CS 213 – Software Methodology

Spring 2023

Sesh Venugopal

Feb 22

Inheritance – equals method

Method Overloading/Overriding

Method **OVERLOADING**:

Two or more methods in a class have the same name but different number, types, or sequences of parameters

```
class Test {
   int m(int x) {…}
   int m(float y) {…}
}
```

Overloaded method `m`

```
class Test {
   int m(int x) {…}
   float m(float y) {…}
}
```

Overloaded method `m`

```
class Test {
   int m(int x) {…}
   float m(int y) {…}
}
```

Error

Two or more methods in a class are **overloaded** if they have the same name but different *signatures*

signature = name + params (return type NOT included in signature)

Method **OVERRIDING**:

A method in a subclass has the same signature as in the superclass

Implementing `equals` — Rookie Version

Implementing `equals` — Rookie Version

Rookie attempt to implement `equals` (e.g. in `Point`):

```
public boolean equals(Point p) {
    return x == p.x && y == p.y;
}
```

```
Point cp =
   new ColoredPoint(3,4,"black");
```

```
cp.equals(cp); // ?  True
```

Ok, inherited `equals(Point p)` in `ColoredPoint` is called

Dynamic type `ColoredPoint` argument at run time matches static type `Point` parameter (every `ColoredPoint` is a `Point`)

Implementing `equals` — Rookie Version

Rookie attempt to implement `equals` (e.g. in `Point`):

```
public boolean equals(Point p) {
    return x == p.x && y == p.y;
}
```

```
String s = "(3,4)";
cp.equals(s); // ? FALSE!!
```

WHY IS IT NOT A COMPILER ERROR??

Since the argument String type does not match expected Point type parameter, the inherited `Object equals(Object o)` is called instead!!!

`equals(Point p)` does NOT override `Object` class's `equals(Object o)`

The `Object` class's `equals(Object o)` compares memory addresses, and returns true if addresses of the compared objects are the same, i.e. they are the exact same object. Otherwise it will return false, no matter the type of the argument sent in.

Implementing `equals` — Rookie Version

Rookie attempt to implement `equals` (e.g. in `Point`):

```
public boolean equals(Point p) {
    return x == p.x && y == p.y;
}
```

```
Object op = new Point(3,4);
Point p = new Point(3,4);
p.equals(op); // ? FALSE!!
```

The inherited `Object equals(Object o)` is called!!!

Reason: the STATIC type of parameter is `Object`, which matches the `Object` parameter type of inherited `equals`

Moral of the story: You `MUST override Object equals(Object o)`

Implementing `equals` — Pro Version

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)

Single Version: Overriding `equals`

```
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
   }
}
```

Calling the `Point equals` method

```
Point p = new Point(3,4);
Point cp =
   new ColoredPoint(3,4,"black");
String s = "(3,4)";
Point p2 = new Point(4,5);
```

```
p.equals(p); // ? True
p.equals(cp); // ? True
p.equals(s); // ? False
p.equals(p2); // ? False
```

`equals` overload + override (both versions present)

```
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
   }

   public boolean equals(Point p) {
     if (p == null) {
        return false;
     }
     return x == p.x && y == p.y
   }
}
```

Given the following setup:

```
Point p = new Point(3,4);

Object o = new Object();

Object op = new Point(3,4);
```

Which method is called in each case, and what's the result of the call?:

```
p.equals(p); // ? True

p.equals(o); // ? False

p.equals(op); // ? True
```

equals overload + override

```
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
   }

   public boolean equals(Point p) {
     if (p == null) {
        return false;
     }
     return x == p.x && y == p.y
   }
}
```

With the same setup as before:

```
Point p = new Point(3,4);

Object o = new Object();

Object op = new Point(3,4);
```

Which method is called in each case, and what's the result of the call?:

```
op.equals(o); // ? False
```
[Same as `p.equals(o)`]

```
op.equals(op); // ? True
```
[Same as `p.equals(op)`]

```
op.equals(p); // ?  True
```

[However, `p.equals(p)`]

Here are the rules for how it all works …

(Applies broadly, not just to `equals`)

Given: Point overrides equals(Object)
and overloads it with equals(Point)

Method Overloading/Overriding
Static and Dynamic Types

What rules determine which method is called?

A. First, the COMPILER determines the *signature* of the method that will be called:

1. Look at the STATIC type of the object ("target") on which method is called.
   Say this type/class is X

```
Object o = new Object();

Point p = new Point(3,4);

Object op = new Point(3,4);
```

```
p.equals(o);
p.equals(p);
p.equals(op);
```

Static type of `p` is `Point`

```
op.equals(o);
op.equals(p);
op.equals(op);
```

