ISEN
ALL IS DIGITAL!
LILLE

yncréa

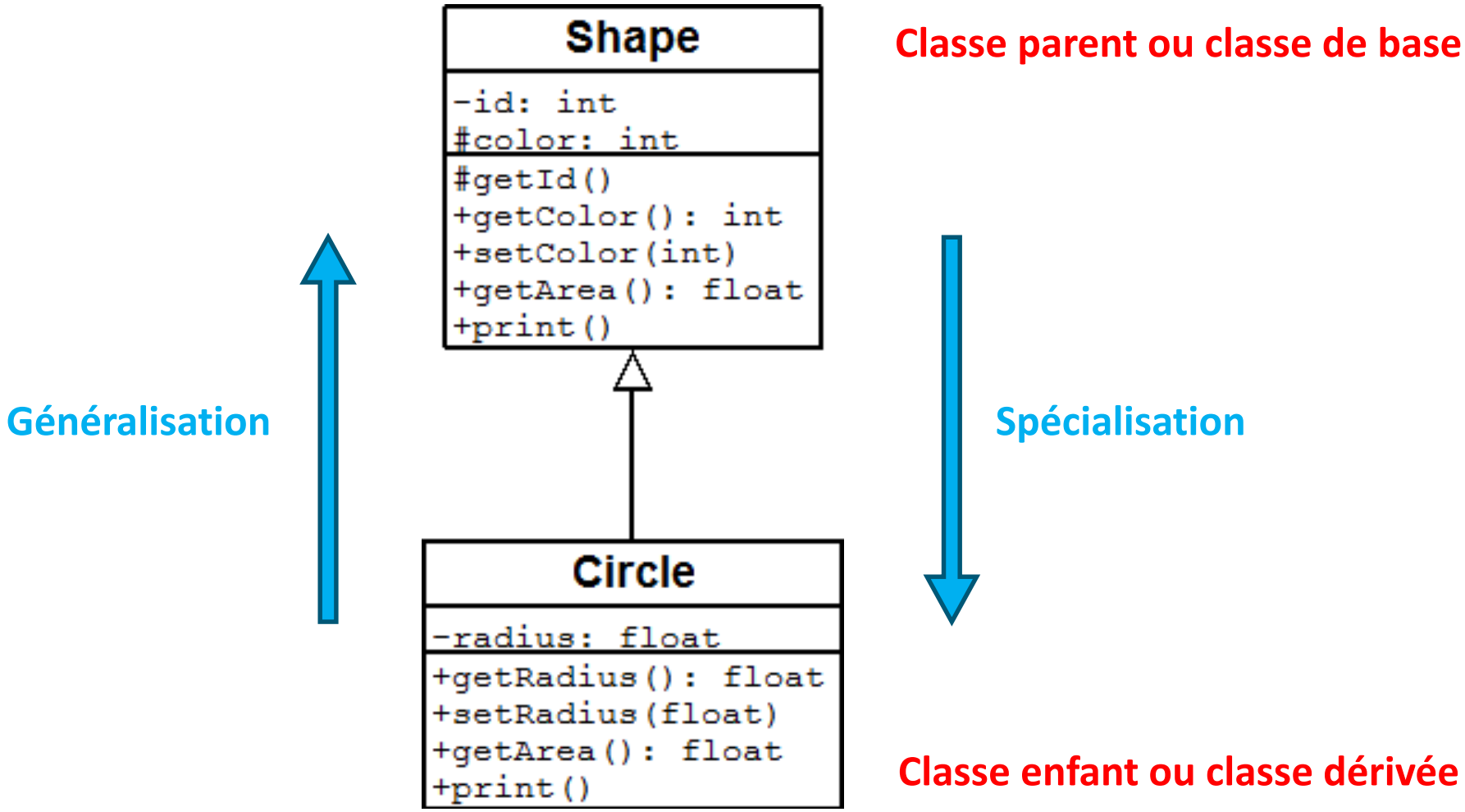
CIR²

Le Langage C++

L'héritage

Nacim IHADDADENE

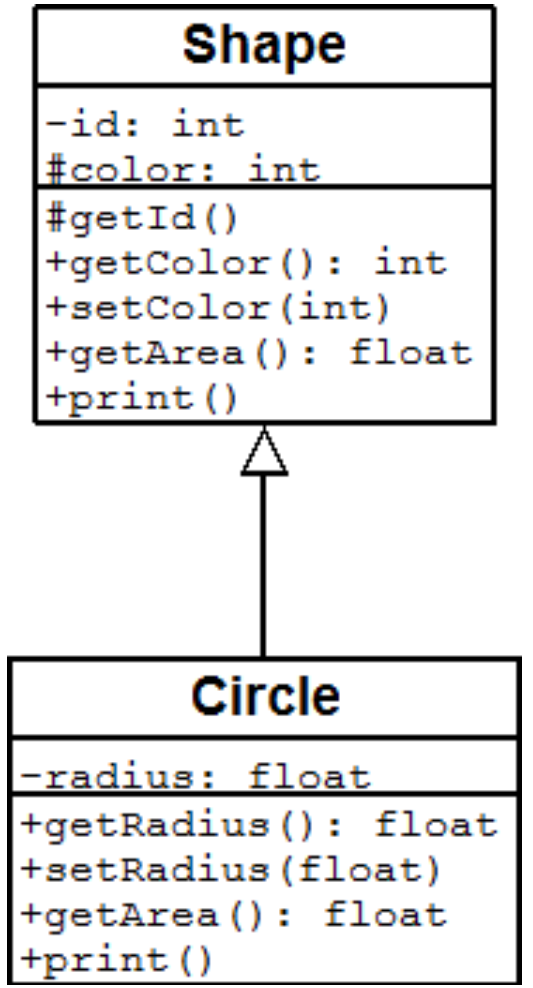
Principe de l'héritage



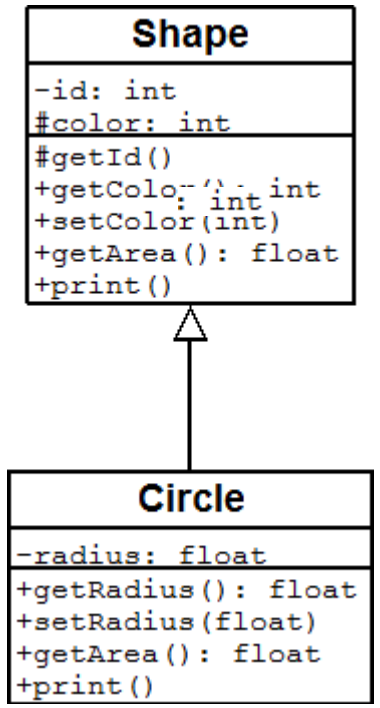
Principe de l'héritage

- La classe enfant :
  1. Hérite de **tous les membres** de la classe parent
  2. Accède seulement au membres **publiques** ou **protégés** du parent
  3. Peut **ajouter** de nouveaux membres
  4. Peut **réécrire** des méthodes de la classe parent

     (polymorphisme dynamique)



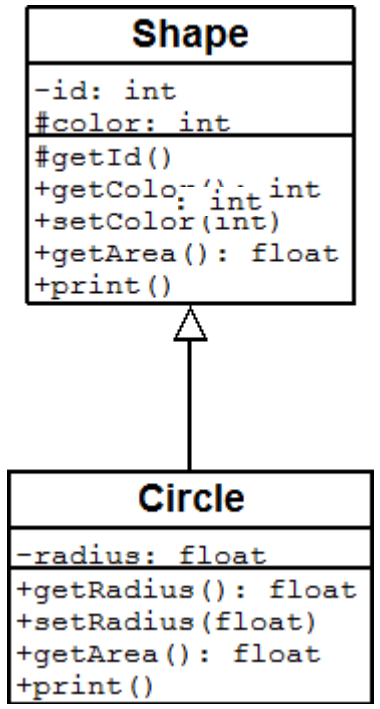
Syntaxe



Shape.h

```
class Shape
{
private:
    int id;

