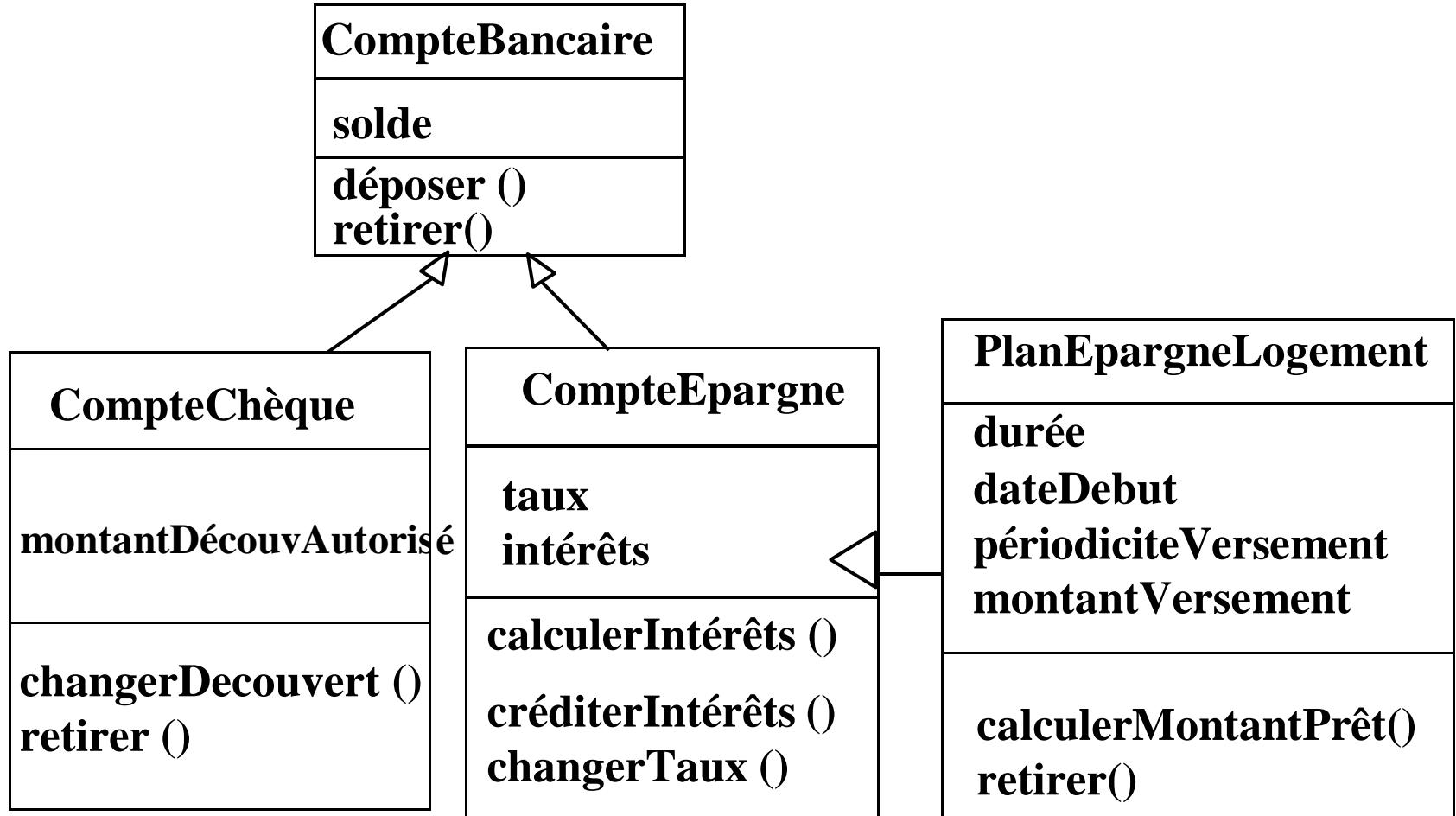
La sémantique de *retirer*
est différente

Spécialisation

Un PlanEpargneLogement
est un CompteEpargne ayant
des caractéristiques propres



Arborescence d'héritage



Redéfinition d'une méthode

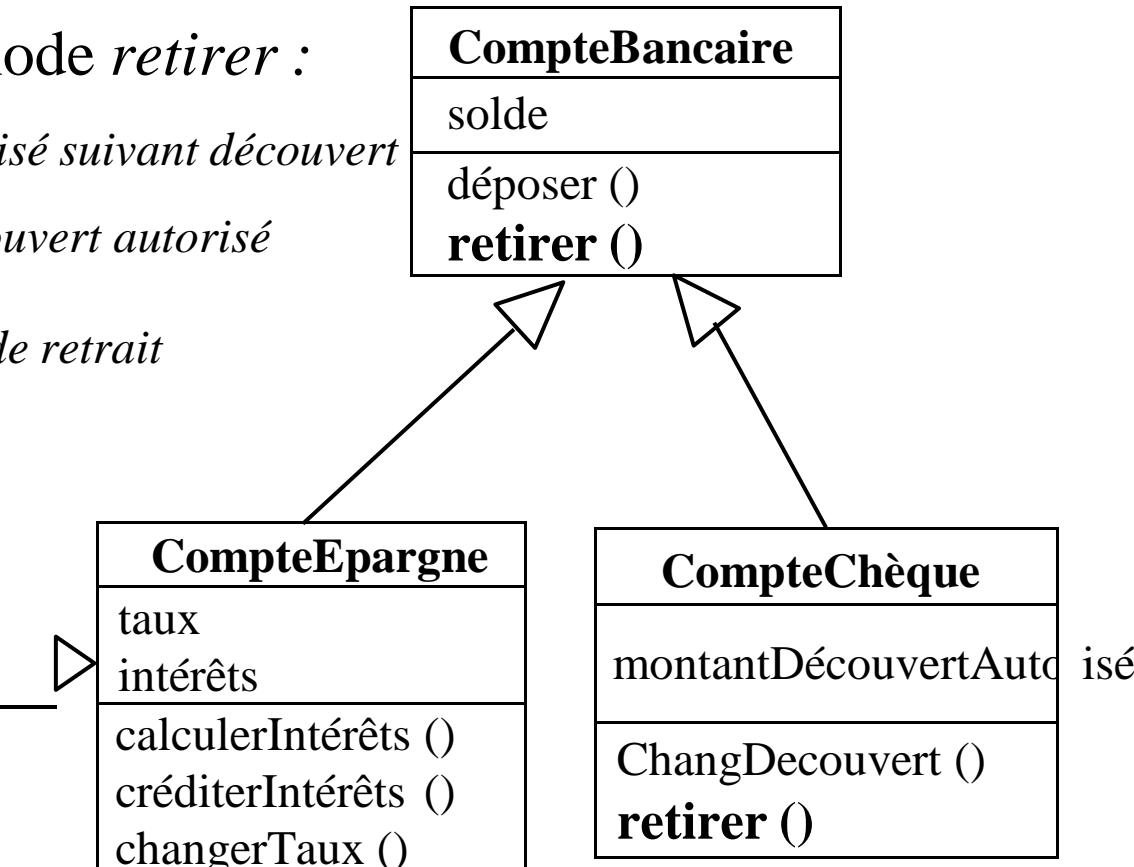
Sémantique de la méthode *retirer* :

CompteChèque : retrait autorisé suivant découvert

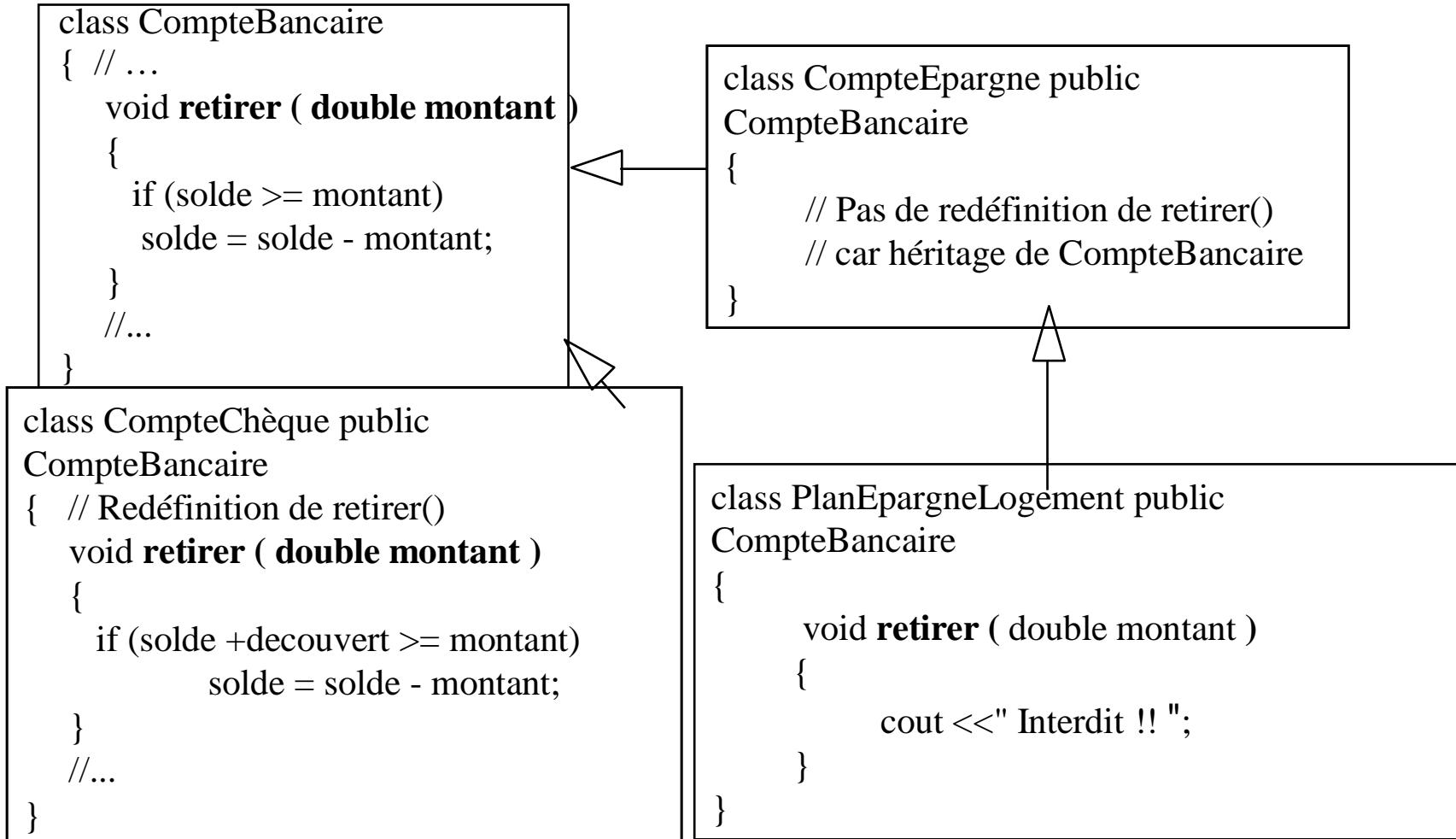
CompteEpargne : pas de découvert autorisé

PlanEpargneLogement : pas de retrait

PlanEpargneLogement	
durée	
dateDebut	
périodicitéVersement	
montantVersement	
calculerMontantPrêt()	
retirer ()	



Redéfinition d'une méthode



Héritage et sous classes

- Héritage
 - Mécanisme permettant de dériver une classe à partir d'une classe existante
 - Permet de classer
 - Etendre structurellement et comportementalement une classe
 - Permet de construire
 - Réutiliser le code d'une classe existante
- Classe dérivée
 - Hérite de toutes les caractéristiques de son (ses) ancêtre(s)
 - Peut avoir des caractéristiques propres
 - Peut redéfinir des caractéristiques héritées

Associer les classes : Les concepts

Conseils pour placer une opération dans une classe

- Rechercher la classe responsable
 - Elle détient les informations nécessaires pour effectuer l'opération
- Préserver la cohésion interne des classes
 - Les opérations doivent former un ensemble cohérent
 - Eviter de placer toutes les opérations dans la même classe
- Limiter les interactions entre les classes
 - Les dépendances entre les classes doivent être minimales et se limiter à ce qui est indispensable

Association



Une agence gère plusieurs clients.

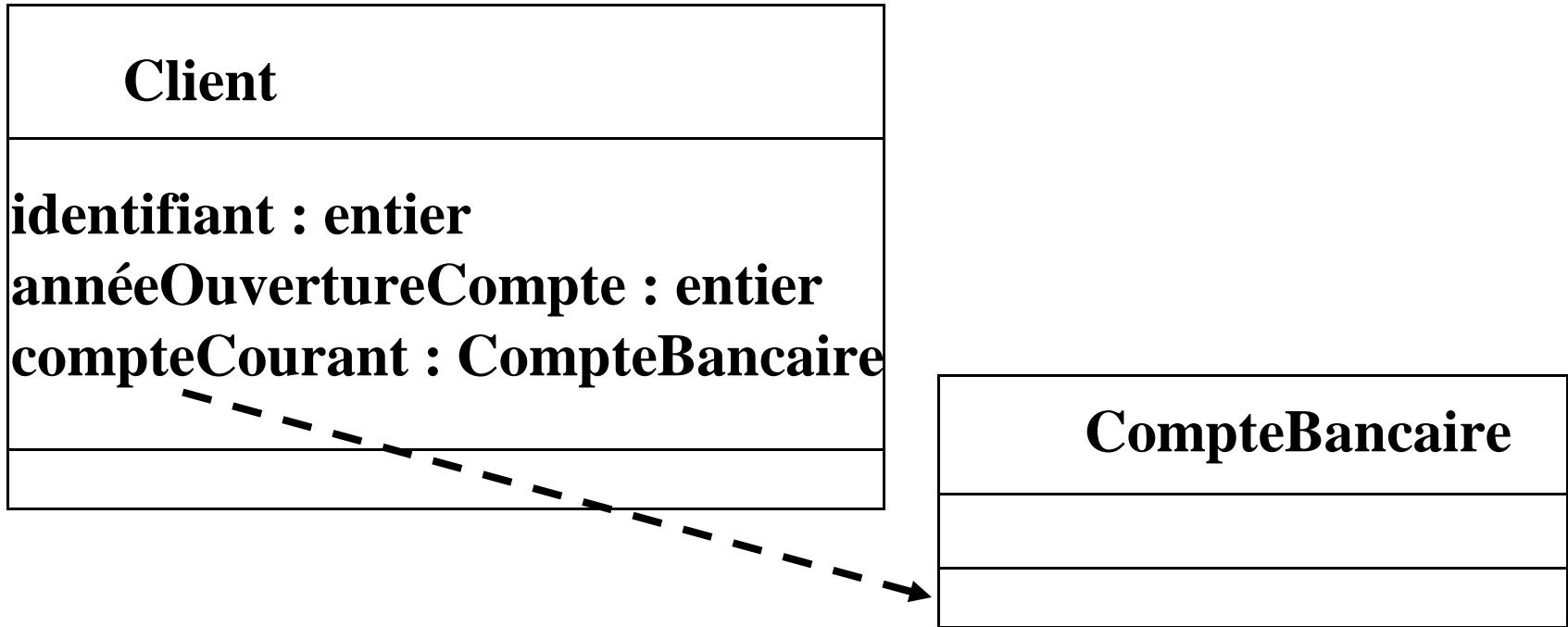
Un client possède un et un seul compte courant.

