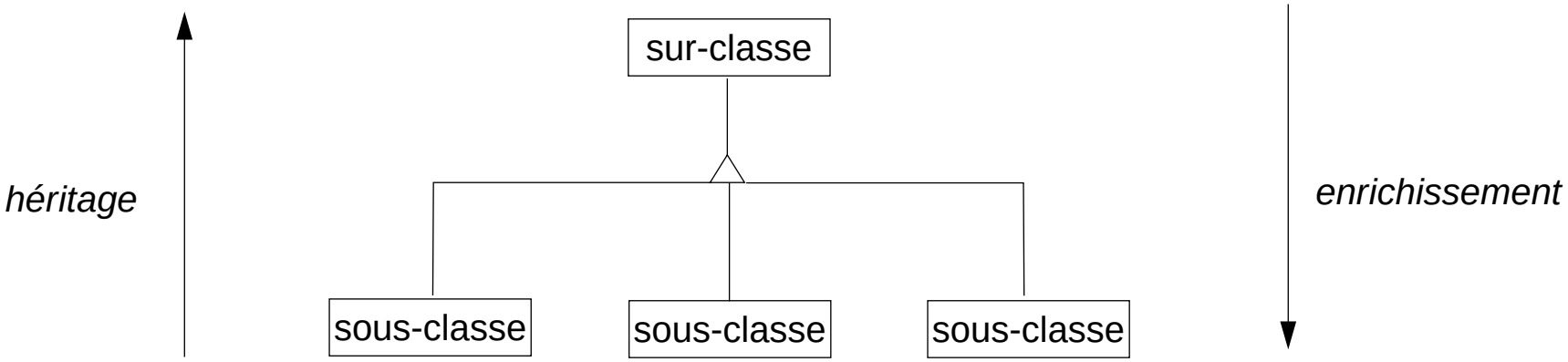
Héritage (1)

Le troisième aspect essentiel[17] des objets est la notion ***d'héritage***

L'héritage est une technique extrêmement efficace
permettant la ***l'enrichissement*** **des classes par la création de classes plus spécifiques, appelées *sous-classes*, à partir de classes plus générales déjà existantes, appelées *sur-classes***[18].



17. Après l'encapsulation (données + comportements) et l'abstraction (classes v.s. instances).

18. Les *sur-classes* sont également appelées classes *parentes*, et les *sous-classes* classes *enfants*.

Héritage (2)

Plus précisement, lorsqu'une sous-classe est définie [à partir d'une sur-classe], elle va ***hériter*** de l'ensemble des attributs et méthodes de sa sur-classe, c'est-à-dire que ces attributs et méthodes seront disponibles pour les instances de la sous-classe, sans qu'il soit nécessaire de les redéfinir explicitement.[19]

Naturellement, des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe, et constituent l'*enrichissement* apportée par cette classe.

L'héritage permet donc à la fois d'expliciter les relations structurelles et sémantiques existant entre classes, et de réduire les redondances de description et de stockage des propriétés.

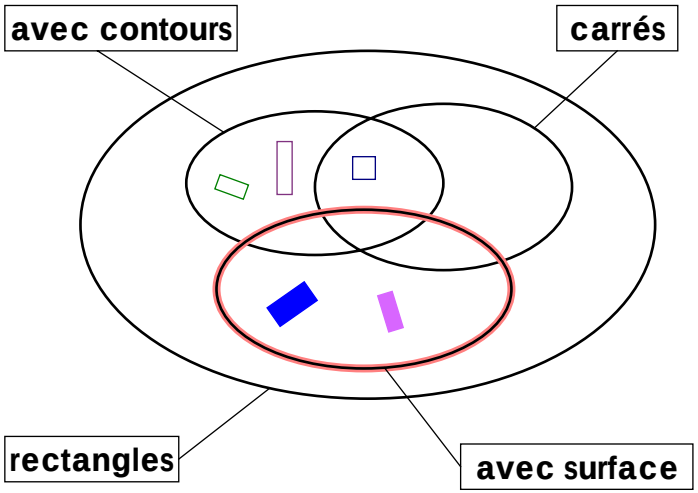
19. Plus précisement, les instances de la sous-classe seront également des instances de la sur-classe, et en auront donc [au moins] toutes les caractéristiques; c'est le concept de polymorphisme objet, qui sera abordé un peu plus loin dans ce cours.

