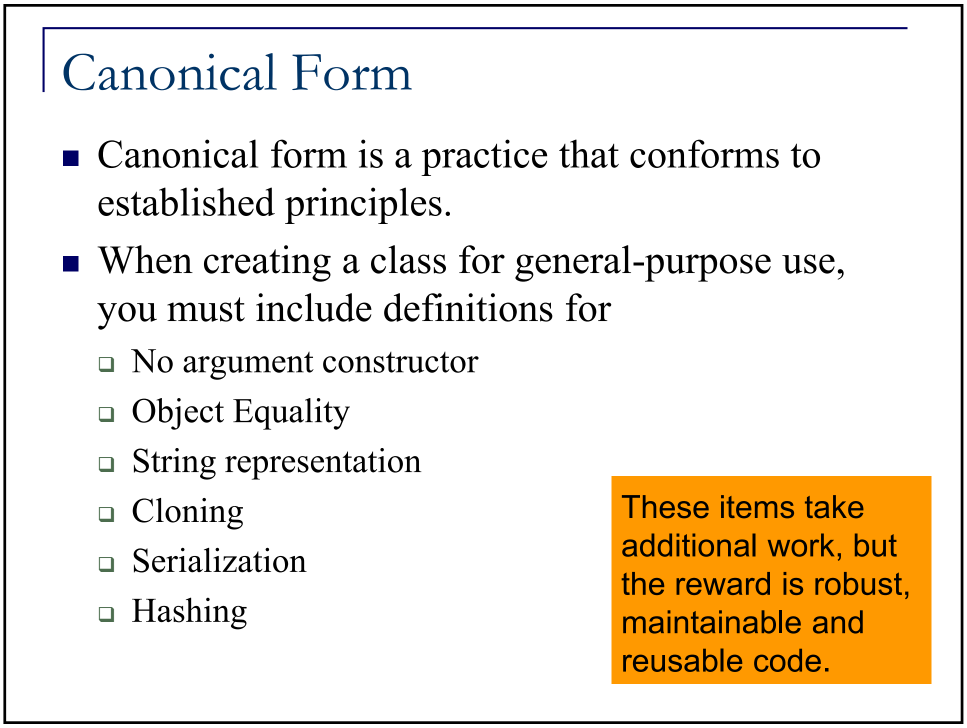
CSC40232: SOFTWARE ENGINEERING

Professor: Jane Cleland-Huang

Canonical Form

sarec.nd.edu/courses/SE2017

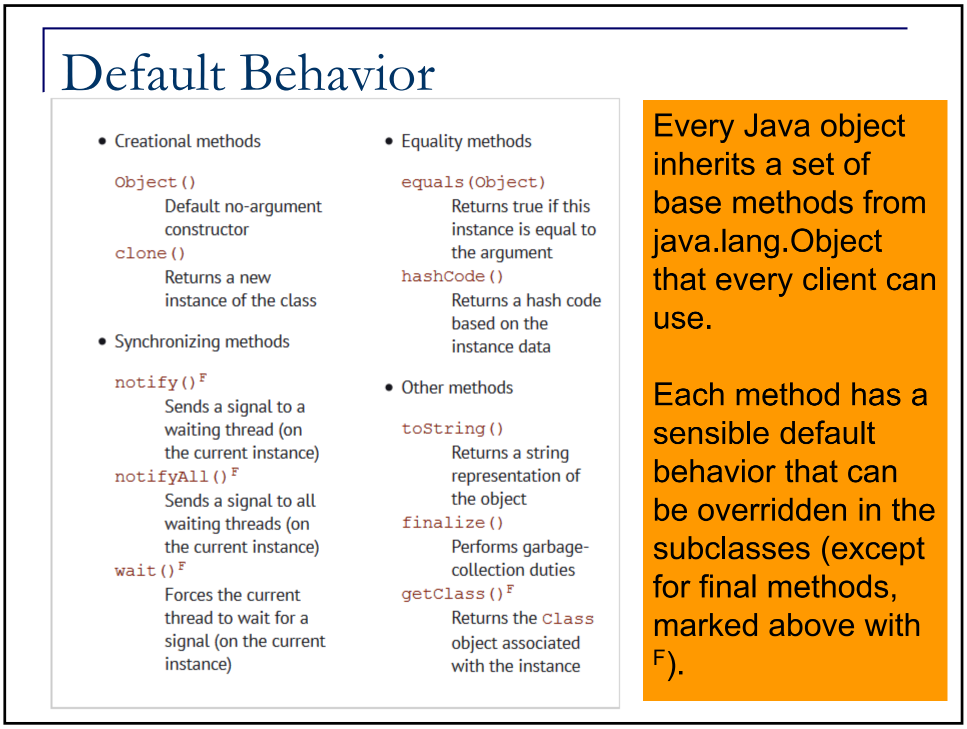
Department of
Computer Science and
Engineering



Canonical Form

- Canonical form is a practice that conforms to established principles.
- When creating a class for general-purpose use, you must include definitions for
  - No argument constructor
  - Object Equality
  - String representation
  - Cloning
  - Serialization
  - Hashing

These items take additional work, but the reward is robust, maintainable and reusable code.



