CSCI 2120:
Software Design & Development II

UNIT2: Data management

Collections Framework & Generics

Comparator

Overview

1. Introduction
2. Comparator interface declaration
3. Comparator Methods
4. Comparator Examples
5. Comparable vs Comparator

Introduction

Comparator in Java is an interface whose task is to compare objects of a user-defined class when the class does not implement the Comparable interface.

In other words, Java Comparator is used when:

- Objects to be ordered do not have a natural ordering defined by Comparable interface.
- You want to order elements something different way other than their natural order.

In both cases, we can specify a `Comparator` object when you construct the set or map.

`Comparator` interface was introduced in Java 1.2 version. It is present in `java.util.Comparator<T>` package.

Comparator interface declaration

Comparator is a generic interface that can be declared in general form like this:

```
public interface Comparator<T>
```

Here, the parameter T represents the type of objects (or elements) that will be compared.

Methods of Java Comparator Interface

Comparator interface provides two methods that are as follows:

Methods of Java Comparator Interface

1. int compare(Object obj1, Object obj2):

It compares the first object obj1 with the second object obj2.

- It returns zero if objects are equal (i.e. `obj1 == obj2`).
- It returns a positive value if obj1 is greater than obj2 (i.e. `obj1 > obj2`).
- Otherwise, a negative value is returned.

Let's take an example to understand better. `Comparator<Country>` looks like this:

```
public interface Comparator<Country> {
   int compare(Country a, Country b);
}
```

Methods of Java Comparator Interface

1. int compare(Object obj1, Object obj2):

The call to `com.compare(a, b);` will return a negative value if `a` comes before `b`, 0 if `a` and `b` are equal, and a positive value if `a` is after `b`. Here, `com` is an object of a class that implements `Comparator<Country>`.

The `compare()` method can throw an exception named `ClassCastException` if types of objects are not compatible for comparison.

By overriding the `compare( )` method, we can change the way that objects are ordered. For example, to sort elements in reverse order, we can create a comparator object that reverses the outcome of a comparison.

Methods of Java Comparator Interface

2. boolean equals(Object obj):

The `equals()` method defined in the `Comparator` interface is overridden of the equals() method of the Object class. It is used to test whether an object equals invoking comparator.

Here, the parameter obj is the object to be tested for equality. If object obj and the invoking object both are Comparator objects and use the same ordering. the `equals()` method returns `true`. Otherwise, it returns `false`.

Overriding `equals( )` is not always necessary. When it is not necessary, there is no requirement to override Object's implementation.

Methods of Java Comparator Interface

2. boolean equals(Object obj):

The complete declaration of the Comparator class that is defined in java.util package is as follows:

```
public interface Comparator<T>{
   // An abstract method declared in the interface
   int compare(Object obj1, Object obj2);

   // Re-declaration of the equals() method of the Object class
   boolean equals(Object obj);
}
```

Examples of Sorting using Comparator

Example 1: Comparator - ascending sorting

1. Let’s take an example program where we will perform sorting of integer elements in ascending order using comparator. Look at the following source code to understand the power of custom comparator.

Example 1: Comparator - ascending sorting

```
import java.util.Comparator;
// To sort elements into ascending order.
class Ascend implements Comparator<Integer> {
   @Override
   public int compare(Integer i1, Integer i2) {
       return i1.compareTo(i2);
   }
   // No need to override equals method.
}
```

Example 1: Comparator - ascending sorting

```
import java.util.Comparator;
import java.util.TreeSet;
public class ComparatorTester1 {
   public static void main(String[] args) {
       // Create an object of Ascend class.
       Ascend as = new Ascend();

       // Create a tree set and pass the reference variable of Ascend class as a parameter.
       TreeSet<Integer> ts = new TreeSet<Integer>(as);

       // Adds elements into treeset.
       ts.add(25);
       ts.add(15);
       ts.add(30);
       ts.add(10);
       ts.add(40);
       ts.add(05);

       // Display the elements in ascending order.
       System.out.println("Sorted in Ascending order");

       for(Integer element : ts)
           System.out.print(element + " ");
       System.out.println();
   }
}
```

Example 1: Comparator - ascending sorting

Output:

```
Sorted in Ascending order
5 10 15 25 30 40
```

Explanation:

Look closely at the `Ascend` class, which implements `Comparator` and overrides `compare( )` method. As explained earlier, overriding `equals( )` method is needed here. Inside `compare( )` method, the `compareTo( )` compares the two integers.

Example 2: Comparator - descending sorting

2. Let’s write a program to sort elements in descending order using Comparator in Java. Look at the source code.

Example 2: Comparator - descending sorting

```
import java.util.Comparator;
// To sort elements into descending order.
class Descend implements Comparator<Integer>{
   @Override // Implement compare() method to reverse for the integer comparison.
   public int compare(Integer i1, Integer i2) {
       // For reverse comparison.
       return i2.compareTo(i1);
   }
}
```

