Object Oriented Programming with Java

PART 1: JAVA PROGRAMMING BASICS

Interfaces, By Aphrodice Rwagaju

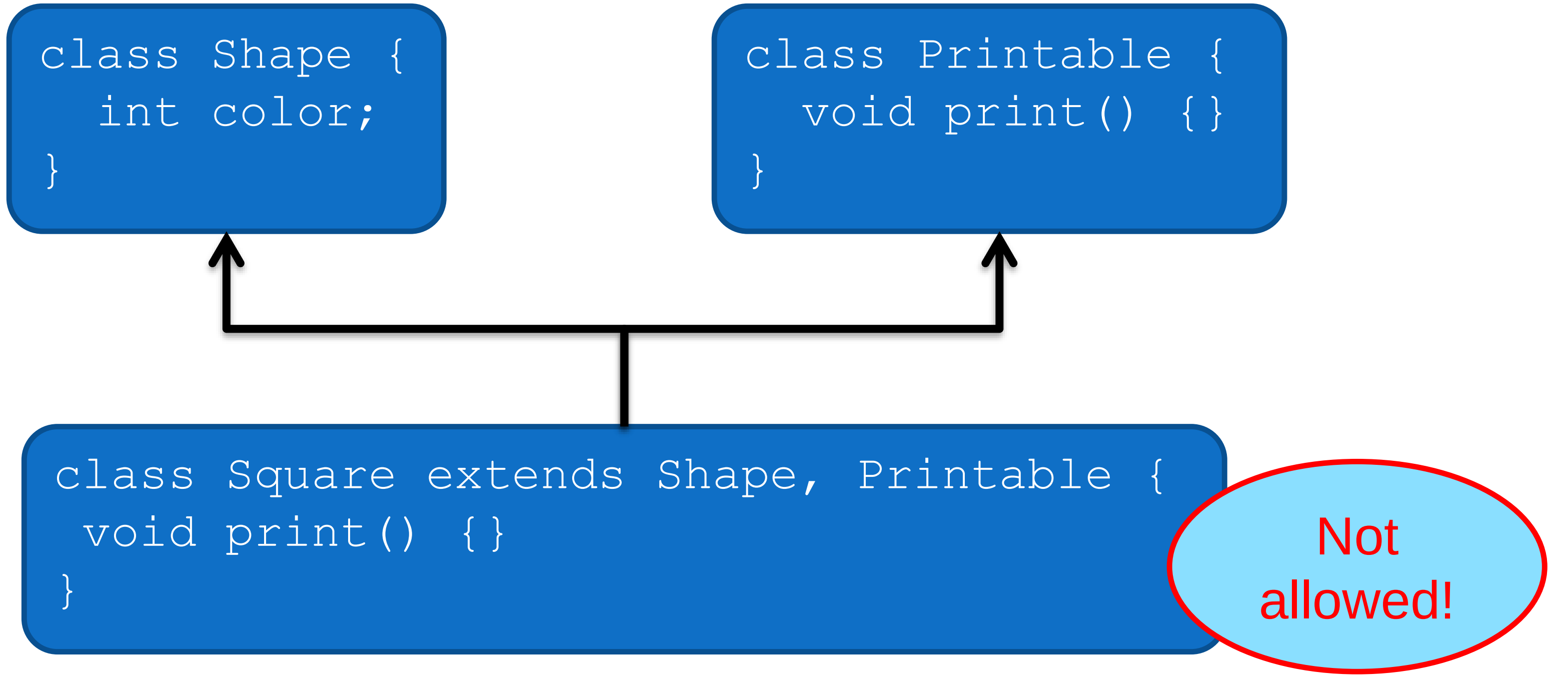
Inheritance Recap …

- We've looked at how classes can extend other classes and override their methods and variables
- For example: `a.foo()` will actually call the `foo` method of `B`

```
class A {
   public void foo() {
   }
}
class B extends A {
   public void foo() {
   }
}
class App{
   App(){
      A a = new B();
      a.foo();
   }
}
```

Multiple Inheritance?

