

ENSF 480
Principles of Software Design
(Review of Basics - Part II)

Relationships Among Classes (Quick Review)

Relationship among classes (Review)

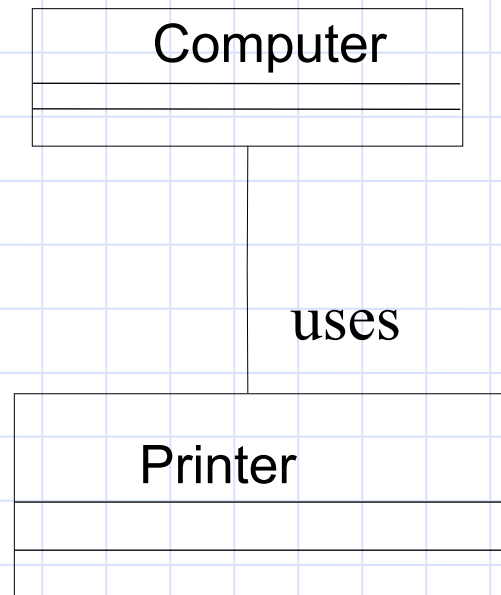
- There are four major type of relationships among classes that most of O.O. programming languages support them:
 - Association
 - Aggregation
 - Composition
 - Inheritance

Association (Review)

- The association relationship expresses a semantic connection between classes:
 - There is no hierarchy.
- It is a relationship where two classes are weakly connected; i.e. they are merely “associates.”
 - All object have their own lifecycle;
 - There is no ownership.
 - There is no whole-part relationship.

Association (Review)

- The association of two classes must be labeled.
- To improve the readability of diagrams, associations may be labeled in active or passive voice.



Implementation of Simple Association in Java (Quick Review)

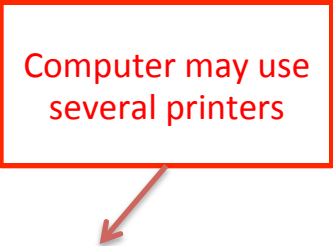
Example

- Each computer May use several printers:

```
class Computer
```

```
{  
    String message;  
  
    public use(Printer [] printers)  
    {  
        int i;  
        for(i = 0; i < printers.length; i++)  
            printers[i].print(message);  
    }  
  
    // rest of class  
    ...  
}
```

Computer may use
several printers

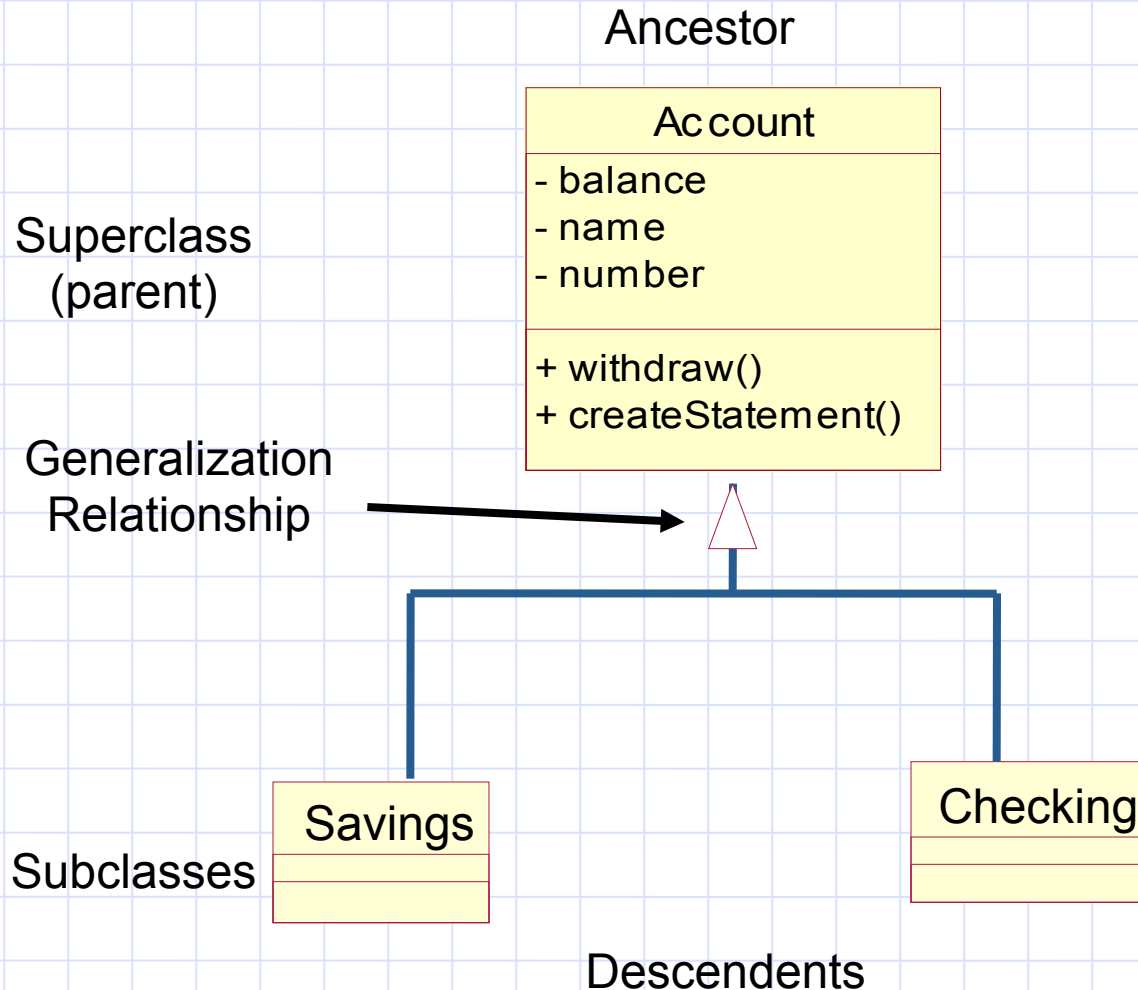


```
class Printer
```

```
    private String type  
    public print(String message)  
    {  
        println(message);  
    }  
  
    // rest of class  
    ...  
}
```

