Object-Oriented Programming

CSCI-UA 0470-001
Class 7
Instructor: Randy Shepherd

Inheritance & Method Overriding

Why Inheritance?

- There are several goals..
  - *Modularity* - do one thing and do it well. moreover, separation of concerns
  - *Reusability* - ability to leverage existing code
  - *Extensibility* - code can later be extended to handle cases not considered originally

Use Case

- Remember our Point implementations?

- Suppose we want to be able to model points that have a color. How could we achieve this?

Possible Approach 1

- Modify existing Point implementation
- Implications:
  - all instances of a Point must have a color
  - less flexible and wastes space if cases where we do not care about color
  - each time we add a property code gets more complex and we might be adding bugs

Possible Approach 2

- Copy the code and create a new class with it that contains both point and color data and methods

- Implications:
  - Effort to rewrite methods of Point to correspond to ColorPoint
  - Worst case, totally different approach
  - Any change to code related to points has to be changed in two places
  - Might copy bugs, which can make debugging very confusing
  - You cannot reuse any code that operates on Point

Possible Approach 3

- Inheritance!
  - ColorPoint (the subclass) inherits from Point (the superclass)
  - The subclass is guaranteed to have all of the behavior and properties of the superclass in addition to newer, more specific properties
  - Anywhere a Point is expected, a ColorPoint can be used.

ColorPoint

- This is called the "**is a**" relationship - e.g. every ColorPoint *is a* Point, but not vice versa
- This is an example of subtype polymorphism
- Abides *substitution principle*, anywhere a superclass is expected, a subclass can be used.

Better Approach

- Using inheritance leads to better correctness, easier debugging, better future-proofing. Less work overall.

Java

java.lang.Object

- Object is the root of the class hierarchy
- Every class has Object as a superclass
- Every class implicitly inherits from it. (There is no need to write *extends Object*)
- Moreover, all objects, including arrays, implement the methods of this class.

Object's Methods

- There are 11 methods, we will support 4 of them for our translator.

```
1   // Indicates whether some other object is "equal to" this one.
2   boolean equals(Object obj)
3
4   // Returns the runtime class of this Object.
5   Class<?> getClass()
6
7   // Returns a hash code value for the object.
8   int hashCode()
9
10  // Returns a string representation of the object.
11  String toString()
```

Object's Methods

```
1  Object o = new Point(0, 1, 2, 3);
2
3  String ss = o.toString();
4  String os = new Object().toString();
5
6  int d1 = o.getDistanceFrom(Point.ORIGIN);
7  int d2 = ((Point) o).getDistanceFrom(Point.ORIGIN);
```

- Is line 1 valid Java?

Object's Methods

```
1  Object o = new Point(0, 1, 2, 3);
2
3  String ss = o.toString();
4  String os = new Object().toString();
5
6  int d1 = o.getDistanceFrom(Point.ORIGIN);
7  int d2 = ((Point) o).getDistanceFrom(Point.ORIGIN);
```

- Yes, everywhere there is an Object a Point may appear.

- Again, this is the *substitution principle.*

Object's Methods

```
1  Object o = new Point(0, 1, 2, 3);
2
3  String ss = o.toString();
4  String os = new Object().toString();
5
6  int d1 = o.getDistanceFrom(Point.ORIGIN);
7  int d2 = ((Point) o).getDistanceFrom(Point.ORIGIN);
```

- Is line 3 valid Java?

Object's Methods

```
1  Object o = new Point(0, 1, 2, 3);
2
3  String ss = o.toString();
4  String os = new Object().toString();
5
6  int d1 = o.getDistanceFrom(Point.ORIGIN);
7  int d2 = ((Point) o).getDistanceFrom(Point.ORIGIN);
```

- Yes! *o* is statically declared to be an Object, which has a toString method.

- Compiler knows this must be an Object and have all of Object's methods

- This will use Point's toString() method because Point overwrites the toString() method in Object, *i.e. it is the most specific version of toString() available*

Object's Methods

```
1  Object o = new Point(0, 1, 2, 3);
2
3  String ss = o.toString();
4  String os = new Object().toString();
5
6  int d1 = o.getDistanceFrom(Point.ORIGIN);
7  int d2 = ((Point) o).getDistanceFrom(Point.ORIGIN);
```

- Given that, would the variables on line 3 and 4 contain the same value?

