CS217 – Object Oriented Programming (OOP)

Week – 09

April 05-09, 2021

Instructor: **Basit Ali**

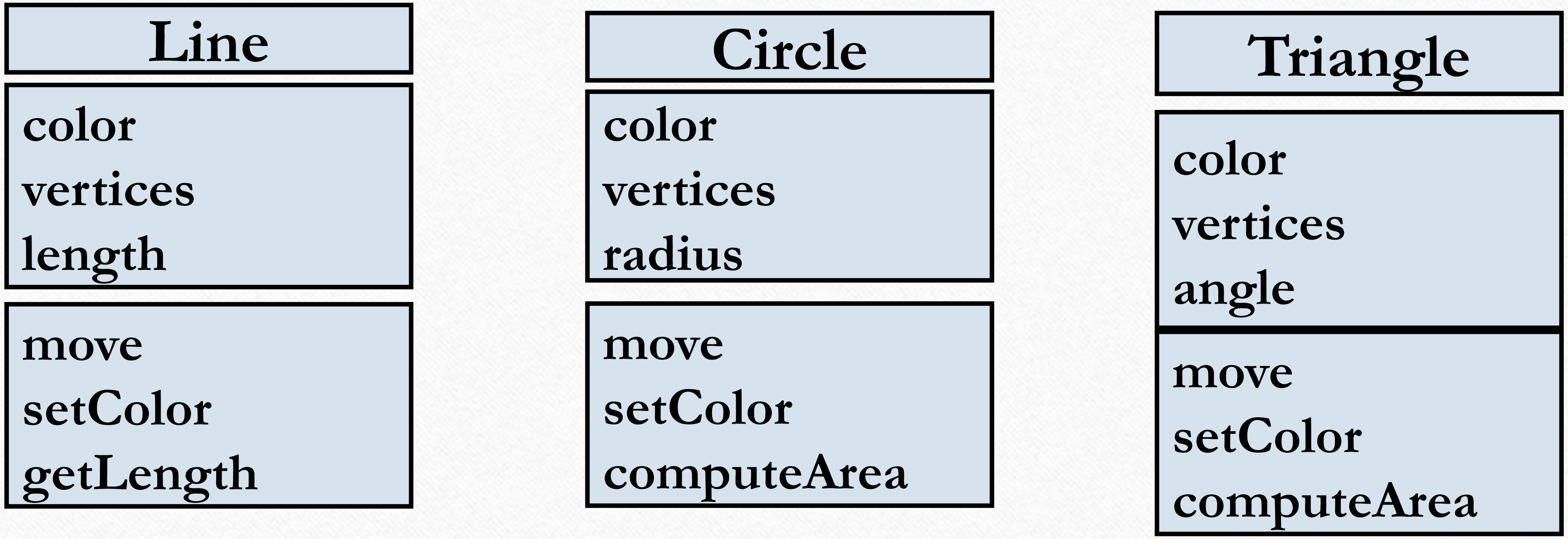
Recap – Inheritance

- Derived class inherits all the characteristics of the base class
- Besides inherited characteristics, derived class may have its own unique characteristics
- Major benefit of inheritance is reuse

Generalization

- In OO models, some classes may have common characteristics
- We extract these features into a new class and inherit original classes from this new class
- This concept is known as Generalization