Default Behavior

- Creational methods

  `Object()`
  Default no-argument constructor

  `clone()`
  Returns a new instance of the class

- Synchronizing methods

  `notify()`^{F}
  Sends a signal to a waiting thread (on the current instance)

  `notifyAll()`^{F}
  Sends a signal to all waiting threads (on the current instance)

  `wait()`^{F}
  Forces the current thread to wait for a signal (on the current instance)

- Equality methods

  `equals(Object)`
  Returns true if this instance is equal to the argument

  `hashCode()`
  Returns a hash code based on the instance data

- Other methods

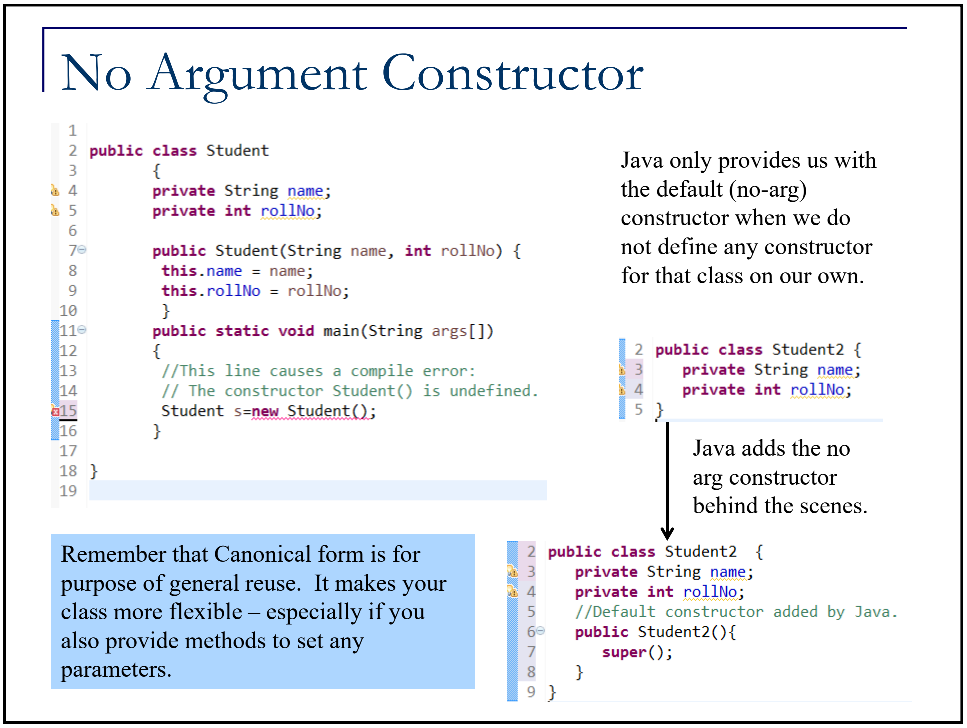
  `toString()`
  Returns a string representation of the object

  `finalize()`
  Performs garbage-collection duties

  `getClass()`^{F}
  Returns the `Class` object associated with the instance

Every Java object inherits a set of base methods from java.lang.Object that every client can use.

Each method has a sensible default behavior that can be overridden in the subclasses (except for final methods, marked above with ^{F}).



No Argument Constructor

```
public class Student
        {
        private String name;
        private int rollNo;

        public Student(String name, int rollNo) {
         this.name = name;
         this.rollNo = rollNo;
         }
        public static void main(String args[])
        {
         //This line causes a compile error:
         // The constructor Student() is undefined.
         Student s=new Student();
        }

}
```

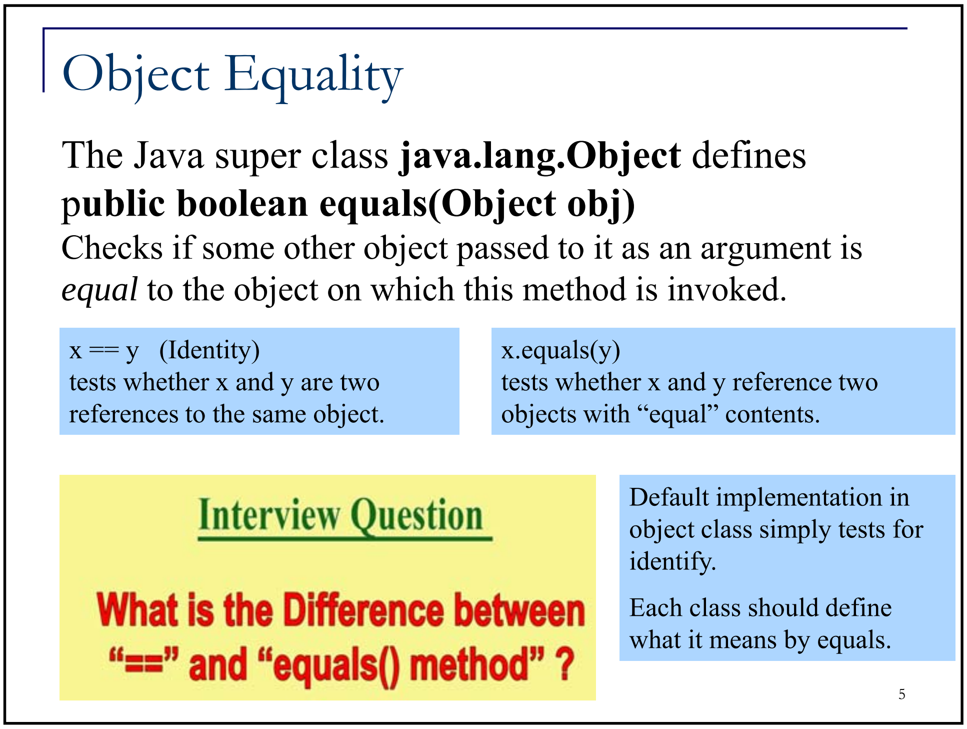
Java only provides us with the default (no-arg) constructor when we do not define any constructor for that class on our own.

```
public class Student2 {
    private String name;
    private int rollNo;
}
```

Java adds the no arg constructor behind the scenes.

```
public class Student2  {
    private String name;
    private int rollNo;
    //Default constructor added by Java.
    public Student2(){
        super();
    }
}
```

Remember that Canonical form is for purpose of general reuse. It makes your class more flexible – especially if you also provide methods to set any parameters.



