COP 3330: Object-Oriented Programming
Summer 2007

Inheritance and Polymorphism – Part 2

Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2007

School of Electrical Engineering and Computer Science
University of Central Florida

Polymorphism

- One of the more important features of object-oriented programming is that – *a code expression can invoke different methods depending on the types of objects using the code*. This language feature is known as *polymorphism*.
  - Some people view method overloading as *syntactic* or *primitive* polymorphism. With method overloading, Java can determine which method to invoke at compile time. The decision is based on the invocation signature.
  - In true polymorphism, sometimes referred to as *pure* polymorphism, the decision as to which method to invoke must be delayed until run-time.

Polymorphism Example

```
//demonstrate polymorphism using various point classes
import java.awt.*;
import geometry.*;

public class PolymorphismDemo{
   public static void main(String[] args) {
      Point[] p = new Point[4];  //array of points
      p[0] = new Colored3DPoint(4,4,4,Color.black);
      p[1] = new ThreeDPoint(2,2,2);
      p[2] = new ColoredPoint(3,3,Color.red);
      p[3] = new Point(4,4);
      for (int i = 0; i < p.length; i++){
           String s = p[i].toString();
           System.out.println("p[" + i + "]: " + s);
      }
      return;
   }
}
```

Output from Polymorphism Example

Command Prompt (2)

```
E:\Program Files\Java\jdk1.6.0\bin>javac Colored3DPoint.java

E:\Program Files\Java\jdk1.6.0\bin>javac PolymorphismDemo.java

E:\Program Files\Java\jdk1.6.0\bin>java PolymorphismDemo
p[0]: class Colored3DPoint[4,4,,4,java.awt.Color[r=0,g=0,b=0]]
p[1]: class ThreeDPoint[2,2,,2]
p[2]: class ColoredPoint[3,3,,java.awt.Color[r=255,g=0,b=0]]
p[3]: java.awt.Point[x=4,y=4]

E:\Program Files\Java\jdk1.6.0\bin>
```

Polymorphism (cont.)

- The for loop in the main method of the code on the previous slide contains the following polymorphic code:

```
for (int i = 0; i < p.length; i++){
          String s = p[i].toString();
          System.out.println("p[" + i + "]: " + s);
      }
```

polymorphic code

- In Java, *it is not the variable type that determines the invocation, but the type of object at the location to which the variable refers*. The code is polymorphic because array p is a heterogeneous list; that is each element reference values of different types.

Polymorphism (cont.)

- That array p is a heterogeneous list is a result of how its elements are set.

```
Point[] p = new Point[4];  //array of points
p[0] = new Colored3DPoint(4,4,4,Color.black);
p[1] = new ThreeDPoint(2,2,2);
p[2] = new ColoredPoint(3,3,Color.red);
p[3] = new Point(4,4);
```

This assignment causes p[1] to refer to a ThreeDPoint object which extends from Point.

This assignment causes p[3] to refer to a Point object.

This assignment makes sense because Colored3DPoint has an *is-a* relationship with Point. Colored3DPoint extends from ThreeDPoint, which extends from Point.

This assignment causes p[2] to refer to a ColoredPoint object which extends from Point.

Polymorphism (cont.)

- Because array `p` is heterogeneous, the `toString()` method that is invoked in the expression `p[i].toString()` depends on the type of `Point` object to which `p[i]` refers.

```
String s = p[0].toString();
String s = p[1].toString();
String s = p[2].toString();
String s = p[3].toString();
```

invokes Colored3DPoint method toString()

invokes ThreeDPoint method toString()

invokes ColoredPoint method toString()

invokes Point method toString()

Flexibility for Future Enhancements

- With polymorphism, the decision on which method to invoke in an expression may have to be determined at run-time.
- This capability makes it possible to compile a method that performs a method invocation in its body even though the subclass eventually supplying the method has not yet been implemented or even defined.

Design with Inheritance in Mind

- When designing a collection of new but related classes, pay attention to what are the common behaviors and characteristics. This process is known as factorization.
  - Try to create a coherent base class that provides this commonality.
  - From the base class, subclasses can be defined to provide the special behaviors and characteristics.
  - Your current work effort should be reduced as common behaviors are implemented just once.
  - Future work efforts should also be minimized as new features can be added by extending the existing classes.

Inheritance Nuances

- There are some important nuances surrounding inheritance behavior in Java. In the next few slides, we'll construct several small classes whose sole purpose is to illustrate the most important of these nuances as they pertain to:

1. Constructors
2. Controlling access
3. Data fields
4. Typing
5. Late binding
6. Finality

Inheritance Nuances - Constructors

- Consider the following class B composed of two constructors. Both constructors produce output when invoked.

```
public class B {
  //B(): default constructor
  public B(){
     System.out.println("Using default constructor in B");
  }

  //B(): specific constructor
  public B(int i){
     System.out.println("Using int specific constructor in B");
  }
}
```

Inheritance Nuances – Constructors (cont.)

