

Programming Languages and Techniques (CIS120)

Lecture 28

November 9th, 2015

Overriding, Equality, Abstract Classes

Announcements

- Midterm 2 has been graded
 - More info available on Weds. (after make-up exams are done)
- HW07: PennPals is available
 - Due Tuesday: November 17th
 - Start Early!
 - Emphasizes: Java Collections, Design

Method Overriding

A Subclass can *Override* its Parent

```
public class C {  
    public void printName() { System.out.println("I'm a C"); }  
}  
  
public class D extends C {  
    public void printName() { System.out.println("I'm a D"); }  
}  
  
// somewhere in main  
C c = new D();  
c.printName();
```

What gets printed to the console?

1. I'm a C
2. I'm a D
3. NullPointerException
4. NoSuchMethodException

A Subclass can *Override* its Parent

```
public class C {  
    public void printName() { System.out.println("I'm a C"); }  
}  
  
public class D extends C {  
    public void printName() { System.out.println("I'm a D"); }  
}  
  
// somewhere in main  
C c = new D();  
c.printName();
```

- Our ASM model for dynamic dispatch already explains what will happen when we run this code.
- Useful for changing the default behavior of classes.
- But... can be confusing and difficult to reason about if not used carefully.

Overriding Example

Workspace

```
C c = new D();  
c.printName();>
```

Stack

Heap

Class Table

Object

String toString(){...}

boolean equals...

...

