CSc 133 Lecture Notes

4 - Inheritance

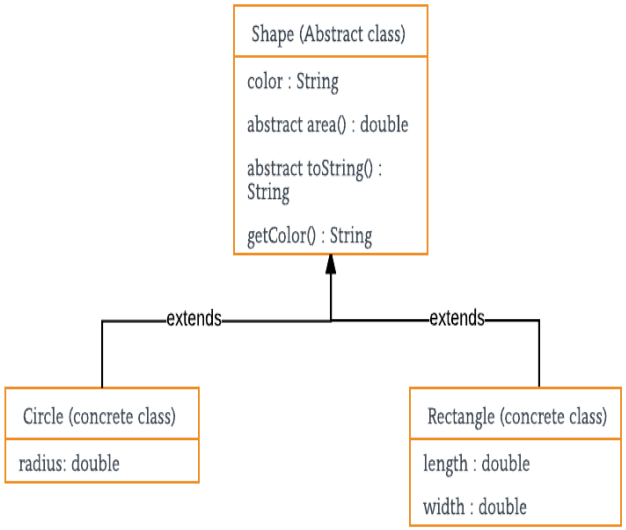
Computer Science Department
California State University, Sacramento

Overview

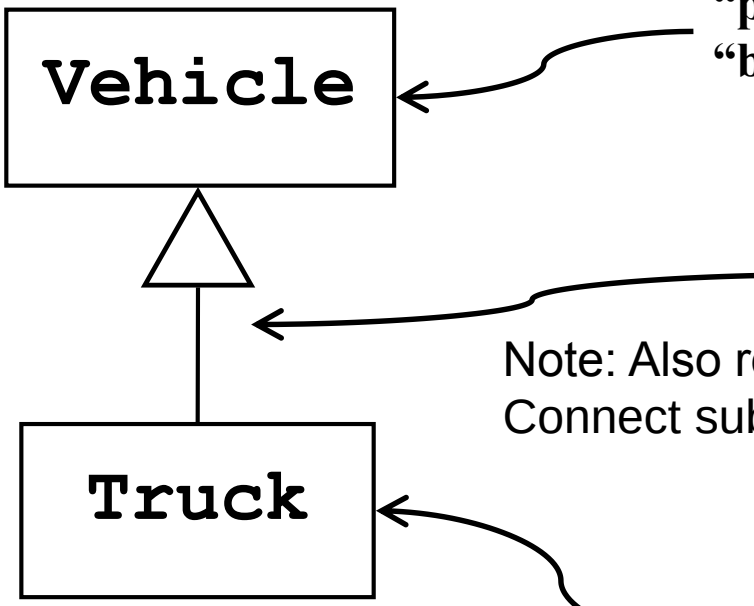
- **Definition**
- **Representation in UML, Implementation in Java, The "IS-A" concept**
- **Inheritance Hierarchies**
- **Overriding, Overloading**
- **Forms of Inheritance: Extension, Specialization, Specification**
- **Implications for Public vs. Private data**
- **Abstract classes and methods**
- **Single vs. Multiple Inheritance**

What Is Inheritance?

- **A specific kind of *association* between classes**
- **Various definitions:**
  - Creation of a *hierarchy of classes*, where lower-level classes share properties of a common "parent class"
  - A mechanism for indicating that one class is "similar" to another but has specific differences
  - A mechanism for enabling properties (attributes and methods) of a "super class" to be propagated down to "sub classes"
  - Using a "base class" to define what characteristics are *common* to all instances of the class, then defining "derived classes" to define what is special about each subgrouping



Inheritance In UML



"parent class"; "super class";
"base class"

"**A Truck is-a Vehicle**"

Note: Also referred to as "**Generalization**" relationship.
Connect sub-class "Truck" to super-class "Vehicle".

"child class"; "subclass";
"derived class"

Inheritance In Java

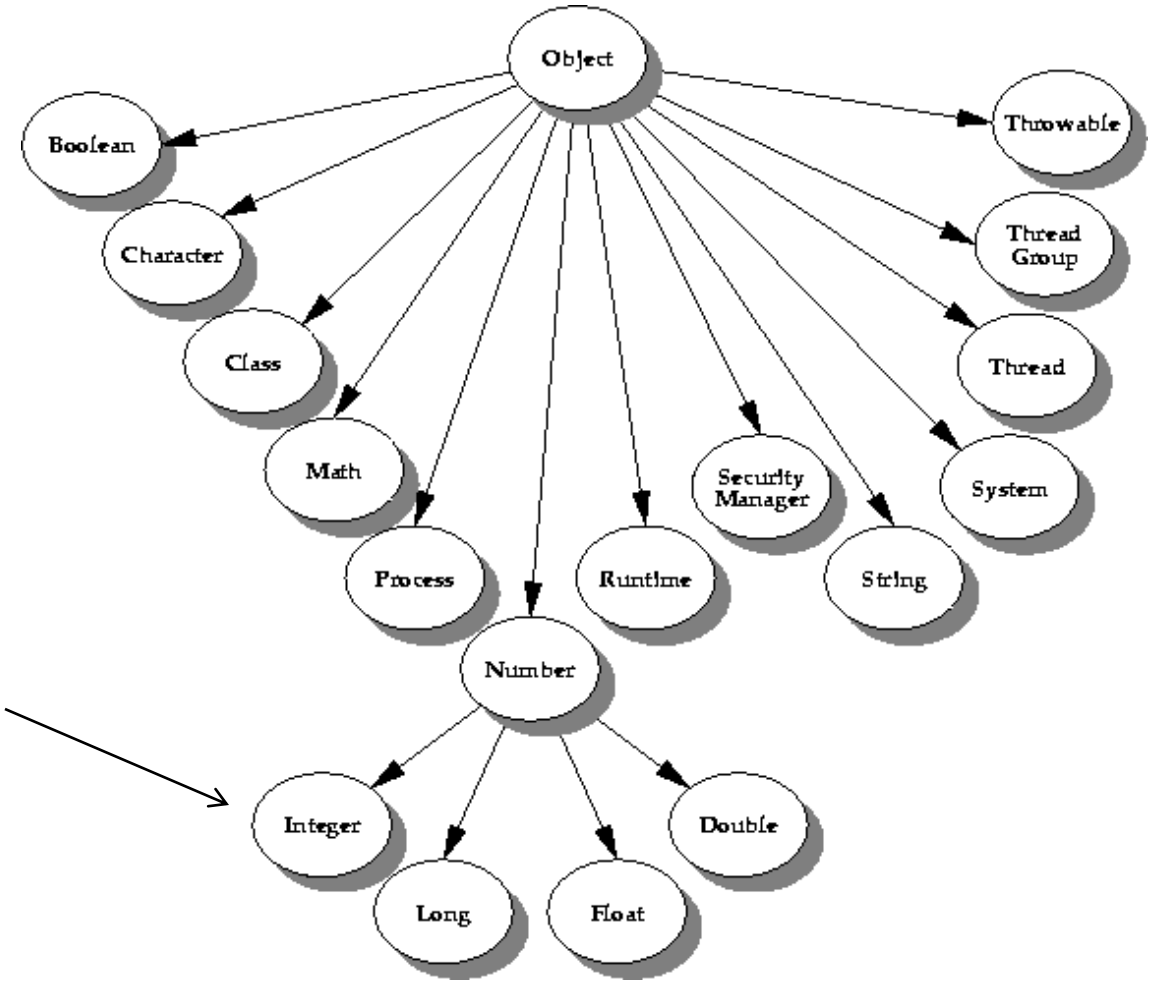
- **Specified with the keyword "extends" :**

```
public class Vehicle {

  private int weight;
  private double purchasePrice;
  //... other Vehicle data here

  public Vehicle ()
  { ... }

  public void turn (int direction)
  { ... }

  // ... other Vehicle methods here
}
```

```
public class Truck extends Vehicle {
  private int freightCapacity;
  //... other Truck data here

  public Truck ()
  { ... }

  // ... Truck-specific methods here
}
```

- **Note: a Truck "is-a" Vehicle**
- **Only a single "extends" allowed (no "multiple inheritance")**
- **Absence of any "extends" clause implies "extends Object"**

Object: The Cosmic Superclass



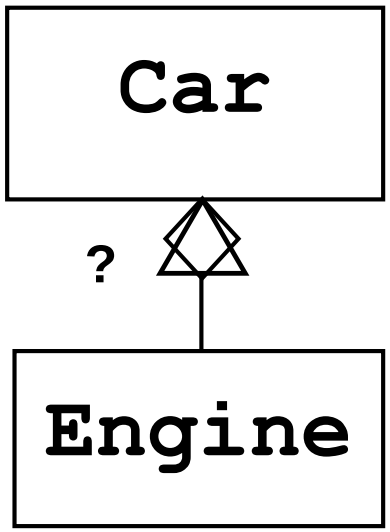
Wrapper Classes

The java.lang package contains the collection of base types (language types) that are always imported into any given compilation unit. This is where you'll find the declarations of Object (the root of the class hierarchy) and Class, plus threads, exceptions, wrappers for the primitive data types, and a variety of other fundamental classes.

