1. Why we need to go for lamba expression.

To give the implementation of a functional Interface .

Syntax

```
(arg1, arg2...) -> { body }
 or
(type1 arg1, type2 arg2...) -> { body }
```

Ex:-

```
    interface Sayable {
        public String say();
    }

    public class LambdaExpressionExample3 {
        public static void main(String[] args) {

            Sayable s = () -> {
                return "I have nothing to say.";
            };
            System.out.println(s.say());
        }
}
```

2. What is functional interface. Is it possible to write default method in function Interface. Can we overide the default methods. Write and demonstrate.

Functional Interface is a interface which contain only one abstract method .

Note:- it may contain multiple default and static method but only one abstact method.

Default method get override but static method never get override.

- Built-in Functional Interfaces in Java 8 are:
  - Consumer – BiConsumer
  - Predicate-BiPredicate
  - Function
  - Unary Operator
  - Binary Operator

3. What is Method reference how you implemented in your project.

A method reference is the shorthand syntax for a lambda expression that contains just one method call.

Syntax:-

Object :: methodName

Types of Method References
There are four type method references that are as follows:

1. Static Method Reference.
2. Instance Method Reference of a particular object.
3. Constructor Reference.

Ex:-

Employee.java:-

```
package com.fullcreative.Entity;
@Data
public class Employee {

      private String name;
      private String address;
      private Long mobilenumber;
      private String email;
      private float salary;


      public Employee() {
    System.out.println("I got call through constructor ref");
      }


      public Employee(String name, String address, Long mobilenumber,
String email, float salary) {
            super();
            this.name = name;
            this.address = address;
            this.mobilenumber = mobilenumber;
            this.email = email;
            this.salary = salary;
      }

      public void display() {
            System.out.println(toString());

      }

}
```

EmployeeResponse.java:-

```java
package com.fullcreative.Entity;

@Data
public class EmployeeResponse {

	private String name;
	private String email;
	private float salary;

	}
```

IconstuctorRef.java:-

```java
package com.fullcreative.Entity;

@FunctionalInterface
public interface IContructorRef {

	public Employee getEmployee();
}
```

Imethodref.java:-

```java
package com.fullcreative.Entity;

public interface IMethodRef {

	public void doTask();
}
```

