COP 3330 – Object Oriented Programming in Java
Polymorphism Lecture 1

One of the advantages of inheritance in an interpreted language is polymorphism. Literally, this translates to "many forms." Though how Java programs with inheritance that make use of polymorphism may seem confusing, like most of Java, key rules are followed exactly, every time. Once you understand those rules, it allows you to write elegant code that allows for natural reuse of code at many levels. In this lecture, instead of focusing on a practical use, we'll only focus on learning the rules that Java follows.

We'll look at one example in this lecture. This example has three classes: A, B and C, where B inherits from A and C inherits from B. Before we look at either example, let's just review the two principal rules that Java follows in determining "which method" when more than 1 with the same name is available, gets executed:

1. First, Java tries to find a method that matches the signature of the method called in the class of the **reference** of the object upon which the method was called on. Thus, if the call to a method is of the form

o.methodcall(parameters);

then there must be a method call in the class of the reference o OR a class that inherits from o directly or indirectly that matches this method call (using reference types for the decision making.)

If no such method exists, the code will not compile!

So, for example, if the reference of o is of type B, and o is referencing an object of type C, then a method of the signature of the method call must be found in either the Object class, the class A or the class B. If it is ONLY in class C, then this would result in a syntax error.

2. Once we find a method call that matches in the class of the reference of o, *then* we check the type of object to which o is referring. To finalize the method actually *called*, **we start in the class of the object**, and work our way down the hierarchy until we find a method that matches the appropriate signature. We are guaranteed that we'll find such a method because the compiler verified that such a method exists via the class of the reference the method was called on.

Basically, once a method call is accepted as being syntactically valid, then we start over in determining which method call actually runs, at run-time.

This practice, of determining at run-time which method ACTUALLY gets called is called "late-binding".

So, for example, if the reference of o is of type A, and there is a suitable method in class A and o is referencing an object of type C, then we would FIRST look in class C to see if there was a method matching the signature of the method call. (**In checking for this match, we use the types of the *references* of the actual parameters.**) If there is, that is the method that gets called. If not,

we then look to class B. If there is a method in class B that matches, then that is the method that gets called. If not, then we are guaranteed a match for a method in class A, and this is the one that gets called.

3. This really isn't a separate rule, but to reiterate, for parameters, there's no dynamic binding. The type of the reference is what's used to determine what method is syntactically correct AND what method gets called. If no method exists in the current class of this object that uses the parameter reference, then Java goes to the super class. (And continues doing so as needed.)

One interesting note to add to #3 which makes things a bit more confusing:

the instanceof operator can be used to determine if a reference is pointing to an object of a particular class.

Thus, we can call a method that takes in an A reference as a parameter, but then, within that method, if we see that A reference is pointing to a C object, we can then call a new method on that object that calls a method in the C class! If that method doesn't exist in the class A, then we would have to cast the reference to type C, because otherwise, the code would not compile.

Before we move onto our main example for this lecture, let's revisit the Coordinate and ColorCoordinate example.

Example: Coordinate & ColorCoordinate – toString()

The Coordinate class manages an ordered pair of an integer and a character. For simplicity's sake (and to focus on polymorphism), the class only has constructors, accessors, a toString() method and a couple equals() methods.

We will examine two possible scenarios for the toString() method:

(1) It is ONLY defined in Coordinate.
(2) It is defined in both Coordinate and ColorCoordinate.

In the former situation, no matter whether the reference is a Coordinate reference or a ColorCoordinate reference, the toString() in Coordinate will be called. This situation is generally straight-forward. There isn't a choice between multiple toString() implementations to really consider. (It's clear that the toString() in Object won't get called.)

In the latter situation, the type of the actual Object matters. If a Coordinate reference c is referring to a ColorCoordinate object, then c.toString() will invoke the toString() method in the ColorCoordinate class.