Object's Methods

```
1  Object o = new Point(0, 1, 2, 3);
2
3  String ss = o.toString();
4  String os = new Object().toString();
5
6  int d1 = o.getDistanceFrom(Point.ORIGIN);
7  int d2 = ((Point) o).getDistanceFrom(Point.ORIGIN);
```

- No, calling Object().toString() you would get the the name of the object's Java class and the object's hash code ("java.lang.Object@38503429")

Object's Methods

```
1  Object o = new Point(0, 1, 2, 3);
2
3  String ss = o.toString();
4  String os = new Object().toString();
5
6  int d1 = o.getDistanceFrom(Point.ORIGIN);
7  int d2 = ((Point) o).getDistanceFrom(Point.ORIGIN);
```

- Is line 6 valid Java?

Object's Methods

```
1  Object o = new Point(0, 1, 2, 3);
2
3  String ss = o.toString();
4  String os = new Object().toString();
5
6  int d1 = o.getDistanceFrom(Point.ORIGIN);
7  int d2 = ((Point) o).getDistanceFrom(Point.ORIGIN);
```

- No, there is no way to know you are always referencing a Point - you can't use something more generic than required

- You can solve this by explicitly casting a type like the last line in line 7

Object's Methods

```
1  Object o = new Point(0, 1, 2, 3);
2
3  String ss = o.toString();
4  String os = new Object().toString();
5
6  int d1 = o.getDistanceFrom(Point.ORIGIN);
7  int d2 = ((Point) o).getDistanceFrom(Point.ORIGIN);
```

- The cast on line 7 is known as a 'downcast', which is a tricky thing.

- Consider this…

```
Object o = new Object();
String s = (String) o;
```

Takeaway…

- Use the superclass to declare the major methods, i.e. to define your object hierarchy's common functionality

- Use subclasses to specialize the methods for particular use cases

C++

Java Comparison

- C++ has no intrinsic class hierarchy like Java, i.e. there is no equivalent to java.lang.Object

- Inheritance is defined explicitly

- Like Java though, there is a ***is-a*** relationship - every ColorPoint is a Point, or every ColorPoint ***extends*** a Point

- Most of the concepts and principles are the same, but there are some different language constructs

Color.h

- line 8 is an *initializer list*
- The first r in r(r) refers to the field r in the scope of the class.
- The second r in r(r) refers to the value r in the scope of the constructor.

```
1   class Color {
2
3   unsigned char r, g, b;
4
5   //...
6
7   Color(unsigned char r, unsigned char g, unsigned char b)
8    : r(r), g(g), b(b) {}
9
10   unsigned char red() const {
11     return r;
12   }
13
14  // ...
15
16  };
```

Color.h

- The class variables are initialized in the order in which they are declared within the class.

- Moreover, the init order of r, g, b is not dictated by the order of line 8, but by the order of line 3.

- Using initializer lists is considered good style in C++

```
1   class Color {
2
3   unsigned char r, g, b;
4
5   //...
6
7   Color(unsigned char r, unsigned char g, unsigned char b)
8    : r(r), g(g), b(b) {}
9
10   unsigned char red() const {
11     return r;
12   }
13
14  // ...
15
16  };
```

Color.h

- Note that the getters (accessors) for the RGB values are inlined into this header file
- Is this the right thing to do?
- (what did we say goes in .h? how about .cpp?)

```
1   class Color {
2
3   unsigned char r, g, b;
4
5   //...
6
7   Color(unsigned char r, unsigned char g, unsigned char b)
8    : r(r), g(g), b(b) {}
9
10   unsigned char red() const {
11     return r;
12   }
13
14  // ...
15
16  };
```

Color.h

- What is right to inline?
  - Trivial actions like ‘return’
  - Initialization of variables
  - *Basic* operations
- Also, why are the accessors marked ‘const’?

```
1   class Color {
2
3   unsigned char r, g, b;
4
5   //...
6
7   Color(unsigned char r, unsigned char g, unsigned char b)
8    : r(r), g(g), b(b) {}
9
10   unsigned char red() const {
11     return r;
12   }
13
14  // ...
15
16  };
```

Point.h

- A ColorPoint can appear everywhere a Point can appear. (substitution principle)

- Note the virtual keyword. This indicates to the compiler the method can be overridden.

