Programming Languages and Techniques (CIS1200)

Lecture 28

Overriding, Equality, Iteration

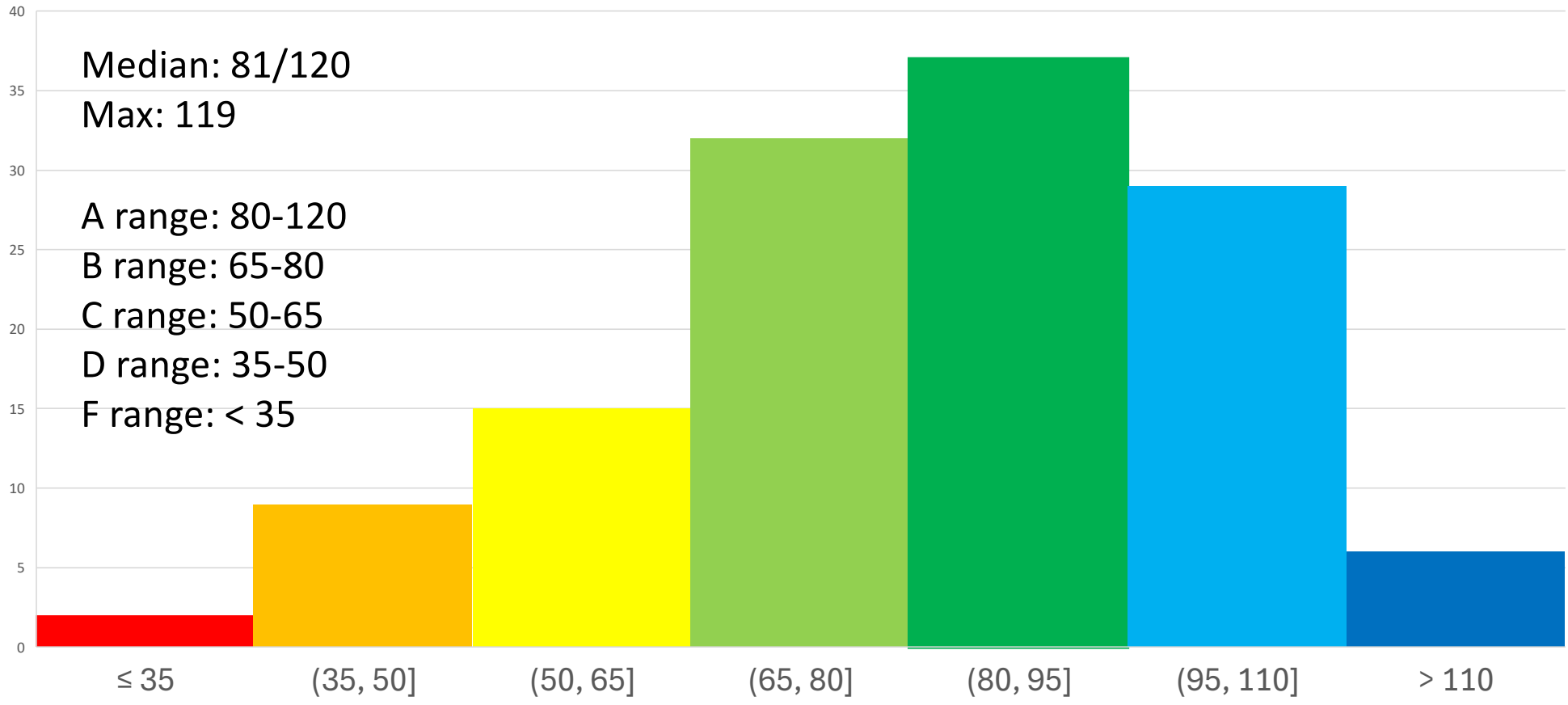
Chapters 25 and 26

Announcements

- HW07: PennPals
  - Programming with Java Collections
  - Due Tuesday, April 8th at 11.59pm

- Midterm 2 results available after class
  - View scores on Gradescope, solutions on website
  - Submit regrade requests in the next two weeks
  - Use letter grade chart to interpret your performance
  - My OH next week are by appointment only (see Ed to schedule)