Multiplicités les plus courantes



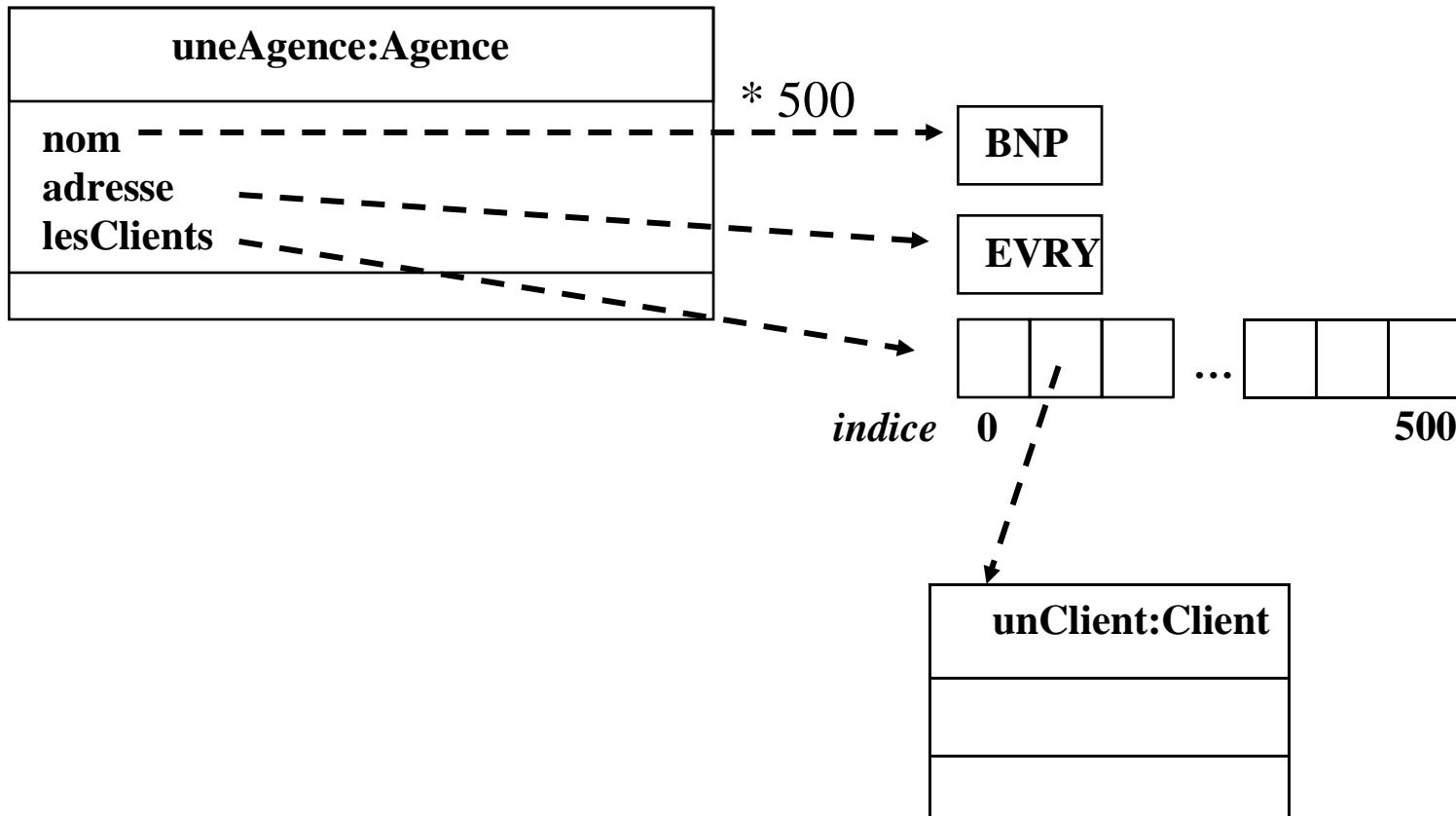
1	Un et un seul
0..1	Zéro ou un
M..N	De M à N (entiers naturels)
*	De zéro à plusieurs
0..*	De zéro à plusieurs
1..*	D'un à plusieurs

Association simple: une référence comme attribut

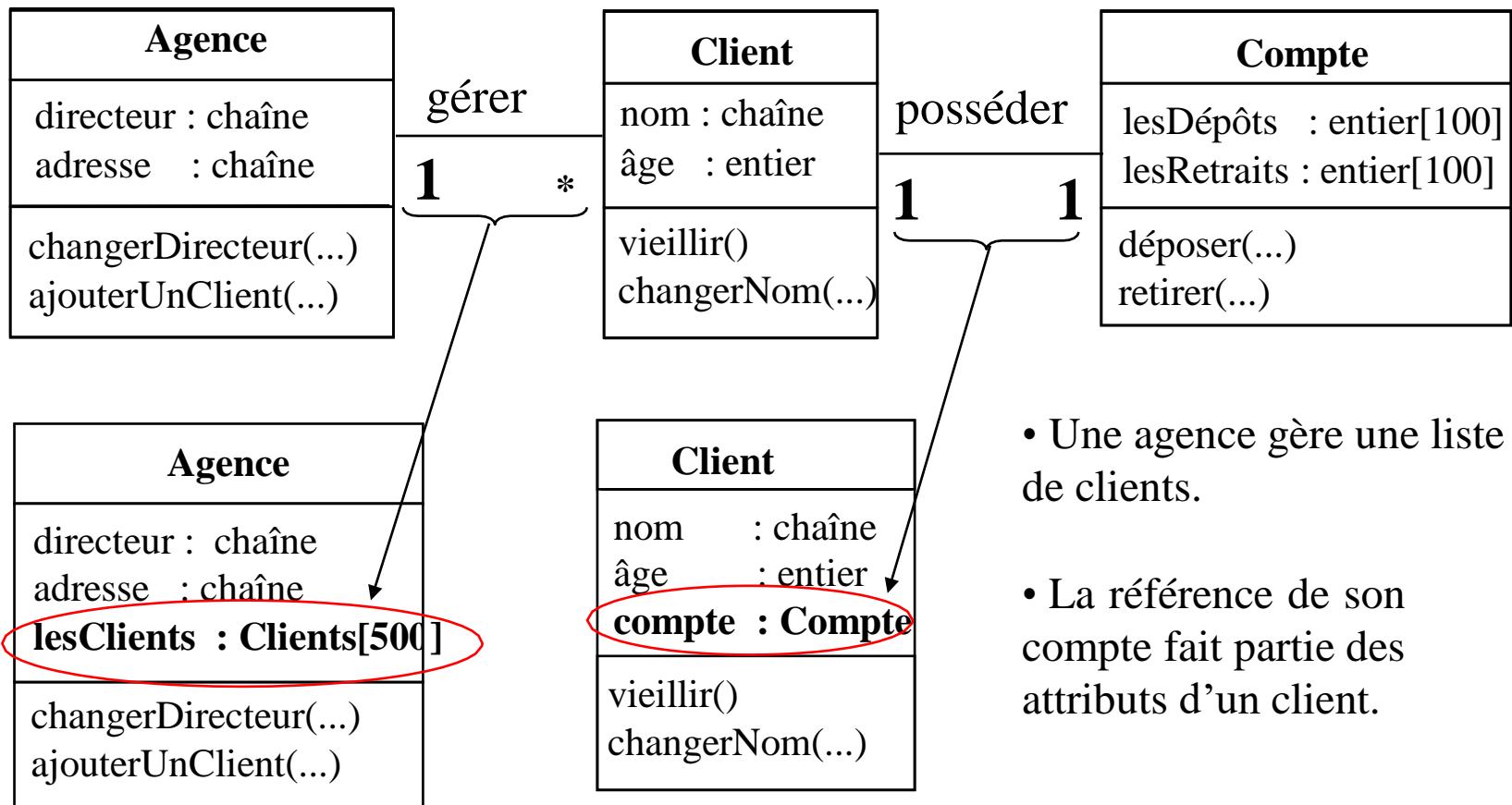


- L'attribut **compteCourant** est une référence sur un objet de la classe **CompteBancaire**

Association multiple : Tableau de références sur des objets



Une traduction d'associations



Modéliser avec des classes : résumé

- Des classes pour les concepts
 - Structure de classe = modèle du monde réel
 - Classe = représentant d'une catégorie d'objets réels ou abstraits
- Prise en compte des relations
 - Associations
 - Généralisation / spécialisation : héritage

Programmation en C++

C++=C+E+S+0

Meilleur C

Le langage C++ se veut un langage C amélioré.
Il possède des fonctionnalités supplémentaires, et
notamment

- * la surcharge de fonctions
- * le passage par référence
- * l'emplacement des déclarations
- * l'allocation dynamique

Les apports spécifiques de C++ sont

- * l'aide à l'abstraction de données: définition de types de données, et de leur implémentation concrète.
- * l'aide à la programmation objet: hiérarchie de classes et héritage.

Incompatibilités entre C et C++

Toute fonction doit

- être définie avant utilisation
- ou être déclarée par un prototype

float fct (int, double, char*);

(En C, une fonction non déclarée est supposée de type de retour int.)

Une fonction qui ne retourne pas de valeur a le type de retour void.

Le qualificatif const peut être utilisé pour une expression

Constante nécessaire pour la taille d'un tableau

```
const int N = 10; // remplace #define N 10  
int valeurs[N];
```

Entrées-sorties

Les entrées et sorties sont gérées dans C++ à travers des objets particuliers appelées streams ou flots. Inclure <iostream.h>

Deux opérateurs sont surchargés de manière appropriée pour les flots:

- * l'opérateur d' insertion << (écriture)
- * l'opérateur d' extraction >> (lecture)

Trois flots prédéfinis sont

- * cout attaché à la sortie standard;
- * cerr attaché à la sortie erreur standard;
- * cin attaché à l'entrée standard;

E/S exemple 1

```
#include <iostream.h>
main() {
    cout << "Bonjour, monde !\n";
}
```

Plusieurs expressions:

```
cout << ex_1 << ex_2 << ... << ex_n ;
```

Plusieurs "lvalues":

```
cin >> lv_1 >> lv_2 >> ... >> lv_n ;
```

Les types écrits ou lus sont

char, short, int, long, float, double, char*

Exemple

```
#include <iostream.h>
int i;
main() {
    cout << "Un entier : ";
    cin >> i;
    cout << "Le carre de " << i << " est " << i*i << endl;
}
```

Commentaires

```
/* commentaire  
// commentaire  
commentaire  
*/  
// commentaire
```

Emplacement des déclarations

Une déclaration peut apparaître en n'importe quelle position d'un corps de classe ou de fonction, mais doit précéder son utilisation.

```
int n;
```

```
n = 3;
```

```
int q = 2*n-1;
```

```
for (int i = 0; i<n ; i++) {...} // déconseillé
```

Arguments par référence

Enfin le passage par référence qui manque au C.

Un paramètre dont le nom est suivi de & est transmis par référence, donc pas de copie de l'argument à l'appel; possibilité de modifier l'argument.

Déclaration

```
void echange(float&, float&);
```

Définition

```
void echange(float& a, float& b) {
    float t = a; a = b; b = t;
}
```

Exemple

```
float x = 2, y = 3;
echange (x, y);
cout << x << "," << y; //Affiche 3,2
```

Références Constantes

Passage par référence constante pour ne pas copier l'argument à l'appel; ne pas modifier l'argument :

void afficher(const objet&);

Passage par référence constante permet d'assurer l'encapsulation en évitant la copie sur la pile de structures trop grande.

Arguments par défaut

Les derniers arguments d'une fonction/méthode peuvent prendre des "valeurs par défaut".

Déclaration

```
float f(char, int = 10, char* = "Tout");
```

Appels

```
f(c, n, "rien")
```

```
f(c, n) // <-> f(c, n, "Tout")
```

```
f(c) // <-> f(c,10, "Tout")
```

```
f() // erreur
```

Seuls les derniers arguments peuvent avoir des valeurs par défaut.

```
float f(char = 'a', int, char* = "Tout"); // erreur
```

Surcharge

Un même identificateur peut désigner plusieurs fonctions, si elles diffèrent par la liste des types de leurs arguments.

```
float max(float a, float b) { return (a > b) ? a : b;}  
float max(float a, float b, float c) { return max(a, max(b, c));}  
float max(int n, float t[]) {  
    if (!n) return 0;  
    float m = t[0]; for (int i = 1 ; i < n; i++)  
        m = max(m, t[i]);  
    return m;  
}  
void main() {  
    float x, y, z;  
    float T[] ={11.1, 22.2, 33.3, 44.4, 7.7, 8.8 };  
    x = max (1.86, 3.14);  
    y = max (1.86, 3.14, 37.2);  
    z = max (6, T);  
    cout << x << " " << y << " " << z;  
}
```

Allocation dynamique

Deux Opérateurs Intégrés au langage.

Les opérateurs new et delete gèrent la mémoire dynamiquement.

new chose[n]

alloue la place pour éléments de type chose et retourne l'adresse du premier élément;

delete adresse

libère la place allouée par new.

```
int* a = new int; // malloc(sizeof(int))
```

```
double* d = new double[100];
```

```
// malloc(100*sizeof(double))
```

Fonctions/Méthodes « en ligne »

Une fonction en ligne (inline) est une fonction dont les instructions sont incorporées par le compilateur dans le module objet à chaque appel. Donc il n'y pas d'appel : gestion de contexte, gestion de pile; Déclaration par qualificatif inline.

inline int sqr(int x) { return x*x; }

[les méthodes définies dans le corps de classe sont inlinées]

[les fonctions trop grandes ou récursives ne sont pas inlinées]

Programmation Orientée Objet

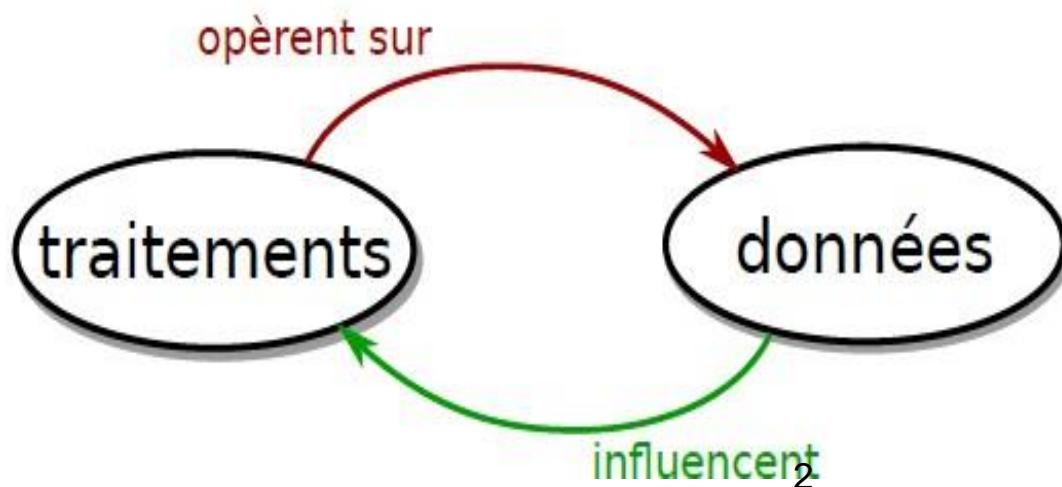
1

Programmation impérative/procédurale (rappel)

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 :



Programmation procédurale : exemple

```
double surface(double largeur, double hauteur);

int main()
{
    double largeur(3.0);
    double hauteur(4.0);

    cout << "La surface du rectangle est : "
        << surface(largeur, hauteur) << endl;

    return 0;
}

double surface(double largeur, double hauteur)
{
    return largeur * hauteur;
}
```

Programmation procédurale : exemple

```
struct Rectangle {  
    double largeur;  
    double hauteur;  
};  
double surface(Rectangle const& rect);  
  
int main()  
{  
    Rectangle rectangle({ 3.0, 4.0 });  
  
    cout << "La surface du rectangle est : "  
        << surface(rectangle) << endl;  
  
    return 0;  
}  
double surface(Rectangle const& rect)  
{  
    return rect.largeur * rect.hauteur;  
}
```

Objets : quatre concepts de base

Un des objectifs principaux de la notion d'**objet** :

organiser des programmes complexes

grâce aux notions :

- ▶ d'encapsulation
- ▶ d'abstraction
- ▶ d'héritage
- ▶ et de polymorphisme

Notions d'encapsulation

Principe d'encapsulation :

regrouper dans le même objet informatique («concept»), les données et les traitements qui lui sont spécifiques :

- ▶ **attributs** : les données incluses dans un objet
- ▶ **méthodes** : les fonctions (= traitements) définies dans un objet
- ▶ Les objets sont définis par leurs attributs et leurs méthodes.