Example – Generalization



Line

color
vertices
length

move
setColor
getLength

Circle

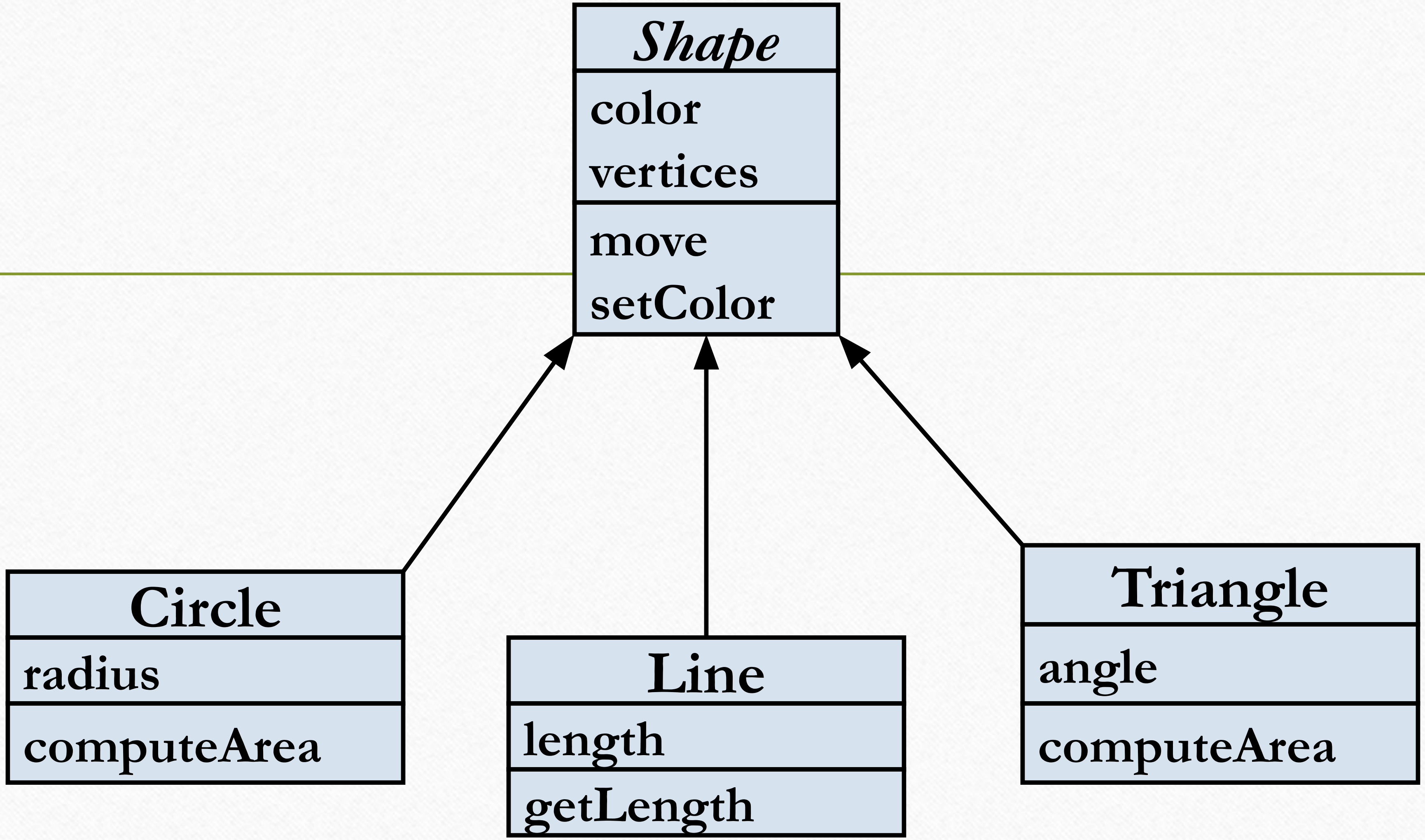
color
vertices
radius

move
setColor
computeArea

Triangle

color
vertices
angle

move
setColor
computeArea



Shape
color
vertices
move
setColor

Circle
radius
computeArea

Line
length
getLength

Triangle
angle
computeArea

Polymorphism

- The process of representing one Form in multiple forms is known as **Polymorphism**
- Polymorphism is derived from 2 Greek words: **poly** and morphs. The word "poly" means many and **morphs** means forms. So polymorphism means many forms.

Real life example of Polymorphism

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person have different-different behaviors.

In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

At Home behave like Son

Type of Polymorphism

- Static / Compile time polymorphism

- Dynamic / Run time polymorphism

Static / Compile time polymorphism

- It is also called Early Binding
- It happens where more than one methods share the same name with different parameters or signature and different return type.
- It is **known** as Early Binding because the **compiler** is aware of the functions with same name and also which overloaded function is to be **called** is **known** at **compile time.**

Function/Method Overloading

- Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**

Static / Compile time polymorphism

- **Overloading**
  - Function Overloading (ALREADY DISCUSSED)
  - Constructor Overloading (ALREADY DISCUSSED)
  - Operator Overloading (TO BE DISCUSSED)

Dynamic / Run time polymorphism

- This refers to the entity which changes its form depending on circumstances at runtime. This concept can be adopted as analogous to a chameleon changing its color at the sight of an approaching object.
- Method Overriding uses runtime Polymorphism.
- It is also called Late Binding.

Dynamic / Run time polymorphism

- Runtime Polymorphism is done using virtual and inheritance.
- When overriding a method, the behavior of the method is changed for the derived class.