Héritage (3)

Exemple 1:

Supposons que l'on souhaite considérer, en plus de l'ensemble tout à fait général des rectangles, celui particulier des rectangles admettant une surface, représentée par une couleur.

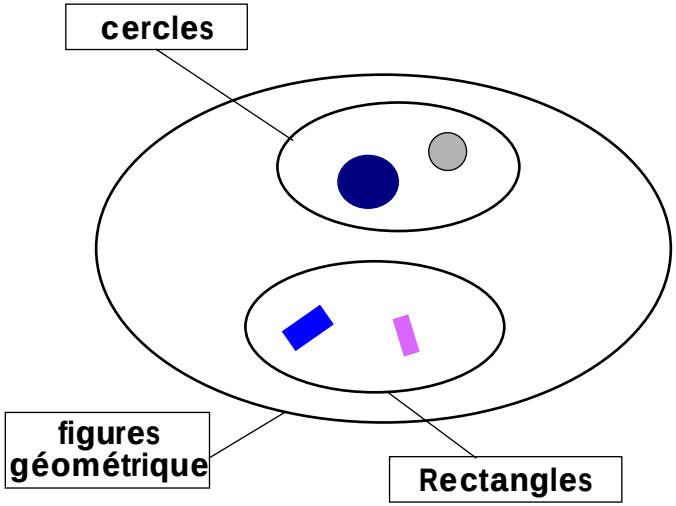
Une façon élégante de procéder est d'étendre la classe `Rectangle` précédemment définie en une sous-classe **`RectanglePlein`**, contenant un nouvel attribut **`couleur`**, correspondant à une couleur.



Exemple 2:

Supposons maintenant que l'on considère que la *couleur* est une propriété fondamentale des *figures géométriques*, et que l'on veuille définir, en plus des rectangles, la classe des *cercles*.

Une façon élégante de procéder est de définir en premier lieu la sur-classe des figures géométriques `FigureGeometrique`, admettant l'attribut `couleur`, et d'enrichir cette classe en deux sous-classes distinctes, celles des rectangles (`Rectangle`) et celle des cercles (`Cercle`).



Héritage (4)

