Write a Program to Insert String Objects into the TreeSet where the Sorting Order is of

Reverse of Alphabetical Order:

Eg: Sorting\_String\_Objects\_InAlphabetical\_Order\_Using\_TreeSet

// go through the code

Eg: Sorting\_String\_Objects\_InReverse\_Order\_Using\_TreeSet

// go through the code

Write a Program to Insert StringBuffer Objects into the TreeSet where Sorting Order is Alphabetical Order:

Eg: Sorting\_StringBuffer\_Objects\_InAlphabetical\_Order\_Using\_TreeSet

In jdk 17 StringBuffer implements Comparable so we get sorted order

In previous versions of jdk StringBuffer does not implement Comparable

Eg: Sorting\_StringBuffer\_Objects\_InReverse\_Order\_Using\_TreeSet

Write a Program to Insert String and StringBuffer Objects into the TreeSet where Sorting Order is Increasing Length Order.

If 2 Objects having Same Length then Consider their Alphabetical Order:

eg: A,ABC,AA,XX,ABCE,A

Eg: String\_StringBuffer\_Objects\_Sorting\_In\_Increasing\_Length

// construct the binary tree

Note :

If we are using TreeSet() , then the condition is

1. Object should be homogenous
2. Object should be comparable ( class should implement Comparable(I))

If we are using TreeSet(Comparator c ) then the condition is

1. Object need not to be homogenous ( like the above example)
2. Object need not to implement comparable

When to go for Comparable interface and when to go for Comparator interface?

For predefined Comparable classes ( like String ) default natural sorting order is already available . if we are not satisfied with that we can define our own sorting by Comparator object

For predefined Non-Comparable classes ( like StringBuffer in jdk 11) default natural sorting is not already available . if we want to define our own sorting we can use comparator object .

For our own Classes (like Employee) the person who is writing the Employee class he is responsible to define his own sorting order by implementing Comparable interface

The person who is using our own class if he is not satisfied with our default natural sorting order he can define his own sorting order by using Comparator Object

If he is satisfied with the default natural sorting order then he can use our class directly

Write a Program to Insert Employee Objects into the TreeSet where default natural sorting order is Based on Ascending Order of Employeeld and

Customized Sorting Order is Based on Alphabetical Order of Names:

Eg: Customized\_Objects\_InTreeSet\_Based\_On\_Comparable\_Compartor

Comparable(I)

Present in java.lang Package

It is Meant for Default Natural Sorting Order.

Defines Only One Method compareTo()

All Wrapper Classes and String Class implements Comparable Interface.

Comparator(I)

Present in java.util Package

It is Meant for Customized Sorting Order.

Defines 2 Methods compare() and equals().

The Only implemented Classes of Comparator are Collator and RuleBaseCollator.

Functional interfaces :

If an interface contains only one abstract method then it is called functional interface

Javap java.util.function.Predicate;

public interface java.util.function.Predicate<T> {

public abstract boolean test(T);

// functional interface

// default methods available for utility methods for devevloper.

public default java.util.function.Predicate<T> and(Predicate p);

public default java.util.function.Predicate<T> negate();

public default java.util.function.Predicate<T> or(Predicate p );

// the above three are concrete methods , body of these methods is available in the Predicate interface , here public is just the useful for the visibility.

public static <T> java.util.function.Predicate<T> isEqual(java.lang.Object);

public static <T> java.util.function.Predicate<T> not(Predicate p );

// static method body is already available .

}

Eg: Predicate\_Test\_Method

Eg: Predicate\_Test\_Method\_Without\_Lambda\_Expression

Eg: Predicate\_Test\_Method\_With\_Lambda\_Expression

// use of test() method .

Take a particular input for that input you make a test give some results , for that purpose we use this method of predicate class.

