**1. MIN and MAX Methods**

We use the **min() and max() methods** to find the min & max value in streams. These methods are used for finding min & max values in different types of streams such as stream of chars, strings, dates.\

List<LocalDate> dates =Arrays.asList(LocalDate.now(),

LocalDate.now().minusDays(9),

LocalDate.now().minusMonths(2),

LocalDate.now().minusDays(30));

//getting max date

LocalDatemaxdate=dates.stream()

.max(Comparator.comparing(LocalDate::toEpochDay))

.get();

//getting min date

LocalDatemindate=dates.stream()

.min(Comparator.comparing(LocalDate::toEpochDay))

.get();

**2. Filter Method**

The [filter](http://www.java67.com/2018/03/java-8-stream-find-first-and-filter-example.html) method, as its name suggests,**filters elements based upon a condition** you gave it. For example, if your list contains numbers and you only want numbers, you can use the filter method to only select a number that is fully divisible by two.

**The map(Function mapper) method** takes a [**Function**](https://javarevisited.blogspot.sg/2018/01/what-is-functional-interface-in-java-8.html)**,** technically speaking, an object of java.util.function.Function interface. This function is then applied to each element of Stream to convert it into the type you want.

Because we need to convert a String to an Integer, we can pass either the Integer.parseInt() or Integer.valueOf() method to the map() function.

List<String>numbers=Arrays.asList("1", "2", "3", "4", "5", "6");

System.out.println("original list: "+numbers);

List<Integer>even=numbers.stream()

.map(s->Integer.valueOf(s))

**.filter(number->number%2==0)**

**.collect(Collectors.toList**());//**which will accumulate all even numbers into a List and return.**

**3. Distinct Method in java**

The Stream API provides the distinct() method that returns different elements of a list based on the equals() method of the Object class.

distinct() uses hashCode() and equals() methods to get distinct elements.

List<String> list = Arrays.asList("AA", "BB", "CC", "BB", "CC", "AA", "AA");

long l = list.stream().distinct().count();

System.out.println("No. of distinct elements:"+l);

String output = list.stream().distinct().collect(Collectors.joining(","));

System.out.println(output);

**4. What is Parallel Stream?**

privatelongcountPrimes(intmax) {

returnrange(1, max).parallel().filter(this::isPrime).count();

}

We have the method **countPrimes** that counts the number of prime numbers between 1 and our max. A stream of numbers is created by a range method. The stream is then switched to parallel mode; numbers that are not primes are filtered out and the remaining numbers are counted.

Parallelization is just a matter of calling the parallel()  method. When we do that, the stream is split into multiple chunks, with each chunk processed independently and with the result summarized at the end

**5. Stack Peek Method in Java?**

The java.util.Stack.peek() method in Java is used to retrieve or fetch the first element of the Stack or the element present at the top of the Stack. The element retrieved does not get deleted or removed from the Stack.

Eg: Stack<String> STACK = newStack<String>();

STACK.push("Welcome");

System.out.println("The element at the top of the"

                           + " stack is: " + STACK.peek())

**6. Difference between comparator and comparable?**

**Comparable** provides a**single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price. Comparable **affects the original class.** Comparable provides **compareTo() method** to sort elements.

**The Comparator** provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc. Comparator **doesn't affect the original class.** Comparator provides **compare() method** to sort elements.

**7. Difference b/w Parallel Stream and Stream?**

for(inti=0;i<1000;i++){

list.add(i);

}

list.stream().forEach(System.out::println);

}

}

You will notice that this program will output the numbers from 0 to 999 sequentially in the order in which they are in the list. If we change stream() to parallelStream(). This is not the case anymore (at least on my computer): all number are written, but in a different order. So, apparently parallelStream() indeed uses multiple threads.

**8. String Constant Pool:-**

The **Java string constant pool** is an area in heap memory where **Java** stores **literal string** values. The heap is an area of memory used for run-time operations. When a new variable is created and given a value, **Java** checks to see if that exact value exists in the **pool**. If not, it creates a new **literal string**.

**9. flatmap():-**

**Stream flatMap(Function mapper)** returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Stream flatMap(Function mapper) is an **intermediate operation.** These operations are always lazy. Intermediate operations are invoked on a Stream instance and after they finish their processing, they give a Stream instance as output.

**Syntax: <R> Stream<R> flatMap(Function<? Super T, ? extends Stream<? extends R>> mapper**

**Eg:**

import java.util.\*;

import java.util.stream.Stream;

class GFG {

    // Driver code

    public static void main(String[] args)

    {

        // Creating a List of Strings

        List<String> list = Arrays.asList("5.6", "7.4", "4",  "1", "2.3");

        // Using Stream flatMap(Function mapper)

        list.stream().flatMap(num -> Stream.of(num)).

                         forEach(System.out::println);

    }

}

Output:

5.6

7.4

4

1

2.3

**10. How can we create UserDefined Unchecked Expections?**

* Create a user defined exception in java.
* Create a custom exception, by extending unchecked exceptions.
  1. We will create CustomArithmeticException by extending ArithmeticException class
  2. We will create CustomNullPointerException by extending NullPointerException class.

