CS 213 : Software Methodology

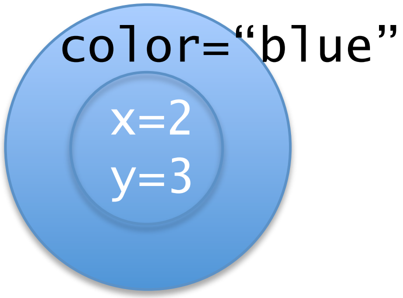
Spring 2016

Lecture 2: Jan 21
OOP/Inheritance/Static and Dynamic Types

Inheritance – Why call super(…)?

Think of a subclass instance having two parts: the inherited part from the superclass, and the special part of the subclass

```
ColoredPoint cp =
   new ColoredPoint(
             2,3,”blue”);
```

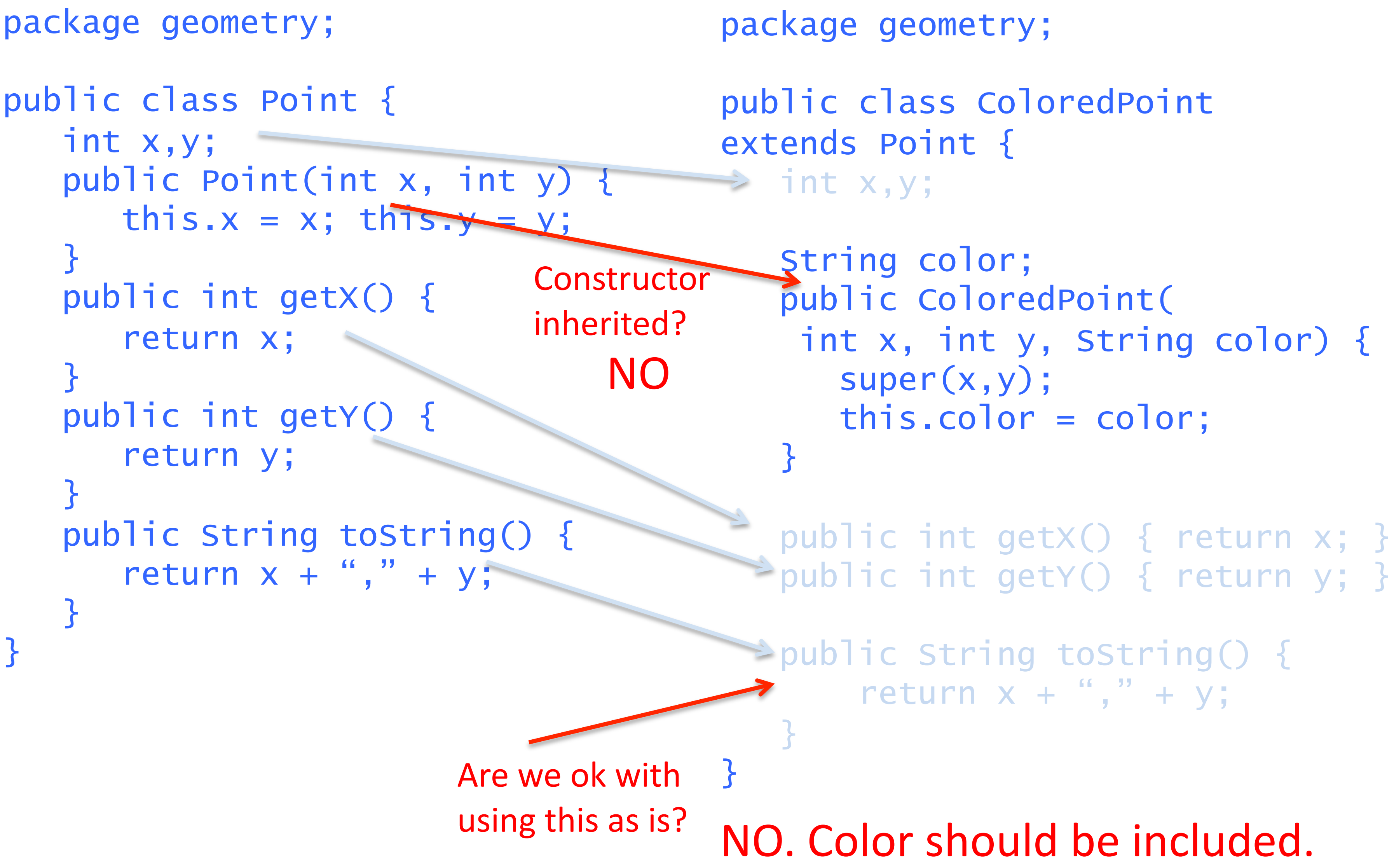


Initialization of the superclass part is best done by a superclass constructor, no point in reinventing the wheel (Code REUSE) – Thus the call to the superclass constructor, to FIRST initialize the superclass part, then code to initialize the subclass part.

Q. When a `ColoredPoint` instance is created, is an inner `Point` instance created as well?

NO.
It’s CODE reuse, not instance reuse

Inheritance – Fields and Methods



```
package geometry;

public class Point {
    int x,y;
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public String toString() {
        return x + “,” + y;
    }
}
```

```
package geometry;

public class ColoredPoint
extends Point {
    int x,y;

    String color;
    public ColoredPoint(
     int x, int y, String color) {
       super(x,y);
       this.color = color;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public String toString() {