- Before reading the explanation on page 13, try to determine the output of the following program.

```
public class C extends B {
  //C(): default constructor
  public C(){
     System.out.println(“Using default constructor in C”);
     System.out.println();
  }
  //C(): specific constructor
  public C(int a){
     System.out.println(“Using int specific constructor in C”);
     System.out.println();
  }
  //C(): specific constructor
  public C(int a, int b){
     System.out.println(“Using int-int specific constructor in C”);
     System.out.println();
  }
  //main(): application entry point
  public static void main (String[] args){
     C c1 = new C();
     C c2 = new C(2);
     C c3 = new C(2,4); return;
  } }
```

Inheritance Nuances – Constructors (cont.)

- Output of the `C.java` program.

```
E:\Program Files\Java\jdk1.6.0\bin>javac B.java

E:\Program Files\Java\jdk1.6.0\bin>javac C.java

E:\Program Files\Java\jdk1.6.0\bin>java C
Using default constructor in B
Using default constructor in C

Using default constructor in B
Using int specific constructor in C

Using default constructor in B
Using int-int specific constructor in C


E:\Program Files\Java\jdk1.6.0\bin>
```

Inheritance Nuances – Constructors (cont.)

```
//main(): application entry point
     C c1 = new C();
```

```
//B(): default constructor
  public B(){
     System.out.println("Using B's default constructor");
  }
```

```
//C(): default constructor
  public C(){
     System.out.println("Using C's default constructor");
     System.out.println();
  }
```

Output

```
Using B's default constructor
Using C's default constructor

Using B's default constructor
Using C's int constructor

Using B's int constructor
Using C's int-int constructor
```

Construction is a two-step process. Construction of c1 begins with the construction of its superclass attributes. Implicit invocation by Java of the superclass default constructor. Exactly the same scenario occurs for object c2, but C's default int constructor is invoked.

Inheritance Nuances – Controlling Access

- Up to this point our member variables and methods have either had `public` or `private` access rights.
  - A public member has no restrictions on its access.
  - A private member can only be used by the class that defines that member.
- With respect to `public` and `private` access rights, a subclass is treated no differently than any other Java class.
  - A subclass can access the superclass's `public` members and cannot access the superclass's `private` members.
- Recall that Java supports two additional access rights: `protected` and the default access. Only two classes in the same package can access each other's default access members.

Inheritance Nuances – Controlling Access (cont.)

- Besides being accessible in its own class, a protected variable or method can be used by subclasses of its class.
- Consider the example in the next couple of slides. Class `P` from package `demo` contains a `private` instance variable `data`, a `public` default constructor, a `public` accessor `getData()`, a `protected` mutator `setData()`, and a facilitator `print()` which has default access.
- Class `Q` extends class `P`. Thus `Q` can invoke `P`'s default constructor, and mutator `setData()`. However, `Q` cannot access directly `P`'s instance variable data.

Inheritance Nuances – Controlling Access (cont.)

```
package demo;
public class P {
   //instance variable
   private int data;
   //P(): default constructor
   public P(){
     setData(0);
   }
   //getData(): accessor
   public int getData(){
     return data;
   }
   //setData: mutator
   protected void setData(int v){
      data = v;
   }
   //print(): facilitator
   void print(){
      System.out.println();
   }
}
```

Inheritance Nuances – Controlling Access (cont.)

```
import demo.P;
public class Q extends P {
   //Q(): default constructor
   public Q(){
     super();
   }
   //Q(): specific constructor
   public Q(int v){
     setData(v);
   }
   //toString(): facilitator
   public String toString(){
      int v = getData();
      return String.valueOf(v);
   }
   //invalid1(): invalid method
   public void invalid1(){
      data = 12;
   }
   //invalid2(): illegal method
   public void invalid2() {
      print();
   }
}
```

Q can access superclass's public default constructor

Q can access superclass's protected mutator method

Q can access superclass's public accessor method

Q cannot access directly superclass's private data field

Q cannot access directly superclass'default access method print() since Q is not in package demo

Inheritance Nuances – Controlling Access (cont.)

```
package demo;   //R is a class in package demo
public class R {
   //instance variable
   private P p;
   //R(): default constructor
   public R(){
     p = new P();
   }
   //set(): mutator
   public void set(int v){
     p.setData(v);
   }
   //get(): accessor
   public int get(){
      return p.getData();
   }
   //use(): facilitator
   public void use(){
      p.print();
   }
   //invalid(): illegal method
   public void invalid() {
      p.data = 12;
   }
}
```

