CS 213 : Software Methodology

Spring 2016

Lecture 3: Jan 26
Overriding `equals`

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)

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");
Point p2 = new Point(4,5);
String s = "(3,4)";
```

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

Background: 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

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 following setup:

```
Object o = new Object();

Point p = new Point(3,4);

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

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

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

p.equals(p); // ? True

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 following setup:

```
Object o = new Object();

Point p = new Point(3,4);

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

op.equals(p); // ? True

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

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:

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

  e.g. in `op.equals(p)` the static type of `op` is `Object`

-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. In class `Object,` method `equals(Object)` is the only match

- So, the signature `equals(Object)` will be used

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 determined signature, and the matching method executed

e.g. in `op.equals(p)` the runtime (actual) type of `op` is `Point`

In class `Point`, method `equals(Object)` is looked for, is found, and is executed

Method Overloading/Overriding
Static and Dynamic Types

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

The previous example of `op.equals(p)` will result in false **(why?),** which will be counter to the intention of having (3,4) be 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