Source: http://www.oracle.com/technetwork/java/libraries-140433.html

The "IS-A" Relationship

- **Inheritance *always* specifies an "is-a" relationship.**
- **If you can't say "A is a B" (or "A is a kind of B"), *it isn't inheritance***



An Engine "is a" Car ? X

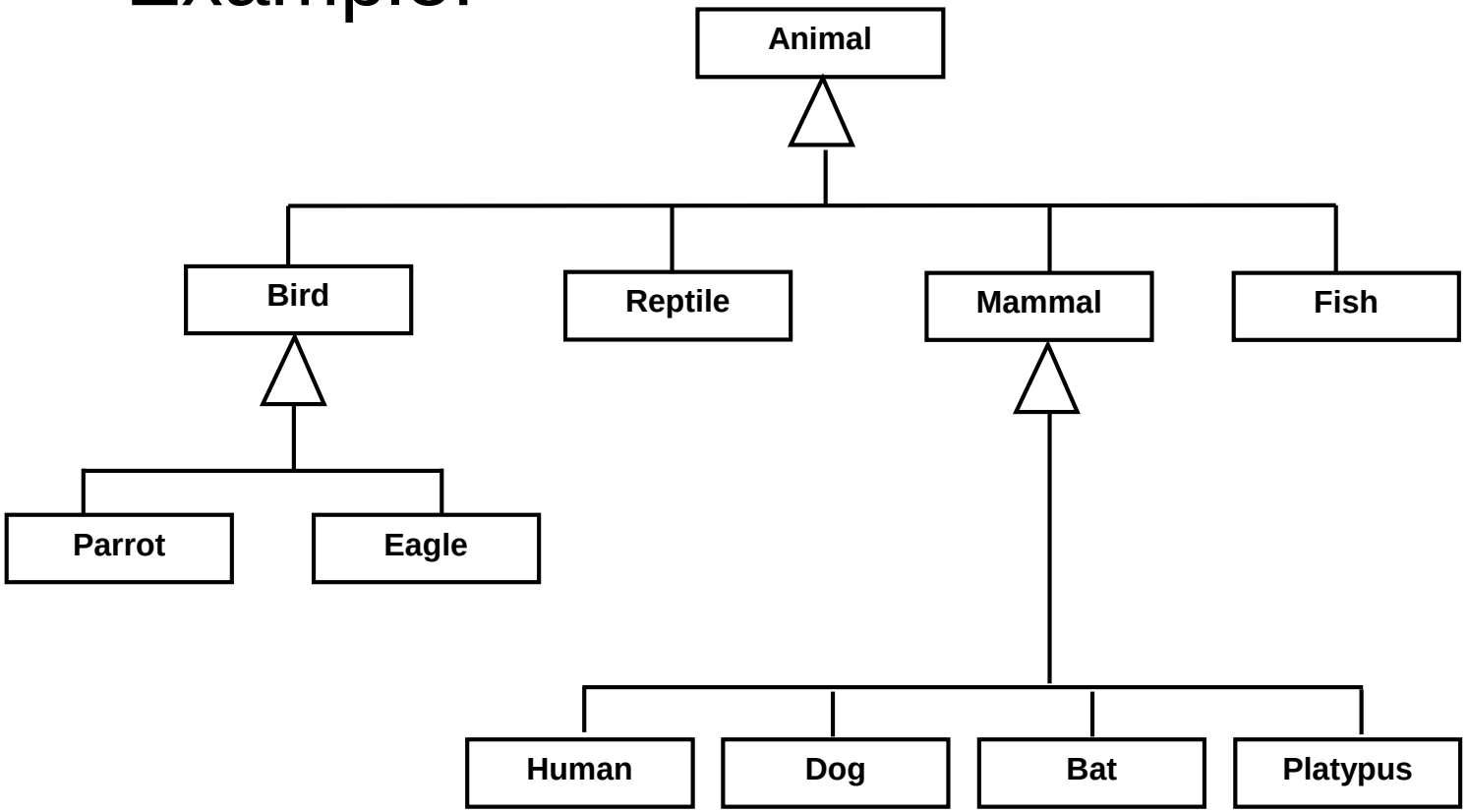
A Car "is an" Engine ? X

A Car "has-an" Engine ✓

An Engine "is a part of" a Car ✓

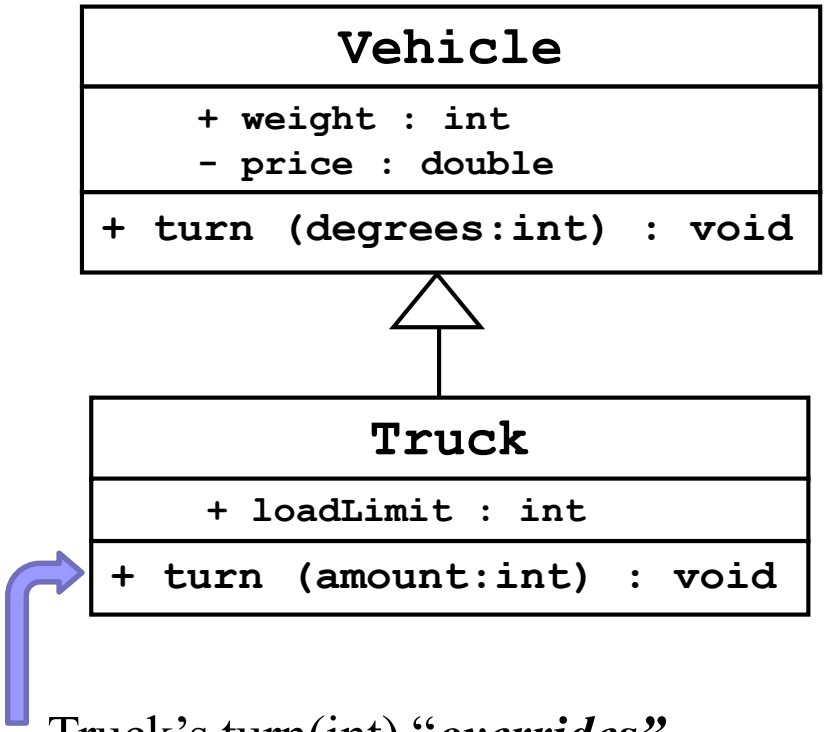
Inheritance Hierarchies

- Example:



Method Overriding

- Inheritance leads to an interesting possibility:
  duplicate method declarations




```
public class Vehicle {
   public int weight ;
   private double price ;

   public void turn (int degrees)
   { // some code to accomplish turning... }

   ...
}
```