protected:
    int color;
    int getId();

public:
    ... // constructors
    int getColor();
    void setColor();
    float getArea();
    void print();
};
```

Syntaxe



Circle.h

```
#include "Shape.h"

Class Circle : public Shape
{
private:
    float radius;

public:
    ... // constructors
    float getRadius();
    void setRadius(float);
    float getArea();
    void print();
};
```

Héritage et accès aux membres

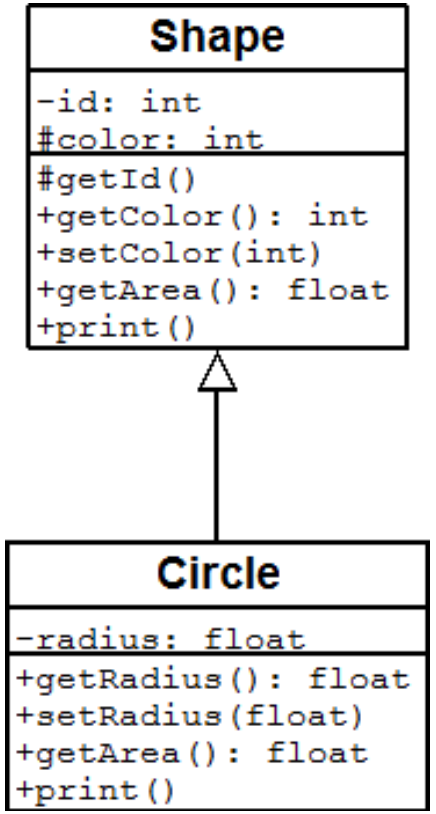
- La classe enfant hérite des membres de la classe parent
- Ce qui implique que la classe enfant connait les membres de la classe parent
- **Mais le contraire est faux**