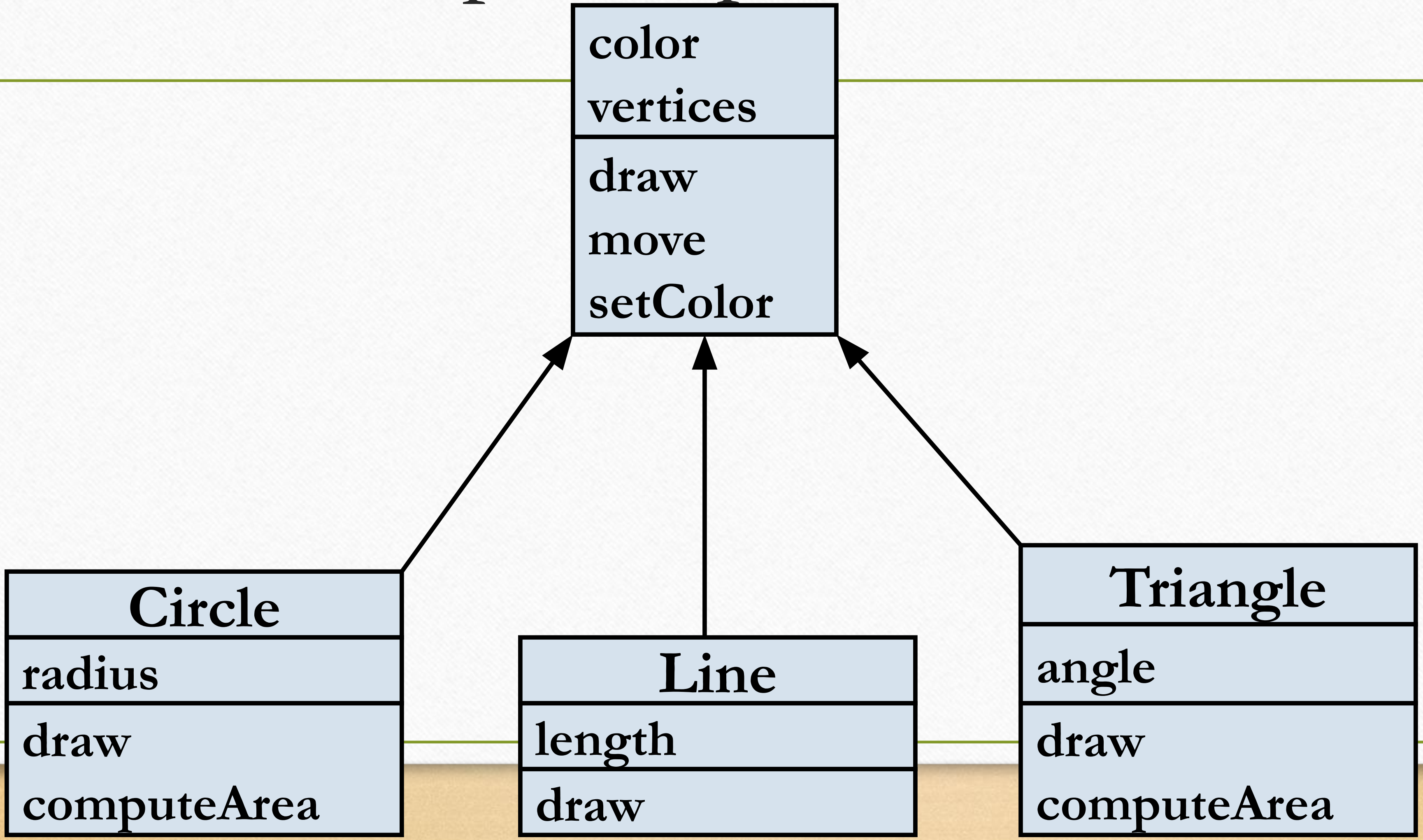
Overriding

- A class may need to override the default behavior provided by its base class
- Reasons for overriding
    - Provide behavior specific to a derived class
    - Extend the default behavior
    - Restrict the default behavior
    - Improve performance

Function/Method Overriding

- Define any method in both base class and derived class with same name, same parameters or signature, this concept is known as **method overriding**

