# 6 - Interfaces

Computer Science Department
California State University, Sacramento



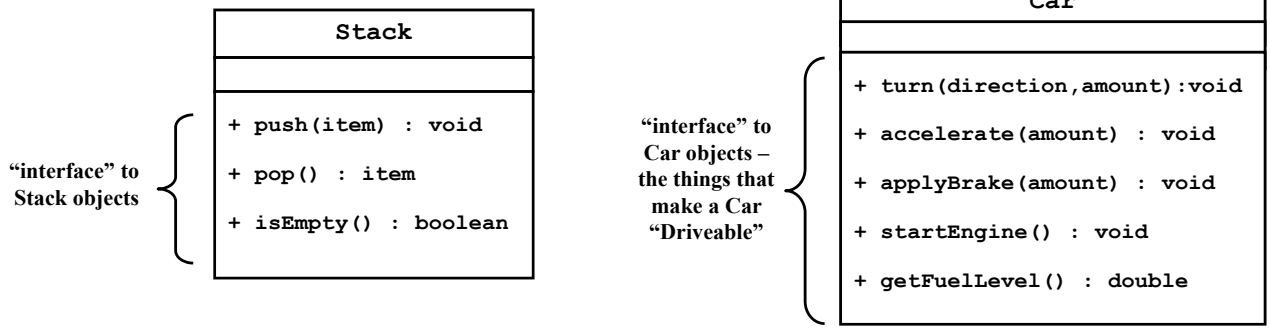
## Overview

- **Class Interfaces, UML Interface Notation, The Java *Interface* Construct**
- **Interfaces in C++**
- **Predefined Interfaces**
- **Interface Hierarchies**
- **Interface Subtypes**
- **Interfaces and Polymorphism**
- **Abstract Classes vs. Interfaces**
- **Multiple Inheritance via Interfaces**

