CSC 133
Object-Oriented Computer Graphics Programming

OOP Concepts III – Abstraction

Dr. Kin Chung Kwan

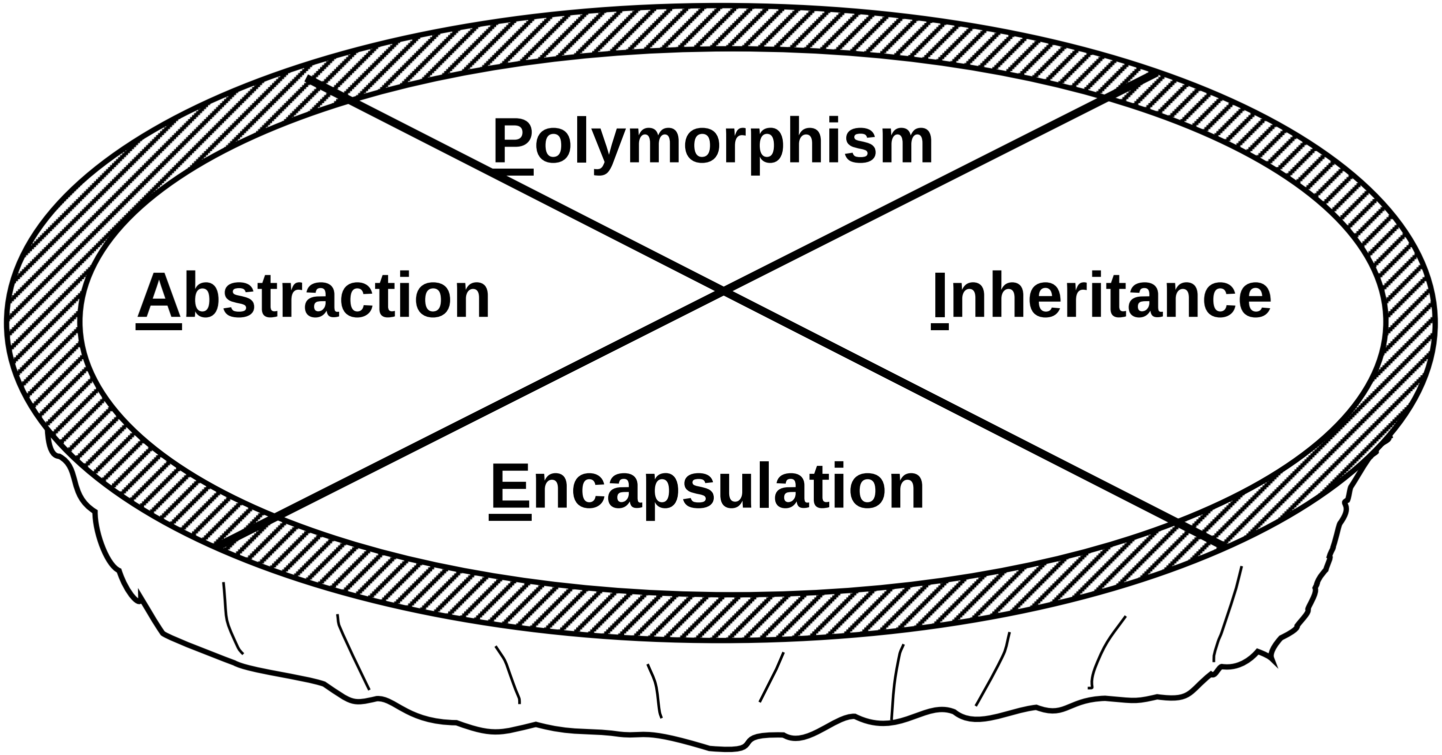
Spring 2023

Computer Science Department
California State University, Sacramento

SACRAMENTO STATE

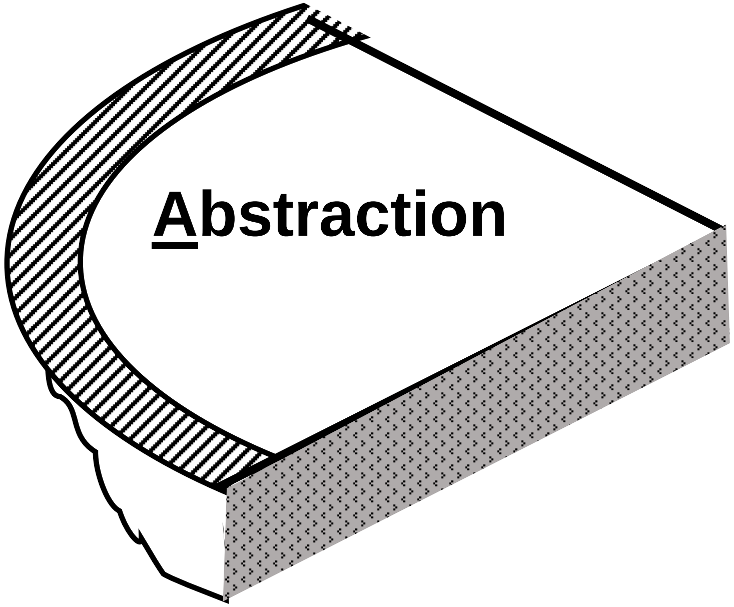
“A Pie”

Four distinct OOP Concepts (or Pillars)