Object Equality

The Java super class **java.lang.Object** defines **public boolean equals(Object obj)**

Checks if some other object passed to it as an argument is *equal* to the object on which this method is invoked.

x == y (Identity)
tests whether x and y are two references to the same object.

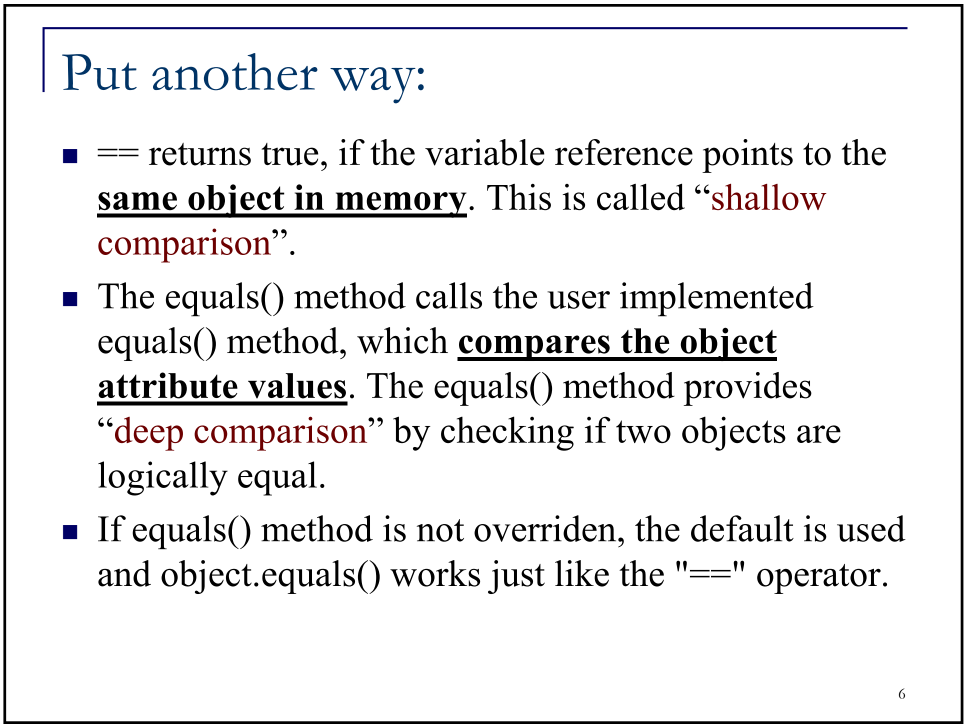
x.equals(y)
tests whether x and y reference two objects with "equal" contents.

Interview Question

What is the Difference between "==" and "equals() method" ?

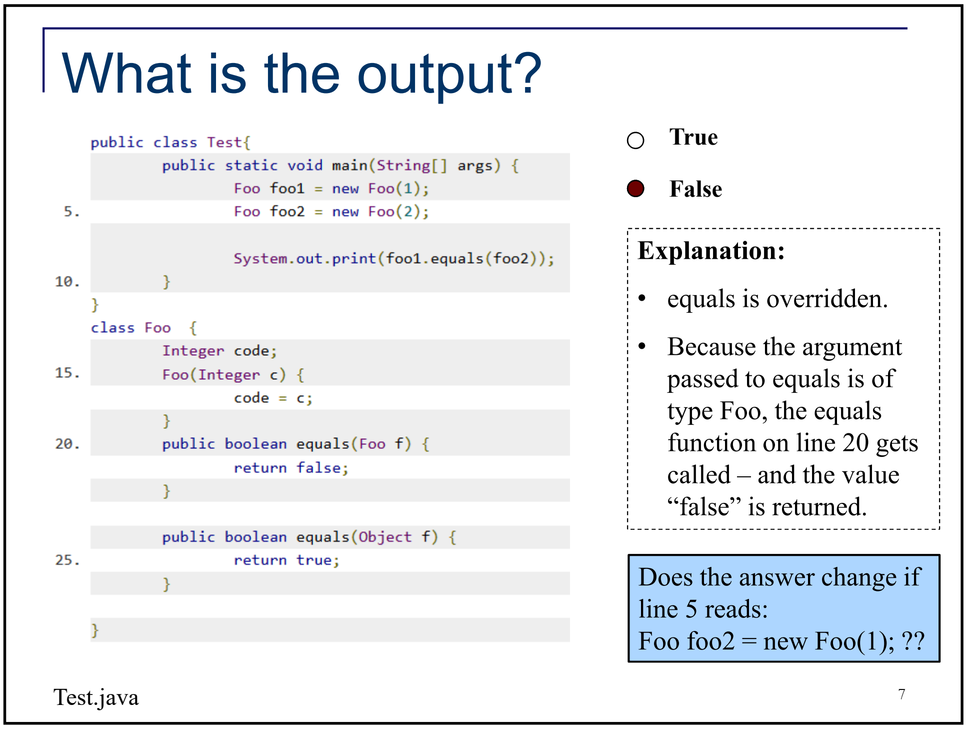
Default implementation in object class simply tests for identify.

Each class should define what it means by equals.



Put another way:

- == returns true, if the variable reference points to the **same object in memory**. This is called "shallow comparison".
- The equals() method calls the user implemented equals() method, which **compares the object attribute values**. The equals() method provides "deep comparison" by checking if two objects are logically equal.
- If equals() method is not overriden, the default is used and object.equals() works just like the "==" operator.