# CLASS INTERFACES

## Every class definition creates an "interface"

- The exposed (non-private) parts of an object

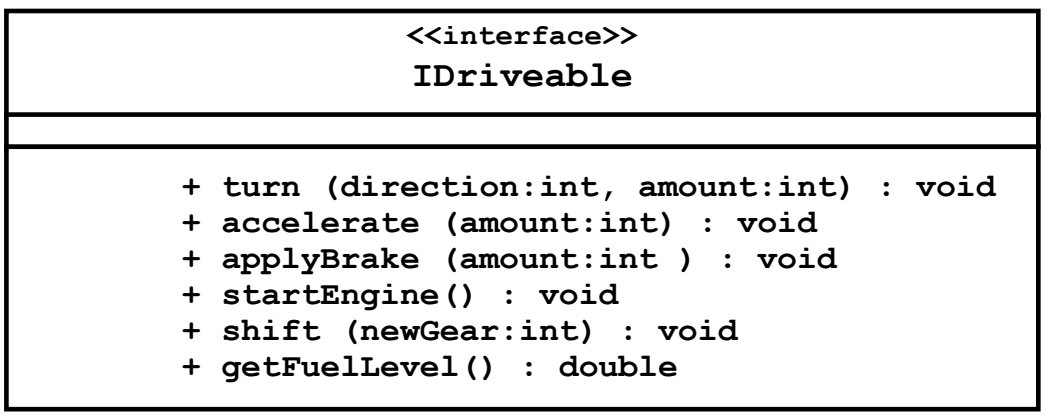






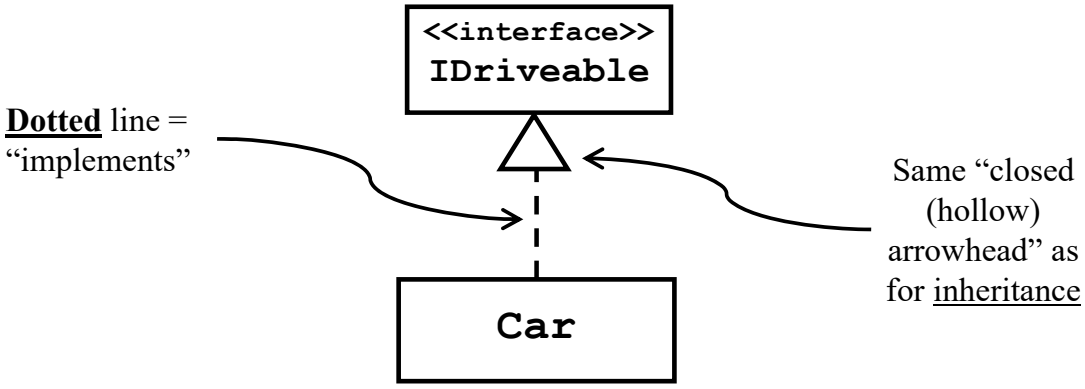

# UML Interface Notation


or



# UML Interface Notation (cont.)

- Class `Car` implements interface `"IDriveable"`:

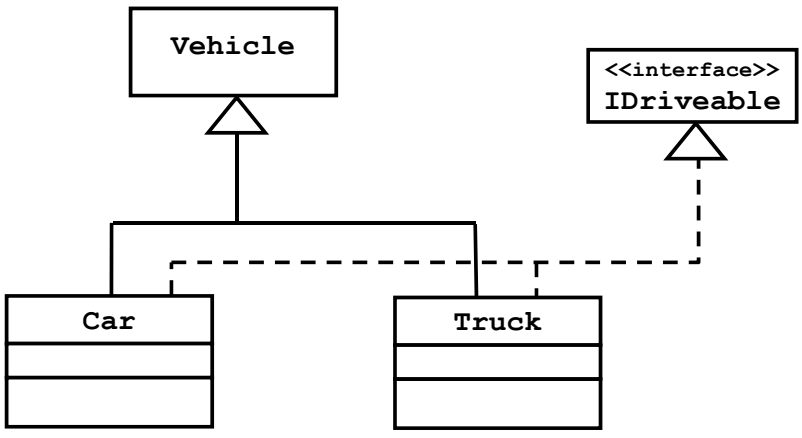







# UML Interface Notation (cont.)

- Car and Truck both derive from "`Vehicle`"
- Car and Truck both implement "`IDriveable`"



# Java *Interface* construct

## Characteristics of an "interface" :

- Defines a set of methods with specific signatures
  - All methods are **public**
- Usually does *not* specify any *implementation* *(generally have abstract methods)*
  - *Java 8 introduced "default" and "static" interface methods that have body*
- Can have fields
  - All fields are **public** AND **static** AND **final**

(default visibility for interface fields and methods is public instead of package-private)





# Java *Interface* construct (cont.)

## Specification of an "interface":

```
public interface IDriveable {
    void turn (int direction, int amount);
    void accelerate (int amount);
    void applyBrake (int amount);
    void startEngine ( );
    void shift (int newGear);
    double getFuelLevel ( );
}
```

# Using Java Interfaces

## Classes can agree to "implement" an interface:

```
public class Car extends Vehicle implements IDriveable {
    public void turn (int direction, int amount) {...}
    public void accelerate (int amount) {...}
    public void applyBrake (int amount) {...}
    public void startEngine() {...}
    public void shift (int newGear) {...}
    public double getFuelLevel ( ) {...}
    /*... other Car methods (if any) here ... */
}
```

- "**`implements`**" in a concrete class == "provides bodies for all abstract methods"
- *Compiler checks!*





# Using Java Interfaces (cont.)

## Multiple classes may provide the same *interface* but with different *implementations*

- **Example: Truck also implements "`IDriveable`" – but in a different way:**

```
public class Truck extends Vehicle implements IDriveable {
    public void turn (int direction, int amount) {...}
    public void accelerate (int amount) {...}
    public void applyBrake (int amount)
      { different code here to apply Truck brakes... }
    public void startEngine()
      { truck engine startup code... }
    public void shift (int newGear)
      { truck shifting code... }
    public double getFuelLevel ( )
      { code to check multiple fuel tanks... }
    /*... other Truck methods here ... */
}
```