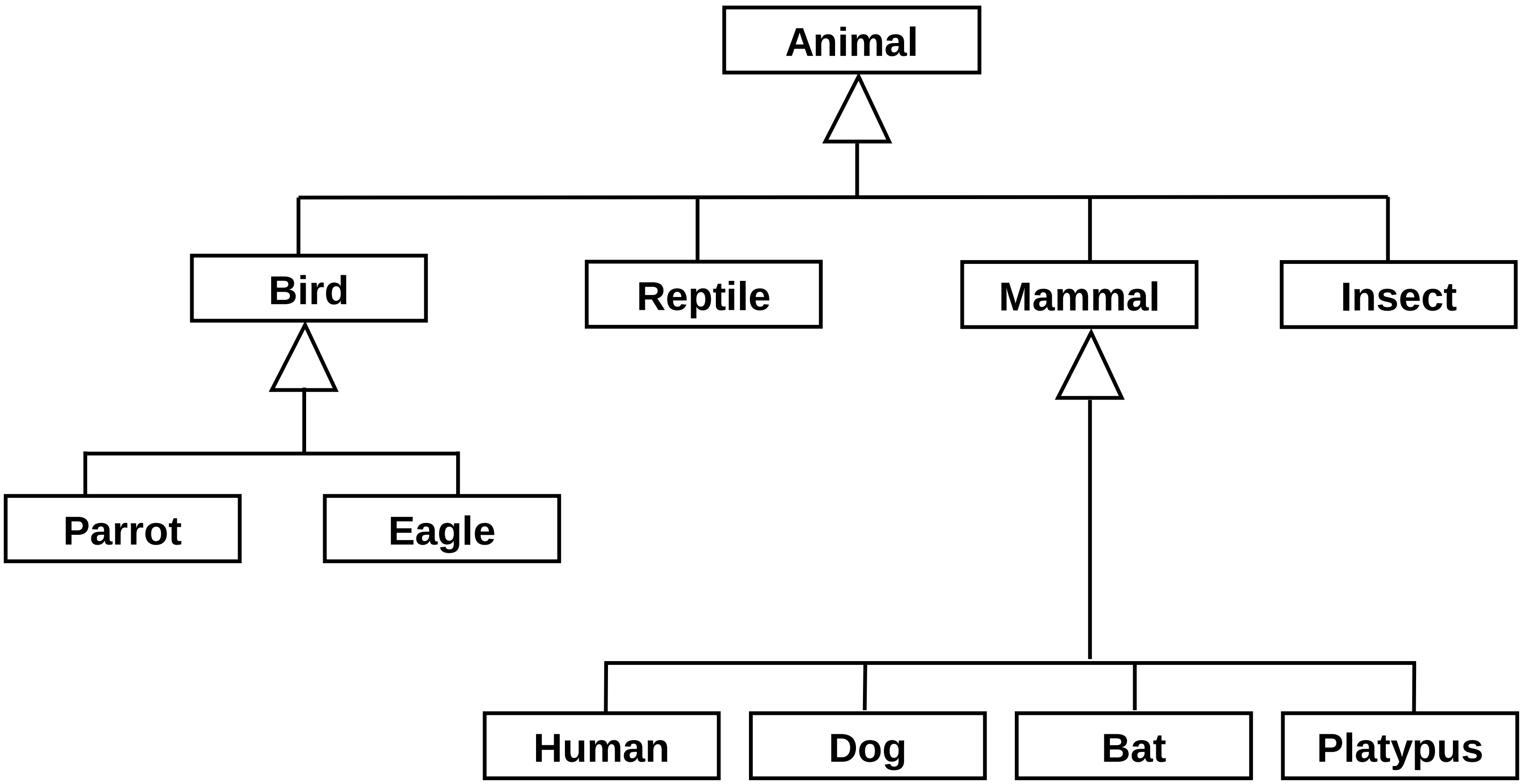
Last Lecture

We ate two more parts



Abstraction

Inheritance hierarchy



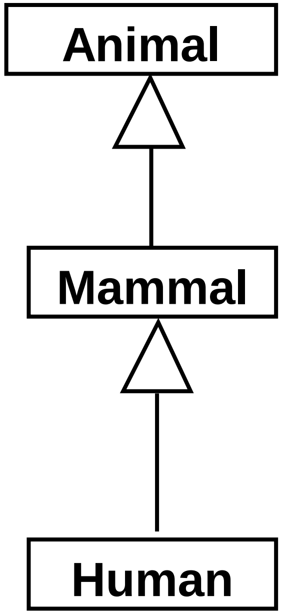
Behaviour?

We know

- human can move
- Mammal can move
- Animal can move

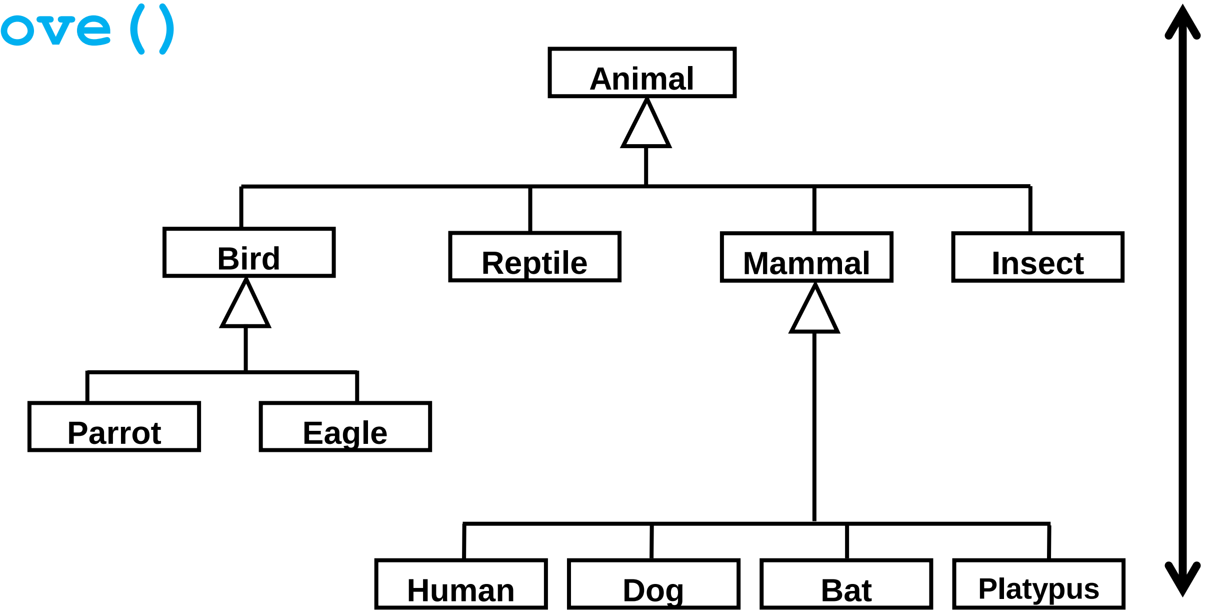
But how?

- Human can move their arm, leg, or head
- Mammal?
- Animal?



Class Level

- Some classes will never logically be instantiated
  - `Animal`, `Mammal`, …
- Some methods cannot be "specified" completely at a given class level
  - `move()`



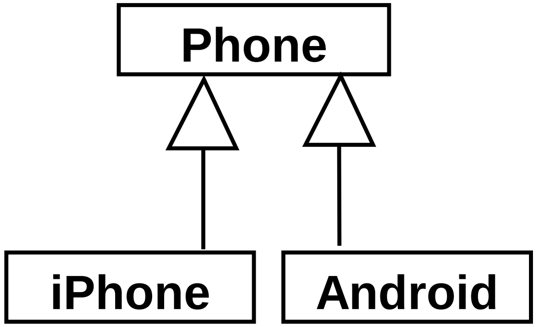
General, abstract

Specific, detail

Cannot Define It

Know it can do, but cannot define how

- When parent class is a general concept



But we need it for polymorphism

```
For all phone
      phone.installOS();
```

Solution?

Empty method

```
    public void installOS() { }
```

Works, but unclear and unsafe

- Other people think you forget to implement it
- We may forget to override it but no compile error
- Want a reminder

Abstract Class and Method

- Use the keyword `abstract`
- Both classes and methods can be declared abstract in Java:

```
public abstract class Animal {
    public abstract void move ();
}
```

- End by semi-colon instead of function body {…}
  - Do not need to implement it

Abstract vs Concrete

Abstract

- With keyword
- No function body
- End with ;

```
public abstract void move();
```

Concrete

- Non-abstract
- With function body
- End with }

```
public void move(){
        x += 1;
        y += 1;
}
```

Inheritance for Specification

The third usage of inheritance:

- Parents declared the behaviours without implementation
- Child now implement it.

Inherit from Abstract

Abstract classes cannot be instantiated

```
Animal a = new Animal(); ✗
```

But can be inherited

```
public mammal extends Animal {
      public void move () {...}
}
```

Then override the abstract method.

Abstract vs Modifier

- `static`
- `final`
- `private`
  - cannot be declared abstract
  - No way to override or change them
  - No way to provide a “specification”

- `public`
- `Protected`
  - can be declared abstract.