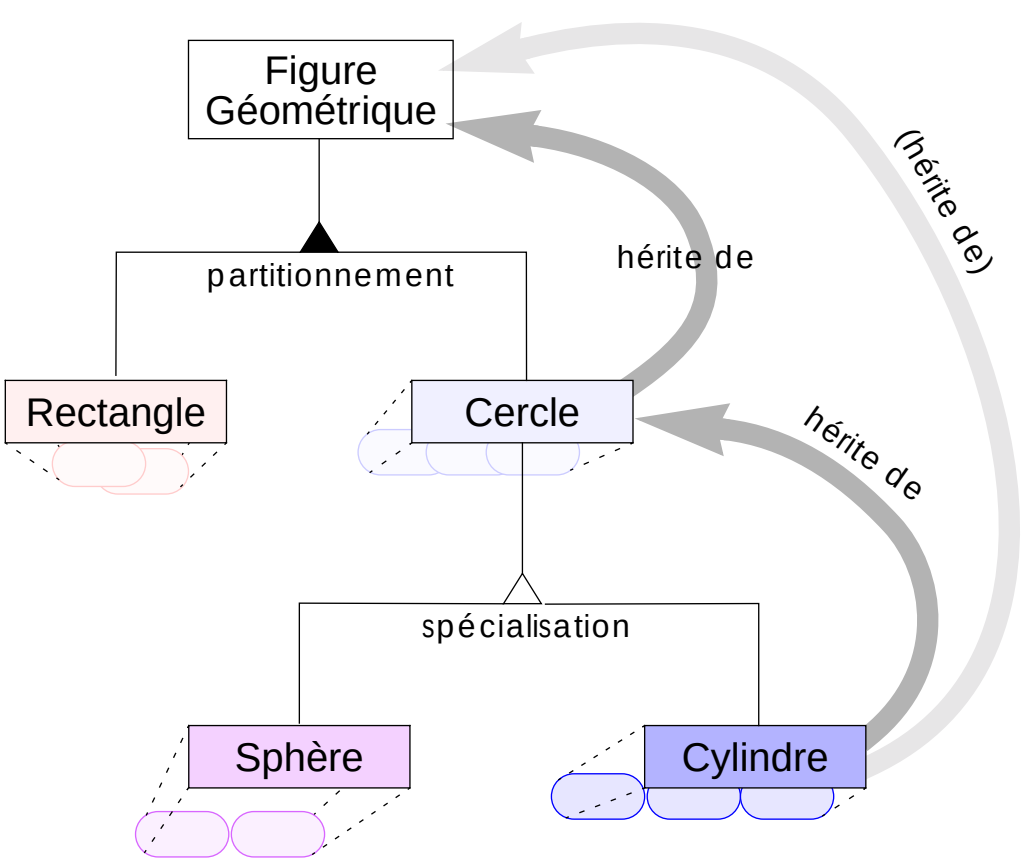
Par transitivité, les instances d'une classe enfant possèdent les attributs et méthodes définis dans la sous-classe et dans l'ensemble des classes dont la sous-classe hérite (i.e. la classe parente, la parente de la classe parente, etc.)[20].

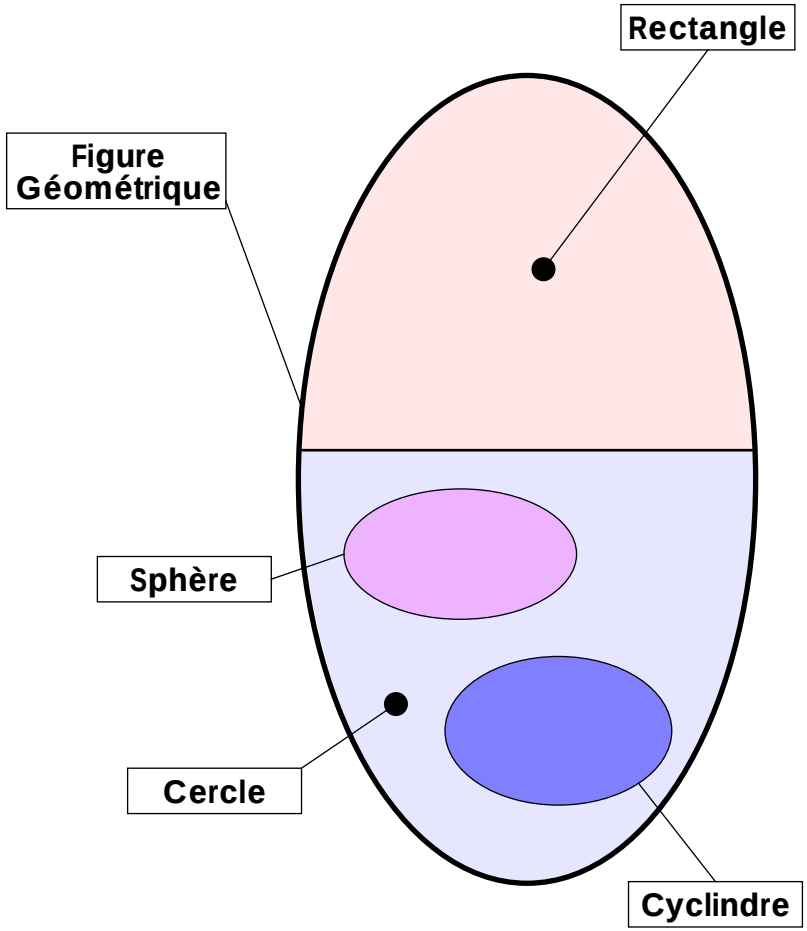
De ce fait, la notion d'enrichissement par héritage (i.e. définition de sous-classes) permettent de créer un **réseau de dépendances** entre classes, organisé en une **structure arborescente**[21] où chacun des noeuds (qui représente une classe donnée) hérite des propriétés de l'ensemble des noeuds subordonnant, c'est-à-dire les noeuds faisant partie du chemin remontant jusqu'à la racine.

