1. **Distinct():**

\*distinct() returns a stream consisting of distinct elements of that stream.

\*It is the method of stream interface.

\*It uses hashcode() and equals() methods to get distinct elements.

\*In the case of unordered stream the selection of distinct elements is not necessarily stable and can change.

**Syntax:** stream<T>distinct()

**Example:**

package com.Concretepage;

import java.util.arrays;

import java.util.list;

import java.util.stream.collectors;

public class DiatinctSampleDemo{

List<String>list=Arrays.asList(“aa”,”bb”,”bb”,”cc”,”aa”,”aa”);

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

System.out.println(“No.of distint elements:”+1);

String output=list.Stream().distinct().collect(collectors.joining(“,”));

System.out.println(output);

}

}

**Output:** No.of distinct elements:3

Aa,bb,cc

1. **peek():**

Looks at the object at the top of this stack without removing it from the stack.

**Example:**

import java.util.stack;

stack<integer>nums=new stack<Integer>();

num.push(1);

num.push(2);

num.push(3);

num.push(4);

System.ou.prinln(nums.peek());

System.out.println(nums);

**Output**: 4

[1,2,3,4]

1. **parallel Streams:**

\*In a parallel stream, the operations are executed in parallel and there are two ways to create a parallel stream using the parallelStream() and parallel) methods.

**Syntax:**

Optional<Integer>calcprod=list1.parallelStream().rduce((a,b)->a\*b));

\*It must be used only with the

1.Stateles operation=>state of one element does not effect the other element.

2.non-interface=>in witch data source is not effected.

3.associative=>the result is not affected by the other of operands.

**Example:**

We have created a list of 600 employees out of which there are 300 employees out of which there are salary is above 15000.

Package com.jcg.java;

Import java.util.ArrayList;

Import class ParalelStreamDemo {

Public static void main(String args[])

{

Long t1,t2;

List<Emlpoyee>eList=new ArrayList<Employee();

For(int i=0;i<100;i++)

{

eList.add(new Employee(“A”,20,000));

eList.add(new Employee(“B”,20,000));

eList.add(new Employee(“C”,20,000));

eList.add(new Employee(“D”,20,000));

eList.add(new Employee(“E”,20,000));

eList.add(new Employee(“F”,20,000));

t1=System.currentTimeMills();

Systum.out.println(“parallelstream count?=”+eList.parallelStream().filter(e->e.getsalary()>15000.count());

t2=System.currentTimeMillis();

System.out.printi(“parallelStream time taken?=”+(t2-t1));

}

}

1. **flatMap():**

\*It returns a stream consisting of the result of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function t each element.

\*It is an intermediate operation. These operations are always lazy.

\*map() take a stream and transform it to another stream.

\*It applies a function on each element of stream and store return value into new stream. It does not flatten the stream.

\*But flatMap() is the combination of map and a flat operation i.e.., it applies a function to elements as well as flatten them.

Syntax: Stream<r>flatMap(Function<?superT,?extends stream<? Extends R>>mapper)

Eample:

Import java.util.\*;

Import java.util.Stream.stream;

Class GFG{

Public static void main(String args[])

{

List<string>list=Array.alist(“5.6”,”7.4”,”4”,”1”,”2.3”);

List.stream()flatMap(num->stream.of(num)). forEachSystem.out::println);

}

}

**Output:**

5.6

7.4

4

1

2.3

**5.Min() and max() methods:**

Collections.min() method return the minimum element in the specified collection and Collections.max () returns the maximum element in the specified collection, according to the natural ordering of its elements.