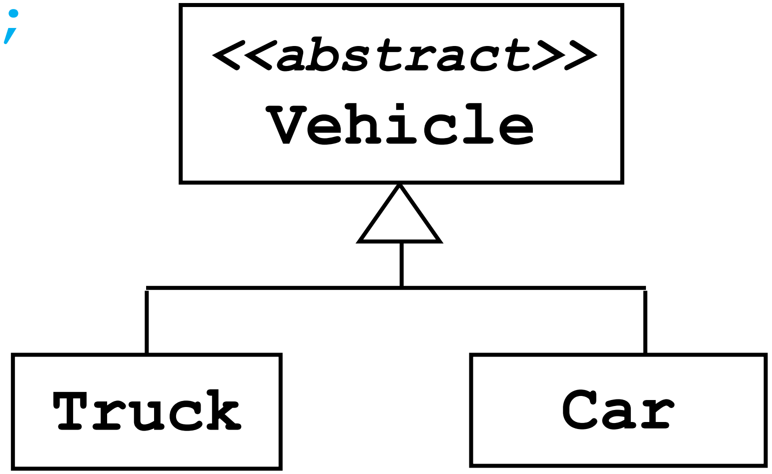
Requirements

- If a class contains an abstract method
  - This class is an abstract class
  - This class must have keyword `abstract`
- Abstract classes can contain concrete methods
- If a child does not implement every abstract methods from parent
  - The child must have keyword `abstract` too

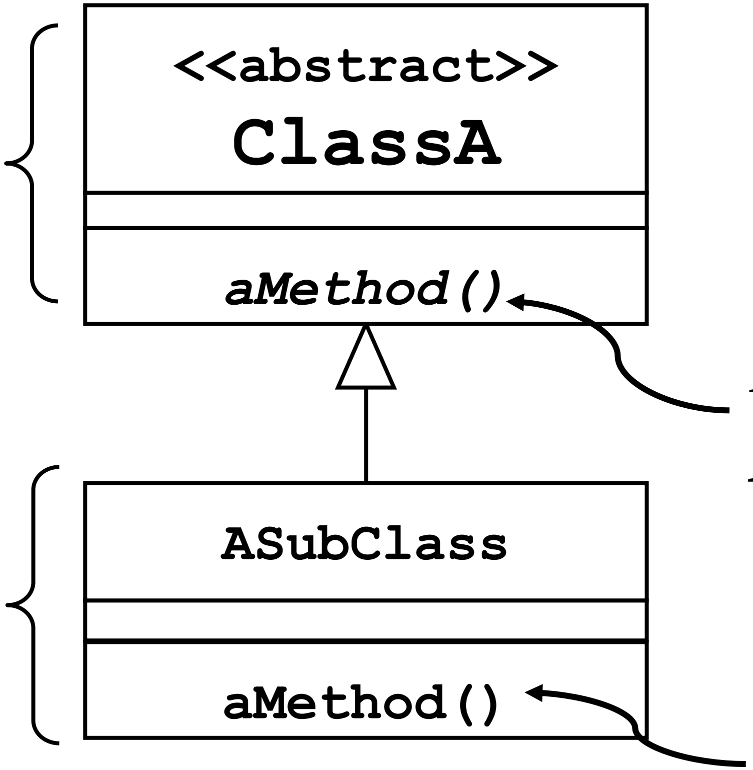
Abstract Class Reference

Reference to abstract type

```
Vehicle v ;
Truck t = new Truck();
Car c = new Car();
...
v = t ;
...
v = c ;
```



UML for Abstraction



<<abstract>>

ClassA

aMethod()

ASubClass

aMethod()