R can access P's public default constructor

R can access P's protected mutator method

R can access P's public accessor method

R can access P's default access method

R cannot access directly P's private data field

Inheritance Nuances – Controlling Access (cont.)

```
import demo.P;    //S is neither part of package demo nor extended from P
public class S {
   //instance variable
   private P p;
   //S(): default constructor
   public S(){
      p = new P();
   }
   //get(): accessor
   public int get(){
      return p.getData(v);
   }
   //illegla1(): illegal method
   public void illegal1 (int v){
      p.setData(v);
   }
   //illegal2(): illegal method
   public void illegal2(){
      p.data = 12;
   }
   //illegal3(): illegal method
   public void illegal3() {
      p.print();
   }
}
```

S can access P's public default constructor

S can access P's public accessor

S cannot access P's protected mutator method

S cannot access directly P's private data field

S cannot access directly P's default access print() method

Summary of Access Rights

Member Restriction	this	subclass	package	general
public	yes	yes	yes	yes
protected	yes	yes	yes	no
default	yes	no	yes	no
private	yes	no	no	no

Inheritance Nuances – Data Fields

- Consider the class `D` shown on the next slide which has a single `protected int` instance variable `d`. Because the variable is protected, classes which extend `D` have direct access to it. The default constructor in `D` explicitly sets `d = 0`. The other `D` constructor sets `d` to the value of its parameter v.

- We'll use this class `D` for the next couple of examples.

Inheritance Nuances – Data Fields (cont.)

```
public class D {
  //D instance variable
  protected int d;

  //D(): default constructor
  public D(){
     d = 0;
  }

  //D(): specific constructor
  public D(int v){
     d = v;
  }

  //printD(): facilitator
  public void printD(){
     System.out.println("Value of d in D: " + d);
     System.out.println();
  }
}
```

Inheritance Nuances – Data Fields (cont.)

Although class `E` extends `D`, it does not introduce any new instance variables. However, class `E` does define a single constructor and a method `printE()`.

```
public class E extends D {
  //E(): specific constructor
  public E(int v) {
    d = v;
    super.d = v*100;
  }

  //printE(): facilitator
  public void printE(){
     System.out.println("Value of d in D: " + super.d);
     System.out.println("Value of d in E: " + this.d);
     System.out.println();
  }
}
```

For class E, both d and `super.d` access superclass instance variable d

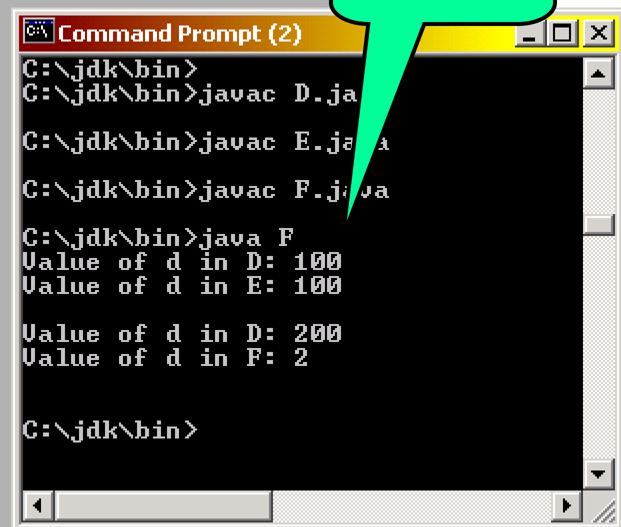
Inheritance Nuances – Data Fields (cont.)

- In contrast, class `F` both extends class `D` and introduces a new instance variable.
  - Because the new variable has the same name `d` as the superclass instance variable, the superclass instance variable is *hidden* in class `F`.
- Class F also defines a constructor, facilitator, and class method main() which will enable the class to serve as an application program.

Inheritance Nuances – Data Fields (cont.)

```
public class F extends D {
  //F instance variable
  int d;  //note default status of d
  //F(): specific constructor
  public F(int v){
     d = v;
     super.d = v*100;
  }
  //printF(): facilitator
  public void printF(){
     System.out.println("D's d: " + super.d);
     System.out.println("F's d: " + this.d);
     System.out.println();
  }
  //main(): application entry point
  public static void main(String[] args){
     E e = new E(1);
     F f = new F(2);
     e.printE();
     f.printF();
     return;
  }
}
```

Output



Inheritance Nuances – Data Fields (cont.)