What is the output?

```
    public class Test{
            public static void main(String[] args) {
                    Foo foo1 = new Foo(1);
 5.                 Foo foo2 = new Foo(2);

                    System.out.print(foo1.equals(foo2));
10.         }
    }
    class Foo  {
            Integer code;
15.         Foo(Integer c) {
                    code = c;
            }
20.         public boolean equals(Foo f) {
                    return false;
            }

            public boolean equals(Object f) {
25.                 return true;
            }

    }
```

○ **True**

● **False**

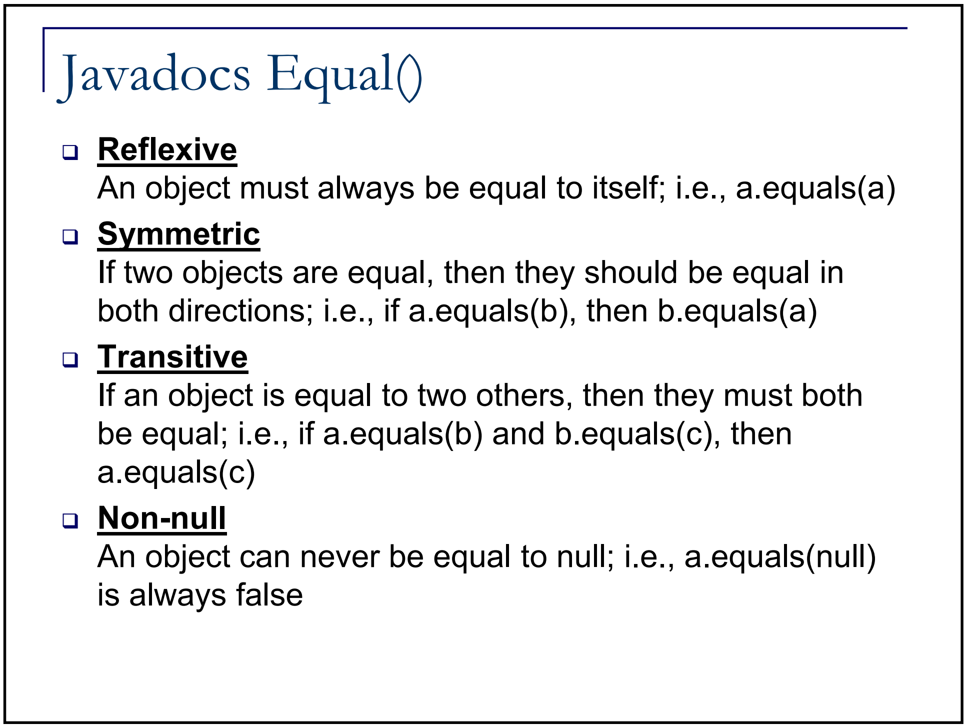
Explanation:

- equals is overridden.
- Because the argument passed to equals is of type Foo, the equals function on line 20 gets called – and the value “false” is returned.

Does the answer change if line 5 reads:
Foo foo2 = new Foo(1); ??

Test.java

7



Javadocs Equal()

- ❑ **Reflexive**
  An object must always be equal to itself; i.e., a.equals(a)
- ❑ **Symmetric**
  If two objects are equal, then they should be equal in both directions; i.e., if a.equals(b), then b.equals(a)
- ❑ **Transitive**
  If an object is equal to two others, then they must both be equal; i.e., if a.equals(b) and b.equals(c), then a.equals(c)
- ❑ **Non-null**
  An object can never be equal to null; i.e., a.equals(null) is always false

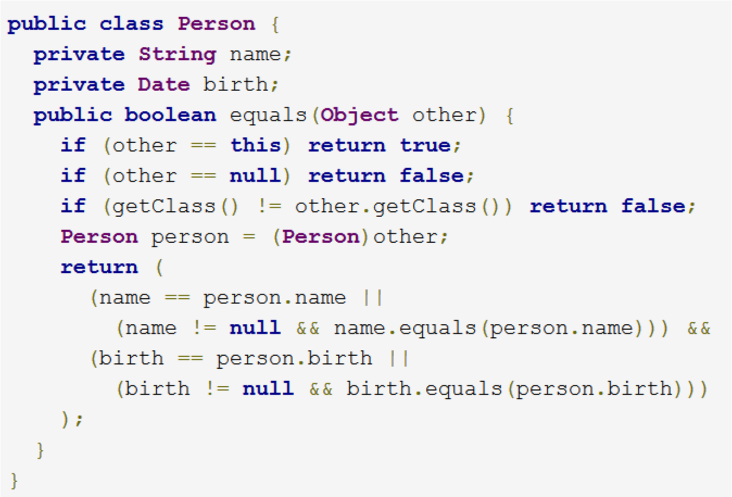
The Equals Method should Compare

```
public class Point {
  private static double version = 1.0;
  private transient double distance;
  private int x, y;
  public boolean equals(Object other) {
    if (other == this) return true;
    if (other == null) return false;
    if (getClass() != other.getClass())
        return false;
    Point point = (Point)other;
    return (x == point.x && y == point.y);
  }
}
```