- Once a method is declared virtual, it is a virtual method in all of its subclasses.

```
1   class Point
2     {
3         double coordinates[DIMENSIONS];
4
5         public:
6
7         static const Point ORIGIN;
8
9         Point(double c1 = 0, double c2 = 0, double c3 = 0, double c4 = 0);
10
11        double getCoordinate(int i) const;
12        double getDistanceFrom(const Point& p) const;
13        virtual string toString() const;
14    };
15
16    class ColorPoint : public Point
17    {
18        Color color;
19
20        public:
21
22        ColorPoint(Color color, double c1, double c2, double c3, double c4)
23            : Point(c1, c2, c3, c4), color(color) {}
24
25        Color getColor() const
26        {
27            return color;
28        }
29
30        string toString() const;
31    };
```

Point.h

- Virtual method calls only work in C++ with pointers and references

- Experiment with virtual methods here: https://github.com/nyu-oop/virtual-methods-cpp

- (remember though that we cannot use virtual methods in our translator target language!)

```
1   class Point
2     {
3         double coordinates[DIMENSIONS];
4
5         public:
6
7         static const Point ORIGIN;
8
9         Point(double c1 = 0, double c2 = 0, double c3 = 0, double c4 = 0);
10
11        double getCoordinate(int i) const;
12        double getDistanceFrom(const Point& p) const;
13        virtual string toString() const;
14    };
15
16    class ColorPoint : public Point
17    {
18        Color color;
19
20        public:
21
22        ColorPoint(Color color, double c1, double c2, double c3, double c4)
23            : Point(c1, c2, c3, c4), color(color) {}
24
25        Color getColor() const
26        {
27            return color;
28        }
29
30        string toString() const;
31    };
```