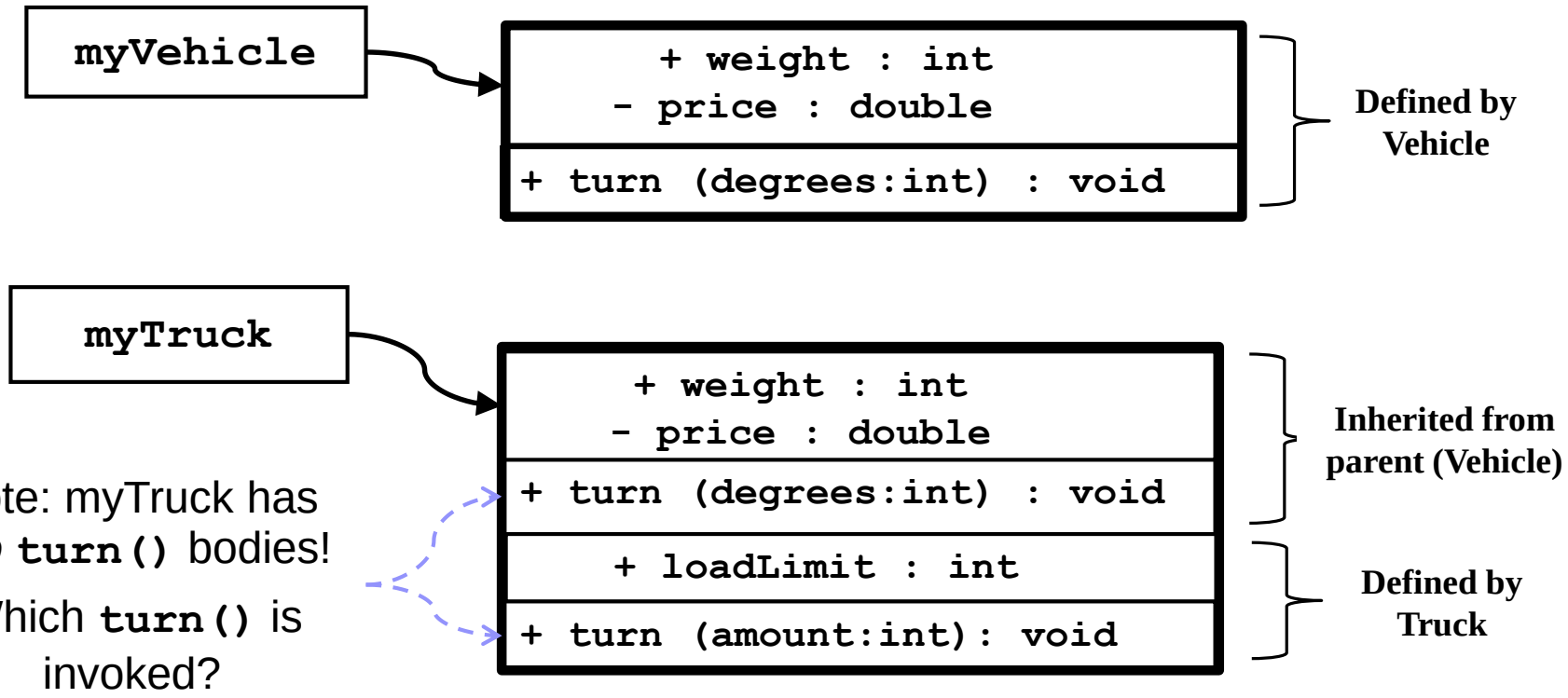
```
public class Truck extends Vehicle {

   public int loadLimit ;

   public void turn (int amount)
   { // different code to accomplish turning... }

   ...
}
```

Effects of Method Overriding

Consider the following code:

```
Vehicle myVehicle = new Vehicle();
Truck myTruck = new Truck();
```

… then we get two objects:



Note: myTruck has *two* `turn()` bodies!

Which `turn()` is invoked?

Method Overriding: Summary

- **Occurs when a child class redefines an inherited method, which:**
  - **has same name**
  - **has same parameters**
  - **returns same type or subtype**
- **Child objects contain the code for both methods**
  - **Parent method code plus the child (overriding) method code**
- **Calling an overridden method (in Java) invokes the child version**
  - **Never invokes the parent version**
  - **The child can invoke the parent method using "super.*xxx (...)*"**
- **It is not legal (in Java) to override and change the *return type which is not a subtype*.**
  - **So for the Vehicle/Truck example, Truck could NOT define**

```
public boolean turn (int amount) { ... }
```

Overloading

- ***Not* the same as "overriding"…**
  - **Overloading == same name but different parameter types**
  - **Can occur *in the same class or split between parent/child classes***

- ***Overloading examples:***
  - Methods with different numbers of parameters:

```
distance(p1);    distance(p1,p2)
```

  - Constructors with different parameter sequences:

```
Circle(); Circle(Color c); Circle(int radius);
 Circle(Color c, int radius);
```

  - Changing parameter type:

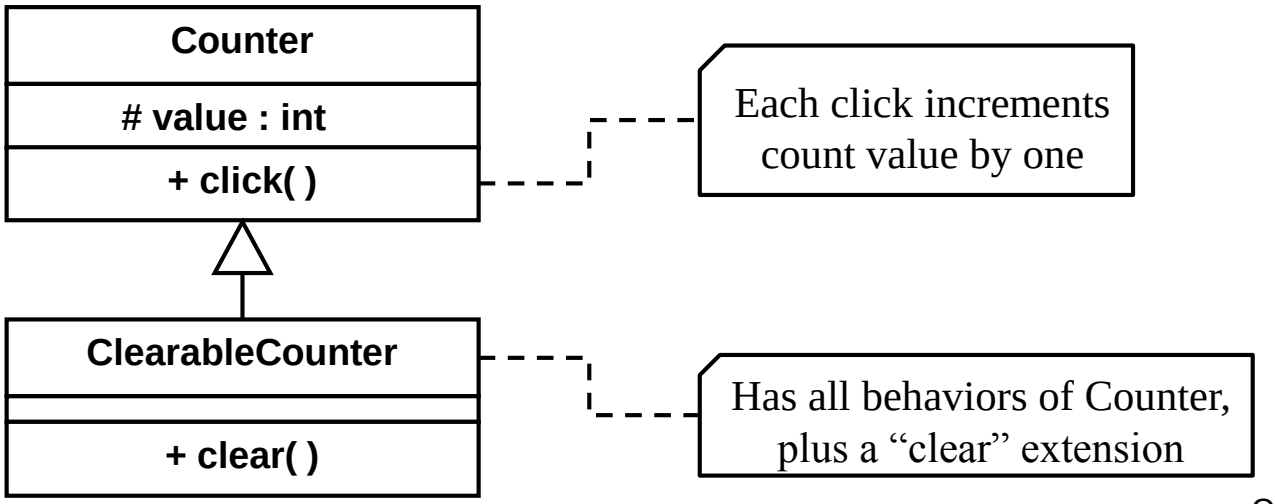
```
computeStandings(int numTeams);
computeStandings(double average);
computeStandings(Hashtable teams);
```

Typical Uses for Inheritance

- **Extension**
  - **Define *new behavior*, and**
  - **Retaining existing behaviors**
- **Specialization**
  - **Modify existing behavior(s)**
- **Specification**
  - **Provide ("specify") the implementation details of "abstract" behavior(s)**

Inheritance for Extension

- **Used to *define new behavior***
  - **Retains parent class' Interface and implementation**
- **Example: Counter**
  - **Base class increments on each "click"**
  - **Extension adds support for "clearing" (resetting)**

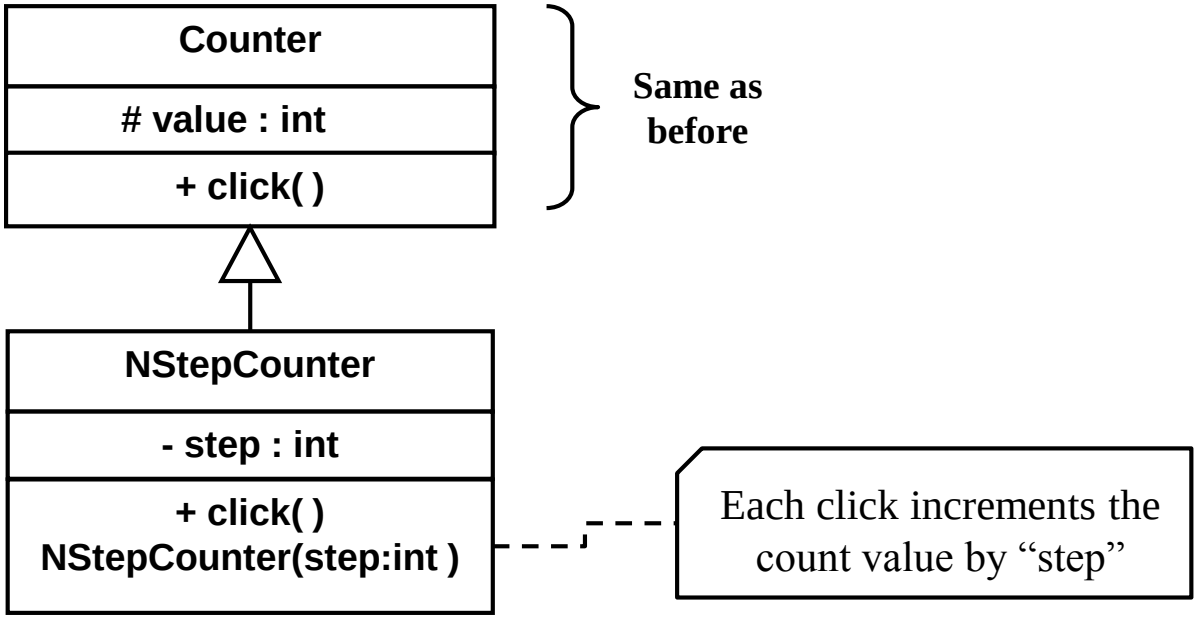


Inheritance for Extension (cont.)

```
/** This class defines a counter which increments on each call to click().
 * The Counter has no ability to be reset.  */
public class Counter {
  protected int value ;

  /** Increment the counter by one. */
  public void click() {
    value = value + 1;
  }
}


/** This class defines an object with all the properties of a Counter, and
 *  which also has a "clear" function to reset the counter to zero. */
public class ClearableCounter extends Counter {

  // Reset the counter value to zero.  Note that this method can
  // access the "value" field in the parent because that field
  //  is defined as "protected".

  public void clear () {
    value = 0 ;
  }
}
```

