1. **Comparable vs Comparator:**

**Comparable** is user for the single object comparrisen with the compairTo method.

For, Example we need to sort the Employee records based on the empId.

Class **Employee** implements Comparable<Employee> {

Public void compreTo(Employee e) {

return e.empId - e1.empId;

}

}

**Collections.sort(empList);**

We can not sort one same model collection with the one with the Id and one with the name in the comparable. The Sort for all the collection of the same type collection is done based on the compairTo method implementation in the Model.

**Comparator** is user to compair the multiple collection attribute with sort that attribute.

For Example: we need to sort the employee based on their id for one collection.

And for the second collection we need to sort based on the name then we use comparator.

For Comparator we need to create the different call for the comparator.

Public class **EmployeeIdComparator** implements Comparator<Employee> {

Public void compir(Employee e, Employee e1) {

return e.empId - e1.empId;

}

}

Public class **EmployeeNameComparator** implements Comparator<Employee> {

Public void compir(Employee e, Employee e1) {

return e.empName.compareTo(e1.empName);

}

}

**Collections.sort(empList, new EmployeeIdComparator());**

**Collections.sort(empList, new EmployeeNameComparator());**

1. **Reference Comparison vs Content Comparison**

**Reference Comparison** is used to comapir the two object address location.

Ex: (ObjStudent1 == ObjStudent2) it will compair the address of both the objects.

**Content Comparison** is used to compair the value of the object

Ex: (ObjStudent1.name.equals(ObjStudent2.name)) it will compair the name of both the objects.

For the Content Comparison we need to override the “equals” method in the Entity class. If not override the that call the parent class Object equals method. That can compair the reference of the object.

1. **How HashMap Works internally:**

HashMap<String,String> map = new HashMap<String,String>();

map.put(“index1”, “test”);

The put method works that create one bucket in the heap memory.

Generate the hashcode of the item and put that in the index that found from the hash code.

Hashmap uses the Linked List,

Element contains: HashCode, Key, Element, Link

HashMap contain the two methods equals and hashcode methods.

Equals method compair the address of the two key by default from the object class equals method if we not override the equals method.

Hashcode check the hashcode of the two element.

**Hash Collision:** Hash of the given element is the same as the element that are stored in the heap memory.

Then in that case key of the both the node is need to compair using the equals method. If both key are differente then we need to store the node in the same location and give reference to the new node.

If key is null in the put method of the hashMap then element is store at the 0th index of the hashtable.

For geting the Element from the HashMap

map.get(“index1”);

In that case, first hash key is generated for the given key with the hash key we get the hashtable index where the element is stored. After that both the key compair with the compairTo method. If match the the value is return.

1. Which datastructure HashMap represent?

Hash map use which DataStructure: **HashMap Data Structure**

1. Which dataStructure used to implement the HashMap.

Array and LinkedList: Array for the Buckets and LinkList for the same element in one index.

1. Is HashMap is the ThreadSafe.

No,

1. What happens when we user the HashMap in the Thread safe application.

Multiple thread can access the same map then all are modify the same data at the same time.

1. How do you remove mapping while iterating the Hashmap.

remove() of iterator.

1. What is the load factor in the HashMap.

Load factor is the function to resize the map size when required.

1. What is the Maximum limit of the Hashmap.

No limit.

size() return the int value. For more value we user mappingCount() method instead of the size(). mappingCount return the long value.

1. Difference between Capacity and size in hashmap.

Capacity: how many element we have store in hashmap.

Size. how many element we stoer in the hashmap.

1. Concurrent HashMap.

We have use that in the thread safe environment.

1. difference between HashTable and ConcurrntHashMap

Performace and scalability is higher in ConcurrntHashMap as compared to HashTable.

1. Duplicate key are allow in the HashMap.

Keys can not the same but duplicate values are allowed.

1. What happens when hashcode of the two different key is the same.

It is the HashColission.