Exemples

CircleTest.cpp

```
... // all required lines

int main()
{
Shape s;

s.setColor(1);

return 0;
}
```

✓ Correct

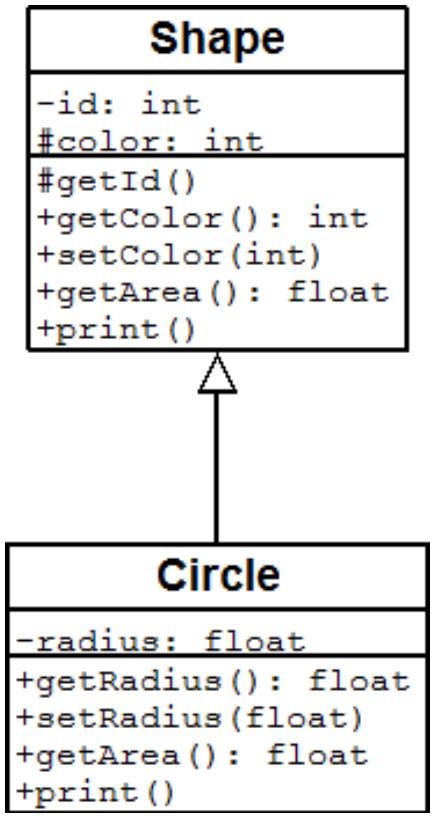


Exemples

CircleTest.cpp

```
... // all required lines

int main()
{
Circle c;

c.setColor(1);

return 0;
}
```

✓ Correct

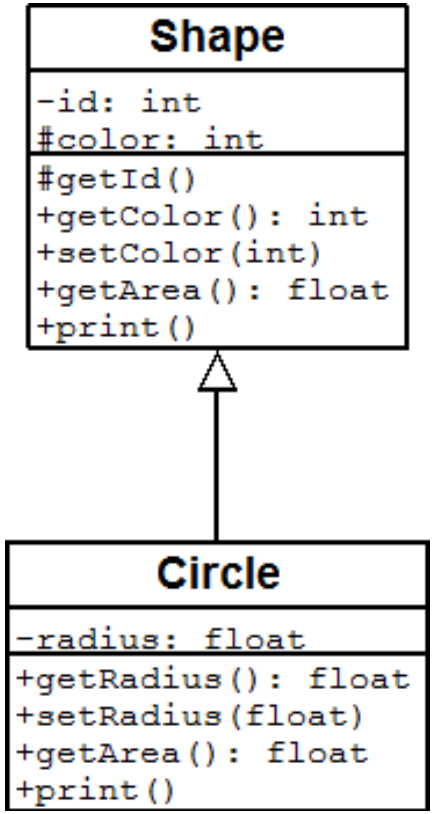


Exemples

CircleTest.cpp

```
... // all required lines

int main()
{
Circle c;

c.setRadius(2.1);

return 0;
}
```

✓ Correct

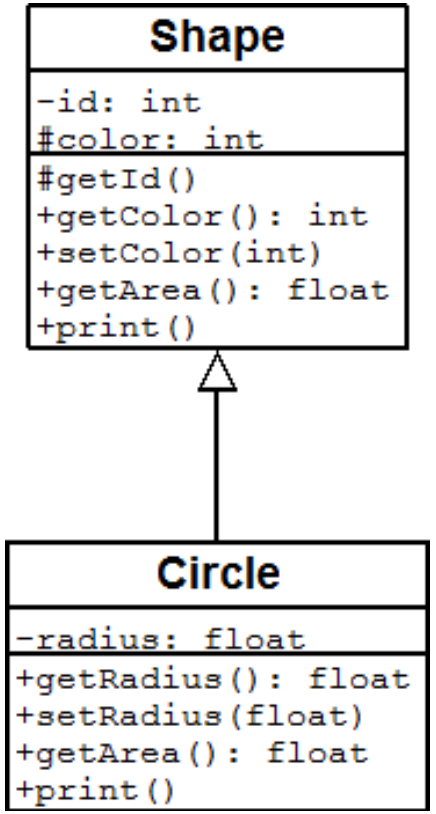


Exemples

CircleTest.cpp

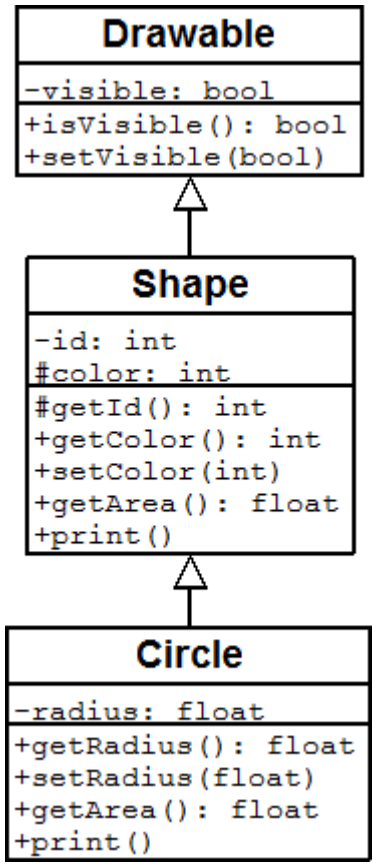
```
... // all required lines

int main()
{
Shape s;

s.setRadius(2.1);

return 0;
}
```

✗ Incorrect



Transitivité de l'héritage

- L'héritage des membres est transitif:
  - Si Shape hérite de Drawable
  - Et Circle hérite de Shape
  - Alors Circle hérite de Drawable



- Une classe connait les membres de ses ancêtres, mais pas ceux de ses descendants.