Example – Specific Behavior



color
vertices

draw
move
setColor

Circle

radius

draw
computeArea

Line

length

draw

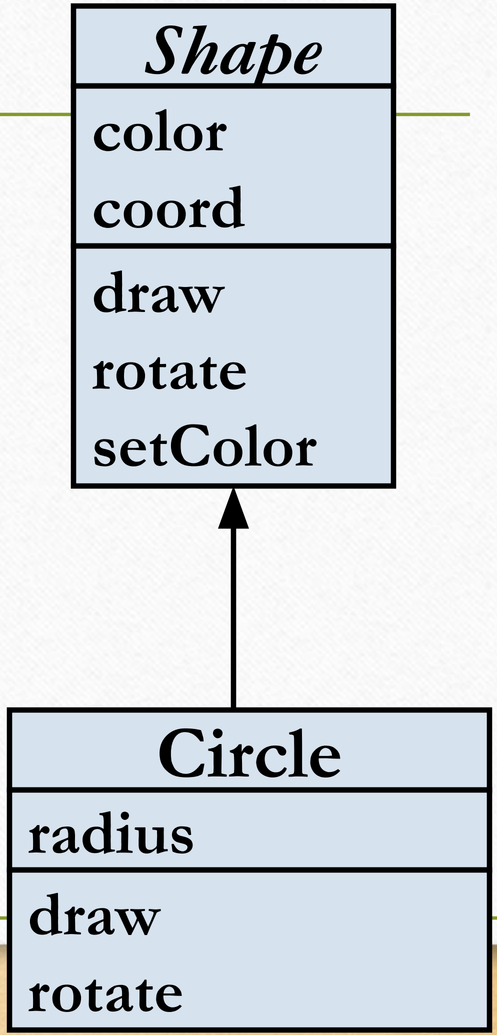
Triangle

angle

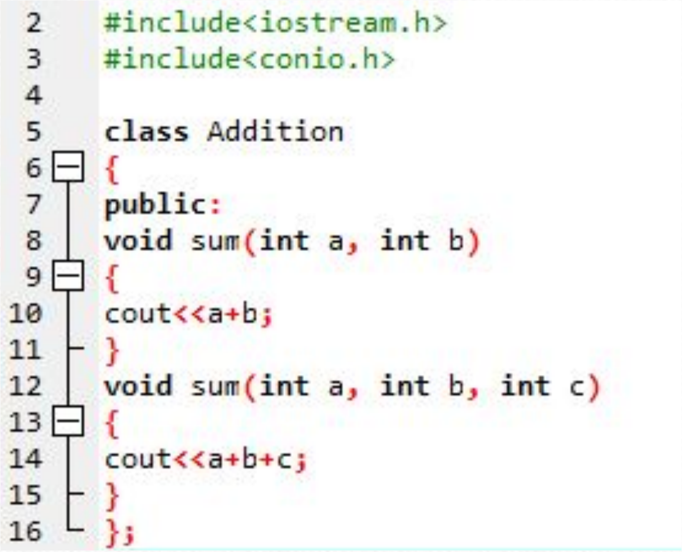
draw
computeArea

Example – Improve Performance

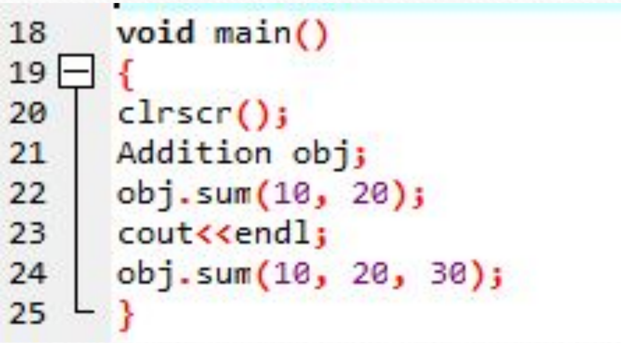
- Class Circle overrides *rotate* operation of class Shape with a Null operation.



Example



```
#include<iostream.h>
#include<conio.h>

class Addition
{
public:
void sum(int a, int b)
{
cout<<a+b;
}
void sum(int a, int b, int c)
{
cout<<a+b+c;
}
};
```

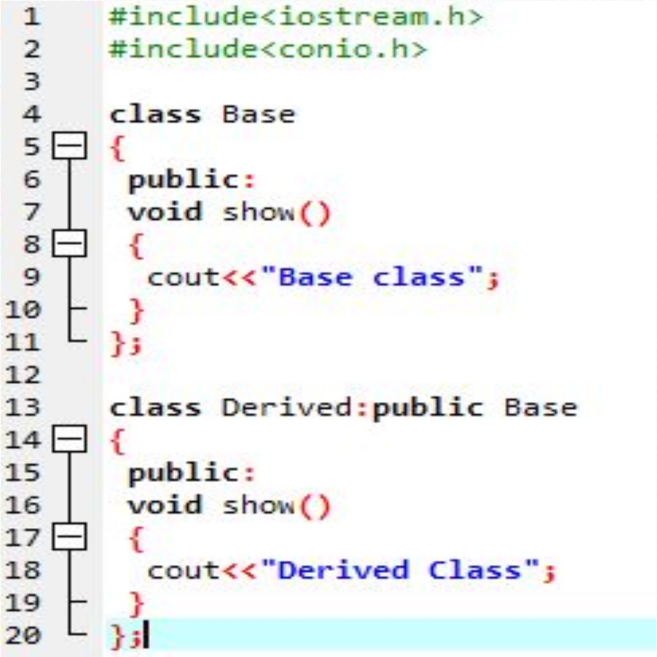


```
void main()
{
clrscr();
Addition obj;
obj.sum(10, 20);
cout<<endl;
obj.sum(10, 20, 30);
}
```

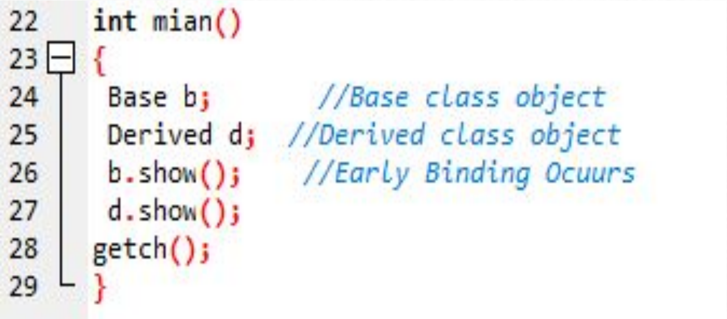
Output:

```
30
60
```

Example



```
#include<iostream.h>
#include<conio.h>

class Base
{
 public:
 void show()
 {
  cout<<"Base class";
 }
};

class Derived:public Base
{
 public:
 void show()
 {
  cout<<"Derived Class";
 }
};
```



```
int mian()
{
 Base b;        //Base class object
 Derived d;   //Derived class object
 b.show();      //Early Binding Ocuurs
 d.show();
getch();
}
```