# Interface Inheritance

- Subclasses *inherit* interface implementations

```
public interface IDriveable {
   void turn (int dir, int amt);
   void accelerate (int amt);
   void applyBrake (int amt);
   void startEngine ( );
   void shift (int newGear);
   double getFuelLevel ( );
}
```

```
public class Vehicle implements IDriveable {
   public void turn(int dir, int amt){...}
   public void accelerate (int amt) {...}
   public void applyBrake (int amt) {...}
   public void startEngine( ) {...}
   public void shift (int newGear) {...}
   public double getFuelLevel ( ) {...}
}
```

```
public class Car extends Vehicle {
   public void applyBrake (int amt) {...}
   public void startEngine ( ) {...}
   public void shift (int newGear) {...}
   public double getFuelLevel( ) {...}
   // Car doesn't need to specify "turn()" or "accelerate()"
   // because they are inherited from Vehicle
}
```





# "Interfaces" In C++

- **"Abstract" Methods:**

```
virtual void turn (int direction, int amount) = 0 ;
```

- **"Abstract" Classes :**

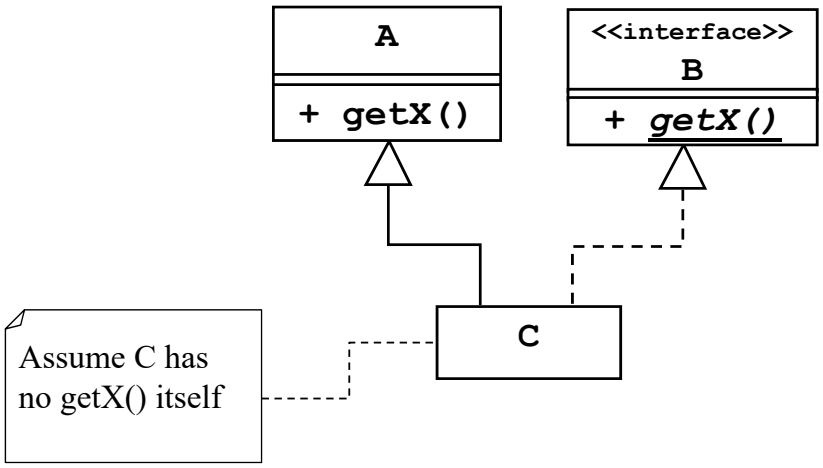
```
class IDriveable {
  public:
        virtual void turn (int direction, int amount) = 0 ;
        virtual void accelerate (int amount) = 0 ;
        ...
};
```

- **"Abstract" Classes as Interfaces :**

```
class Vehicle { ... };
class Car : public IDriveable, Vehicle
{ ... };
```

# Quiz:

Which **`getX()`** is called in objects of type C?








# *Predefined* Interfaces in CN1

- **Many CN1 classes implement built-in interfaces**
- **User classes can also implement them**

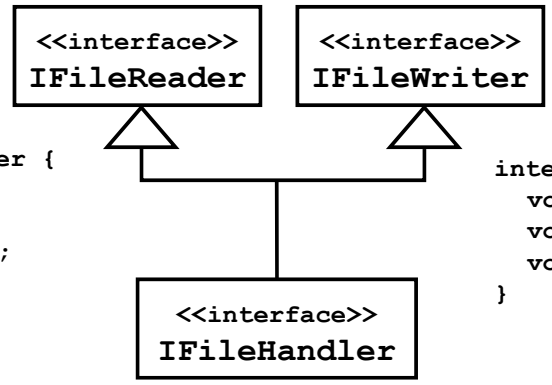
## Examples:

```
interface Shape {
  boolean contains(int x, int y);
  Rectangle getBounds();
  Shape intersection(Rectangle rect);
  //other methods...
}


interface Comparable {
  int compareTo (Object otherObj);
}
```

