CptS 355- Programming Language Design

Object Oriented Programming and Object Oriented Languages

Instructor: Sakire Arslan Ay

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

5 Insights About OO Programming

1. Data and operations belong together
2. Abstraction and interface: implementation details are hidden
3. Subtyping
4. Inheritance
5. Static and Dynamic Method Binding (i.e., Static and Dynamic Dispatch)

5 insights about OO Programming

1. Data and operations belong together
   - By packaging up data and operations/methods it will be clearer and easier to write and understand programs.
   - This is closely related to “data abstraction” or “abstract data types”

5 insights about OO Programming

2. **Abstraction and interface**:
   - The only access to the data should be only via the defined (abstract) operations.
   - Implementation details are hidden
   - **Interface:** a set of operations expressed in application-level terms rather than implementation-level terms.
     - Includes: operations, their arguments, their results, and their meaning.
   - In Java, the use of the term “interface” is little bit different

5 insights about OO Programming

3. Subtyping
   - A is a subtype of type B when:
     - methods (A) $\supseteq$ methods (B), and
     - fields(A) $\supseteq$ fields (B)
   - Notation: **A<:B**
   - If A<:B then:
     - A can be used whenever B can.

- If A<:B, then “A provides all operations B provides, in terms of signatures.”
  - However the operations may not have the same meaning. To avoid that languages require the programmer to explicitly declare subtype relationships.

5 insights about OO Programming

4. Inheritance
   – The implementations of methods of a supertype are available (and used) in a subtype unless they are overridden.

- Note: Subtyping vs inheritance
  – subtyping is about compatibility of interfaces (in the general sense)
  – inheritance is about re-use of implementations.

5 insights about OO Programming

5. Static and Dynamic Method Binding

   (Static and Dynamic Dispatch)

   – Dynamic Dispatch:
     - When we have subtyping, the method to call depends on the actual value contained in variable, not its type.
     - Binding of the method is determined at run time depending on the type of the object pointed to.

Dynamic vs Static Dispatch

- **C++ Example**

```
class Person
{
  public:
    Person() { }
    void setSSN(int myssn){
      ssn=myssn;
    }
    void print {
       out << ssn;
    }
  private:
    int ssn;
}
```

```
class Student : public Person
{
  public:
    Student() { }
    void setGPA(float mygpa){
        gpa=mygpa;
    }
    void print {
       out << gpa;
    }
  private:
    double gpa;
}
```

```
void main(){
    Student s ;
    s.setSSN("119-23-3212");
    s.setGPA(3.41);
    Person *p = &s;
    s.print();
    (*p).print();
}
```

The compile time type of (*p) is Person.
The runtime type of (*p) is Student.

Dynamic Dispatch

```
void main(){
    Student s ;
    s.setSSN('119-23-3212');
    s.setGPA(3.41);
    Person *p = &s;
    s.print();
    (*p).print();
}
```

- In C++, by default, the decision on which member function to invoke (base or overriden) is made on the basis of the compile-time type.
  - This is called *static* dispatch
- The decision is made based on the runtime type, when the member function is defined as a <u>virtual</u> function.
  - This is called *dynamic* dispatch
- How about Java?
- Why the static dispatch is the default in C++?

Dynamic Dispatch

- **C++ Example**

```
class Person
{
  public:
    Person() { }
    string& setSSN(int myssn){
      ssn=myssn;
    }
    virtual void print {
       out << ssn;
    }
  private:
    int ssn;
}
```

```
class Student : public Person
{
  public:
    Student() { }
    string& setGPA(float mygpa){
       gpa=mygpa;
    }
    virtual void print {
       out << gpa;
    }
  private:
    double gpa;
}
```

```
void main(){
    Student s ;
    s.setSSN('119-23-3212');
    s.setGPA(3.41);
    Person *p = &s;
    s.print();
    (*p).print();
}
```

- A virtual function uses dynamic dispatch
- A non-virtual function uses static dispatch.

How are virtual methods implemented in C++?

- On a per class basis (not per instance) there is a run-time data structure called a *v-table* that contains pointers to the code for the virtual methods.

How are objects and virtual methods implemented in C++?