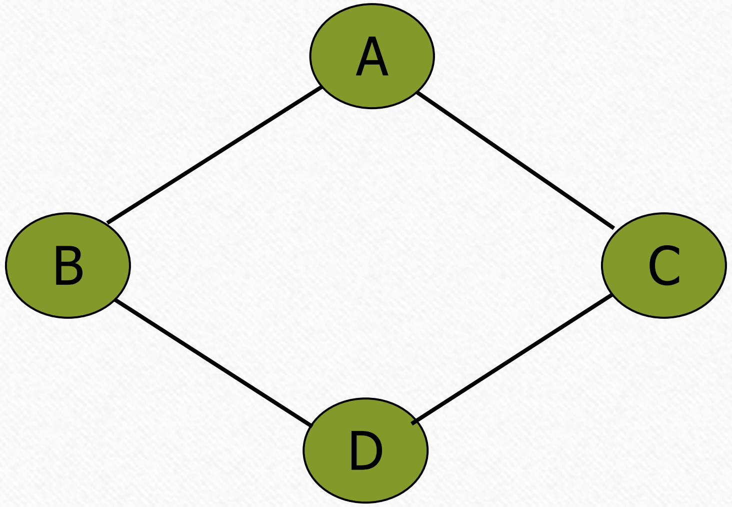
Output:

Base class

Derived class

Hybrid Inheritance: Potential problem

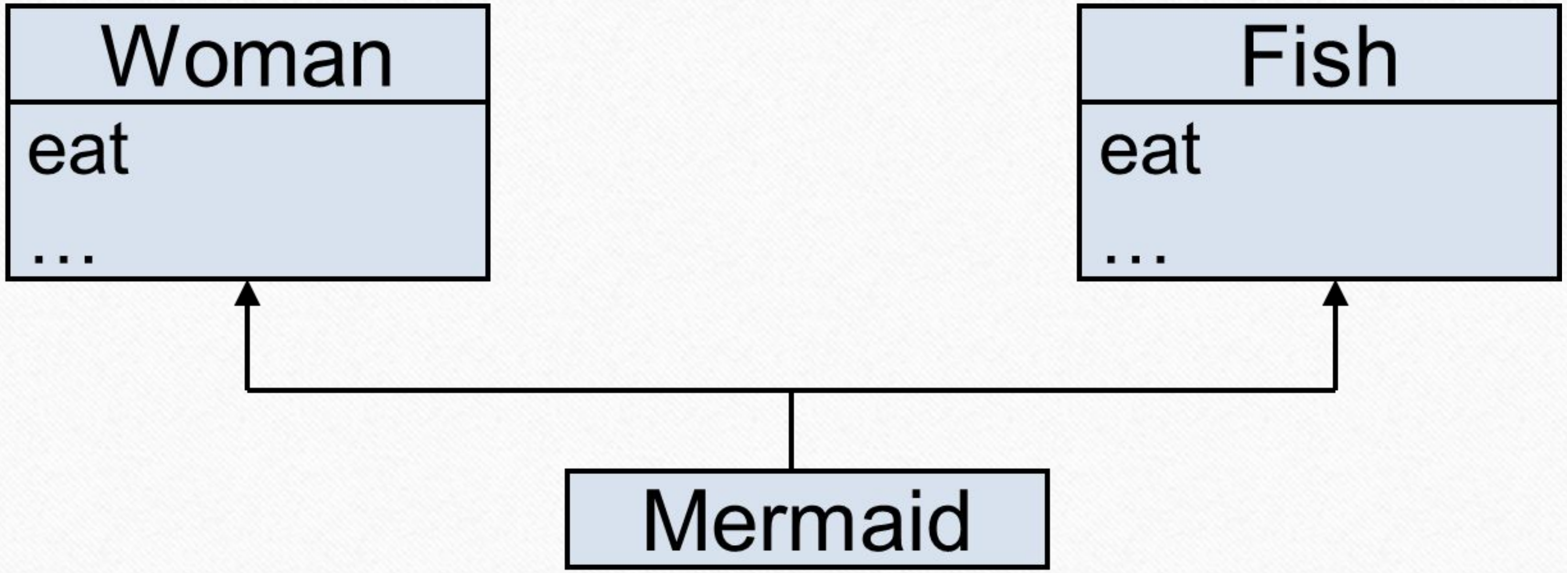
- common dangerous pattern: "The Diamond"
  - Classes B and C extend A
  - Class D extends A and B
  - Class D extends A and C



