Option 2: The equals() Method

- Object defines an equals() method. By default, this method just does the same as ==.
  - Returns boolean, so can only test equality
  - Override it if you want it to do something different
  - Most (all?) of the core Java classes have properly implemented equals() methods

```
public EqualsTest {
    public int x = 8;

    public boolean equals(Object o) {
        EqualsTest e = (EqualsTest)o;
        return (this.x==e.x);
    }

    public static void main(String args[]) {
        EqualsTest t1 = new EqualsTest();
        EqualsTest t2 = new EqualsTest();
        System.out.println(t1==t2);
        System.out.println(t1.equals(t2));
    }
}
```

Equality

1. Override equals(Object o) in Object class
2. Check the argument is the correct type
3. Compare each element of state in the two objects

Java Collection
(LinkedList, TreeMap)

Comparisons

(Providing a default comparison for your class)

1. Implement Comparable<MyClass>
2. Create public int compareTo(MyClass m)
3. Make your comparisons
4. Sort → Collections.sort()

Option 3: Comparable<T> Interface I

int compareTo(T obj);

- Part of the Collections Framework
- Doesn't just tell us true or false, but smaller, same, or larger: useful for sorting.
- Returns an integer, r:
  - r<0 This object is less than obj
  - r==0 This object is equal to obj
  - r>0 This object is greater than obj

Option 3: Comparable<T> Interface II

```
public class Point  implements Comparable<Point> {
    private final int mX;
    private final int mY;
    public Point (int, int y) { mX=x; mY=y; }

    // sort by y, then x
    public int compareTo(Point p) {
        if ( mY>p.mY) return 1;
        else if (mY<p.mY) return -1;
        else {
            if (mX>p.mX) return 1;
            else if (mX<p.mX) return -1;
            else return 0.
        }
    }
}


// This will be sorted automatically by y, then x
Set<Point> list = new TreeSet<Point>();
```

Option 4: Comparator<T> Interface

int compareTo(T obj1, T obj2)

- Also part of the Collections framework and allows us to specify a particular comparator for a particular job
- E.g. a Person might have a compareTo() method that sorts by surname. We might wish to create a class AgeComparator that sorts Person objects by age. We could then feed that to a Collections object.

Comparator ← Custom (non-default) sorting order

1. Make a new class MyComparator
2. Implement Comparator<MyClass>
3. Create int compare(MyClass m1, MyClass m2)
4. Make your comparisons
5. Feed an object of type MyComparator to the sort method in Collections framework.

```
Collections.sort(mylist, new MyComparator())
```