# The Hidden Contract Between equals and Comparable

In Java we use the **equals**operation to check if two objects are meaningfully equal or not, on other hand we use the **comparable**to compare two objects.

**The question is: Do equals and comparable have any hidden relationship? Do they have to maintain any contract?**

We know when we override the **equals**method in Java, we always have to override  **hashcode,**as they maintain a contract, but at the same time we may or may not implement the comparable interface. If we don’t implement the comparable interface, then we won't be able to compare objects, and thus we can’t do any sorting on Java Objects.

To enable sorting, either an Object has to implement the comparable interface, or we can use the comparator interface

The Comparable and Comparator interfaces use the **compareTo**and **compare**methods, respectively, and they return int.

If int's value is:

* **Positive**:  Current instance is greater than Other.
* **Zero**:  Two instances are equal.
* **Negative**: Current instance is smaller than Other.

Now, if **Comparable**returns Zero, it means two objects are the same by comparison. If two objects are the same by using the **equals method**, it returns true.

**So the question is, if the Comparable interface says that two objects are the same, and equals method says they are unequal, then what problem do we face?**

Let’s, examine it by using code:

package com.example.contract;

public class Glass implements Comparable<Glass>{

public enum Size{

BIG(3),MEDIUM(2),SMALL(1);

private int size;

Size(int size){this.size=size; }

public int getSize(){return size;}

};

private String material;

private Size size;

public Glass(Size size,String material) {

this.size=size;

this.material=material; }

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + ((material == null) ? 0 : material . hashCode());

result = prime \* result + ((size == null) ? 0 : size.hashCode());

return result; }

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Glass other = (Glass) obj;

if (material == null) {

if (other.material != null)

return false;

} else if (!material.equals(other.material))

return false;

if (size != other.size)

return false;

return true;

}

@Override

public int compareTo(Glass o) {

if(this.size.getSize() == o.size.getSize())

{

return 0;

}

else if(this.size.getSize() > o.size.getSize())

{

return 1;

}

else

{

return -1;

}

}

@Override

public String toString() {

return "Glass [material=" + material + ", size=" + size + "]";

}

}

package com.example.contract;

import java.util.HashSet;

import java.util.Set;

import java.util.TreeSet;

public class ContractBreaker {

public static void main(String[] args) {

Glass plastic = new Glass(Glass.Size.BIG,"Plastic");

Glass glass = new Glass(Glass.Size.BIG,"glass");

Set<Glass> set = new HashSet<Glass>();

set.add(plastic);

set.add(glass);

System.out.println(set);

Set<Glass> treeSet = new TreeSet<Glass>();

treeSet.add(plastic);

treeSet.add(glass);

System.out.println(treeSet);

}

}

Oops, While HashSets treats both Glass classes as unequal, TreeSet treats them as equal.

So while coding, if we change HashSet to TreeSet or vice versa, weird results can occur.

The root of the problem is, when we implement the equals method in a Glass class, we consider two properties of Glass, i.e. material and size, so if both are equal, then we say both Glasses are meaningfully equivalent.

But with the CompareTo method we only consider size. If both have the same size, we treat them as equal.

We made a mistake here. **HashMap, ArrayList,**and**HashSet** add elements based on the **equals** method, so when we use HashSet, it treats two objects as different objects, as their materials are different.

But **TreeMap** and **TreeSet** are ordered and use the **compareTo** method, so TreeSet treats them as the same Object.

So by looking at the above problem, we can see that there is a hidden contract between equals and comparable. If you do not maintain the contract no catastrophic failure will happen but when you use it with HasSet or TreeSet it can produce wired result. So You must be aware of that. It is not a Strong contract like equals and hashcode but tries to maintain It.

**If two objects are considered equal by the equals operation, then try to make those objects must equal by the Comparable or Comparator test.**

In Java, the BigDecimal class breaks this contract, so when use BigDecimal, please remember the scenario stated above.

<https://dzone.com/articles/the-hidden-contract-between-equals-and-comparable>