- Example:

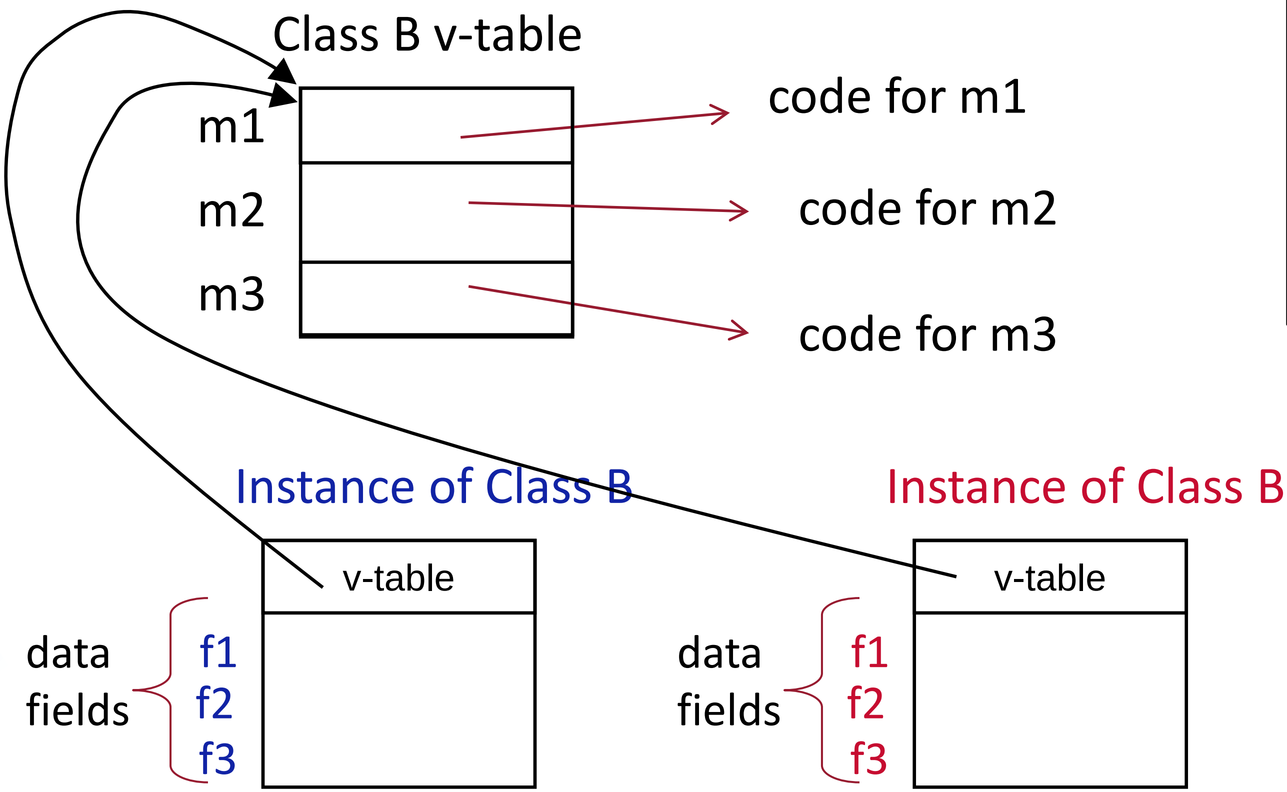
```
class B
{
  public:
    int f1;
    int f2;
    int f3;
    virtual void m1{ some code  }
    virtual void m2{ some code  }
    virtual void m3{ some code  }
    void m4{ some code }
}
```

```
class A : public B
{
  public:
    int f4;
    virtual void m2{ some code  }
    virtual void m3{ some code  }
    virtual void m5{ some code  }
    void m4{ some code }
}
```

```
int main(){
    B p ;
    p.m1();
    B *q =  new A();
    q->m1();
    q->m2();
    q->m4();
}
```

How are objects and virtual methods implemented in C++?

- *v-table:* run-time data structure which contains pointers to the code for the <u>virtual</u> methods.