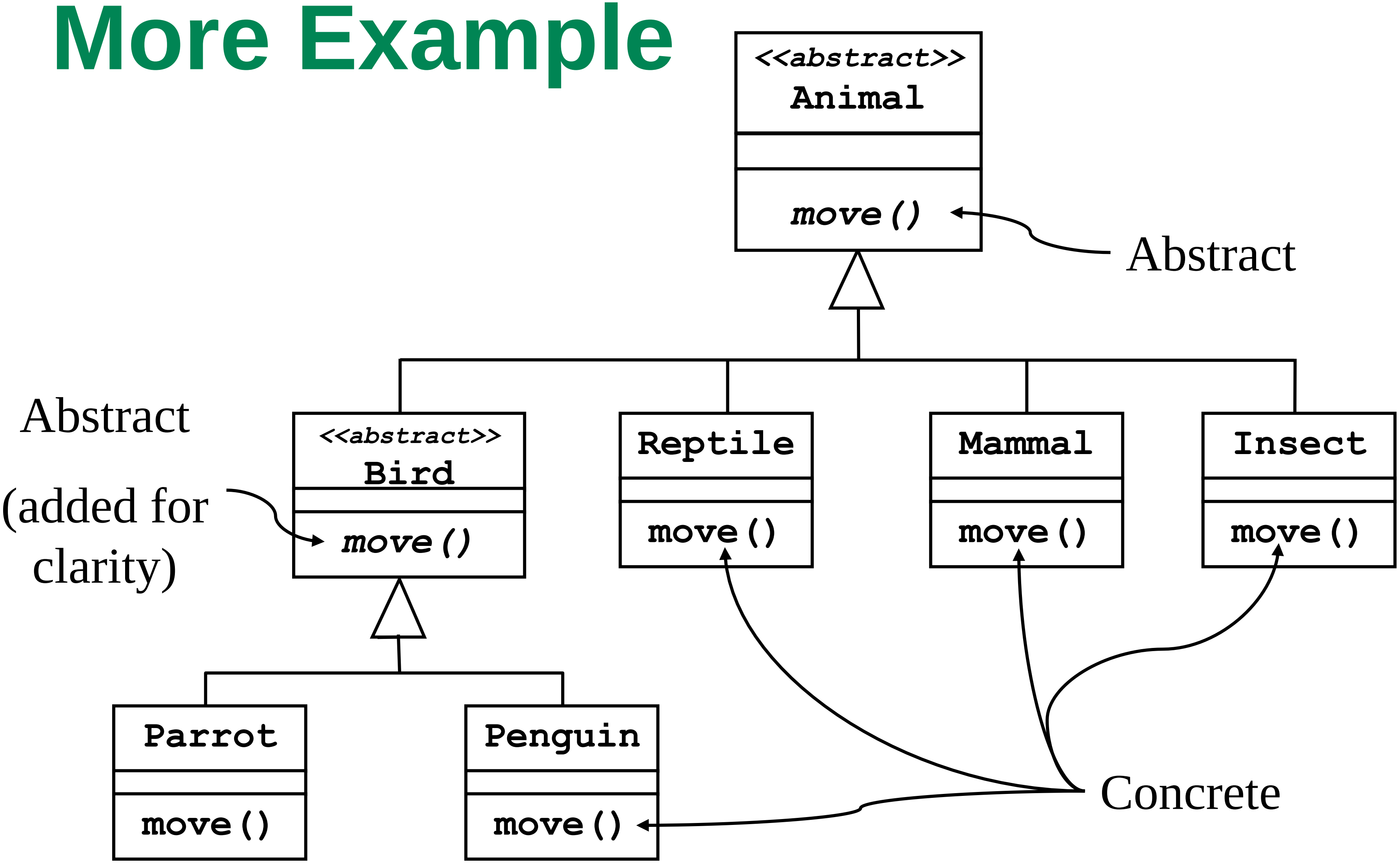
Abstract class

Concrete class

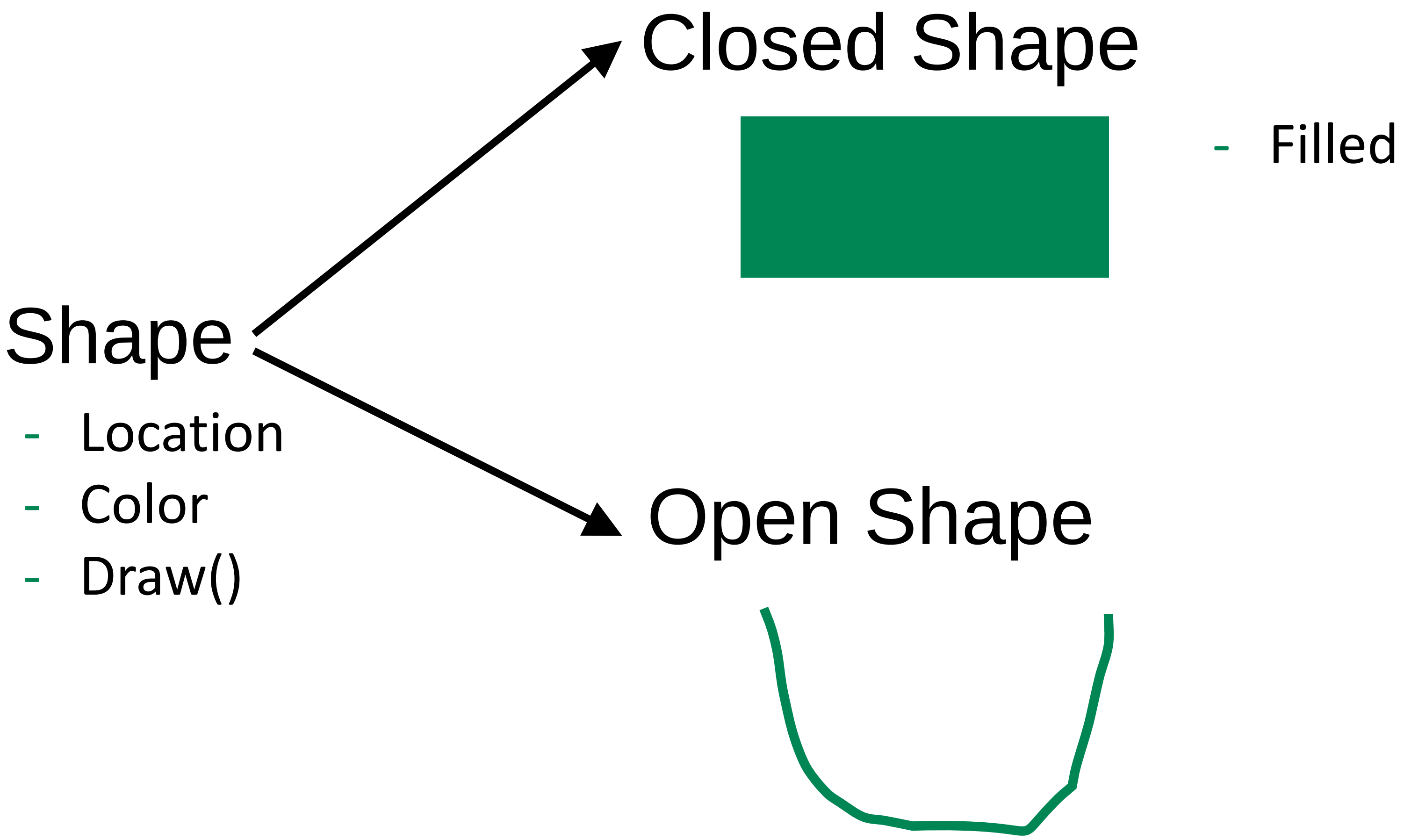
Abstract method: ***Italics***

Implementation

More Example



Example: Shapes



Shape

- Location
- Color
- Draw()

Closed Shape

- Filled

Open Shape

Example: Shape class

```
public abstract class Shape {
  private int color;
  private Point location;
  public Shape()   {
    color = ColorUtil.rgb(0,0,0);
    location = new Point (0,0);
  }
  public Point getLocation() { return location;  }
  public int getColor() { return color; }
  public void setLocation (Point newLoc) {
    location = newLoc;
  }
  public void setColor (int newColor) {
    color = newColor;
  }
  public abstract void draw(Graphics g);
}
```

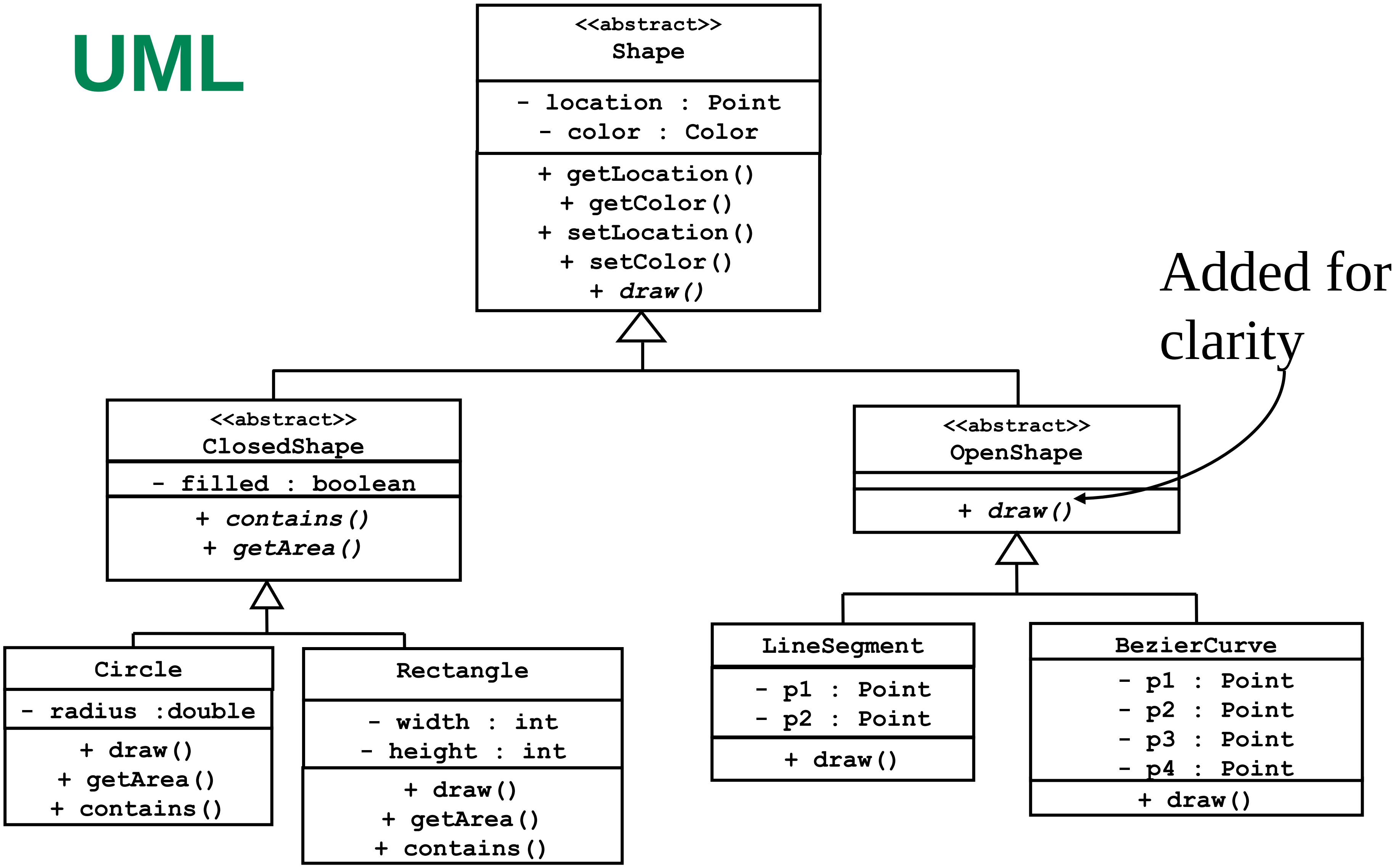
Example: ClosedShape

```
public abstract class ClosedShape extends Shape {
 private boolean filled;   // attribute common
 public ClosedShape() {
  filled = false;
 }
 public ClosedShape(boolean filled) {
  this.filled = filled;
 }
 public boolean isFilled() { return filled;}
 public void setIsFilled(boolean filled) {
  this.filled = filled;
 }
 public abstract boolean contains(Point p);
 public abstract double getArea();
}
```

Example: Rectangle

```
public class Rectangle extends ClosedShape {
  private int width;
  private int height;
  public Rectangle() {
    super(true);
    width = 2;
    height = 1;
  }
  public boolean contains(Point p) { ... }
  public double getArea() {
    return (double) (width * height) ;
  }
  public void draw (Graphics g) {
    if (isFilled()) {
      //  code here to draw a filled (solid) rectangle
    } else {
      //  code here to draw a lined rectangle
    }
  }
}
```