C

extends

C() { }

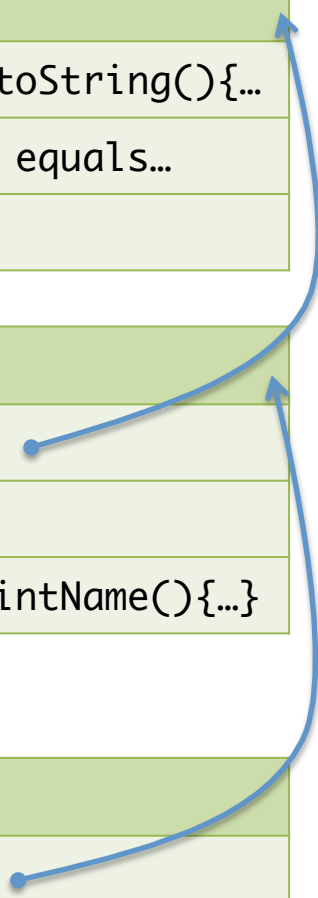
void printName(){...}

D

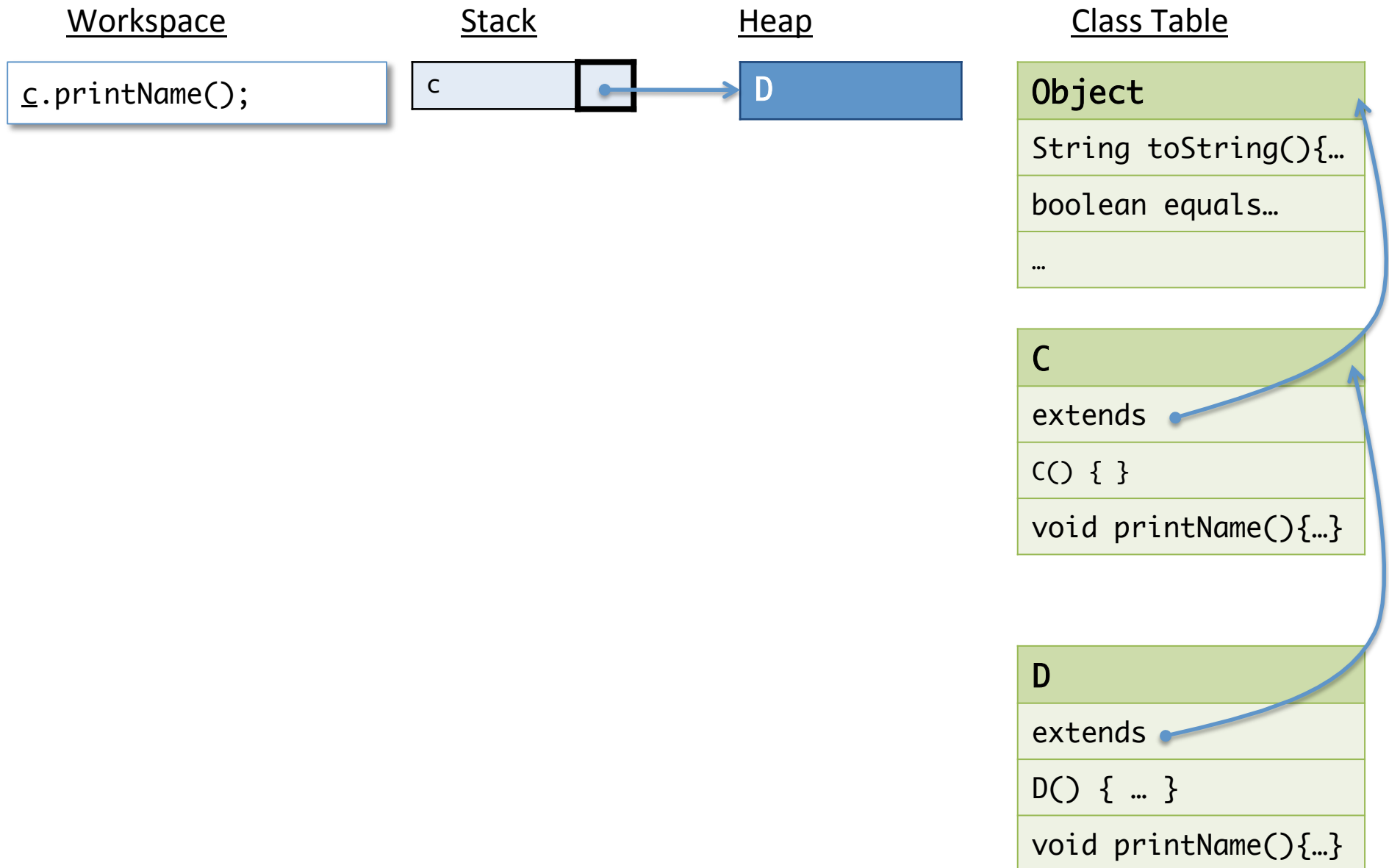
extends

D() { ... }

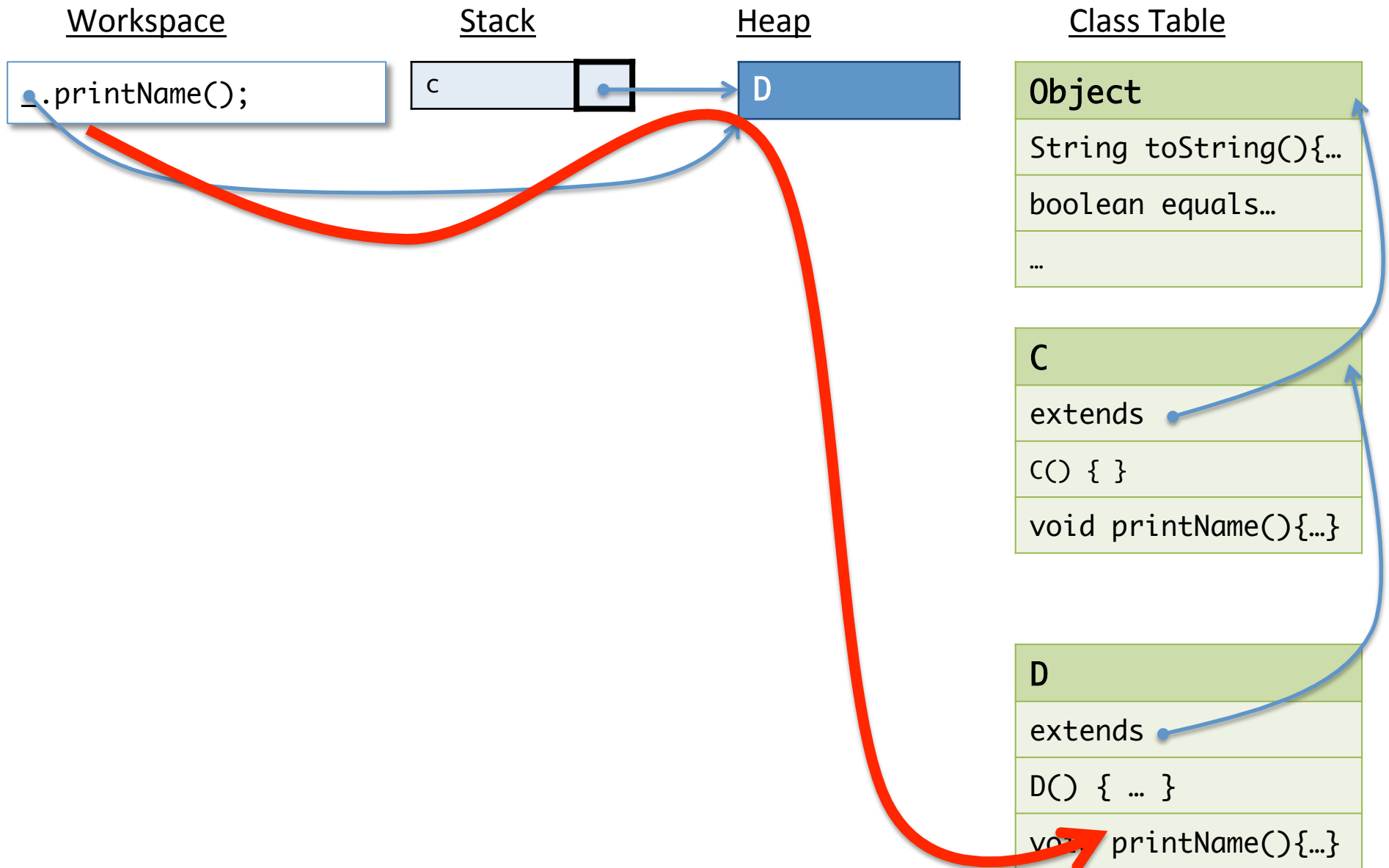
void printName(){...}



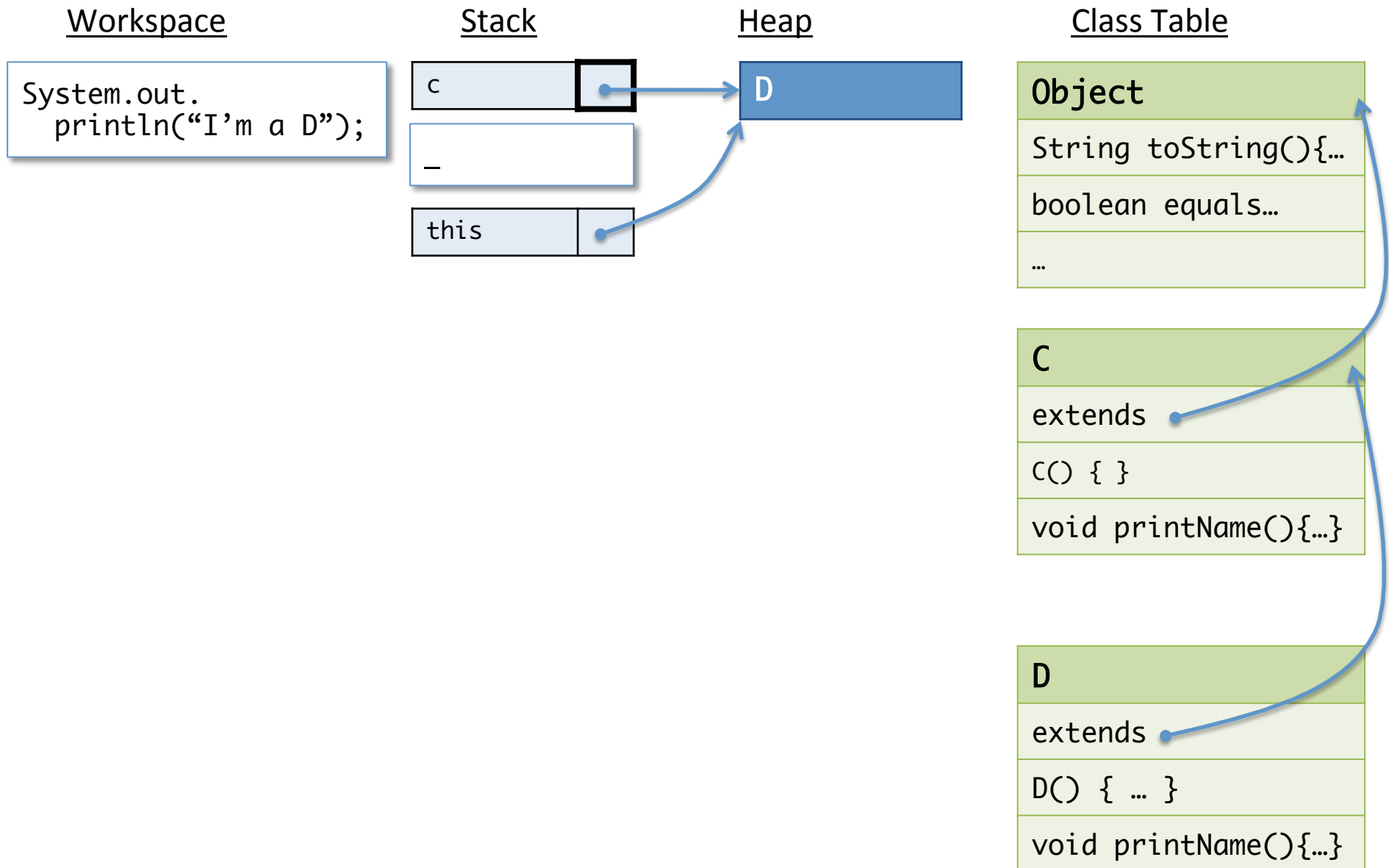
Overriding Example



Overriding Example



Overriding Example



Difficulty with Overriding

```
class C {  
    public void printName() {  
        System.out.println("I'm a " + getName());  
    }  
  
    public String getName() {  
        return "C";  
    }  
}  
  
class E extends C {  
    public String getName() {  
        return "E";  
    }  
}  
  
// in main  
C c = new E();  
c.printName();
```

What gets printed to the console?

1. I'm a C
2. I'm a E
3. I'm an E
4. NullPointerException

Difficulty with Overriding

```
class C {  
    public void printName() {  
        System.out.println("I'm a " + getName());  
    }  
  
    public String getName() {  
        return "C";  
    }  
}  
  
class E extends C {  
    public String getName() {  
        return "E";  
    }  
}  
  
// in main  
C c = new E();  
c.printName();
```

The C class might be in another package, or a library...

Whoever wrote E might not be aware of the implications of changing `getName`.

Overriding the method causes the behavior of `printName` to change!

- Overriding can break invariants/abstractions relied upon by the superclass.

Case study: Equality

Consider this example

```
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; }  
}
```

```
// somewhere in main...  
List<Point> l = new LinkedList<Point>();  
l.add(new Point(1,2));  
System.out.println(l.contains(new Point(1,2)));
```

What gets printed to the console?

1. true
2. false

Why?