20. L'ensemble de ces classes est souvent appelé «*l'ascendance*» de la sous-classe

21. Du moins tant qu'on se limite à un héritage simple (1 seul parent)

Héritage (5)[22]





22. Il est d'usage dans les méthodes Objets d'orienter les arcs de la classe enfant vers la classe parente, afin de bien marquer le sens des dépendances.

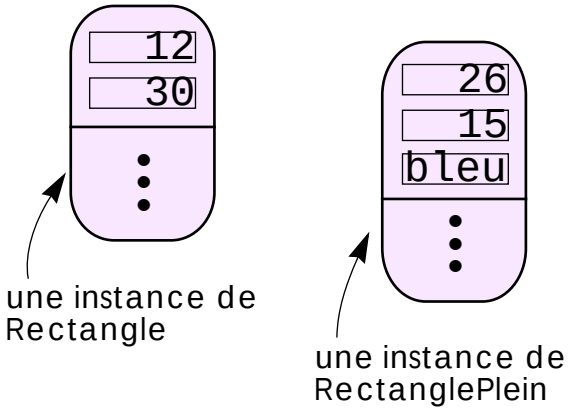
Héritage: syntaxe

La syntaxe permettant de définir des sous-classes est:

```
class <nom de la classe enfant> : <nom de la classe parente>
{
   [public:]
   // déclaration des attributs et méthodes
   // spécifiques à la sous-classe
   ...
};
```

Exemple:

```
class RectanglePlein : Rectangle
{
   public:
      Color couleur;
      // ...
};
```



Accès aux propriétés d'une classe

Dans les objets tels que nous les avons vus jusqu'à maintenant, l'ensemble des propriétés était rendues systématiquement accessibles [par le biais du mot-clef `public`]:

Dans ce cas, il est possible, depuis n'importe quel endroit du programme, d'invoquer une méthode et de lire ou modifier la valeur d'un attribut.

Cette politique étant contraire aux objectifs de l'encapsulation, nous allons voir comment limiter l'accès aux propriétés d'une classe par un mécanisme de ***masquage d'information***.

On définit trois niveaux d'accès possible aux constituants d'une classe:

- ⇨ le niveau **public** (*public*):
  visibilité totale;
- ⇨ le niveau **protégé** (*protected*):
  visibilité restreinte aux méthodes de la classe et de sa descendance;
- ⇨ le niveau **privé** (*private*):
  visibilité strictement restreinte aux méthodes de la classe

Définition des niveaux d'accès

Chaque niveau d'accès est introduit par un mot-clef précédant les propriétés auxquelles il se rapporte:

```
class <nom de classe> [: <nom de la classe parente>]
{
   // par défaut: niveau privé

   public:
   // attributs et méthodes du niveau public

   protected:
   // attributs et méthodes du niveau protégé

   private:
   // attributs et méthodes du niveau privé
};
```

Remarquons que l'ordre n'est pas imposé (même si celui indiqué est conseillé), et qu'il n'est, de plus, pas obligatoire de regrouper les propriétés ayant le même niveau d'accès (i.e. on peut définir plusieurs zones «publiques», «protégées» ou «privées».

Accès «public»

Les propriétés définies comme ***publiques***
sont accessibles sans restriction dans l'ensemble du programme:

```
class Insouciant
{
public:
   int x;
   Insouciant(const int x)
   :x(x)
   { }
};

int main()
{
   Insouciant a(3);
   a.x++;
}
```

Comme il l'a été mentionné précedemment, les **attributs** publics, sorte de variables globales, constituent une entorse au principe d'encapsulation en rendant visible la structure interne des objets.

