Relationships Among Classes

Relationship among classes

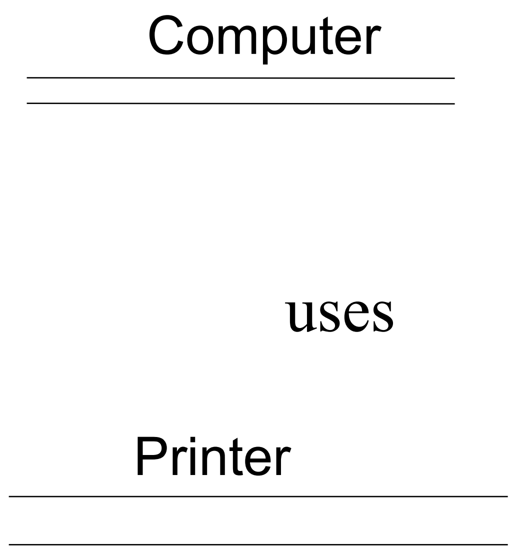
- There are three major type of relationships among classes that most of O.O. programming languages support them:
  - Association
  - Aggregation/Composition
  - Inheritance

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.
- In another words, if inheritance and aggregation/composition doesn't apply, it is most likely a simple association

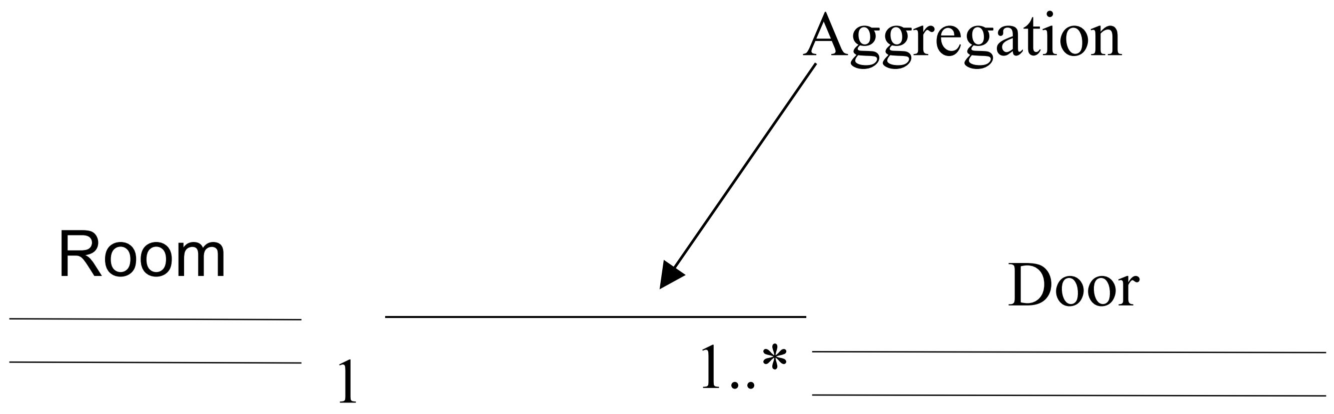
Association (Review)

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



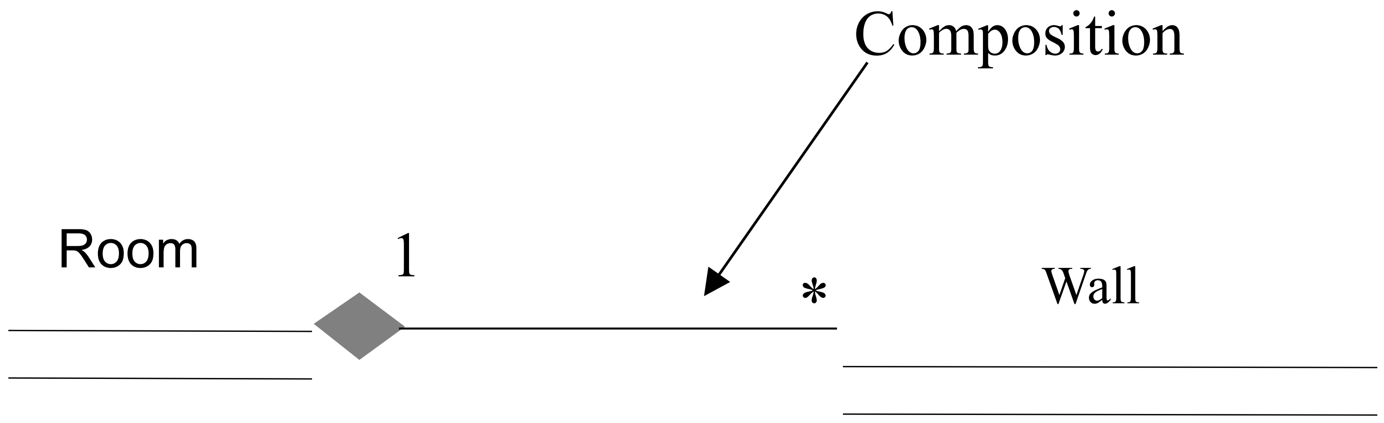
Aggregation (Review)