- La recherche des membres se fait de l'enfant vers le parent, tant qu'il existe un parent.

Accès aux membres

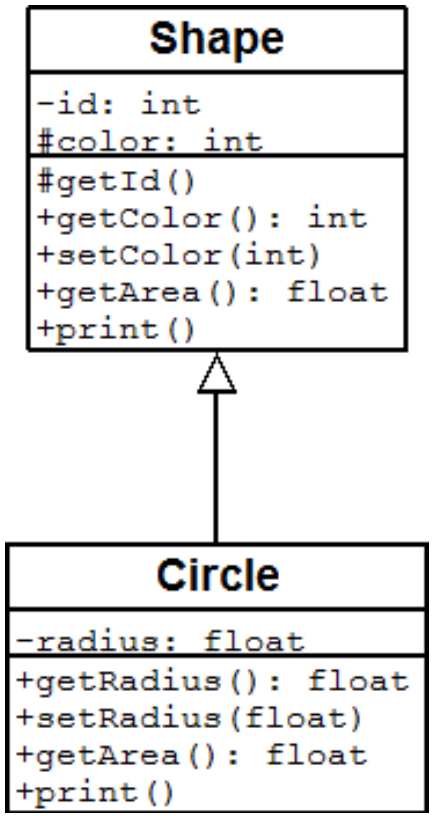
- Une méthode d'une classe enfant **ne peut pas accéder** directement aux membres **privés** de la classe parent.

- Une méthode d'une classe enfant **peut accéder** directement aux membres **publics** ou **protégés** de la classe parent.

Exemples

CircleTest.cpp

```
... // all required lines

int main()
{
    Circle c;
    cout << "My color is " << c.getColor() << endl;
    return 0;
}
```



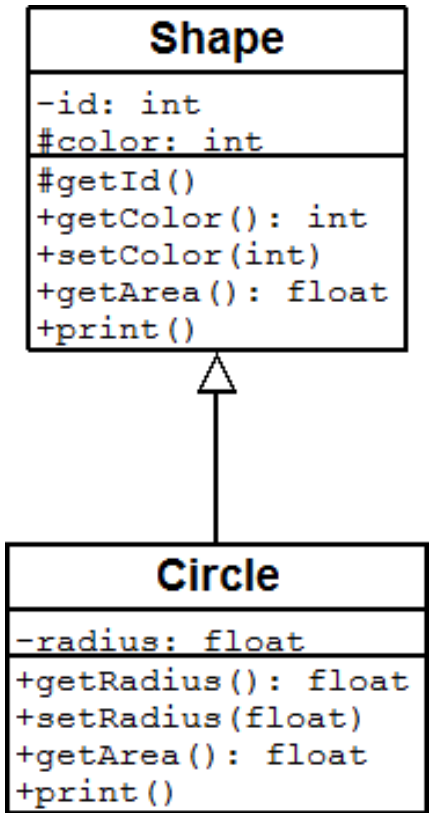
✓ Correct, getColor() est publique

Exemples

CircleTest.cpp

```
... // all required lines

void Circle::print()
{
    cout << "My color is " << getColor() << endl;
}
```

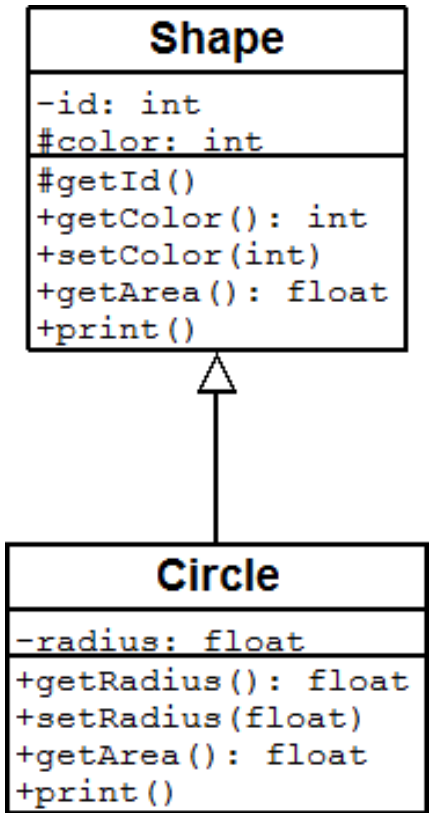
✓ Correct, getColor() est publique



Exemples

CircleTest.cpp

```
... // all required lines

int main()
{
    Circle c;
    cout << "My color is " << c.color << endl;
    return 0;
}
```