Notion d'abstraction

Pour être véritablement intéressant, un objet doit permettre un certain degré d'**abstraction**.

Le processus d'abstraction consiste à identifier pour un ensemble d'éléments :

- ▶ des caractéristiques communes à tous les éléments
- ▶ des mécanismes communs à tous les éléments
- ☞ description **générique** de l'ensemble considéré :
se focaliser sur l'essentiel, cacher les détails.

Notion d'abstraction : exemple

Exemple : Rectangles

- ▶ la notion d'« *objet rectangle* » n'est intéressante que si l'on peut lui associer des propriétés et/ou mécanismes généraux (valables pour l'ensemble des rectangles)
- ▶ Les notions de *largeur* et *hauteur* sont des propriétés générales des rectangles (**attributs**),
- ▶ Le mécanisme permettant de calculer la surface d'un rectangle (*surface = largeur × hauteur*) est commun à tous les rectangles (**méthodes**)

Abstraction et Encapsulation

En plus du regroupement des données et des traitements relatifs à une entité, l'encapsulation permet en effet de définir **deux niveaux** de perception des objets :

- ▶ niveau *externe* : partie « *visible* » (par les programmeurs-utilisateurs) :
 - ▶ l'**interface** : *entête* de quelques méthodes bien choisies
 - ☞ résultat du processus d'*abstraction*
- ▶ niveau *interne* : (détails d')**implémentation**
 - ▶ **corps** :
 - ▶ méthodes et attributs accessibles uniquement depuis l'intérieur de l'objet (ou d'objets similaires)
 - ▶ définition de toutes les méthodes de l'objet

Exemple d'interface

L'interface d'une voiture

- ▶ Volant, accélérateur, pédale de frein, etc.
- ▶ Tout ce qu'il faut savoir pour la conduire
(mais pas la réparer ! ni comprendre comment ça marche)
- ▶ L'interface ne change pas, même si l'on change de moteur...
...et même si on change de voiture (dans une certaine mesure) :
abstraction de la notion de voiture (en tant qu'« objet à conduire »)

Encapsulation et Interface

Il y a donc deux facettes à l'encapsulation :

1. regroupement de tout ce qui caractérise l'objet : données (attributs) **et** traitements (méthodes)
2. isolement et dissimulation des détails d'implémentation

Interface = ce que le programmeur-utilisateur (hors de l'objet) peut utiliser

- ☞ Concentration sur les attributs et les méthodes concernant l'objet (*abstraction*)

Les « 3 facettes » d'une classe

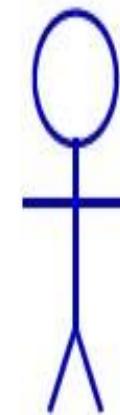
programmeur
utilisateur



utilisation du type

accord

programmeur
concepteur/développeur



définition d'un nouveau type (classe)



Pourquoi abstraire/encapsuler ?

1. L'intérêt de regrouper les traitements et les données conceptuellement reliées est de permettre une *meilleure visibilité* et une meilleure cohérence au programme, d'offrir une plus grande modularité.

```
double largeur(3.0);
double hauteur(4.0);

cout << "Surface : "
    << surface(largeur, hauteur))
    << endl;
```

```
Rectangle rect(3.0, 4.0);
cout << "Surface : "
    << rect.surface()
    << endl;
```

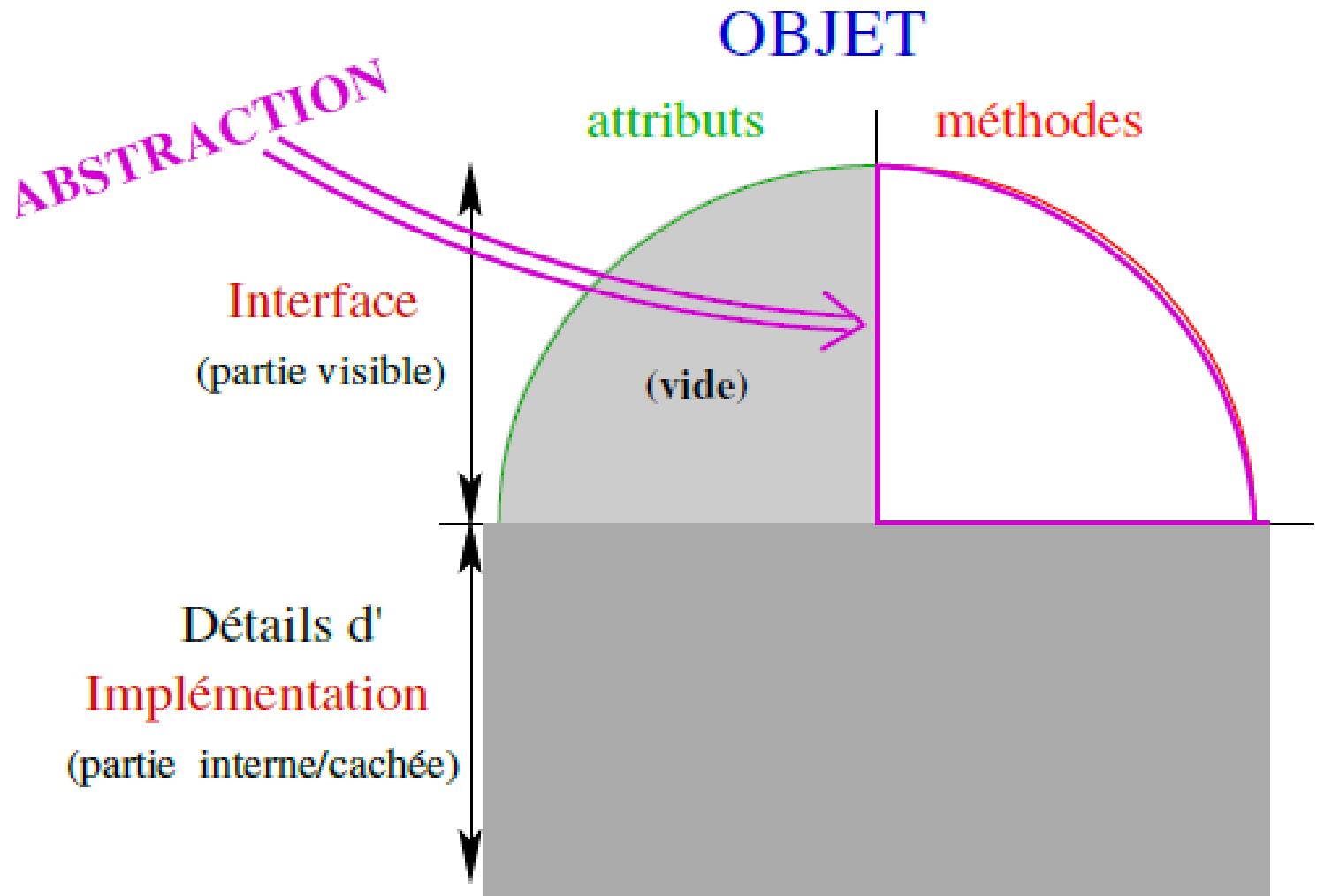
Pourquoi abstraire/encapsuler ? (2)

2. L'intérêt de séparer les niveaux *interne* et *externe* est de donner un **cadre** plus **rigoureux** à l'utilisation des objets utilisés dans un programme

Les objets ne peuvent être utilisés qu'au travers de leurs interfaces (niveau externe) et donc les éventuelles **modifications** de la structure interne restent **invisibles** à l'extérieur

Règle : les attributs d'un objet ne doivent pas être accessibles depuis l'extérieur, mais uniquement par des méthodes.

Encapsulation / Abstraction : Résumé

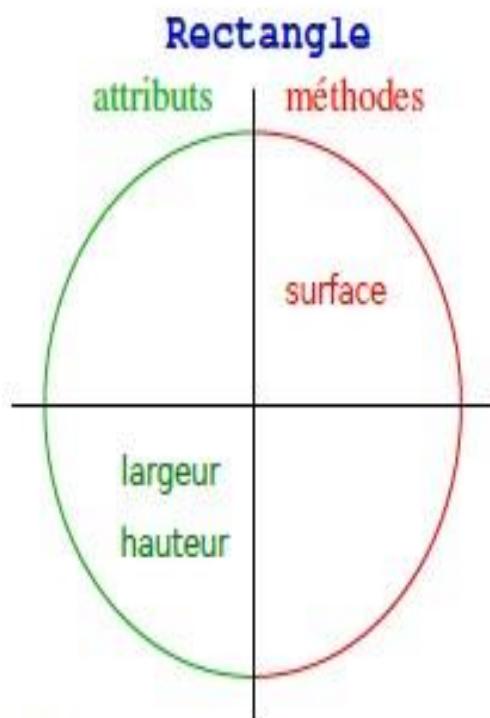


Classes et Instances, Types et Variables

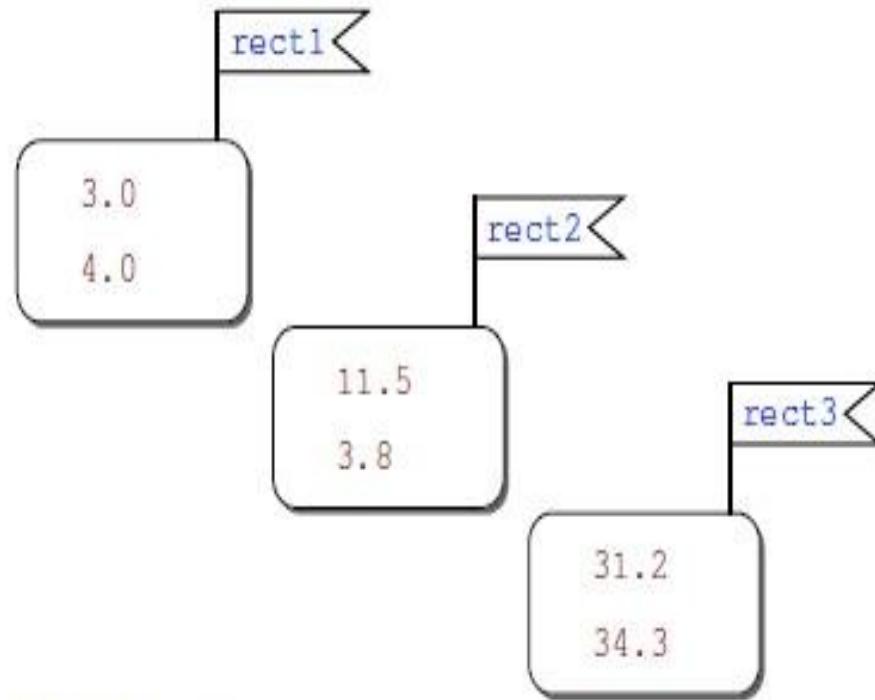
En programmation Objet :

- ▶ le résultat des processus d'encapsulation et d'abstraction s'appelle une **classe**
classe = catégorie d'objets
- ▶ une classe définit un **type**
- ▶ une réalisation particulière d'une classe s'appelle une **instance**
instance = **objet**
- ▶ un **objet** est une **variable**

Classes et Instances, Types et Variables (illustration)



classe
type (abstraction)
existence conceptuelle
(écriture du programme)

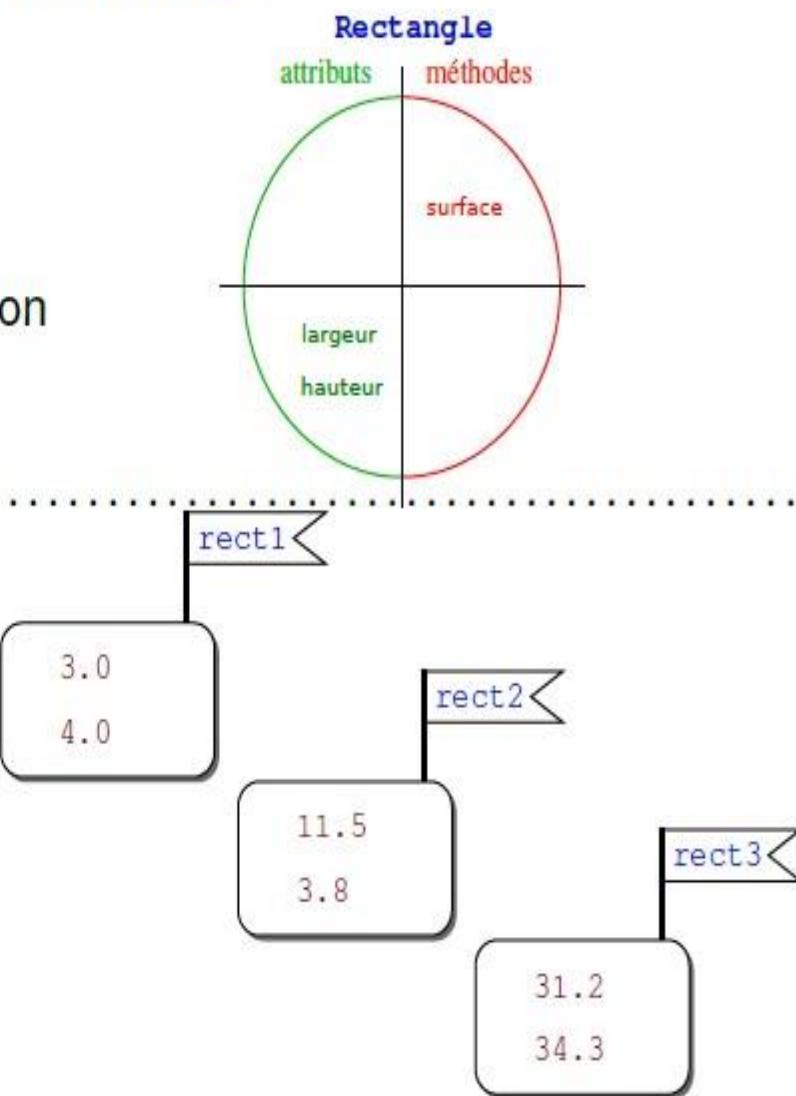


objets/instances
variables en mémoire
existence concrète
(exécution du programme)

Classes et Instances, Types et Variables

En programmation Objet :

- ▶ le résultat des processus d'encapsulation et d'abstraction s'appelle une **classe**
classe = catégorie d'objets
- ▶ une classe définit un **type**
- ▶ une réalisation particulière d'une classe s'appelle une **instance**
instance = **objet**
- ▶ un objet est une **variable**



Les classes en C++

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

Exemple : `class Rectangle { ... };`

Ceci définit un nouveau **type** du langage.