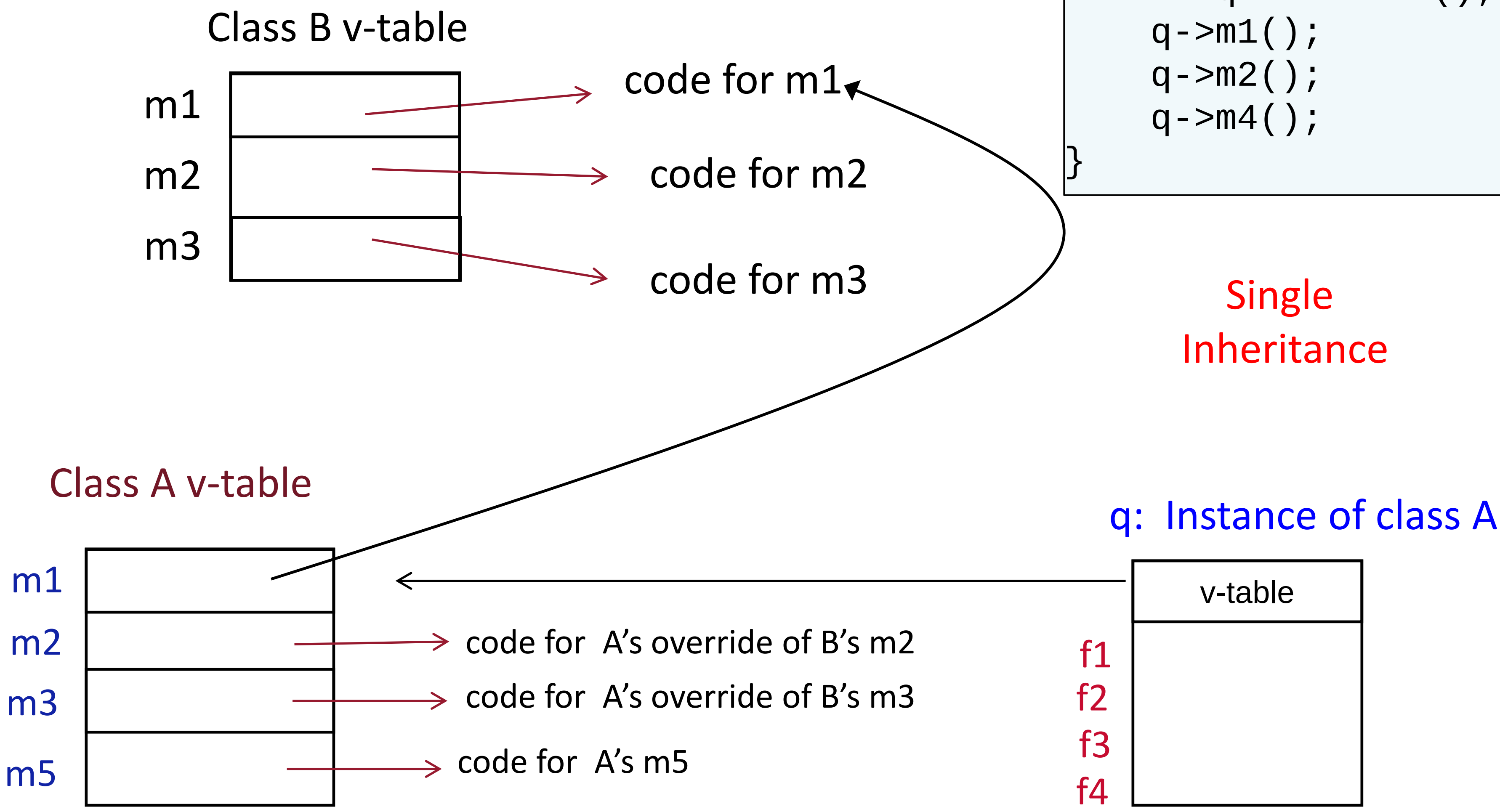

```
int main(){
    B x ;
    x.m1();
    B *y = new B();
    y->m2();
}
```

If m4 is a non-virtual method, what happens?

How are objects and virtual methods implemented in C++?

- class A is subclass of class B

```
int main(){
    B p ;
    p.m1();
    B *q =  new A();
    q->m1();
    q->m2();
    q->m4();
}
```



How are objects and virtual methods implemented in C++?

- Example:

```
class B
{
  public:
    int f1;
    int f2;
    int f3;
    virtual void m1{ some code  }
    virtual void m2{ some code  }
    virtual void m3{ some code  }
    void m4{ some code }
}
```

```
class A : public B
{
  public:
    int f4;
    virtual void m2{ some code  }
    virtual void m3{ some code  }
    virtual void m5{ some code  }
    void m4{ some code }
}
```

```
int main(){
    B p ;
    p.m1();
    B *q =  new A();
    q->m1();
    q->m2();
    q->m4();
}
```

How are objects and virtual methods implemented in Java?