Because class E does not define any instance variables itself, the instance variable it manipulates is its superclass's instance variable.

```
//E(): specific constructor
public E(int v) {
  d = v;
  super.d = v*100;
}
```

Two separate modifications of superclass's instance variable d

The fact that d and super.d are referencing the same variable means that E's facilitator prints the same value twice.

```
public void printE(){
  System.out.println("D's d: " + super.d);
  System.out.println("E's d: " + this.d);
  System.out.println();
}
```

In the case of the E object referenced by e from F's `main()` method: `E e = new E(1)`, the value 100 is printed by both of these statements.

Inheritance Nuances – Data Fields (cont.)

Because class F defines an instance variable d, that definition results in the superclass instance variable being hidden. Inside an F method, the expression d refers to the F instance variable and not the superclass instance variable.

```
//F(): specific constructor
 public F(int v){
    d = v;
    super.d = v*100;
 }
```

modifies this.d

modifies super.d
superclass's d

Thus, the two assignments in the constructor manipulate different instance variables. When the constructor completes, the value of the object's F variable d is equal to v and the value of the object's superclass variable d is 100*v.

Inheritance Nuances – Typing

- Consider the class `X` shown on the next slide which has a default constructor, a `boolean` class method `isX()` that reports whether its parameter `v` is of type `X`, and a `boolean` class method `isObject()` that reports whether its parameter `v` is of type `Object`.

```
public class X {
   //default constructor
   public X() {
      //no body necessary
   }
   //isX(): class method
   public static boolean isX(Object v){
      return (v instanceof X);
   }
   //isObject(): class method
   public static boolean isObject(X v){
      return (v instanceof Object);
   }
}
```

Inheritance Nuances – Typing (cont.)

- Now consider program `Y.java` shown on the next slide.
- Class `Y` extends class `X`. Besides defining method `main()`, class `Y` provides a default constructor and a `boolean` class method `isY()` that reports whether its parameter `v` is of type `Y`.
- Before reading the analysis that begins on page 31, determine the output of program `Y.java`.

Inheritance Nuances – Typing (cont.)

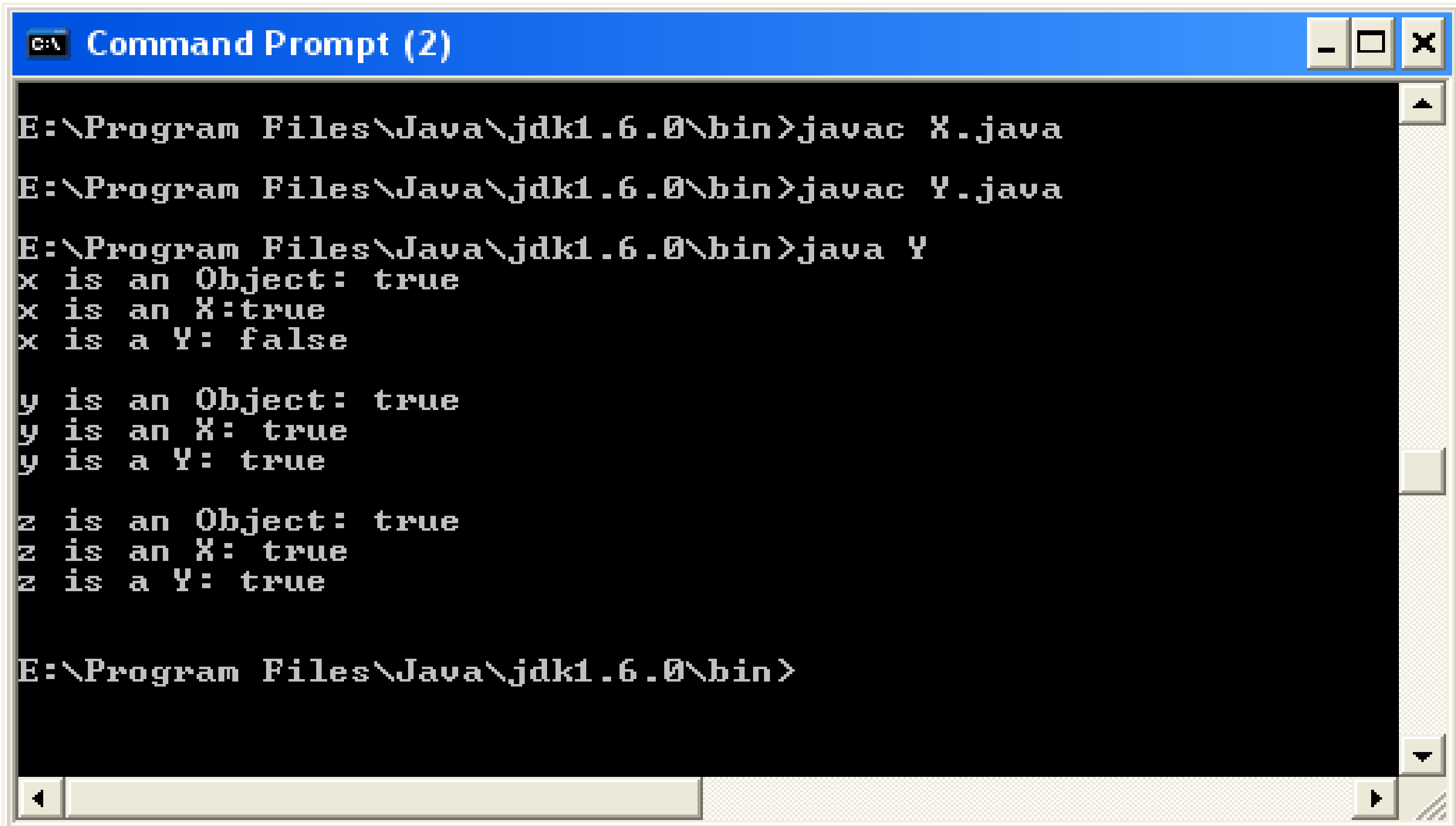
```
public class Y extends X {
  //Y() default constructor
  public Y() {
    //no body needed
  }
  //isY(): class method
  public static boolean isY(Object v){
     return (v instanceof Y);
  }
  //main(): application entry point
  public static void main(String[] args){
     X x = new X();
     Y y = new Y();
     X z = y;
     System.out.println("x is an Object: " + X.isObject(x));
     System.out.println("x is an X: " + X.isX(x));
     System.out.println("x is a Y: " + Y.isY(x));
     System.out.println();
     System.out.println("y is an Object: " + X.isObject(y));
     System.out.println("y is an X: " + X.isX(y));
     System.out.println("y is a Y: " + Y.isY(y));
     System.out.println();
     System.out.println("z is an Object: " + X.isObject(z));
     System.out.println("z is an X: " + X.isX(z));
     System.out.println("z is a Y: " + Y.isY(z));
     System.out.println();
     return;
  }}
```