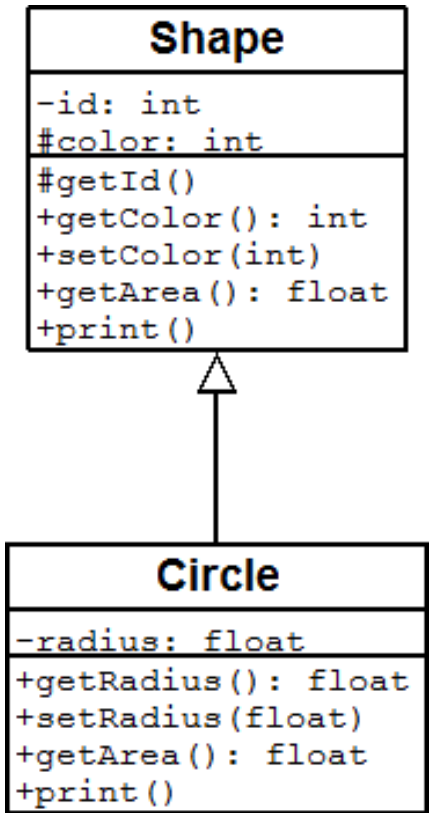


✗ Incorrect: color est protégé

Exemples

CircleTest.cpp

```
... // all required lines

void Circle::print()
{
    cout << "My color is " << color << endl;
}
```



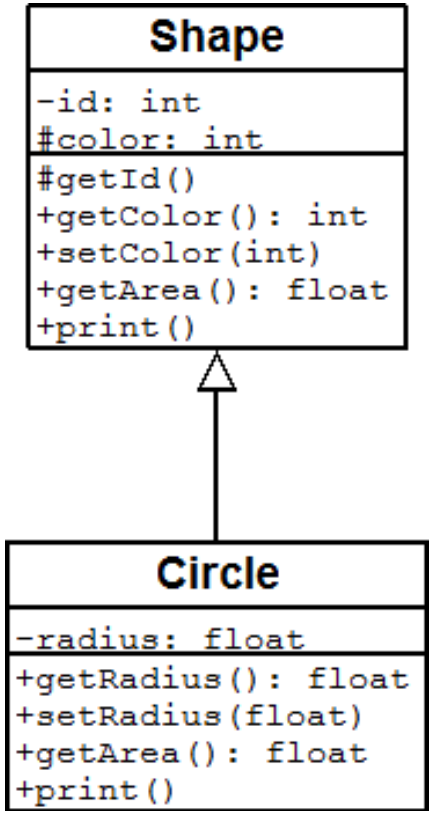
✓ Correct, color est protégé

Exemples

CircleTest.cpp

```
... // all required lines

int main()
{
    Circle c;
    cout << "My id is " << c.id << endl;
    return 0;
}
```

✗ Incorrect: id est privé

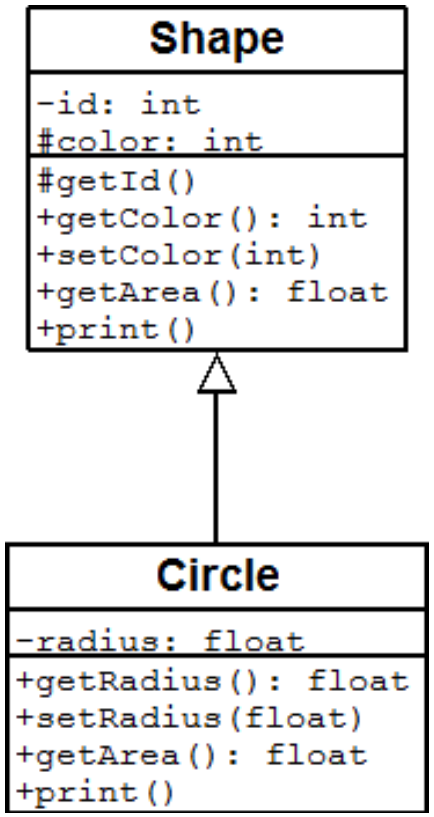


Exemples

CircleTest.cpp

```
... // all required lines

void Circle::print()
{
    cout << "My id is " << id << endl;
}
```

✗ Incorrect: id est privé



Le polymorphisme

Le polymorphisme

- Le mot polymorphie vient du grec et signifie : « qui peut prendre plusieurs formes ».
- Ce concept consiste à fournir une interface unique à des entités pouvant avoir différents types.
- Le polymorphisme (en plus de l'encapsulation et de l'héritage) est un des concept fondamentaux de tout langage orienté objet.