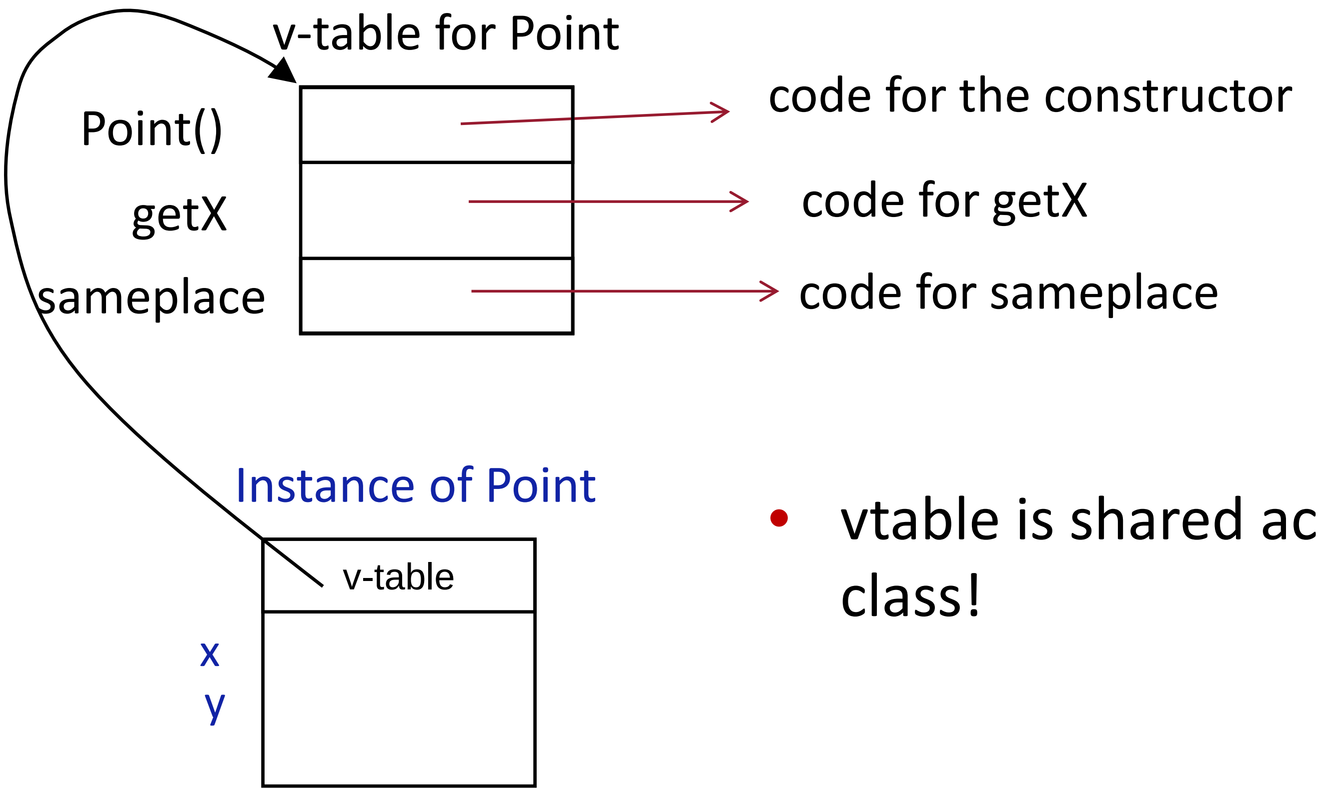
- Example:

```
class Point
{
  public:
    double x;
    double y;
    void Point (double x,double y){
       this.x = ; this.y=y;
    }
    double getX(){
       return x;
    }
    boolean sameplace (Point p){
       return (x==p.x) && (y==p.y)
    }
}
```

```
class PtSubClass extends Point
{
 public:
   int aNewField;
   void PtSubClass(double x,double y){
      super(x,y)
   }
   boolean sameplace (Point p){
      return false;
   }
   void sayHi () {
    System.out.println(“hello!”);
   }
}
```

```
int main(){
    Point p = new Point();
    Point q = new PtSubClass ();
    ...
}
```

How are objects and virtual methods implemented in Java?



- vtable is shared across all objects in class!

