# CS 213 : Software Methodology

*Sesh Venugopal*

Inheritance - Object equals method

# Method Overloading/Overriding

## Method **OVERLOADING**:

Two methods in a class have the same name but different numbers, 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

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,"blue");
Point p = new Point(3,4);
cp.equals(p); // ?   True
```

Inherited `equals(Point p)` in `ColoredPoint` is called

---

```
String s = "(3,4)";
```

```
p.equals(s); // ?   FALSE!!
```

The inherited `Object equals(Object o)` is called!!!
Otherwise, this should give a compiler error

**`equals(Point p)` does NOT override `Object equals(Object o)`**

# Implementing `equals` — Rookie Version

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

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

`p.equals(op); // ?` FALSE!!

The inherited `Object equals(Object o)` is called!!!
Because 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` — Grad Version

# Overriding `equals`

Boiler-plate way to override equals (e.g. `Point`):

```
public class Point {
   int x,y;
   ...
   public boolean equals(Object o) { // override!!
      if (o == null || !(o instanceof Point)) {
         return false;
      }
      Point other = (Point)o;

      return x == other.x && y == other.y;
   }
   ...
}
```

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

```
Point cp =
   new ColoredPoint(3,4,"black");
```
`p.equals(cp); // ?` True

```
String s = "(3,4)";
```
`p.equals(s); // ?` False

```
Point p2 = new Point(4,5);
```
`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
   }
}
```

With 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
   }
}
```

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
```

[ But `p.equals(p)` ]

Here are the rules for how it all works …

(For ANY class, or super/sub classes, and ANY method)

# 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);
```

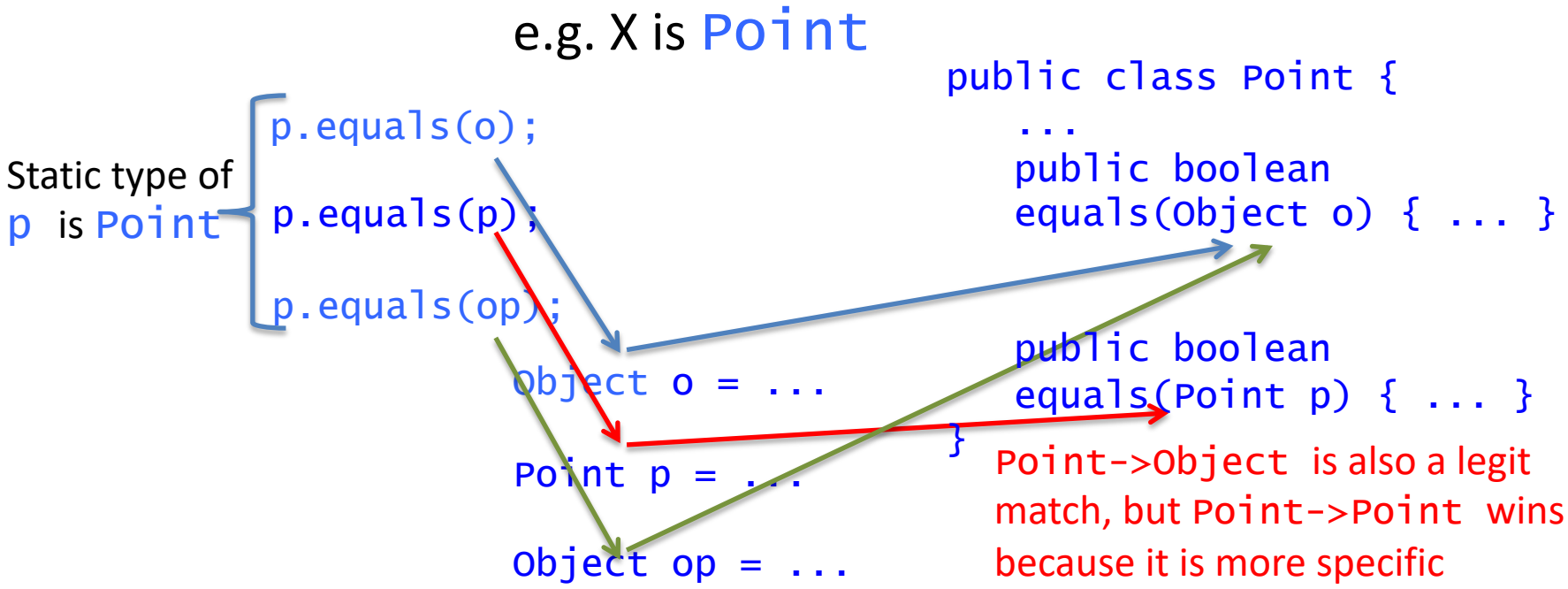
Static type of `p` is `Point`

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

Static type of `op` is `Object`

# Method Overloading/Overriding
# Static and Dynamic Types

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

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 runtime/actual "target" (called) object, 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 p = new Point(3,4);
```

Dynamic type of `p` is `Point`,

IF `Point` overrides `equals(Object)` and also implements `equals(Point)`, compiler would have determined signatures based on closest parameter type matches

# Method Overloading/Overriding
# Static and Dynamic Types

What if `Point` did NOT override `equals(Object)`

Static type of `p` is `Point`

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

`p.equals(o);` → Would call inherited version of `equals`

`p.equals(op);` → Would call inherited version of `equals`
Bad news: result is false, even though but objects are (3,4)!!

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

Dynamic type of `p` is `Point`

# Method Overloading/Overriding
# Static and Dynamic Types

Static type of `op` is `Object`

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

IF `Point` overrides `equals(Object)` then that version will be called, otherwise the inherited version will be called

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

Dynamic type of `op` is `Point`

# Conclusion

**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 NOT implementing `equals(Point)` is unambiguous and therefore a better design