**11 Difference between ADD and SET methods in arraylist?**

* set replaces the element at the given index.
* add inserts the element at the given index and moves all elements ahead of it one position
* set replaces the element at the given index.
* add inserts the element at the given index and moves all elements ahead of it one position

Eg :-

ArrayList<Integer>arrlist = new ArrayList<Integer>();

inti = arrlist.set(3, 30);

Before operation : [1, 2, 3, 4, 5]

After operation : [1, 2, 3, 30, 5]

Replaced element : 4

inti = arrlist.set(3, 30);

Before operation : [1, 2, 3, 4, 5]

After operation : [1, 2, 3, 30,4, 5]

**12. String Pool:-**

**String Pool in java** is a pool of Strings stored in [**Java Heap Memory**](https://www.journaldev.com/4098/java-heap-space-vs-stack-memory). We know that String is special class in java and we can create String object using new operator as well as providing values in double quotes.

String Pool is possible only because [String is immutable in Java](https://www.journaldev.com/802/string-immutable-final-java) and its implementation of [String interning](https://en.wikipedia.org/wiki/String_interning) concept.

String pool helps in saving a lot of space for Java Runtime although it takes more time to create the string.

**13. String Constant Pool:-**

The **Java string constant pool** is an area in heap memory where **Java** stores **literal string** values. The heap is an area of memory used for run-time operations. When a new variable is created and given a value, **Java** checks to see if that exact value exists in the **pool**. If not, it creates a new **literal string**.

**14.Difference between HashMap and HashTable.**

* HashMap is non synchronized. It is not-thread safe and can’t be shared between many threads without proper synchronization code whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads.
* HashMap allows one null key and multiple null values whereas Hashtable doesn’t allow any null key or value.
* HashMap is generally preferred over HashTable if thread synchronization is not needed
* HashMap is a new class introduced in JDK1.2, whereas HashTable is a legacy class.
* HashMap is fast and HashTable is slow.
* HashMap is traversed by Iterator whereas HashTable can be traversed by Enumerator and Iterator.
* HashMap inherits AbstractMap class and HashTable inherits Dictionary class.

**15.Intern method:-**

The intern() method of the string method returns a canonical representation of the string object.

For any two strings s and t, s.intern()==t.intern() is true if and only if s.equals(t) is true. This method is used to return the string from the string pool.

**16.Why Checked Expections are Compile Time Exceptions?**

Checked Exceptions are compile time exceptions since these exceptions are checked at compile time. It should handle the exception

Using try-catch block or it should declare the exception using throws keyword. Otherwise the program will give compilation error.

**17.Difference between Linked List and Vector?**

* Linked list is implemented as a double linked list. Vector is similar with arraylist.
* Its Performance on add and remove is better than arraylist but worse on get and set methods. Vector each time doubles its arraysize.
* Linked list can act as a list cost. Vector is slow because it is synchronized.
* Linked list is better for manipulating data. Vector can use Iterator interface or Enumerator Interface to transfer elements.

**18. What are Vector**

**Vector class:-**

The Vector class implements a growable array of objects.

Vectors is fully compatible with collections.

* Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
* They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.

It extends AbstractList and implements List interfaces

**19Queue:-**

* The Queue interface is available in java.util package and extends the Collectiointerface.
* The queue collection is used to hold the elements about to be processed
* Provides various operations like the insertion, removal etc.
* It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of list (FIFO principle).
* Being an interface the queue needs a concrete class for the declaration and the most common classes are the [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/) and [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/) in Java.
* Both the implementations are not thread safe.
* PriorityBlockingQueue is one alternative implementation if thread safe implementation is needed.

Methods in queue:

add()- This method is used to add elements at the tail Methods in queue:

* add()- This method is used to add elements at the tail of queue.
* peek()- This method is used to view the head of queue without removing it. It returns Null if the queue is empty.
* element()- This method is similar to peek(). It throws NoSuchElementException when the queue is empty.
* remove()- This method removes and returns the head of the queue. It throws NoSuchElementException when the queue is empty.
* poll()- This method removes and returns the head of the queue. It returns null if the queue is empty.
* size()- This method return the no. of elements in the queue.

**20.Dequeue :**

* The java.util.Deque interface is a subtype of the [java.util.Queue](https://www.geeksforgeeks.org/queue-interface-java/) interface.
* The Deque is related to the double-ended queue that supports addition or removal of elements from either end of the data structure.
* It can be used as a [queue (first-in-first-out/FIFO)](https://www.geeksforgeeks.org/queue/) or as a [stack (last-in-first-out/LIFO)](https://www.geeksforgeeks.org/stack/).
* These are faster than Stack and LinkedList.

Methods of Dequeue

* a[dd(element)](https://www.geeksforgeeks.org/deque-add-method-in-java/" \t "_blank): Adds an element to the tail.
* [addFirst(element)](https://www.geeksforgeeks.org/deque-addfirst-method-in-java-with-examples/): Adds an element to the head.
* [addLast(element)](https://www.geeksforgeeks.org/deque-addlast-method-in-java/): Adds an element to the tail.
* [offer(element)](https://www.geeksforgeeks.org/deque-offer-method-in-java/): Adds an element to the tail and returns a boolean to explain if the insertion was successful.
* [offerFirst(element)](https://www.geeksforgeeks.org/deque-offerfirst-method-in-java/): Adds an element to the head and returns a boolean to explain if the insertion was successful.
* [offerLast(element)](https://www.geeksforgeeks.org/deque-offerlast-method-in-java/): Adds an element to the tail and returns a boolean to explain if the insertion was successful.
* [iterator()](https://www.geeksforgeeks.org/deque-iterator-method-in-java/): Returns an iterator for this deque.
* [descendingIterator()](https://www.geeksforgeeks.org/deque-descendingiterator-method-in-java/): Returns an iterator that has the reverse order for this deque.