Inheritance for Specialization

- **Used to *modify existing behavior* (i.e. behavior defined by parent)**
- **Uses *overriding* to change the behavior**
- **Example: N-Step Counter**

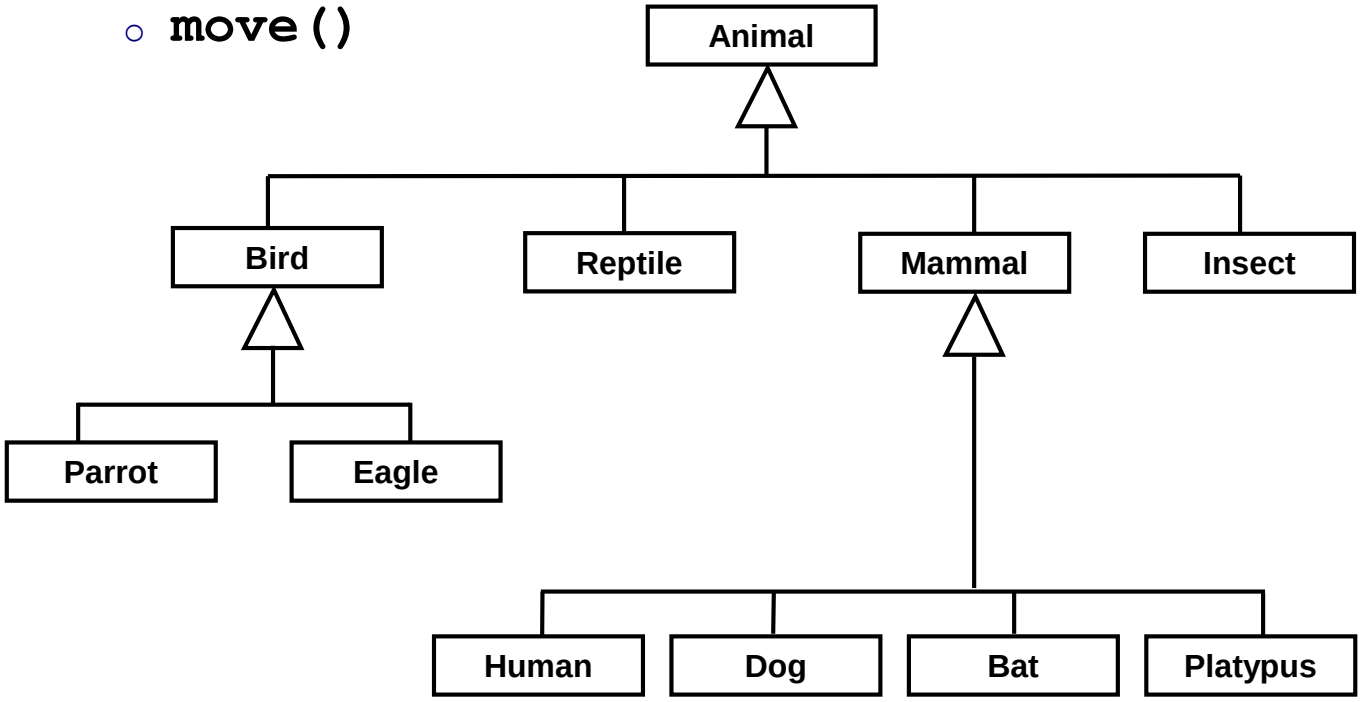


Inheritance for Specification

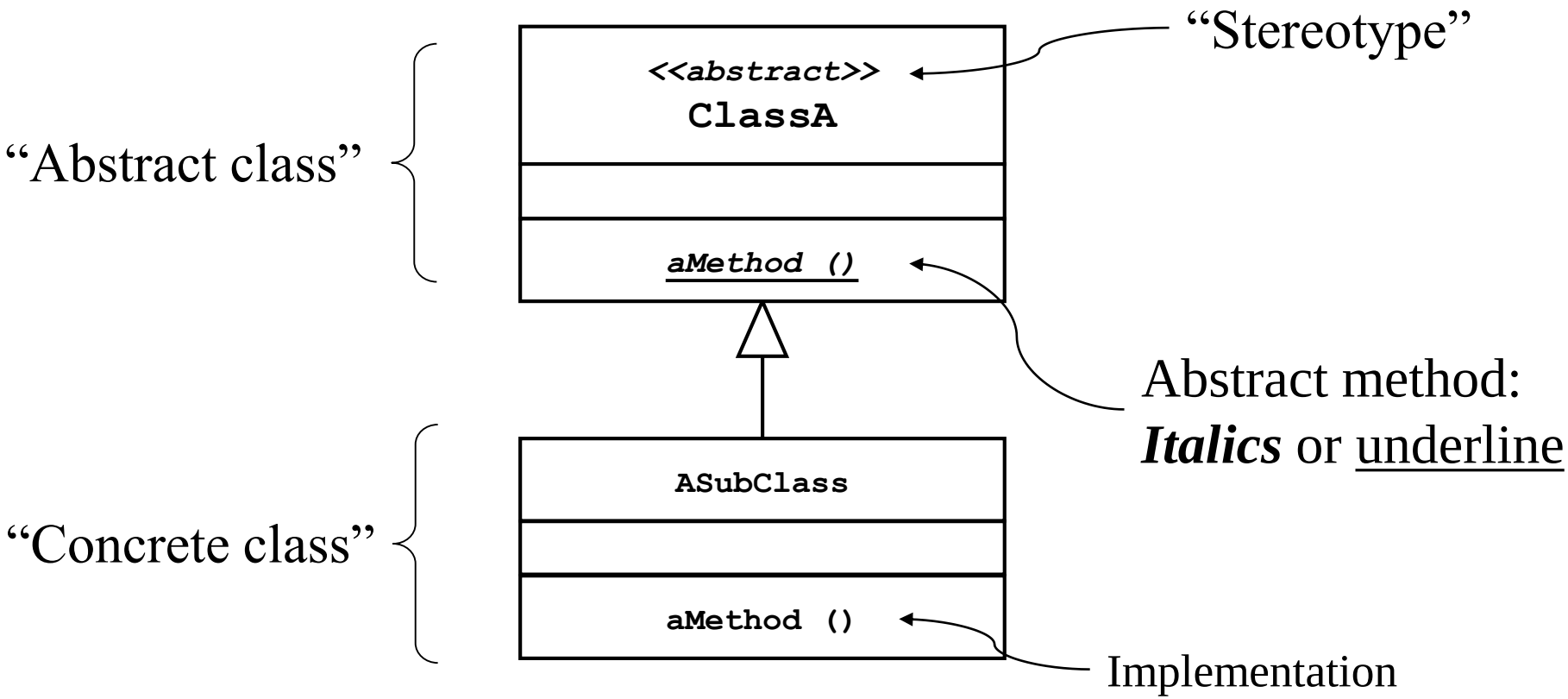
- **Used to *specify (define)* behavior *declared* (but not *defined*) by the parent**
  - **Classes which declare but don't define behavior: Abstract Classes**
  - **Methods which don't contain implementations: *Abstract methods***