          return x + “,” + y;
    }
}
```

Constructor inherited?

NO

Are we ok with using this as is?

NO. Color should be included.

Inheritance – Overriding Method

```
package geometry;

public class ColoredPoint
extends Point {
   int x,y;

   String color;
   public ColoredPoint(
    int x, int y, String color) {
      super(x,y);
      this.color = color;
   }

   public int getX() { return x; }
   public int getY() { return y; }

   public String toString() {
       return x + “,” y + “,” + color;
   }
}
```

This implementation overrides the inherited code

Inheritance – Reusing inherited method code in overriding method

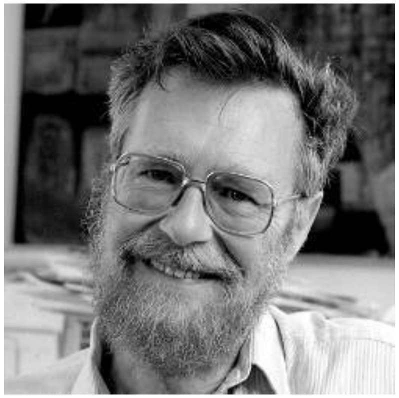
```
package geometry;

public class ColoredPoint
extends Point {
   int x,y;

   String color;
   public ColoredPoint(
    int x, int y, String color) {
      super(x,y);
      this.color = color;
   }

   public int getX() { return x; }
   public int getY() { return y; }

   public String toString() {
       return super.toString() + “,” + color;
   }
}
```

Calls inherited method

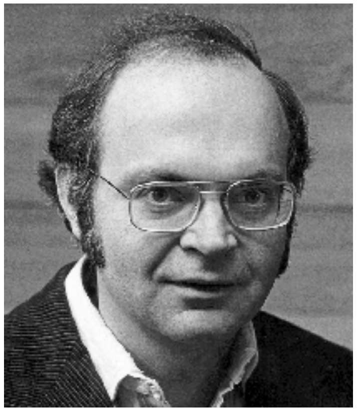
Speaking of good and bad programming practices….

```
FOR I = 1 to 10
    ...
    IF ... THEN GOTO 10
    ...
NEXT I
10 ...
```