Inheritance Nuances – Typing (cont.)

- The definition of variable `x` assigns it to a new object of type `X`. Because all classes in Java are extensions of the class `Object`, both `X.isObject(x)` and `X.isX(x)` report true.
- Class `X` is not extended directly or indirectly from class `Y`, so `Y.isY(x)` reports false.
- Because variable `y` is defined to be of class type `Y`, where `Y` is a subclass of `X`, each of `X.isX(y)`, `Y.isY(y)`, and `X.isObject(y)` report true.
- Although the apparent type of `z` is `X`, the object referenced by `z` is the same object that `y` references. Therefore, variable `z` references a `Y` object and the tests regarding `z` report the same results as the tests regarding `y`.

Inheritance Nuances – Typing (cont.)

Output of program Y.java



Inheritance Nuances – Late Binding

Consider the following class `L`.

```
public class L {
   //L(): default constructor
   public L(){
      //no body necessary
   }

   //f(): facilitator
   public void f(){
      System.out.println("Using method f() in L");
      g();
   }

   //g(): facilitator
   public void g(){
      System.out.println("Using method g() in L");
   }
}
```

Inheritance Nuances – Late Binding (cont.)

- Now consider class `M` that extends `L`. Together classes `L` and `M` further demonstrate the power of polymorphism in Java.

```
public class M extends L {
   //M(): default constructor
   public M(){
      //no body necessary
   }

   //g(): facilitator
   public void g(){
      System.out.println("Using method g() in M");
   }

   //main(): application entry point
   public static void main(String[] args) {
      L l = new L();
      M m = new M();
      l.f();
      m.f();
      return;
} }
```

Inheritance Nuances – Late Binding (cont.)

- The statements of interest in main() are:

```
l.f();
m.f();
```

- Because class `M` does not override superclass method `f()`, both `f()` invocations are invocations of `L`'s method `f()`. However, the invocations produce different results!
  - The invocation `l.f()` causes statements

```
System.out.println("Using method f() in L");
g();
```

  from the `L` definition of `f()` to be executed with respect to the object referenced by variable `l`. Because `l` references an `L` object, it is `L` method `g()` that is invoked. This produces the first two lines of output from the program.

Inheritance Nuances – Late Binding (cont.)

- The invocation m.f() again causes the statements:

```
System.out.println("Using method f() in L");
g();
```

from the L definition of f() to be executed. They are executed with respect to the object referenced by variable m. Because m references an M object, it is M method g() that is invoked. This produces the last two lines of output from the program.

Inheritance Nuances – Late Binding (cont.)