- What if we need our class to inherit the functionality of more than one superclass?



```
class Shape {
  int color;
}
```

```
class Printable {
  void print() {}
}
```

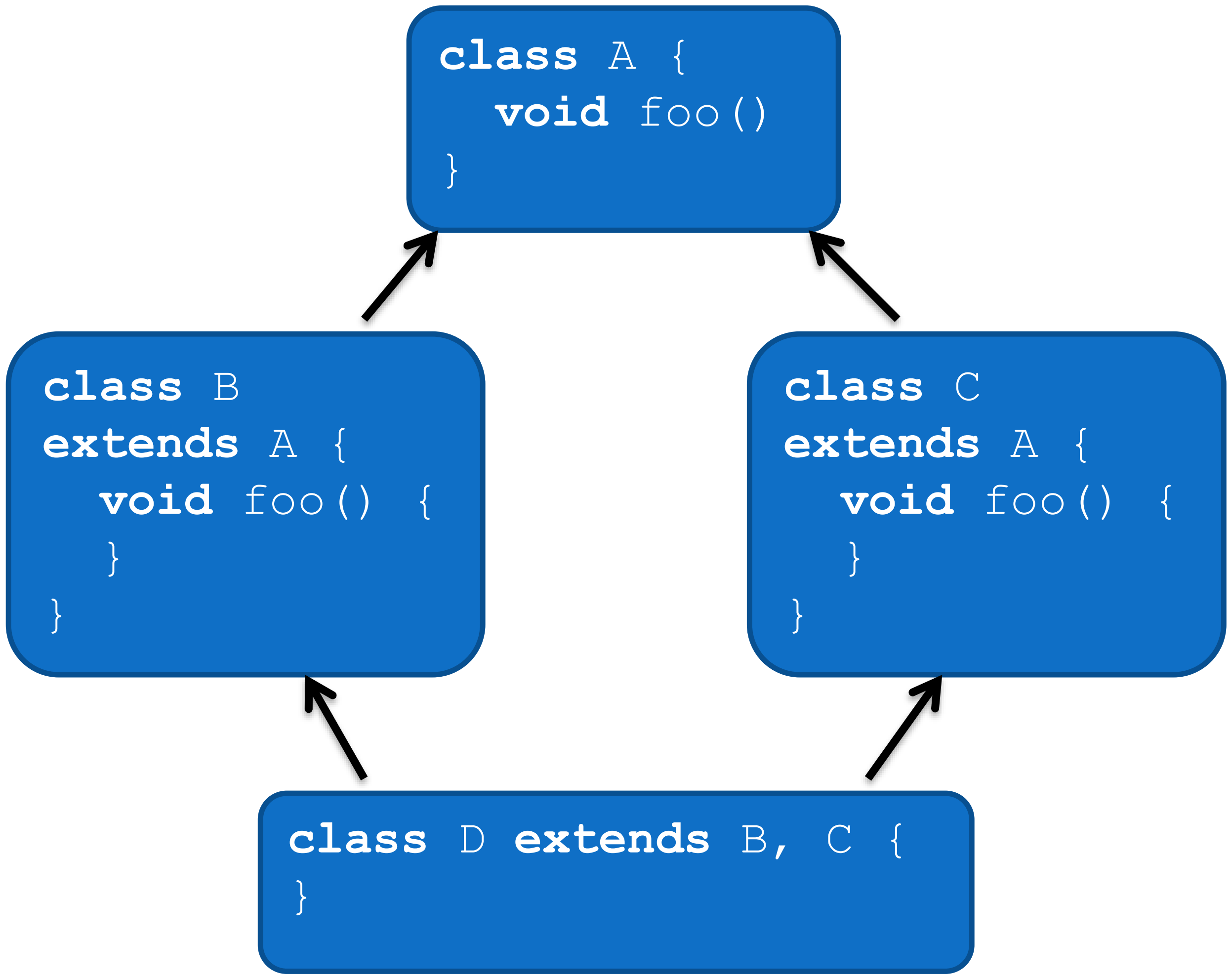
```
class Square extends Shape, Printable {
 void print() {}
}
```

Multiple Inheritance?