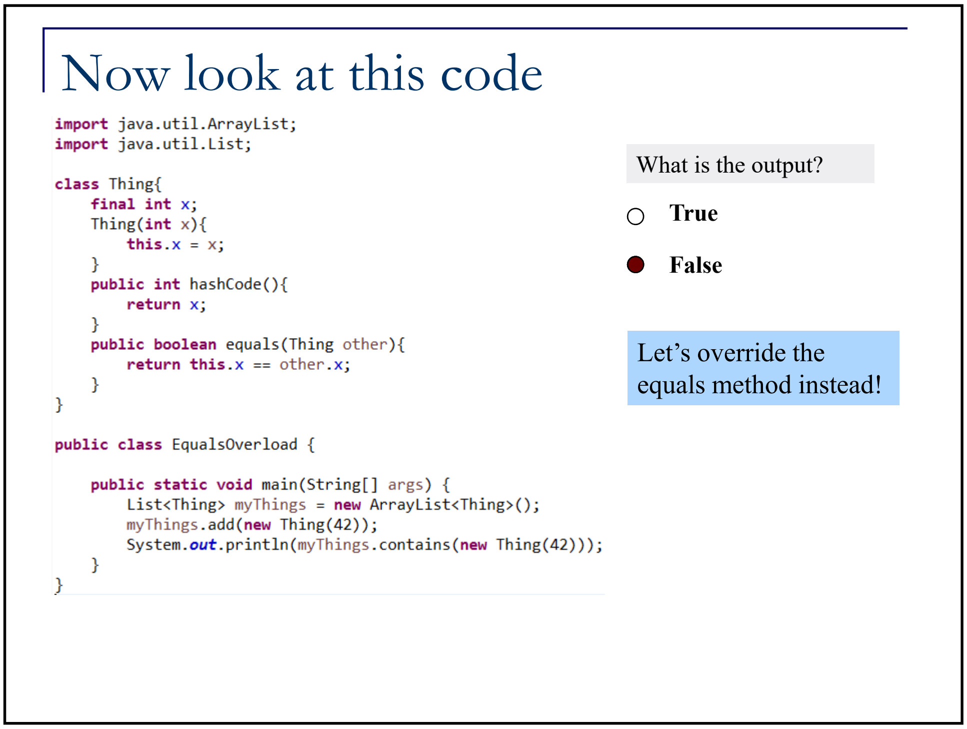
- If the argument is **this**;
  - return true (reflexivity)
- If the argument is null
  - return false (non-null)
- If the argument is of a different type
  - return false (symmetry)

https://www.youtube.com/watch?v=7V3589CReug

Another Example



```
public class Person {
  private String name;
  private Date birth;
  public boolean equals(Object other) {
    if (other == this) return true;
    if (other == null) return false;
    if (getClass() != other.getClass()) return false;
    Person person = (Person)other;
    return (
      (name == person.name ||
        (name != null && name.equals(person.name))) &&
      (birth == person.birth ||
        (birth != null && birth.equals(person.birth)))
    );
  }
}
```



Now look at this code

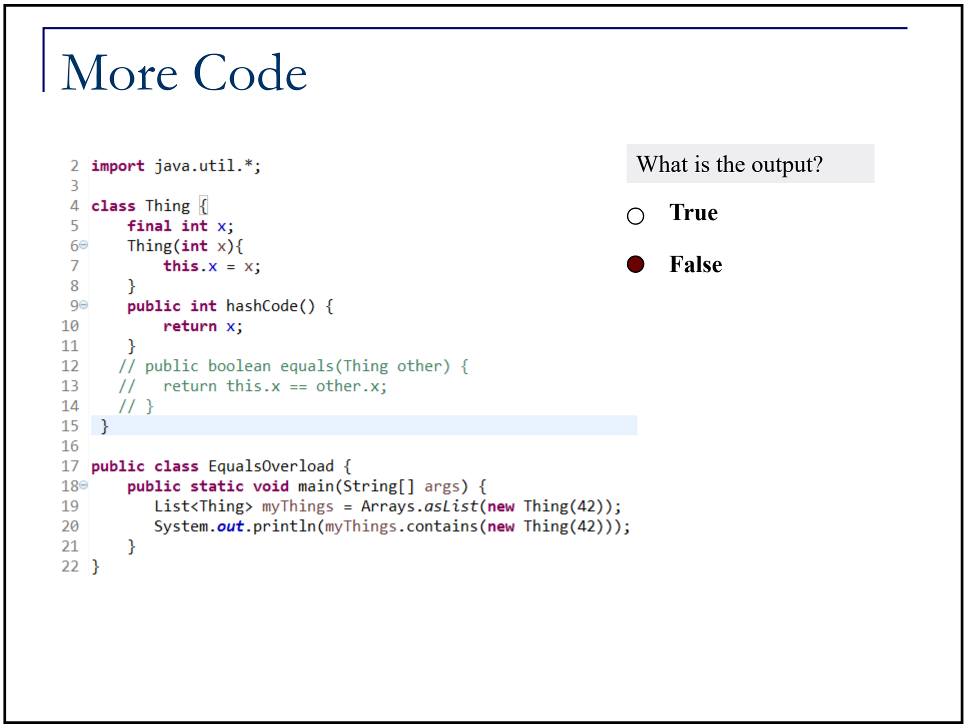
```
import java.util.ArrayList;
import java.util.List;

class Thing{
    final int x;
    Thing(int x){
        this.x = x;
    }
    public int hashCode(){
        return x;
    }
    public boolean equals(Thing other){
        return this.x == other.x;
    }
}

public class EqualsOverload {

    public static void main(String[] args) {
        List<Thing> myThings = new ArrayList<Thing>();
        myThings.add(new Thing(42));
        System.out.println(myThings.contains(new Thing(42)));
    }
}
```

What is the output?

○ **True**

● **False**

Let's override the equals method instead!