Problems with Hybrid Inheritance

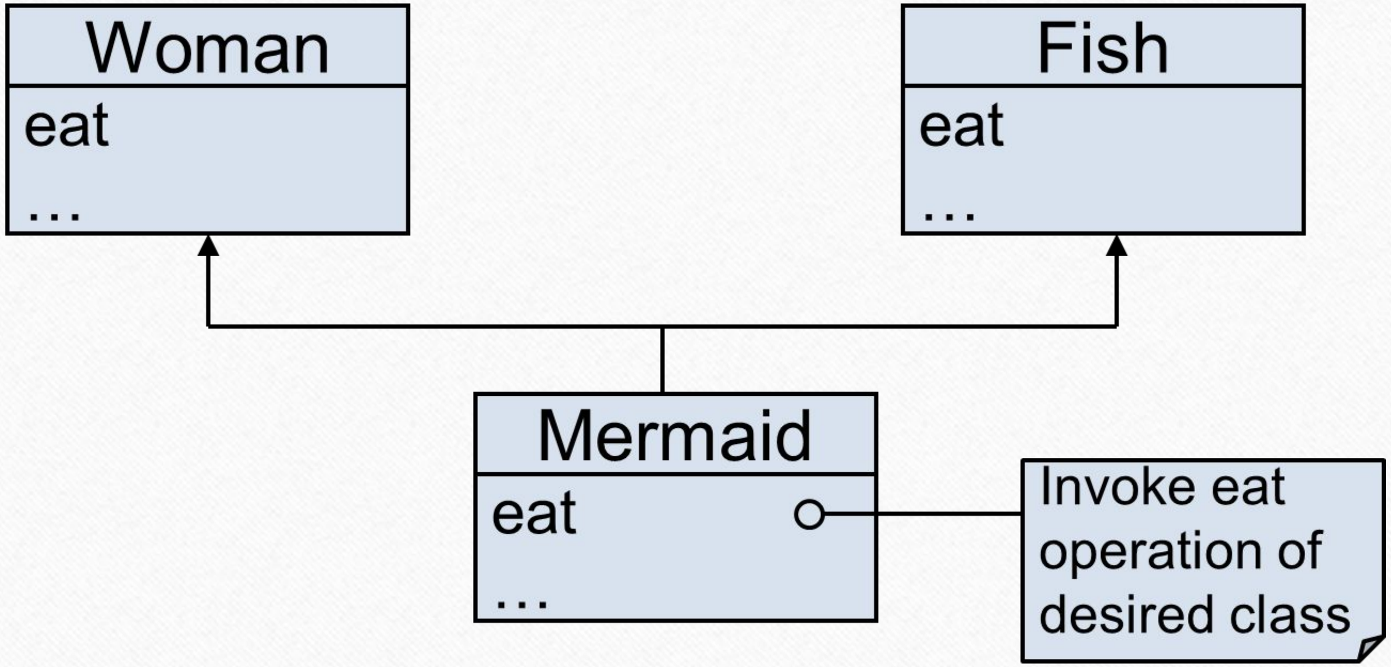
- Increased complexity
- Reduced understanding
- Duplicate features