FORTRAN code



Dijkstra - goto is harmful

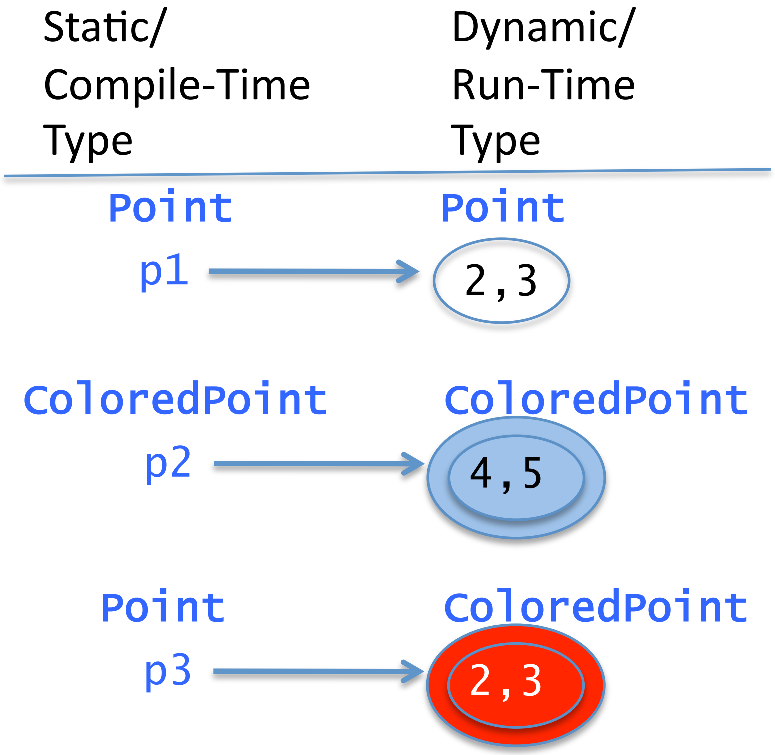


Knuth - Depends

I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE

Static and Dynamic Types

```
public class PointApp {
   public static void
   main(String[] args) {

      Point p1 = new Point(2,3);


      ColoredPoint p2 =
        new ColoredPoint(4,5,"blue");


      Point p3 =
        new ColoredPoint(2,3,"red");
}
```



Static/ Compile-Time Type

Dynamic/ Run-Time Type

Point

Point

p1

2,3

ColoredPoint

ColoredPoint

p2

4,5

Point

ColoredPoint

p3

2,3

Every ColoredPoint is a Point (just like every Student is a Person) – so any ColoredPoint instance (dynamic type) can be referred to by a Point variable (static type)

Dynamic Binding

```
public class PointApp {
   public static void
   main(String[] args) {
      Point p1 = new Point(2,3);
      ColoredPoint p2 =
        new ColoredPoint(4,5,”blue”);
      Point p3 =
        new ColoredPoint(2,3,”red”);

      System.out.println(p2.getColor()); // ? “blue”

      System.out.println(p3.getX()); // ? 2

      System.out.println(“p3 = “ + p3); // ?  “p3 = 2,3,red”

}
```

Dynamic Binding

Static type of p3 is `Point`, but dynamic type (type of instance it points to) is `ColoredPoint`. So, the `p3.toString()` static call is bound to the dynamic type, `ColoredPoint`. This results in the overridding version of `toString`() being executed.

Static and Dynamic Types

```
public class PointApp {
   public static void
   main(String[] args) {
      ...
      ColoredPoint p4 = new Point(5,6); // ?
```

WILL NOT COMPILE
Every Point (RHS) is
NOT a ColoredPoint
(LHS), so a Point instance
cannot be referenced
by a ColoredPoint variable

```
      Point p5 = new ColoredPoint(1,2,”green”);
      System.out.println(p5.getColor()); // ?
}
```

WILL NOT COMPILE
Because the static type of
p5 is Point, ONLY members of
Point class can be syntactically
referenced by p5. Since
getColor is not in the Point
class, compiler flags error

Inheritance - Private Fields

```
public class Point {
   private int x,y;
   ...
}
```

```
public class ColoredPoint extends Point {
   // x and y inherited but HIDDEN
   ...
   public int getX() { // override
      return x;
   }
}
```