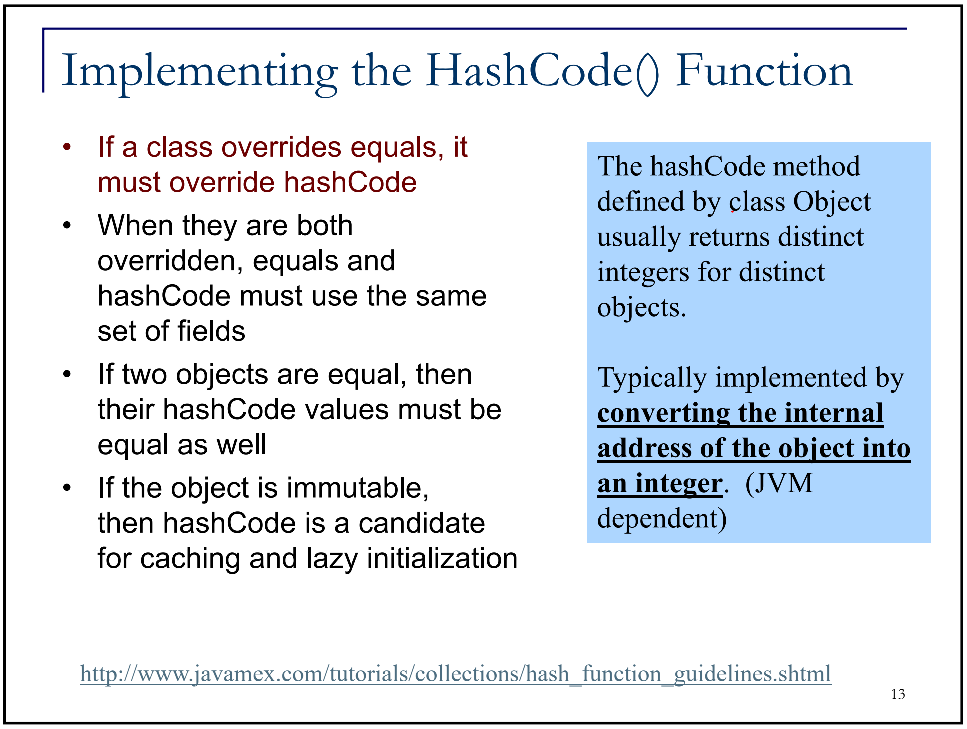
More Code

```
import java.util.*;

class Thing {
    final int x;
    Thing(int x){
        this.x = x;
    }
    public int hashCode() {
        return x;
    }
   // public boolean equals(Thing other) {
   //   return this.x == other.x;
   // }
}

public class EqualsOverload {
    public static void main(String[] args) {
        List<Thing> myThings = Arrays.asList(new Thing(42));
        System.out.println(myThings.contains(new Thing(42)));
    }
}
```

What is the output?

○ **True**

● **False**



Implementing the HashCode() Function

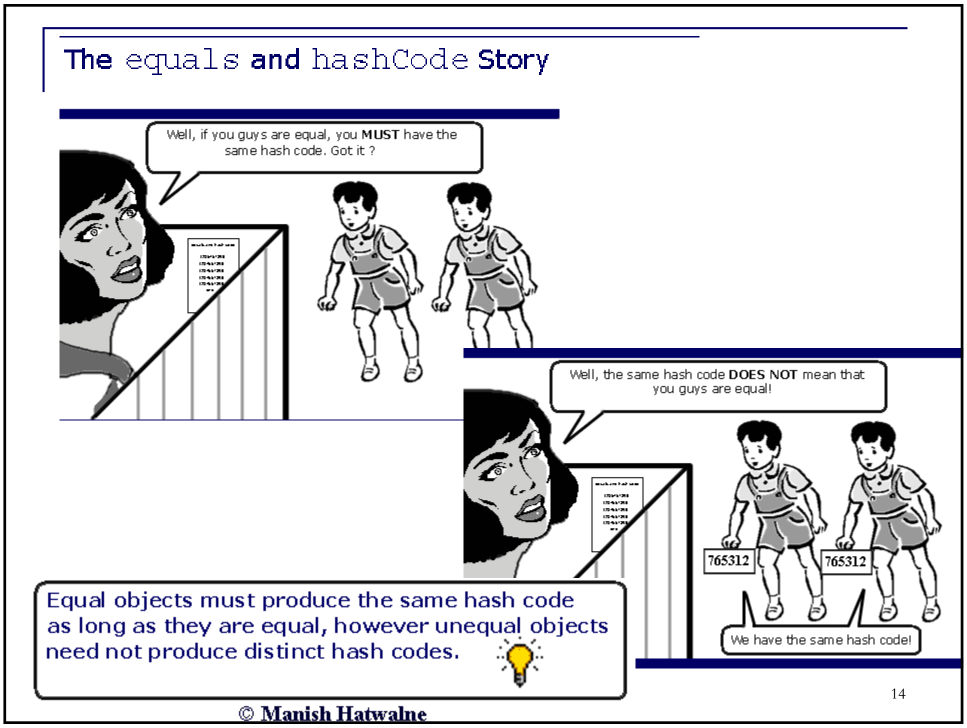
- If a class overrides equals, it must override hashCode
- When they are both overridden, equals and hashCode must use the same set of fields
- If two objects are equal, then their hashCode values must be equal as well
- If the object is immutable, then hashCode is a candidate for caching and lazy initialization

The hashCode method defined by class Object usually returns distinct integers for distinct objects.

Typically implemented by **converting the internal address of the object into an integer**. (JVM dependent)