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class GFG {

 public static Integer findMin(List<Integer> list)

    {

if (list == null || list.size() == 0) {

 return Integer.MAX\_VALUE;

 }

public static Integer findMax(List<Integer> list)

   {

 if (list == null || list.size() == 0) {

            return Integer.MIN\_VALUE;

        }

|  |
| --- |
| return Collections.max(list);      }        public static void main(String[] args)      {              List<Integer> list = new ArrayList<>();              list.add(44);          list.add(11);          list.add(22);          list.add(33);            System.out.println("Min: " + findMin(list));          System.out.println("Max: " + findMax(list));      }  } |

**Output:**

Min: 11

Max: 44

1. String Constant pool:

When you declare a new string in Java, there are some interesting things that happen behind the scenes. This is a basic string declaration. We create a new string variable called *employee* and give it a value, like you can see in the code:

String employee = "Edgar Allen Poe";

Not only will Java create the variable *employee*, it will allocate space in the memory for the literal value 'Edgar Allen Poe.' This area in memory is called the **string constant pool**. It's like a pool of string values that are available to other parts of the program.

**String Pool in Java**

String Pool is a storage area in Java heap.

String allocation, like all [object allocation](https://www.edureka.co/blog/java-object/), proves to be a costly affair in both the cases of time and memory. The JVM performs some steps while initializing string literals to increase performance and decrease memory overhead. To decrease the number of String objects created in the JVM, the String class keeps a pool of strings.

Each time a string literal is created, the [JVM](https://www.edureka.co/blog/java-architecture/#componentsofjava) checks the string literal pool first. If the string already exists in the string pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object initializes and is placed in the pool.

**Example:**

public class StringPoolExample

{

public static void main(String[] args)

{

String s1=”Rachel”;== s

String s2=”Rachel”;==

String s3=new String(”Rachel”);==

String s4=new String(”Rachel”).intern();

System.out.println(s1==s2);

System.out.println(s1==s3);

System.out.println(s1==s4);

}

}

== 3); //

**Output:**

True  
False  
True

1. **Program to revere a string using method of string class?**

import java.uil.Scanner;

public class ReverseString

{

public static void main(String args[])

{

System.ou.println(“Enter string to reverse:”);

Scanner read=new Scanner(System.in);

String str=read.nextLine();

String reverse=” ”;

for(int i=str.length()-1;i>=0;i--)

{

reverse=reverse+str.charAt(i);

}

System.out.println(“reverse string is:”);

System.out.println(reverse);

}

}

1. **Program to print alternate value of a 2D Array?**

Class Sample1{

static final int R=3;

static final int C=3;

static void convert(int arr[][]) {

boolen leftTpRight=true;

for(int i=0;i<R;i++)

{

if(leftToRight) {

for(int j=0;j<C;j++)

{

System.out.println(“%d”,arr[i][j]);

}

}

else

{

for(int j=C-1;j>=0;j--)

{

System.out.println(:%d”,arr[i][j]);

}

}

leftToRight=!leftToRight;

}

}

static public void main(String args[])

int arr[][]={{1,2,3}{3,2,1}{4,5,6}}

convert(arr);

}

}

1. **What is immutability?**

In java, String object are immutable. Immutable simply means modifiable or unchangeable.

Once string object is created its data or state can’t be changed but a new string object is created.

Example:

Class TestImmutableString{

Public static void main(String args[]){

String s=”Sachin”;

s.concat(“Tendulkar”);//concat() method appends the string at the end

System.out.println(s);//will print Sachin because strings are immutable

} objects.

}

1. **Difference between hashMap and hashTable?**

**hashMap hashTable**

1. Introduced in 1.2 version 1.There since java was

introduced.

2. It is not thread safe2.It is thread safe and is

and unsynchronized.Synchronized.

3. It is fact. 3. It is slow.

4. Works with single thread 4.Works with multiple threads.

5. Allows one null key. 5. Does not allow null key.

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

for (int i = 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.

1. **Difference between ADD and SET methods in arraylist?**

set replaces the element at the given index. add inserts the element at the givenindex 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

EX :-

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

int i = arrlist.set(3, 30);

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

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

Replaced element : 4

int i = arrlist.set(3, 30);

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

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

1. **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. omparator **doesn't affect the original class.** Comparator provides **compare() method** to sort elements.