In general, the rule is that syntactically, the method call must be valid within the class of the REFERENCE. So, since c is a Coordinate reference, we MUST have a toString() method defined for a Coordinateobject. But, once this is satisfied, then we find that at run-time, if we notice that c is referring to an Object of the inherited(subclass) class, and that inherited(subclass) class overrides that particular method, then that is the one that gets called at run time.

Example: Coordinate & ColorCoordinate – equals

We could potentially define four separate equals methods total

(1) public boolean equals(Coordinate c); // in Coordinate
(2) public boolean equals(ColorCoodinate c); // in Coordinate
(3) public boolean equals(Coordinate c); // in ColorCoordinate
(4) public boolean equals(ColorCoodinate c); // in ColorCoor.

In the chart below, each column refers to one of the four methods above and each row represents a possible scenario of which methods are defined. Here are some of the situations we'll consider:

Situation	Method #1	Method #2	Method #3	Method #4
1	Yes	No	No	No
2	Yes	Yes	No	No
3	Yes	No	Yes	No
4	Yes	No	No	Yes
5	Yes	Yes	Yes	No
6	Yes	Yes	Yes	Yes

One key idea to remember is the following: type matching (to determine which method gets called) for PARAMETERS is NOT dynamic. The compiler identifies the type of the parameter and that is what is used to determine what method gets called.

In our situations above, it's clear what happens in situation #1.

But after that it gets a bit more confusing, so let's try to lay down some rules to help use deal with the cases above, without having to remember 6 separate sets of rules.

Applying our Rules

Let's apply our rules to situation #6 for the equals method in the Coordinate and ColorCoordinate classes. Here are the four objects initially created:

```
Coordinate test = new ColorCoordinate(3,'a',"Green");
ColorCoordinate red = new ColorCoordinate(3,'a',"Red");
ColorCoordinate blue = new ColorCoordinate(3,'a',"Blue");
Coordinate nocolor = new Coordinate(3,'a');
```

Now, let's consider determining WHICH of the four methods gets called for each of the following calls:

nocolor.equals(red): this code compiles because the Coordinate class has a method that takes in a ColorCoordinate, so we bind to Method #2. At run time, Java sees that nocolor is referencing just a regular Coordinate object, so the first method we check is Method #2 which gets executed.

red.equals(nocolor): We first look in the ColorCoordinate class for a method that takes in a Coordinate, based on the references, binding to Method #3. At run-time, this is also the first method we'll try to run, which gets executed.

red.equals(blue): At compile time we bind to method #4 since both references are of type ColorCoordinate. This is also the one that gets executed.

test.equals(blue): At compile time, we look in the Coodinate class for a method that takes in a ColorCoordinate, binding the compilation to Method #2. At run-time however, since test is a ColorCoordinate method, we search for a matching method in the ColorCoordinate class first, finding Method #4, which is the one that gets executed.

blue.equals(test): In this example, the compiler binds us to Method #3 in the ColorCoordinate class which takes in a Coordinate. Since there is no late-binding on parameters, it's Method #3 that executes.

Now, let's move onto our main tracing example for this lecture on the next page.

Example Program

```
// Arup Guha
// 4/9/07
// An example illustrating polymorphism and late binding.

class A {

    protected int a;

    public A() {
        System.out.println("in A");
        a = 0;
    }

    public void f(A x) {
        System.out.println("f(a)");
        x.g();
    }

    public void f() {
        System.out.println("In default f(a)");
    }

    public void g() {
        System.out.println("g(a)");
    }
}

class B extends A {

    protected int b;

    public B() {
        System.out.println("in B");
        b = 1;
    }

    public void f(B x) {
        System.out.println("f(b)");
        x.g();
    }

    public void f() {
        System.out.println("In default f(b)");
    }

    public void g() {
        System.out.println("g(b)");
    }
}
```

```
public class C extends B {

    private int c;

    public C() {
        System.out.println("in C");
        c = 2;
    }

    public void f(C x) {
        System.out.println("f(c)");
        x.g();
    }

    public void g() {
        System.out.println("g(c)");
    }

    // Try to guess what happens before you run it.
    public static void main(String[] args) {
        A mine = new B();
        B other = new B();
        A atoc = new C();
        A plain = new A();

        System.out.println();
        other.f(mine);

        System.out.println();
        mine.f(other);

        System.out.println();
        mine.f();

        System.out.println();
        ((B)mine).f(other);

        System.out.println();
        atoc.f(plain);

        System.out.println();
        atoc.f(atoc);
    }
}
```