WILL NOT COMPILE
because x is hidden

Inheritance - Private Fields

```
public class Point {
   private int x,y;
   ...
}
```

```
public class ColoredPoint extends Point {
   // x and y inherited but HIDDEN
   ...  // getX() is NOT overridden
}
```

```
public class PointApp {
   public static void
   main(String[] args) {

      ColoredPoint cp = new ColoredPoint(4,5,”blue”);

      System.out.println(cp.x); // ? WILL NOT COMPILE

      System.out.println(cp.getX()); // ? 4

}
```

Inherited `getX()` method is able to access the `x` field

Inheritance - Static Members

```
public class Supercl {
   static int x;
   public static void m() {
       System.out.println(
         “in class Supercl”);
       }
       ...
   }
   ...
}
```

```
public class Subcl
extends Supercl { }
```

```
public class StaticTest {
   public static void main(String[] args) {
       Supercl supercl = new Supercl();
       System.out.println(supercl.x);  // ? 0
       supercl.m(); // ? “in class Supercl”
       Subcl subcl = new Subcl();
       System.out.println(subcl.x); // ? 0 – inherited from Supercl
       subcl.m(); // ? “in class Supercl” – inherited from Supercl
   }
}
```

Inheritance - Static Fields

```
public class Supercl {
   static int x;
   public static void m() {
       System.out.println(
         “in class Supercl”);
       }
   }
}
```

```
public class Subcl
extends Supercl {
   int x=3;
}
```

```
public class StaticTest {
   public static void main(String[] args) {
      Subcl subcl = new Subcl();
      System.out.println(subcl.x); // ? 3 – instance field x
      Supercl supercl = new Subcl();
             ↑                  ↑
        static type       dynamic type
      System.out.println(supercl.x); // ? 0 – inherited static field x  ! ! !
   }
}
```

INHERITED STATIC FIELDS ARE STATICALLY BOUND (TO REFERENCE TYPE), NOT DYNAMICALLY BOUND (TO INSTANCE TYPE)

Static Method Call Binding

```
public class Sorter {

   public static void
   sort(String[] names) {
       System.out.println(
         "simple sort";
       }
   }
}
```

```
public class IllustratedSorter
extends Sorter {

   // override
   public static void
   sort(String[] names)
       System.out.println(
         "illustrated sort";
       }
   }
}
```

```
   Sorter p = new IllustratedSorter();
```

static type

dynamic type

```
   p.sort(); // ?
```

"simple sort"

sort() is statically bound to p, meaning since Sorter is the static type of p, the sort() method in Sorter is called

Alan Kay on Learning/CS

https://www.youtube.com/watch?v=Ud8WRAdihPg

Object Class

- Root of java class hierarchy
  - Every class ultimately is a subclass of `java.lang.Object`
- Methods in `Object` you have seen – all of these are inherited by ANY class (since every class is implicitly a subclass of `Object`):
  - `equals`: compares address of objects
  - `toString`: returns address of object
  - `hashCode`: returns hash code value for object
- Must generally override `equals` and `toString`

Writing code banking on equals being there

- Because the `Object` class defines equals, you—as an algorithm designer—can *independently* write code to compare two objects using the `equals` method, and the code will compile (And when an application sends in, say, `Point` objects, the overridden equals will be called)

```
public class Searcher {
   ...
   public static <T> boolean
   sequentialSearch(T[] list, T target) {
       for (int i=0; i < list.length; i++) {
           if (target.equals(list[i]) {
              return true;
           }
       }
       return false;
   }
   ...
}
```

Don't know what T will be at runtime, but it is guaranteed to have the `equals` method

Overriding equals

Boiler-plate way to override equals (e.g. Point):

```
public class Point {
   int x,y;
   ...
   public boolean equals(Object o) {
      if (o == null || !(o instanceof Point)) {
         return false;
      }

      Point other = (Point)o;


      return x == other.x && y == other.y;
   }
   ...
}
```

1 Header must be same as in Object class

2 Check if actual object (runtime) is of type Point, or a subclass of Point

3 Must cast to Point type before referring to fields of Point

4 Last part is to implement equality as appropriate (here, if x and y coordinates are equal)