Comparison and Bounds

26 October 2025 12:05

Comparable<T> and Comparator<T>, which are used to support comparison on elements.

Comparable<T>

The interface Comparable<T> declares a single instance method for comparing one object with another:

```
interface Comparable<T>{
   public int compareTo(T o);
}
```

The compareTo method returns an integer value that is -ve, 0 or positive depending upon whether the receiver-this object-is less than, equal to, or greater than the argument. When a class implements Comparable, the ordering specified by this interface is called the natural ordering for that class.

Typically, an object belonging to a class can only be compared with an object belonging to the same class. For instance, Integer implements Comparable<Integer>:

```
Integer int0 = 0;
Integer int1 = 1;
int0
<object reference>   (Integer)
int1
<object reference>   (Integer)
System.out.print(int0.compareTo(int1));
int1 = -10
<object reference>   (Integer)
System.out.print("\n"+int0.compareTo(int1));


assert int0.compareTo(int1) < 0;
Exception: java.lang.AssertionError (null)
assert int0.compareTo(int1) > 0;
```

Junit.

The comparison returns a negative number since zero precedes one under numerical ordering. Similarly String implements Comparable<String>:

```
String s1 = "zero";
String s2 = "one";
assert s1.compareTo(s2) < 0;
Exception: java.lang.AssertionError (null)
assert s1.compareTo(s2) > 0;
```

returns tve number, since 'zero' follows 'one' under alphabetic ordering.

```
int1
<object reference>   (Integer)
s1
"zero"   (String)
assert int1.compareTo(s1) < 0;
Error: incompatible types: java.lang.String cannot be converted to java.lang.Integer
```

You can compare an integer with an integer, or a string with a string, but attempting to compare an integer with a string is a compile-time error.

But comparison is not supported between arbitrary numerical types:

```
Number m = Integer.valueOf(2);
Number n = Double.valueOf(3.14);
assert m.compareTo(n) < 0;
```

Here the comparison is illegal, because the Number class does not implement

```
Number m = Integer.valueOf(2);
Number n = Double.valueOf(3.14);
assert m.compareTo(n) < 0;
Error: cannot find symbol
  symbol:   method compareTo(java.lang.Number)
  location: variable m of type java.lang.Number
```

Here the comparison is illegal, because the Number class does not implement the Comparable interface.

Contract for Comparable: The contract for the Comparable<T> Interface specifies 3 properties. The properties are defined using the sine function, which is defined such as sgn(x) returns -1, 0, or 1, depending on whether x is negative, zero or positive.

First comparison is anti-symmetric. Reversing the order of argument reverses the result:

```
sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
```

This generalizes the property for numbers: X < Y If and only if Y > X. It is also required that x.compareTo(y) raises an exception if and only if y.compareTo(x) raises an exception.

Taking x and y to be same gives us sgn(x.compareTo(x)) == -(sgn(x.compareTo(x)). It follows that

```
x.compareTo(x) == 0
```

So comparison is reflexive-that is, every value compares as the same as itself.

Second, comparison is transitive. If one value is smaller than a second and the 2nd is smaller than a 3rd, then the 1st is smaller than the 3rd.

```
if x.compareTo(y) < 0 and y.compareTo(z) < 0 then x.compareTo(z) < 0
```

This generalizes the property of numbers: if x < y and y < z then x < z.

Third, Comparison is congruence. If two values compare as the same, then they compare the same way with any 3rd value.

```
if x.compareTo(y) == 0 then sgn(x.compareTo(z)) == sgn(y.compareTo(z))
```

This generalizes the property for numbers: if x == y then x < z iff y < z. Presumably, it is also required that if x.compareTo(y) == 0 then x.compareTo(z) raises an exception iff y.compareTo(z) raises an exception, although this is not explicitly stated.

Consistent with Equals: The contract for Comparable strongly recommends that the compare to method should be consistent with equals. That tells the two objects. Should satisfy the equals method if and only if they compare as the same:

```
x.equals(y)    if and only if x.compareTo(y) == 0
```

Notice that this is a recommendation, not a mandatory part of the contract. But in fact it is the normal case: Most classes having a natural ordering, do comply with this recommendation. If you are. But if you are designing a class that implements Comparable, you should not ignore it without good reason.
The best known example in the platform library that contravenes this recommendation is java.math.BigDecimal: Two instances of this class representing the same value but with different precisions. For example 4.0 and 4.00 will compare as the same but not satisfy the equals method. One consequence of this design choice is that values of such class cannot be reliably stored in an internally ordered collection like NavigableSet or NavigableMap.