Example 2: Comparator - descending sorting

```
import java.util.TreeSet;
public class ComparatorTester2 {
   public static void main(String[] args) {
       // Create an object of Descend class.
       Descend ds = new Descend();

       // Create a tree set and pass the reference variable of Descend class as a parameter.
       TreeSet<Integer> ts = new TreeSet<Integer>(ds);

       // Adds elements into tree set.
       ts.add(25);
       ts.add(15);
       ts.add(30);
       ts.add(10);
       ts.add(40);
       ts.add(05);

       // Display the elements in ascending order.
       System.out.println("Sorted in Descending order");

       for(Integer element : ts)
           System.out.print(element + " ");
       System.out.println();
   }
}
```

Example 2: Comparator - descending sorting

Output:

```
Sorted in Descending order
40 30 25 15 10 5
```

Example 3: Comparator - reverse string sorting

3. Let’s take one more example program similar to the above program where we will perform reverse string comparison.

Example 3: Comparator - reverse string sorting

```
import java.util.Comparator;
// To sort elements into descending order.
class RevStrComp implements Comparator<String> {
   @Override // Implements compare() method for reverse string comparison.
   public int compare(String str1, String str2) {
       // For reverse comparison.
       return str2.compareTo(str1);
   }
}
```

Example 3: Comparator - reverse string sorting

```
import java.util.TreeSet;
public class ComparatorTester3 {
   public static void main(String[] args) {
       // Create an object of RevStrComp class.
       RevStrComp rsc = new RevStrComp();

       // Create a tree set and pass reference variable of RevStrComp class as parameter.
       TreeSet<String> ts = new TreeSet<String>(rsc);

       // Adds elements into tree set.
       ts.add("Cat");
       ts.add("Elephant");
       ts.add("Lion");
       ts.add("Dog");
       ts.add("Tiger");
       ts.add("Horse");

       // Display the elements in ascending order.
       System.out.println("Sorted in reverse order");

       for(String element : ts)
           System.out.print(element + " ");
       System.out.println();
   }
}
```

Example 3: Comparator - reverse string sorting

Output:

```
Sorted in reverse order
Tiger Lion Horse Elephant Dog Cat
```

Example 4: Comparator - Sorting Array

4. Let’s take an example program where we will sort an array with group of integer elements (objects) using Comparator in java.

Here, we will accept array elements from the keyboard and sort them in ascending and descending order. Look at the following source code.

Example 4: Comparator - Sorting Array

```
import java.util.Comparator;

// To sort elements into ascending order.
class AscendArray implements Comparator<Integer> {
   @Override
   public int compare(Integer i1, Integer i2) {
       return i1.compareTo(i2);
   }
}

// To sort elements into descending order.
class DescendArray implements Comparator<Integer>{
   @Override
   public int compare(Integer i1, Integer i2) {
       // For reverse comparison.
       return i2.compareTo(i1);
   }
}
```

Example 4: Comparator - Sorting Array

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
public class ComparatorTester4 {
   public static void main(String[] args) throws NumberFormatException, IOException {

       InputStreamReader isr = new InputStreamReader(System.in); // Create object InputStreamReader to accept array elements from the keyboard.
       BufferedReader br = new BufferedReader(isr);              // Create object BufferedReader and pass the reference variable to its constructor.
       System.out.println("Enter count of elements:");
       int size = Integer.parseInt(br.readLine());
       Integer arr[] = new Integer[size];                       // Create an array to store Integer type elements or objects.

       for(int i = 0; i < size; i++){                           // Now convert int values into Integer objects and then pass to array to store them.
           System.out.println("Enter your number:");
           arr[i] = Integer.parseInt(br.readLine());
       }

       AscendArray as = new AscendArray();                       // Create an object of Ascend class.
       Arrays.sort(arr, as);                                     // Call sort() method to sort array elements in ascending order.
       System.out.println("\nSorted in Ascending order: ");      // Display the sorted array elements.
       display(arr);

       DescendArray ds = new DescendArray();                     // Create an object of Descend class.
       Arrays.sort(arr, ds);                                     // Call sort() method to sort array elements in descending order.
       System.out.println("\nSorted in Descending order: ");     // Display the sorted array elements.
       display(arr);
   }

   static void display(Integer arr[]){
       for(Integer element : arr){
           System.out.println(element + "\t");
       }
   }
}
```

Example 4: Comparator - Sorting Array

Output:

```
How many elements do you want to enter?
3
Enter your number:
100
Enter your number:
0
Enter your number:
1

Sorted in Ascending order:
0
1
100

Sorted in Descending order:
100
1
0
```

Comparable interface vs. Comparator interface

Both Comparable and Comparator interfaces are used to sort the collection or array of elements (objects). But there are the following differences in use. They are as follows:

Use of Comparable interface in Java

1. Comparable interface is used for the natural sorting of objects. For example, if we want to sort Employee class by employeeId is natural sorting.

2. Comparable is used to sort collection of elements based on only a single element (or logic) like name.

3. Comparable interface is used when we want to compare itself with another object.

Comparable interface vs. Comparator interface

Both Comparable and Comparator interfaces are used to sort the collection or array of elements (objects). But there are the following differences in use. They are as follows:

Use of Comparator interface in Java

1. Comparator interface is used when we want to perform custom sorting on the collection of elements. It gives exact control over the ordering. For example, if we want to sort Employee class by name and age, it will be custom sorting.

2. Comparator is used to sort collection of elements based on multiple elements (or logics)like name, age, class, etc.

3. Comparator interface is used when we want to compare two different objects.

END