UML



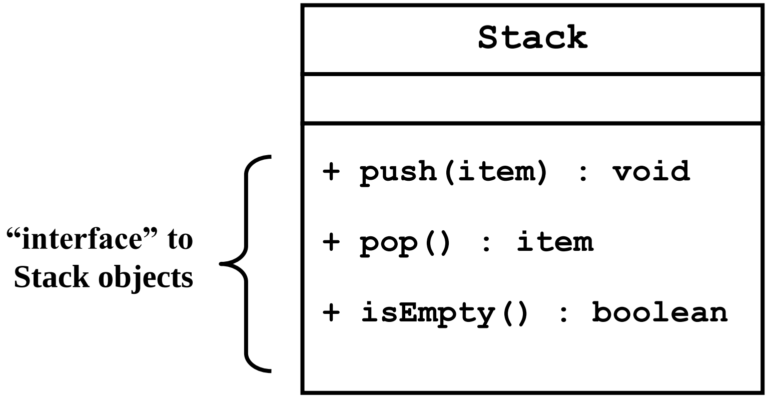
<<abstract>>
Shape
- location : Point
- color : Color
+ getLocation()
+ getColor()
+ setLocation()
+ setColor()
+ *draw()*
<<abstract>>
ClosedShape
- filled : boolean
+ *contains()*
+ *getArea()*
<<abstract>>
OpenShape
+ *draw()*
Added for clarity
Circle
- radius :double
+ draw()
+ getArea()
+ contains()
Rectangle
- width : int
- height : int
+ draw()
+ getArea()
+ contains()
LineSegment
- p1 : Point
- p2 : Point
+ draw()
BezierCurve
- p1 : Point
- p2 : Point
- p3 : Point
- p4 : Point
+ draw()

Interface

Class Interface

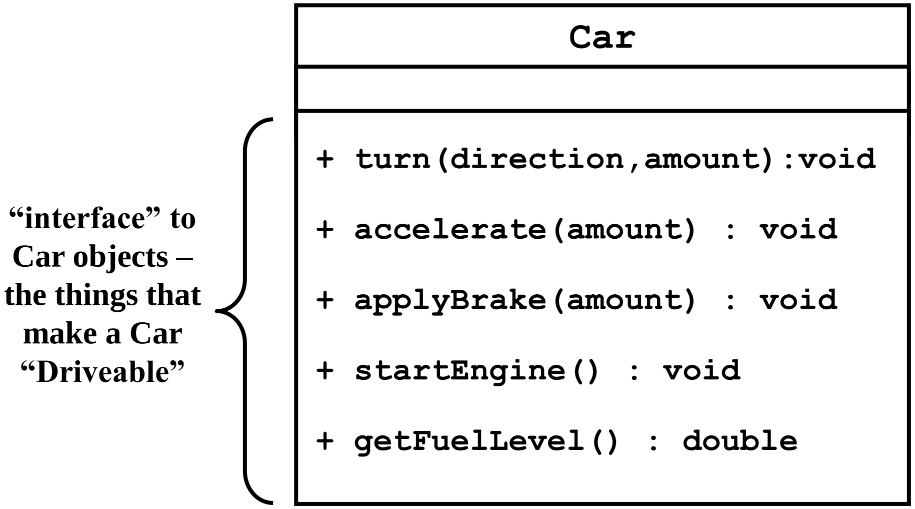
A special class that define behaviors

- Without any implementation
- Defines a set of methods with specific signatures



“interface” to Stack objects

```
Stack

+ push(item) : void
+ pop() : item
+ isEmpty() : boolean
```



“interface” to Car objects – the things that make a Car “Driveable”

```
Car

+ turn(direction,amount):void
+ accelerate(amount) : void
+ applyBrake(amount) : void
+ startEngine() : void
+ getFuelLevel() : double
```

Java Interface Characteristic

- All methods must be public
  - Default visibility
- Usually, no method implementation
  - After Java 8, default or class methods can have body in interface
- All fields must be public and static and final
  - Public constant variables
  - Default

Java Interface Construct

Keyword `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

Use keyword `implement`:

```
public class Car implements IDriveable {
    public void accelerate (int amount){...}
    public void applyBrake (int amount){...}
    ...
}
```

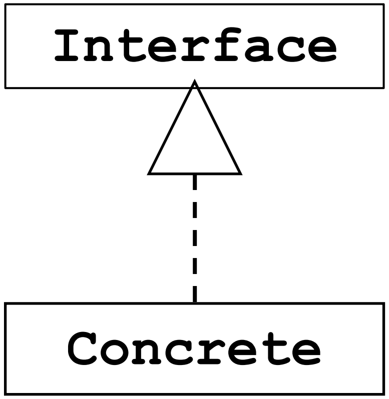
We must provide bodies for all methods

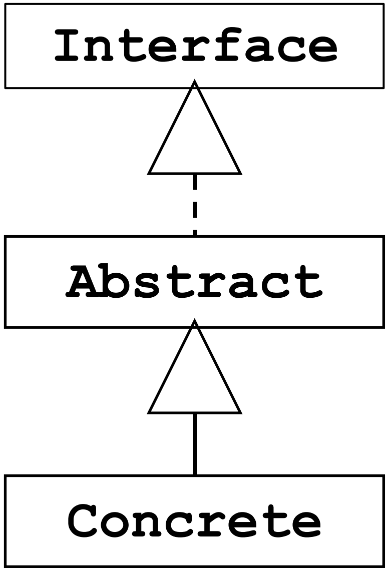
- Compiler checks!

Provide Bodies

You can provide all function bodies in the child class of the interface.

- Or set the child as an abstract class





Interface Relationship

“is”

- No “a” as it usually follow an adjective.

Inheritance: is-a

- Car is a vehicle

Interface: is

- Car is driveable

Different implementation

Different class can have different implementation for the same interface

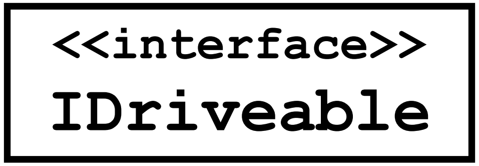
- Brake for `car` and `truck`

```
interface IDrivable{
   public void brake();
}
```

```
class car implements IDrivable{
   public void brake(){
      //short time...
   }
}
```

```
class truck implements IDrivable{
   public void brake(){
      //longer time...
   }
}
```