http://www.javamex.com/tutorials/collections/hash_function_guidelines.shtml

13



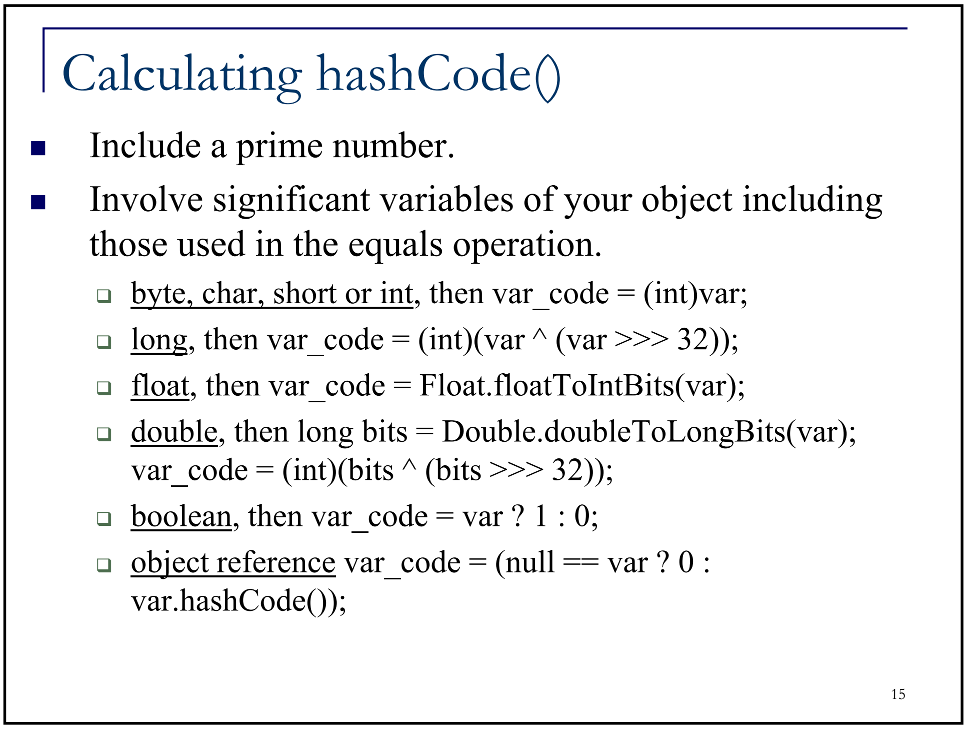
The equals and hashCode Story

Well, if you guys are equal, you MUST have the same hash code. Got it ?
Well, the same hash code DOES NOT mean that you guys are equal!
765312
765312
We have the same hash code!

Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.

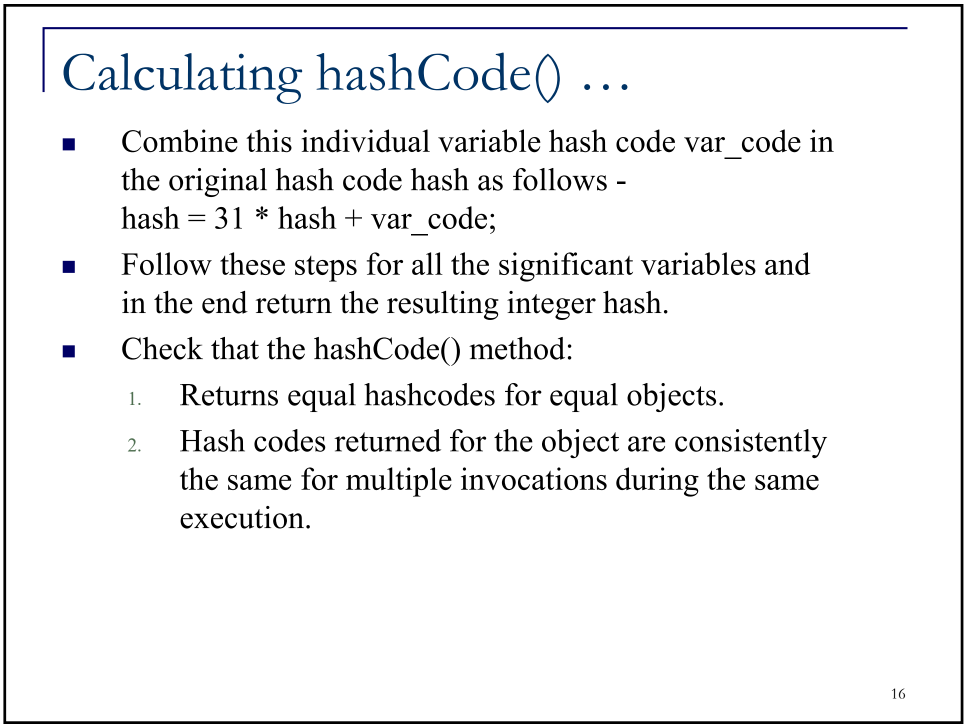
© Manish Hatwalne

14



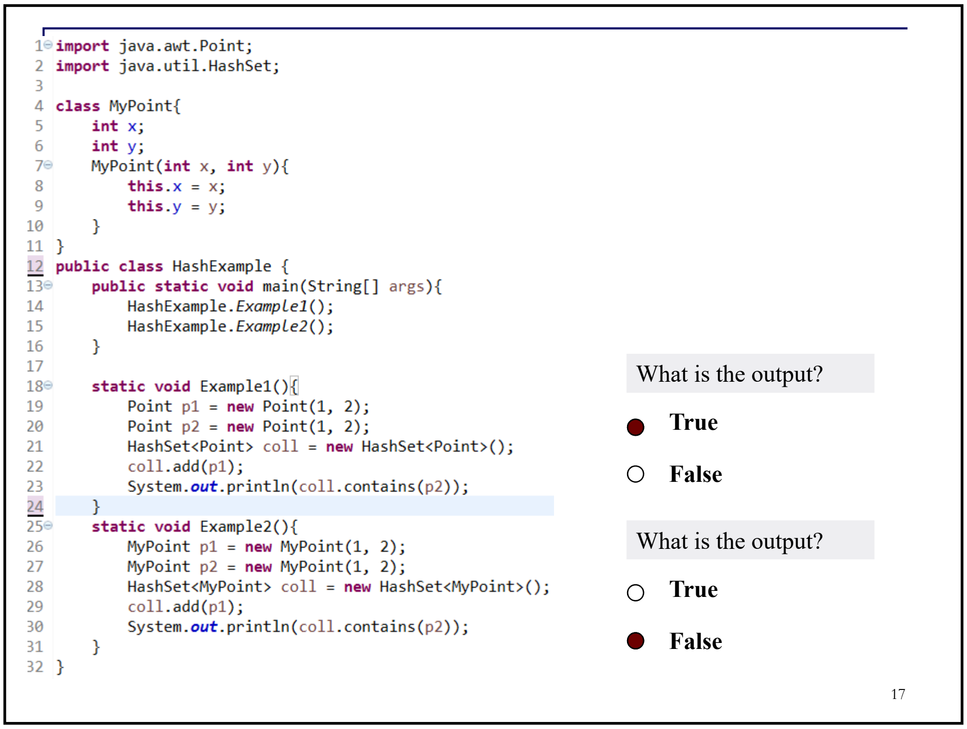
Calculating hashCode()