Eg: Array\_With\_Predicate\_Test\_Method\_Eg1

// go through the code

Eg: Array\_With\_Predicate\_Test\_Method\_Eg2

// go through the code

Eg: Array\_With\_Predicate\_Test\_Method\_Eg3

// go through the code

Eg: Array\_With\_Predicate\_Test\_Method\_Eg4

// go through the code

Eg: Array\_With\_Predicate\_Test\_Method\_Eg5

// go through the code

Function (I) :

public interface java.util.function.Function<T, R> {

// functional interface

public abstract R apply(T);

T- Input type

R - return type

public default <V> Function <V,R> compose(Function<? super V, ? extends T>);

public default <V> Function<T, V> andThen( Function<? super R, ? extends V>);

public static <T> Function<T, T> identity();

}

Eg: Function\_Apply\_Method

Eg: Function\_Apply\_Using\_lambda

When to go for Predicate and when to go for Function?

Predicate -> To implement some conditional checks we should go for Predicate

Function -> To perform some operation and return some result we should go for Function

Method(::)Reference and constructor(::)Reference

========================================

:: -> scope resolution operator

Syntax for method reference

========================

1. static method

Classname::methodName

1. instance method

object::methodName

// first check this example before going to method reference

Eg: Thread\_With\_Lambda\_Expression

Eg: Method\_Reference\_Using\_Static\_Method

Eg: Method\_Reference\_Using\_Instance\_Method

Eg: Lambda\_Expresion\_With\_Constructor\_Reference

Consumer :

public interface java.util.function.Consumer<T> {

public abstract void accept(T);

public default java.util.function.Consumer<T> andThen(java.util.function.Consumer<? super T>);

}

Note: forEach() method can be used for every collection.

Eg: Consumer\_Apply\_Method\_Traditional\_Approach

Eg: Counsumer\_Apply\_Method\_With\_Lambda

Eg: Consumer\_Apply\_Method\_With\_Method\_Reference

Stream API:

Stream : channel through which there is free flow movement of data

Streams :

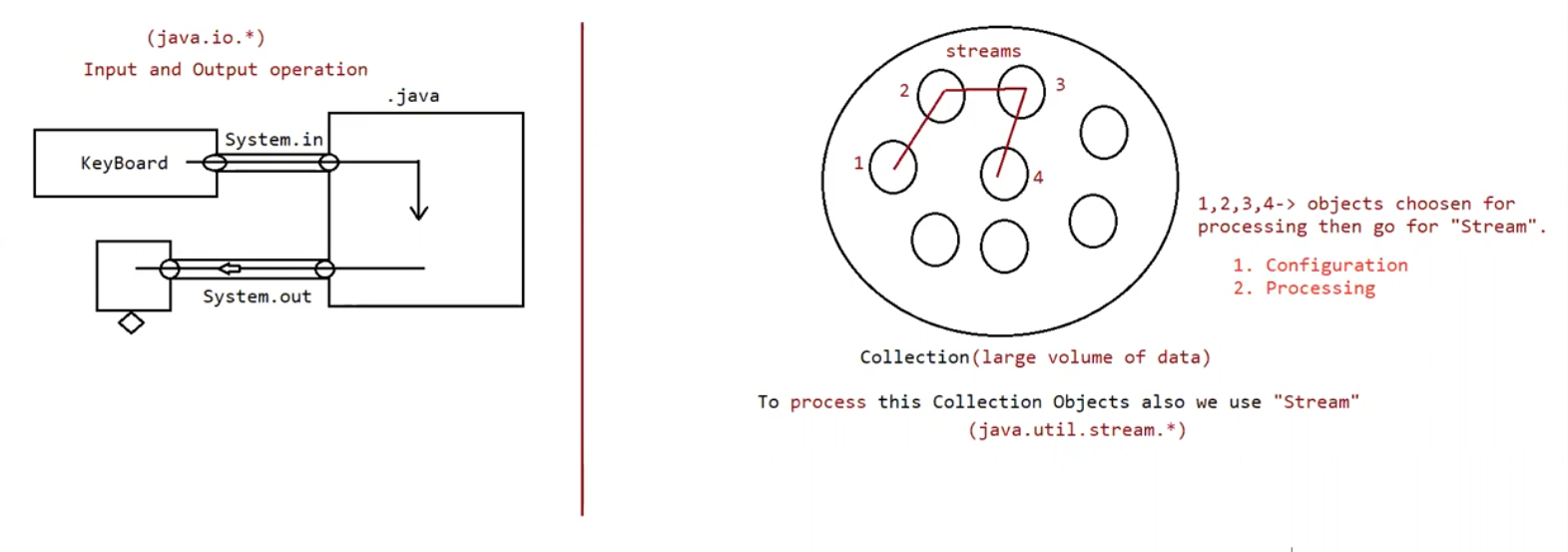
to process objects of collection , in version 1.8 Streams concept is introduced.