- Le polymorphisme est présent en C++ via les mécanismes de
  - **surcharge** (polymorphisme statique)
  - **réécriture ou redéfinition** (polymorphisme dynamique)

La réécriture d'une méthode

- Une méthode du parent peut être **réécrite** ou **redéfinie** dans la classe enfant.
- En anglais : *method overriding*

- Dans le but :
  - De redéfinir ou de remplacer le comportement d'une méthode
  - Ou de compléter le comportement d'une méthode

IMPORTANT : Une méthode redéfinie dans la classe enfant **cache (ou remplace)** toutes les versions de cette méthode dans la classe parent.

Exemple : Remplacer un comportement

Shape.cpp

```
... // all required lines

void Shape::print()
{
    cout << "I am a Shape" << endl;
}

...
```

Circle.cpp

```
... // all required lines

void Circle::print()
{
    cout << "I am a Circle" << endl;
}

...
```

Exemple : Remplacer un comportement

ShapeTest.cpp

```
... // all required lines

int main()
{
Shape s;
Circle c;

s.print();

cout << "------------" << endl;

c.print();

return 0;
}
```

Standard output

```
I am a Shape
------------
I am a Circle
```

Exemple : Compléter un comportement

Shape.cpp

```
... // all required lines

void Shape::print(){
    cout << "I have an id" << endl;
    cout << "I have a color" << endl;
}

...
```

Circle.cpp

```
... // all required lines

void Circle::print(){
Shape::print();
cout << "I have a radius" << endl;
}

...
```

Exemple : Compléter un comportement

ShapeTest.cpp

```
... // all required lines

int main()
{
Shape s;
Circle c;

s.print();

cout << "------------" << endl;

c.print();

return 0;
}
```

Standard output

```
I have an id
I have a color
------------
I have an id
I have a color
I have a radius
```

Les règles d’affectation

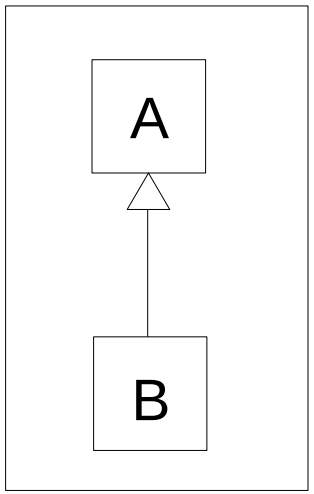
Les règles d'affectation

Une instance de l'enfant peut être affecté à une instance du parent

✓ `parent = child`

✓ `parent* = child*`

✓ `parent& = child`




```
A a;
B b;
```

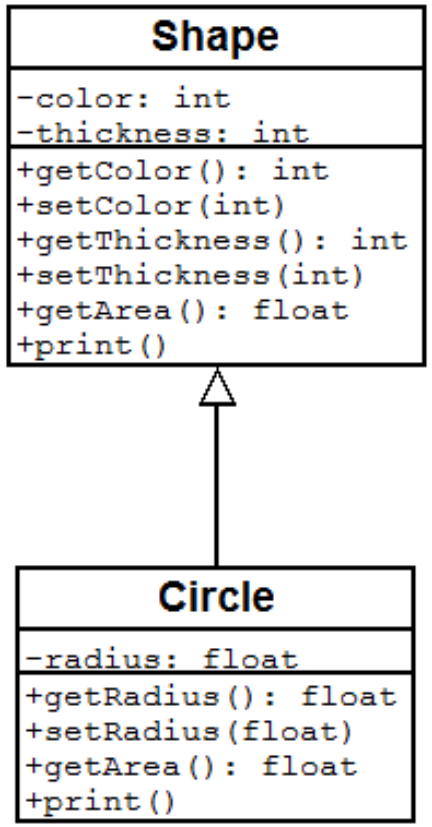
✓ `A a1 = b;`

✗ ~~`B b1 = a;`~~

Les méthodes virtuelles

Exemple

```
... // all required lines

int main()
{
Shape* s;
Circle* c = new Circle();

s = c;      // correct

s->print();

return 0;
}
```



Question : Quelle version de print() sera appelée ??
Réponse : Celle de Shape (même si elle pointe vers une instance de Circle)

Les méthodes virtuelle

- Etant donné un pointeur d'une classe parent qui contient l'adresse d'une instance d'une de ses classes enfants;

- Comment faire pour que le compilateur reconnaisse le vrai type de l'instance (enfant)
- et appelle la méthode adéquate (redéfinie chez l'enfant) ??

- C'est le rôle du mot clé C++ : `virtual`

Syntaxe

Shape.h

```
... // all required lines

class Shape{
    ...
    public:
    virtual void print(){cout << "I am a Shape" << endl;};
    ...
};
```

Circle.h

```
... // all required lines

class Circle: public Shape{
    ...
    public:
    void print(){cout << "I am a Circle" << endl;};
    ...
};
```

Reprenons l'exemple précédent

ShapeTest.cpp

```
... // all required lines

int main()
{
Shape* s;
Circle* c = new Circle();

s = c;      // correct

s->print();

return 0;
}
```

Standard output

```
I am a Circle
```

Intérêt des méthodes virtuelles

- Les méthodes virtuelles permettent de réaliser des opérations génériques
- Elles permettent par exemple de manipuler des objets hétérogènes qui ont le même parent
- Exemple : on peut manipuler un tableau de (Shape*) contenant différentes formes (Circle, Square, Triangle, …).