Output from M.java

Command Prompt (2)

```
E:\Program Files\Java\jdk1.6.0\bin>javac L.java

E:\Program Files\Java\jdk1.6.0\bin>javac M.java

E:\Program Files\Java\jdk1.6.0\bin>java M
Using method f() in L
Using method g() in L
Using method f() in L
Using method g() in M

E:\Program Files\Java\jdk1.6.0\bin>_
```

Inheritance Nuances – Finality

- Just as Java permits a constant data field to be defined through the use of the keyword `final`, it also permits `final` methods and classes.
- A **`final class`** is a class that cannot be extended.
- A **`final method`** is a method that cannot be overridden.
- The developer of a class might make it `final` for economic or security reasons. If clients have access only to the `.class` version of a final class, then they must look to the developer for additional features. As another example, a class or method may be crucial to a system. By declaring its finality, it is harder to tamper with the system by introducing classes that override the existing code.

Inheritance Nuances – Finality (cont.)

- Consider class U which is a final class. Therefore, class U cannot be extended.

```
final public class U {
   //U(): default constructor
   public U(){
      //no body necessary
   }

   //f(): facilitator
   public void f(){
      System.out.println("Method f() can't be overridden: U is final");
   }
}
```

Inheritance Nuances – Finality (cont.)

- Consider class V which contains a method f() which is declared as final. Therefore, method f() cannot be overridden if V is extended.

```
public class V {
   //V(): default constructor
   public V(){
      //no body necessary
   }

   //f(): facilitator
   final public void f(){
      System.out.println("Method f() can't be overridden: it is final");
   }
}
```

Abstract Base Classes

- In program development, a need sometimes arises for defining a superclass where for some of its methods there are no sensible definitions; that is, it is necessary to make some methods part of the superclass so that other code can exploit Java's polymorphic capabilities.

- Such classes are known as *abstract* classes.

Abstract Base Classes (cont.)

- For example, in developing a geometric shape hierarchy, a suitable superclass GeometricObject for the hierarchy might have two data fields:
  - **`Point position`**
    - defining the upper northwest corner of the shape's bounding box.
  - **`Color color`**
    - defining the color of the shape.
- The following methods are reasonable for the superclass GeometricObject. You might think of some others that would also be reasonable to include.

Abstract Base Classes (cont.)

- `Point getPosition()`
  - returns the upper northwest corner of the shape's bounding box.
- `void setPosition(Point p)`
  - sets the upper northwest corner of the shape's bounding box to `p`.
- `Color getColor()`
  - returns the color of the shape.
- `void setColor(Color c)`
  - sets the color of the shape to `c`.
- `void paint(Graphics g)`
  - renders the shape in graphics context `g`.

Abstract Base Classes (cont.)

- We can only define sensible implementations for `getPosition()`,`setPosition()`, `getColor()`, and `setColor()`.

```
//getPosition(): return object position
Point getPosition(){
   return position;
}
//setPosition(): update object position
void setPosition(Position p){
    position = p;
}
//getColor(): return object color
Color getColor(){
    return color;
}
//setColor(): update object color
void setColor(Color c){
     color = c;
}
```

Abstract Base Classes (cont.)

- Method `paint()` has no sensible implementation because it is shape-specific. The rendering of a rectangle is different from the rendering of a line, which is different from the rendering of a circle.

- Because there is no sensible implementation for `paint()`, it makes sense to use the Java modifier **`abstract`** to make `GeometricObject` an **`abstract`** class and to make its method `paint()` an **`abstract`** method.

Abstract Base Classes (cont.)

- The keyword **`abstract`** at the start of a class definition indicates that the class can be instantiated; i.e., you cannot create directly a new `GeometricObject`.
- The keyword **`abstract`** at the start of a method definition indicates that the definition of the method will not be supplied. Non-abstract subclasses of the abstract superclass *must* provide their own definitions of the abstract method, which would not have been the case if the superclass had instead defined the method with an empty statement list for its method body.
- The complete definition of the abstract class `GeometricObject` appears in the next slide.

```java
//GeometricObject:  abstract superclass for geometric objects
import java.awt.*;

abstract public class GeometricObject{
   //instance variables
   Point position;
   Color: color;

   //getPosition(): return object position
   Point getPosition(){
      return position;
   }
   //setPosition(): update object position
   public void setPosition(Position p){
       position = p;
   }
   //getColor(): return object color
   public Color getColor(){
       return color;
   }
   //setColor(): update object color
   public void setColor(Color c){
        color = c;
   }
   //paint(): render the shape to a graphics context g
   abstract public void paint(Graphics g);
}
```

Extending Abstract Base Classes

- In the next couple of slides we'll extend the abstract `GeometricObject` class with two subclasses called `Box` and `Circle`.
- Because classes `Box` and `Circle` both define a `paint()` method, the classes are non-abstract and can be instantiated.

```
Circle c = new Circle(1, new Point(0,0), Color.blue);
Box r = new Box(1, 2, new Point(3,4), Color.red);
```

