Today …

• Quiz 6
• More on the Object class
• Reminder: Tutorials on Thursday
• Readings:
  – Style: Ch. 5 pp. 57-75, 85-88
The Object class

Inheriting from Object also means
  – Object methods can be called on all objects

So what are these methods?
  – What do they do?
  – When should they be used?
  – What are their default implementations?
  – How should we override them?

We'll focus on the Object methods:
  – toString(), clone(), equals(), and hashCode()

The equals() Object method

public boolean equals(Object obj)

What does it do?
  – Tests whether two objects are considered "equal"

When should it be used?
  – If there can be multiple objects that have the same "value"
    • And there is only one thing represented by the value
  – Often used for "value" classes
    • Integers, strings, dates, phone numbers, etc.
    • Can be more complex too (e.g., Employees, Movies, etc.)
    • For these, two objects with the same value(s) are the same
  – Can depend on how objects are created, managed, and used in
    your program
      As well as if objects are cloned!
The equals() Object method

What is the default implementation (in Object)?

– Checks if object references are equal
  
  ```java
  public boolean equals(Object obj) {
      return this == obj;
  }
  ```

– The default checks if they are the same object!
– Sometimes referred to as “object identity”

How should we override equals()?

– Very carefully!
– Best not to override it (you’ll see why soon)
– When is overriding unnecessary?
  
  • Each instance of the class is inherently unique
  • Clients will not need or want logical equality testing
  • A superclass overrides equals, and behavior is fine for the subclass
  • The class is not visible (e.g., inner class, private class, etc.)
The equals() Object method

How should we override equals()?

- To override, you must follow a "contract" ...
- Your equals() method must satisfy an equivalence relation
  - Reflexive
  - Symmetric
  - Transitive
- Also, for equals()
  - "Consistent"
  - "Non-nullity"

equals() must satisfy an equivalence relation

- **Reflexive:** $\forall x \ (x \ R \ x)$
  - For any non-null reference value $x$
  - $x\.equals(x)$ must return true

- **Symmetric:** $\forall x \forall y \ (x \ R \ y \rightarrow y \ R \ x)$
  - For any non-null reference values $x$ and $y$
  - if $x\.equals(y) == true$ then $y\.equals(x) == true$

- **Transitive:** $\forall x \forall y \forall z \ (x \ R \ y \land y \ R \ z \rightarrow x \ R \ z)$
  - For any non-null reference values $x$, $y$, and $z$
  - if $x\.equals(y) == true$ and $y\.equals(z) == true$ then $x\.equals(z) == true$
Example

Does this satisfy the equals contract?

public class CaseInsensitiveString {
    private String s;
    public CaseInsensitiveString(String s) {
        if(s == null) throw new NullPointerException();
        this.s = s;               \textit{Symmetry problem:}
        String s = ...          \textit{String s = ...}
        CaseInsensitiveString cis = ...    \textit{CaseInsensitiveString cis = ...}
        s.equals(cis)            \textit{s.equals(cis)}
    }
    ...
    public boolean equals(Object obj) {
        if(obj instanceof String)
            return s.equalsIgnoreCase((String)obj);
        return false;
    }
}

Exercise 9
Some details

The problem here is that

– The subclass *adds values (fields)* to the superclass
– Equality is *different for subclasses* than for superclasses

So how should we implement `equals()` in this case?
– Lots of *controversy* on this issue!

---

Some details

The *textbook’s* solution for `Point`

```java
public boolean equals(Object obj) {
    if(obj.getClass() != getClass())
        return false;
    Point p = (Point)obj;
    return p.x == x && p.y == y;
}
```

And for `ColorPoint`

```java
public boolean equals(Object obj) {
    if(!super.equals(obj))
        return false;
    ColorPoint cp = (ColorPoint)obj;
    return color == cp.color;
}
```

*Two objects must be of exactly the same type!*  
*Add tests for extra values*
Some details

• How it works out:

```java
ColorPoint p1 = new ColorPoint(0, 0, Color.RED);
Point p2 = new Point(0, 0);
ColorPoint p3 = new ColorPoint(0, 0, Color.BLUE);
p1.equals(p2);  ← False since diff. types
p2.equals(p3);  ← Again false since diff. types
p1.equals(p3);  ← False - same points, different colors
```

The controversy arises from the “substitution principle”

– a.k.a. “Strong behavioral subtyping”
– Says that a property of a type should hold for its subtypes
– So, any method written for the type, should work just as well for its subtypes

Dr. Barbara Liskov
(Liskov substitution principle with Dr. Jeannette Wing)
The controversy arises from the “substitution principle”

– a.k.a. “Strong behavioral subtyping”
– Says that a property of a type should hold for its subtypes
– So, any method written for the type, should work just as well for its subtypes

• Consider this (contrived example)

```java
public static boolean isUnitPoint(Point p) {
    Point unit = new Point(1, 1);
    return unit.equals(p);
}
```

What if we call `isUnitPoint` with a `ColorPoint` object?

What is the solution?

– You’ll have to wait until next time!!!