# Interface Hierarchies

## Interfaces can *extend* other interfaces




```
interface IFileReader {
  byte readByte();
  int readInt();
  String readLine();
}
```

```
interface IFileWriter {
  void writeByte (byte b);
  void writeInt (int theInt);
  void writeString (String s);
}
```

```
interface IFileHandler extends IFileReader, IFileWriter {
  void open (String filename);
  void close ( );
}
```

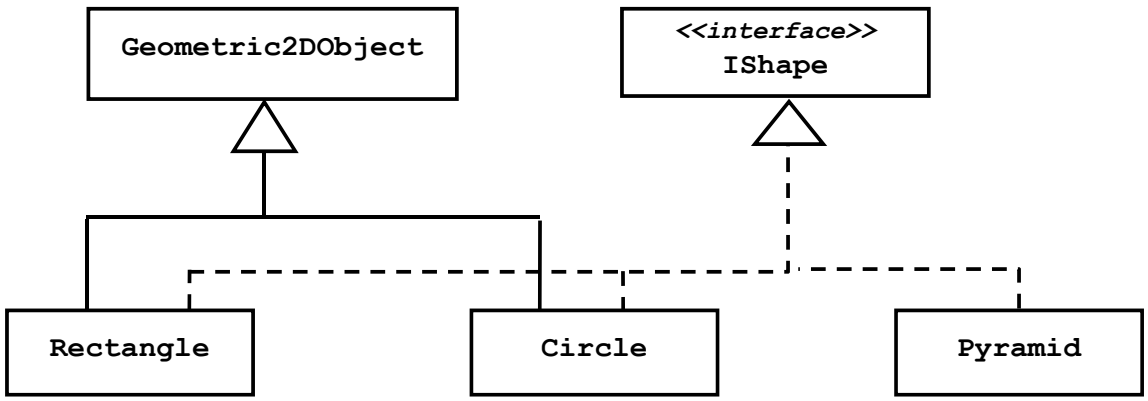




# Interface Subtypes

**If a class implements an interface, it is considered a "subtype" of the "interface type":**

- **A Circle "IS-A" Geometric2DObject**
- **A Circle "IS-A" IShape**



# Interface Subtypes (cont.)

- **Objects can be upcast to *interface types:***

```
Circle myCircle = new Circle();
IShape myShape = (IShape) myCircle ;
```

- **Interfaces, like superclasses, provide objects with:**

  *"apparent type"* vs. *"actual type"*

- ***Variable of interface type, like superclass type, can hold many different types of objects!***





# Interfaces and Polymorphism

- *Apparent* type = What does it look like at a particular place in program (changes).

Determines: What methods may be invoked

- *Actual* type = What was it created from (never changes)