CIS 1200 25sp Midterm 2



Review: Comparable

```
import java.util.*;

class Point {
    private final int x, y;
    public Point(int x0, int y0) { x = x0; y = y0; }
    public int getX(){ return x; }
    public int getY(){ return y; }
}

public class TreeSetDemo {
    public static void main(String[] args) {
        Set<Point> s = new TreeSet<>();
        s.add(new Point(1,1));
    }
```

RUNTIME ERRROR

```
Exception in thread "main" java.lang.ClassCastException:
                    Point cannot be cast to java.base/java.lang.Comparable
at java.base/java.util.TreeMap.compare(TreeMap.java:1291)
at java.base/java.util.TreeMap.put(TreeMap.java:536)
at java.base/java.util.TreeSet.add(TreeSet.java:255)
at TreeSetDemo.main(TreeSetDemo.java:14)
```

Adding Comparable to Point

```java
import java.util.*;

class Point implements Comparable<Point> {
    private final int x, y;
    public Point(int x0, int y0) { x = x0; y = y0; }
    public int getX(){ return x; }
    public int getY(){ return y; }

    public int compareTo(Point o) {
        if (this.x < o.x) {
            return -1;
        } else if (this.x > o.x) {
            return 1;
        } else if (this.y < o.y) {
            return -1;
        } else if (this.y > o.y) {
            return 1;
        }
        return 0;
    }
}
```

```java
Point p1 = new Point(0,1);
Point p2 = new Point(0,2);
p1.compareTo(p2);  // -1
p2.compareTo(p1);  // 1
p1.compareTo(p1);  // 0
```

Digging Deeper into Comparable

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. *In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals method.*

How do we change the definition of equals?

Equality

A case study in overriding

29: What gets printed to the console?

0

True

0%

False

0%

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

Answer: 2

Why?

Overriding Example

```
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. NullPointerException

Answer: I'm a E

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 object references:

- 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 objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

Directly from https://docs.oracle.com/en/java/javase/17/docs/api/java.base/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.x == that.x &&
            this.y == that.y);
  }

}
```

28: What is the result of: p1.equals(p2) ?

```
public class Point {
  …
    public boolean equals(Point that) {
      return (this.x == that.x &&
              this.y == that.y);
  }
}

// somewhere in main
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
```

true 0%

false 0%

NullPointerException 0%

Unsure 0%

28: What is the result of: p1.equals(null)?

```
public class Point {
  …
    public boolean equals(Point that) {
      return (this.x == that.x &&
              this.y == that.y);
  }
}

// somewhere in main
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
```

true	0%
false	0%
NullPointerException	0%
Unsure	0%

28: What is result of: Object o = p2; p1.equals(o) ?

```
public class Point {
  …
    public boolean equals(Point that) {
      return (this.x == that.x &&
              this.y == that.y);
  }
}

// somewhere in main
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
```

true	0%
false	0%
NullPointerException	0%
Unsure	0%

Got'cha: *overloading*, vs. *overriding*

```
public class Point {
  …
  // overloaded, not overridden
  public boolean equals(Point that) {
    return (this.x == that.x &&
            this.y == that.y);
  }
}
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
  - modern IDEs such as IntelliJ will automatically add/suggest these annotations
- 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

- 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 instanceof Point) {
        Point that = (Point) o;
        result = (this.x == that.x &&
                  this.y == that.y);
    }
    return result;
}
```

This cast is guaranteed to succeed.

"*dynamic cast*" or "*type cast*"
"*downcast*" or "*coercion*"

What about subtypes?

