**Comparable vs. Comparator**

|  |  |
| --- | --- |
| 1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of **single** element such as id or name or price etc. | Comparator provides **multiple sorting sequence**. Ex. other words, we can sort the collection on the basis of **multiple** elements such as id, name and price etc. |
| 2) Comparable ***affects* the original class** i.e. actual class is modified. | Comparator ***doesn't* affect the original class**  i.e. actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is found in **java.lang** package. | Comparator is found in **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by  **Collections.sort(List,Comparator)** method or by the  **Arrays.sort(T[],Comparator < ? super T > c)** for sorting a *user-defined* object array. |

**Understanding deeper Arrays.sort() – with a Comparator**

ref: [http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html - sort(T[], java.util.Comparator)](http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html#sort(T[],%20java.util.Comparator))

Arrays.sort(T[], Comparator < ? super T > c) is a **method** for sorting user-defined object array.

1. How to Use Arrays.sort(): A Simple Example

By reading the following example, you can quickly get an idea of how to use this method correctly. A Comparator is defined for comparing Dogs by size and then the Comparator is used as a parameter for the sort method.

|  |
| --- |
| **import** java.util.Arrays;  **import** java.util.Comparator;    **class** Dog{  **int** size;  **public** Dog(**int** s){  size = s;  }  }    **class** DogSizeComparator **implements** Comparator<Dog>{    @Override  **public** **int** compare(Dog o1, Dog o2) {  **return** o1.size - o2.size;  }  }    **public** **class** ArraySort {    **public** **static** **void** main(String[] args) {  Dog d1 = **new** Dog(2);  Dog d2 = **new** Dog(1);  Dog d3 = **new** Dog(3);    Dog[] dogArray = {d1, d2, d3};  printDogs(dogArray);    Arrays.sort(dogArray, **new** DogSizeComparator());  printDogs(dogArray);  }    **public** **static** **void** printDogs(Dog[] dogs){  **for**(Dog d: dogs)  System.out.print(d.size + " " );    System.out.println();  }  } |

Output:

2 1 3

1 2 3

**Compator interface debunked**

**The Comparator interface.** Compares values of two objects. This is implemented as part of the *Comparator<T>* interface, and the typical use is to define one or more small utility classes that implement this, to pass to methods such as sort() or for use by sorting data structures such as TreeMap and TreeSet. You might want to create a Comparator object for the following.

1. **Multiple comparisons.** To provide several different ways to sort something. For example, you might want to sort a Person class by name, ID, age, height, ... You would define a Comparator for each of these to pass to the sort() method.
2. **System class.** To provide comparison methods for classes that you have no control over. For example, you could define a Comparator for Strings that compared them by length.
3. **Strategy pattern.** To implement a *strategy* pattern, which is a situation where you want to represent an *algorithm* as an object that you can pass as a parameter, save in a data structure, etc.

*If your class objects have one natural sorting order, you may not need this.*

**Compare() method overriding debunked**

***format*: int compare(**[**Object**](eclipse-javadoc:%E2%98%82=Lab3BankComparator/src%3C%7BsexComparator.java%E2%98%83sexComparator~compare~QBankRecords;~QBankRecords;%E2%98%82BankRecords) **o1, Object o2)**

Compares its *two* arguments for **order** where…

**o1** the first object to be compared.

**o2** the second object to be compared.

**Returns:**

a negative integer (-1), zero (0), or a positive integer (1) as the first argument is less than, equal to, or greater than the second.

2. The Strategy Pattern Used in Arrays.sort()

Our example used thus far is a perfect example of Strategy (or policy) pattern, it is worth to mention here why strategy pattern is good for this situation. In brief, Strategy pattern enables *different* algorithms to be selected at run-time. In this case, by passing a different **Comparator**, different algorithms can get selected. Based on the example above and now assuming you have *another* Comparator which compares Dogs by weight instead of just by size, you can simply create a new Comparator like the following.

|  |
| --- |
| **class** Dog{  **int** size;  **int** weight;    **public** Dog(**int** s, **int** w){  size = s;  weight = w;  }  }  **class** DogSizeComparator **implements** Comparator<Dog>{    @Override  **public** **int** compare(Dog o1, Dog o2) {  **return** o1.size - o2.size;  }  }  **class** DogWeightComparator **implements** Comparator<Dog>{    @Override  **public** **int** compare(Dog o1, Dog o2) {  **return** o1.weight - o2.weight;  }  }  **public** **class** ArraySort {    **public** **static** **void** main(String[] args) {  Dog d1 = **new** Dog(2, 50);  Dog d2 = **new** Dog(1, 30);  Dog d3 = **new** Dog(3, 40);    Dog[] dogArray = {d1, d2, d3};  printDogs(dogArray);    Arrays.sort(dogArray, **new** DogSizeComparator());  printDogs(dogArray);    Arrays.sort(dogArray, **new** DogWeightComparator());  printDogs(dogArray);  }    **public** **static** **void** printDogs(Dog[] dogs){  **for**(Dog d: dogs)  System.out.print("size="+d.size + " weight=" + d.weight + " ");    System.out.println();  }  } |

size=2 weight=50 size=1 weight=30 size=3 weight=40

size=1 weight=30 size=2 weight=50 size=3 weight=40

size=1 weight=30 size=3 weight=40 size=2 weight=50

As we can see, Comparator is just an interface. *Any* Comparator that implements this interface can be used during run-time. This is the key idea of Strategy design pattern.

3. Why Use "super"?

It is straightforward if “Comparator < T > c” is the parameter, but the second parameter is “Comparator< ? super T > c”. < ? super T > means the type can be T or its super types. Why it allows super types? The answer is: This approach allows using same comparator for

*all* sub classes. This is almost obvious in the following example.

|  |
| --- |
| **import** java.util.Arrays;  **import java.util.Comparator;**    **class** Animal{  **int** size;  }    **class** Dog **extends** Animal{  **public** Dog(**int** s){  size = s;  }  }    **class** Cat **extends** Animal{  **public** Cat(**int** s){  size = s;  }  }    **class** AnimalSizeComparator **implements** Comparator<Animal>{    @Override  **public** **int** compare(Animal o1, Animal o2) {  **return** o1.size - o2.size;  }  *//in this way, all sub classes of Animal can use this comparator.*  }    **public** **class** ArraySort {    **public** **static** **void** main(String[] args) {  Dog d1 = **new** Dog(2);  Dog d2 = **new** Dog(1);  Dog d3 = **new** Dog(3);    Dog[] dogArray = {d1, d2, d3};  printDogs(dogArray);    Arrays.sort(dogArray, **new** AnimalSizeComparator());  printDogs(dogArray);    System.out.println();    *//when you have an array of Cat, same Comparator can be used.*  Cat c1 = **new** Cat(2);  Cat c2 = **new** Cat(1);  Cat c3 = **new** Cat(3);    Cat[] catArray = {c1, c2, c3};  printDogs(catArray);    Arrays.sort(catArray, **new** AnimalSizeComparator());  printDogs(catArray);  }    **public** **static** **void** printDogs(Animal[] animals){  **for**(Animal a: animals)  System.out.print("size="+a.size + " ");  System.out.println();  }  } |

size=2 size=1 size=3

size=1 size=2 size=3

size=2 size=1 size=3

size=1 size=2 size=3

4. Summary- the takeaway messages from Arrays.sort():

To summarize

1. Use of generic - super
2. Use of strategy pattern
3. Use of merge sort - nlog(n) time complexity
4. Use of Java.util.Collections.sort(List < T > list, Comparator < ? super T > c) has similar idea with Arrays.sort.