Taking out the object and working with the object is called process. For that process also we use Stream

What is the difference between java.util.streams and java.io.streams?

Java.util.streams is meant for processing objects from the collection . I.e it represents the stream of objects from the collection .

But java.io.streams is meant for processing binary and character data with respect to file i.e it represents stream of binary data or character data from the file . hence we can say both are different.



What is the difference between collection and stream?

If we want to represent a group of individual objects as a single entity then we should go for collection

If we want to process a group of objects from the collection then we should go for streams

We can create the stream object to the collection by using stream() method of collection interface . stream() method is default method added to collection in version 1.8

Stream is an interface present in java.util.stream. Once we got the stream , by using that we can process objects of that collection

We can process objects in the 2 phases

1. Configuration
2. Processing
3. Configuration :

We can configure either by using filter mechanism or by using map mechanism

Filtering:

We can configure a filter to configure the elements from the collection based on some boolean condition by using filter() method of Stream interface

public Stream filter (Predicate<T> t)

here (Predicate<T> t) can be boolean valued function / lambda expression

Ex:

Stream s = c.stream();

Stream s1 = s.filter(i -> i%2 == 0);

Hence to filter elements of collection based on some Boolean condition we should go for filter() method.

Mapping:

If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for

map() method of Stream interface.

public Stream map (Function f);

It can be lambda expression also

Stream s = c.stream();

Stream s1 = s.map(i-> i+10);

Once we performed configuration we can process objects by using several methods.

Eg: Collection\_Without\_Using\_Stream

Eg: Collection\_By\_Using\_Stream

Note:

On a objects if you want to do filtering ( on a collection if you want to remove some objects then go for filter() method)

No removal here for every object a new object has to be created then you go for map().

Eg: Collection\_Without\_Using\_Stream\_Eg1

Eg: Collection\_By\_Using\_Stream\_Eg1

Eg: Collection\_Without\_Using\_Stream\_Eg2

Eg: Collection\_By\_Using\_Stream\_Eg2

Note: Rather than using iterators it is preferred to use Stream.

2) Processing

processing by collect() method

Processing by count()method

Processing by sorted()method

Processing by min() and max() methods

forEach() method

toArray() method

Stream.of()method

Eg: Collection\_By\_Using\_Stream\_Eg3

On a sorted() method w.r.t the list , if you don’t pass any arguments , whatever objects you have added on that default natural sorting order is performed.

Internally Comparable compareTo() method will be called , based on that balanced binary tree will be created. Based on the balanced binary tree elements gets sorted and internally LVR technique is used and the result gets stored

Eg: Collection\_By\_Using\_Stream\_Eg4

Eg: Collection\_By\_Using\_Stream\_Eg5

Eg: Collection\_By\_Using\_Stream\_Eg6

Eg: Collection\_By\_Using\_Stream\_Eg7

Eg: Collection\_By\_Using\_Stream\_Eg8

Eg: Collection\_By\_Using\_Stream\_Eg9