La déclaration d'une **instance** d'une classe se fait de façon similaire à la déclaration d'une **variable** :

`nom_classe nom_instance;`

Exemple :

`Rectangle rect1;`

déclare une instance `rect1` de la classe `Rectangle`.

Notre programme (1/4)

```
class Rectangle {  
};  
  
int main()  
{  
    Rectangle rect1;  
  
    return 0;  
}
```

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 :

la valeur de l'attribut `hauteur` d'une instance `rect1` de la classe `Rectangle` sera référencée par l'expression :

`rect1.hauteur`

Notre programme (2/4)

```
#include <iostream>
using namespace std;

class Rectangle {
    double hauteur;
    double largeur;
};

int main()
{
    Rectangle rect1;

    rect1.hauteur = 3.0;
    rect1.largeur = 4.0;

    cout << "hauteur : " << rect1.hauteur
        << endl;

    return 0;
}
```

Déclaration des méthodes

Les méthodes sont donc :

- ▶ des fonctions propres à la classe
- ▶ qui ont donc accès aux attributs de la classe
- ☞ Il ne faut donc **pas** passer les attributs comme arguments aux méthodes de la classe !

Exemple :

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

Portée des attributs

Les attributs d'une classe constituent des variables *directement accessibles* dans toutes les méthodes de la classe (*i.e.* des « variables *globales à la classe* »).

On parle de « *portée de classe* ».

Il n'est donc **pas nécessaire de les passer comme arguments des méthodes**.

Par exemple, dans toutes les méthodes de la classe `Rectangle`, l'identificateur `hauteur` (resp. `largeur`) fait *a priori* référence à la valeur de l'attribut `hauteur` (resp. `largeur`) de l'instance concernée (par l'appel de la méthode en question)

Déclaration des méthodes

Les méthodes sont donc :

- ▶ des fonctions propres à la classe
 - ▶ qui ont donc accès aux attributs de la classe
- ☞ Il ne faut donc **pas** passer les attributs comme arguments aux méthodes de la classe !

Exemple :

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

Paramètres des méthodes

Mais ce n'est pas parce qu'on n'a pas besoin de passer les attributs de la classe comme arguments aux méthodes de cette classe, que les méthodes n'ont *jamais* de paramètres.

Les méthodes **peuvent avoir des paramètres** : ceux qui sont nécessaires (et donc *extérieurs à l'instance*) pour exécuter la méthode en question !

Exemple :

```
class FigureColoree {  
    // ...  
    void colorie(Couleur c) { /* ... */ }  
    // ...  
};  
  
FigureColoree une_figure;  
Couleur rouge;  
// ...  
une_figure.colorie(rouge);  
// ...
```

Définition externe des méthodes

Il est possible d'écrire les définitions des méthodes à l'extérieur de la déclaration de la classe

☞ meilleure lisibilité du code, modularisation

Pour relier la définition d'une méthode à la classe pour laquelle elle est définie, il suffit d'utiliser l'opérateur `::` de résolution de portée :

- ▶ La déclaration de la classe contient les *prototypes des méthodes*
- ▶ les définitions correspondantes spécifiées à l'extérieur de la déclaration de la classe se font sous la forme :

```
typeRetour NomClasse::nomFonction(type1 param1  
, type2 param2  
, ...)  
{ ... }
```

Définition externe des méthodes : exemple

```
class Rectangle {  
    // ...  
    double surface(); // prototype  
};  
  
// définition  
double Rectangle::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* (« **prédictats** »).

On peut pour cela ajouter le mot `const` **après** la liste des paramètres de la méthode :

`type_retour nom_methode (type_param1 nom_param1, ...) const`

Exemple :

```
class Rectangle {  
    // ...  
    double surface() const;  
};  
  
double Rectangle::surface() const  
{  
    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 (« **prédicts** »).

On peut pour cela ajouter le mot **const** **après** la liste des paramètres de la méthode :

type_retour nom_methode (type_param1 nom_param1, ...) **const**

Si vous déclarez une action en tant que prédicat (**const**), vous aurez à la compilation le message d'erreur :

assignment of data-member ‘...’ in read-only structure

Appels 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 : la méthode

`void surface() const;`

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

`rect1.surface()`

Autres exemples :

`une_figure.colorie(rouge);`

`i < tableau.size()`

Notre programme (3/4)

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

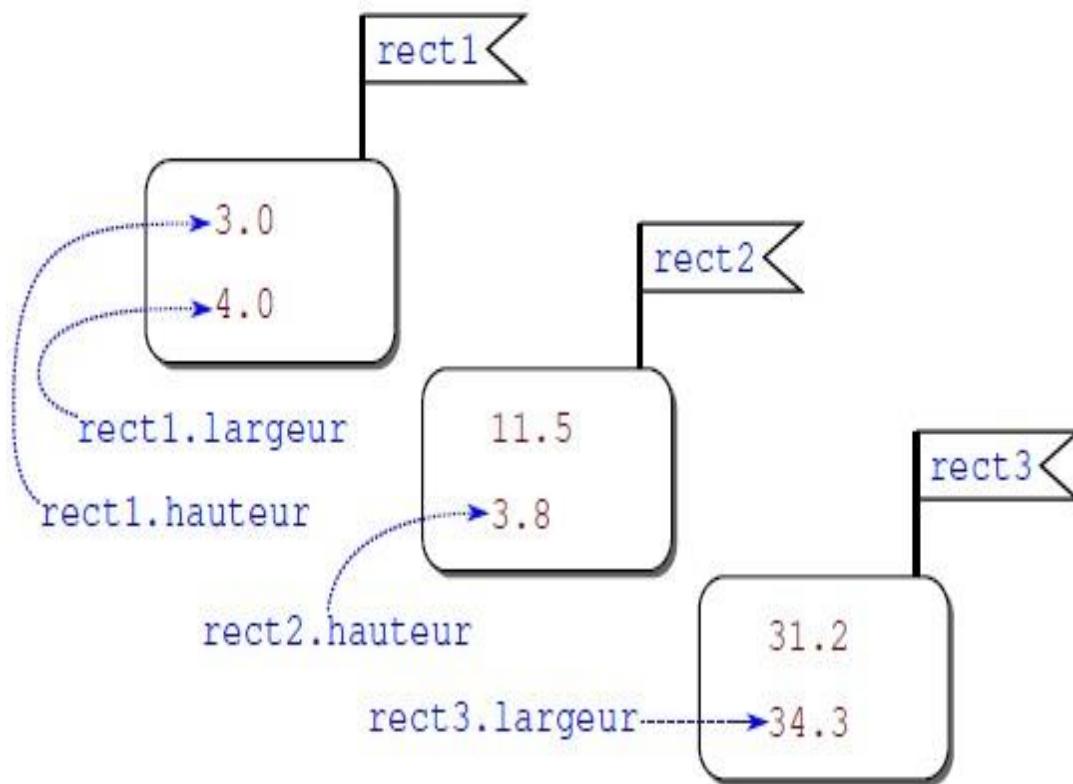
int main()
{
    Rectangle rect1;

    rect1.hauteur = 3.0;
    rect1.largeur = 4.0;

    cout << "surface : " << rect1.surface()
        << endl;
// ...
```

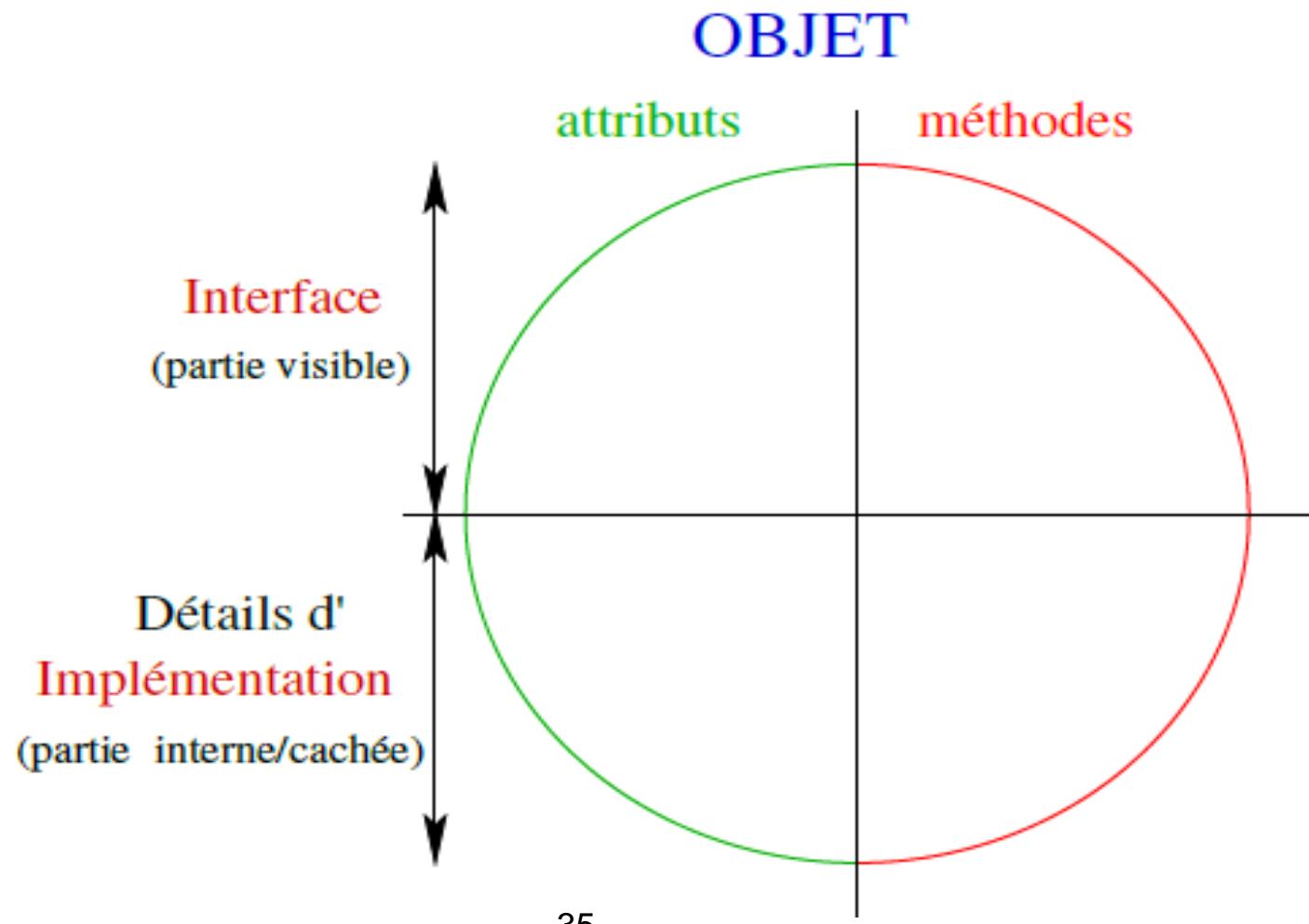
Résumé : Accès aux attributs et méthodes

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



`rect1.surface()` : méthode `surface` de la classe `Rectangle` s'appliquant à `rect1`

Encapsulation / Abstraction



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
```

Note : Si aucun droit d'accès n'est précisé, c'est `private` par défaut.

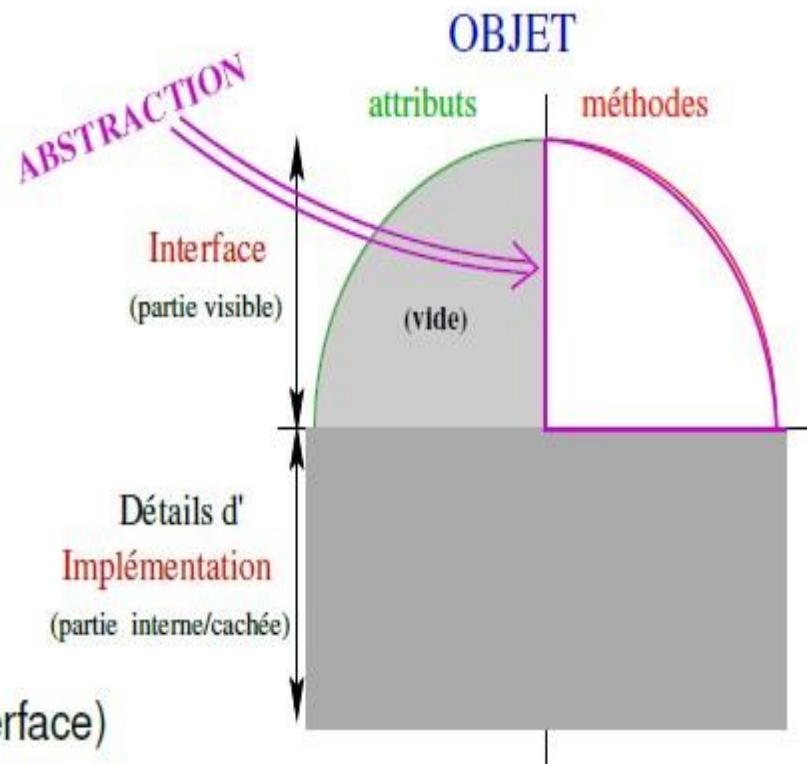
Encapsulation et interface (2)

À l'inverse, 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
- ▶ **Public :**
 - ▶ Quelques méthodes bien choisies (interface)



« Accesseurs » et « manipulateurs »

Tous les attributs sont privés ?

- ▶ Et si on a besoin de les utiliser depuis l'extérieur de la classe ? !

Par exemple, comment « manipuler » la largeur et la hauteur d'un rectangle ?

« Accesseurs » et « manipulateurs »

Si le programmeur *le juge utile*, il **inclus les méthodes publiques nécessaires ...**

1. Accesseurs (« méthodes get » ou « getters ») :

- ▶ Consultation (i.e. « prédictat »)
- ▶ Retour de la valeur d'une variable d'instance précise

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

2. Manipulateurs (« méthodes set » ou « setters ») :

- ▶ Modification (i.e. « action »)
- ▶ Affectation de l'argument à une variable d'instance précise

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

Notre programme (4/4)

```
#include <iostream>
using namespace std;

class Rectangle {
public:
    double surface() const
    { return hauteur * largeur; }

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

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

```
private:
    double hauteur;
    double largeur;
};

int main()
{
    Rectangle rect1;

    rect1.setHauteur(3.0);
    rect1.setLargeur(4.0);

    cout << "hauteur : "
        << rect1.getHauteur()
        << endl;

    return 0;
}
```

« Accesseurs », « manipulateurs » et encapsulation

Fastidieux d'écrire des « Accesseurs »/« manipulateurs » alors que l'on pourrait tout laisser en public ?