Box Class

```
//Box:  rectangle shape representation
import java.awt.*;

public class Box extends GeometricObject{
   //instance variables
   int length;
   int height;

   //Box(): default constructor
   public Box(){
      this(0,0, new Point(), Color.black);
   }
   //Box(): specific constructor
   public Box(int l, int h, Point p, Color c){
       setLength(l);
       setHeight(h);
       setPosition(p);
       setColor(c);
   }
   //getLength(): get the rectangle length: accessor
    public int getLength(){
       return length;
   }
```

```
    //getHeight(): get the rectangle height: accessor
    public int getHeight(){
         return height;
    }
    //setLength(): rectangle length mutator
    public void setLength(int l){
       length = l;
    }
    //setHeight: rectangle height mutator
    public void setHeight(int h){
        height = h;
    }
    //paint(): render the rectangle in graphics context g
     public void paint(Graphics g){
        Point p = getPosition();
        Color c = getColor();
        int l = getLength();
        int h = getHeight();
        g.setColor(c);
        g.fillRect((int) p.getX(), (int) p.getY(), l, h);
    }
}
```

Circle Class

```
//Circle:  circle shape representation
import java.awt.*;

public class Circle extends GeometricObject{
   //instance variable
   int radius;

   //Circle(): default constructor
   public Circle(){
      this(0, new Point(), Color.black);
   }
   //Circle(): specific constructor
   public Circle(int r, Point p, Color c){
       setRadius(r);
       setPosition(p);
       setColor(c);
   }
   //getRadius(): get the radius of the circle: accessor
    public int getRadius(){
       return radius;
   }
```

```
    //setRadius(): circle radius mutator
    public void setRadius(int r){
       radius = r;
    }

    //paint(): render the circle in graphics context g
     public void paint(Graphics g){
        Point p = getPosition();
        Color c = getColor();
        int r = getRadius();
        g.setColor(c);
        g.fillOval((int) p.getX(), (int) p.getY(), r, r);
    }
 }
```

More On Abstract Base Classes

- Abstract classes are valid types. Therefore, you can initialize a variable of an abstract type to reference an existing subclass object of that type.

```
GeometricObject g = c;//a circle is a GeometricObject
```

- You can also use an abstract base type to define a polymorphic method.

```
public void renderShapeInBlue(Graphics g, GeometricObject s){
   g.setColor(Color.blue);
   s.paint(g); //drawing is based on s’s subclass type
}
```

  – This method will set the color for the current graphical context `g` to blue and then draw the object referenced by s in that context. Because Java invokes the `paint()` method for the type of object to which parameter `s` refers, the type of drawing depends on the `GeometricObject` subclass type of `s`.

Interfaces

- In addition to allowing programmer to define abstract classes, Java also allows the programmer to define **`interfaces`**.
- An **`interface`** is not a class but is instead a partial template of what must be in a class that *implements* the interface. Every method in the interface must be implemented by any class that implements the interface.
- An interface is a Java type and can be used as such.

```
public interface name {
       //constants


       //method declarations
}
```

All variables are either explicitly or implicitly public class constants (i.e., public static final)

All listed methods are either explicitly or implicitly public

Interfaces (cont.)

- An interface definition differs from an abstract class definition in three important ways:

1. An interface cannot specify any method implementations.
2. All of the methods of an interface are public.
3. All of the variables defined in an interface are public, final, and static.

Interfaces (cont.)

- Why use an interface when an abstract class offers greater flexibility? There are two primary reasons:

  1. Java allows a class to implement more than one interface, whereas a class can extend only one superclass.
  2. An interface is not a class, it is not part of a class hierarchy. Two unrelated classes can implement the same interface with objects of those unrelated classes having the same type – the interface their class types implement.

Interfaces (cont.)

- As an example, consider the interface `Colorable` shown below.
- Implementing the interface requires two methods – `getColor()` and `setColor()`.

```
package geometry;
import java.awt.*;

public interface Colorable {
   //getColor(): color accessor
   public Color getColor();
   //setColor(): color mutator
   public void setColor(Color c);
}
```

- The next couple of pages are reworkings of the classes ColoredPoint and Colored3DPoint from Day 19.

