10/7/2026 COP 3330 (Recorded Early)

1. Review the Polymorphism Rules
2. Mixed Fraction Example – More detail (toy example stock prices)
3. Pac Mac Example
4. Tracing Example (C.java) #1 (doesn't do anything useful, but illustrates the nuances of polymorphism

Leave as an exercise for the reader: Tracing Exercise #2

Review of Rules

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** 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.

if B inherits A, C inherits from B, D inherits from C and E inherits from D

```
C temp = new E(...);
//Reference type is C, the object is E.
temp.method(a,b)

In order for this to compile, either the class C, or
the class B or the class A must have a method that
matches this signature!!!
```

2. **Once we find a method call that matches in the class of the reference of o, then, AT RUN TIME, 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.

In the example above, as long as the code compiles, when we run it, we'll first look to see if there's a matching method in E, if not, then D, and then eventually we are guaranteed to find a match.

This is late binding...

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 **"latebinding".**

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.

Picture of

tmp.add(1)

tmp --> num 3, den 5

num 8, den 5 returned from method, but no one points to this, so it gets garbage collected and tmp still equals its old value

in memory for add, we created a variable num
we set num = 8

Return a new fraction(8, 3)
BUT IN main I do nothing this

vs.

tmp = tmp.add(1)

num 3, den 5 NOW this gets garbage collected.

tmp --> num 8, den 5

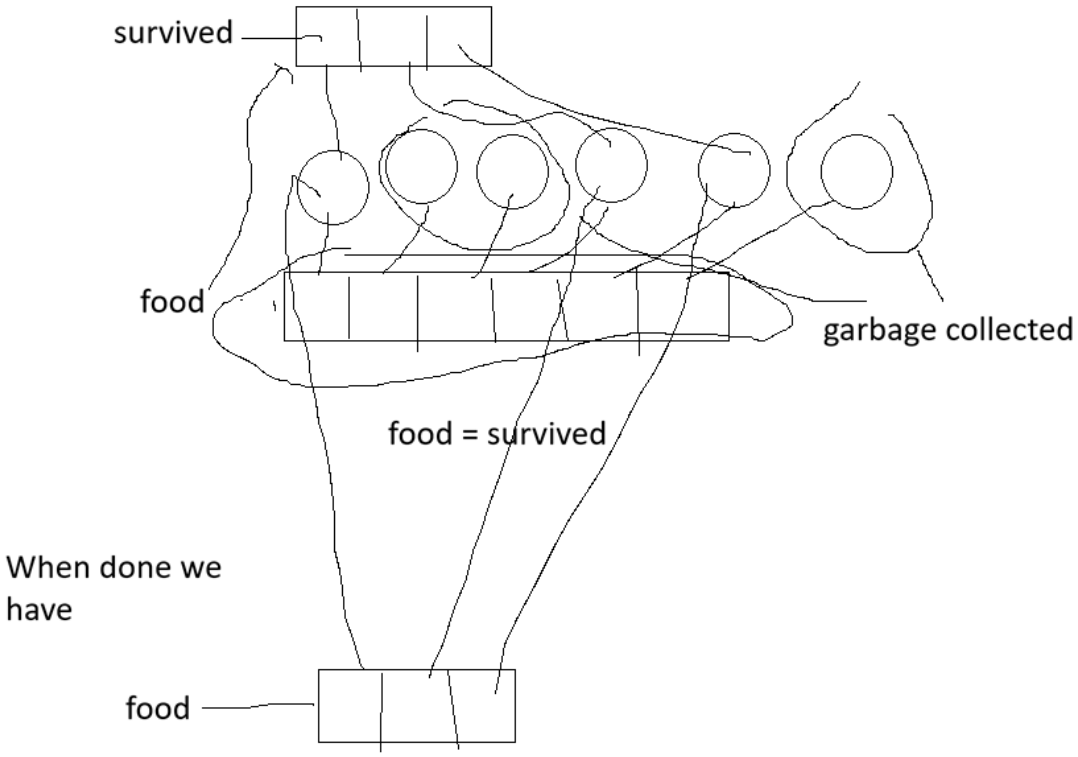
surv	T	F	F	T	T	F
food	A	C	E	G	H	K

i = 0 1 2 3 4 5
j = 0 1 2 3

survived	0	1	2
	A	G	H

food --->

Here is a much better picture of what just happened there:


survived
food
garbage collected
food = survived
When done we have
food

Tracing Example #1 C.java

```
A mine = new B();
```

First, B's constructor gets called, but because no super is explicitly written, super() gets called and A's constructor prints out:

```
in A
```

It also sets a to 0. Then it continues with B's constructor code and prints out:

```
in B
```

and sets b = 1

Consider this line

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

mine's reference type is A
other's object type is B
For compilation we use other's reference type also B. When we look in class B, there is no method f that takes in an A...
So we go down to class A to find:

f(A x)

We run this, but because the object x is referencing is an actual B object when we call

x.g()

Late binding kicks in recognizes that x is an referencing an object of type B and calls g from the class B.

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

mine is reference type A object type B, for compilation we make sure that the A class has a method f that takes in a B, it does since it has an f that takes in an A and a B is an A.

At run time, Java uses the fact that mine's object is B and calls f(B) in the class B

Printed

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