CS 251
Intermediate Programming
Overriding equals and hashCode

Brooke Chenoweth

University of New Mexico

Spring 2025

equals method in Object

```
public boolean equals(Object obj)
```

- The Object class provides the equals method to indicate if some other object is “equal to” this one.
- Implementation in Object class checks to see if the two objects are the same object in memory. (That is, uses the == operator to compare the references)
- Often want to override to compare fields instead.
- If you override `equals`, you generally also should override `hashCode`

Requirements for equals

The `equals` method implements an *equivalence relation* on non-null object references.
For any non-null objects `x` and `y`

- Reflexive – `x.equals(x)` returns true
- Symmetric – `x.equals(y)` iff `y.equals(x)`
- Transitive – if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`
- Consistent – multiple calls to equals with same x and y give the same value (unless the objects have changed)
- `x.equals(null)` returns false

Example – 2D Point

```
public class Point {

  private int x;
  private int y;

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

  // equals, hashcode, etc.
}
```

Want two points to be equal iff x and y are equal.

equals – attempt 1

```
public boolean equals(Point other) {
  return x == other.x && y == other.y;
}
```

What is wrong?

equals – attempt 1

```
public boolean equals(Point other) {
  return x == other.x && y == other.y;
}
```

What is wrong?
Not actually overriding equals!
Parameter should be an Object, not a Point, so this is overloading.

equals – attempt 2

```
@Override
public boolean equals(Object other) {
  return x == other.x && y == other.y;
}
```

What is wrong?

equals – attempt 2

```
@Override
public boolean equals(Object other) {
  return x == other.x && y == other.y;
}
```

What is wrong?
Won't compile since Object doesn't have x or y fields.
We'll need to cast to a Point to access them.
How can we be sure the cast will succeed?

equals – working version

```
@Override
public boolean equals(Object obj) {
  if(obj instanceof Point) {
    Point other = (Point)obj;
    return x == other.x && y == other.y;
  } else {
    return false;
  }
}
```

equals – working version

```
@Override
public boolean equals(Object obj) {
  if(obj instanceof Point) {
    Point other = (Point)obj;
    return x == other.x && y == other.y;
  } else {
    return false;
  }
}
```

JDK 17 introduced pattern matching in instanceof, so we don't need the separate cast line anymore

equals – shorter version

```
@Override
public boolean equals(Object obj) {
  if(obj instanceof Point other) {
    return x == other.x && y == other.y;
  } else {
    return false;
  }
}
```

equals – shorter version

```
@Override
public boolean equals(Object obj) {
  if(obj instanceof Point other) {
    return x == other.x && y == other.y;
  } else {
    return false;
  }
}
```

I'll still compare the fields in this version even if I'm checking if object is equal to itself. Would be nice to skip that. . .

equals – check for self compare

```
@Override
public boolean equals(Object obj) {

  if(obj == this) return true;

  if(obj instanceof Point other) {
    return x == other.x && y == other.y;
  } else {
    return false;
  }
}
```

equals – alternate

```java
@Override
public boolean equals(Object obj) {

  if(obj == this) return true;

  try {
    Point other = (Point)obj;
    return x == other.x && y == other.y;
  } catch (ClassCastException ex) {
    return false;
  }
}
```

This version *may* be faster if cast always will succeed.

What is a hash code?

- A *hash function* maps arbitrary data to fixed-size values. These values are *hash codes*
  In Java, the `hashCode` method maps this Object to an `int` value.
- We use hash codes to index a fixed size table called a *hash table*.
  In Java, hashtables are used in `HashSet`, `HashMap`, etc.

Requirements for hashCode

- Multiple calls to hashCode on the same Object should always produce the same int result (unless the object has been modified)
- If two objects are equal according to the equals method, they *must* have the same hashCode result.
- If two objects are unequal, they do not have to have different hashCodes, but producing different results for unequal objects will give better performance.

Tips for implementing hashCode

- If single field is an int (or small enough to safely cast to int), just use that value
- If single field is an Object, call hashCode on field and use that
- If multiple fields, combine hashcode values in a way that reduces collisions.
- Most IDEs can help you generate reasonable equals and hashCode implementations.

hashCode

```
@Override
public int hashCode() {
  return 37*x + y;
}
```

The `Objects` class has a `hash` method to get a combined hashcode for multiple values.

```
@Override
public int hashCode() {
  return Objects.hash(x,y);
}
```

What about Comparable?

If you implement Comparable, make sure its result is consistant with your equals method.

```
public class Point implements Comparable<Point> {

  @Override
  public int compareTo(Point p) {
    if(x == p.x) {
      return y - p.y;
    } else {
      return x - p.x;
    }
  }
}
```

Please note: the subtraction trick I used here will fail if you run into integer overflow. More robust solution should compare with less than operator.

Avoid coding with records

If your data type is immutable (all the fields are final), you might prefer to use a record instead.

```
public record Point(int x, int y) {}
```

A record provides a constructor, accessors for all fields, and default implementations for equals, hashCode, and toString. (You can override these and/or add more methods, but all fields will be final.)