Today ...

• Quick note on “assert”
• More on the Object class (equals and hashCode)
The “assert” keyword

• assert is a special Java keyword for checking Boolean conditions
  
  assert expression;

• If expression evaluates to true, then the assert statement “succeeds”
  — that is, nothing happens

• If expression evaluates to false, an AssertionError exception is thrown

The “assert” keyword

• To check assert statements you need to run the JVM with the –ea or –enableassertions flag:
  
  java –ea MyClass

• Here is an example:
  
  public class MyClass {
    public static void main(String[] args) {
      assert 1 == 2;
    }
  }

• And the result: This isn’t a very informative message!
  
  $ java –ea MyClass
  Exception in thread "main" java.lang.AssertionError
  at MyClass.main(MyClass.java:3)
The “assert” keyword

• You can also provide a “value” (detailed) expression

   `assert expression1 : expression2;`

• Here is an example:

   ```java
   public class MyClass {
       public static void main(String[] args) {
           assert 1 == 2 : "1 == 2";
       }
   }
   ```

• And the result:

   ```
   $ java -ea MyClass
   Exception in thread "main" java.lang.AssertionError:
   1 == 2
   at MyClass.main(MyClass.java:3)
   ```

The “assert” keyword

• When *should* assert be used?
  
  – There is no performance penalty (unless enabled)
  
  – Code invariants (results of loops, conditionals, etc.)
  
  – Preconditions and postconditions
  
  – Basic unit testing (like in a main method)

• When *shouldn’t* assert be used?
  
  – Argument checking (throw an exception instead)
  
  – If the expression has side-effects (like adding an element to a list) … assert statements aren’t executed if disabled
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"

![Diagram showing object identity]

equals() must satisfy an equivalence relation

- Reflexive: \( \forall x \ (x \ R \ x) \)
  - every object is equal to itself!
  - For any non-null reference value \( x \)
  - \( x \).equals(\( x \)) must return true

- Symmetric: \( \forall x \forall y \ (x \ R \ y \rightarrow y \ R \ x) \)
  - Objects agree on their equality
  - 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?

```java
public class CaseInsensitiveString {
    private String s;
    public CaseInsensitiveString(String s) {
        if(s == null) throw new NullPointerException();
        this.s = s;
    }
    ...
    public boolean equals(Object obj) {
        if(obj instanceof String)
            return s.equalsIgnoreCase((String)obj);
        return false;
    }
}
```

Symmetry problem:
String s = ...
CaseInsensitiveString cis = ...
s.equals(cis)

Exercise 9

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

- Within ColorPoint (version 2)
  ```java
  public boolean equals(Object obj) {
      if(!(obj instanceof Point))
          return false;
      if(!(obj instanceof ColorPoint))
          return obj.equals(this);
      ColorPoint cp = (ColorPoint)obj;
      return super.equals(obj) && color == cp.color;
  }
  ```

What was the problem with this version?
Transitivity...
ColorPoint p1 = ...
Point p2 = ...
ColorPoint p2 = ...
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;
}
```

Two objects must be of exactly the same type!

And for ColorPoint

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

Add tests for extra values
Some details

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?
}
```
Some details

What is the solution?

- **Do not override equals!** *(not a very satisfactory answer)*
- Use `instanceof` if equality is *fixed* in the superclass
  - i.e., if we do not need to override `equals` in subclasses
- Favor *composition over inheritance* *(best practice)*
  - ColorPoint no longer extends Point ... instead has Point as a variable
  - Define a “public Point `asPoint()`” method on ColorPoint
  - This means we can call “isUnitPoint(cp.asPoint())”
  - `equals()` for ColorPoint checks point equality + color equality

Exercise 10

- Do parts 1 and 2
- Try using the `assert` keyword for your tests
- Remember to use the `java -ea` flag