```
class Rectangle
{
public:
    double largeur;
    double hauteur;
    string label;
};
```

mais dans ce cas ...

```
Rectangle rect;
rect.hauteur = -36;
cout << rect.label.size() << endl;
```

Masquage (shadowing)

masquage = un identificateur « cache » un autre identificateur

```
int main() {
    int i(120);

    for (int i(1); i < MAX; ++i) {
        cout << i << endl;
    }

    cout << i << endl;
    return 0;
}
```

Situation typique en POO : un paramètre cache un attribut

```
void setHauteur(double hauteur) {
    hauteur = hauteur; // Hmm.... pas terrible !
}
```

Masquage et pointeur this

Si, dans une méthode, un attribut est **masqué** alors la valeur de l'attribut peut quand même être référencée à l'aide du mot réservé **this**.

this est un **pointeur sur l'instance courante**

this \simeq « mon adresse »

Syntaxe pour spécifier un attribut *en cas d'ambiguïté* :

this->nom_attribut

Exemple :

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

L'utilisation de **this** est obligatoire dans les situations de **masquage** (mais évitez ces situations !)

Portée des attributs (résumé)

La portée des attributs dans la définition des méthodes est résumée par le schéma suivant :

```
class MaClasse {  
private:  
    int x;  
    int y;  
public:  
    void une_methode( int x ) {  
        ... y ...  
        ... x ...  
        ... this->x ...  
    }  
};
```

Classes = super struct

Pour résumer à ce stade, une classe est une `struct`

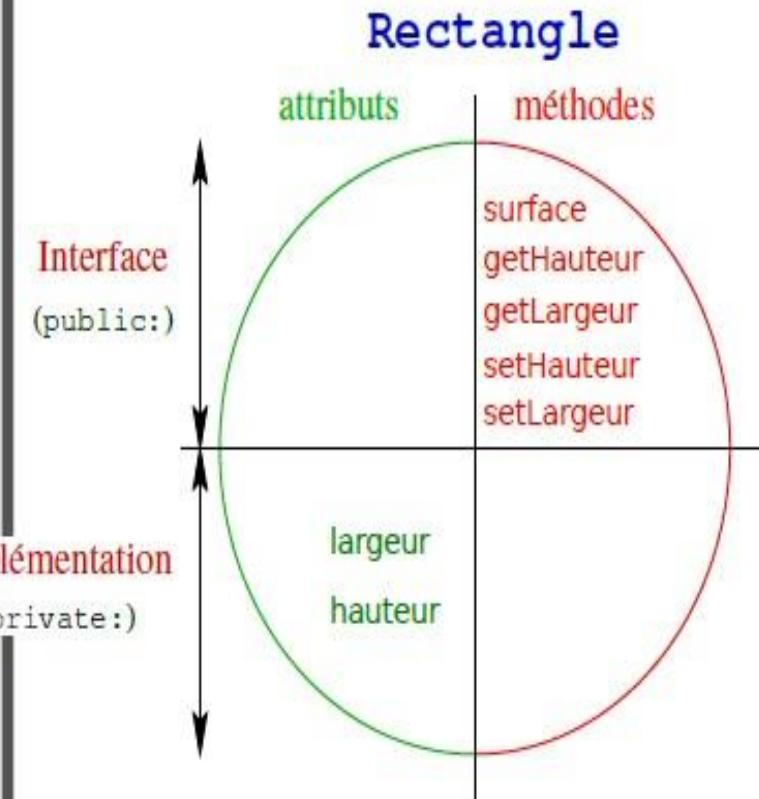
- ▶ qui contient aussi des fonctions (« méthodes »)
- ▶ dont certains champs (internes) peuvent être cachés (`private:`)
- ▶ et dont d'autres constituent l'interface (`public:`)

Un exemple complet de classe (1/2)

```
#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 h) { hauteur = h; }
    void setLargeur(double l) { largeur = l; }

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



Un exemple complet de classe (2/2)

```
//utilisation de la classe

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

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

    return 0;
}
```

Exemple avec définitions externes à la classe

```
class Rectangle
{
public:
    // prototypes des méthodes
    double surface() const;

    // accesseurs
    double hauteur() const;
    double largeur() const;

    // manipulateurs
    void hauteur(double);
    void largeur(double);

private:
    // déclaration des attributs
    double hauteur_;
    double largeur_;
};
```

```
double Rectangle::surface() const
{
    return hauteur_ * largeur_;
}

double Rectangle::hauteur() const
{
    return hauteur_;
}

... // idem pour largeur

void Rectangle::hauteur(double h)
{
    hauteur_ = h;
}

... // idem pour largeur
```

Où en est-on ?

On peut par exemple déclarer une instance de la classe `Rectangle` :

```
Rectangle rect;
```

Une fois que l'on a fait cette déclaration, comment faire pour *initialiser les attributs* de `rect` ?

```
class Rectangle {  
public:  
    double surface() const;  
    double getHauteur() const;  
    double getLargeur() const;  
    void setHauteur(double h);  
    void setLargeur(double l);  
private:  
    double hauteur;  
    double largeur;  
};
```

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 :

- ▶ elle implique que tous les attributs fassent partie de l'interface (`public`) ou soient assortis d'un manipulateur
 - ▣ casse l'encapsulation
- ▶ oblige le programmeur-utilisateur de la classe à initialiser explicitement tous les attributs
 - ▣ risque d'oubli

Initialisation des attributs (2)

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

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

Pour faire ces initialisations, il existe en C++ des méthodes particulières appelées **constructeurs**.

Les constructeurs

Un constructeur est une méthode :

- ▶ invoquée *automatiquement* lors de la déclaration d'un objet
- ▶ chargée d'effectuer toutes les opérations requises en « début de vie » de l'objet (*dont l'initialisation des attributs*)

Syntaxe de base :

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

Exemple (à améliorer par la suite) :

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

Les constructeurs (2)

Les constructeurs sont des méthodes presque comme les autres. Les différences sont :

- ▶ *pas de type de retour*
(pas même `void`)
- ▶ *même nom que la classe*
- ▶ *invoqués automatiquement*
à chaque fois qu'une instance est créée.

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

Comme les autres méthodes :

- ▶ les constructeurs peuvent être surchargés
- ▶ on peut donner des valeurs par défaut à leurs paramètres
(exemples dans la suite)

Une classe peut donc avoir **plusieurs constructeurs**, pour peu que leur liste de paramètres soit différente.

Notre programme (1/3)

```
class Rectangle {
public:
    Rectangle(double h, double L)
    {
        hauteur = h;
        largeur = L;
    }
    double surface() const
    { return hauteur * largeur; }
    // accesseurs/modificateurs si nécessaire
    // ...
private:
    double hauteur;
    double largeur;
};
```

Initialisation par constructeur

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

Syntaxe :

NomClasse instance(valarg₁, ..., valarg_N);

où *valarg₁*, ..., *valarg_N* sont les valeurs des arguments passés au constructeur.

Exemple :

```
Rectangle r1(18.0, 5.3); // invocation du constructeur à 2 paramètres
```

Notre programme (2/3)

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

int main()
{
    Rectangle rect1(3.0, 4.0);
// ...
}
```

Construction des attributs

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

Exemple :
(à améliorer dans
une prochaine leçon)

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

mauvaise solution :

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

- Il faut initialiser *directement* les attributs en faisant appel à *leurs propres constructeurs* !

Appel aux constructeurs des attributs

Un constructeur devrait normalement contenir une section d'appel aux constructeurs des attributs....

...ainsi que l'initialisation des attributs de type de base.

C'est ce qu'on appelle la « **liste d'initialisation** » du constructeur.

Syntaxe générale :

```
NomClasse(liste_paramètres)
// liste d'initialisation
: attribut1(...), // appel au constructeur de attribut1
  ...
  attributN(...) // appel au constructeur de attributN
{ // autres opérations }
```

Appel aux constructeurs des attributs

Exemple :

```
class Rectangle {  
    Rectangle(double h, double L);  
    // ...  
};  
  
class RectangleColore {  
  
    RectangleColore(double h, double L, Couleur c)  
        : rectangle(h, L), couleur(c)  
    {}  
  
private:  
    Rectangle rectangle;  
    Couleur couleur;  
};
```

Liste d'initialisation (2)

Cette section introduite par « : » est optionnelle mais **recommandée**.

Par ailleurs :

- ▶ les attributs non-initialisés dans cette section
 - ▶ prennent une valeur par défaut si ce sont des objets ;
 - ▶ restent indéfinis s'ils sont de type de base ;
- ▶ les attributs initialisés dans cette section peuvent (bien sûr) être changés dans le corps du constructeur.

Exemple :

```
Rectangle(double h, double L)
    : hauteur(h) //initialisation
{
    // largeur a une valeur indéfinie jusqu'ici
    largeur = 2.0 * L + h; // par exemple...
    // la valeur de largeur est définie à partir d'ici
}
```

Notre programme (3/3)

```
// ...
class Rectangle {
public:
    Rectangle(double h, double L)
        : hauteur(h), largeur(L)
    {}
    double surface() const
        { return hauteur * largeur; }
    // accesseurs/modificateurs
    // ...
private:
    double hauteur;
    double largeur;
};

int main()
{
    Rectangle rect1(3.0, 4.0);
    // ...
}
```

Constructeur par défaut

Le constructeur par défaut est un constructeur qui **n'a pas de paramètre** ou dont **tous** les paramètres ont des **valeurs par défaut**.

Exemple :

```
// 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)
{}
```

Constructeur par défaut : autre exemple

Autre façon de faire : regrouper les 2 premiers constructeurs en utilisant les valeurs par défaut des paramètres :

```
// DEUX constructeurs dont le constructeur par défaut
Rectangle(double c = 1.0) : hauteur(c), largeur(2.0*c)
{}

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

Constructeur par défaut par défaut

Si aucun constructeur n'est spécifié, le compilateur génère automatiquement une **version minimale du constructeur par défaut**

qui :

- ▶ appelle le constructeur par défaut des attributs objets.
- ▶ laisse non initialisés les attributs de type de base.

Dès qu'**au moins un constructeur a été spécifié**, ce constructeur par défaut par défaut *n'est plus fourni*.

Si donc on spécifie *un* constructeur sans spécifier de constructeur par défaut, on ne peut plus construire d'objet de cette classe sans les initialiser (ce qui est voulu !) puisqu'il n'y a plus de constructeur par défaut.

C++11 ...mais on peut le rajouter si on veut (voir plus loin).

A:

```
class Rectangle {  
private:  
    double h; double L;  
    // suite ...  
};
```

B:

```
class Rectangle {  
private:  
    double h; double L;  
public:  
    Rectangle()  
        : h(0.0), L(0.0)  
    {}  
    // suite ...  
};
```

C:

```
class Rectangle {  
private:  
    double h; double L;  
public:  
    Rectangle(double h=0.0,  
              double L=0.0)  
        : h(h), L(L)  
    {}  
    // suite ...  
};
```

D:

```
class Rectangle {  
private:  
    double h; double L;  
public:  
    Rectangle(double h,  
              double L)  
        : h(h), L(L)  
    {}  
    // suite ...  
};
```

Constructeur par défaut : exemples

	constructeur par défaut	Rectangle r1;	Rectangle r2(1.0, 2.0);
(A)	constructeur par défaut par défaut	?	Illicite !

```
class Rectangle {  
};
```

Constructeur par défaut : exemples

	constructeur par défaut	Rectangle r1;	Rectangle r2(1.0, 2.0);		
(A)	constructeur par défaut par défaut	<table border="1"><tr><td>?</td><td>?</td></tr></table>	?	?	Illicite !
?	?				
(B)	constructeur par défaut explicitement déclaré	<table border="1"><tr><td>0</td><td>0</td></tr></table>	0	0	Illicite !
0	0				

```
class Rectangle {  
    Rectangle()  
        : h(0.0), L(0.0)  
    {}  
};
```

Constructeur par défaut : exemples

	constructeur par défaut	Rectangle r1;	Rectangle r2(1.0, 2.0);				
(A)	constructeur par défaut par défaut	<table border="1"><tr><td>?</td><td>?</td></tr></table>	?	?	Illicite !		
?	?						
(B)	constructeur par défaut explicitement déclaré	<table border="1"><tr><td>0</td><td>0</td></tr></table>	0	0	Illicite !		
0	0						
(C)	un des trois constructeurs est par défaut	<table border="1"><tr><td>0</td><td>0</td></tr></table>	0	0	<table border="1"><tr><td>1</td><td>2</td></tr></table>	1	2
0	0						
1	2						

```
class Rectangle {  
    Rectangle(double h=0.0,  
             double L=0.0)  
        : h(h), L(L)  
    {}  
};
```

Constructeur par défaut : exemples

	constructeur par défaut	Rectangle r1; Rectangle r2(1.0, 2.0);	
(A)	constructeur par défaut par défaut	?	Illicite !
(B)	constructeur par défaut explicitement déclaré	0	Illicite !
(C)	un des trois constructeurs est par défaut	0	1 2
(D)	pas de constructeur par défaut	Illicite !	1 2

```
class Rectangle {  
    Rectangle(double h,  
              double L)  
        : h(h), L(L)  
    {}  
};
```

C++11 Remettre le constructeur par défaut par défaut

Dès qu'au moins un constructeur a été spécifié, ce constructeur par défaut par défaut n'est plus fourni.

C'est très bien si c'est vraiment ce que l'on veut
(c'est-à-dire forcer les utilisateurs de la classe à utiliser nos constructeurs).

Mais si l'on veut quand même avoir le constructeur par défaut par défaut, on peut le re-demande en écrivant dans la définition de la classe :

NomClasse() = default;

Exemple (modification du cas (D) précédent) :

```
class Rectangle {  
public:  
    Rectangle() = default; // mais peu pertinent ici  
    Rectangle(double h, double L) : h(h), L(L) {}  
    // suite ...  
};
```

C++11 Appel aux autres constructeurs

C++11 autorise les constructeurs d'une classe à appeler n'importe quel autre constructeur de cette même classe

Exemple :

```
class Rectangle {  
private:  
    double hauteur; double largeur;  
public:  
    Rectangle(double h, double L) : hauteur(h), largeur(L) {}  
  
    Rectangle() : Rectangle(0.0, 0.0) {}  
    // bien mieux que le =default précédent  
  
    // suite ...  
};
```

C++11 Initialisation par défaut des attributs

C++11 permet de donner directement une valeur par défaut aux attributs.

Si le constructeur appelé ne modifie pas la valeur de cet attribut, ce dernier aura alors la valeur indiquée.

Exemple :

```
class Rectangle {  
    // ...  
private:  
    double hauteur = 0.0;  
    double largeur = 0.0;  
    // ...  
};
```

Conseil : préférez l'utilisation des constructeurs.

Constructeur de copie

C++ offre un moyen de créer la **copie** d'une instance :
le *constructeur de copie*

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

r1 et r2 sont deux *instances distinctes*
mais ayant des mêmes valeurs pour leurs attributs
(au moins juste après la copie).

Autre exemple de copie (invocation du constructeur de copie) :

```
double f(Rectangle r);  
...  
x = f(r1);
```

Constructeur de copie (2)

Le constructeur de copie permet d'initialiser une instance en *copiant* les attributs d'une *autre instance* du même type.