Abstract Classes & Methods

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

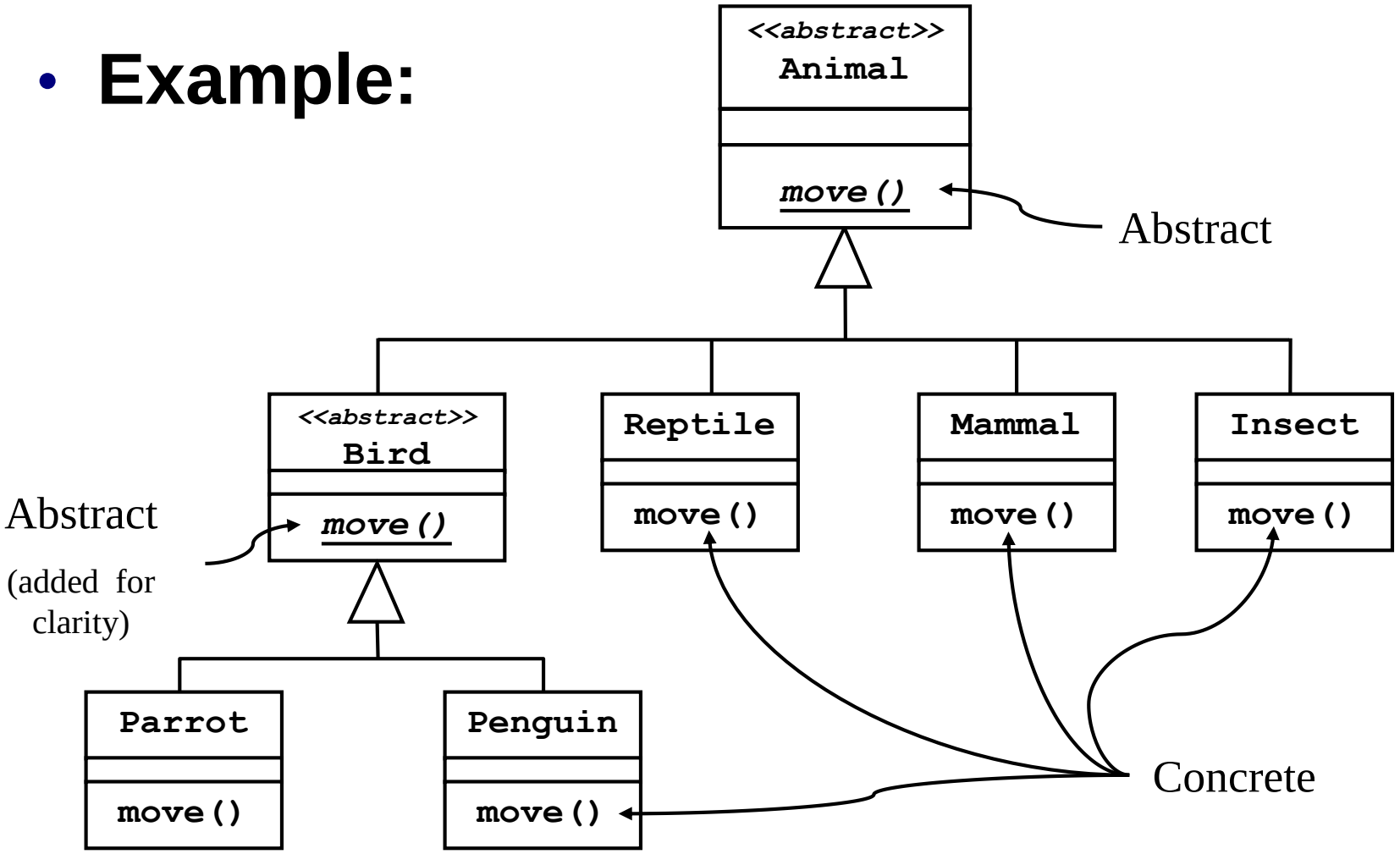


Inheritance for Specification (cont.)



Inheritance for Specification (cont.)

- **Example:**

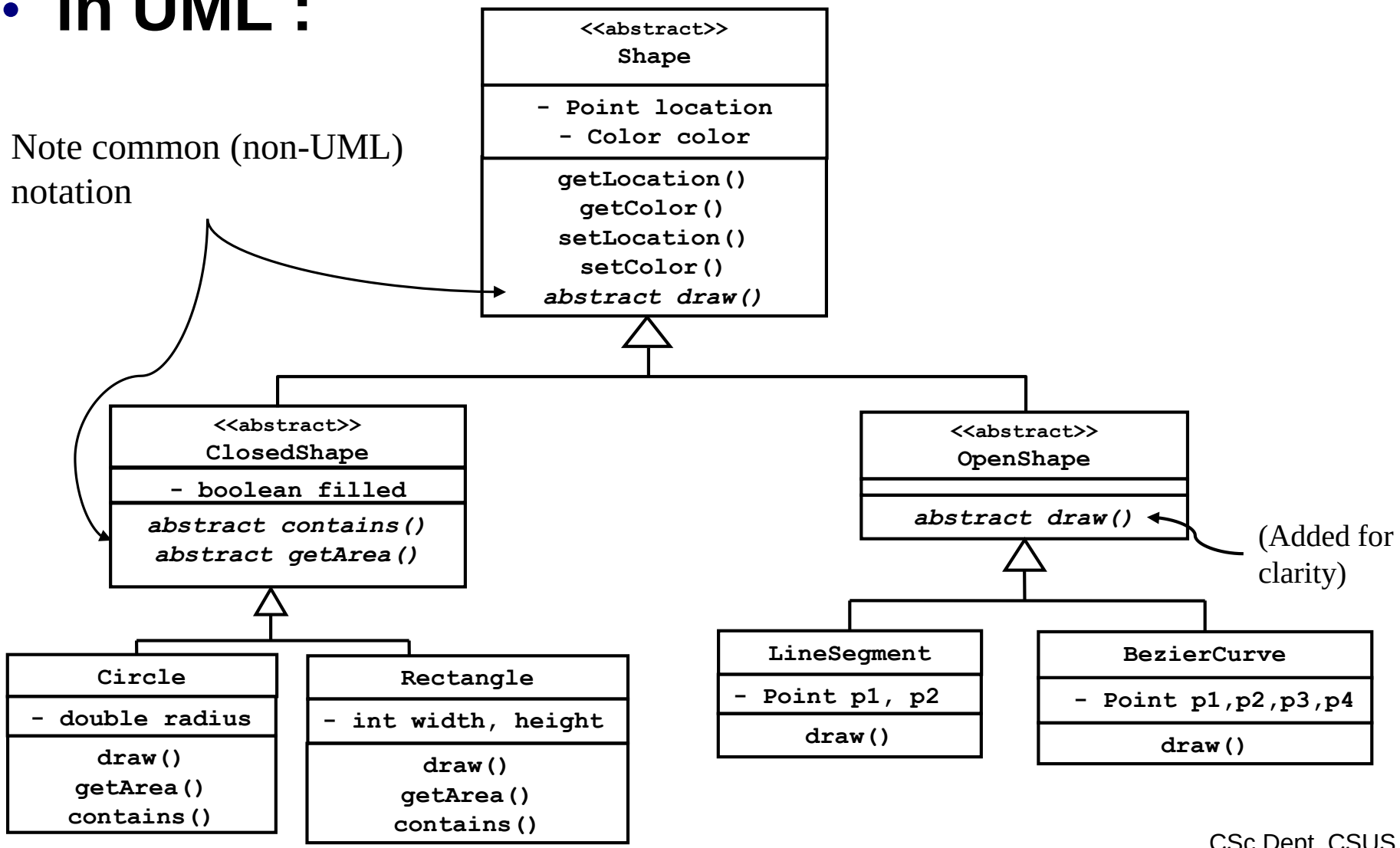


Inheritance for Specification (cont.)

- **Another example: abstract shapes**
  - Different kinds of shapes:
    - **`Line  Circle  Rectangle  BezierCurve  ...`**
  - Common (shared) characteristics :
    - a “Location”
    - a Color
    - …
  - Common operations (methods) :
    - **`getLocation()`**
    - **`setLocation()`**
    - **`getColor()`**
    - **`setColor()`**
    - **`draw()`** ← Depends on the shape!
    - **`getArea()`** ← Might be undefined!

Inheritance for Specification (cont.)

