Administrivia

- Student hours (or by appointment):
  - Listed on Canvas in Syllabus
  - MTWTh 10:30 – 11:30 a.m. via Zoom
- Assignment five posted
  - Due Friday 14 July by **10pm**
- Midterm exam 2 next Monday
  - Review Wednesday

CptS 355- Programming Language Design

Object-Oriented Programming and Object-Oriented Languages - Multiple Inheritance

Instructor: Jeremy E. Thompson

WASHINGTON STATE UNIVERSITY

World Class. Face to Face.

RECALL: How are ***virtual*** methods implemented in C++?

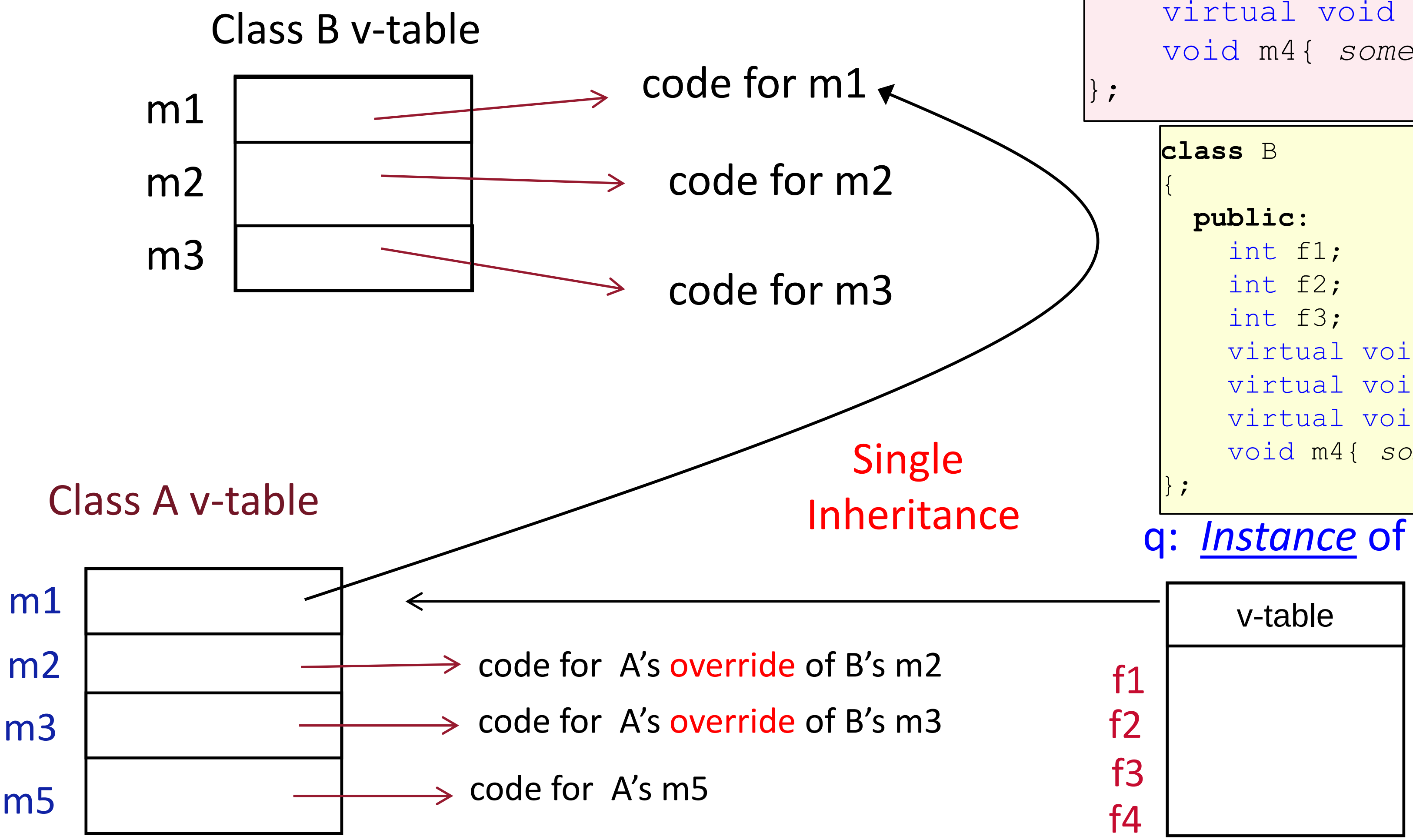
- On a *per-class* basis (not *per-instance*)
  - run-time data structure called a *v-table* that contains pointers to the code for *virtual* methods

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

- class A is *subclass* of class B

```
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 }
};
```

```
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 }
};
```