- An aggregation represents an asymmetric association, in which one of the ends plays a more important role than the other one.
- The following criteria imply an aggregation:
  - A class is part of another
  - The objects of one class are subordinates of the objects of another class
- Denotes a whole/part hierarchy with the ability to navigate from whole (aggregate) to its parts (attributes).
- The part is normally being referenced to either a pointer or by a reference in C++



Aggregation (Review)

- A strong type of aggregation, when deletion of the whole causes the deletion of the part, is called ***composition.***
- The "part" objects are usually created in the constructor of the container class.



Composition and Aggregation Differences in C++:

Consider the following definition of class Engine:

```
class Engine {
   private:
       int size;
   public:
       Engine (int size) // AN INLINE DEFINITION OF CTOR
           {
          this ->size = size;
       }
      ...
    // ASSUME MORE METHODS HERE
};
```

Now let's assume we need to design a class in C++, called Car to contain an object of class Engine.

- How can we implement Composition?
- How can we implement Aggregation?

Composition Solution 1: Using static object of Engine in class Car

```
class Engine {
   private:
        int size;
   public:
        Engine (int size) // AN INLINE DEFINITION OF CTOR
           {
           this ->size = size;
        }
      ...
     // ASSUME MORE METHODS HERE
};

class Car {
   private:
        Engine engine;
   public:
        Car (int size): engine(size) // AN INLINE DEFINITION OF CTOR
        {

        }
      ...
     // ASSUME MORE METHODS HERE
};
```

This is a call to Engine ctor

Composition Solution 2: Using dynamic memory allocation for object engine in class Car

```
class Engine {
   private:
        int size;
   public:
        Engine (int size) // AN INLINE DEFINITION OF CTOR
           {
            this ->size = size;
        }
      ...
     // ASSUME MORE METHODS HERE
};

class Car {
   private:
        Engine* engine;
   public:
        Car (int size) {
            engine = new Engine(size);
        }
      ...
     // ASSUME MORE METHODS HERE
};
```

This is a call to Engine ctor

Aggregation Solution:

```
class Engine {
   private:
        int size;
   public:
        Engine (int size)
          {
           this ->size = size;
        }
      ...
    // ASSUME MORE METHODS HERE
};

class Car {
   private:
        Engine* engine;
   public:
        Car (Engine* engine) {
           this -> engine = engine;

        }
      ...
    // ASSUME MORE METHODS HERE
};
```

No call to Engine ctor

Notice that lifetime of Object engine is not depending on object of class Car

```
int main() {
   Engine x(300);
   Car y(&x);
   ...
   return 0;
}
```