Maximum of a Collection

Here we have used the Comparable

Maximum of a Collection

here we have used the Comparable interface to find the maximum element in a Collection.

```
public static <T extends Comparable<T>> T max(Collection<T> coll){
    Iterator<? extends T> i = coll.iterator();
    T candidate = i.next();
    while(i.hasNext()){
        T next = i.next();
        if(next.compareTo(candidate) > 0){
            candidate = next;
        }
    }
    return candidate;
}
```

This code to find the maximum element in a non-empty collection, from the class Collections.

We first saw generic methods that declare new type variables in the signature. For instance, the method as list takes an array of type E[] And returns a result of type List<E> and does not so for any type E. Here we have the generic method that declares a bound on the type variable. The method Max takes a collection of type Collection <T > and returns a T and it does this for any type T such that T is a subtype of Comparable <T>. The highlighted phrase in angle brackets at the beginning of the type signature declares the type variable T, and we say that T is bounded by Comparable<T>. As the wildcard bound for type variable are always indicated by the keyword extends even when the bound is an interface rather than a class as is the case here. Unlike wild cards, type variable can only be bounded using extends, not super.

In this case the bound is recursive. In that the bound on T itself depends upon T. It is even possible to have mutually recursive bounds such as:

```
<T extends C<T, U>, U extends D<T, U>>
```

An example of mutually recursive bounds appear.

The method body first obtains an iterator over the collection, then calls the next method to select the first element as a candidate for the maximum. The specification of this method allows it to throw a NoSuchEleementException if it is supplied with an empty collection. It then compares the candidate with each element in the collection. Setting the candidate to the element when the element is larger.

```
jshell> import  MaximumAndMinimum.*;

jshell> List<Integer> ints = Arrays.asList(10,-10, 20, -20, 19)
ints ==> [10, -10, 20, -20, 19]

jshell> MaximumOfCollection.max(ints);
$3 ==> 20

jshell> List<String> strs = Arrays.asList("ten", "minus ten", "twenty", "minus twenty",
"nineteen");
strs ==> [ten, minus ten, twenty, minus twenty, nineteen]

jshell> MaximumOfCollection.max(strs);
$5 ==> "twenty"

jshell> List<Number> nums = Arrays.asList(1, 1.1, 2, 3.4, 5)
nums ==> [1, 1.1, 2, 3.4, 5]

jshell> MaximumOfCollection.max(nums);
|  Error:
|  method max in class MaximumAndMinimum.MaximumOfCollection cannot be applied to given types;
|    required: java.util.Collection<T>
|    found:    java.util.List<java.lang.Number>
```

Compile time error.

```
|    reason: inference variable T has incompatible bounds
|      equality constraints: java.lang.Number
|      upper bounds: java.lang.Comparable<T>
|  MaximumOfCollection.max(nums);
|  ^---------------------^

jshell>
```

Declarations for methods should be as general as possible to maximize utility. If you can replace a type parameter with a wild card, then you should do so. So we can improve the declaration of max() by replacing:

```
public static <T extends Comparable<T>> T max(Collection<T> coll)
```

With:

```
<T extends Comparable<? super T>> T max(Collection <? extends T> coll)
```

Following the get and put principle, we use extends with Collection because we get values of type T from the collection and we use super with Comparable because we put values of type T into **compareTo** method.

Questions to practice

1. Create a Java class Student that implements Comparable<Student>. The students should be compared first by CGPA (higher is better) and then by name (alphabetical order) if CGPA is the same. Write a main program to sort a list of students.

2. Write a generic method that returns the minimum element from a Collection<T>, where T implements Comparable<T>. Demonstrate it with a list of integers and a list of strings.

3. Implement a class Book with fields for title and price. Write a Comparator<Book> that sorts books by price in ascending order, and then by title in case of a tie. Test the comparator by sorting a list of books.

4. Given a list of Person objects (name, age), use Collections.sort() with both natural ordering (Comparable) and a custom comparator (by age descending). Compare the results.

5. What happens if you try to compare two objects of different classes (e.g., an Integer and a String) using Comparable? Write code to demonstrate the type safety enforced by Java.

