Comparable, Comparator and Sorting

**Overview**

This tutorial will focus on two interfaces, the Comparable and Comparator interfaces, which can be used for comparing elements and thus sorting routines.

**Reasoning**

An open question you will often be faced with is how to compare data elements. There are built in operators for comparing numerical elements in Java. For example:

int a = 10;  
int b = 20;

if (a < b)  
{  
    //do something  
}

But these operators don't work for other popular types. For example, the following code sample will not compile:

String title = "Mrs";  
String name = "Smith";

if (title < name)  
{  
    //do something  
}

This may be surprising as you can order words. A typical ordering that is chosen will be the lexicographic order (or dictionary ordering). With this ordering we compare individual letters in a word sequentially. If two words share the the same letters at a position such as:

aura  
apple

We would then compare the character at the next position (from left to right). Once a letter differs we can then order the words alphabetically. This should feel very familiar as this is how we organize words in an actual dictionary (go figure). So we could sort the following words lexicographically:

|  |  |
| --- | --- |
| **Unsorted** | **Sorted** |
| car canopy carp cannabis | cannabis canopy car carp |

Yet, how would order an objects such as an address. Addresses have street numbers and names, cities, zip codes and states! Should we order addresses by number and name? Should we order them by zip? Or perhaps we could use several values, by first ordering elements by zip and then by number and name. For example:

|  |  |
| --- | --- |
| **Unsorted** | **Sorted** |
| 1000 6th St. Seattle, WA 98999 22 Pearl St. Seattle, WA 98000 310 Tan Ave. Seattle, WA 98999 5000 1st St. Seattle WA, 98999 101 Evergreen Way Seattle WA, 98000 | 101 Evergreen Way Seattle WA, 98000 22 Pearl St. Seattle, WA 98000 5000 1st St. Seattle WA, 98999 1000 6th St. Seattle, WA 98999 310 Tan Ave. Seattle, WA 98999 |

Note: Notice how the sorted results are first ordered by zip, then any elements with shared zip codes are then ordered lexicographically by street name.

**Comparable Interface**

This problem becomes more difficult as we try to order an arbitrary object. How can we write a routine that requires the comparison of elements (such as a sorting routine) if we don't have standard way to compare arbitrary objects. The Comparable interface provides this standard mechanism for comparisons.

The Comparable interface has a single method:

public int compareTo(Object other)

You can view the interface API here: [link (Links to an external site.)](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html). You should notice that there are many classes in the basic Java libraries that use this interface. That is, this interface is very popular to implement as it allows objects to be sorted! You can see a partial list of classes below:

At this point you are probably wondering why this method returns an integer value. Moreover, how is this method used to compare elements? Each method defined in an interface is expected to be defined by an author of a class that uses that interface. Yet, an interface typically have semantics (expected behavior) tied to a method. So while a method defined in an interface will be implemented often by a different author in a separate class, there are still some expectations on how an author implements the interface method.

The semantics of the compareTo() method are defined as so:

if we have two objects a & b that both use the Comparable interface, then given a.compareTo(b), the return value of compareTo() should be

|  |  |
| --- | --- |
| **Relationship** | **Return value** |
| a < b | negative number |
| a > b | positive number |
| a == b | zero |

**Example Implementation**

Suppose we have the following class, which is used to represent post stamps:

public class PostageStamp  
{  
    private double value;  
    private String image;  
    private int yearRelease  
  
    //constructors...  
  
    //methods...    
}

You can begin ordering postage stamps by first add the Comparable interface to your class:

public class PostageStamp implements Comparable  
{  
    private double value;  
    private String image;  
    private int yearRelease;  
  
    //constructors...  
  
    public int compareTo(Object other)  
    {  
        //do something...  
    }  
}

Assuming I wanted to order postage stamps by the year they are released, I could implement compareTo() like so:

public int compareTo(Object other)  
{  
    //make sure that other is a postage stamp  
    if (!(other instanceof PostageStamp))  
    {  
        throw new IllegalStateException("Cannot compare postage stamps with other types of objects");  
    }  
  
    PostageStamp otherStamp = (PostageStamp)other;  
    if (this.yearRelease < otherStamp.yearRelease)  
    {  
        return -1;  
    }  
    else if (this.yearRelease > otherStamp.yearRelease)  
    {  
        return 1;  
    }  
    else //otherwise they are equal  
    {  
        return 0;  
    }  
}

You could then test this method using a driver class. An example is given below:

public class StampDriver  
{  
    public static void main(String[] args)  
    {  
        PostageStamp stampA = new PostageStamp(0.10, "georgeWashington.png", 1990);  
        PostageStamp stampB = new PostageStamp(0.50, "jefferson.png", 2010);  
  
        if (stampA.compareTo(stampB) < 0)  
        {  
            System.out.println("StampA was released before StampB!");  
        }  
    }  
}

**Using Comparable with Other Fields**

Alternatively, I could order postage stamps by value:

public int compareTo(Object other)  
{  
    //make sure that other is a postage stamp  
    if (!(other instanceof PostageStamp))  
    {  
        throw new IllegalStateException("Cannot compare postage stamps with other types of objects");  
    }  
  
    PostageStamp otherStamp = (PostageStamp)other;  
    if (this.value < otherStamp.value)  
    {  
        return -1;  
    }  
    else if (this.value > otherStamp.value)  
    {  
        return 1;  
    }  
    else //otherwise they are equal  
    {  
        return 0;  
    }  
}