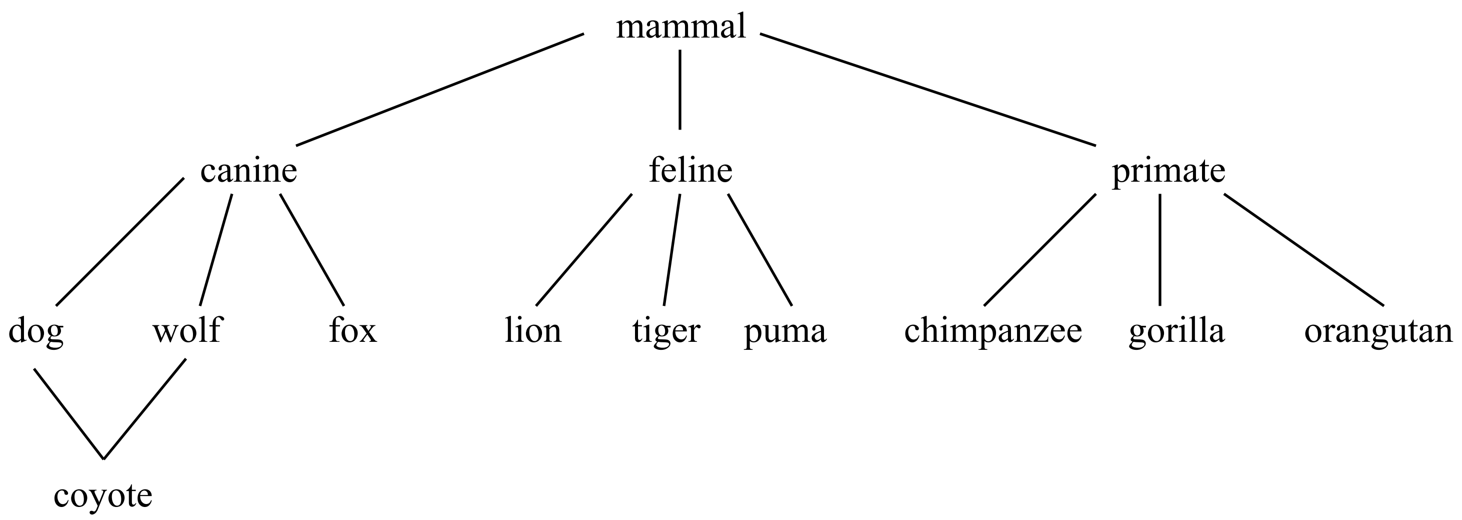
Derivation/Inheritance

Inheritance

- Inheritance is a relationship among classes where a subclass inherits the structure and behavior of its super-class.
  - Defines the “is a” or generalization/specialization hierarchy.
  - Structure: instance variables.
  - Behavior: instance methods.

Inheritance in C++

- C++ supports single and multiple inheritance



Class Derivation (Inheritance)

- In order to derive a class, the following two extensions to the class syntax are necessary
  - class heading is modified to allow a derivation list of classes from which to inherit members.
  - An additional class level, that of ***protected***, is provided. A protected class member behaves as a public member to a derived class

```
class Cat : public Animal
{
   protected:

   // data members

};
```

Class derivation - Example

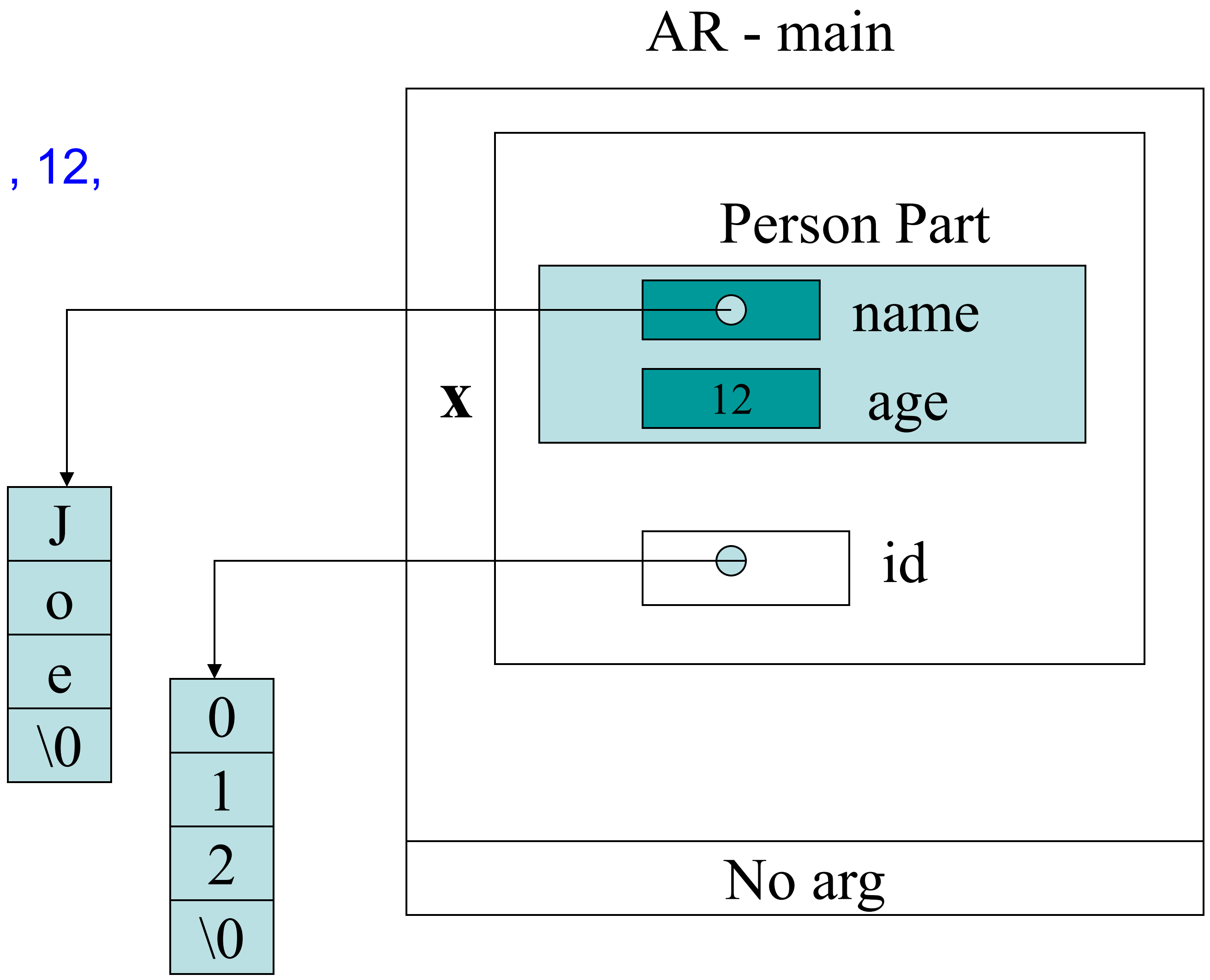
```
class Person {
public:
  Person(char* n, int n)
  …
Protected:
  int age;
   char *name;
};
```