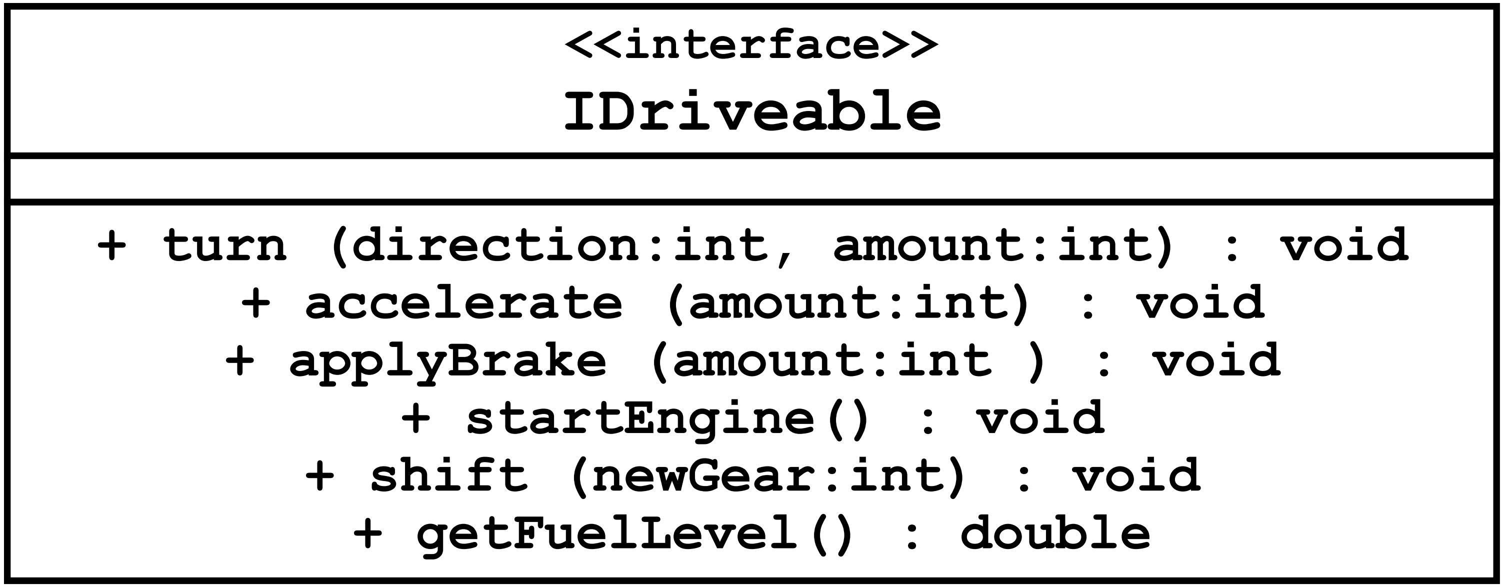
UML Interface Notation



<<interface>>

IDriveable

or



<<interface>>

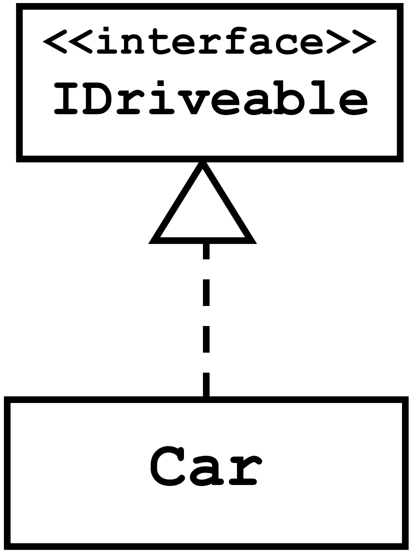
IDriveable

+ turn (direction:int, amount:int) : void
+ accelerate (amount:int) : void
+ applyBrake (amount:int) : void
+ startEngine() : void
+ shift (newGear:int) : void
+ getFuelLevel() : double

UML Interface Relationship

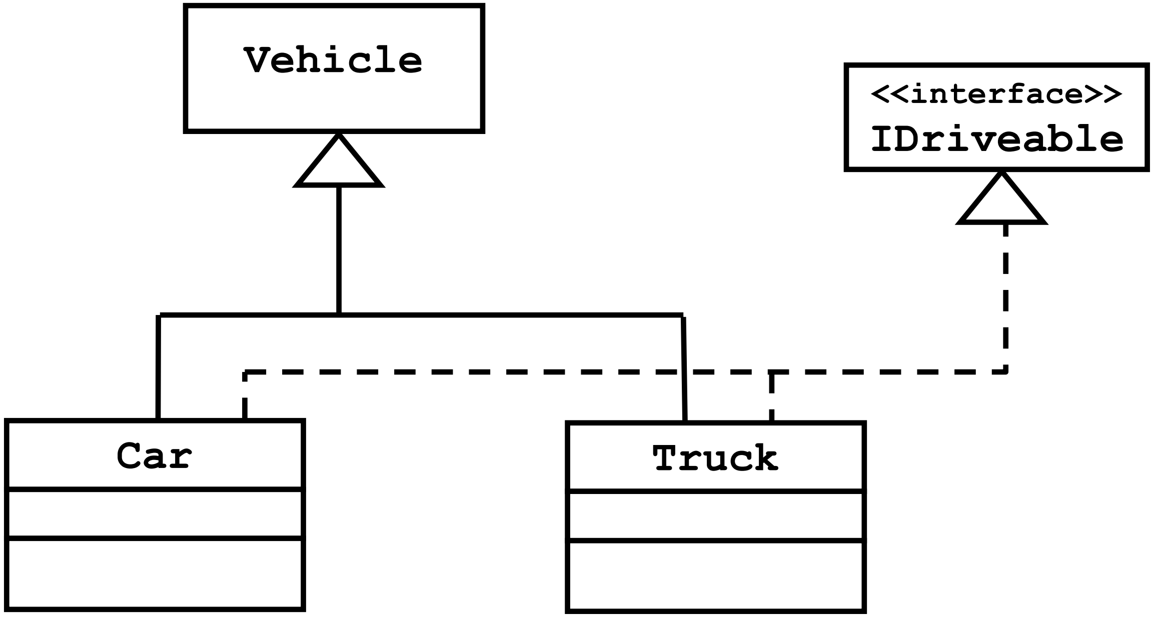
Class `Car` implements interface `Idriveable`

- Dotted arrow
- Same arrowhead as for inheritance



Multiple Relationship

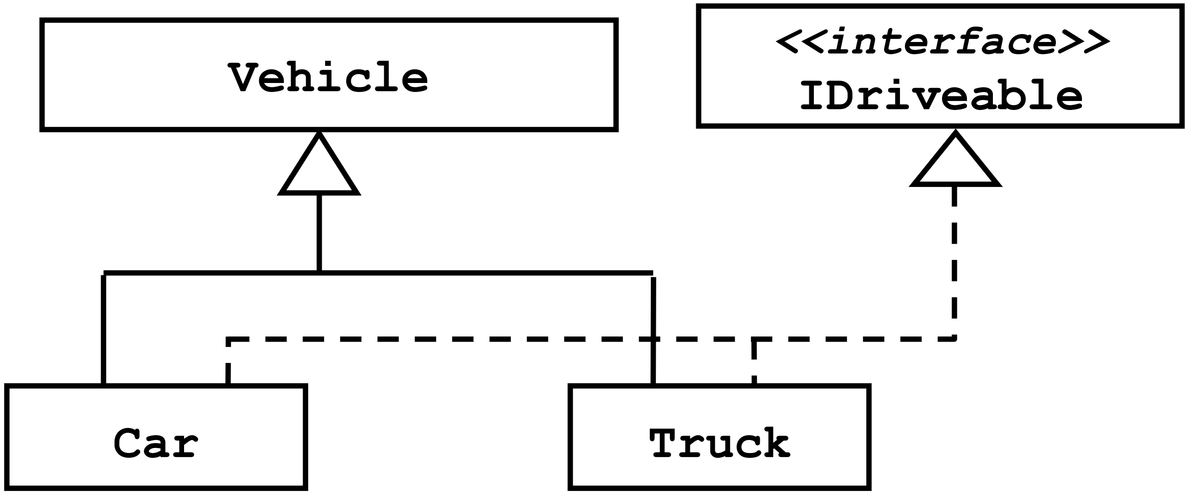
- Car and Truck can both
  - Derive from Vehicle
  - Implement IDriveable



Interface Subtypes

If a class implements an interface

- considered a “subtype” of the “interface type”:
- `Car` and `truck` subtype of IDriveable