Application

ShapeTest.cpp

```
... // all required lines

int main(){
    Shape* s = new Shape();
    Circle* c = new Circle();
    Rectangle* r = new Rectangle();

    Shape** array = new Shape*[3];

    array[0] = s;
    array[1] = c;
    array[2] = r;

    for(int i=0; i<size; i++)
        array[i]->print();

    return 0;
}
```

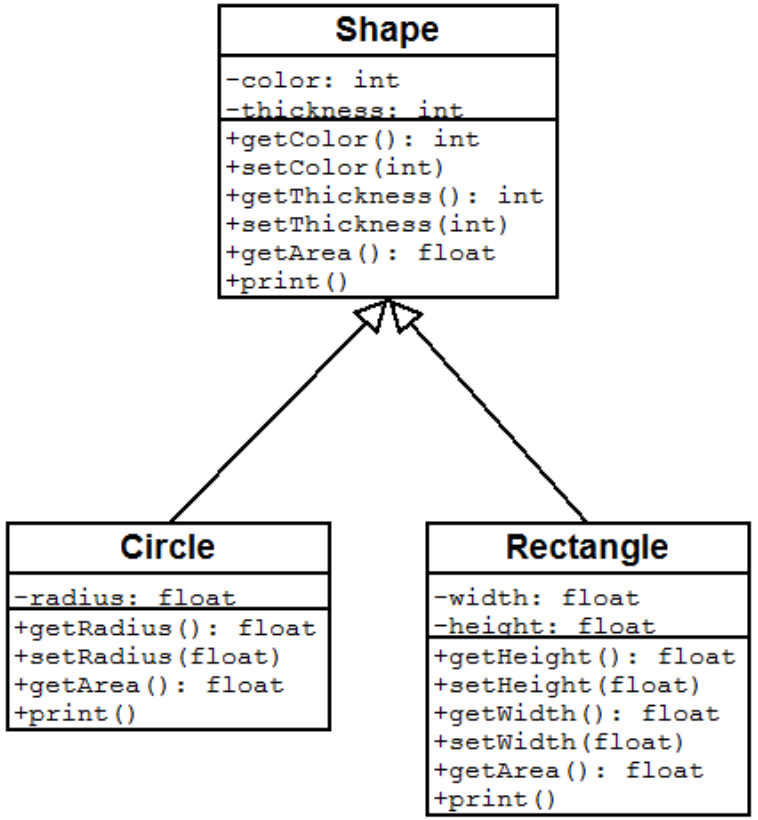
Standard output

```
I am a Shape
I am a Circle
I am a Rectangle
```

Les méthodes virtuelles pures

Principe

- Soit la méthode virtuelle getArea()
- Nous savons calculer l'aire du cercle:
  `area = PI * radius * radius`
- Nous savons calculer l'aire du rectangle:
  `area = width * height`
- Mais qu'en est-il de la surface d'une forme quelconque de la classe `Shape` ??

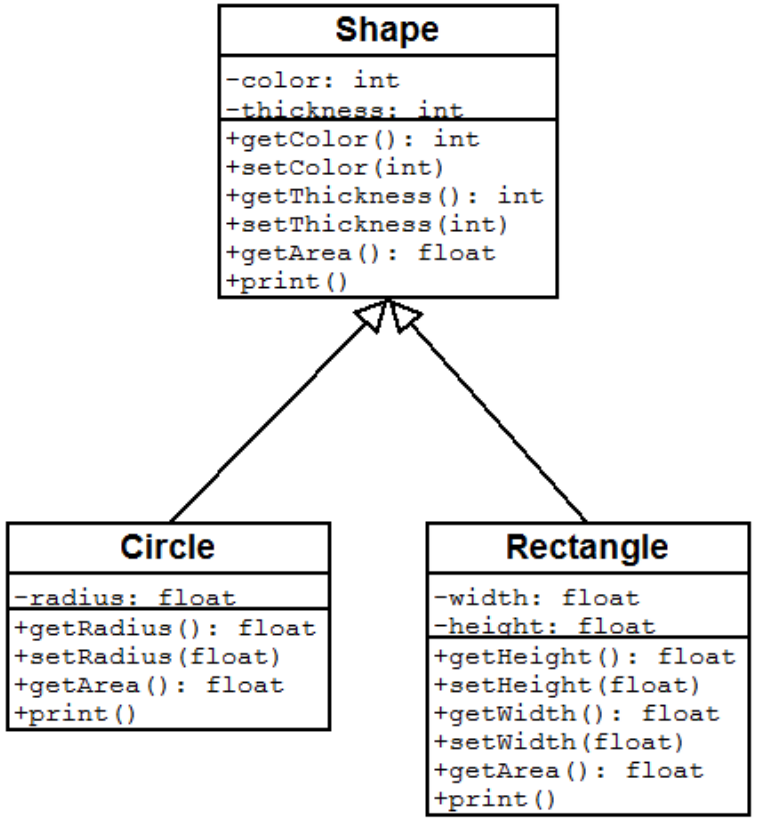


Principe

- Concrètement cela signifie que
  la méthode virtuelle Shape::getArea()
  n'a pas de code.

