Consider this example

```java
public class Point {
    private final int x; // see note about final*
    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 is printed to the terminal? Why?

*Recall that the ‘final’ modifier means that the field (or variable) cannot be modified once it is initialized – it is the closest analog in Java to OCaml’s let-binding for names.

---

Identity vs. Equality

- Object **identity** is “pointer equality”
  - Indicates where in the heap the object is located
  - Tested using ==

- Object **equality** is “value”, “logical”, or “deep” equality
  - Indicates when two objects are “the same” as values
  - Tested using the equals method inherited from Object

- The default implementation of equals is ==
  - In this case, instances are equal only to themselves

- Classes can override the default implementation to provide a different “logical” notion of equality.
  - e.g. String’s equals tests for identical sequences of characters.

---

Equality*

1. Identity vs. Equality
2. Pitfalls with implementing equals (with digressions)
3. Recipe for implementing equals

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

- What does it mean for two things to be equal?
  - It depends!
  - In what way is the equality being used?
- Answer 1: Mutable objects are (usually) only equal to themselves
  - Why?
- Answer 2: Two immutable objects (of the same type) are equal if their corresponding fields are equal
  - What if there are “unimportant” fields?
  - What if the objects are of different types?
- What is a reasonable definition of equality?

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.

Overriding equals

```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; }
    public boolean equals(Point p) {
        return (this.getX() == p.getX() &&
                this.getY() == that.getY());
    }
}
// somewhere in main
List<Point> l = new LinkedList<Point>();
l.add(new Point(1,2));
System.out.println(l.contains(new Point(1,2)));
```

First attempt
Overloading: A digression

How does an overloaded method invocation get resolved?

Overloaded method declaration

- Overloaded methods share the same name.
- Overloaded methods can differ based on the number and/or types of their parameters

```java
public class OverloadExample {
    public int m() { return 3; }  
    public int m(int x) { return x; }
    public int m(Integer i) { return i.intValue() + 100; }
    public int m(int x, int y) { return x + y; }
    public int m(Object o) { return 42; }
}
```

*This code uses overloading in poor style for the sake of example—overloaded methods should all behave “basically the same”. Otherwise, the programmer will be confused when m does one thing when passed an int and something completely different when passed an integer.*

equals was overloaded *not* overridden

```java
public class Point {
    // overloaded, not overridden
    public boolean equals(Point p) {
        return (this.getX() == p.getX() && this.getY() == that.getY());
    }

    // Illustrate the problem without using Lists:
    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!

Overloaded method invocation resolution

- Which version of an overloaded method is invoked depends on the number and *static* types of the arguments.

```java
OverloadExample oe = new OverloadExample();
System.out.println(oe.m());                  // prints 3
System.out.println(oe.m(17));               // prints 17
System.out.println(oe.m(new Integer(120))); // prints 220
System.out.println(oe.m(3,4));              // prints 7
System.out.println(oe.m("hello"));         // prints 42
```

Notes:
- This means methods *cannot* be overloaded based on return type.
Precise matching

If more than one method’s types match the invocation, the “most precise” match is chosen:

- Type $S$ is more precise than type $T$ if $S$ is a strict subtype of $T$.
- In our example there are two methods $m$ that could be chosen for $\text{oe.m(new Integer(3));}$

\[
\text{int } m(\text{Integer } i) \{ \ldots \} \quad \text{or } \quad \text{int } m(\text{Object } o) \{ \ldots \}
\]

- Precise matching says that the left method is the one used, because $\text{Integer}$ is more precise than $\text{Object}$.
- For methods with multiple arguments, the most precise match is the one that has the greatest number of more precisely matching arguments.
  - Is there always such a method?

Overloading Summary

- Overloading is useful:
  - to provide flexible handling of similar kinds of data (e.g. $\text{int}$ vs. $\text{Integer}$)
  - to provide default arguments (i.e. overloading the number of arguments)

```java
// Typical good use of overloading
public class Point {
    private int x, y;
    public Point() { this.x = 0; this.y = 0; }
    public Point(int x, int y) { this.x = x; this.y = y; }
    ...
}
```

- Avoid ambiguous overloading!