←

```
class Student: public Person
{
      public:
      Student(char* n, int a, char* i);
      …
      protected:
      char *id;
};
```

Example Continued

```
int main ()
{
   Student  x (  “Joe”  , 12,
      “012”  );
   return 0;
}
```



Base Class Design

- 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 functions*.

Base Class Design (Continued)

```
class Person {
    public:
      Person();
      virtual ~Person();
      virtual display();
      …
    protected:
      int age;
      char *name;
};
```

Inherited member access

- The derived class member functions can have access to inherited members directly or by using the the scope resolution operator:

```
void Student :: display() {
    cout << Person::name << age;
    }
```

In this example name also could be accessed directly without using scope operator.

Inherited Member Access (Continued)

- In most cases, use of the class scope operator is redundant. In two cases, however, this additional aid is necessary:
  - When an inherited member’s name is reused in the derived class.
  - When two or more base classes define an inherited member with the same name.

Base Class Initialization

- Member initialization list is used to pass arguments to a base class constructor. The tag name of a base class is specified, followed by its argument list enclosed in parentheses.

```
class A {
int a;
public:
A(int x) {a = x;}
};
```

```
class B: public A{
      int b;
      public:
      B(int x, int y) : A(x){
      b = y;
       }
};
```

call to ctor

Special Relationship between Base and Derived Class

- A derived class can be assigned to any of its public base classes without requiring an explicit cast.

  For example, consider class Student is derived from class Person and class Monitor is derived from class Student :

```
Person x;
Student y;
Monitor z;
x = y;                  // OK
y =  (Student) x;                 // Needs cast
x = z;                  // OK
```

What is a ‘virtual Function’ & When Do Need It?

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 instance invoked is determined by the class type of the actual object addressed by the pointer or reference.
- Resolution of a virtual function is transparent to the user

Virtual functions

- A virtual function is specified by prefacing a function declaration with keyword virtual.
- 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

E.g.

```
class Animals
{
  char name[20];
  public:
  virtual display() = 0;
};
```

```
class Fish :public Animal
{
  char color[20];
  public:
  display() {
   cout << “This is a fish.”;
  }
}
```

```
class Cat :public Animal
{
  char color[20];
  public:
   display() {
     cout << “This is a cat.”;
    }
}
```

Virtual functions

- If class Fish needs to have an object, but you still don't want to define function *display(),* you can define a null instance of display function;

```
class Fish: public Animal{
  char color[20];
  …
  public:
  display() {} // null function
}
```

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 return type and signature, it is not handled as virtual. However, the subsequent class still can redefine a virtual function.

A Good Reason to Declare a Destructor, Virtual

Why Virtual Destructor?

- Let’s have a look at the following example:

```
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];}
       ~B() { delete [] s2;}
};
```

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

Class Discussion:

- **What happens when we delete p, considering that pointer p is an A type?**

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;}
};
```

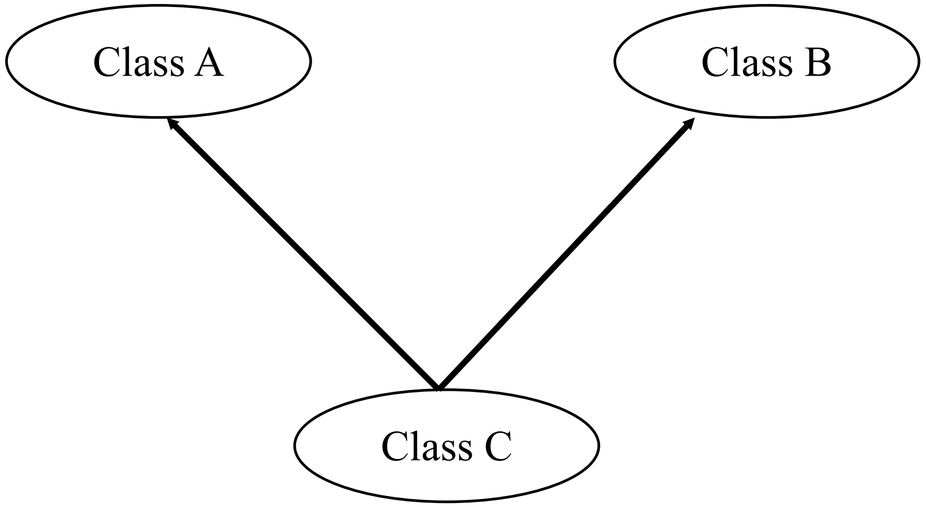
```
class B : public A
{
   char * s2;
    public:
        B(int n, int m):  A(n)  { s2 = new char[m];}
       ~B() { delete [] s2;}
};
```

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

What is Multiple Inheritance?

Multiple Inheritance

- A class can be derived from more than one parent