Pour cette raison, il est recommandé de les éviter le plus possible.

Accès «privé»

Les propriétés définies comme ***privées*** ne sont accessibles que pour les méthodes de la classe.

Remarquons qu'en C++, les niveaux d'accès sont définis sur les classes et pas sur les instances. La conséquence est qu'une méthode invoquée pour une instance X pourra accéder sans restriction aux propriétés d'une autre instance Y de la même classe, et en particulier à ses attributs «privés».

```
class Parano {
public:
   Parano(const int x):x(x) { }
   int getValue() const { return x;}
   void setValue(const int x) {
      if (x>0) {this->x = x;} }
private:
   int x;
};
int main() {
   Parano a(3);
   a.setX(a.getX()++);
}
```

Si la restriction d'accès aux attributs en général et aux méthodes sensibles est préférable, elle implique toutefois l'écriture d'un certain nombres de méthodes supplémentaires, permettant l'accès et la modification (contrôlée) des attributs.

Accès «protégé»

Les propriétés définies comme ***protégées*** ne sont accessibles que pour les méthodes de la classe et de sa descendance.

Le niveau protégé correspond donc à une extension du niveau privé aux propriétés des sous-classes
(ce qui n'est effectivement pas le cas avec le niveau privé !)

```
class ChefDeFamille{
public:
   ChefDeFamille():x(0) { }
   int getValue() const { return x;}
protected:
   int x;
};

class Rejeton : ChefDeFamille {
public:
   void setValue(const int x) { this->x = x; }
};
```

Restriction des accès lors de l'héritage

Il est possible de **modifier les niveaux d'accès lors de l'héritage**;
i.e. les propriétés héritées peuvent avoir [dans les sous-classes] des niveaux d'accès différents de ceux définis dans la sur-classe.

Remarquons toutefois que les droits ne peuvent être que conservés ou restreints par l'héritage, mais en aucun cas relâchés !

Le niveau d'accès souhaité pour les propriétés héritées est spécifié en préfixant l'identificateur de la classe parente par l'un des 3 mots-clef désignant les niveaux d'accès:

```
class <classe enfant> : [<accès>] <classe parente>
{
   // déclaration des attributs et méthodes
   // spécifiques à la sous-classe
};
```

Où *<accès>* peut être `public`, `protected` ou `private` (par défaut).

Restriction des accès lors de l'héritage

La table ci-dessous résume les changements de niveaux d'accès aux propriétés héritées, en fonction du niveau initial et du type d'héritage:

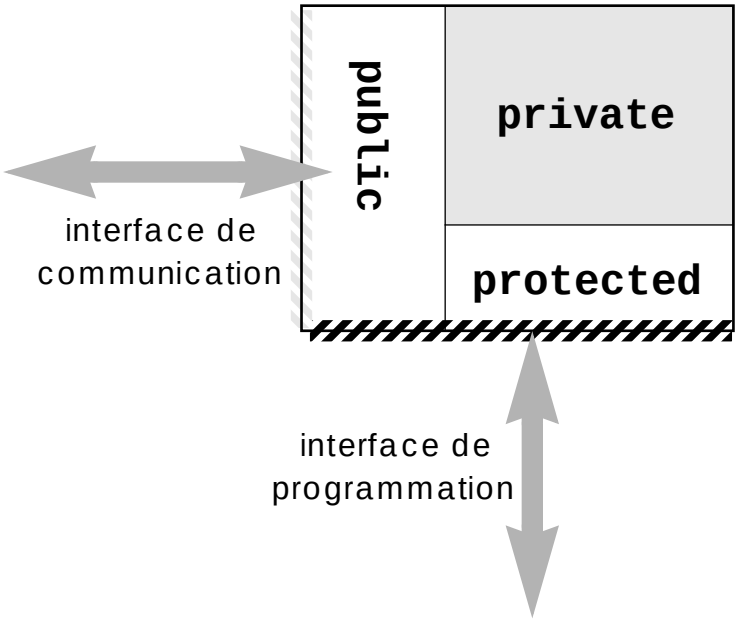
accès initial \ héritage	***public***	***protected***	***private***
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	*pas d'accès*	*pas d'accès*	*pas d'accès*