When to override equals

- In classes that represent immutable *values*
 - String already overrides equals
 - Our Point class is a good candidate
- When there is a “logical” notion of equality
 - The collections library overrides equality for Sets (e.g. two sets are equal if and only if they contain equal elements)
- Whenever instances of a class might need to serve as *elements of a set* or as *keys in a map*
 - The collections library uses `equals` internally to define set membership and key lookup
 - (This is the problem with the example code)

When *not* to override equals

- When each instance of a class is inherently unique
 - *Often* the case for mutable objects (since its state might change, the only sensible notion of equality is identity)
 - Classes that represent “active” entities rather than data (e.g. threads, gui components, etc.)
- When a superclass already overrides equals and provides the correct functionality.
 - Usually the case when a subclass is implemented by adding only new methods, but not fields

How to override equals

*See the very nicely written article “How to write an Equality Method in Java” by Oderski, Spoon, and Venners (June 1, 2009) at <http://www.artima.com/lejava/articles/equality.html>

The contract for equals

- The equals method implements an *equivalence relation* on non-null objects.
- It is *reflexive*:
 - for any non-null reference value x, x.equals(x) should return true
- It is *symmetric*:
 - for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- It is *transitive*:
 - for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent:
 - for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified
- For any non-null reference x, x.equals(null) should return false.

Directly from: [http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals\(java.lang.Object\)](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object))

First attempt

```
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; }  
    public boolean equals(Point that) {  
        return (this.getX() == that.getX() &&  
                this.getY() == that.getY());  
    }  
}
```

Gocha: *overloading*, vs. *overriding*

```
public class Point {  
    ...  
    // overloaded, not overridden  
    public boolean equals(Point that) {  
        return (this.getX() == that.getX() &&  
                this.getY() == that.getY());  
    }  
}  
Point p1 = new Point(1,2);  
Point p2 = new Point(1,2);  
Object o = p2;  
System.out.println(p1.equals(o));  
// prints false!  
System.out.println(p1.equals(p2));  
// prints true!
```

The type of equals as declared in Object is:

```
public boolean equals(Object o)
```

The implementation above takes a Point *not* an Object!

Overriding equals, take two

Properly overridden equals

```
public class Point {  
    ...  
    @Override  
    public boolean equals(Object o) {  
        // what do we do here??  
    }  
}
```

- Use the `@Override` annotation when you *intend* to override a method so that the compiler can warn you about accidental overloading.
- Now what? How do we know whether the `o` is even a `Point`?
 - We need a way to check the *dynamic* type of an object.

instanceof

- The `instanceof` operator tests the *dynamic* type of any object

```
Point p = new Point(1,2);
Object o1 = p;
Object o2 = "hello";
System.out.println(p instanceof Point);
    // prints true
System.out.println(o1 instanceof Point);
    // prints true
System.out.println(o2 instanceof Point);
    // prints false
System.out.println(p instanceof Object);
    // prints true
```

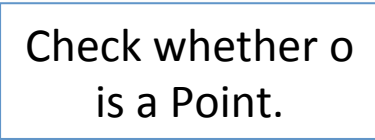
What gets printed? (1=true, 2=false)

- In the case of equals, instanceof is appropriate because the method behavior depends on the dynamic types of *two* objects: `o1.equals(o2)`
- But... use instanceof judiciously – usually dynamic dispatch is better.

Type Casts

- We can test whether o is a Point using instanceof

```
@Override
public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof Point) {
        // o is a point - how do we treat it as such?
    }
    return result;
}
```



Check whether o is a Point.

- Use a type *cast*: (Point) o
 - At compile time: the expression (Point) o has type Point.
 - At runtime: check whether the dynamic type of o is a subtype of Point, if so evaluate to o, otherwise raise a ClassCastException
 - As with instanceof, use casts judiciously – i.e. almost never. Instead use generics

Refining the equals implementation

```
@Override
public boolean equals(Object o) {
    boolean result = false;
    if (o.getClass() != getClass()) {
        Point that = (Point) o;
        result = (this.getX() == that.getX() &&
            this.getY() == that.getY());
    }
    return result;
}
```

This cast is
guaranteed to
succeed.