```
class C : pubic A, public B
{
      ....
}
```

Order of Construction

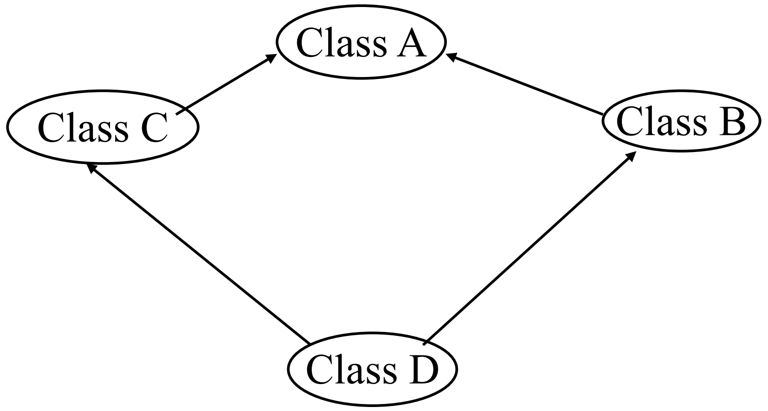
- The constructor member initialization list controls only the values that are used to initialize the base classes, not the order in which the base classes are constructed:
- The base class constructors are called in the order in which they appear in the class derivation list:

```
class C : public B, public A {
            ....
}
```

- In this example the constructor of class B is called first, and then the constructor of A.

Multiple Inheritance

- Although a base class may legally appear only once in a derivation list, a base class data member can appear multiple times within a derivation hierarchy that form a diamond shape as show, below. This can cause a member ambiguity.
- Why? (a class discussion)



Possible Ambiguity Error

```
class A
{
   private:
     int a;
};

class B: public A
{
 private:
     int b;
};

class C: public A
{
 private:
     int c;
}
```

```
class D: public  B, public C
{
    …
};
```

```
int main() {
   D myd;
   // trying to use a in class D,
   // gives ambiguity error. Why?
   return 0;
}
```

Possible Ambiguity Error (Continued)

- In the previous example class D will contain two copies of each member of class A. This can be a source of ambiguity.
  - To avoid this problem, class A must be declared as virtual base class for classes B and C.
- A virtual base class must be initialized by its most derived class. Means class D constructor is responsible to invoke the constructors of classes A, B and C.

What is Virtual Base Class

Declaring a base class, virtual, can resolve the issue of ambiguity.

```
class A
{
   …
};

class B: virtual public A
{
   …
};

class C: virtual public A
{
  …
}
```

```
class D: public  B, public C
{
   …
};
```

Inheriting operators and functions

- A derived class inherits all the member functions of each of its base classes. However a child class still needs to have its own constructors (including copy constructor), and possibly destructor, and assignment operators to manage tasks such as memory management for its own data members (not those inherited from base class).

Derived Class Copy Constructor

- If a derived class explicitly defines its own copy constructor, the definition completely overrides the default definition, and is responsible for copying its base class component.
- Assuming we have a class called Base:

```
class Base { …};
```

- And, a class called Derived, as followed. How its copy-constructor should be?

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

Derived Class Assignment Operator

- If a derived class explicitly defines its own assignment operator, the definition completely overrides the default definition, and is responsible for copying its base class component.
- Assuming we have previous classes Base, and Derived. How should assignment operator of class Derived be?

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

Derived Class Destructor

- Derived class destructor is never responsible for destroying the members of the base object.
- The compiler will always implicitly invoke the destructor of the base part of the derived object:

```
class Base { …};

class Derived: public Base {
public
      ~Derived () { /* ~Base is automatically called */ }
};
```

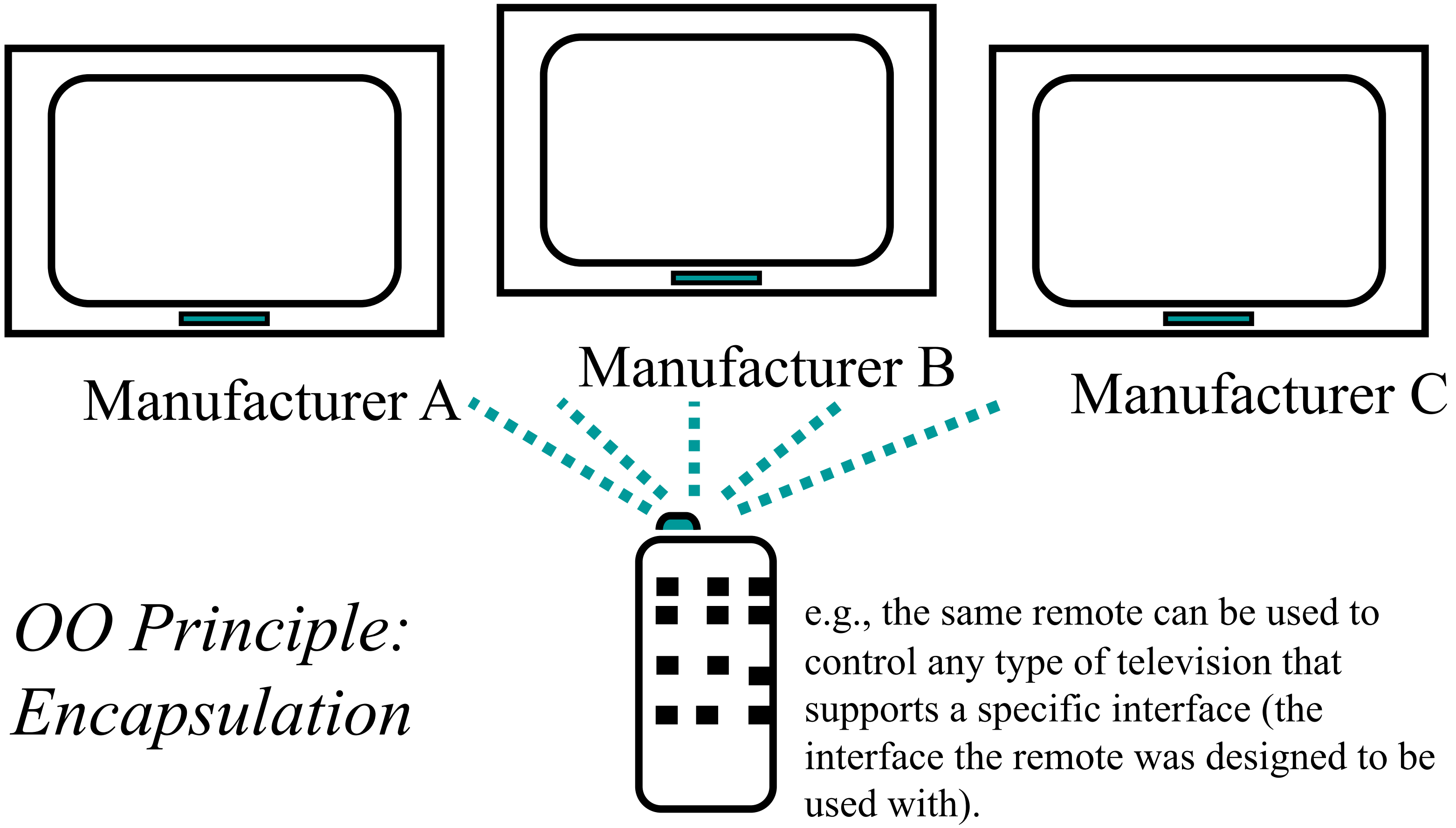
What is Polymorphism?
(A Closer Look)

Polymorphism

- In Biology, the term polymorphism describes the characteristic of an element that may take on different forms.
- In other words, the occurrence of more than one *form* or *morphs*.
- Then, what does it mean in object-oriented programming?

What Is Polymorphism?

- The ability to hide many different implementations behind a single interface



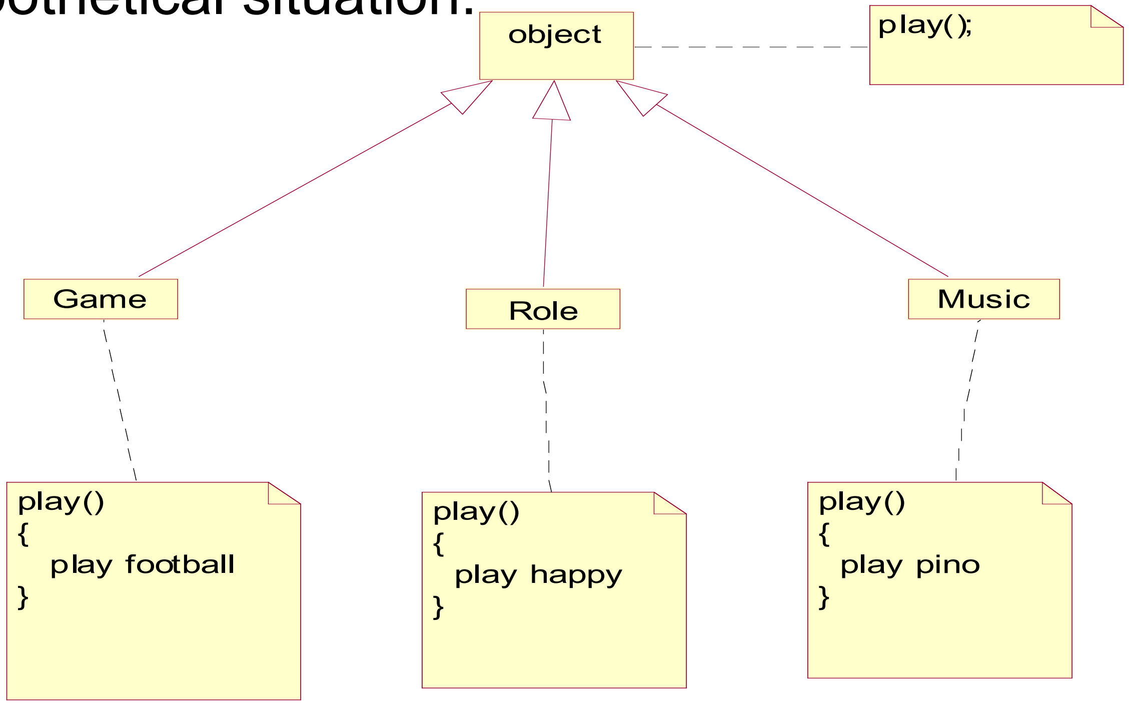
OO Principle: Encapsulation

e.g., the same remote can be used to control any type of television that supports a specific interface (the interface the remote was designed to be used with).

Figure From: IBM Rational Rose

Polymorphism in Object-Oriented Methodology

- In O.O. languages, polymorphism is the property that different objects may react to the same message, differently.
- Lets consider this hypothetical situation:



- The following statements will receive the same message ‘play’ but implement different operations:
  - game.play()
  - music.play()
  - role.play

Implementation of Polymorphism in C++

Example of Polymorphism in C++

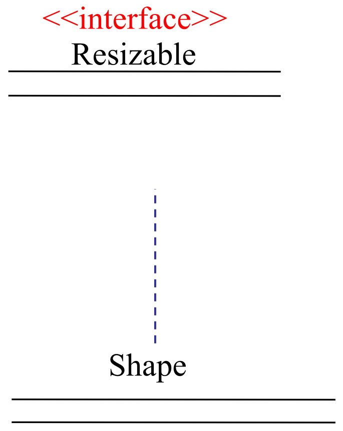
```
class Shape {
  public:
     Shape(double x, double y): origin(x,y) {}
     virtual double area() = 0;
  private:
    Point origin;
};
class Rectangle: public Shape{
  public:
      Rectangle(…);
     double area() { return width * height;}
 private:
    double width, height;
};