Or even by image. To order strings you must use the lexicographic ordering. Luckily for us, this has already been implemented in the String class, as String objects are already comparable! The compareTo() method in the String class follows the semantics described above:

public int compareTo(Object other)  
{  
    //make sure that other is a postage stamp  
    if (!(other instanceof PostageStamp))  
    {  
        throw new IllegalStateException("Cannot compare postage stamps with other types of objects");  
    }  
  
    PostageStamp otherStamp = (PostageStamp)other;  
    return this.image.compareTo(otherStamp); //use the compareTo() method in the String class!  
}

Note that you can simplify some of the calculations above. You are only required some negative or positive numbers in semantics below. So comparing stamps by release year above can be re-written like so:

public int compareTo(Object other)  
{  
    //make sure that other is a postage stamp  
    if (!(other instanceof PostageStamp))  
    {  
        throw new IllegalStateException("Cannot compare postage stamps with other types of objects");  
    }  
  
    PostageStamp otherStamp = (PostageStamp)other;  
    return this.yearRelease - otherStamp.yearRelease;  
}

**Sorting with Comparable**

So why are working so hard to compare arbitrary object types? There are many routines and structures that require elements to be comparable. These include:

* Most sorting routines
* Binary search trees
* Heap structures

There are built-in routines for sorting in Java. These are found in the Arrays class (for sorting arrays) and the Collections class (for sorting collection objects). Whether you are using an array or collection to store elements that are passed to these routines, these elements must use the Comparable interface!

public class StampDriver  
{  
    public static void main(String[] args)  
    {  
        PostageStamp stampA = new PostageStamp(0.10, "georgeWashington.png", 1990);  
        PostageStamp stampB = new PostageStamp(0.50, "jefferson.png", 2010);  
        PostageStamp stampC = new PostageStamp(1.25, "wilson.png", 2015);  
  
        PostageStamp[] stamps = {stampA, stampB, stampC};  
  
        //sort our array (based on a comparable method given above)  
        **Arrays.sort(stamps);**  
  
        //print the sorted array  
        System.out.println(Arrays.toString(stamps));  
    }  
}

**Comparable<T> Interface**

If you have been coding alongside this tutorial, you have probably seen compiler warnings about using raw types. These warnings are in place because the Comparable interface is generic. If we give the interface a type, we can avoid calling compareTo() for any object. Instead we can use strong typing to verify the input to compareTo().

public class PostageStamp implements Comparable**<PostageStamp>**  
{  
    private double value;  
    private String image;  
    private int yearRelease;  
  
    //constructors...  
  
    public int compareTo(**PostageStamp** other)  
    {  
        return this.yearRelease - other.yearRelease;  
    }  
}

**Comparator Interface**

There is an alternative to the Comparable<T> interface, that allows you to create a separate class for ordering objects. The Comparator<T> interface has a compare(T first, T second) method that can be used for ordering. The semantics are similar to the Comparable<T> interface:

Suppose you have two objects a & b of the same type, and a Comparator object c. Then:

|  |  |
| --- | --- |
| **Relationship** | **Return value of c.compare(a, b)** |
| a < b | negative number |
| a > b | positive number |
| a == b | zero |

An example Comparator<T> class is given below:

public class StampComparator implements Comparator<Stamp>  
{  
    public int compare(Stamp first, Stamp second)  
    {  
        if (first.value < second.value)  
        {  
            return -1;  
        }  
        else if (first.value > second.value)  
        {  
            return 1;  
        }  
        else   
        {  
            return 0;  
        }  
    }  
}  
  
public class ComparatorDriver  
{  
    public static void main(String[] args)  
    {  
        PostageStamp stampA = new PostageStamp(0.10, "georgeWashington.png", 1990);  
        PostageStamp stampB = new PostageStamp(0.50, "jefferson.png", 2010);  
        PostageStamp stampC = new PostageStamp(1.25, "wilson.png", 2015);  
  
        //Comparator<T> objects can be use with sorting also!  
        **Comparator<Stamp> comparator = new Comparator<Stamp>();**  
  
        //sort our array (based on a comparable method given above)  
        Arrays.sort(stamps, **comparator**);  
  
        //print the sorted array  
        System.out.println(Arrays.toString(stamps));  
    }  
}

There are several benefits to using the Comparator<T> interface in lieu of Comparable<T>.

* You can create several Comparator<T> objects and use them separately in your code segments (ie. you can change how you sort elements using several stored objects).
* The Comparator<T> interface allows you to perform several sorts sequentially, for multi-sorting.

**Multi Sorting!**

To perform multiple sorts on an array or collection or elements can be done easily with the Comparable<T> interface. For example, I may want to order stamps first by release year and then by value:

public int compareTo(**PostageStamp** other)  
{  
    if(this.yearRelease < other.yearRelease)  
    {  
        return -1;  
    }  
    else if (this.yearRelease > other.yearRelease)  
    {  
        return 1;  
    }  
    else  
    {  
        //if the year is the same then order them by value  
        if (this.value < other.value)  
        {  
            return -1;  
        }  
        else if (this.value > other.value)  
        {  
            return 1;  
        }  
        else  
        {  
            return 0;  
        }  
    }  
}