Generalization/Specialization Relationship Among Classes

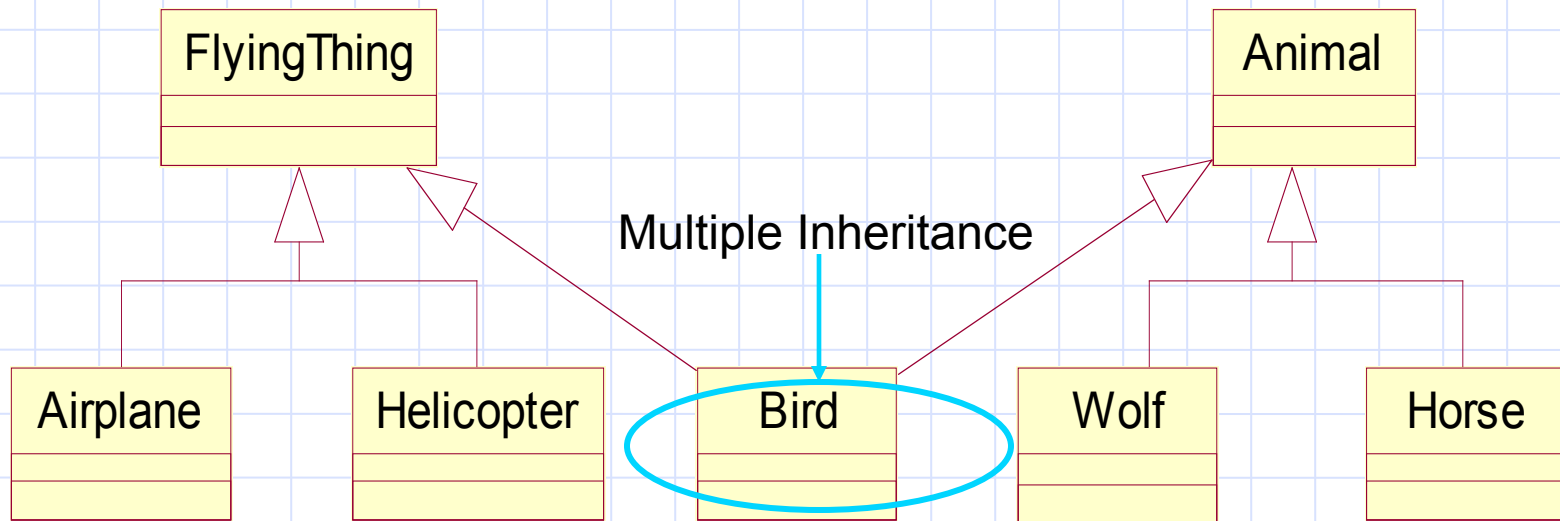
Class Relationship - Single Inheritance



The generalization is drawn from the subclass class to the superclass/parent class.

The terms “ancestor” and “descendent” may be used instead of “superclass” and “subclass.”

Example: Multiple Inheritance



Use multiple inheritance only when needed and always with caution!

Implementation of Inheritance in Java (Quick Review)

Base Class Design in Java

- Syntax for defining a base class is the same as an ordinary class with two exceptions:
 - Members intended to be inherited but not intended to be public are declared as *protected* members.
- Member functions whose implementation depends on representational details of subsequent derivations that are unknown at the time of the base class design are declared as *abstract*.

