# Collections

## Tree Set

A TreeSet in Java is an implementation of the SortedSet interface, which extends the Set interface. It is part of the Java Collections Framework and is used to store elements in a sorted order. Unlike HashSet, which uses a hash table for storage, TreeSet uses a self-balancing binary search tree (specifically, a red-black tree) to maintain the elements in sorted order.

## NavigableSet

A NavigableSet is a subinterface of the SortedSet interface in the Java Collections Framework. It extends the functionalities of a SortedSet by providing navigation methods for accessing elements based on their relative order. NavigableSet was introduced in Java 6.

1. Navigation Methods:
   1. NavigableSet provides methods for navigating the set in both ascending and descending order. For example, lower, floor, ceiling, and higher methods return elements related to a specified element based on their order.
2. Range Views:
   1. NavigableSet offers methods for obtaining sub-sets or views of the set within a specified range. The subSet, headSet, and tailSet methods allow you to create subsets of the original set.
3. Polling:
   1. The pollFirst and pollLast methods remove and return the first and last elements in the set, respectively.

System.out.println("Lower(25): " + navigableSet.lower(25));

The lower method returns the greatest element in the set strictly less than the given element (25 in this case).

System.out.println("Floor(25): " + navigableSet.floor(25));

The floor method returns the greatest element in the set less than or equal to the given element.

System.out.println("Ceiling(25): " + navigableSet.ceiling(25));

The ceiling method returns the smallest element in the set greater than or equal to the given element.

System.out.println("Higher(25): " + navigableSet.higher(25));

The higher method returns the smallest element in the set strictly greater than the given element.

# Multithreading

## Callable:

Java offers two ways for creating a thread, i.e., by extending the Thread class and by creating a thread with a Runnable. There is a drawback of creating a thread with the Runnable interface, i.e., we cannot make a thread return result when it terminates, i.e., when the run() completes. In order to overcome this drawback, Java provides the Callable interface.

These are the two main differences between the Callable and Runnable methods:

The run() method is used for implementing the Runnable, whereas the call() method is used for implementing the Callable. The run() method doesn't return anything, whereas the call() method returns a result of completion.

The call() method can throw an exception, whereas the run() method cannot.

Note: We can create a thread by using Runnable, not by using Callable.

## Future:

In the main() method, the returned value of the call() method(after completion) should be stored in an object to know about the result that the thread returns. In order to store the returned data in the main() method, we use a Future object.

The future is one of the ways through which we can keep track of the progress and result from other threads. We need to override the five methods for implementing the Future interface. The cancel(), get() and isDone() methods are the most important methods from the Future interface.

## ExecutorService:

The Java ExecutorService is the interface which allows us to execute tasks on threads asynchronously. The Java ExecutorService interface is present in the java.util.concurrent package.

The ExecutorService helps in maintaining a pool of threads and assigns them tasks. It also provides the facility to queue up tasks until there is a free thread available if the number of tasks is more than the threads available.

Methods:

1. execute(Runnable command):

Submits a Runnable task for execution. The task will be executed asynchronously in the background.

1. submit(Callable<T> task):

Submits a Callable task for execution and returns a Future representing the pending result of the task.

1. shutdown():

Initiates an orderly shutdown of the ExecutorService, allowing previously submitted tasks to be executed before terminating.

1. isShutdown():

Returns true if this ExecutorService has been shut down.

The Executors class provides several factory methods for creating different types of thread pools. Here are a few examples:

1. Fixed-size Thread Pool:

This is the method you mentioned in your question. It creates a thread pool with a fixed number of threads.

**ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);**

1. Single-threaded Executor:

Creates a single-threaded executor, which is suitable for tasks that need to be processed sequentially.

**ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();**

1. Cached Thread Pool:

Creates a thread pool that can dynamically adjust the number of threads based on the workload. Threads that have been idle for a specified duration are terminated.

**ExecutorService cachedThreadPool = Executors.newCachedThreadPool();**

1. Scheduled Thread Pool:

Creates a thread pool that can schedule tasks to run after a certain delay or at fixed intervals.

**ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(3);**