- Include a prime number.
- Involve significant variables of your object including those used in the equals operation.
  - byte, char, short or int, then var_code = (int)var;
  - long, then var_code = (int)(var ^ (var >>> 32));
  - float, then var_code = Float.floatToIntBits(var);
  - double, then long bits = Double.doubleToLongBits(var); var_code = (int)(bits ^ (bits >>> 32));
  - boolean, then var_code = var ? 1 : 0;
  - object reference var_code = (null == var ? 0 : var.hashCode());



Calculating hashCode() …

- Combine this individual variable hash code var_code in the original hash code hash as follows -
  hash = 31 * hash + var_code;
- Follow these steps for all the significant variables and in the end return the resulting integer hash.
- Check that the hashCode() method:
  1. Returns equal hashcodes for equal objects.
  2. Hash codes returned for the object are consistently the same for multiple invocations during the same execution.



```java
import java.awt.Point;
import java.util.HashSet;

class MyPoint{
    int x;
    int y;
    MyPoint(int x, int y){
        this.x = x;
        this.y = y;
    }
}
public class HashExample {
    public static void main(String[] args){
        HashExample.Example1();
        HashExample.Example2();
    }

    static void Example1(){
        Point p1 = new Point(1, 2);
        Point p2 = new Point(1, 2);
        HashSet<Point> coll = new HashSet<Point>();
        coll.add(p1);
        System.out.println(coll.contains(p2));
    }
    static void Example2(){
        MyPoint p1 = new MyPoint(1, 2);
        MyPoint p2 = new MyPoint(1, 2);
        HashSet<MyPoint> coll = new HashSet<MyPoint>();
        coll.add(p1);
        System.out.println(coll.contains(p2));
    }
}
```

What is the output?

● **True**

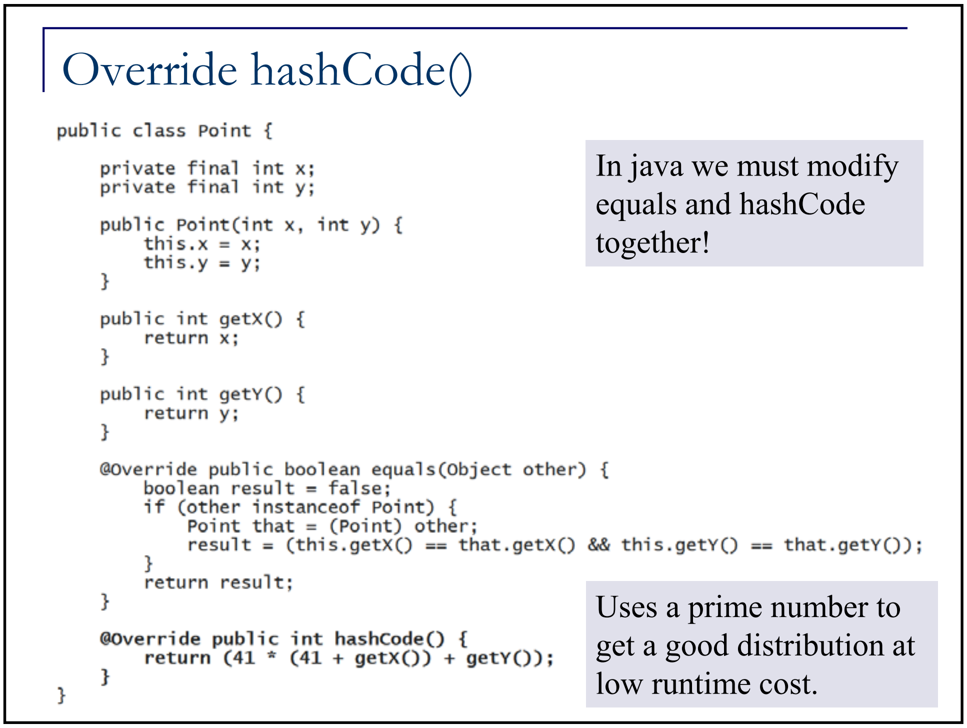
○ **False**

What is the output?

○ **True**

● **False**

17



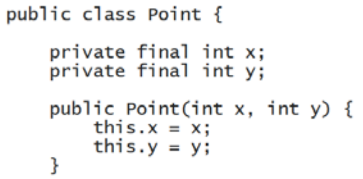
Override hashCode()

```java
public class Point {

    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof Point) {
            Point that = (Point) other;
            result = (this.getX() == that.getX() && this.getY() == that.getY());
        }
        return result;
    }

    @Override public int hashCode() {
        return (41 * (41 + getX()) + getY());
    }
}
```

In java we must modify equals and hashCode together!

Uses a prime number to get a good distribution at low runtime cost.

Defining equals in terms of Mutable Fields



```
public class Point {

    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
```

What if we hadn’t defined equals in terms of immutable (final) fields?

What might happen if an object was placed into a hash bucket and then its values were changed???

Mistakes like this create ‘hard to fix’ bugs!! Try to avoid them by following good practices.

toString()

- Returns a string representation of the object that "textually represents" the object.
- The result should be a concise but informative representation that is easy for a person to read.

```
class Vec {
  Vec() { }
  Vec(double xx, double yy) {
       x = xx;
       y = yy;
  }
  double x, y;
}
```

- return "X: " + Double.toString(x) + ", Y: " + Double.toString(y);