6. Write a generic method that finds the maximum in a collection, with this signature:public static <T extends Comparable<T>> T max(Collection<T> coll). Explain and test its use.

7. Modify the max method to allow for more general types by using wildcard bounds: <T extends Comparable<? super T>>. Show with a code example how this improves flexibility.

8. Create a class Fraction (numerator, denominator) that implements Comparable<Fraction> based on the numeric value of the fraction. Write code to sort a list of fractions.

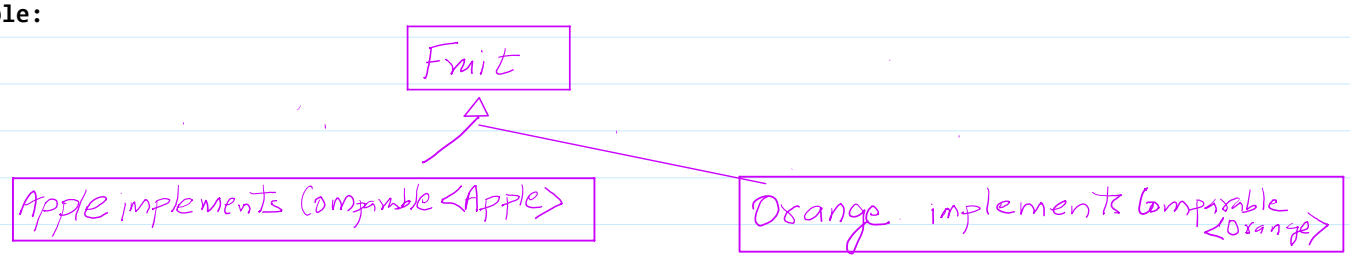
9. Explain (with code) the difference between implementing sorting with Comparable inside a class and sorting with a separate Comparator class. When would you use each approach?

10. Given two classes, Employee and Manager (where Manager extends Employee), make Employee implement Comparable<Employee>. Show an example that highlights problems with comparison if subclass-specific fields are involved.

```
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

The highlight bound is there for backward compatibility.
The Comparable<T> Interface gives fine control over what can and cannot be compared.

Example:




```
// Here in this example. It prohibits comparison of apples with oranges.
 class Fruit{…}
 class Apple extends Fruit implements Comparable<Apple>{…}
 class Orange extends Fruit implements Comparable<Orange>{…}
```

Each fruit has a name and a size. And two fruits are equal if they have the same name and the same size. Since we have overridden equals, we have also overridden hash code to ensure that objects have the same hash code. **Apple implements Comparable<Apple>,** it is clear that you can compare Apple with Apples, but not with Oranges.

```
List<Apple> appleList = new ArrayList<>();    //Ok
List<Orange> orangeList = new ArrayList<>();  //Ok
List<Fruit> mixedFruit = new ArrayList<>();   //Compile time error
```

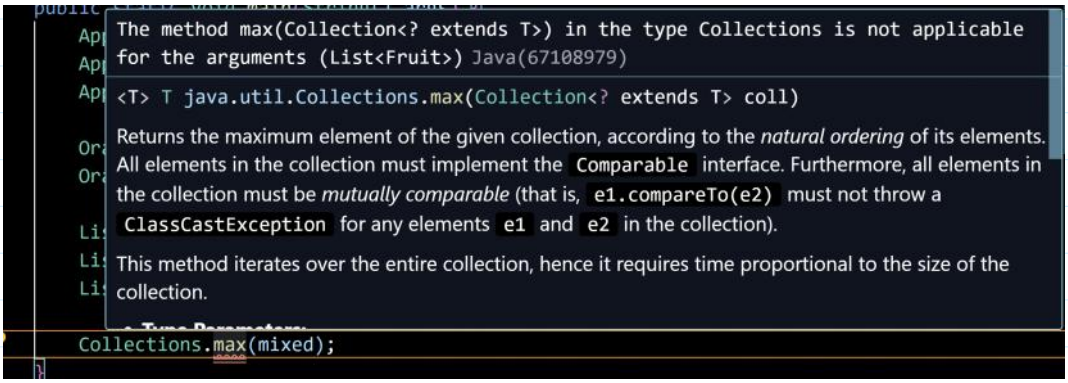
We may find the maximum of the first two list, but attempting to find the maximum of the mixedFruit it signals an error at compile time.