- This is allowed in some programming languages such as C++, but NOT in Java
  - It makes code overly complex
  - The diamond problem…

The Diamond Problem

```
A a = new D();
a.foo();
```



```
class A {
  void foo()
}
```

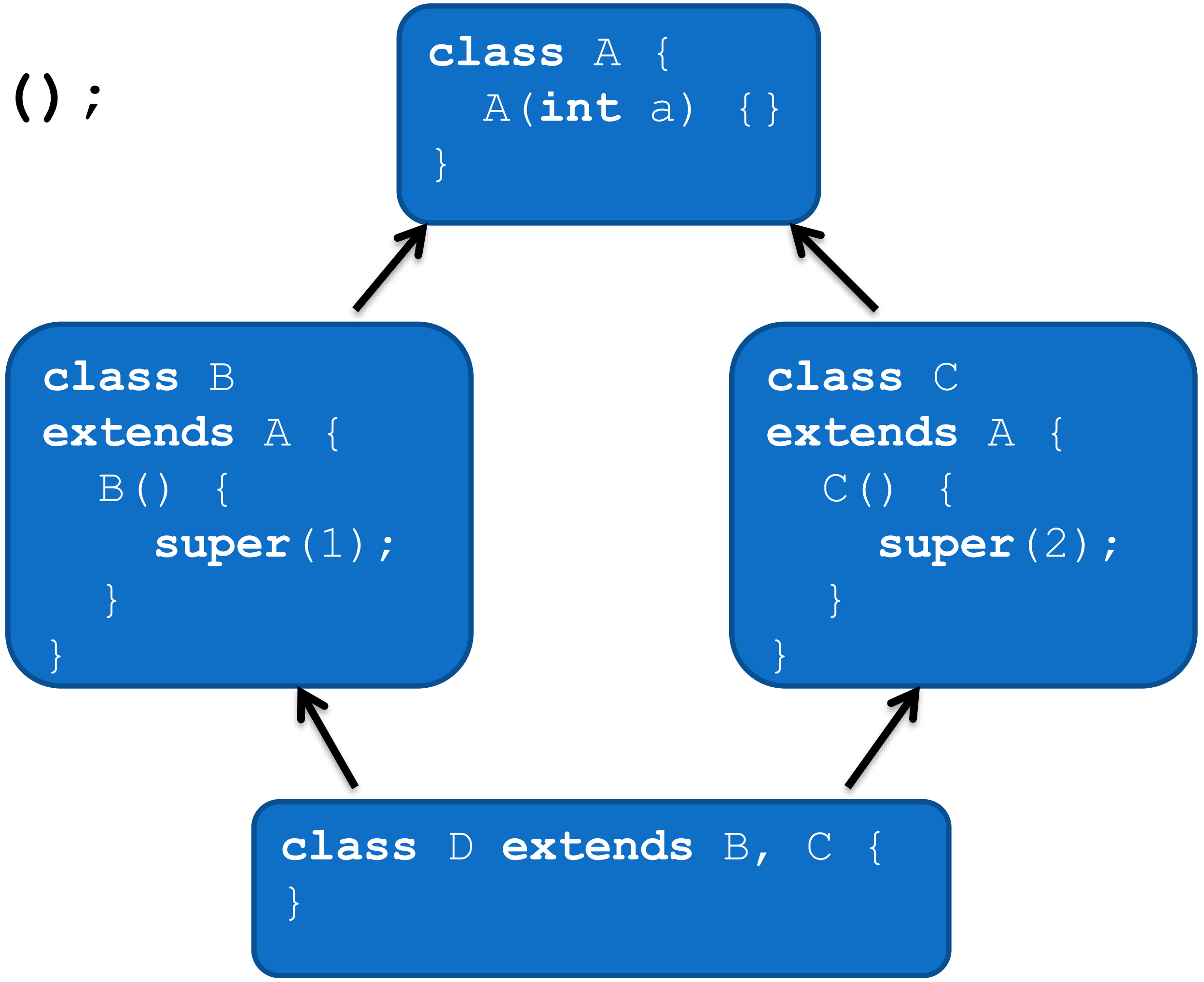
```
class B
extends A {
  void foo() {
  }
}
```

```
class C
extends A {
  void foo() {
  }
}
```

```
class D extends B, C {
}
```