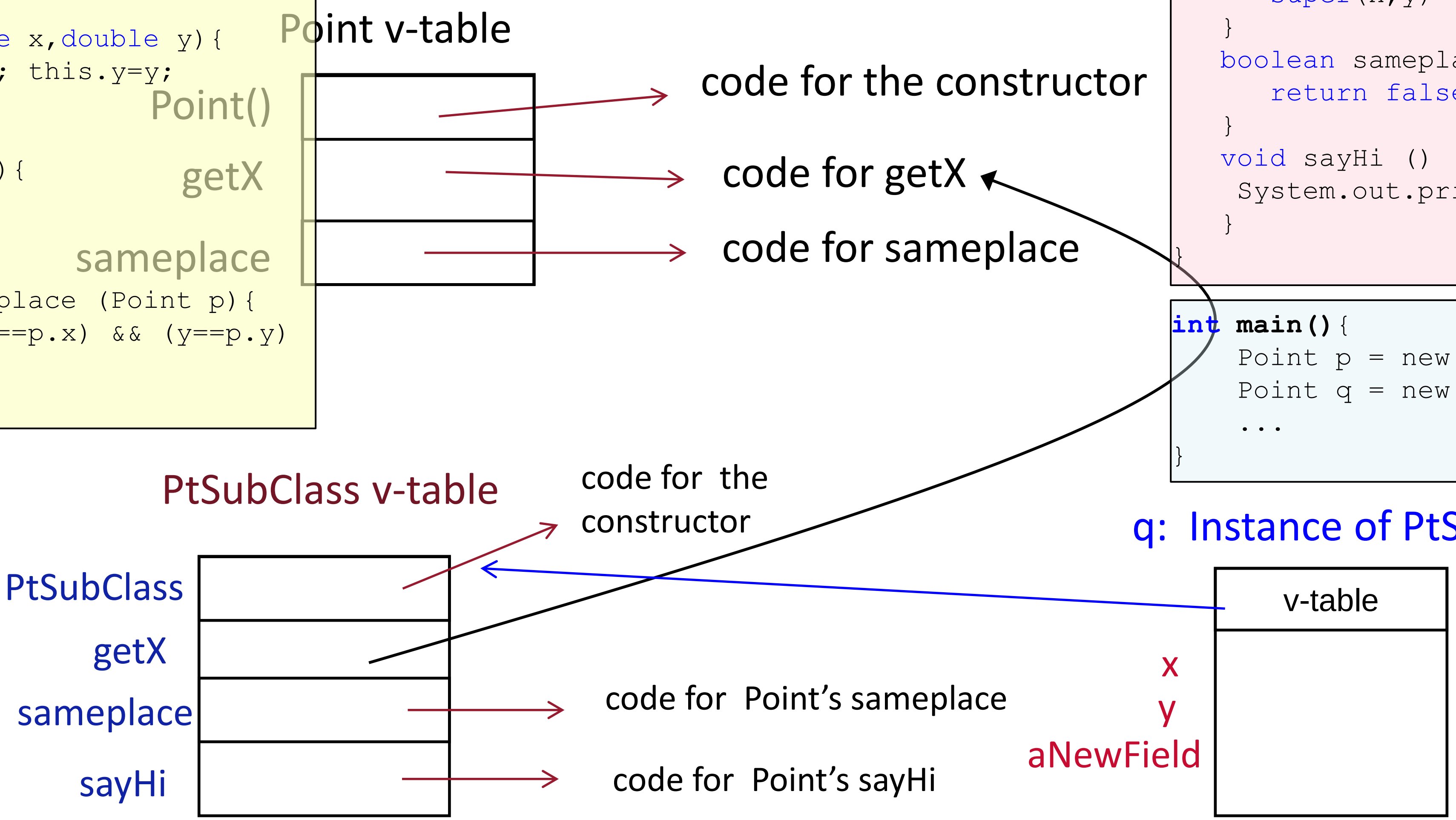
RECALL: How are objects and virtual methods implemented in Java?

```
class Point
{
  public:
    double x;
    double y;
    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)
    }
}
```

- 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

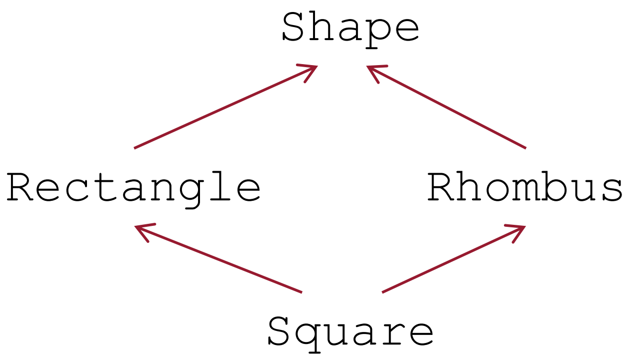
- A class inherits from 2 or more other classes
- Why multiple inheritance?
  - When modeling a domain, you often want to express more than one "kind-of" relationship for an object
    - Example: `ReadWriteStream` (representing a readable *and* writeable file) is *both* a `ReadStream` *and* a `WriteStream`
- 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

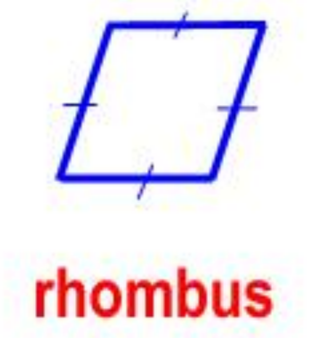


Shape

Rectangle

Rhombus

Square



This Photo by Unknown Author is licensed under CC BY

```
class Shape { float area() { ... } }
class Rectangle subclasses 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 actual syntax

   - Advantages:
     - User has *flexibility* to select which inherited methods get invoked for which messages
     - User gets *feedback (error)* if they *forget 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
   - Java does not allow multiple inheritance, except for ***interfaces***
     - Why interfaces, do you think?
   - Disadvantage:
     - In practice, there are too many opportunities that one must forego

Multiple inheritance vs. multiple subtyping

- Recall: inheritance and subtyping are different
  - inheritance concerns implementations
  - 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
- i.e., Implementation is *difficult*

Miscellaneous

- Method overloading *(vs overriding)*

```
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);
  }
}
```

- What's the *difference*?

Questions?