Problem – Duplicate Features

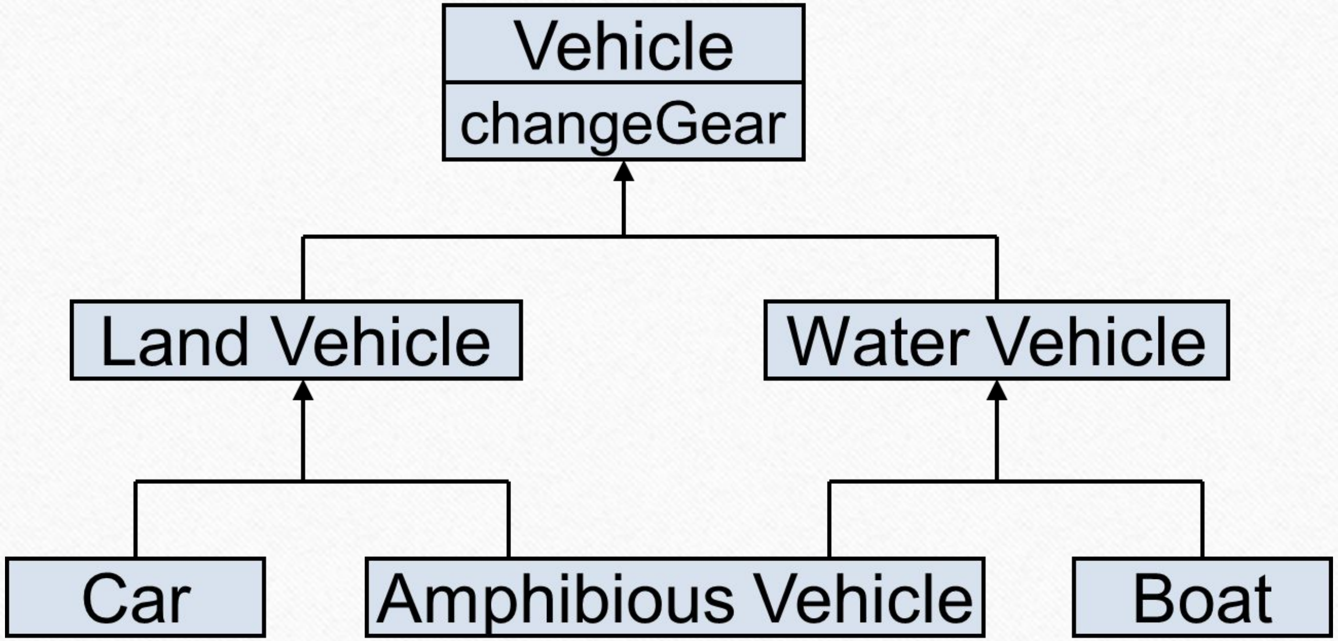


Which *eat* operation *Mermaid* inherits?

Solution – Override the Common Feature



Problem – Duplicate Features (Diamond Problem)



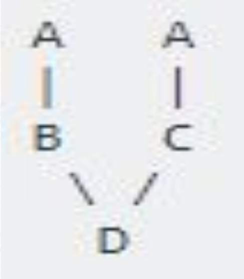
Which *changeGear* operation Amphibious Vehicle inherits?

Solution to Diamond Problem

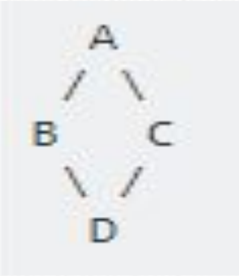
- Some languages disallow diamond hierarchy
- Others provide mechanism to ignore characteristics from one side

Solution to Diamond Problem (Virtual Inheritance)

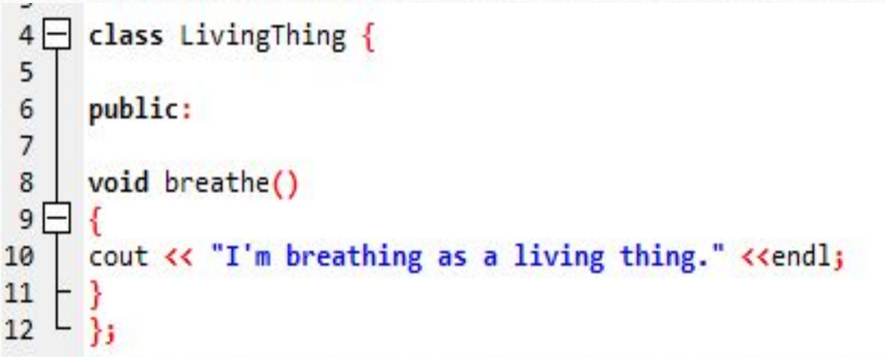
- What happens without virtual inheritance:



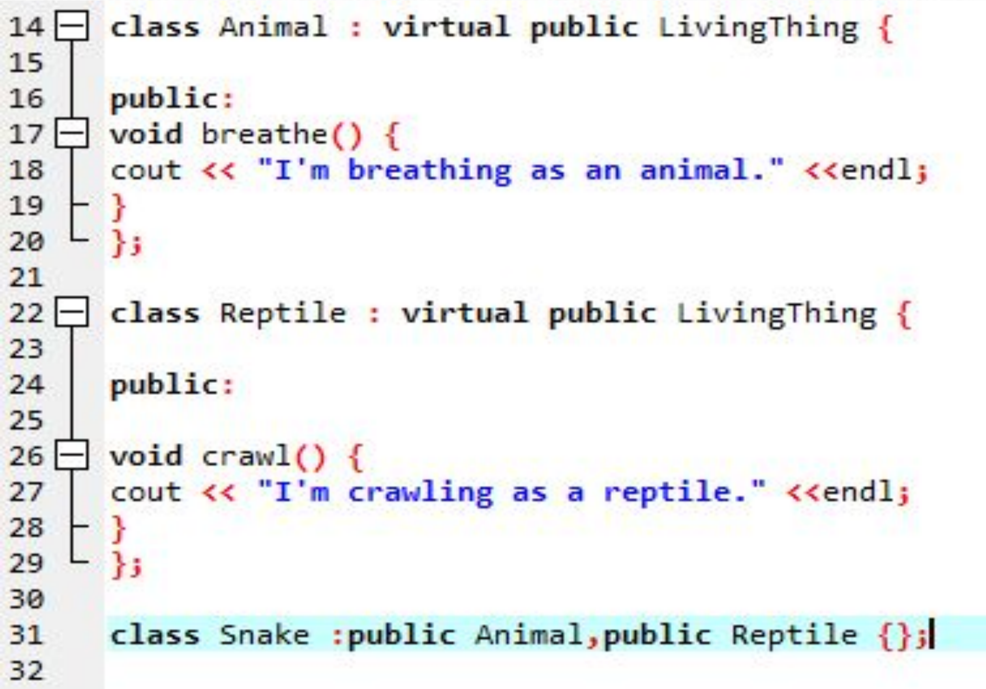
- You want: (Achievable with virtual inheritance)