1. ConcurrentModificationException: when muliple theread can access the same object.
2. **Threads**

Runnable interface and extends the Thred class.

Which is the better way to create thread? : Runnable and callable is the best way, because if we extend the Thread class then we can not extend another classes because multiple inheritance is not supported by the java.

run() vs start():

Start method start the thread to call the run method.

Run method call the run method of the running thread.

1. **Dynamic Proxy:** Dynamic proxy is the logic that is to be execute before the actual code.

InvocationHandler is the interface for the dynamic proxy creation.

And invoke() method is used for the proxy logic.

**Number of ways we can iterate Java collection**

1. using the for loop
2. Using the for each loop
3. Using the iterator
4. Using the Java 8 stream api forEach loop
5. Using the ItemIterator

**4. Serialize and Deserialization:**

Seialization: convert object to byte.

Deserialization: converting the byte to object.

We can store the object in file bytes and get the object with that file when ever we require.

For Serialize we can implements serializable interface in the model class.

We can not read directly from the file first we need to Deserialization object and then get the data.

**5.** **List vs Set:**

List maintains the insertion order. Set not maintain the insertion order.

ArrayList and LinkList, HashSet and LinkeHashSet.

Set don’t have the duplicate element. List have the duplicate element.

1. **Thread-safe, fail-safe class in java**

**Stack, HashTable, Vector, Properties**

**String vs StringBuilder vs StringBuffer**

String is immutable whereas StringBuffer and StringBuilder are mutable classes.

StringBuffer is thread-safe and synchronized whereas StringBuilder is not. That’s why StringBuilder is faster than StringBuffer.

String concatenation operator (+) internally uses StringBuffer or StringBuilder class.

For String manipulations in a non-multi threaded environment, we should use StringBuilder else use StringBuffer class.

**What is the Docker:**

Docker is the platform to run and develop the application at any platform.

We have images ready we can user it from the docker hub.

Docker images is the software that we need to run and Docker container is the container where we run that image. With the “docker run” command.

**Continuous integration** is focused on automatically building and testing code, as compared to continuous delivery, which automates the entire software release process up to production. That is in aws.

**Auto Scaling:** we can scale up and down when ever required.

**Kubernetes** is used to create applications that are easy to manage and deploy anywhere.

**JUNIT:** Testcase we basically use the mokito which create the dummy data and test that.

JUnit annotation: @Test, @Test(timeout=500)= define some time out when we test some service then test should be complete in the given timeout, @Test(expected=IllegalArgumentException.class)= this method is using to catch some exception, @Before= run before each test, @After=run after each test to clean the resources, @BeforeClass=it's preferable to execute it only once before running all tests, @AfterClass=it's preferable to execute it only once after running all tests, @Ignores.

@BeforeEach= run before all test case. and @BeforeAll = run once before start test case. are the JUnit 5 equivalents of @Before and @BeforeClass

JUnit Assert Class:

void assertEquals(boolean expected, boolean actual)

void assertFalse(boolean condition)

void assertNotNull(Object object)

void assertNull(Object object)

void assertTrue(boolean condition)

void assertSame([String message]

void assertNotSame([String message]

**service-oriented architecture:**

It is the Architecture where each service can communicate with the each other.

**Coupling: loose coupling and tight coupling**

loose coupling is recommended because it's easier to test and maintain.

**Cohesion:**

Suppose we have the one class with the two method one add two number and one to concat the string. There two method with perform some different activity and that is not related to each other.

So what we do we can create the two different class for the two different methods and call using the reference of the object that is the use of the cohesion in java.