Syntaxe :

NomClasse(NomClasse const& autre) { ... }

Exemple :

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

Constructeur de copie (3)

- ▶ Un constructeur de copie est *automatiquement généré* par le compilateur s'il n'est pas explicitement défini
(constructeur de copie par défaut)
- ▶ Ce constructeur opère une initialisation *membre à membre* des attributs (si l'attribut est un objet le constructeur de cet objet est invoqué)

Tout se passe comme si le constructeur précédent avait été écrit :

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

mais il n'est *pas nécessaire* de l'écrire !

Constructeur de copie (3)

- ▶ Un constructeur de copie est *automatiquement généré* par le compilateur s'il n'est pas explicitement défini
(constructeur de copie par défaut)
- ▶ Ce constructeur opère une initialisation *membre à membre* des attributs (si l'attribut est un objet le constructeur de cet objet est invoqué)
- ▶ Cette copie suffit dans la plupart des cas.

Cependant, il est parfois nécessaire de redéfinir le constructeur de copie, en particulier lorsque certains attributs sont des *pointeurs* (des exemples arriveront plus tard dans le cours) !

C++11 Suppression du constructeur de copie

C++11 Par ailleurs, si l'on souhaite *interdire* la copie, il suffit de **supprimer** le constructeur de copie par défaut avec la commande « `= delete` »

Exemple :

```
class PasCopiable {  
    /* ... */  
    PasCopiable(PasCopiable const&) = delete;  
};
```

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, l'*invocation explicite* de méthodes de libération n'est pas satisfaisante (fastidieuse, source d'erreur, affaiblissement de l'encapsulation).

☞ C++ offre une méthode appelée *destructeur* invoquée automatiquement en fin de vie de l'instance.

Destructeur (2)

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 paramètre*
☒ **pas de surcharge possible**
- ▶ Son nom est celui de la classe, précédé du signe `~` (tilde).
- ▶ Si le destructeur n'est pas défini explicitement par le programmeur, le compilateur en génère automatiquement une version minimale.

Exemple

Supposons que l'on souhaite compter le nombre d'instances d'une classe actives à un moment donné dans un programme.

```
int main()
{
    // compteur = 0
    Rectangle r1;
    // compteur = 1
    {
        Rectangle r2;
        // compteur = 2
        // ...
    }
    // compteur = 1
    return 0;
} // compteur = 0
```

Exemple

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

```
long compteur(0); /* Hmm.... On y reviendra  
                     dans une autre séquence vidéo ! */  
  
class Rectangle {  
    //...  
    Rectangle(): hauteur(0.0), largeur(0.0) { //constructeur  
        ++compteur; }  
    // ...
```

Attributs de classe

Revenons à notre exemple du *comptage des instances* :

```
int compteur(0); // Hmm....  
  
class Rectangle {  
    // constructeur par défaut  
    Rectangle(): hauteur(0.0), largeur(0.0)  
    { ++compteur; }  
  
    // destructeur  
    ~Rectangle() { --compteur; }  
    //...
```

Oh horreur !.. Nous avons utilisé une *variable globale* !

C'est une **très mauvaise solution !** (contraire au principe d'encapsulation, effets de bord, mauvaise modularisation)

Attributs de classe (2)

La solution à ce problème consiste à utiliser un **attribut de classe** :

```
class Rectangle {  
private:  
    double hauteur, largeur;  
    static int compteur;  
    //...  
};
```

- ▶ La déclaration d'un attribut de classe est précédée du mot clé **static**
- ▶ Un attribut de classe est **partagé par toutes les instances** de la même classe (on parle aussi d'« *attribut statique* »)
- ▶ Il existe même lorsqu'aucune instance de la classe n'est déclarée
- ▶ Un attribut de la classe peut être **privé** ou **public**

Initialisation des attributs de classe

Un attribut de classe doit être initialisé explicitement à l'extérieur de la classe

```
/* Initialisation de l'attribut de classe dans le fichier de
   définition de la classe, mais HORS de la classe.
*/
int Rectangle::compteur(0);
/* Rectangle::compteur existe même si l'on n'a déclaré
   aucune instance de la classe Rectangle */
```

Les attributs de classe sont très pratiques lorsque **differents** objets d'une classe doivent accéder à une **même** information.

Ils permettent notamment d'**éviter** que cette information soit **dupliquée** au niveau de chaque objet.

- Concrètement : réserver cet usage à des **constantes** utiles pour toutes les instances de la classe.

Méthodes de classe

Similairement, si on ajoute `static` à une méthode :

- ▶ on peut accéder aussi à la méthode *sans* objet, à partir du nom de la classe et de l'opérateur de résolution de portée « `::` »

```
class A {  
public:  
    static void methode1() { cout << "Méthode 1" << endl; }  
    void methode2() { cout << "Méthode 2" << endl; }  
};  
  
int main () {  
    A::methode1(); // OK  
    A::methode2(); // ERREUR  
    A x;  
    x.methode1(); // OK  
    x.methode2(); // OK  
}
```

Restrictions sur les méthodes de classe

Puisqu'une méthode de classe peut être appelée sans objet :

- ▶ elles n'ont pas le droit d'utiliser de méthode ni d'attribut d'instance (y compris `this`)
 - ▶ elles ne peuvent accéder *seulement* **qu'à** d'autres méthodes ou attributs *de classe*
- ☞ Ce sont simplement des *fonctions usuelles* mises dans une classe.

Le recours à des méthodes de classe ne se justifie que dans des situations **très** particulières :

- ▶ affichage spécifique d'attributs de classe
 - ▶ manipulation d'attributs de classe non constants et `private`
- ☞ Préférez toujours les fonctions usuelles et évitez absolument la prolifération de `static` !

Surcharge des opérateurs

Intérêt ?

Exemple avec les nombres complexes :

```
class Complexe { ... };  
Complexe z1, z2, z3, z4;
```

Il est quand même plus naturel d'écrire :

`z4 = z1 + z2 + z3;`

que `z3 = addition(addition(z1, z2), z3);`

De même, on préfèrera unifier l'affichage :

`cout << "z3 = " << z3 << endl;`

plutôt que d'écrire :

```
cout << "z3 = " ;  
affiche(z3) ;  
cout << endl ;
```

En pratique, quelle surcharge des opérateurs ?

Dans votre pratique du C++, vous pouvez, en fonction de votre niveau :

- ① ne pas faire de surcharge des opérateurs ;
- ② surcharger simplement les opérateurs arithmétiques de base (+, -, ...) sans leur version « auto-affectation » (+=, -=, ...); surcharger l'opérateur d'affichage (<<);
- ③ surcharger les opérateurs en utilisant leur version « auto-affectation », mais sans valeur de retour pour celles-ci :

`void operator+=(Classe const&);`

- ④ faire la surcharge avec valeur de retour des opérateurs d'« auto-affectation » :

`Classe& operator+=(Classe const&);`

Opérateur ?

Rappel : un opérateur est une opération sur un ou entre deux opérande(s) (variable(s)/expression(s)) :

opérateurs arithmétiques (+, -, *, /, ...), opérateurs logiques (and, or, not),
opérateurs de comparaison (==, >=, <=, ...), opérateur d'incrément (++), ...

Un appel à un opérateur est un appel à une fonction ou une méthode spécifique :

a *Op* b → operator*Op*(a, b) ou a.operator*Op*(b)

Op a → operator*Op*(a) ou a.operator*Op*()

Exemples d'appels d'opérateurs

$a + b$	correspond à	<code>operator+(a, b)</code>	ou	<code>a.operator+(b)</code>
$b + a$		<code>operator+(b, a)</code>		<code>b.operator+(a)</code>
$-a$		<code>operator-(a)</code>		<code>a.operator-()</code>
<code>cout << a</code>		<code>operator<<(cout, a)</code>		<code>cout.operator<<(a)</code>
$a = b$		—		<code>a.operator=(b)</code>
$a += b$		<code>operator+=(a, b)</code>		<code>a.operator+=(b)</code>
$++a$		<code>operator++(a)</code>		<code>a.operator++()</code>
$\text{not } a$		<code>operator not(a)</code>		<code>a.operator not()</code>
	ou	<code>operator!(a)</code>		<code>a.operator!()</code>

Surcharge ?

Rappel : surcharge de fonction

- deux fonctions ayant le même nom mais pas les mêmes paramètres

Exemple :

```
int max(int, int);
double max(double, double);
```

De la même façon, on va pouvoir écrire plusieurs fonctions pour les opérateurs ; par exemple :

```
Complexe operator+(Complexe, Complexe);
Matrice operator+(Matrice, Matrice);
```

Surcharge interne et surcharge externe

Presque tous les opérateurs sont surchargeables
(sauf, parmi ceux que vous connaissez, :: et .)

La surcharge des opérateurs peut être réalisée

- ▶ soit à l'*extérieur*,
- ▶ soit à l'*intérieur* de la classe à laquelle ils s'appliquent.

```
Complexe operator+(Complexe, Complexe);
```

```
class Complexe {  
public:  
    Complexe operator+(Complexe) const;  
};
```

- ☞ Les opérateurs externes sont des *fonctions* ; les opérateurs internes sont des *méthodes*.

Exemple de surcharge externe

```
Complexe z1;  
Complexe z2;  
Complexe z3;  
// ...  
z3 = z1 + z2;
```

```
class Complexe {  
public:  
    Complexe(double abscisse, double ordonnee)  
        : x(abscisse), y(ordonnee) {}  
    // ...  
    double get_x() const;  
    double get_y() const;  
    // ...  
private:  
    double x;  
    double y;  
};  
  
const Complexe operator+(Complexe z1, Complexe const& z2)  
{  
    Complexe z3( z1.get_x() + z2.get_x(),  
                z1.get_y() + z2.get_y() );  
    return z3;  
}
```

Nécessité de la surcharge externe

La surcharge **externe** est nécessaire pour des opérateurs concernés par une classe, mais pour lesquels la classe en question **n'est pas** l'opérande de gauche.

Exemples :

1. multiplication d'un nombre complexe par un double :

```
double x;  
Complexe z1, z2;  
// ...  
z2 = x * z1;
```



ou `z2 = x.operator*(z1);`
 ou `z2 = operator*(x, z1);`

Le premier **n'a pas de sens** (`x` n'est pas un objet, mais de type élémentaire `double`)

2. écriture sur cout : `cout << z1;`

ou `cout.operator<<(z1);` ou `operator<<(cout, z1);`,

mais on souhaite le surcharger pour la classe `Complexe` et non pas dans la classe de `cout` (`ostream`).

Exemple de la multiplication externe

```
double x;  
Complexe z1, z2;  
// ...  
z2 = x * z1;
```

```
const Complexe operator*(double a, Complexe const& z)  
{  
    /* Soit l'écrire explicitement,  
       soit, quand c'est possible, utiliser l'opérateur interne :  
    */  
    return z * a;  
}
```

Surcharge interne des opérateurs

Pour surcharger un opérateur *Op* dans une classe *NomClasse*, il faut **ajouter la méthode** `operatorOp` dans la classe en question :

```
class NomClasse {  
    ...  
    // prototype de l'opérateur Op  
    type_retour operatorOp(type_parametre);  
    ...  
};  
  
// définition de l'opérateur Op  
type_retour NomClasse::operatorOp(type_parametre)  
{  
    ...  
}
```

Rappel : les méthodes ne doivent pas recevoir l'instance courante en paramètre

Surcharge interne des opérateurs : exemple

```
Complexe z1, z2;  
// ...  
z1 += z2;
```

```
class Complexe {  
public:  
    // ...  
    void operator+=(Complexe const& z2); // z1 += z2;  
    // ...  
};  
  
void Complexe::operator+=(Complexe const& z2) {  
    x += z2.x;  
    y += z2.y;  
}
```

Retour sur l'addition externe

```
class Complexe {  
public:  
    // ...  
    void operator+=(Complexe const& z2); // z1 += z2;  
    // ...  
};  
  
const Complexe operator+(Complexe z1, Complexe const& z2) {  
    z1 += z2; // utilise l'opérateur += redéfini précédemment  
    return z1;  
}
```

Surcharge interne ou surcharge externe ?

Les opérateurs propres à une classe peuvent être surchargés en interne ou en externe :

```
z3 = z1 + z2; // appel équivalent : SOIT z3 = z1.operator+(z2);  
           // SOIT z3 = operator+(z1, z2);  
  
class Complexe {  
public:  
    const Complexe operator+(Complexe const& z2) const;  
    // ....  
};
```

Surcharge interne ou surcharge externe ?

Les opérateurs propres à une classe peuvent être surchargés en interne ou en externe :

```
z3 = z1 + z2; // appel équivalent : SOIT z3 = z1.operator+(z2);  
//                                     SOIT z3 = operator+(z1, z2);  
  
const Complexe operator+(Complexe z1, Complexe const& z2);
```

Surcharge interne ou surcharge externe ?

Les opérateurs propres à une classe peuvent être surchargés en interne ou en externe :

```
z3 = z1 + z2; // appel équivalent : SOIT z3 = z1.operator+(z2);  
           // SOIT z3 = operator+(z1, z2);  
  
const Complexe operator+(Complexe z1, Complexe const& z2);  
  
// ...  
  
class Complexe {  
    friend const Complexe operator+(Complexe z1, Complexe const& z2);  
    // ...  
};
```

Surcharge interne ou surcharge externe ?

- ▶ préférez la *surcharge externe* chaque fois que vous pouvez le faire **SANS friend**
c.-à-d. chaque fois que vous pouvez écrire l'opérateur à l'aide de l'interface de la classe
(et sans copies inutiles)
- ▶ si l'opérateur est « proche de la classe », c.-à-d. nécessite des accès internes ou des copies supplémentaires inutiles (typiquement `operator+=`), utilisez la *surcharge interne*

En pratique, quelle surcharge des opérateurs ?

Dans votre pratique du C++, vous pouvez, en fonction de votre niveau :

- ① ne pas faire de surcharge des opérateurs ;
- ② surcharger simplement les opérateurs arithmétiques de base (+, -, ...) sans leur version « auto-affectation » (+=, -=, ...);
surcharger l'opérateur d'affichage (<<);
- ③ surcharger les opérateurs en utilisant leur version « auto-affectation », mais sans valeur de retour pour celles-ci :

`void operator+=(Classe const&);`

- ④ faire la surcharge avec valeur de retour des opérateurs d'« auto-affectation » :

`Classe& operator+=(Classe const&);`

Exemples de surcharges de quelques opérateurs usuels

(au niveau ④ du tr. précédent)

```
bool operator==(Classe const&) const; // ex: p == q  
bool operator<(Classe const&) const; // ex: p < q
```

```
Classe& operator+=(Classe const&); // ex: p += q  
Classe& operator-=(Classe const&); // ex: p -= q
```

```
Classe& operator*=(autre_type const); // ex: p *= x;
```

```
Classe& operator++(); // ex: ++p  
Classe& operator++(int inutile); // ex: p++
```

```
const Classe operator-() const; // ex: r = -p;
```

// ===== surcharges externes -----

```
const Classe operator+(Classe, Classe const&); // r = p + q  
const Classe operator-(Classe, Classe const&); // r = p - q
```

```
ostream& operator<<(ostream&, Classe const&); // ex: cout << p;  
const Classe operator*(autre_type, Classe const&); // ex: q = x * p;
```

Pourquoi const en type de retour ?

```
const Complexe operator+(Complexe, Complexe const&);
```

```
z3 = z1 + z2;
```

```
++(z1 + z2);
```

```
z1 + z2 = f(x);
```

Pourquoi operator<< retourne-t-il un ostream& ?

```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

```
cout << z1 << endl;
operator<<(cout << z1, endl);
operator<<(operator<<(cout, z1), endl);
```

Quel type de retour pour operator+= ?