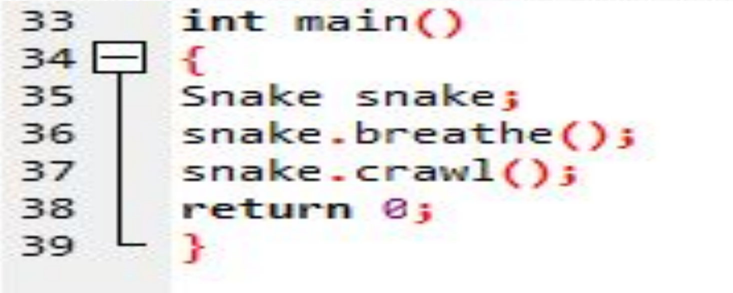
Solution..



```
class LivingThing {

public:

void breathe()
{
cout << "I'm breathing as a living thing." <<endl;
}
};
```

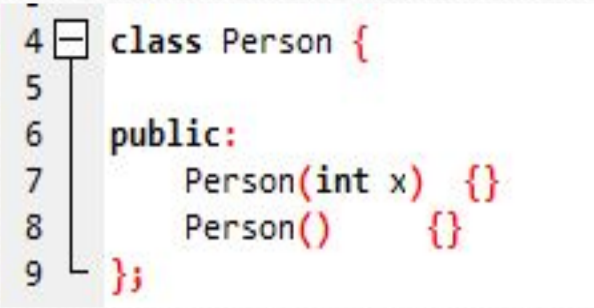


```
class Animal : virtual public LivingThing {

public:
void breathe() {
cout << "I'm breathing as an animal." <<endl;
}
};

class Reptile : virtual public LivingThing {

public:

void crawl() {
cout << "I'm crawling as a reptile." <<endl;
}
};

class Snake :public Animal,public Reptile {};

```

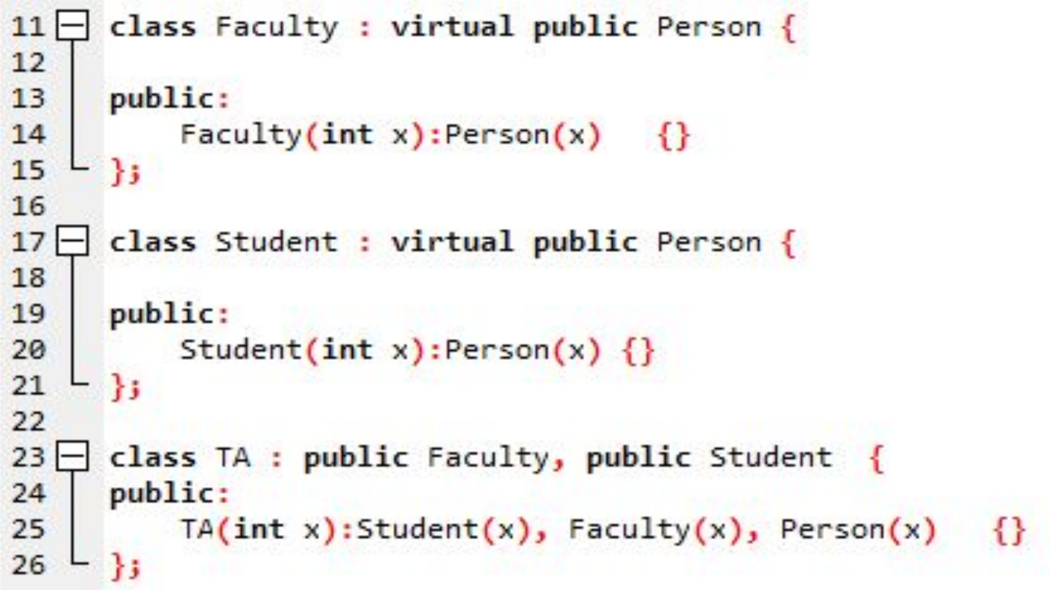


```
int main()
{
Snake snake;
snake.breathe();
snake.crawl();
return 0;
}
```

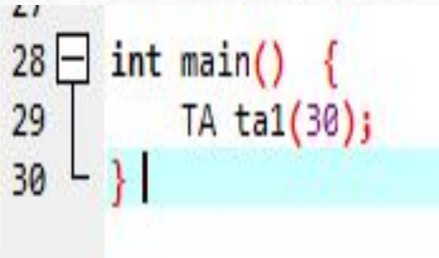
Diamond Problem Solution (With constructor)



```
class Person {

public:
    Person(int x)  {}
    Person()     {}
};
```



```
class Faculty : virtual public Person {

public:
    Faculty(int x):Person(x)   {}
};

class Student : virtual public Person {

public:
    Student(int x):Person(x) {}
};

class TA : public Faculty, public Student  {
public:
    TA(int x):Student(x), Faculty(x), Person(x)   {}
};
```



```
int main()  {
    TA ta1(30);
}
```