The `ColorablePoint` Class

```
//representation of a colored 2-d point
package geometry;
import java.awt.*;

public class ColorablePoint extends Point implements Colorable{
   //instance variable
   Color color;

   //ColorablePoint():  default constructor
   public ColorablePoint(){
      super();
      setColor(Color, blue);  //default color
   }

   //ColorablePoint(): specific constructor
   public ColorablePoint(int x, int y, Color c){
      super(x,y);
      setColor(c);
   }
```

extends Point class

implement Colorable interface

```
//getColor(): color property accessor
public Color getColor(){
   return color;
}

//setColor(): color property mutator
public void setColor(Color c){
   color = c;
}


//toString(): string representation facilitator
public String toString(){
   Color c = getColor();
   return “[“ + super.toString() + “,” c.toString() + “]”;
}
```

```java
//equals(): equality facilitator
public boolean equals(Object v){
   if (v instanceof ColorablePoint){
       Color c1 = getColor();
       Color c2 = ((ColorablePoint) v).getColor();
       return super.equals(v) && c1.equals(c2);
   }
   else {
       return false;
   }
}

//clone(): cloning facilitator
public Object clone(){
  return new ColorablePoint(x,y,getColor());
}
}//end class ColorablePoint
```

The Colorable3DPoint Class

```
//representation of a colored 3-d point
package geometry;
import java.awt.*;

public class Colorable3DPoint extends ThreeDPoint implements Colorable{
   //instance variable
   Color color;

   //Colorable3DPoint():  default constructor
   public Colorable3DPoint(){
      setColor(Color, blue);  //default color
   }

   //Colorable3DPoint(): specific constructor
   public Colorable3DPoint(int a, int b, int c, Color d){
      super(a,b,c);
      setColor(d);
   }
```

```
//getColor(): color property accessor
public Color getColor(){
   return color;
}

//setColor(): color property mutator
public void setColor(Color c){
   color = c;
}


//toString(): string representation facilitator
public String toString(){
   Color d = getColor();
   return "[" + super.toString() + c.toString() + "]";
}
```

```java
    //equals(): equality facilitator
    public boolean equals(Object v){
       if (v instanceof Colorable3DPoint){
           Color c1 = getColor();
           Color c2 = ((Colorable3DPoint) v).getColor();
           return super.equals(v) && c1.equals(c2);
       }
       else {
           return false;
       }
    }

    //clone(): cloning facilitator
    public Object clone(){
      return new Colorable3DPoint(a,b,c,getColor());
    }
}//end class Colored3DPoint
```

Interfaces (cont.)

- `ColorablePoint` is a reworking of the class `ColoredPoint` from Day 19 notes. Similarly, `Colorable3DPoint` is a reworking of the class `Colored3DPoint` also from Day 19.

- Except for the cosmetic differences between the two versions of each of these classes, the only other difference is that `ColorablePoint` and `Colorable3DPoint` implement interface `Colorable`.

Interfaces (cont.)

- Because both `ColorablePoint` and `Colorable3DPoint` implement the `Colorable` interface, a polymorphic code segment such as the following is possible:

```
ColorablePoint u = new ColorablePoint();
Colorable3DPoint v = new Colorable3DPoint();
Colorable w = u;
w.setColor(Color.black);
w = v;
w.setColor(Color.red);
```

ColorablePoint method setColor() is invoked

Colorable3DPoint method setColor() is invoked

Interfaces (cont.)

- In the polymorphic segment of the previous slide, variable `w` used in the invocation of two different `setColor()` methods.
- This polymorphic circumstance is possible because points `u` and `v` share the common interface type `Colorable`.

Interfaces (cont.)

- A polymorphic method can be defined to take advantage of interface-implemented commonality.
- Consider the following class `Blue` with polymorphic class method `setBlue()`.

```
public class Blue {
   public static void setBlue(Colorable p){
       p.setColor(Color.blue);
   }
}
```

Parameter can be either a ColorablePoint or a Colorable3DPoint.

invokes setColor() method in ColorablePoint class

```
ColorablePoint a = new ColorablePoint();
Colorable3DPoint b = new Colorable3DPoint();
Blue.setBlue(a);
Blue.setBlue(b);
```

invokes setColor() method in Colorable3DPoint() class

Interfaces (cont.)

- Even though both `ColoredPoint` and `Colored3DPoint` implement `getColor()` and `setColor()` methods, comparable code segments with `ColoredPoint` and `Colored3DPoint` being used are not possible. WHY?

- Because, `ColoredPoint` and `Colored3DPoint` are not of type `Colorable`, whereas `ColorablePoint` and `Colorable3DPoint` are, as the example on the next slide illustrates.

Interfaces (cont.)

```
ColoredPoint c = new ColoredPoint();

Colored3DPoint d = new Colored3DPoint();

Colorable e = c;

e.setColor(Color.cyan);

e = d;

e.setColor(Color.green);

Blue.setBlue(c);

Blue.setBlue(d);
```

Illegal: c does not reference a Colorable object

Illegal: d does not reference a Colorable object

Illegal: c does not reference a Colorable object

Illegal: d does not reference a Colorable object