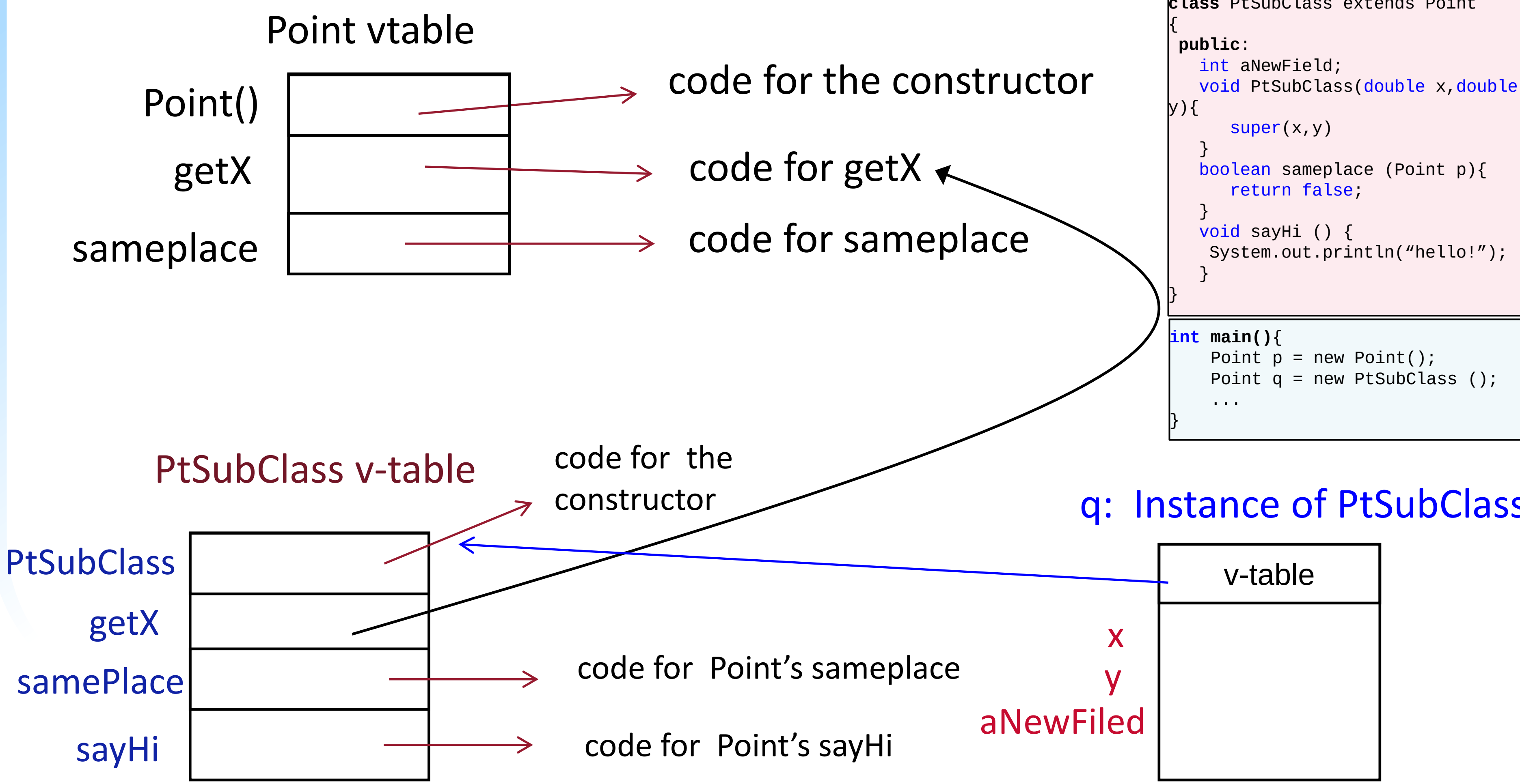
- If the object instance of Point is no longer needed, what will happen to it’s vtable?

How are objects and virtual methods implemented in Java?

- PtSubClass is subclass of Point

```
class PtSubClass extends Point
{
 public:
   int aNewField;
   void PtSubClass(double x,double
y){
      super(x,y)
   }
   boolean sameplace (Point p){
      return false;
   }
   void sayHi () {
    System.out.println("hello!");
   }
}
```

```
int main(){
    Point p = new Point();
    Point q = new PtSubClass ();
    ...
}
```