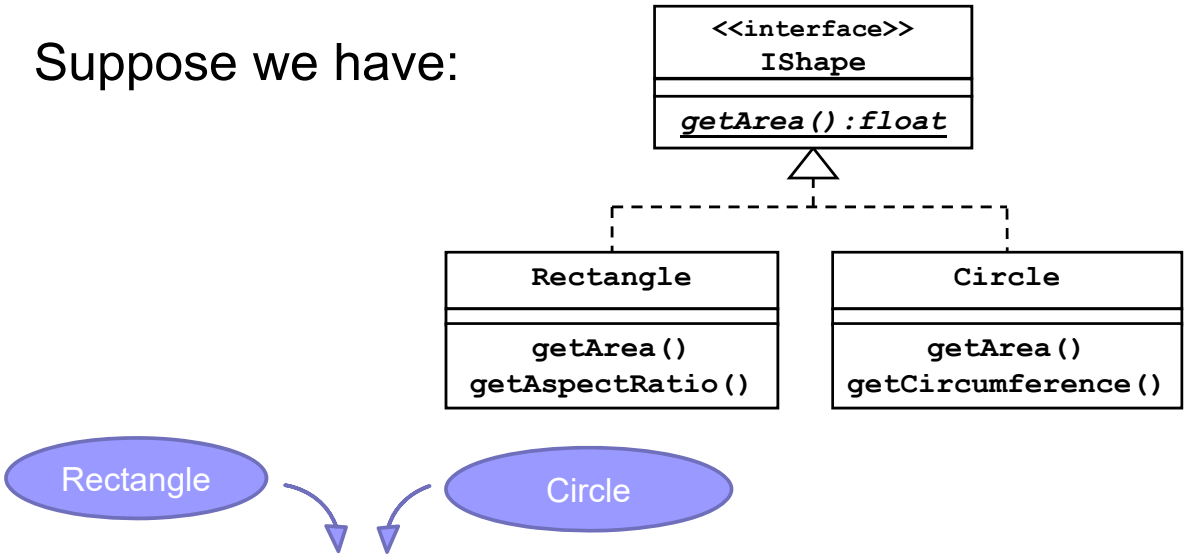
Determines: Which implementation to call when the method is invoked

```
IShape [ ] myThings = new IShape [10] ;
myThings[0] = new Rectangle();
myThings[1] = new Circle();
//...code here to add more rectangles, circles, or other "shapes"

for (int i=0; i<myThings.length; i++) {
  IShape nextThing = myThings[i];
  process ( nextThing );
}
...
void process (IShape aShape) {
  // code here to process a IShape object, making calls to IShape methods.
  // Note this code only knows the apparent type, and only IShape methods
  // are visible - but any methods invoked are those of the actual type.
}
```

# Interface Polymorphism Example

- Suppose we have:



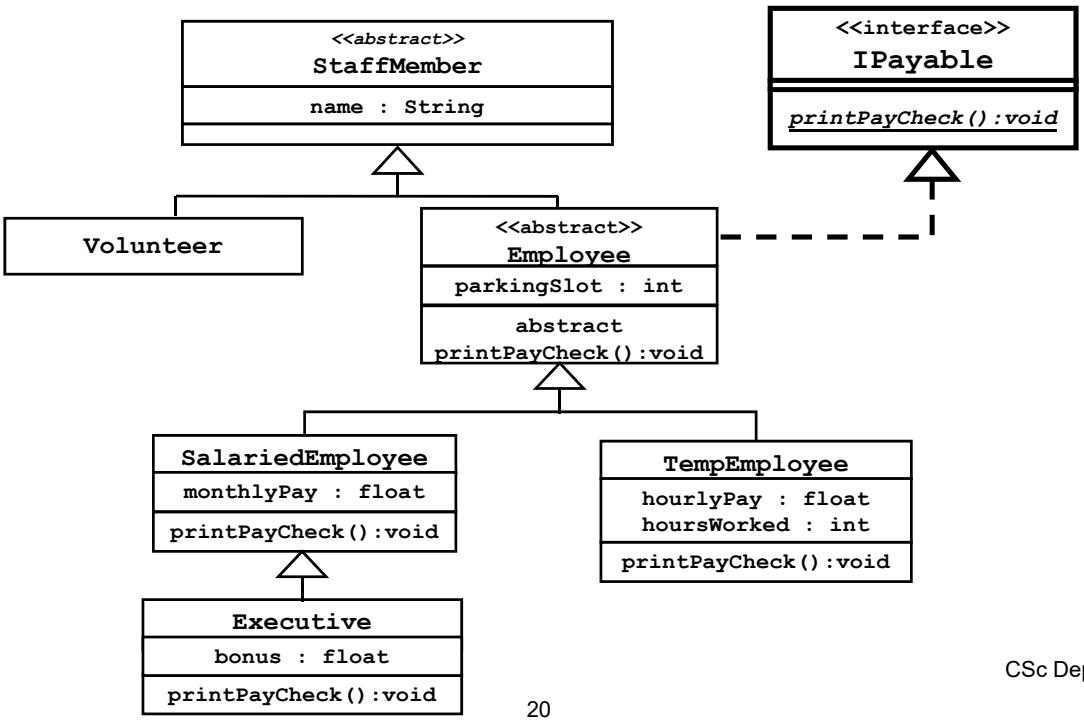

```
void process (IShape s) {
  ...
  s.getArea();              //legal; all IShapes have getArea()
  ...
  s.getAspectRatio();       //illegal,  even if 's' is a Rectangle
                            //  (generates a compiler error)
}
```