**Functional Interface:**

@FunctionalInterface

interface Shape {

void printArea(int x);

}

public class SquareTest {

public static void main (String args[]) {

Shape square = (x) -> { // Lambda Expression

System.out.println("The area of a square is: "+ x\*x);

};

square.printArea(7);

}

}

**Java 8:** <https://www.youtube.com/watch?v=oUdENE7ljjw&list=PLyHJZXNdCXsdeusn4OM33415DCMQ6sUKy>

<https://www.youtube.com/watch?v=PMhrMDHFFW4&list=PLyHJZXNdCXsfcMboYwGoL6wKAFPxljz5W>

Predicate<Integer> pre= (a) -> a > a+2; pre.test(10);

Predicate chain: and,or,negate ex: pre.and(pre1).test(10);

Function<Integer,Integer> fn = (a) -> a\*a; Integer result = fn.apply(2);

Function chan: andThen, compose fn.andThen(f2).apply(2); fn.compose(f2).apply(2);

Consumer<Integer> co= (i) -> syso(i); co.accept(10);

Consumer chain: andThen co.andThen(co1);

Supplier<Integer> su = () -> return 10; su.get();

Supplier chain is not possible because it not take any input.

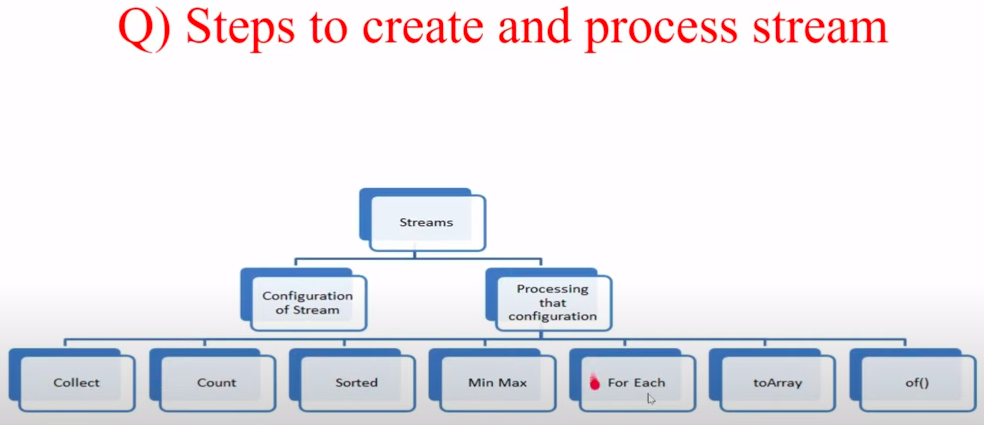
BiConsumer, Bisupplier, BiFunction, BiPredicate: take two inputs from the user.

BiPredicate<Integer,Integer> pre = (a,b) -> a+b; pre.test(10,20);

BiSupplier<Integer,Integer> sup = (a,b) -> return a+b; sup.get();

BiConsumer<Integer,Integer> con = (a.b) -> sop(a+b); con.accept(10,20);

BiFunction<Integer,Integer, Integer> fn = (a,b) -> a+b; fn.apply(10)