- le niveau d'accès des propriétés initialement définies «privées» ne sont pas affectés puisque ces propriétés sont de toutes les manières masquées hors de la classe;
- pour le reste, le type d'héritage constitue en fait une «limite supérieure à la visibilité»: les propriétés initialement définies «publiques» le restent, deviennent «protégées» ou «privées» en fonction du type d'héritage, et les propriétés initialement définies «protégées» le restent ou deviennent «privées» en fonction du type d'héritage.

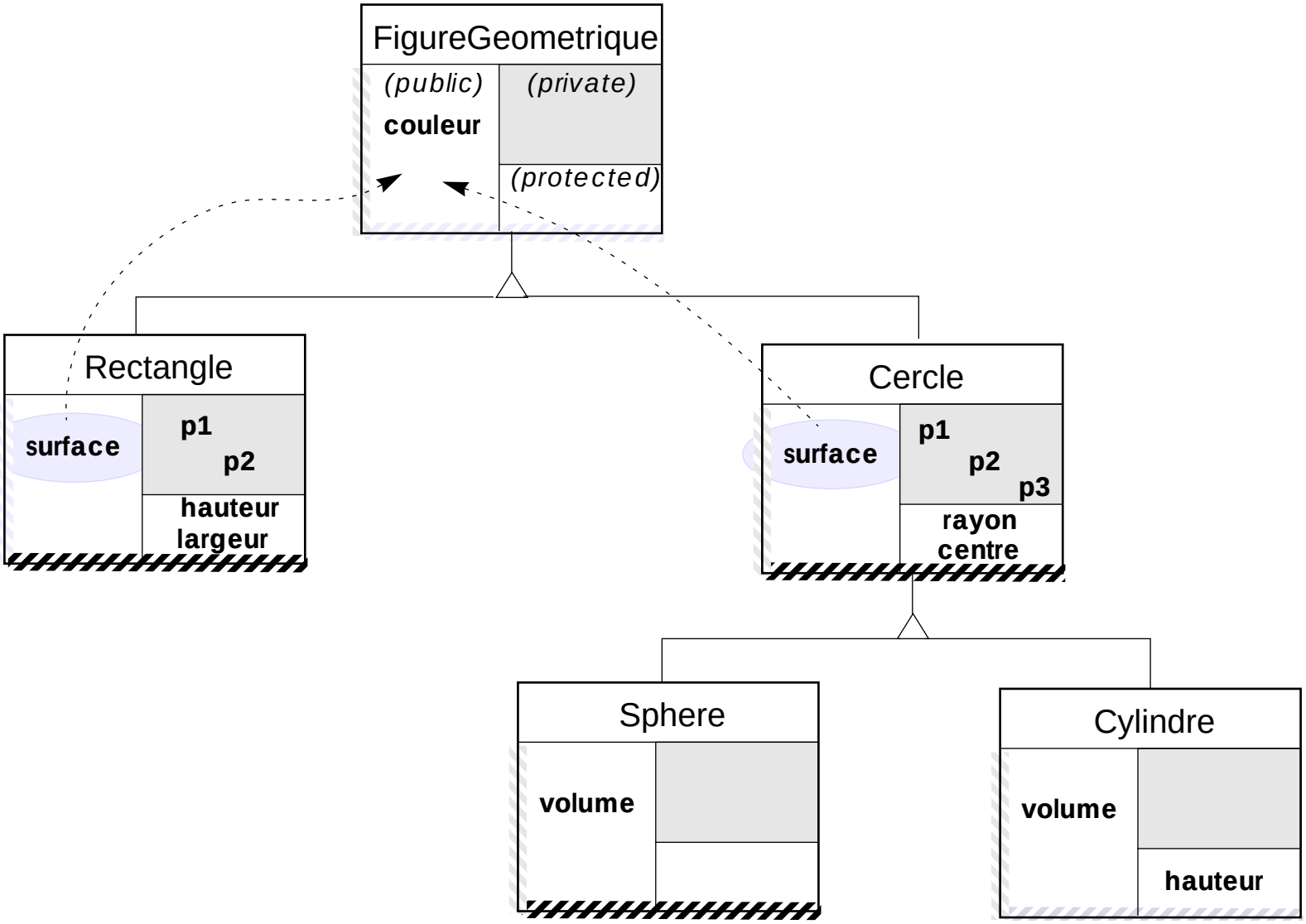
Utilisation des droits d'accès (synthèse)