# Polymorphic Safety Revisited

- StaffMember hierarchy using Interfaces:



# Interface Polymorphic Safety

```
public class StaffMember {
    ...
}

public interface IPayable {
    public void printPayCheck() ;
}

//Every kind of "Employee" IS-A "payable" (must provide printPayCheck())
public abstract class Employee extends StaffMember implements IPayable {
    ...
    abstract public void printPayCheck() ;
}
```

---

```
//client using interface polymorphism to safely print paychecks:
for (int i=0; i<staffList.length; i++) {
    if (staffList[i] instanceof IPayable)
        ((IPayable)staffList[i]).printPayCheck() ;
}
```







# Abstract Classes vs. Interfaces

```
abstract class Animal {
  abstract void talk();
}

class Dog extends Animal {
  void talk() {
    System.out.println("Woof!");
  }
}

class Cat extends Animal {
  void talk() {
    System.out.println("Meow!");
  }
}
```

```
class Example {
  ...
  Animal animal = new Dog();
  Interrogator.makeItTalk(animal);
  animal = new Cat();
  Interrogator.makeItTalk(animal);
  ...
}
```

```
class Interrogator {
 static void
    makeItTalk(Animal subject) {
      subject.talk();
    }
}
```

Example after Bill Venners, www.javaworld.com

# Abstract Classes vs. Interfaces (cont.)

- We can easily add a Bird and "make it talk":

```
class Bird extends Animal {
  void talk() {
    System.out.println("Tweet! Tweet!");
  }
}
```

- Making a CuckooClock "talk" is a problem:

```
class Clock {... }
class CuckooClock extends Clock {
  void talk() {
    System.out.println("Cuckoo! Cuckoo!");
  }
}
```

*We can't pass a CuckooClock to Interrogator – it's not an animal.*
*And it is illegal (in Java) to also extend animal (can only "extend" once!)*





# Abstract Classes vs. Interfaces (cont.)

*The interface of an abstract class can be separated:*

```
interface ITalkative {
  void talk();
}

abstract class Animal implements ITalkative {
  abstract void talk();
}

class Dog extends Animal {
  void talk() { System.out.println("Woof!"); }
}

class Cat extends Animal {
  void talk() { System.out.println("Meow!"); }
}
```

# Abstract Classes vs. Interfaces (cont.)

Use of interfaces can *increase Polymorphism:*

```
class CuckooClock extends Clock implements ITalkative {
  void talk() {
    System.out.println("Cuckoo! Cuckoo!");
  }
}

class Interrogator {
  static void makeItTalk(ITalkative subject) {
      subject.talk();
    }
}
```

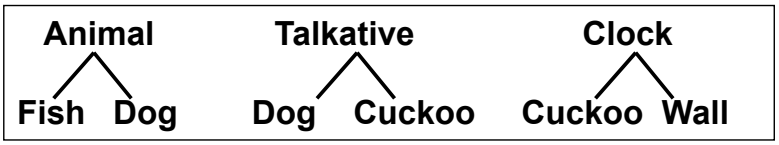
*Now we can pass a CuckooClock to an Interrogator!*





# Abstract Classes vs. Interfaces (cont.)

Interfaces allow for *multiple hierarchies*:




```
interface ITalkative {
  void talk();
}

abstract class Animal {
  abstract void move();
}

class Fish extends Animal { // not talkative!
  void move() { //code here for swimming }
}

class Dog extends Animal implements ITalkative {
  void talk() { System.out.println("Woof!"); }
  void move() { //code here for walking/running }
}

class CuckooClock extends Clock implements ITalkative {
  void talk() { System.out.println("Cuckoo!"); }
}
```