Multiple Inheritance

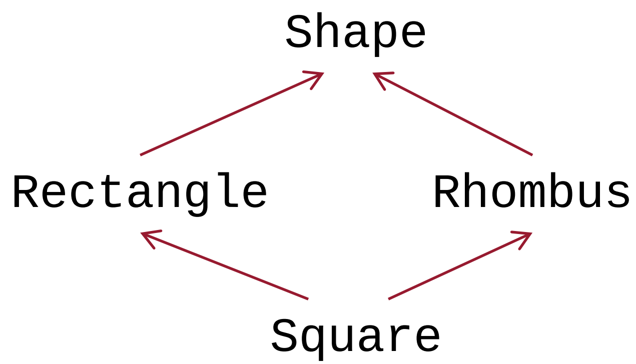
- A class inherits from 2 or more other classes
- Why multiple inheritance?
  - When modeling a domain, you often want to express more than one "kind-of" relationship for an object.
    - <u>Example</u>: `ReadWriteStream` (representing a readable and writeable file) is both a `ReadStream` and a `WriteStream`
- However, since there is more than one superclass, problems with ambiguity arise.

Multiple Inheritance

- Problems with ambiguity:
  - A class C may inherit from two base classes A and B that both define a method for a message **`M`** or both define an instance variable **`v`**.
    - When we access **`C.M`**, should A's method for **`M`** be invoked, or should B's method for **`M`** be invoked?
    - Should two copies of V be inherited, or one?
    - This is called a **name clash**.

Multiple Inheritance

- Problems with ambiguity:
  – Diamond inheritance: Some base class has two kinds of extensions, and one would like to combine them into a third kind that has the properties of both.
    - Many "natural" inheritance hierarchies have this form




```
class Shape { float area() { ... } }
class Rectangle sublasses Shape { float area() { ... } }
class Rhombus subclasses Shape { float area() { ... } }
class Square subclasses Rectangle, Rhombus {}
```

Multiple Inheritance

Duplicate method solutions:

1. User resolves ambiguity by overriding in subclass and directing resends to one class
   - C++ uses this approach

```
class Square subclasses Rectangle, Rhombus {
    Float area() {
        return super(Rhombus).area(); }
}
```

Not an actual syntax

   - Advantages:
     - The user has the flexibility to select which inherited methods get invoked for which messages.
     - The user gets feedback if (s)he forgets to override an ambiguously inherited method

Multiple Inheritance

2. User resolves ambiguity by specifying textual ordering.
   - For example: Superclasses are searched from left-to-right (in order of textual declaration at the class definition) for methods. The first one found is the one executed.
   - Python uses this approach.

```
class Shape { float area() { ... } }
class Rectangle sublasses Shape { float area() { ... } }
class Rhombus subclasses Shape { float area() { ... } }
class Square subclasses Rectangle, Rhombus {}
```

   - `Square.area`?
   - Disadvantage:
     - Lacks flexibility --- what if you wanted to inherit some methods from `Rectangle`, and other methods from `Rhombus`?

Multiple Inheritance

3. Prohibit multiple inheritance with overlapping methods.
   - Disadvantage:
     - In practice, there are too many opportunities that one must forego. Can't take advantage of those.

Multiple Inheritance

Duplicate instance variable solutions:

```
class Shape { float area() { ... } }
class Rectangle sublasses Shape {
    float area() { ... }
    Point topLeft;
    Point bottomRight;
}
class Rhombus subclasses Shape {
    float area() { ... }
    Point topLeft;
    Point topRight;
    Point bottomRight;
}
class Square subclasses Rectangle, Rhombus {}
```

Not an actual syntax

1. Merge duplicates
2. Always duplicate
3. Merge if originally from same declaration

```
class Window { List menuItems; }
class Restaurant { List menuItems; }
class RestaurantWindow subclasses Window, Restaurant {}
```

Multiple inheritance vs. multiple subtyping

- Recall: inheritance and subtyping are different---inheritance concerns implementations, and subtyping concerns interfaces
- Java prohibits multiple inheritance of implementation. However, it supports "multiple inheritance of interface".

```
interface IShape { Float area(); }
interface IRectangle extends IShape { ... }
interface IRhombus extends IShape { ... }
interface ISquare extends IRhombus, IRectangle { ... }

abstract class Shape implements IShape {}

class Rectangle extends Shape implements IRectangle {
    float area() { ... }
}
class Rhombus extends Shape implements IRhombus {
    float area() { ... }
}
class Square extends Rhombus  implements ISquare {
}
```

Multiple Inheritance

- The problem of doing multiple inheritance "right" is still an open problem in language design.
- Implementation is hard.

Miscellaneous

- Method overloading

```
class Calculate{
  void sum(int a,int b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
        Calculate c = new Calculate ();
        c.sum(10,10,10);
        c.sum(20,20);
  }
}
```