```
 class Fruit implements Comparable<Fruit>{…}
 class Apple extends Fruit {…}
 class Orange extends Fruit{…}
```

As before, each fruit has a name and a size, and two fruits are equal if they have the same name and the same size. Now since **Fruit implements Comparable<Fruit>, a**ny two fruits may be compared by comparing their sizes. So the test code can find the maximum of all three list, including the one that mixes apple with oranges.

```
<T extends Comparable<? super T>> T max(Collection <? extends T> coll)
```

Why this wildcard is needed. If we want to compare two Oranges, we take T in the preceding code to be Orange.
→ class Orange extends Comparable <? super Orange>
And this is true both of the following hold:
class Orange extends Comparable<Fruit>




```
package MaximumAndMinimum;
```

```java
import java.util.Objects;
abstract public class Fruit {
    protected String name;
    protected int size;
    protected Fruit(String name, int size){
        this.name = name; this.size = size;
    }
    public boolean equals(Object o){
        if (o instanceof Fruit){
            Fruit that = (Fruit)o;
            return this.name.equals(that.name) && this.size == that.size;
        }
        else{
            return false;
        }
    }
    public int hashCode(){
        return Objects.hash(name, size);
    }
    protected int compareTo(Fruit that){
        return Integer.compare(this.size, that.size);
    }
}
```

```java
package MaximumAndMinimum;
public class Apple extends Fruit implements Comparable<Apple>{
    protected Apple(int size) {
        super("Apple", size);
    }
    @Override
    public int compareTo(Apple o) {
        return super.compareTo(o);
    }
}
```

```java
package MaximumAndMinimum;
public class Orange extends Fruit implements Comparable<Orange> {
    protected Orange(int size) {
        super("Orange", size);

    }
    @Override
    public int compareTo(Orange o) {
        return super.compareTo(o);
    }
}
```

```java
package MaximumAndMinimum;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class TestFruit {
    public static void main(String[] args) {
        Apple a1 = new Apple(1);
        Apple a2 = new Apple(2);
        Apple a3 = new Apple(3);
        Orange o1 = new Orange(4);
        Orange o2 = new Orange(5);
        List<Apple> apples = Arrays.asList(a1, a2, a3);
        List<Orange> oranges = Arrays.asList(o1, o2);
        List<Fruit> mixed = List.of(a1,a2, o1);
```

```
        Collections.max(mixed); //  compile-time error
    }
}
```

In the above code we Prohibiting comparison of Apple with Orange.

In the code below we Permitting comparison of Apple with Orange.

```
package MaximumAndMinimum;
import java.util.Objects;
abstract public class Fruit implements Comparable<Fruit> {
    protected String name;
    protected int size;
    protected Fruit(String name, int size){
        this.name = name; this.size = size;
    }
    public boolean equals(Object o){
        if (o instanceof Fruit){
            Fruit that = (Fruit)o;
            return this.name.equals(that.name) && this.size == that.size;
        }
        else{
            return false;
        }
    }
    public int hashCode(){
        return Objects.hash(name, size);
    }
    public int compareTo(Fruit that){
        return Integer.compare(this.size, that.size);
    }
}
```

```
package MaximumAndMinimum;
public class Apple extends Fruit {
    protected Apple(int size) {
        super("Apple", size);
    }
}
```

```
package MaximumAndMinimum;
public class Orange extends Fruit  {
    protected Orange(int size) {
        super("Orange", size);

    }
}
```

```
package MaximumAndMinimum;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class TestFruit {
    public static void main(String[] args) {
        Apple a1 = new Apple(1);
        Apple a2 = new Apple(2);
        Apple a3 = new Apple(3);
```

```
        Orange o1 = new Orange(4);
        Orange o2 = new Orange(5);
        List<Apple> apples = Arrays.asList(a1, a2, a3);
        List<Orange> oranges = Arrays.asList(o1, o2);
        List<Fruit> mixed = List.of(a1,a2, o1);
        System.out.println("Maximum in mixed fruit: "+Collections.max(mixed).size);
        System.out.println("Maximum in apples: " + Collections.max(apples).size);
        System.out.println("Maximum in oranges: " + Collections.max(oranges).size);
    }
}
```

Output:

```
Maximum in mixed fruit: 4
Maximum in apples: 3
Maximum in oranges: 5
```