What about subtypes?

Equality and Subtypes

Suppose we extend Point like this

```
public class ColoredPoint extends Point {  
    private final int color;  
    public ColoredPoint(int x, int y, int color) {  
        super(x,y);  
        this.color = color;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        boolean result = false;  
        if (o instanceof ColoredPoint) {  
            ColoredPoint that = (ColoredPoint) o;  
            result = (this.color == that.color &&  
                super.equals(that));  
        }  
        return result;  
    }  
}
```

This version of equals is suitably modified to check the color field too.

Keyword **super** is used to invoke overridden methods.

Broken Symmetry

```
Point p = new Point(1,2);
ColoredPoint cp = new ColoredPoint(1,2,17);
System.out.println(p.equals(cp));
    // prints true
System.out.println(cp.equals(p));
    // prints false
```

What gets printed? (1=true, 2=false)

- The problem arises because we mixed Points and ColoredPoints, but ColoredPoints have more data that allows for finer distinctions.
- Should a Point *ever* be equal to a ColoredPoint?

Suppose Points *can* equal ColoredPoints

```
public class ColoredPoint extends Point {  
    ...  
    public boolean equals(Object o) {  
        boolean result = false;  
        if (o instanceof ColoredPoint) {  
            ColoredPoint that = (ColoredPoint) o;  
            result = (this.color == that.color &&  
                    super.equals(that));  
        } else if (o instanceof Point) {  
            result = super.equals(o);  
        }  
        return result;  
    }  
}
```

I.e., we repair the symmetry violation by checking for Point explicitly

Does this really work? (1=yes, 2=no)

Broken Transitivity

```
Point p = new Point(1,2);
ColoredPoint cp1 = new ColoredPoint(1,2,17);
ColoredPoint cp2 = new ColoredPoint(1,2,42);
System.out.println(p.equals(cp1));
    // prints true
System.out.println(cp1.equals(p));
    // prints true(!)
System.out.println(p.equals(cp2));
    // prints true
System.out.println(cp1.equals(cp2));
    // prints false(!!)
```

What gets printed? (1=true, 2=false)

- We fixed symmetry, but broke transitivity!
- Should a Point *ever* be equal to a ColoredPoint?

No!

Equality and Hashing

- Whenever you override equals you **must also** override hashCode in a compatible way
 - hashCode is used by the HashSet and HashMap collections
- Forgetting to do this can lead to extremely puzzling bugs!

Intentional Overriding

Abstract Classes

Abstract Classes

- Are like classes, but with some method implementations omitted.
 - They are instead declared abstract
- Must declare the class itself as abstract
- Non-abstract subclasses must provide an implementation of the missing methods
- Why? When there is a general algorithm whose implementation depends on the behavior of subtypes' implementation
 - E.g. remove-all defined in terms of remove in the Collections library

When To Override?

- Only override methods when the parent class is *designed* specifically to support such modifications:
 - If the library designer specifically describes the behavioral contract that the parent methods assume about overridden methods (e.g. equals, paintComponent)
 - If you're writing the code for both the parent and child class (and will maintain control of both parts as the software evolves) it might be OK to override.
 - Either way: document the design
 - Use the `@Override` annotation to mark intentional overriding
- Look for other means of achieving the desired outcome:
 - Use composition & delegation (i.e. wrapper objects) rather than overriding

How to prevent overriding

- By default, methods can be overridden in subclasses.
- The `final` modifier changes that.
- Final methods *cannot* be overridden in subclasses
 - Prevents subclasses from changing the “behavioral contract” between methods by overriding
 - `static final` methods cannot be hidden
- Similar, but not the same as final fields and local variables:
 - Act like the immutable name bindings in OCaml
 - Must be initialized (either by a static initializer or in the constructor) and cannot thereafter be modified.
 - `static final` fields are useful for defining constants (e.g. `Math.PI`)