The Diamond Problem with Constructors

```
D d = new D();
```



```
class A {
   A(int a) {}
}
```

```
class B
extends A {
   B() {
      super(1);
   }
}
```

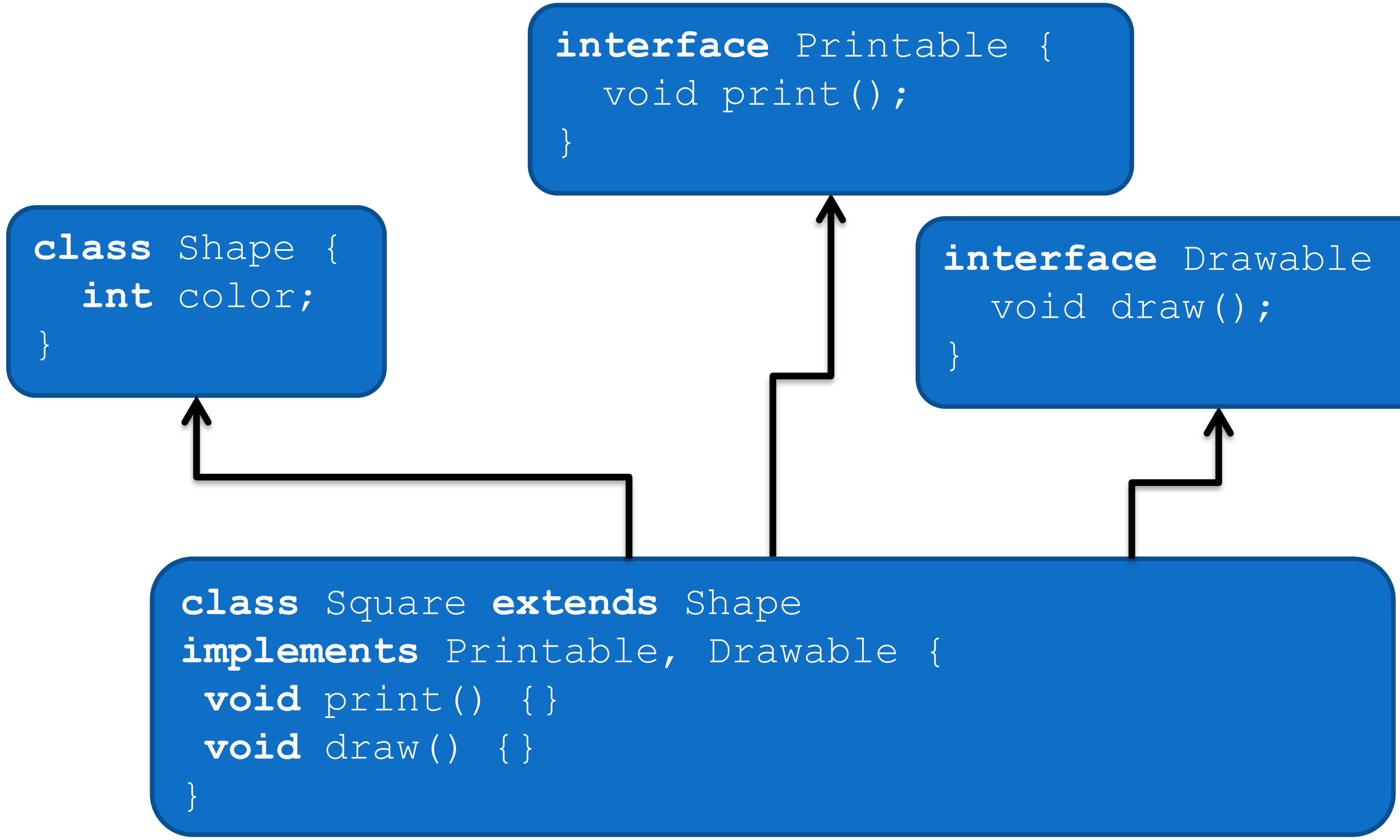
```
class C
extends A {
   C() {
      super(2);
   }
}
```

```
class D extends B, C {
}
```