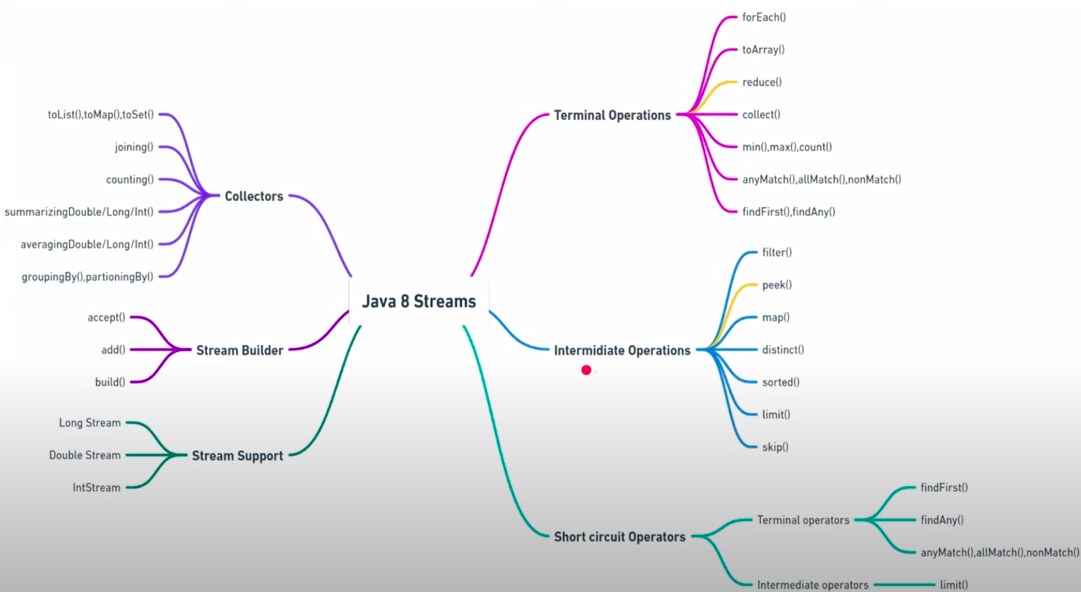


Convert stream to Array:

Object[] obj= stremobj.toArray();

Covert the elements into strem

Strem.of(“1”,”2”,”3”);



Reduce: preform the action on the all the element in the list and return one single result.

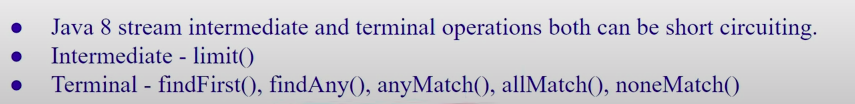
Int total = inList.stream().reduce((a,b)-> a+b).get();

Short Circute operator:

Ex: 10>2 && 10<2 return false

10>2 && 10<12 return true

&& , || are the operator used in that.



import java.util.stream.\*; - Collectors

import java.util.function.\*; - function

-> Find the duplicate Element in the List

Integer a[] = {10,20,30,40,50,10,30,50,30};

Set<Integer> listSet = new HashSet<Integer>();

Arrays.asList(a).stream().filter((no)-> **!**listSet.add(no)).collect(Collectors.toSet()).forEach(System.out::println);

-> remove duplicate element from the List

Arrays.asList(a).stream().filter((no)-> listSet.add(no)).collect(Collectors.toSet()).forEach(System.out::println);

**->** count the number of word occurrence in the String

String str="Hello world Hello User world is here";

List<String> stringList = Arrays.asList(str.split(" "));

Map<String, Long> finalmap = stringList.stream().collect(Collectors.groupingBy(Function.identity(),Collectors.counting()));

System.out.println(finalmap);

-> check the number is prime or not

int no =10;

boolean ans = IntStream.range(2,no/2).noneMatch(n-> no%n==0);

System.out.println(ans);

-> random number generator

Random ram = new Random();

ram.ints(0,10).limit(1).forEach(System.out::println);

-> print the first 10 prime numbers

// print the N prime numbers

Stream.iterate(2,i-> i+1).filter(Main::checkprimeNumber).limit(10).forEach(System.out::println);

public static boolean checkprimeNumber(Integer no) {

return IntStream.range(2,no/2).noneMatch(n -> no%n==0);

}

-> find all the city of the employees works.

Ex: class Employee{

String empId;

String empName;

List<String> cities;

}

Set<String> cities = empList.stream().flatMap((e) -> e.getCities().stream()).collect(Collectors.toSet());

-> **Optional**

Sting str =null;

Optional<String> str = Optional.**ofNullable**(str);

Optional<String> empName = Optional.**of**(emp.getName().get()); // if we are sure that we get value null then we use of instead of the ofNullable.

empName.**isPresent**() empName.**isEmpty**()

empName.**ifPresent**((e) -> sop(e))