Les troix niveaux d'accès (public, protégé et privé) définis pour les constituants des objets peuvent (par exemple) être interpretés de la façon suivante:

- les propriétés «publiques» sont celles qui doivent être accessibles pour les **utilisateurs** des instances d'une classe; elles constituent de ce fait ***l'interface de communication*** définie pour ces instances (pensez à l'utilisation que vous faites de la classe `vector`);
- les propriétés «protégées» sont celles que le concepteur de la classe met à disposition des **programmeurs** pour d'éventuelles modifications [par héritage] de cette classe (enrichissement); elles constituent de ce fait ***l'interface de programmation*** définie pour cette classe.
- les propriétés «privées» sont celles qui constituent la structure interne de la classe, que le concepteur ne veut pas rendre visible, de façon à pouvoir les modifier si nécessaire sans que cela ait un impact ni sur les utilisateurs ni sur les programmeurs.



Utilisation des droits d'accès (exemple)



Modifications de propriétés héritées (1)

L'héritage permet non seulement d'ajouter des propriétés et d'utiliser (dans la mesure des droits d'accès) celles définies dans les classes de l'ascendance, mais il permet également de *modifier* ces attributs et méthodes[23]:

Par exemple, il est possible de *remplacer* une méthode héritée:

```
class C
{
   TypeRetour methode1(arg1, ...)
   {
      // définition originelle
      // de la méthode
   }
};
```

```
class Csub : C
{
   TypeRetour methode1(arg1, ...)
   {
      // «nouvelle» définition
      // de la méthode
   }
};
```

Dans ce cas, pour les instances de la classe `C1`, la nouvelle définition de la méthode `methode1` *remplace* celle héritée de la classe parente `C`.

23. Formellement, l'utilisation du terme «modifier» est erronée dans ce contexte puisqu'il s'agit en réalité d'ajouts, et qu'il est toujours possible de faire référence à la propriété originelle, au moyen de l'opérateur de résolution de portée (voir l'exemple dans la suite du cours).

Modification de propriétés héritées (2)

Exemple:

Redéfinition de la méthode surface pour la sous-classe `RectanglePlein`

```
class Rectangle
{
public:
   int hauteur;
   int largeur;

   // les constructeurs ...

   int surface()
   {
      return (hauteur*largeur);
   }

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

```
class RectanglePlein: Rectangle
{
public:
   int interieur;

   // les constructeurs ...

   int surface()
   {
      return abs(hauteur*largeur);
   }

   // le reste de la sous-classe ...
};
```

Modification de propriétés héritées (3)

Lors de redéfinition d'attributs ou de méthodes,
les propriétés originelles ne sont pas réellement remplacées
dans la sous-classe mais simplement masquées par les nouvelles.

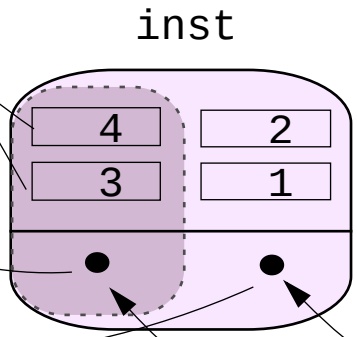
De ce fait, il est toujours possible d'y faire référence,
en utilisant l'opérateur de résolution de portée « :: »

<nom de surclasse> `::` *<méthode ou attribut>*

Exemple:

```
class C {
   int a,b;
   int m(){ return (a+=b);}
};

class C1: public C {
   int b,c;
   int m(){
      b = C::m();
      return (c += C::a;)
   }
};
```




```
int main() {
   C1 inst;
   inst.a = 4;
   inst.b = 2;
   inst.c = 1;
   inst.C::b = 3;
   inst m();
   inst.C::m();
}
```

Constructeurs et héritage (1)

Lors de l'instanciation d'une sous-classe,
il est non seulement nécessaire d'initialiser les attributs propres
à cette sous-classe, **mais également ceux hérités des sur-classes**.

Il n'est pas raisonnable d'obliger le concepteur de la sous-classe
à réaliser lui-même l'initialisation des attributs hérités, car en particulier, il est
fort possible que l'accès à ces attributs soit purement et simplement interdit.[24]

> L'initialisation des attributs hérités doit [donc] se faire au niveau des classes dans lesquelles ces attributs sont explicitement définis.

La solution pour cela est d'invoquer les constructeurs de ces classes.[25]

24. Il existe une seconde raison, plus fondamentale, liée à la nature polymorphe des instances de classes dérivées (c.f suite du cours).

25. L'ordre dans lequelle ces invocations doivent être faites est évidemment celui des dérivations (des ancètres vers les descendants), puisque les attributs hérités sont potentiellement utilisables par les constructeurs des sous-classes (il faut donc qu'ils soient initialisés).

Constructeurs et d'héritage (2)

L'invocation [explicite] du constructeur de la sur-classe se fait **au début de la section d'initialisation** des constructeurs de la sous-classe, et sa syntaxe est identique à celle de l'invocation d'une méthode quelconque:

```
classe SousClasse : public SurClasse
{
   SousClasse(<liste de paramètres>)
   :  SurClasse(<arguments>),
      <attribut1>(<valeur1>),
      ...
      <attributn>(<valeurn>)
   {
      // corps du constructeur
   } ... };
```

Remarquons que l'invocation explicite du constructeur de la sur-classe n'est pas obligatoire dans le cas où cette classe admet un constructeur par défaut; le compilateur se chargera alors de réaliser l'invocation de ce constructeur.

Constructeurs et héritage (3)

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

⇨ La sous-classe doit obligatoirement admettre (au moins) un constructeur explicite:

```
class Rectangle
{
public:
   int hauteur;
   int largeur;

   Rectangle(int h, int l)
   :  hauteur(h),
      largeur(l)
   { }
   // le reste de la classe ...
};
```

```
class RectanglePlein: public Rectangle
{
public:
   int couleur;

   RectanglePlein(int i, int h, int l)
   :  Rectangle(h,l),
      couleur(i)
   { }
   // le reste de la sous-classe ...
};
```

Dans cet exemple, le programmeur est contraint de définir au moins un constructeur pour `RectanglePlein`, afin d'indiquer au compilateur comment invoquer le constructeur de `Rectangle`.
Remarquons que les listes de paramètres des deux constructeurs sont totalement indépendantes (*i.e. l'ordre et la nature des arguments des cstr. de la sur-classe n'imposent pas de contraintes sur les cstr. de la sous-classe*).

Constructeurs et héritage (4)

Exemple II:

Dans cet autre exemple, la classe `Rectangle` admet un constructeur par défaut (explicite):

```
class Rectangle
{
public:
   int hauteur;
   int largeur;

   Rectangle()
   :  hauteur(0),
      largeur(0)
   { }
   // le reste de la classe ...
};
```

invocation implicite

```
class RectanglePlein: public Rectangle
{
public:
   int couleur;

   RectanglePlein(int i, int h, int l)
   :  couleur(i)
   { }
   // le reste de la sous-classe ...
};
```

Il n'est donc pas nécessaire d'invoquer explicitement ce constructeur depuis celui de la sous-classe `RectanglePlein`, et il n'est donc pas non plus obligatoire de fournir de constructeur explicite pour cette classe, celui généré par défaut par le compilateur pouvant suffire (du moins en ce qui concerne l'initialisation des attributs hérités).

Ordre d'appel des constructeurs