# Abstract Class vs. Interface: Which?

Abstract classes are a good choice when:

- There is a clear *inheritance hierarchy* to be defined (e.g. "kinds of animals")
- We need non-public, non-static, or non-final fields OR private or protected methods
- Before Java 8:
  - There are at least *some* concrete methods shared between subclasses
  - We need to *add new methods* in the future (adding concrete methods to an abstract class does NOT break its subclasses)





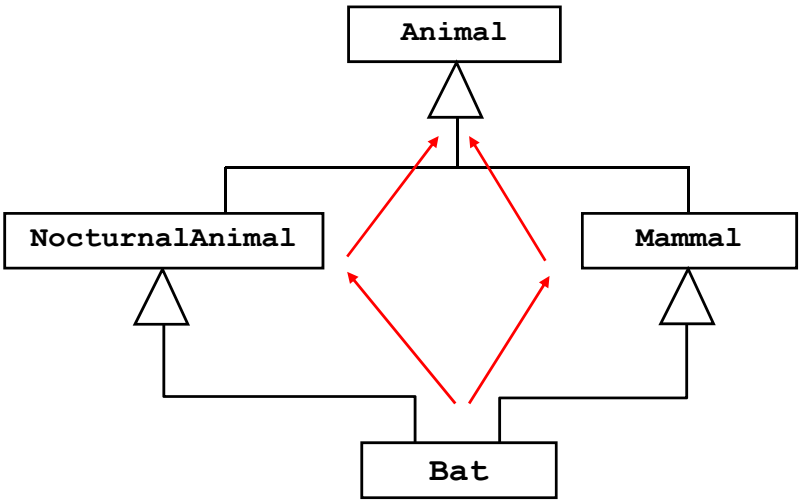
# Abstract Class vs. Interface: Which?

Interfaces are a good choice when:

- The relationship between the methods and the implementing class is not extremely strong
  - Example: many classes implement "Comparable" or "Cloneable"; these concepts are not tied to a specific class
- Before Java 8:
  - An API is likely to be stable (again: adding interface methods *breaks implementing classes)*
- Something like Multiple Inheritance is desired

(see next slides…)

# Multiple Inheritance Revisited



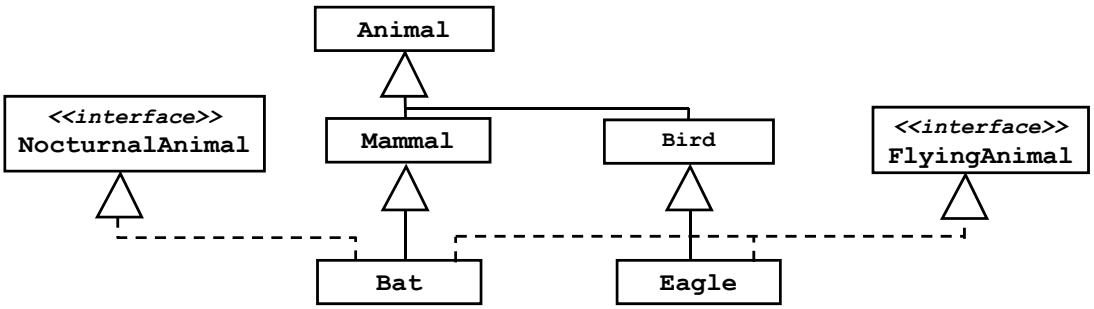

**A possible alternative Animal Hierarchy**





# Multiple Inheritance via *Interfaces*

**Can say this exactly in Java:**




```
public class Animal {...}
public class Mammal extends Animal {...}
public interface NocturnalAnimal {...}
public class Bat extends Mammal implements NocturnalAnimal {...}
```

**and more:**

```
public interface FlyingAnimal {...}
public class Bat extends Mammal implements NocturnalAnimal,
                            FlyingAnimal  {...}
```

# Does Java support multiple inheritance?

- Of interfaces – Yes
- Of implementations – No (before Java 8)