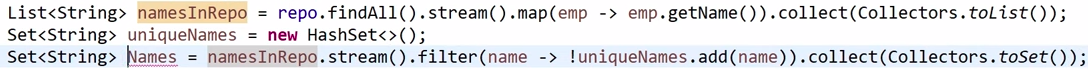
empName.**ifPresentOrElse**((e)-> sop(e), “Not present”);

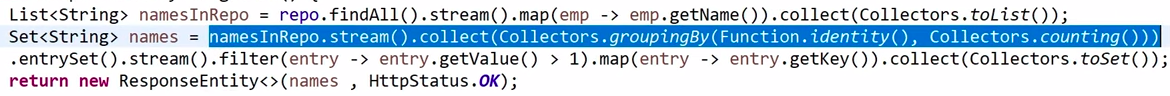
String str = Optional.**ofNullable**(str).**orElse**(“Error”);

String str = Optional.**ofNullable**(str).**orElseGet**(() -> “Error”);

Optional.**ofNullable**(str).**orElseThrow**(new Exception(“Not Found”));

-> Find the Repeted values in field:







-> Reverse the list in descending order

Integer: arrList.stream().sorted((n1,n2) -> n2-n1).collect(Collectors.toList());

String: arrList.stream().sorted((n1,n2) -> n2.compairTo(n1)).collect(Collectors.toList());

-> get min.max, avg, sum, count

arr.stream().mapToInt(x->x).summaryStatistics() // get all things sum, avg, min, max, …

arr.stream().mapToInt(x->x).summaryStatistics().getSum() // get sum only

-> return all the String in Uppercase in one String

{Test, test2, test3}

String result= strList.stream().map((s)-> s.toUpperCase()).collect(Collectors.joining(“, “));

// TEST, TEST2, TEST3

-> get all the Employees which have the group by age

Map<Integer,Map<Employee>> mapData = empList.stream().collect(Collectors.groupingBy(e-> e.getAge()));

If we need the unique data,

Map<Integer, Set<Employee>> mapData = empList.stream().collect(Collectors.groupingBy(e-> e.getAge(), Collectors.toSet()));

If er need unique data with sorted in order

Map<Integer, Set<Employee>> mapData = empList.stream().collect(Collectors.groupingBy(e-> e.getAge(),**TreeMap::new** ,Collectors.toSet()));

-> find the second largest in the List

numList.stream().sorted((n1,n2)-> n2-n1).skip(1).findFirst();

-> find the duplicate elements from two different array

Int a1={10,20,30,40}

Int a2={20,50,10,60.80}

a1.stream().filter((ar1)-> a2.stream().anyMatch(ar1) -> ar1== ar2).boxed().collect(Collectors.toList());

-> find the longest string from the array

lagestString.stream().sorted((s1,s2) -> s2.length() - s1.length()).limit(1).forEach(System.out::println);

-> remove duplicate element with preserv the order

List<String> removeDuplicate = Arrays.asList("test1", "test2","test1", "test1sds", "test2");

Set<String> data = new LinkedHashSet<>();

Set<String> strData = removeDuplicate.stream().filter(s-> data.add(s)).collect(Collectors.toSet());

System.out.println(strData);

**SOLID** Principal:

Single Responsibility principle: Class has only one reason to change.

Open close Principle: open for the Extension but close for the Modification.

Liskov substitution: If class B is the subtype of A, then we should be able to replace object of A with the B without breaking the behaviour of the program.

Interface Segment: implement the unnecessary function they do not need.

Dependency Inversion: class should depends on the interface rather than the Concrete class.

**ACID** property in Database:

**Automicity**: Either all or Null,

**Consistancy**: before the transaction start and after the transaction end sum of amount is same Ex: BankA transfer to BankB. BankA: 2000 - 500 (Debit) BankB: 3000+500(Creadit), so before total 2000+3000 after commit total = 1500+ 3500 same,

**Integrity**: Convert the paralal trasaction to single transaction,

**Durability**: Commit the chage in the database perminantly.