Ambiguous overloading

```java
public class OverloadExample {
    public int m(Integer i, Object o) {
        return i.intValue() + 42;
    }

    public int m(Object o, Integer i) {
        return i.intValue() - 42;
    }
}
```

OverloadExample $\text{oe} = \text{new OverloadExample()}$;
$\text{System.out.println(oe.m(new Integer(3), "hello"));}$
$\text{System.out.println(oe.m("hello", new Integer(3)));}$
$\text{System.out.println(oe.m(new Integer(3), new Integer(4)));}$

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.

Type Casts

public class Point {
    @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;
        }
    }
}

• We can test whether o is a Point using instanceof

• 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

instanceof

• Java provides the instanceof operator that tests the dynamic type of any object.
  – Note: (null instanceof C) returns false for all C

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

// Some instanceof tests are nonsensical:
System.out.println(p instanceof String);  // compile-time error

• 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 preferred.
  – In fact, one could argue that overriding equals (and related “multimethods”) is the only time one should use instanceof

Refining the equals implementation

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

This cast is guaranteed to succeed.

Now the example code from the slide 2 will behave as expected.
But... are we done? Does this implementation satisfy the contract?
Equality and Subtypes

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;
    }
}

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

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

Suppose we extend Point like this

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

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(that);
        }
        return result;
    }
}

• We can repair the symmetry violation by checking for Point explicitly.
• Does this work?
Broken Transitivity

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

• We fixed symmetry, but broke transitivity!
• Should a Point ever be equal to a ColoredPoint?
  – No!

A Recipe for Equality*

*Even this isn’t the final story – there is another version that uses reflection to check for class names. It doesn’t work well with anonymous classes or subclasses that add only methods, but is simpler in other ways. See the Odersky article for a discussion of the tradeoffs.

Add a canEqual method.

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

Override equals and canEqual

public class ColoredPoint extends Point {
  ...
  @Override public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof ColoredPoint) {
      ColoredPoint that = (ColoredPoint) o;
      result = (that.canEqual(this) &&
                this.color == that.color &&
                super.equals(that));
    }
    return result;
  }
  @Override public boolean canEqual(Object other) {
    return (other instanceof ColoredPoint);
  }
}
The equals Recipe

```java
public class C extends D {
    
    @Override public boolean equals(Object o) {
        if (this == o) return true;
        boolean result = false;
        if (o instanceof C) {
            C that = (C) o;
            result = (that.canEqual(this) &&
                this.field1 == that.field1 &&
                this.field2.equals(that.field2) &&
                super.equals(that));
        }
        return result;
    }

    public boolean canEqual(Object other) {
        return (other instanceof C);
    }
}
```

1. Override equals at the right type.
2. Return true in the case of identity. (This is an optimization.)
3. Use instanceof and cast to check the other object’s type.
4. Implement the canEqual method and use the other object’s version of it.
5. Compare all the corresponding fields, deferring to the superclass if needed.

Field Comparison

- When do you use == to compare fields?
  - for fields that store primitive types (int, boolean, etc.)
  - (often) when the field is a reference to a mutable object
  - when you want “shallow” equality (you don’t want to follow pointers)

- When do you use equals to compare fields?
  - for references to immutable “value” types (like String or Point)
  - when you want to do “deep” equality (e.g. for singly-linked lists)
  - my model: use equals to compare objects whose representations in OCaml would not use the “ref” keyword (e.g. trees, etc.)

- Be careful about cycles!
  - It’s easy to cause equals to go into an infinite loop for cyclic (often mutable) data structures.

- It’s usually appropriate to defer the superclass to check its fields.
  - But not in the first class to override equals! (Object uses pointer equality, remember!)

- Fields might be accessed directly or through accessor methods.

Some caveats

- Whenever you override equals you must also override canEqual (assuming you follow the recipe given here)
  - or provide it if the class is the first one in the inheritance tree to override equals

- Whenever you override equals you must also override hashCode in a compatible way
  - hashCode is used by the HashSet and HashMap collections

- We’ll talk about hashing more in the next lecture.

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 the same elements)

- Whenever instances of a class can 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 one slide 2.)
When *not* to override equals

- 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 and not data (e.g. threads or gui components, etc.)

- A superclass already overrides equals and provides the correct functionality.
  - Usually the case when a subclass adds only new methods, not fields