class Circle: public Shape{
  public:
     Circle(….);
     double area() { return radius * radius *PI;}
  …
} ;
```

```
class User {
  public:
     double area(Shape* x) {
        return x ->area();
    }

};


void main() {
       User user;
        Rectangle r = Rectangle(6, 7, 8, 9);
       Circle c = Circle(2, 3, 4);
       cout << user.area(&r);
       cout << user.area(&c);
 }
```

What is Realization?

Realization

- In UML modeling, a realization relationship is a relationship between two model elements, in which one model element (the client) realizes (accepts to implement or execute) the behavior that the other model element (the supplier) specifies.



Implementation of Realization in Java (Quick Review)

Example of Realization in Java

```
interface Resizable
{
   public void enlarge();
   public void shrink();
}
```

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

```
class Shape implements Resizable,  Calculable{
   …
}
```

Java Realization Relationship

- Interfaces give Java some of the power of multiple inheritance, however, there is no code reuse, since each class must re-implement the methods.
- A reference of an interface type can refer to the instances of any classes that implement that interface.

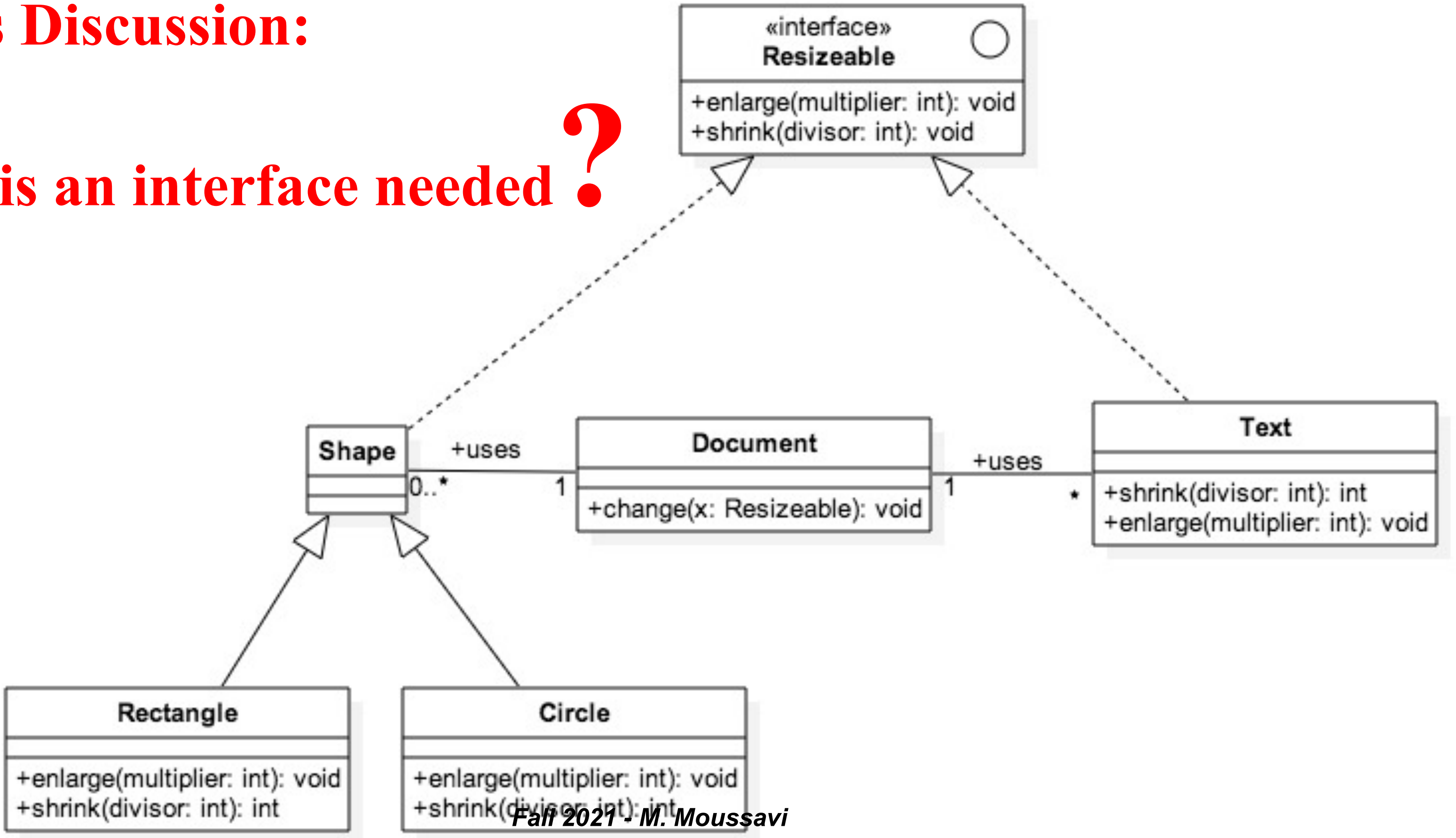
Example in Java

Class Exercise

Assume a word processing software, needs a few class such as Text, Shape, Document, etc. Class Document uses the other two classes: Text and Shape that implement Java interface called Resizable, allowing the shapes and text to enlarged or shrunk.

- **Class Discussion:**

Why is an interface needed?



«interface»
Resizeable
+enlarge(multiplier: int): void
+shrink(divisor: int): void
Shape
+uses
0..*
1
Document
+change(x: Resizeable): void
+uses
1
*
Text
+shrink(divisor: int): int
+enlarge(multiplier: int): void
Rectangle
+enlarge(multiplier: int): void
+shrink(divisor: int): int
Circle
+enlarge(multiplier: int): void
+shrink(divisor: int): int

Answer

- The client class that uses classes Shape, and Text can provide a single method (polymorphism) to resize either a shape or a text:

object can be
Shape or Text

```
class WordProcessing {
        private Text t;
       private Shape sh;
       public void enlarge (Resizeable object) {
          object.enlarge();
      }
      // More code
}
```

- Future changes: If in future you need to add new resizable objects you don't need to make changes to the client classes.

Class Discussion:

- What about Implementation of realization in C++? How can realization be implemented, while C++ doesn’t have a feature such as ‘Java interface’?

Example of Realization in C++

- C++ doesn't support specific syntax for interfaces.
- However the concept can be implemented by using a class with abstract, and pure virtual functions.

```
class Resizable
{
  virtual void enlarge() =0;
   virtual void shrink()  =0;
};
```

```
class Calculable
{
   virtual double area() = 0;
};
```

```
class Shape: public Resizable, public Calculable{
   private:



};
```