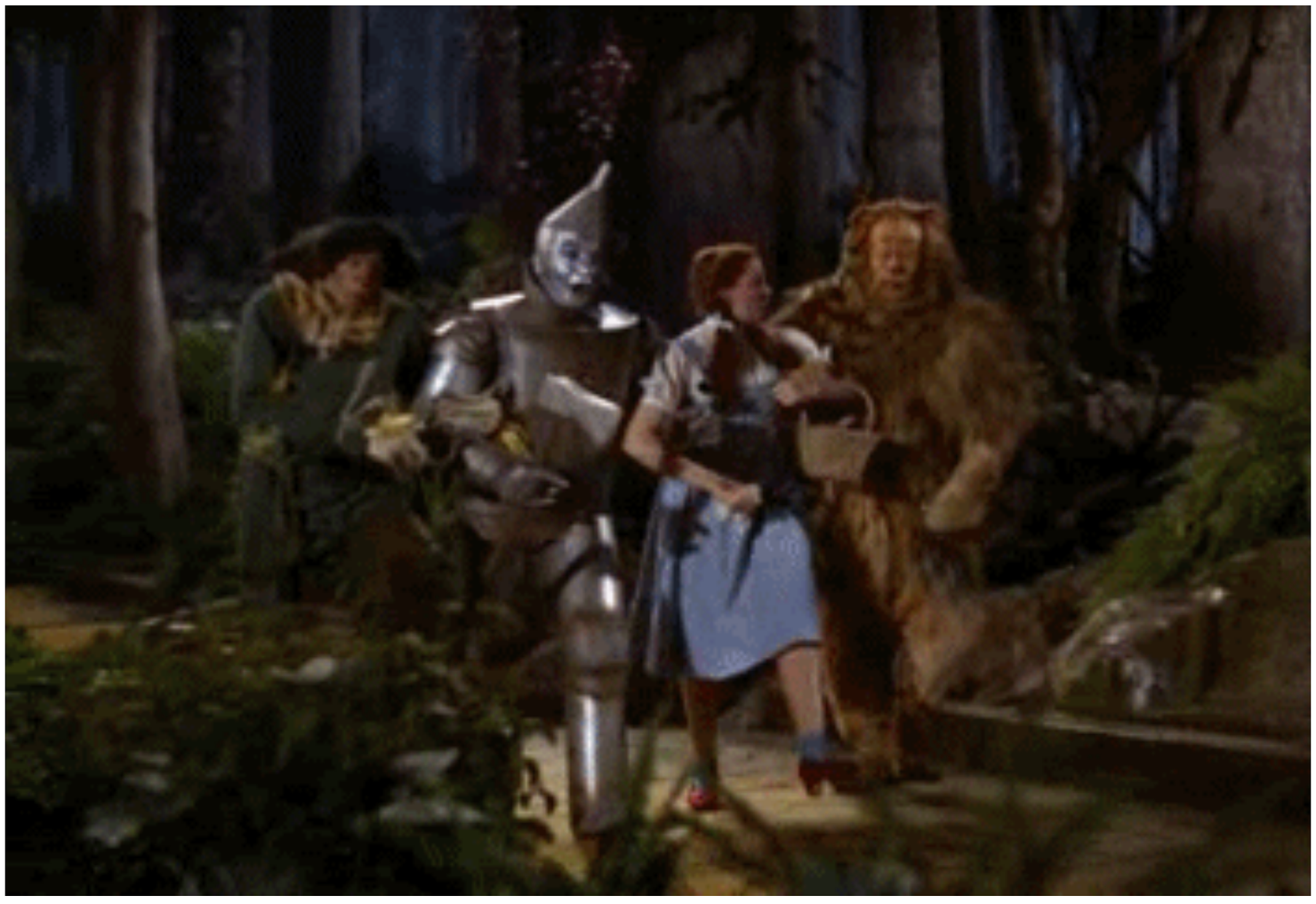
Point.h

- Note the public keyword in the ColorPoint definition
- When you inherit a base class publicly, all members keep their original access specifications
- Thats good enough for this class.

```
1   class Point
2     {
3         double coordinates[DIMENSIONS];
4
5         public:
6
7         static const Point ORIGIN;
8
9         Point(double c1 = 0, double c2 = 0, double c3 = 0, double c4 = 0);
10
11        double getCoordinate(int i) const;
12        double getDistanceFrom(const Point& p) const;
13        virtual string toString() const;
14    };
15
16    class ColorPoint : public Point
17    {
18        Color color;
19
20        public:
21
22        ColorPoint(Color color, double c1, double c2, double c3, double c4)
23            : Point(c1, c2, c3, c4), color(color) {}
24
25        Color getColor() const
26        {
27            return color;
28        }
29
30        string toString() const;
31    };
```

Static Typing, Dynamic Typing and Dynamic Dispatch

(oh my!)



Static Vs Dynamic Types

- For variables p that point to an object, we distinguish between p's static and dynamic types.
    - **Static type**: the type of p known to the compiler at compile time, i.e., from the declaration of p.
    - **Dynamic type**: the actual type of p at runtime determined by the object to which p points.
    - Without inheritance, static and dynamic types are the same
    - With inheritance, the dynamic type is the type of the object the variable was last changed to.

Consider this code..

```
1  List<String> l = null;
2
3  if(new Random().nextInt() % 2 == 0) {
4    l = new ArrayList<>();
5  } else {
6    l = new LinkedList<>();
7  }
8
9  System.out.println("Static type: " + List.class.getName());
10 System.out.println("Dynamic type: " + l.getClass().getName());
```

Virtual Method Dispatch

- In C++, methods are called based on the dynamic type only if they are virtual; otherwise they are called based on the static type

- In Java, all methods are virtual, except private methods and static methods (why?)

- Virtual methods in Java and C++ look for the most specific implementation of a method, i.e. the one furthest down the inheritance chain.

Dynamic Dispatch in C++

- The new operator allocates memory for the object on the heap (like in Java) rather than inline

- The dynamic type of both p and cp is a pointer to ColorPoint

- The static type for p is a pointer of Point and the static type for cp is a pointer to ColorPoint

```
1  Point* p = new ColorPoint(Color::WHITE, 0, 1, 2, 3);
2
3  p->toString();
4
5  ColorPoint* cp = new ColorPoint(Color::RED, 0, 1, 2, 3);
6
7  cp->toString();
8
9  Point::ORIGIN.getDistanceFrom(*cp)
```

Dynamic Dispatch in C++

- Since cp is a pointer to ColorPoint, line 7 will call ColorPoint's toString()

```
1  Point* p = new ColorPoint(Color::WHITE, 0, 1, 2, 3);
2
3  p->toString();
4
5  ColorPoint* cp = new ColorPoint(Color::RED, 0, 1, 2, 3);
6
7  cp->toString();
8
9  Point::ORIGIN.getDistanceFrom(*cp)
```

- line 3 will also call ColorPoint's toString() based on our implementation (the method is declared as virtual and p's dynamic type is ColorPoint*)

- If we did not declare toString virtual, line 3 would call Point's toString() based on its static type.

Follow-up

- Code that should make it easy to see how virtual method dispatch in C++ works is located here..
  - https://github.com/nyu-oop/virtual-methods-cpp
- The C++ implementation of ColorPoint is located here…
  - https://github.com/nyu-oop/color-point-cpp

Reminder

- Translating these types of Java methods to C++ code that ***does not use virtual methods*** is the CORE OF THE PROJECT.