Java's Solution: Interfaces

- You can think of an interface as a 100% abstract class – all the methods are abstract and thus not implemented
- We can only **extend** one class, but we can **implement** as many interfaces as we want
- Interfaces can extend other interfaces, but because they are 100% abstract, they can't override methods

Example



```
interface Printable {
  void print();
}
```

```
class Shape {
  int color;
}
```

```
interface Drawable {
  void draw();
}
```

```
class Square extends Shape
implements Printable, Drawable {
 void print() {}
 void draw() {}
}
```

Inheritance

- When one interface extends another, it inherits its methods
- Any implementing class must implement all of the inherited methods as well

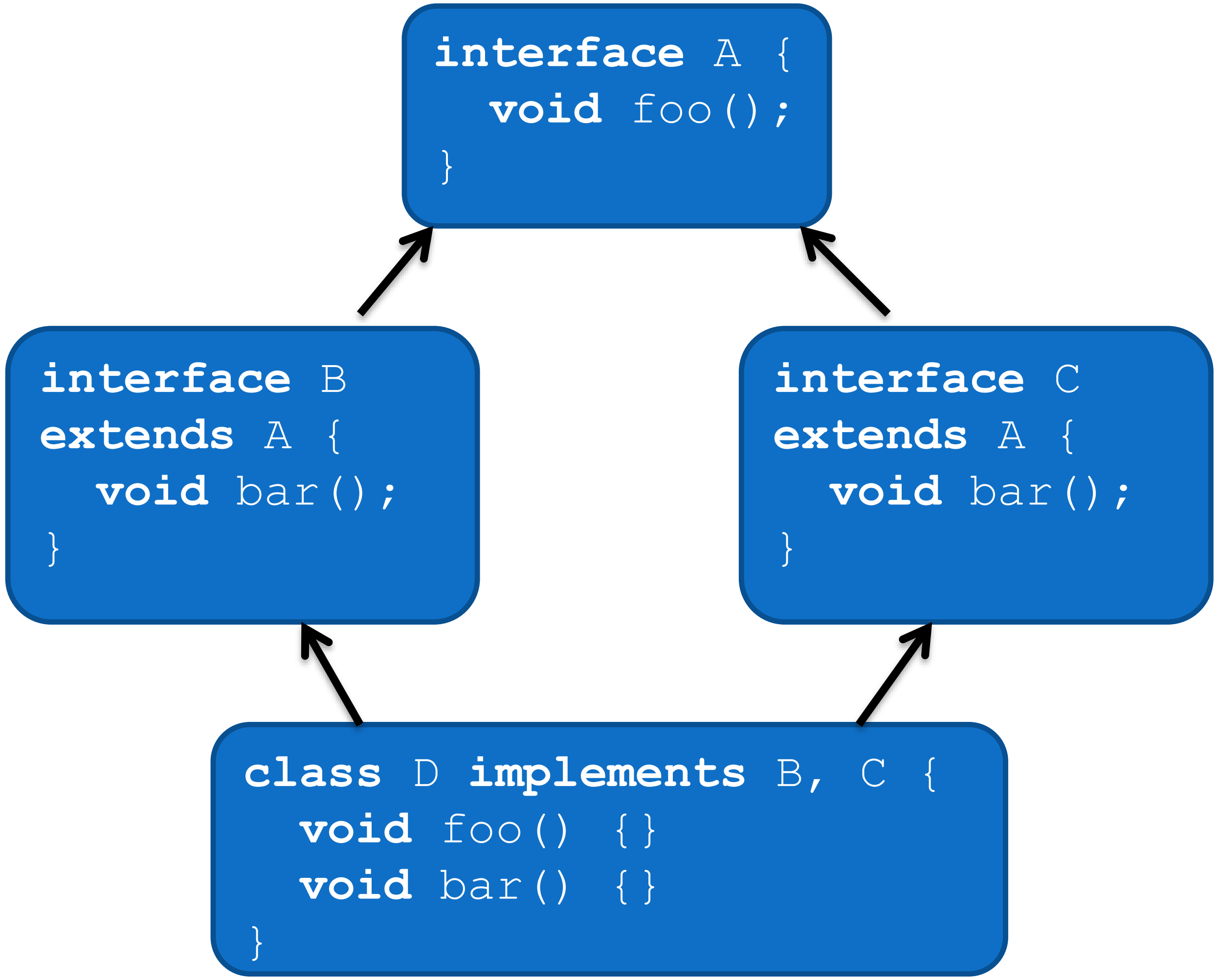
```
interface A {
  public void foo();
}

interface B
extends A {
  public void bar();
}

class App
implements B {
  public void foo() {}
  public void bar() {}
}
```