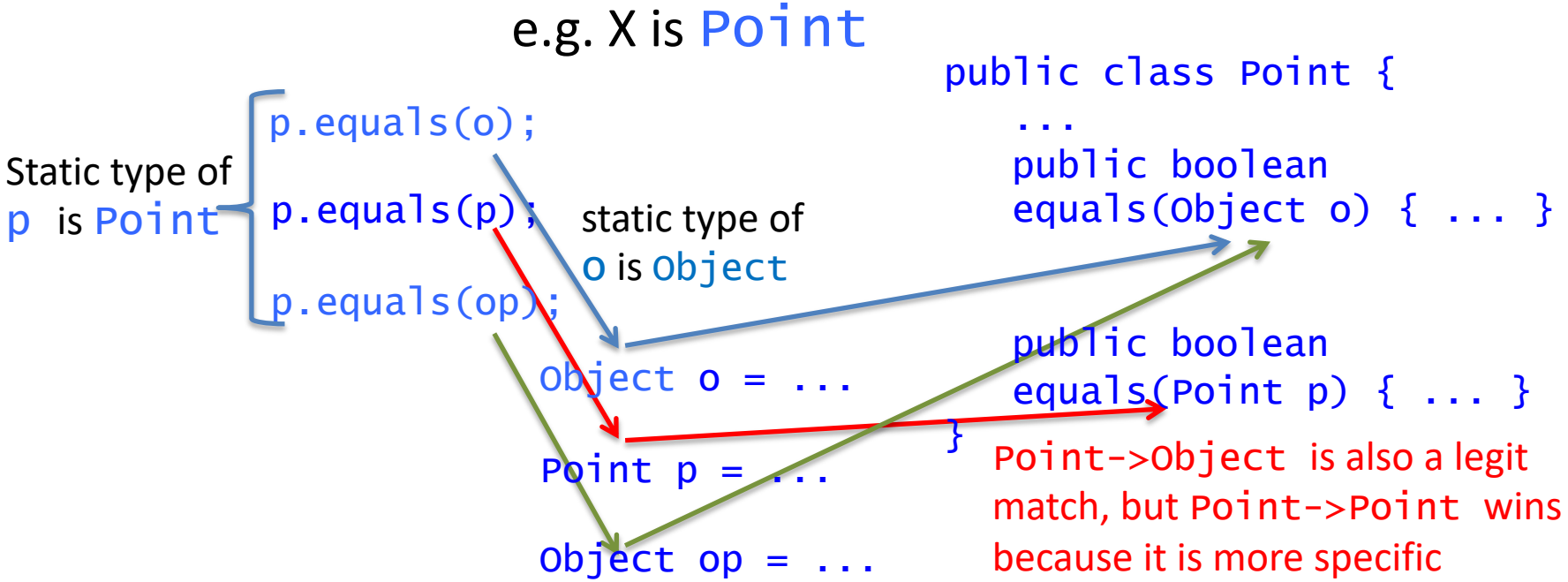
Static type of `op` is `Object`

Method Overloading/Overriding
Static and Dynamic Types

What rules determine which method is called?

A. First, the COMPILER determines the *signature* of the method that will be called:

2. In the class X, find a method whose name matches the called method, and whose parameters most specifically match the STATIC types of the arguments at call



Method Overloading/Overriding
Static and Dynamic Types

What rules determine which method is called?

A. First, the COMPILER determines the *signature* of the method that will be called:

2. In the class X, find a method whose name matches the called method, and whose parameters most specifically match the static types of the arguments at call

e.g. X is `Object`

```
op.equals(o);
op.equals(p);
op.equals(op);
```

Static type of `op` is `Object`

`Object` has a single `equals` method that matches all of these calls

Method Overloading/Overriding
Static and Dynamic Types

What rules determine which method is called?

B. At run time, the actual "target" object (dynamic type), or its superclass chain is searched for the pre-determined signature, and the matching method executed

Static type of `p` is `Point`

```
p.equals(o);
p.equals(p);
p.equals(op);
```

`Point` defines `equals(Object)` as well as `equals(Point)`, which match with the respective statically bound method signatures

```
Point p = new Point(3,4);
```

Dynamic type of `p` is `Point`

Method Overloading/Overriding
Static and Dynamic Types

What rules determine which method is called?

B. At run time, the actual "target" object (dynamic type), or its superclass chain is searched for the pre-determined signature, and the matching method executed

What if `Point` did NOT override `equals(Object)` inherited from `Object`?

Static type of `p` is `Point`

```
p.equals(o);
p.equals(p);
p.equals(op);
```

Bad news: original `Object` `equals` is called, result is false, even though both points are are (3,4)!!

```
Point p = new Point(3,4);
```

Dynamic type of `p` is `Point`

Method Overloading/Overriding
Static and Dynamic Types

What rules determine which method is called?

B. At run time, the actual "target" object (dynamic type), or its superclass chain is searched for the pre-determined signature, and the matching method executed

Static type of `op` is `Object`

```
op.equals(o);
op.equals(p);
op.equals(op);
```

`Point` defines `equals(Object)` which matches with the statically bound methods, so gives correct results

```
Object op = new Point(3,4);
```

Dynamic type of `op` is `Point`

Method Overloading/Overriding
Static and Dynamic Types

What rules determine which method is called?

B. At run time, the actual "target" object (dynamic type), or its superclass chain is searched for the pre-determined signature, and the matching method executed

What if `Point` did NOT override `equals(Object)` inherited from `Object`?

```
Object op = new Point(3,4);
```

Dynamic type of `op` is `Point`

Static type of `op` is `Object`

```
op.equals(o); // ?    False
op.equals(p); // ?    False
op.equals(op); // ?   True
```

Bad news: result is false, even though both objects are (3,4)!!

All these calls would be to the `Object` version of `equals`

Method Overloading/Overriding
Static and Dynamic Types

What if the inherited `equals(Object)` is not overridden, and only `equals(Point)` is coded?

The call `op.equals(p)` will result in false,
which fails the requirement of (3,4) being equal to (3,4),
even if the point objects are physically different

So, the inherited `equals(Object)` must be overridden

Is it sufficient to only override the inherited `equals(Object)`, and not code an `equals(Point)` method?

Yes

Is it detrimental/inadvisable to have both?

Yes, it leads to avoidable confusion, so removing `equals(Point)` is clearer/unambiguous/better design