```
z1 += z2;  
  
void Complexe::operator+=(Complexe const&);
```

En C++, chaque expression **fait** quelque chose et **vaut** quelque chose :

x = Expression; *z3 = (z1 += z2);*

```
class Complexe {  
    // ...  
    Complexe& operator+=(Complexe const& z2);  
    // ...  
};  
  
Complexe& Complexe::operator+=(Complexe const& z2)  
{  
    x += z2.x;  
    y += z2.y;  
    return *this;  
}
```

Opérateur d'affectation

L'opérateur d'affectation = (utilisé par exemple dans `a = b`) :

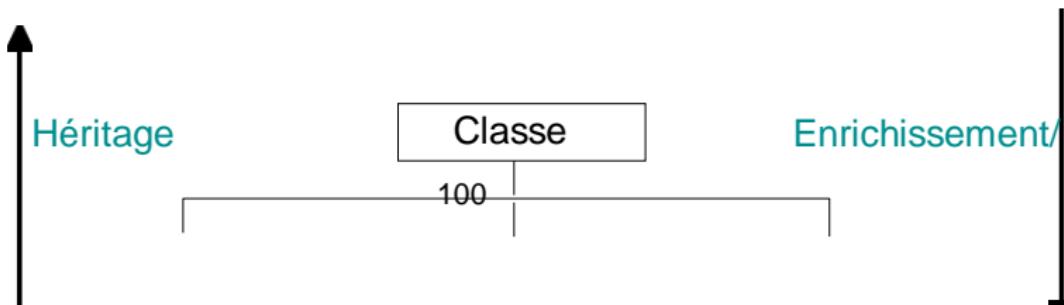
- ▶ est le seul opérateur universel
(il est fourni de toutes façons par défaut pour toute classe)
- ▶ est très lié au constructeur de copie,
sauf que le premier s'appelle lors d'une `affectation` et le second lors d'une `initialisation`
- ▶ la version par défaut, qui fait une copie de surface, est suffisante dans la très grande majorité des cas
- ▶ si nécessaire, on peut supprimer l'opérateur d'affectation :

```
class EnormeClasse {  
    // ...  
private:  
    EnormeClasse& operator=(EnormeClasse const&) = delete;  
};
```

Héritage

- Après les notions d'**encapsulation** et d'**abstraction**, le troisième aspect essentiel des objets est la notion d'**héritage**
- L'héritage est une technique extrêmement efficace pour créer des classes **plus spécialisées**, appelées **sous-classes**, à partir de classes plus générales déjà existantes, appelées **super-classes**.

Elle représente la relation «**est-un**»



Spécial

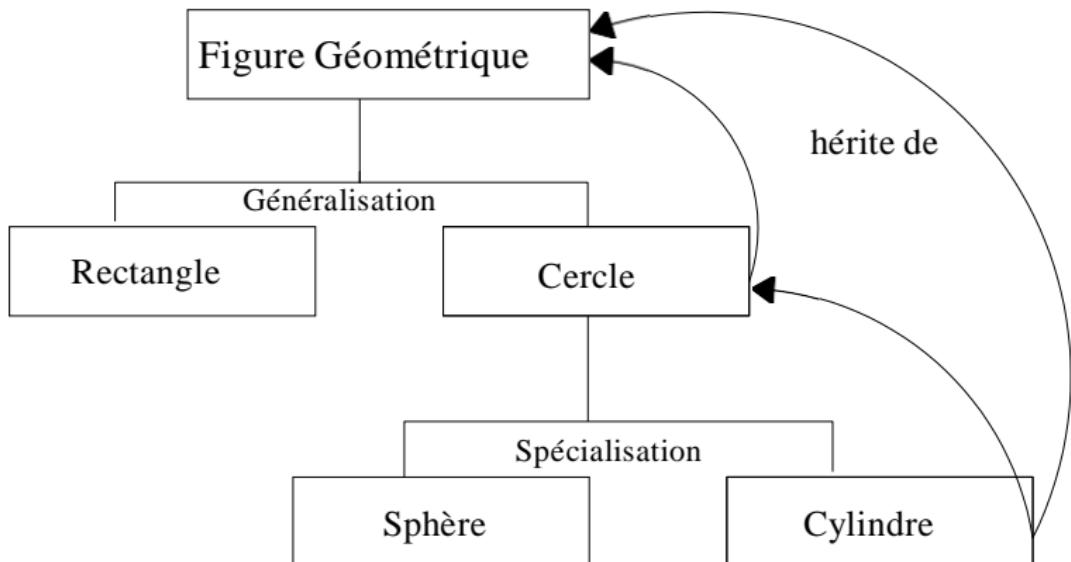
isation

Sous-classe

Sous-classe

Sous-classe

Hiérarchie de classes un exemple



Héritage (2)

Plus précisément, lorsqu'une sous-classe `C1` est créée à partir d'une classe `C`, `C1` va **hériter** de (i.e. *recevoir*) l'ensemble :

- ▶ des attributs de `C`
 - ▶ des méthodes de `C` (sauf les constructeurs/destructeurs)
- ☞ Les attributs et méthodes de `C` vont être disponibles pour `C1` sans que l'on ait besoin de les redéfinir explicitement dans `C1`.

De plus :

- ▶ le type est aussi hérité : `C1` **est** (aussi) **un** `C` :
Pour un `C x;` et un `C1 y;`, on peut tout à fait faire : `x = y;`
(mais bien sûr pas `y = x; !!!`)

Par ailleurs :

- ▶ des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe `C1`
- ☞ ces membres constituent l'**enrichissement** apporté par cette sous-classe.

Héritage (3)

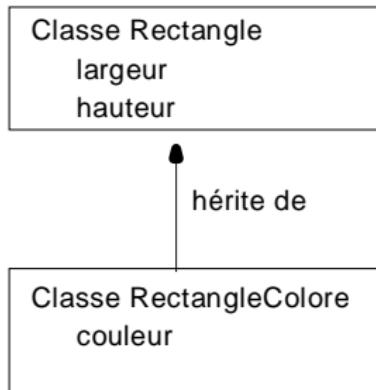
L'héritage permet donc :

- ▶ d'**expliciter des relations** structurelles et sémantiques entre classes
- ▶ de **réduire les redondances** de description et de stockage des propriétés

Héritage (4)

Supposons par exemple que l'on veuille étendre la classe `Rectangle` précédemment définie en lui ajoutant le nouvel attribut `couleur`.

Une façon de procéder est de créer une nouvelle classe, par exemple `RectangleColore`, définie comme une sous-classe de `Rectangle` et contenant le nouvel attribut `couleur`.



- ☞ on évite ainsi de dupliquer inutilement du code commun aux classes `Rectangle` et `RectangleColore` !

Héritage (5)

Par **transitivité**, les instances d'une sous-classe possèdent :

- ▶ les attributs et méthodes (hors constructeurs/destructeur) de l'ensemble des classes parentes (classe parente, classe parente de la parente, etc ...)

La notion d'**enrichissement par héritage** :

- ▶ crée un *réseau de dépendances* entre classes,
 - ▶ ce réseau est organisé en une *structure arborescente* où chacun des nœuds hérite des propriétés de l'ensemble des nœuds du chemin remontant jusqu'à la racine.
- ☞ ce réseau de dépendance définit une **hiérarchie de classes**

Passons à la pratique...

Définition d'une sous-classe en C++ :

Syntaxe :

```
class NomClasseEnfant : public NomClasseParente
{
    /* Déclaration des attributs et méthodes
       spécifiques à la sous-classe */
    //...
};
```

Exemple :

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

Accès aux membres d'une sous-classe

Jusqu'à maintenant, l'accès aux membres (attributs et méthodes) d'une classe pouvait être :

- ▶ soit **public** : visibilité totale à l'intérieur et à l'extérieur de la classe (mot-clé **public**)
- ▶ soit **privé** : visibilité uniquement à l'intérieur de la classe (mot-clé **private** ou par défaut)

Un troisième type d'accès régit l'accès aux attributs/méthodes au sein d'une hiérarchie de classes :

- ▶ l'accès **protégé** : assure la visibilité des membres d'une classe dans les classes de sa descendance [et uniquement elles, et uniquement dans ce rôle (de sous classe, voir exemple plus loin)]. Le mot clé est «**protected**».

Définition des niveaux d'accès

L'ordre de définition conseillé est le suivant :

```
// lien d'heritage si nécessaire seulement
class NomClasse : public NomClasseParente
{
    // par defaut : private

    public:
        // attributs et methodes public
    protected:
        // attributs et methodes protected

    private:
        // attributs et methodes private
};
```

Mais il peut y avoir plusieurs zones publiques, protégées ou privées dans une même définition de classe.

Accès protégé

Le niveau d'accès protégé correspond à une **extension du niveau privé** aux membres des sous-classes

Exemple :

```
class Rectangle
{
public:
    Rectangle(): largeur(1.0), hauteur(2.0) {}
protected:
    double largeur;  double hauteur;
};

class RectangleColore : public Rectangle
{
public:
    // on profite ici de protected
    void carre() { largeur = hauteur; }
protected:
    Couleur couleur;      109
};
```

Accès protégé (2)

Le niveau d'accès protégé correspond à une extension du niveau privé aux membres des sous-classes... **mais uniquement dans ce rôle** (de sous-classe) pas dans le rôle de classe extérieure :

Exemple :

```
class A {  
//...  
protected: int a;  
private:   int prive;  
};  
  
class B: public A {  
public:  
    //...  
    void f(B autreB, A autreA, int x) {  
        a = x; // OK A::a est protected => acces possible  
// prive = x; // erreur : A::prive est private  
  
        // a += autreB.prive; // erreur (meme raison)  
        a += autreB.a; // OK : dans la meme classe (B)  
        // a += autreA.a; // INTERDIT ! : this n'est pas de la meme  
                           // classe que autreA (role externe)  
    }  
};
```

Restriction des accès lors de l'héritage

Les niveaux d'accès peuvent être **modifiés lors de l'héritage**

Syntaxe :

```
class ClasseEnfant: [accès] classeParente
{
    /* Déclaration des membres
       spécifiques à la sous-classe */
    //...
};
```

où **accès** est le mot-clé `public`, `protected` ou `private`. Les crochets entourant un élément [] indiquent qu'il est optionnel.

Les droits peuvent être conservés ou restreints, mais **jamais relachés**!

Par défaut, l'accès est **privé**.¹¹¹

Restriction des accès lors de l'héritage (2)

Récapitulatif des changements de niveaux d'accès aux membres hérités, en fonction du niveau initial et du type d'héritage :

héritage	accès initial		
	public	protected	private
public	public	protected	pas d'accès
protected	protected	protected	pas d'accès
private	private	private	pas d'accès

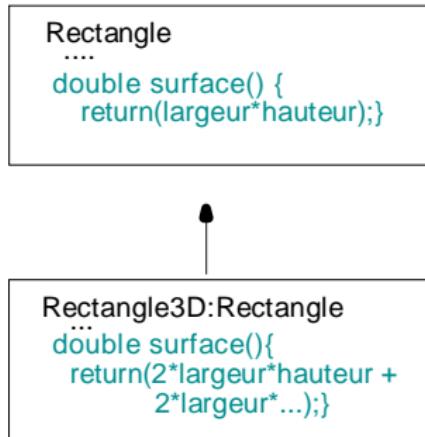
Utilisation des droits d'accès

- ▶ Membres *publics* : accessibles pour les **programmeurs utilisateurs** de la classe
- ▶ Membres *protégés* : accessibles aux **programmeurs d'extensions** par héritage de la classe
- ▶ Membres *privés* : pour le **programmeur de la classe** : structure interne, **modifiable** si nécessaire **sans répercussions** ni sur les utilisateurs ni sur les autres programmeurs.

Masquage dans une hiérarchie

- ▶ Masquage : un identificateur qui en cache un autre
- ▶ Situation possible dans une hiérarchie :
 - ▶ Même nom d'attribut/méthode utilisé sur plusieurs niveaux
 - ▶ Peu courant pour les attributs
 - ▶ Très courant et **pratique** pour les méthodes
- ▶ Exemple :
 - ▶ Rectangle3D hérite de Rectangle
 - ▶ calcul de la surface pour les Rectangle3D
$$2 * (\text{largeur} * \text{hauteur}) + 2 * (\text{largeur} * \text{profondeur}) + 2 * (\text{hauteur} * \text{profondeur})$$
 - ▶ calcul de la surface pour tous les autres Rectangle :
$$(\text{largeur} * \text{hauteur})$$
- ▶ Faut-il re-concevoir toute la hiérarchie ?
 - ▶ Non, on ajoute simplement une méthode **surface** spéciale à Rectangle3D

Masquage dans une hiérarchie (2)



La méthode `surface` de `Rectangle3D` **masque** celle de `Rectangle`

- ▶ Un objet de type `Rectangle3D` n'utilisera donc **jamais** la méthode `surface` de la classe `Rectangle`
- ▶ Vocabulaire OO :
 - Méthode héritée = méthode générale, *méthode par défaut*
 - Méthode qui masque la méthode héritée = *méthode spécialisée*

Masquage dans une hiérarchie (3)

115

Masquage dans une hiérarchie (4)

```
class Rectangle {
public:
    // les constructeurs seraient ici...
    double surface() {return (largeur*hauteur);}

protected:
double largeur; double hauteur;
// le reste de la classe...
};

class Rectangle3D : public Rectangle {
public:
    // les constructeurs seraient ici...
    double surface() { // Masquage !
        return(2.0*(largeur*hauteur) + 2.0*(largeur*profondeur)
               + 2.0*(hauteur*profondeur));
    }

protected:
double profondeur;

// le reste de la classe.1.1.6
};
```

Accès à une méthode masquée

- ▶ Il est parfois souhaitable d'accéder à une méthode/un attribut caché(e)
- ▶ Exemple :
 - ▶ surface des `Rectangle3D` ayant une profondeur nulle (`largeur*hauteur`)
 - ☞ identique au calcul de surface pour les `Rectangle`
- ▶ Code désiré :
 1. Objet non-`Rectangle3D` :
 - ▶ Méthode générale (`surface de Rectangle`)
 2. Objet `Rectangle3D` :
 - ▶ Méthode spécialisée (`surface de Rectangle3D`)
 3. Objet `Rectangle3D` de profondeur nulle :
 - ▶ D'abord la méthode spécialisée
 - ▶ Ensuite appel à la méthode générale depuis la méthode spécialisée

Accès à une méthode cachée (2)

- ▶ Pour accéder aux attributs/méthodes caché(e)s de la super-classe :
 - on utilise **l'opérateur de résolution de portée**
 - Syntaxe : NomClasse::methode ou attribut
 - Exemple : Rectangle::surface()