Suppose we define a subclass of Point

```
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.

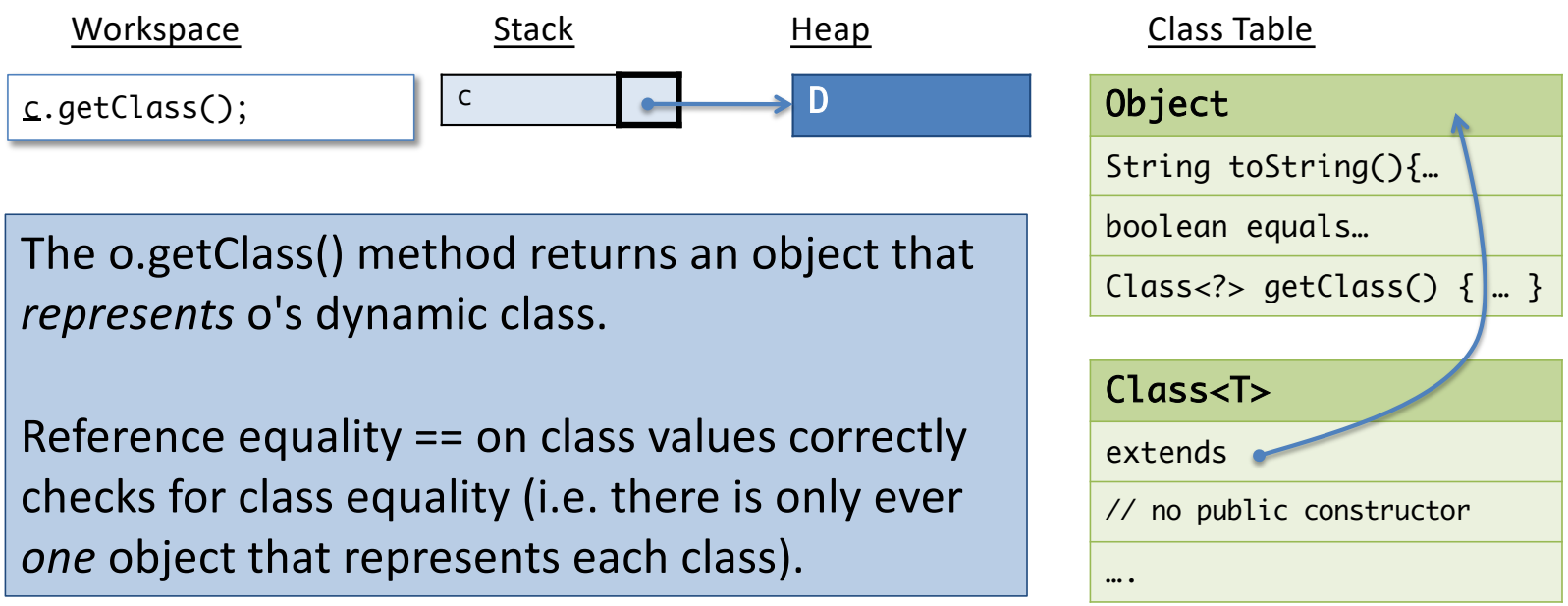
Broken Symmetry

```
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
```

- 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?

Java Reflection: getClass

- To correctly account for subtyping, we need the classes of the two objects to match *exactly*.
- instanceof only lets us ask about the subtype relation
- How do we access the dynamic class?



The o.getClass() method returns an object that *represents* o's dynamic class.

Reference equality == on class values correctly checks for class equality (i.e. there is only ever *one* object that represents each class).

Correct Implementation: Point

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) {
      return false;
    }
    Point point = (Point) o;
    return {
      x == point.x && y == point.y;
    }
}
```

Check whether o is a Point

"*dynamic cast*" or "*type cast*"
"*downcast*" or "*coercion*"

The class cast expression "(T)e" is a runtime test of the dynamic class of of e.
If T is not a subtype of the dynamic class, then a ClassCastException is thrown.
The static type of the expression "(T)e" is T.

Compatibility with compareTo

- For classes that implement the Comparable<E> interface, the equals and compareTo methods should agree:
  - o.compareTo(p) == 0 exactly when o.equals(p)

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) {
      return false;
    }
    Point point = (Point) o;
    return (this.compareTo(point) == 0);

}
```

Can implement equals by using compareTo.

Overriding Equality in Practice

- IntelliJ can autogenerate equality methods of the kind we developed.
  - But you need to specify which fields should be taken into account.
  - and you should know why some comparisons use == and some use .equals

- 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 puzzling bugs!

When to override equals?

- In classes that represent immutable *values*
  - String already overrides equals
  - The 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 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`)