<https://www.javatpoint.com/Comparable-interface-in-collection-framework>

Java Comparable interface

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

compareTo(Object obj) method

**public int compareTo(Object obj):** It is used to compare the current object with the specified object. It returns

* positive integer, if the current object is greater than the specified object.
* negative integer, if the current object is less than the specified object.
* zero, if the current object is equal to the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

Collections class

**Collections** class provides static methods for sorting the elements of collections. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

Method of Collections class for sorting List elements

**public void sort(List list):** It is used to sort the elements of List. List elements must be of the Comparable type.

#### **Note: String class and Wrapper classes implement the Comparable interface by default. So if you store the objects of string or wrapper classes in a list, set or map, it will be Comparable by default.**

## **Java Comparable Example**

Let's see the example of the Comparable interface that sorts the list elements on the basis of age.

*File: Student.java*

1. **class** Student **implements** Comparable<Student>{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
11. **public** **int** compareTo(Student st){
12. **if**(age==st.age)
13. **return** 0;
14. **else** **if**(age>st.age)
15. **return** 1;
16. **else**
17. **return** -1;
18. }
19. }

*File: TestSort1.java*

1. **import** java.util.\*;
2. **public** **class** TestSort1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<Student> al=**new** ArrayList<Student>();
5. al.add(**new** Student(101,"Vijay",23));
6. al.add(**new** Student(106,"Ajay",27));
7. al.add(**new** Student(105,"Jai",21));
9. Collections.sort(al);
10. **for**(Student st:al){
11. System.out.println(st.rollno+" "+st.name+" "+st.age);
12. }
13. }
14. }

105 Jai 21

101 Vijay 23

106 Ajay 27

## **Java Comparable Example: reverse order**

Let's see the same example of the Comparable interface that sorts the list elements on the basis of age in reverse order.

*File: Student.java*

1. **class** Student **implements** Comparable<Student>{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
11. **public** **int** compareTo(Student st){
12. **if**(age==st.age)
13. **return** 0;
14. **else** **if**(age<st.age)
15. **return** 1;
16. **else**
17. **return** -1;
18. }
19. }

*File: TestSort2.java*

1. **import** java.util.\*;
2. **public** **class** TestSort2{
3. **public** **static** **void** main(String args[]){
4. ArrayList<Student> al=**new** ArrayList<Student>();
5. al.add(**new** Student(101,"Vijay",23));
6. al.add(**new** Student(106,"Ajay",27));
7. al.add(**new** Student(105,"Jai",21));
9. Collections.sort(al);
10. **for**(Student st:al){
11. System.out.println(st.rollno+" "+st.name+" "+st.age);
12. }
13. }
14. }

106 Ajay 27

101 Vijay 23

105 Jai 21

# Lifecycle and States of a Thread in Java

A [thread](http://www.geeksforgeeks.org/multithreading-in-java/) in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated
7. **public void run():**is used to perform action for a thread.
8. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread.
9. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
10. **public void join():**waits for a thread to die.
11. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds.
12. **public int getPriority():**returns the priority of the thread.
13. **public int setPriority(int priority):**changes the priority of the thread.
14. **public String getName():**returns the name of the thread.
15. **public void setName(String name):**changes the name of the thread.
16. **public Thread currentThread():**returns the reference of currently executing thread.
17. **public int getId():**returns the id of the thread.
18. **public Thread.State getState():**returns the state of the thread.
19. **public boolean isAlive():**tests if the thread is alive.
20. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute.
21. **public void suspend():**is used to suspend the thread(depricated).
22. **public void resume():**is used to resume the suspended thread(depricated).
23. **public void stop():**is used to stop the thread(depricated).
24. **public boolean isDaemon():**tests if the thread is a daemon thread.
25. **public void setDaemon(boolean b):**marks the thread as daemon or user thread.
26. **public void interrupt():**interrupts the thread.
27. **public boolean isInterrupted():**tests if the thread has been interrupted.
28. **public static boolean interrupted():**tests if the current thread has been interrupted.

**Explanation:**When a new thread is created, the thread is in the NEW state. When .start() method is called on a thread, the thread scheduler moves it to Runnable state. Whenever join() method is called on a thread instance, the current thread executing that statement will wait for this thread to move to Terminated state. So, before the final statement is printed on the console, the program calls join() on thread2 making the thread1 wait while thread2 completes its execution and is moved to Terminated state. thread1 goes to Waiting state because it is waiting for thread2 to complete it’s execution as it has called join on thread2.