- Son rôle est juste de préparer les appels
  vers Circle::getArea() et Rectangle::getArea()

- Ce type de méthodes sont appelées
  méthode virtuelle **pures**



Syntaxe

Shape.h

```
... // all required lines

class Shape{
public:
virtual float getArea()=0;
};
```

Suite de l'exemple

Circle.h

```
... // all required lines

class Circle: public Shape{
...
private:
float radius;
...
public:
float getArea(){ return M_PI * radius * radius; };
// the macro M_PI is defined in math.h
};
```

Suite de l'exemple

Rectangle.h

```
... // all required lines

class Rectangle: public Shape{
...
private:
float height, width;
...
public:
float getArea(){ return height * width; };
};
```

Suite de l'exemple

ShapeTest.cpp

```
... // all required lines

int main()
{
    Circle* c = new Circle(1);
    Rectangle* r = new Rectangle(2,3);
    Shape** array = new Shape*[2];
    array[0] = c;
    array[1] = r;

    float total = 0.0;
    for(int i=0; i < 2; i++){
        cout << "Shape area: " << array[i]->getArea() << endl;
        total += array[i]->getArea();
    }

    cout << "Total area: " << total << endl;

    return 0;
}
```

Standard output

```
Shape area: 3.14159
Shape area: 6.00000
Total area: 9.14159
```

Les classes abstraites

- Toute classe contenant au moins une méthode virtuelle pure est appelée **classe abstraite**.

- Les classes abstraites **ne peuvent pas être instanciées** (elles sont incomplètes, certaines parties du code sont manquantes et seront définies dans les classes enfants)

dynamic_cast

Question

- Les méthodes virtuelles permettent de modéliser un comportement générique pour les méthodes communes de l'ensemble des classes enfants.
- Cependant les classes enfants peuvent avoir des méthodes spécifiques qu'elles ne partagent pas avec les autres classes enfants.
- Comment récupérer un pointeur vers une classe enfant à partir d'un pointeur de la classe parent (si possible !) pour pouvoir appeler ces méthodes spécifiques ??
- C'est le rôle de l'opérateur C++ : `dynamic_cast`

Exemple

ShapeTest.cpp

```
... // all required lines

int main()
{
   Circle* c = new Circle(1);
   Rectangle* r = new Rectangle(2,3);

   Shape** array = new Shape*[2];

   array[0] = c;
   array[1] = r;

   ShapeUtils::printRadiusIfAny(array, 2);

   return 0;
}
```

Exemple

ShapeUtils.cpp [Principle]

```
... // all required lines

void ShapeUtils::printRadiusIfAny(Shape** array, int size)
{
   Shape* s;

   for(int i=0; i<size; i++){
      s = array[i];

      if(??? "s is an instance of Circle") {
         cout << s->getRadius() << endl;
      }
   }
}
```

Exemple

ShapeUtils.cpp [Syntax]

```
... // all required lines

void ShapeUtils::printRadiusIfAny(Shape** array, int size)
{
   Circle* c;

   for(int i=0; i<size; i++){
      c = dynamic_cast<Circle*>(array[i]);

      if(c != NULL) {
         cout << c->getRadius() << endl;
      }
   }
}
```

Si array[i] est une instance de Circle, alors le dynamic_cast renvoie un pointeur de type Circle sinon il renvoie NULL

Exemple : écriture équivalente

ShapeUtils.cpp [Syntax]

```
... // all required lines

void ShapeUtils::printRadiusIfAny(Shape** array, int size)
{
   Circle* c;

   for(int i=0; i<size; i++){
      c = dynamic_cast<Circle*>(array[i]);
      // the NULL pointer is 0, interpreted as false
      if(c) {
         cout << c->getRadius() << endl;
      }
   }
}
```

Si array[i] est une instance de Circle, alors le dynamic_cast renvoie un pointeur de type Circle sinon il renvoie NULL

Exemple : une autre écriture équivalente

ShapeUtils.cpp [Syntax]

```
... // all required lines

void ShapeUtils::printRadiusIfAny(Shape** array, int size)
{
   Circle* c;

   for(int i=0; i<size; i++){

      // the = operator returns the right value
      if(c = dynamic_cast<Circle*>(array[i])) {
         cout << c->getRadius() << endl;
      }
   }
}
```

Si array[i] est une instance de Circle, alors le dynamic_cast renvoie un pointeur de type Circle sinon il renvoie NULL