No more Diamond Problem…

- `B` and `C` can't override `foo()`
- Doesn't matter that `B` and `C` both define `bar()`



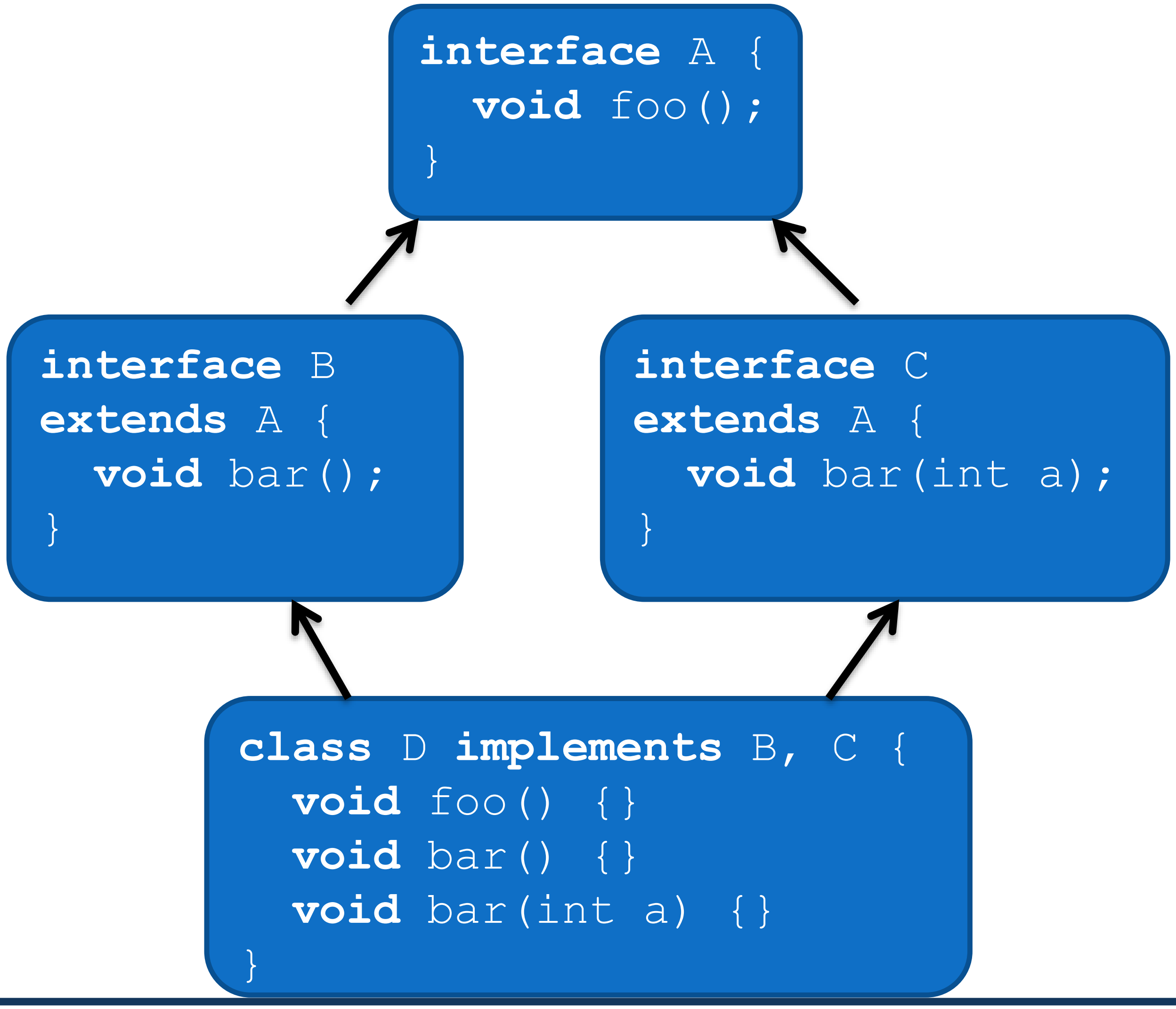
```
interface A {
  void foo();
}
```

```
interface B
extends A {
  void bar();
}
```

```
interface C
extends A {
  void bar();
}
```

```
class D implements B, C {
  void foo() {}
  void bar() {}
}
```