```
class Rectangle3D : public Rectangle {  
    //... constructeurs, attributs comme avant  
  
    double surface () {  
        if (profondeur == 0.0)  
            // Acces a la methode masquee  
        return Rectangle::surface();  
    else  
        return (2.0*(largeur*hauteur)  
            + 2.0*(largeur*profondeur)  
            + 2.0*(hauteur*profondeur));  
    }  
};
```

Constructeurs et héritage

Lors de l'instanciation d'une sous-classe, il faut initialiser :

- ▶ les attributs *propres à la sous-classe*
- ▶ les attributs *hérités des super-classes*

MAIS...

...il ne doit pas être à la charge du concepteur des sous-classes de réaliser lui-même l'*initialisation des attributs hérités*

L'**accès** à ces attributs peut notamment être **interdit** ! (*private*)

L'initialisation des attributs hérités doit donc se faire au niveau des classes où ils sont explicitement définis.

Solution : l'initialisation des attributs hérités doit se faire en invoquant les *constructeurs des super-classes*.

Constructeurs et héritage (2)

L'invocation du constructeur de la super-classe se fait au **début de la section d'appel aux constructeurs des attributs**.

Syntaxe :

```
SousClasse(liste d'arguments)
: SuperClasse(Arguments),
attribut1(valeur1),
...
attributN(valeurN)
{
    // corps du constructeur
}
```

Lorsque la super-classe admet un constructeur par défaut, l'invocation explicite de ce constructeur dans la sous-classe n'est pas obligatoire

- ☞ le compilateur se charge de réaliser l'invocation du constructeur par défaut^{1,20}

Constructeurs et héritage (3)

Si la classe parente n'admet pas de constructeur par défaut,
l'invocation explicite d'un de ses constructeurs **est obligatoire**
dans les constructeurs de la sous-classe

- ☞ La sur-classe doit admettre *au moins un constructeur explicite*.

Exemple :

```
class Rectangle {  
protected:    double largeur;    double hauteur;  
public:  
    Rectangle(double l, double h) : largeur(l), hauteur(h)  
    {}  
    // le reste de la classe...  
};  
  
class Rectangle3D : public Rectangle {  
protected: double profondeur;  
public:  
    Rectangle3D(double l, double h, double p)  
        // Appel au constructeur de la super-classe  
        : Rectangle(l,h), profondeur(p)  {}  
    // le reste de la classe...  
};
```

Constructeurs et héritage (4)

Autre exemple (qui ne fait pas la même chose) :

```
class Rectangle {
protected:    double largeur;    double hauteur;
public:
    // il y a un constructeur par défaut !
    Rectangle() : largeur(0.0), hauteur(0.0)
    {}
// le reste de la classe...
};

class Rectangle3D : public Rectangle {
protected:    double profondeur;
public:
    Rectangle3D(double p)
        : profondeur(p)
    {}
// le reste de la classe...
};
```

Ici il n'est pas nécessaire d'invoquer explicitement le constructeur de la classe parente puisque celle-ci admet un constructeur par défaut.

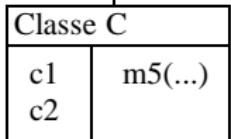
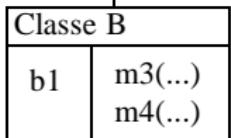
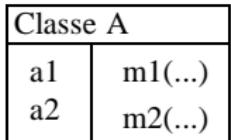
Encore un exemple

Il n'est pas nécessaire d'avoir des attributs supplémentaires...

```
class Carre : public Rectangle {  
public:  
    Carre(double taille) : Rectangle(taille, taille)  
    {}  
    // et c'est tout ! (sauf s'il y avait des "methodes set")  
};
```

Ordre d'appel des constructeurs

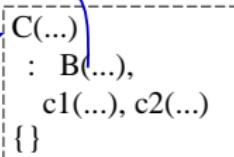
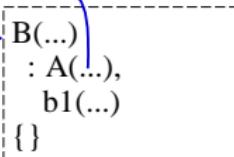
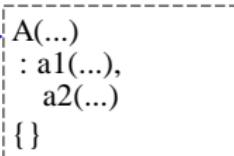
Hierarchie de classes



instanciation :

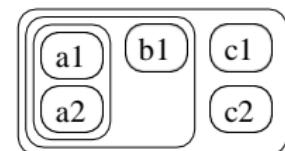
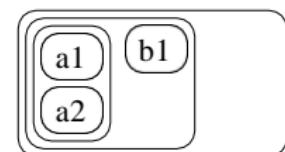
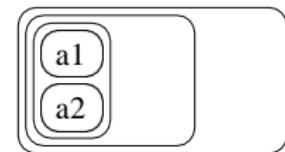
`C mon_c(...);`

Constructeurs



124

Instance



`mon_c`

Ordre d'appel des destructeurs

Les destructeurs sont toujours appelés dans l'ordre inverse (/symétrique) des constructeurs.

Par exemple dans l'exemple précédent, lors de la destruction d'un C, on aura appel et exécution de :

- ▶ C::~C()
- ▶ B::~B()
- ▶ A::~A()

(et dans cet ordre)

(puisque les constructeurs avaient été appelés dans l'ordre

- ▶ A::A()
- ▶ B::B()
- ▶ C::C()

)

Héritage simple et constructeurs par copie (1/7)

```
#include <iostream>
using namespace std ;

class point
{
    int x, y ;
public :
    point (int abs=0, int ord=0)    // constructeur usuel
    { x = abs ; y = ord ;
        cout << "++ point      " << x << " " << y << endl ;
    }
    point (point & p)    // constructeur de recopie
    { x = p.x ; y = p.y ;
        cout << "CR point      " << x << " " << y << endl ;
    }
} ;
```

Héritage simple et constructeurs par copie (1/7)

```
#include <iostream>
using namespace std ;

class point
{
    int x, y ;
public :
    point (int abs=0, int ord=0) // constructeur usuel
    { x = abs ; y = ord ;
        cout << "++ point      " << x << " " << y << endl ;
    }
    point (point & p) // constructeur de recopie
    { x = p.x ; y = p.y ;
        cout << "CR point      " << x << " " << y << endl ;
    }
} ;
```

Rappel : **appel du constructeur par copie** lors

- de l'initialisation d'un objet par un objet de même type
- de la transmission de la valeur d'un objet en argument ou en retour de fonction

Héritage simple et constructeurs par copie (2/7)

```
class pointcol : public point
{
    int coul ;
public :
    // constructeur usuel
    pointcol (int abs=0, int ord=0, int cl=1) : point (abs, ord)
    {
        coul = cl ;
        cout << "++ pointcol " << coul << endl ;
    }
    // constructeur de recopie
    // il y aura conversion implicite de p dans le type point
    pointcol (pointcol & p) : point (p)
    {
        coul = p.coul ;
        cout << "CR pointcol " << coul << endl ;
    }
};
```

Héritage simple et constructeurs par copie (2/7)

```
class pointcol : public point
{
    int coul ;
public :
    // constructeur usuel
    pointcol (int abs=0, int ord=0, int cl=1) : point (abs, ord)
    {
        coul = cl ;
        cout << "++ pointcol " << coul << endl ;
    }
    // constructeur de recopie
    // il y aura conversion implicite de p dans le type point
    pointcol (pointcol & p) : point (p)
    {
        coul = p.coul ;
        cout << "CR pointcol " << coul << endl ;
    }
};
```



Si pas de constructeur par copie défini dans la sous-classe \Rightarrow
Appel du constructeur par copie par défaut de la sous-classe
et donc du constructeur par copie de la super-classe

Héritage simple et constructeurs par copie (3/7)

```
void fct (pointcol pc)
{
    cout << "*** entrée dans fct ***" << endl ;
}

int main()
{
    void fct (pointcol) ; // Déclaration de f
    pointcol a (2,3,4) ;
    fct (a) ; // appel de fct avec a transmis par valeur
}
```

Héritage simple et constructeurs par copie (3/7)

```
void fct (pointcol pc)
{
    cout << "*** entrée dans fct ***" << endl ;
}

int main()
{
    void fct (pointcol) ; // Déclaration de f
    pointcol a (2,3,4) ;
    fct (a) ; // appel de fct avec a transmis par valeur
}
```

Résultat :

```
++ point    2 3
++ pointcol 4
CR point    2 3
CR pointcol 4
*** entrée dans fct ***
```

Héritage simple et constructeurs par copie (3/7)

```
void fct (pointcol pc)
{
    cout << "*** entrée dans fct ***" << endl ;
}

int main()
{
    void fct (pointcol) ; // Déclaration de f
    pointcol a (2,3,4) ;
    fct (a) ; // appel de fct avec a transmis par valeur
}
```

Résultat :

```
++ point    2 3      pointcol a (2,3,4) ;
++ pointcol 4
CR point    2 3
CR pointcol 4
*** entrée dans fct ***
```



Héritage simple et constructeurs par copie (3/7)

```
void fct (pointcol pc)
{
    cout << "*** entrée dans fct ***" << endl ;
}

int main()
{
    void fct (pointcol) ; // Déclaration de f
    pointcol a (2,3,4) ;
    fct (a) ; // appel de fct avec a transmis par valeur
}
```

Résultat :

```
++ point    2 3 } pointcol a (2,3,4) ;
++ pointcol 4   }
CR point    2 3 } fct (a) ;
CR pointcol 4   }
*** entrée dans fct ***
```

Héritage simple et constructeurs par copie (4/7)

Soit une classe B, dérivant d'une classe A :

```
B b0;  
B b1 (b0); // Appel du constructeur par copie de B  
B b2 = b1; // Appel du constructeur par copie de B
```

- Si aucun constructeur par copie défini dans B :
 - ⇒ Appel du constructeur par copie par défaut faisant une copie membre à membre
 - ⇒ Traitement de la partie de b1 héritée de la classe A comme d'un membre du type A ⇒ Appel du constructeur par copie de A
- Si un constructeur par copie défini dans B :
 - B ([const] B&)
 - ⇒ Appel du constructeur de A sans argument ou dont tous les arguments possède une valeur par défaut
 - B ([const] B& x) : A (x)
 - ⇒ Appel du constructeur par copie de A

Héritage simple et constructeurs par copie

Soit une classe B, dérivant d'une classe A :

```
B b0;  
B b1 (b0); // Appel du constructeur par copie de B  
B b2 = b1; // Appel du constructeur par copie de B
```

- Si aucun constructeur par copie défini dans B :
 - ⇒ Appel du constructeur par copie par défaut faisant une copie membre à membre
 - ⇒ Traitement de la partie de b1 héritée de la classe A comme d'un membre du type A ⇒ Appel du constructeur par copie de A
- Si un constructeur par copie défini dans B :
 - B ([const] B&)
 - ⇒ Appel du constructeur de A sans argument ou dont tous les arguments possède une valeur par défaut
 - B ([const] B& x) : A (x)
 - ⇒ Appel du constructeur par copie de A



Le constructeur par copie de la classe dérivée doit prendre en charge l'intégralité de la recopie de l'objet et également de sa partie héritée

Héritage simple et constructeurs par copie (5/7)

```
#include <iostream>
using namespace std;

// Exemple repris et adapté de "C++ - Testez-vous"
// de A. Zerdouk, Ellipses, 2001

class Classe1
{ public :
    Classe1(){ cout << "Classe1::Classe1()" << endl; }

    Classe1(const Classe1 & obj)
        { cout << "Classe1::Classe1(const Classe1&)" << endl; }

};

class Classe2 : public Classe1
{ public:
    Classe2() { cout << "Classe2::Classe2()" << endl; }

    Classe2(const Classe2 & obj)
        { cout << "Classe2::Classe2(const Classe2&)" << endl; }

};
```

Héritage simple et constructeurs par copie (6/7)

```
int main()
{
    Classe2 obj1;
    Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1()
Classe2::Classe2(const Classe2&)
```

Héritage simple et constructeurs par copie (6/7)

```
int main()
{
    →Classe2 obj1;
    Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1()
Classe2::Classe2(const Classe2&)
```

Héritage simple et constructeurs par copie (6/7)

```
int main()
{
    →Classe2 obj1;
    Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1()
Classe2::Classe2(const Classe2&)
```

Héritage simple et constructeurs par copie (6/7)

```
int main()
{
    →Classe2 obj1;
    →Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe{2::Classe2()
Classe1::Classe1()
Classe2::Classe2(const Classe2&)
```

Héritage simple et constructeurs par copie (6/7)

```
int main()
{
    → Classe2 obj1;
    → Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe{2::Classe2()
Classe1::Classe1()
Classe2::Classe2(const Classe2&)

{
```

Héritage simple et constructeurs par copie (6/7)

```
int main()
{
    → Classe2 obj1;
    → Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe{2::Classe2()
Classe1::Classe1()
Classe2::Classe2(const Classe2&)
```



Appel du constructeur de la classe mère car pas d'appel explicite
au copy const. de la classe mère dans le copy const. de la classe fille

Héritage simple et constructeurs par copie (7/7)

Si le constructeur par recopie de la Classe2 défini comme suit :

```
// Appel explicite au copy const. de la classe mère
Classe2(const Classe2 & obj) : Classe1(obj)
    { cout << "Classe2::Classe2(const Classe2&)" << endl; }

int main()
{
    Classe2 obj1;
    Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1(const Classe1&)
Classe2::Classe2(const Classe2&)
```

Héritage simple et constructeurs par copie (7/7)

Si le constructeur par recopie de la Classe2 défini comme suit :

```
// Appel explicite au copy const. de la classe mère
Classe2(const Classe2 & obj) : Classe1(obj)
    { cout << "Classe2::Classe2(const Classe2&)" << endl; }

int main()
{
    →Classe2 obj1;
    Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1(const Classe1&)
Classe2::Classe2(const Classe2&)
```

Héritage simple et constructeurs par copie (7/7)

Si le constructeur par recopie de la Classe2 défini comme suit :

```
// Appel explicite au copy const. de la classe mère
Classe2(const Classe2 & obj) : Classe1(obj)
    { cout << "Classe2::Classe2(const Classe2&)" << endl; }

int main()
{
    → Classe2 obj1;
    Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1(const Classe1&)
Classe2::Classe2(const Classe2&)
```

Héritage simple et constructeurs par copie (7/7)

Si le constructeur par recopie de la Classe2 défini comme suit :

```
// Appel explicite au copy const. de la classe mère
Classe2(const Classe2 & obj) : Classe1(obj)
    { cout << "Classe2::Classe2(const Classe2&)" << endl; }

int main()
{
    → Classe2 obj1;
    → Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1(const Classe1&)
Classe2::Classe2(const Classe2&)
```

Héritage simple et constructeurs par copie (7/7)

Si le constructeur par recopie de la Classe2 défini comme suit :

```
// Appel explicite au copy const. de la classe mère
Classe2(const Classe2 & obj) : Classe1(obj)
    { cout << "Classe2::Classe2(const Classe2&)" << endl; }

int main()
{
    → Classe2 obj1;
    → Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1(const Classe1&)
Classe2::Classe2(const Classe2&)
```

{

Héritage simple et constructeurs par copie (7/7)

Si le constructeur par recopie de la Classe2 défini comme suit :

```
// Appel explicite au copy const. de la classe mère
Classe2(const Classe2 & obj) : Classe1(obj)
    { cout << "Classe2::Classe2(const Classe2&)" << endl; }

int main()
{
    →Classe2 obj1;
    →Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1(const Classe1&) ←
Classe2::Classe2(const Classe2&)

{
```