COP 3330 – Object Oriented Programming in Java
Polymorphism Lecture 2

As previously stated in the previous lecture, basic idea behind polymorphism is that the decision of which method is called can sometimes be delayed till run-time.

To make a real-life analogy, consider the following:

Imagine that you invited a guest speaker from NASA. Imagine that everyone who works at NASA is of type NASAEmployee. In code, imagine that you instantiated a variable of type NASAEmployee named UnknownSpeaker. Then, you would have her speak:

```
UnknownSpeaker.speak();
```

Now, you could relay this command before knowing WHO was speaking. But, at run-time, depending on who the speaker is, they may talk about different things. If you get an astronaut like Sally Ride, she might talk about what it's like to experience zero gravity for several days straight. If on the other hand you got a project manager, she might talk about how to create a time line.

Since you don't know beforehand that you are going to get a speaker, it would be illegal to do the following:

```
UnknownSpeaker.zerogravitydemo();
```

since only an Astronaut can do the zero gravity demo and when you make the initial invitation, you DON'T know that you're going to get an Astronaut.

But since all NASA Employees can speak, it's perfectly valid when you extend the invitation to ask them to do so and then when the actual speech happens, it is tailored by the specific NASA Employee that comes to speak.

Motivation behind polymorphism

An example of a situation where polymorphism would be desirable is processing payroll. Each employee must get paid, but different employees get paid differently. An inheritance structure more easily encapsulates the differences between how everyone gets paid in a seamless manner than other systems.

For example, you could take care of paying everyone as follows:

```
Employee[] workers = new Employee[100];

// Stuff happens here…

//Here we pay everyone:
for (Employee worker: workers)
    worker.pay();
```

The neat thing here is that even though each array element is a reference to an Employee, the actual object to which is being referred could easily be a specialized Employee, such as a SalariedEmployee. Even though we might not know that workers[3] is a SalariedEmployee before run-time, when the program runs, it can pay the SalariedEmployee in the appropriate manner because the pay() would be overridden in the SalariedEmployee class.

In general, it allows you to have a collection of general Objects and perform some action on all of them, but that action can be modified if some of the general Objects are of a more specific type.

Polymorphism Example #2

In the last lecture we traced through a detailed example with classes A, B and C. We'll do this again, bringing up some more use cases that apply the general rules laid out in the last lecture. To make this example slightly more involved, the objects involved will have instance variables. A will have instance variable x, B will add instance variable y, and C will add instance variable z.

First, let's look at the code on the next couple of pages:

```
class A {

	protected int x;

	public A() {
		x = 2;
	}

	public A(int a) {
		x = a;
	}

	// This updates our x to the sum of the two x's.
	public void f(A a) {
		System.out.println("1");
		x = a.x + x;
	}

	// Updates our x to add in b's x and y.
	public void f(B b) {
		System.out.println("2");
		x = x + b.x + b.y;
	}

	public String toString() {
		return "("+x+")";
	}
}

/*** First level of inheritance, has a new instance variable y.
     Also has two methods f with the same signature as class A
***/
class B extends A {

	protected int y;

	public B(int a) {
		y = a;
	}

	// It's different than A's f...this updates B's y not x.
	public void f(A a) {
		System.out.println("3");
		y = a.x + x + y;
	}

	// This updates both x and y of a based on b.
	public void f(B b) {
		System.out.println("4");
		x = b.x + x;
		y = b.y + y;
	}

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

```
/*** Last level of inheritance. C inherits from B.
     z is the new instance variable here.
       We don't have an f that takes in B though.
***/

public class C extends B {

      private int z;

      public C(int a, int b, int c) {
            super(a);
            z = c;
      }

      public void f(A a) {
            System.out.println("5");
            z = z + a.x;
      }

      public void f(C c) {
            System.out.println("6");
            x = x + c.x;
            y = y + c.y;
            z = z + c.z;
      }

      public String toString() {
            return "("+x+","+y+","+z+")";
      }

      public static void main(String[] args) {

            A one = new A();
            A two = new B(4);
            B three = new B(7);
            B four = new C(8,1,3);
            C five = new C(2,6,9);
            C six = new C(5,4,6);

            one.f(two);
            System.out.println("one.f(two)");
            System.out.println(one);
            System.out.println(two);
            System.out.println("---------------------------");

            one.f(four);
            System.out.println("one.f(four)");
            System.out.println(one);
            System.out.println(four);
            System.out.println("---------------------------");

            two.f(three);
            System.out.println("two.f(three)");
            System.out.println(two);
            System.out.println(three);
            System.out.println("---------------------------");
```

```
            four.f(five);
            System.out.println("four.f(five)");
            System.out.println(four);
            System.out.println(five);
            System.out.println("----------------------------");


            six.f(five);
            System.out.println("six.f(five)");
            System.out.println(six);
            System.out.println(five);
            System.out.println("----------------------------");


            two.f(one);
            System.out.println("two.f(one)");
            System.out.println(two);
            System.out.println(one);
            System.out.println("----------------------------");


            four.f(two);
            System.out.println("four.f(two)");
            System.out.println(four);
            System.out.println(two);
      }
}
```

Let's do the same thing we did last lecture, go through each of these method calls and the corresponding prints, one by one.

`one.f(two)`
Compilation binding: both references in class A, looking for f(A) in class A.

Run-time binding: one is referring to a class A object, so we look in class A for a method f(A). We find the method that was bound at compilation, so method #1 runs. Here is what is printed:

```
1
one.f(two)
(4)
(2,4)
```

As described, method that gets called is f(A) in class A. This method prints out 1, and then it will add the two x coordinates together, getting 4 and storing this in the x component of the object pointed to by one. When we print one, it shows this change, printing out 4. The object two is unchanged, printing out (2, 4), which is what it was set to when constructed.

`one.f(four)`

Compilation binding: we look in class A for a method that takes in a reference to class B. There is exactly a method that does this, f(B b), so this is the method we bind to for compilation.

Run-time binding: one is referring to a class A object, so we look in class A for a method f(B). We find the method that was bound at compilation, so method #1 runs. Here is what is printed:

```
2
one.f(four)
(14)
(2,8,3)
```

As described, method that gets called is f(B) in class A. This method prints out 2, and then it will add the x and y components of the object referred to by y (which is referred to by four in main), and add this to the x component of one getting 14 and storing this in the x component of the object pointed to by one. When we print one, it shows this change, printing out 14. The object four is unchanged, printing out (2, 8, 3), which is what it was set to when constructed.

Notice that the constructor C never uses the value of b it's given, so when we do

```
B four = new C(8, 1, 3);
```

The x coordinate is set to 2 by the default A constructor, y is set to 8 and z is set to 3.

`two.f(three)`

Compilation binding: two is a class A reference three is a class B reference, so we're looking for a method with the signature f(B b) in the class A. This method exists so we bind to this method (same one as in the previous case.)

Run-time binding: two is referring to a class B object, so we look in class B for a method f(B). We find a suitable method, different than the one we bound to in compilation, this is the method with the number 4 in it, so this is what gets executed. Here is the corresponding output:

```
4
two.f(three)
(4,11)
(2,7)
```

As described, method that gets called is f(B) in class B. This method prints out 4, and then it will add the two x-coordinates together (2 + 2 = 4) and the two y coordinates together (4 + 7 = 11), and stores these as the new x and y for the object referred to by two. Thus, when we print, two prints out as (4, 11) and three remains unchanged, printing out as (2, 7).

`four.f(five)`

Compilation binding: four is a class B reference five is a class C reference, so we're looking for a method with the signature f(C c) in the class B. The closest method that exists in the class is f(B b), since a C IS-A B. We bind to this method at compilation time: f(B b) in class B.

Run-time binding: four is actually referencing an object of class C. Thus, we'll look for a method with the signature f(B b) in the class C. No such method exists (method 5 isn't a match because even though a B is-a A, we had bound at compilation time to matching a B exactly), so we fall down to class B, where we find the method we bound compilation to, so we run this method. Here is what gets printed:

```
4
four.f(five)
(4,10,3)
(2,2,9)
```

For tracing purposes note the object that four references before the call stores (2,8,3) and object that five references before the call stores (2, 2, 9). We call method 4 and print this out. Then, in the method we add 2 + 2 = 4 and 8 + 2 = 10, and store 4 and 10 respectively as x and y in the object referenced by four. The object referenced by five stays unchanged.

`six.f(five)`

Compilation binding: six is a class C reference five is a class C reference, so we're looking for a method with the signature f(C c) in the class C. This method exists, so we bind to method 6.

Run-time binding: The actual object six refers to is of type C, so our run-time binding will be the same as our compilation binding and method 6 runs and here's what gets printed:

```
6
six.f(five)
(4,7,15)
(2,2,9)
```

For tracing purposes note the object that six references before the call stores (2,5,6) and object that five references before the call stores (2, 2, 9). We call method 6 and print this out. Then, in the method we add 2 + 2 = 4, 5 + 2 = 7 and 6+9 = 15, and store 4, 7, and 15 respectively as x, y and z in the object referenced by six. The object referenced by five stays unchanged.

`two.f(one)`

Compilation binding: both references in class A, looking for f(A) in class A.

Run-time binding: two is referring to a class B object, so we look in class B for a method f(A). We find the method labeled #3 and run this one. Here is the output produced:

```
3
two.f(one)
(4,29)
(14)
```

First let's recall that the object one is referring two stores x = 14 and the object that two is referring to stores (4, 11). After calling method #3, we add 14 + 4 + 11 = 29 and store this value in the y coordinate of two to update the object to (4, 29). That's why (4, 29) prints for two and (14) prints for one.

`four.f(two)`

Compilation binding: four is a class B reference and two is a class A reference. Thus, for compilation, we first look in class B for a method f(A). One exists exactly like this, so this is the method that allows the code to compile.

Run-time binding: The object to which four is referring is a class C object, so we start our search for the method f(A) in class C. class C has exactly this method, so it's the one that runs. It will print out 5 and then continue execution.

```
5
four.f(two)
(4,10,7)
(4,29)
```

First let's recall that the last time we modified the object pointed to by four, it stored (4,10,3) at the completion of the previous method call. This is the state of that object right before this call. The code in f(A) in class C increments the z instance variable by a.x, in this case, 4 (value of x in object referenced by two). Thus, adding 3 + 4 = 7, so the z variable in the object referenced by four will change to 7. Thus, this prints out as (4,10,7). The object referenced by two stays the same and prints out as (4,29), its state after the previous method call completed.