No more Diamond Problem…



```
interface A {
   void foo();
}
```

```
interface B
extends A {
   void bar();
}
```

```
interface C
extends A {
   void bar(int a);
}
```

```
class D implements B, C {
   void foo() {}
   void bar() {}
   void bar(int a) {}
}
```

Interface Constants

- Interfaces cannot include instance variables, but they can include constants
- Any variable declared within an interface is implicitly public static final, so these keywords can be omitted

```
interface A {
  double PI = 3.14159;
  public static final double E = 2.718282;
}
```

Interfaces with Access Modifiers

- Recap: methods and variables within classes default to package-private
- However, methods and constants within interfaces default to public
- This can lead to compiler errors if you omit access modifiers from your code…

```
interface A {
      void foo();
}
class B implements A {
      void foo() {} // Error!! - can't reduce the visibility
                    // of foo()
}
```

Example: Sorting an Array

- Interfaces allow us to define what methods are required by a class, without defining anything about their implementation. For example:
- The sort function of Arrays requires a class with a method that can compare two objects in an array.

Sorting using Comparator

- `Arrays.sort` is declared as

```
void sort(T[], Comparator<? super T>)
```

- So, to sort an array of `Shape` objects we need to pass it the array to be sorted, and an instance of a class which implements `Comparator<Shape>`

Sorting using Comparator

- The interface `Comparator<T>` defines one method

```
int compare(T arg0, T arg1);
```

- So implementing `Comparator<Shape>` means that our class must include the following method

```
int compare(Shape arg0, Shape arg1) {
  // Do the comparison
}
```

Sorting using Comparator

```
class Sorter implements Comparator<Shape> {

  Shape[] shapes = new Shape[5];
  void doSort() {
     Arrays.sort(shapes, this);
  }

  int compare(Shape s1, Shape s2) {
    return s1.getZIndex() - s2.getZIndex();
  }
}
```

Reference

https://docs.oracle.com/javase/tutorial/java/IandI/

EoF