Example of Single Inheritance in Java

```
class Animal
{
    protected double weight;
    public Animal(double w)
    {
        weight = w;
    }
    public abstract void move();
}
```

```
class Cat extends Animal
{
    private color;
    public Cat(double w, int
c)
    {
        super(w);
        color = c
    }
    public void move() {
        // more code
    }
};
```

Implementation of Inheritance in C++ (Quick Review)

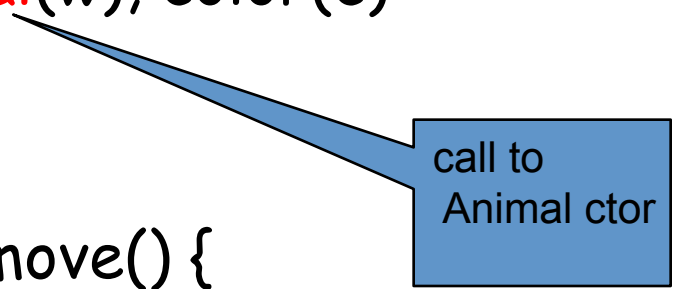
Base Class Design in C++

- Syntax for defining a base class is the same as an ordinary class with two exceptions:
 - Members intended to be inherited but not intended to be public are declared as *protected* members.
- Member functions whose implementation depends on representational details of subsequent derivations that are unknown at the time of the base class design are declared as *virtual*.

Example of Inheritance in C++

```
class Animal
{
    protected:
    double weight;
    public:
    Animal(double w)
    {
        weight = w;
    }
    virtual void move() =0;
    virtual void display() {
        cout << " Animal";
    }
}
```

```
class Cat: public Animal
{
    protected:
    int color;
    public Cat(double w, int c):
        Animal(w), color(c)
    {
    }
    void move() {
        // more code
    }
};
```



call to
Animal ctor

- Discuss differences between two virtual functions move and display

Virtual functions

- A virtual function is a special function invoked through a public base class reference or pointer; it is bound dynamically at run time.
- The class that declares a function as virtual must provide a definition for the function or declare it as pure virtual:
 - If definition is provided, serves as default instance for subsequent derived classes.
 - If pure virtual is declared, the class will be considered as abstract class. Means instances of that class cannot be created. The derived class from an abstract base class can define the function or will be also considered as an abstract class.

Virtual Functions

- The redefinition of a virtual function must match exactly the name, signature and the return type of the base class instance.
- Use of keyword virtual is optional.
- The virtual mechanism is handled implicitly by compiler.
- If redefinition of a virtual function does not match exactly, the function is not handled as virtual. However the subsequent class still can redefine a virtual function.

Abstract and Concrete Classes (Quick Review)

Abstract vs. Concrete Classes

- Example of abstract class in Java

```
class Shape {  
    public Shape() {  
        // more code  
    }  
  
    abstract double area();  
    // rest of code  
}
```

- Example of abstract class in C++

```
class Shape {  
    public:  
        Shape();  
        virtual double area() = 0;  
        // rest of code  
};
```

- Function area is called a **pure virtual function**

- Questions:

- What is a “concrete” class and What are the differences between abstract and concrete classes?

More on abstract classes in Java

- What is the other way to have an abstract class in Java?

- Java interface

```
interface Calculable  
{  
    double area();  
    double perimeter();  
}
```

- Java

```
class Shape implements Calculable {  
    public:  
        Shape() { // more code }  
  
        // rest of code  
}
```

- If Shape doesn't implement area and perimeter, it remains as an abstract class

Inheriting operators, destructor, and constructors

- A derived class inherits all the member functions of each of its base classes except:
 - the constructors (including copy constructor),
 - destructor, and
 - assignment operators of each of its base classes.

Should Destructor Be Virtual?

```
class A
{
    char * s1;
public:
    A(int n) { s1 = new char[n];}
    ~A() { delete [] s1}
};
```

```
class B : public A
{
    char * s2;
public:
    B(int n, int m): A(n) { s2 = new char[m];}
    ~A() { delete [] s2}
};
```

```
int main(void) {
    A * p = new B(5, 5);
    delete p;
}
```

Virtual Destructor

- A destructor should be usually declared virtual if it is responsible to remove an allocated memory.
 - This is to avoid undefined behavior or a memory leak
 - It reminds the inheriting classes to redefine this function to do their own cleanup.

```
class A
{
    char * s1;
public:
    A(int n) { s1 = new char[n];}
    virtual ~A() { delete [] s1;}
};
```

```
class B : public A {
    char * s2;
public:
    B(int n, int m): A(n) { s2 = new
                           char[m];
    }
    ~B() { delete [] s2; }
};
```

```
int main(void) {
    A * p = new B(5, 6);
    delete p;
}
```


Derived Class Copy Constructor

- If a derived class that explicitly defines its own copy constructor, the definition completely overrides the default definition, and is responsible for copying its base class component:

```
class Base { ...};
```

```
class Derived: public Base {  
public  
    Derived (const Derived& x): Base(x) { }  
};
```

Derived Class Assignment Operator

- If a derived class that explicitly defines its own assignment operator, the definition completely overrides the default definition, and is responsible for copying its base class component:

```
Derived& Derived::operator = (const Derived& rhs) {  
    Base::operator = (rhs);  
    // clean up the old values of derived part if necessary  
    // assign members from derived part  
}
```