- **in UML :**



Overview

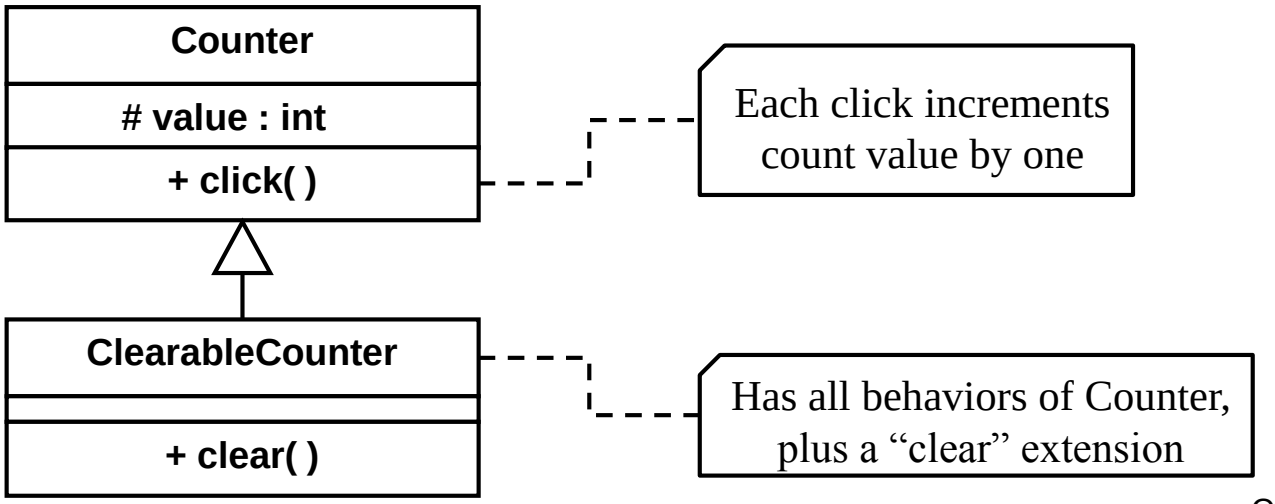
- **Definition**
- **Representation in UML, Implementation in Java, The "IS-A" concept**
- **Inheritance Hierarchies**
- **Overriding, Overloading**
- **Forms of Inheritance: Extension, Specialization, Specification**
- **Implications for Public vs. Private data**
- **Abstract classes and methods**
- **Single vs. Multiple Inheritance**

Typical Uses for Inheritance

- **Extension**
  - **Define *new behavior*, and**
  - **Retaining existing behaviors**
- **Specialization**
  - **Modify existing behavior(s)**
- **Specification**
  - **Provide ("specify") the implementation details of "abstract" behavior(s)**

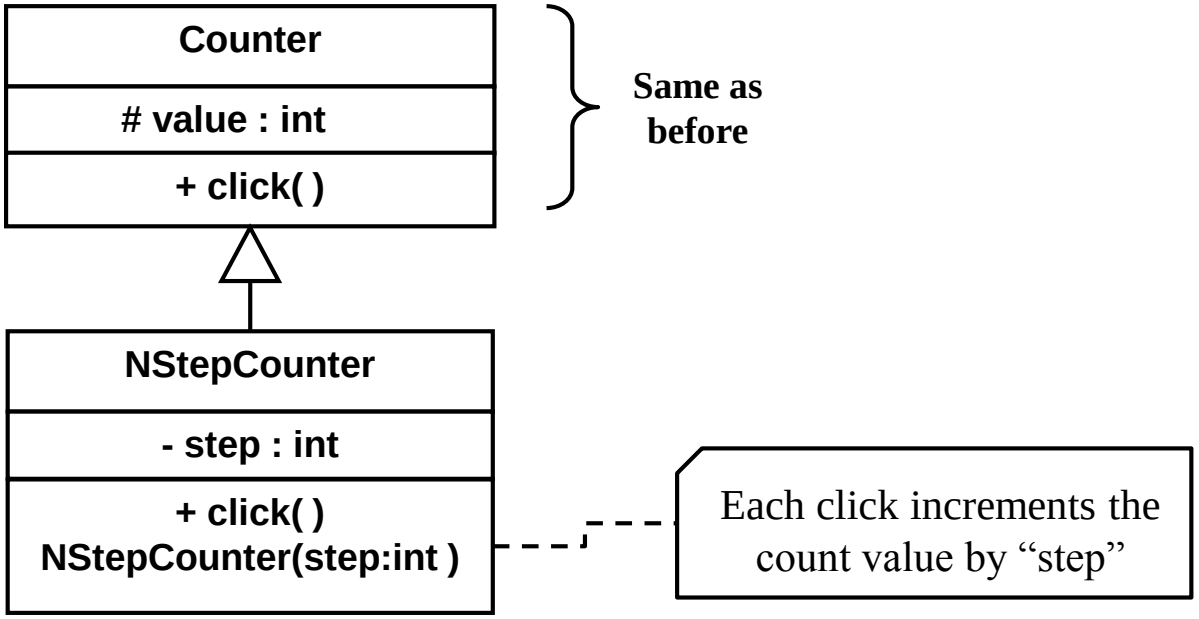
Inheritance for Extension

- **Used to *define new behavior***
  - **Retains parent class' Interface and implementation**
- **Example: Counter**
  - **Base class increments on each "click"**
  - **Extension adds support for "clearing" (resetting)**



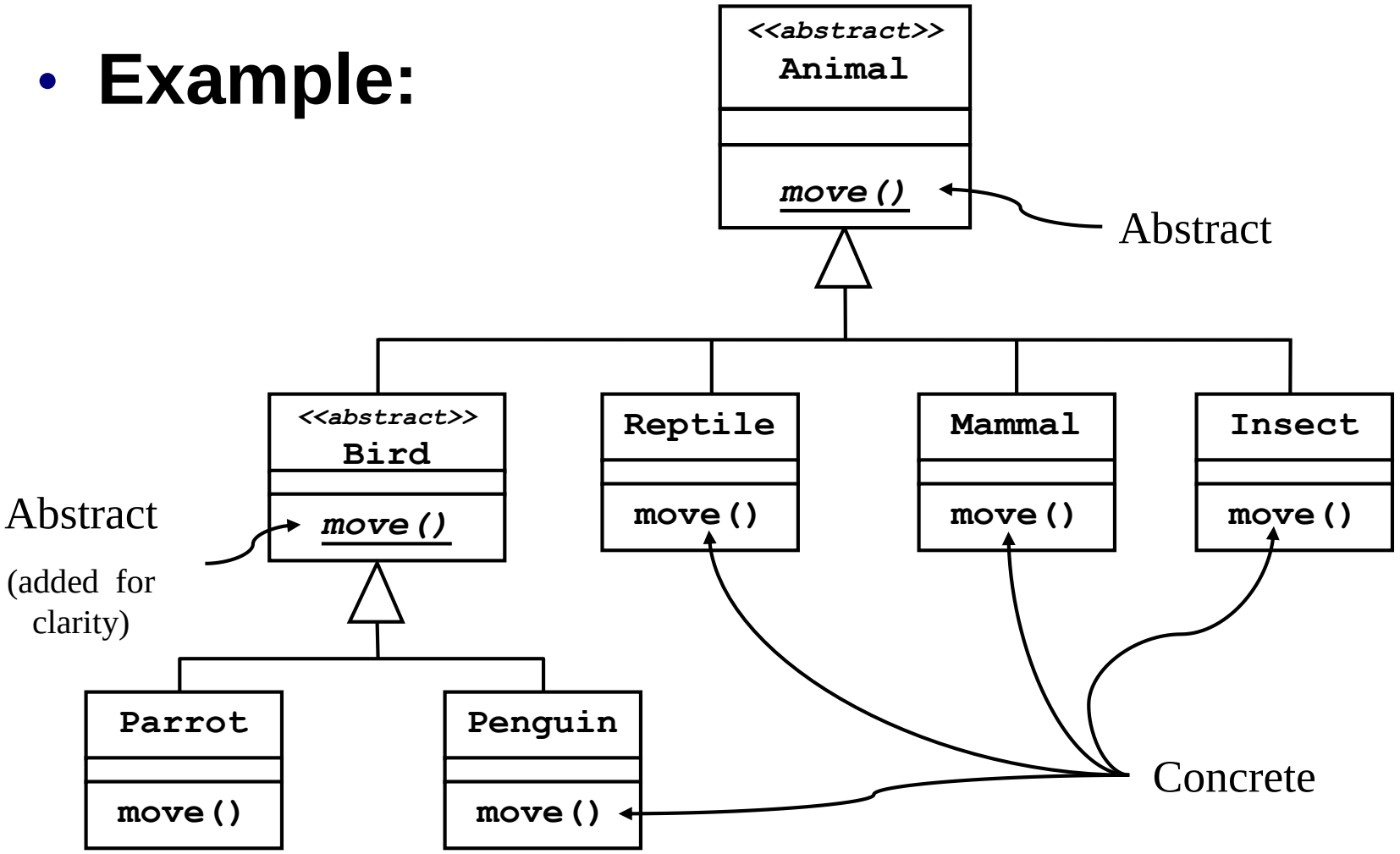
Inheritance for Specialization

- **Used to *modify existing behavior* (i.e. behavior defined by parent)**
- **Uses *overriding* to change the behavior**
- **Example: N-Step Counter**



Inheritance for Specification (cont.)