Let's go through these method calls one by one:

```
other.f(mine);
```

Compilation Binding: other's reference is class B, mine's reference is class A In class B, there is no method f that takes in an A, so we look in class A. In class A there is a method f that takes in an A, so we bind to this method (Class A, f(A x)).

Run-time Execution: At run time we see that our object is of Class B. We look for a matching method in class B and find none. We move to class A and run the method f(A x) on object other. Here is the output of that method call and the next one that ensues:

```
f(a)
g(b)
```

The key here is to notice that although the reference time of x is A, x is referring to an object of type B (which is what mine was referring to). Thus, when we run the line of code:

```
x.g();
```

in the method f(A x) in class A, Java recognizes that x is referencing an object of class B and looks to see if a corresponding method g exists there. It does, so this late binding occurs and that method is executed.

Next method call is:

```
mine.f(other);
```

Compilation Binding: mine's reference is class A, other's reference is class B. In class A, there is a method f that takes in a B. This is the method f(A x) because recall that a B IS-A A. Thus, we we bind to this method (Class A, f(A x)).

Run-time Execution: At run time we see that our object, that mine is referring to, is of Class B. **We look for a matching method in class B and find none.** This is because at compilation time we bound to a method with the signature f(A x) and no method in class B matches this. We move to class A and run the method f(A x) on object mine. Here is the output of that method call and the next one that ensues:

```
f(a)
g(b)
```

Since other is referencing an object of type B, Java binds the method call late and knows to call the method g in the class B.

Now let's look at the third call:

```
mine.f();
```

Compilation Binding: mine is a reference of class A, so we look for a method called f() in class A and find it. Thus, for compilation, we bind to Class A f().

Run-time Execution: At run-time we recognize that mine is referencing an object of Class B. and look for a method called f() in class B. There is such a method and this is what gets printed:

```
In default f(b)
```

Next method call is

```
((B)mine).f(other);
```

This is EXACTLY like the second method call but with the explicit cast of mine to class B. This time, see what occurs differently at Compilation time:

Compilation binding: since mine has been cast to class B, we treat mine as a class B reference for this computation only. The method name is f and it takes in a reference of class B. Thus, for compilation, we first look for a method f(B x) in the class B. We find one, binding compilation to the method class B f(B x).

Run-time binding: We see that mine is referencing an object of class B (not C), so we start looking for suitable methods in class B. We find the method f(B x). Thus, we'll print out:

```
f(b)
g(b)
```

The fifth method call is

```
atoc.f(plain);
```

Compilation binding: atoc is an A refernence and plain is also an A reference. We bind to the method f(A x) in the class A.

Run-time binding: We see that atoc is referencing an object of class C, so we look for a method f(A x) in the class C. No such method exists, so we now look to class B. Again, no such method exists, so we fall back down again to class A. Here we find a suitable method that we bound compilation to and run it. Since plain is referencing an object of class A, this is what gets printed:

```
f(a)
g(a)
```

The last method call is

```
atoc.f(atoc);
```

Compilation binding: Both references are of class A, so we bind to f(A x) in the class A.

Run-time binding: We see that atoc is referencing an object of class C, so we look for a method f(A x) in the class C. No such method exists, so we now look to class B. Again, no such method exists, so we fall back down again to class A. Here we find a suitable method that we bound compilation to and run it. Since atoc is referencing an object of class C, this is what gets printed:

```
f(a)
g(c)
```