Interface Inheritance

- Subclasses inherit interface implementations

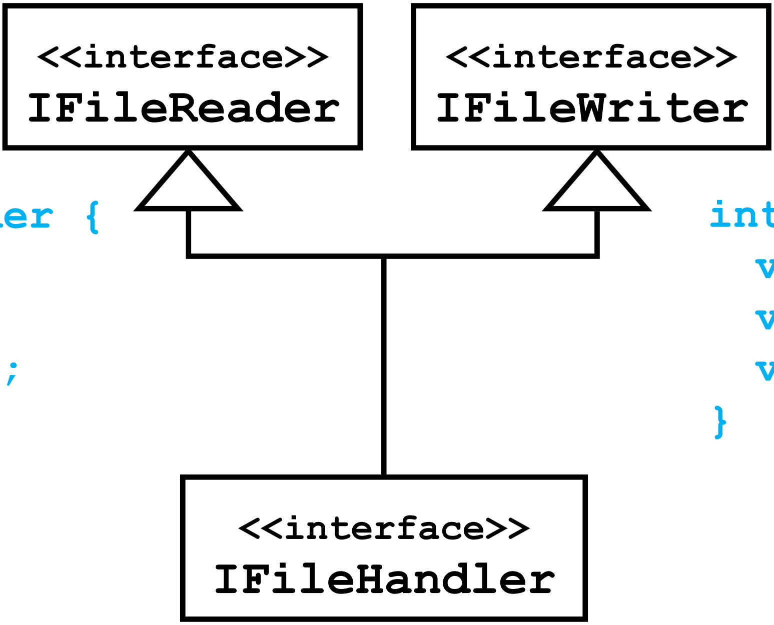
```
public interface IDriveable {
  void turn (int dir, int amt);
  void accelerate (int amt);
  void applyBrake (int amt);
  void startEngine ( );
  void shift (int newGear);
  double getFuelLevel ( );
}
```

```
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
}
```

Interface Hierarchies

Interfaces can extend other interfaces



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

```
interface IFileWriter {
  void writeByte ();
  void writeInt ();
  void writeString ();
}
```

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

Problem

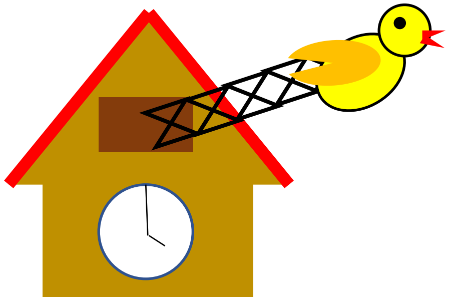
```
abstract class Animal {
  abstract void talk();
}

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

Reuse same function



- Bird is an animal it can talk:

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

- CuckooClock is a clock, it can talk too

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

- Cannot pass a CuckooClock to Interrogator – it's not an animal.

```
makeItTalk(Animal subject) { ... }
```

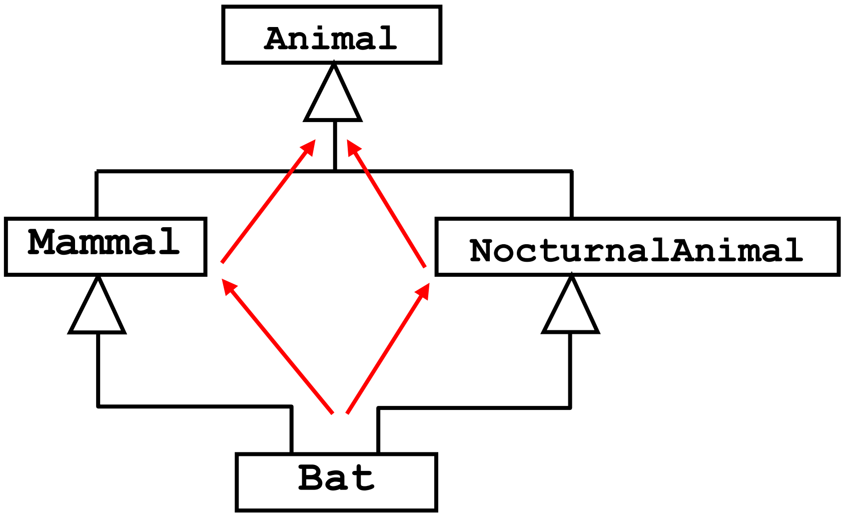
Multiple Inheritance?

Can we extend multiple class?

```
class CuckooClock extends Clock, Animal
```

Java did not support

And CuckooClock is not an animal!



A possible alternative Animal Hierarchy

Solution

Separate the interface and the abstract class

```
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

Use of interfaces can *increase Polymorphism:*

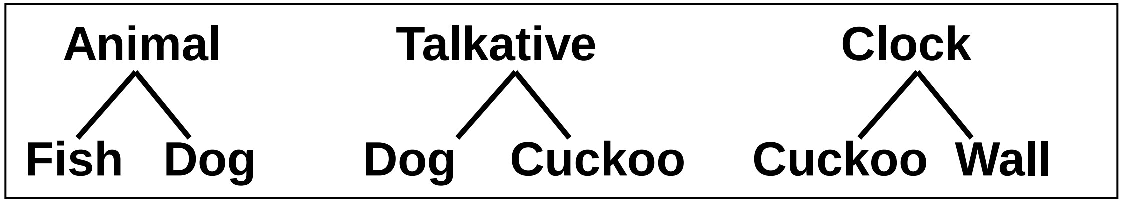
```
class CuckooClock extends Clock implements ITalkative {
  void talk() {
    System.out.println(“Cuckoo! Cuckoo!”);
  }
}
```

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

Now we can pass a CuckooClock to an Interrogator!

Interfaces for multiple hierarchies

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

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



Animal
Fish Dog

Talkative
Dog Cuckoo

Clock
Cuckoo Wall

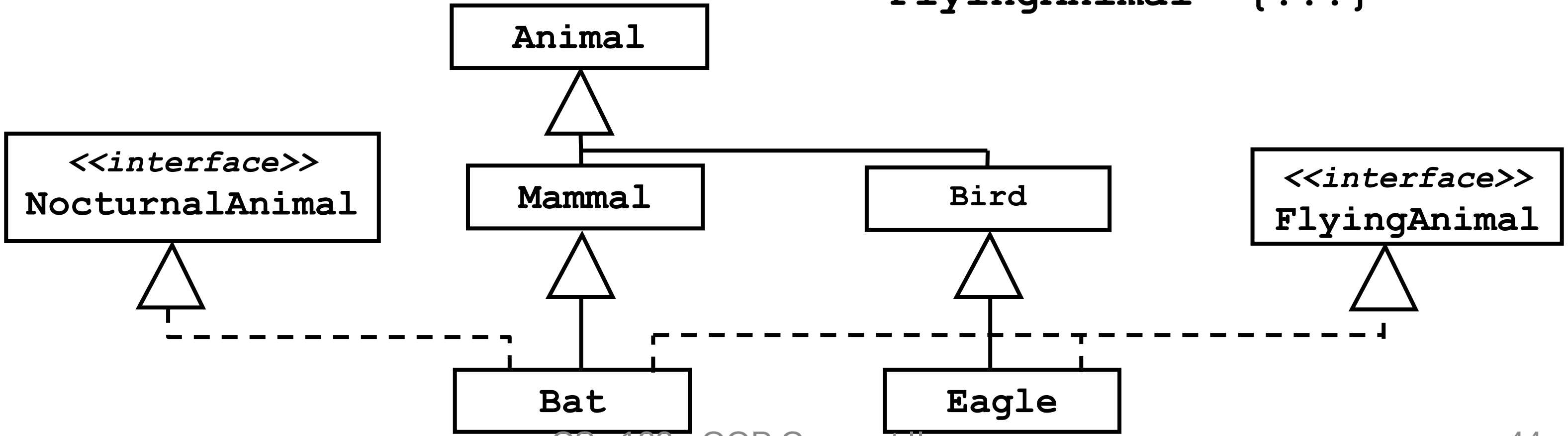
```
class Fish extends Animal { // not talkative!
  void move() { //code here for swimming }
}

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!”); }
}
```

Multiple Inheritance via Interfaces

```
  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  {...}
```



Abstract Class?