- **Example:**



Implications for Public vs. Private data

recall, from the encapsulation section:

`Point` (without "Accessors"):

```
public class Point {

   public double x, y ;

   public Point () {
      x = 0.0 ;
      y = 0.0 ;
   }
}
```

BAD

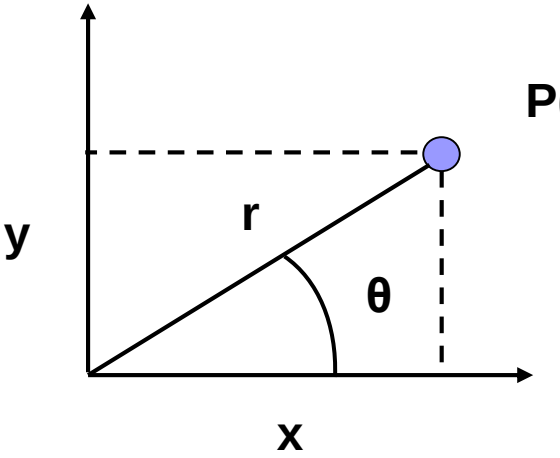
Now we will learn why!

`Point` (with "Accessors"):

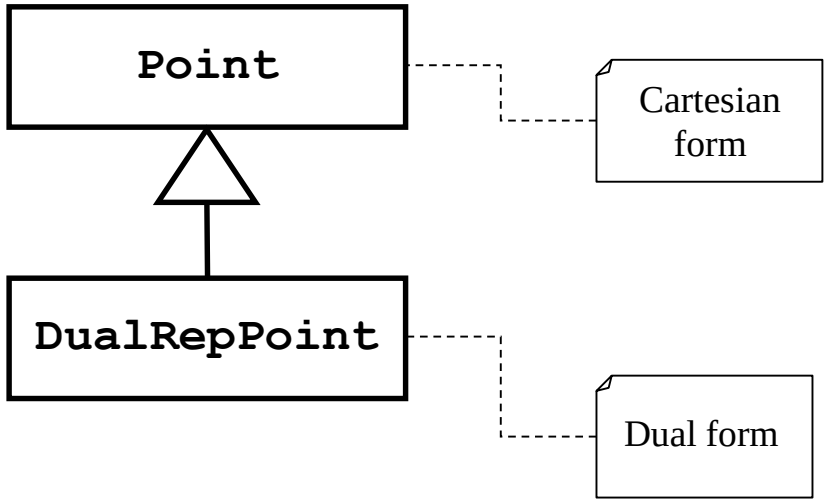
```
public class Point {
   private double x, y ;
   public Point (){
      x = 0.0 ;
      y = 0.0 ;
   }
   public double getX() {
      return x ;
   }
   public double getY() {
      return y ;
   }
   public void setX (double newX) {
      x = newX ;
   }
   public void setY (double newY) {
      y = newY ;
   }
}
```

GOOD

Example: extend "`Point`" to create "`DualRepPoint`"



P(x,y) = P(r,θ)



DualRepPoint (DRP): Ver. 1

← Note public access

```
public class DualRepPoint extends Point {

  public double radius, angle ;

  /** Constructor: creates a default point with radius 1 at 45 degrees */
  public DualRepPoint () {
    radius = 2.0 ;
    angle = 45 ;
    updateRectangularValues();
  }

  /** Constructor: creates a point as specified by the input parameters */
  public DualRepPoint (double theRadius, double angleInDegrees) {
    radius = theRadius ;
    angle = angleInDegrees;
    updateRectangularValues();
  }

  /** Force the Cartesian values (inherited from Point) to be consistent */
  private void updateRectangularValues() {
    x = radius * Math.cos(Math.toRadians(angle));     // legal assignments
    y = radius * Math.sin(Math.toRadians(angle));     // (x & y are public)
  }
}
```

Client Using Public Access

```
/** This shows a "client" class that makes use of the "V. 1 DualRepPoint" class.
 *  It shows how the improper implementation of DualRepPoint (that is, use of
 *  fields with public access) leads to problems...
 */
public class SomeClientClass {

  private DualRepPoint myDRPoint ;   //declare client's local DualRepPoint

  // Constructor: creates a DualRepPoint with default values,
  // then changes the DRP's radius and angle values
  public SomeClientClass() {
       myDRPoint = new DualRepPoint() ;      //create private DualRepPoint
       myDRPoint.radius = 5.0 ;              //update myPoint's values
       myDRPoint.angle = 90.0 ;
  }
  ...
}
```

Anything wrong?

DualRepPoint: Ver. 2

```
/** This class maintains a point representation in both Polar and Rectangular
 *  form and protects against inconsistent changes in the local fields */
public class DualRepPoint extends Point {

   private double radius, angle ;

   // constructors as before (not shown) ...
   public double getRadius() { return radius ; }
   public double getAngle() { return angle ; }

   public void setRadius(double theRadius) {
      radius = theRadius ;
      updateRectangularValues();
   }

   public void setAngle(double angleInDegrees) {
      angle = angleInDegrees;
      updateRectangularValues();
   }

   // force the Cartesian values (inherited from Point) to be consistent
   private void updateRectangularValues() {
      x = radius * Math.cos(Math.toRadians(angle));
      y = radius * Math.sin(Math.toRadians(angle));
   }
}
```

← New: private access

New: public accessors

Client Using DRP Accessors

```
/** This new version of the client code shows how requiring the use of accessors
 *  when manipulating the DualRepPoint radius & angle fields fixes (one) problem ...
 */

public class SomeClientClass {

  private DualRepPoint myDRPoint ;

  public SomeClientClass() {                  // client constructor
      myDRPoint = new DualRepPoint();         // create a private DualRepPoint
      myDRPoint.setRadius(5.0) ;  // alter DRP's values (safely): client has
      myDRPoint.setAngle(90.0) ;  // no way to access radius/angle directly
  }

  .... etc.
}
```

Problem solved?

Accessing Other DRP Fields

```
/** This newer version of the client code shows how requiring the use of accessors
 *  when manipulating the DualRepPoint radius & angle fields fixes (one) problem
 *  ... but not all problems...
 */
public class SomeClientClass {

  private DualRepPoint myDRPoint ;

  public SomeClientClass() {                    // client constructor as before
      myDRPoint = new DualRepPoint();
      myDRPoint.setRadius(5.0) ;
      myDRPoint.setAngle(90.0) ;
  }

  //a new client method which manipulates the portion inherited from Point
  public void someMethod() {
      myDRPoint.x = 2.2 ;
      myDRPoint.y = 7.7 ;
      ...
  }
  ... etc.
}
```

Anything wrong?

Public Fields *Break Code*

- **`Point` (without "Accessors"):**

```
public class Point {

      public double x, y ;

      public Point () {
        x = 0.0 ;
        y = 0.0 ;
      }
      . .
}
```

BAD BAD BAD

Using Accessors

- **`Point` (with "Accessors"):**

Good !

Good !

Good !

Good !

```
public class Point {

    private double x, y ;

    public Point (){
      x = 0.0 ;
      y = 0.0 ;
    }

    public double getX() { return x ; }

    public double getY() { return y ; }

    public void setX (double newX) {
      x = newX ;
    }

    public void setY (double newY) {
      y = newY ;
    }

    // other methods here...
}
```

Accessors Don't Solve All Problems

```
/** This new version of the client code shows how requiring the use of accessors
 *  in ALL classes may have fixed ONE problem ... but another still exists
 */

public class SomeClientClass {

  private DualRepPoint myDRPoint ;

  public SomeClientClass() {                  // client constructor
       myDRPoint = new DualRepPoint();        // create a private DualRepPoint
       myDRPoint.setX(2.2) ;          // alter DRP's inherited X,Y values
       myDRPoint.setY(7.7) ;          //   using inherited accessors
  }

  .... etc.
}
```

- **Problem still exists! (Calling parent setX/setY)**
- **Solution ?**

DualRepPoint: Correct Version

```
public class DualRepPoint extends Point  {    //uses "Good" Point with accessors

   private double radius, angle ;

   //...constructors and accessors for radius and angle here as before ...

   // Override inherited accessors

   public void setX (double xVal) {
      super.setX(xVal) ;
      updatePolarValues() ;
   }

   public void setY (double yVal) {
      super.setY(yVal) ;
      updatePolarValues() ;
   }

   private void updateRectangularValues() {
      super.setX(radius * Math.cos(Math.toRadians(angle))) ;
      super.setY(radius * Math.sin(Math.toRadians(angle))) ;
   }

   //new private method to maintain consistent state
   private void updatePolarValues() {
      double x = super.getX() ;      // note: some people would use protected to
      double y = super.getY() ;      // allow direct subclass access to X & y
      radius = Math.sqrt (x*x + y*y) ;
      angle = Math.atan2 (y,x) ;
   }
}
```

```
//note that overriding the parent accessors
// makes it impossible for a client to put
// put a DRP into an inconsistent state
```

Java Abstract Classes

- Both classes and methods can be declared abstract

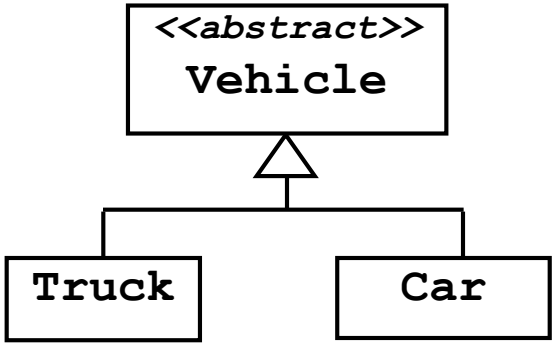
```
public abstract class Animal {
      public abstract void move () ;
}
```

- Abstract classes cannot be instantiated
  - But they can be extended
- If a class contains an abstract method, the class must be declared abstract
  - But abstract classes can also contain concrete methods
- For a subclass to be concrete, it must implement bodies for all inherited abstract methods
  - Otherwise, the subclass is also automatically abstract (and must be declared as such)

Abstract Classes (cont.)