EmployeeDao.java:-

```java
package com.fullcreative.Dao;

import java.util.ArrayList;
import java.util.List;
import com.fullcreative.Entity.Employee;

public class EmployeeDao {

	public static List<Employee> getEmployee() {
		List<Employee> list = new ArrayList<>();

list.add(new Employee("Rustam","Banglore",8808444331,"rustam@gmail.com", 1000.0f));
list.add(new Employee("amir", "Delhi", 9857745678l, "amir@gmail.com", 900.0f));
list.add(new Employee("atul", "Banglore", 9839373231, "atul@gmail.com", 1500.0f));
list.add(new Employee("saddam","Banglore",7884757693l,"saddam@gmail.com", 800.0f));
list.add(new Employee("reshabh","Lukhnow",398784331, "reshabh@gmail.com", 550.0f));
list.add(new Employee("Roy", "Telangana", 8975847381, "roy@gmail.com", 700.0f));
list.add(new Employee("akhil", "Chennai", 5927483648l, "akhil@gmail.com", 200.0f));
list.add(new Employee("raj", "Patna", 893748481, "raj@gmail.com", 100.0f));
list.add(new Employee("deepak", "Nanital", null, "deepak@gmail.com", 300.0f));
list.add(new Employee("niketan", "Banglore", 8974574541, null, 600.0f));
list.add(new Employee("anmol", null, 8974574541, null, 400.0f));
		return list;
	}

//	public static List<Employee> getEmployee() {
//		return IntStream.rangeClosed(1, 10)
```

```
//                      .mapToObj(i->new Employee("employee"+i, "city"+i,
new Random().nextLong(7777777777l, 9999999999l),
"employee"+i+"@gmail.com",new Random().nextFloat(5000)))
//                      .collect(Collectors.toList());
//     }
      static float avg=0;
      public static void average(Employee list) {
            float salary = list.getSalary();
            avg+=salary;
            System.out.println(avg);
      }
}
package com.fullcreative.service;

import java.util.List;
import java.util.stream.Collectors;

import com.fullcreative.Dao.EmployeeDao;
import com.fullcreative.Entity.Employee;
import com.fullcreative.Entity.EmployeeResponse;
import com.fullcreative.Entity.IContructorRef;
import com.fullcreative.Entity.IMethodRef;

public class MethodReference {

    public static void main(String[] args) {

List<Employee> employee = EmployeeDao.getEmployee();
for (Employee emp : employee) {
System.out.println(emp);
}
System.out.println("***************************");

// 1 static method reference
employee.forEach(MethodReference::print);

MethodReference mapper = new MethodReference();

//2 giving pre defined functional interface
implemention through method ref
Runnable runnable = mapper::threadTask;
Thread thread = new Thread(runnable);
thread.start();

// 3 custom functional interface implementation using
method reference
IMethodRef ref = mapper::threadTask;
ref.doTask();

// 4 calling custom method of convoter
```

```
List<EmployeeResponse> collect =
employee.stream().map(mapper::convoter).collect(Colle
ctors.toList());
System.err.println(collect);

// 5 By defualt getter method Employee::getSalary are
static in lamda exp
float totalSum=employee.stream().map(Employee::getSalary)
             .reduce((float) 0, (v1, v2) -> v1 + v2);
System.out.println(totalSum);


//-----------Constructor implementation--->

IContructorRef ref2=Employee::new;
Employee employee2 = ref2.getEmployee();
employee2.display();


    }

    // ---------All these method are written in
different class------------>

    private static void print(Employee list) {
        System.out.println(list);
    }

    private EmployeeResponse convoter(Employee emp) {
EmployeeResponse response = new EmployeeResponse();
        response.setName(emp.getName());
        response.setEmail(emp.getEmail());
        response.setSalary(emp.getSalary());
        return response;
    }

    private void threadTask() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i * 2);
            try {
                Thread.sleep(1000);
            } catch (Exception e) {
                e.printStackTrace();
            }
```

```
                }
        }

    }
```

Output:-

```
Employee [name=Rustam, address=Banglore, mobilenumber=8808404433,
email=rustam@gmail.com, salary=1000.0]
.
.
.
Employee [name=anmol, address=null, mobilenumber=897457454, email=null,
salary=400.0]
**********************************************************
Employee [name=Rustam, address=Banglore, mobilenumber=8808404433,
email=rustam@gmail.com, salary=1000.0]
.
.
.
Employee [name=anmol, address=null, mobilenumber=897457454, email=null,
salary=400.0]
0
0
2
2
.
.
.
18
18
[EmployeeResponse [name=Rustam, email=rustam@gmail.com, salary=1000.0],
EmployeeResponse [name=amir, email=amir@gmail.com, salary=900.0],
.
.
.
EmployeeResponse [name=niketan, email=null, salary=600.0], EmployeeResponse
[name=anmol, email=null, salary=400.0]]
7050.0
I got call through constructor ref
Employee [name=null, address=null, mobilenumber=null, email=null,
salary=0.0]
```

4. Explain about stream api. What are the benefits. explain few stream methods you used in your project.

Stream API is used to process collections of objects.

A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

Note:-

A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.

Method used in Stream Api:-

1. forEach

it loops over the stream elements, calling the supplied function on each element.

Ex:- empList.stream().forEach(e -> e.salaryIncrement(10.0));

2. map

map() produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.

Ex:-

```
Integer[] empIds = { 1, 2, 3 };

 List<Employee> employees = Stream.of(empIds)
                                  .map(employeeRepository::findById)
                                  .collect(Collectors.toList());
```

3. collect

```
List<Employee> employees = empList.stream().collect(Collectors.toList());
```

4. filter

this produces a new stream that contains elements of the original stream that pass a given test (specified by a Predicate).

Ex:-

```
Integer[] empIds = { 1, 2, 3, 4 };

List<Employee> employees = Stream.of(empIds)
  .map(employeeRepository::findById)
  .filter(e -> e != null)
  .filter(e -> e.getSalary() > 200000)
  .collect(Collectors.toList());
```

In the example above, we first filter out *null* references for invalid employee ids and then again apply a filter to only keep employees with salaries over a certain threshold.