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)

Or Interface?

Interfaces are a good choice when:

- A class is an imagine concepts
  - Example: many classes implement “Comparable” or “Cloneable”; these concepts are not tied to a specific class
- Something like Multiple Inheritance is desired

Reference Type

1. Apparent type

- What does it look like at a particular place in program (changes).
- Reference type
- Determines what methods can be called

2. Actual type

- What was it created from (never changes)
- Determines which implementation to call

1

```
StaffMember [ ] staffList = new StaffMember[6];
staffList[0] = new SalariedEmployee (“Sam”);
```

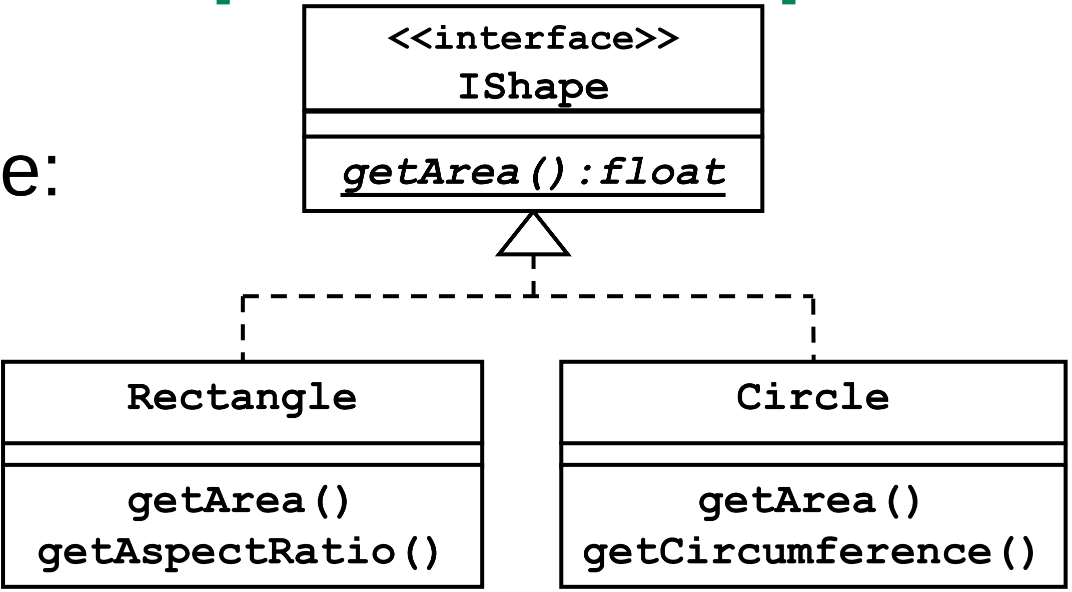
2

Interfaces and Polymorphism

```
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:



<<interface>>
IShape

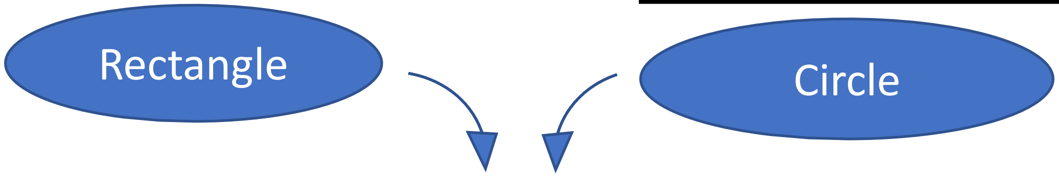
getArea():float

Rectangle

getArea()
getAspectRatio()

Circle

getArea()
getCircumference()



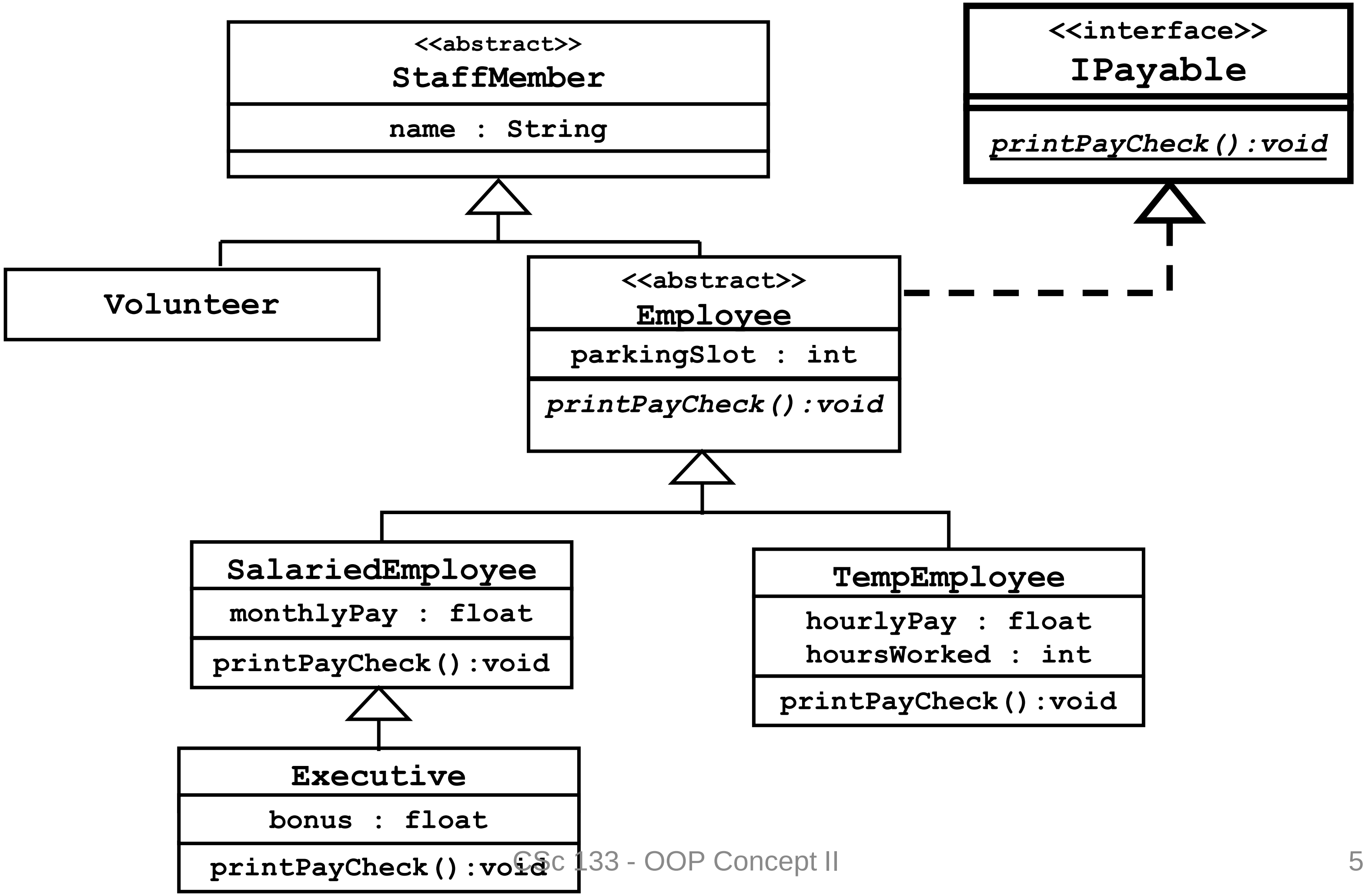
Rectangle

Circle

```
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() ;
}
```

CN1 Predefined Interfaces

- CN1 provide built-in interfaces class
  - You 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);
}
```

Any Questions?