- Can *declare* a variable of abstract type
- Cannot *instantiate* such a variable

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



Abstract Classes (cont.)

- **`static`**, **`final`**, and/or **`private`** methods ***cannot*** be declared abstract
  - No way to override or change them; no way to provide a "specification"

- **`protected`** methods *can* be declared abstract.

- Java "abstract method" = C++ "pure virtual function":

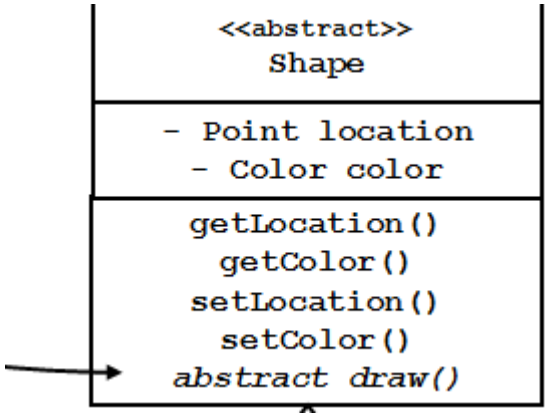
```
abstract void move () ;          //Java
```

vs.

```
virtual void move() = 0 ;        //C++
```

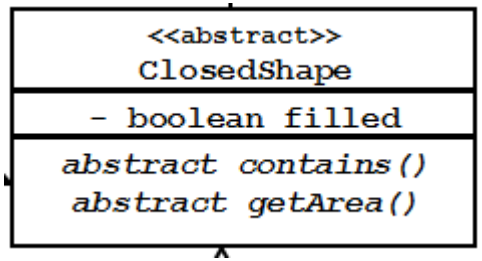
Example: Abstract Shapes

```
/** This class is the abstract superclass of all "Shapes". Every Shape has a
 * color, a "location" (origin), accessors, and a draw() method.  */
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);
}
```



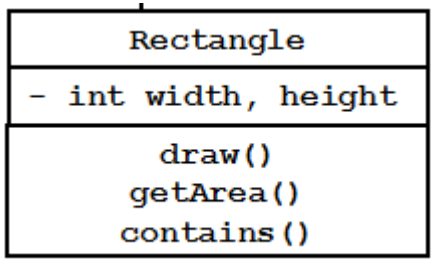
Example: Abstract Shapes (cont.)

```
/** This class defines Shapes which are "closed" - meaning the Shape has a
 *  boundary which delineates "inside" from "outside".  Closed Shapes can either be
 *  "filled" (solid) or "not filled"  (interior is empty). Every ClosedShape must
 *  have a method "contains(Point)", which determines whether a given Point is inside
 *  the shape or not, and a method "getArea()" which returns the area inside the shape.
 */
public abstract class ClosedShape extends Shape {

  private boolean filled;          // attribute common to all closed shapes

  public ClosedShape() {
    //automatically calls super() - no-arg constructor of its parent (Shape)
    filled = false;
  }
  public ClosedShape(boolean filled) {
    //automatically calls super() - no-arg constructor of its parent (Shape)
    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();
}
```

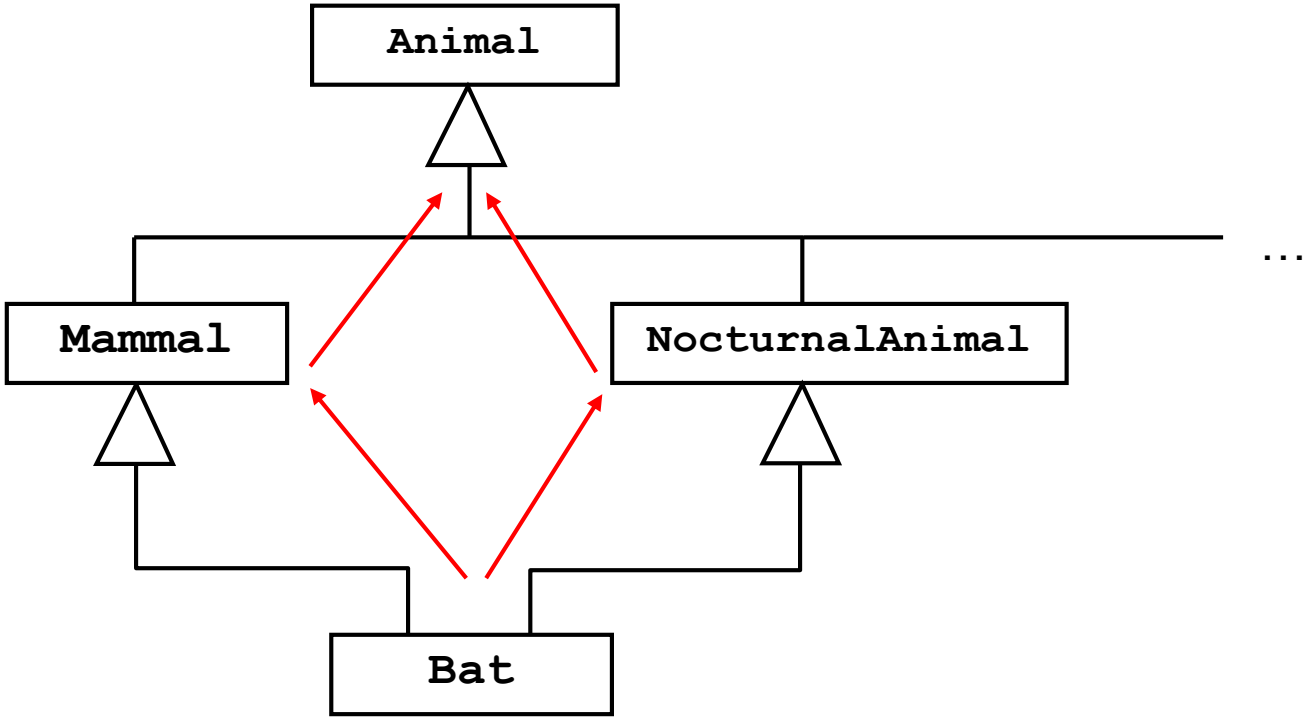


Example: Abstract Shapes (cont.)

```
/** This class defines closed shapes which are rectangles. */
public class Rectangle extends ClosedShape {

  private int width;
  private int height;

  public Rectangle() {
    super(true);  //no-arg constructor of its parent (ClosedShape) is not called
    width = 2;
    height = 1;
  }

  public boolean contains(Point p) {
    //... code here to return true if p lies inside this rectangle,
    //    or return false if not.
  }

  public double getArea() {
    return (double) (width * height) ;
  }

  public void draw (Graphics g) {
    if (isFilled()) {
      //  code here to draw a filled (solid) rectangle using
      //  Graphics object "g"
    } else {
      //  code here to draw an empty rectangle using
      //  Graphics object "g"
    }
  }
}
```



Multiple Inheritance



A possible alternative Animal Hierarchy

Multiple Inheritance (cont.)

- C++ allows multiple inheritance:

```
class Animal{...};

class Mammal : Animal {
  public : void sleep() {...} ;
  ...
};

class NocturnalAnimal : Animal {
  public : void sleep() {...} ;
  ...
};

class Bat : Mammal, NocturnalAnimal {...};
```

- Programmer must disambiguate references:

```
void main (int argc, char** argv) {
  Bat aBat;
  aBat.NocturnalAnimal::sleep();
}
```