5. findFirst

findFirst() returns an *Optional* for the first entry in the stream; the *Optional* can, of course, be empty:

Ex:-

```
Integer[] empIds = { 1, 2, 3, 4 };

Employee employee = Stream.of(empIds)
  .map(employeeRepository::findById)
  .filter(e -> e != null)
  .filter(e -> e.getSalary() > 100000)
  .findFirst()
  .orElse(null);
```

Here, the first employee with the salary greater than 100000 is returned. If no such employee exists, then *null* is returned.

6. toArray

We saw how we used *collect()* to get data out of the stream. If we need to get an array out of the stream, we can simply use *toArray()*:

```
Employee[] employees = empList.stream().toArray(Employee[]::new);
```

The syntax *Employee[]::new* creates an empty array of *Employee* – which is then filled with elements from the stream.

7. flatMap

A stream can hold complex data structures like *Stream<List<String>>*. In cases like this, *flatMap()* helps us to flatten the data structure to simplify further operations:

```
List<List<String>> namesNested = Arrays.asList(
  Arrays.asList("Jeff", "Bezos"),
  Arrays.asList("Bill", "Gates"),
  Arrays.asList("Mark", "Zuckerberg"));

List<String> namesFlatStream = namesNested.stream()
  .flatMap(Collection::stream)
  .collect(Collectors.toList());
```

Notice how we were able to convert the *Stream<List<String>>* to a simpler *Stream<String>* – using the *flatMap()* API.

8. peek

We saw *forEach()* earlier in this section, which is a terminal operation. However, sometimes we need to perform multiple operations on each element of the stream before any terminal operation is applied.

peek() can be useful in situations like this.

```
Employee[] arrayOfEmps = {
    new Employee(1, "Jeff Bezos", 100000.0),
    new Employee(2, "Bill Gates", 200000.0),
    new Employee(3, "Mark Zuckerberg", 300000.0)
  };

  List<Employee> empList = Arrays.asList(arrayOfEmps);

  empList.stream()
    .peek(e -> e.salaryIncrement(10.0))
    .peek(System.out::println)
    .collect(Collectors.toList());
```

Here, the first *peek()* is used to increment the salary of each employee. The second *peek()* is used to print the employees. Finally, *collect()* is used as the terminal operation.

Method Types and Pipelines

As we've been discussing, Java stream operations are divided into intermediate and terminal operations.

Intermediate operations such as *filter()* return a new stream on which further processing can be done. Terminal operations, such as *forEach()*, mark the stream as consumed, after which point it can no longer be used further.

A stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.

Here's a sample stream pipeline, where *empList* is the source, *filter()* is the intermediate operation and *count* is the terminal operation:

```
Long empCount = empList.stream()
  .filter(e -> e.getSalary() > 200000)
  .count();
```

Some operations are deemed **short-circuiting operations**. Short-circuiting operations allow computations on infinite streams to complete in finite time:

```
Stream<Integer> infiniteStream = Stream.iterate(2, i -> i * 2);

List<Integer> collect = infiniteStream
  .skip(3)
  .limit(5)
  .collect(Collectors.toList());
```

Here, we use short-circuiting operations *skip()* to skip first 3 elements, and *limit()* to limit to 5 elements from the infinite stream generated using *iterate()*.

9. sorted

sorts the stream elements based on the comparator passed we pass into it.

For example, we can sort *Employee*s based on their names:

```
List<Employee> employees = empList.stream()
      .sorted((e1, e2) -> e1.getName().compareTo(e2.getName()))
      .collect(Collectors.toList());
```

Note that short-circuiting will not be applied for *sorted()*.

This means, in the example above, even if we had used *findFirst()* after the *sorted()*, the sorting of all the elements is done before applying the *findFirst()*. This happens because the operation cannot know what the first element is until the entire stream is sorted.

10.min *and* max

As the name suggests, *min()* and *max()* return the minimum and maximum element in the stream respectively, based on a comparator.

They return an *Optional* since a result may or may not exist (due to, say, filtering):

```
Employee firstEmp = empList.stream()
      .min((e1, e2) -> e1.getId() - e2.getId())
      .orElseThrow(NoSuchElementException::new);
```

We can also avoid defining the comparison logic by using *Comparator.comparing()*:

```
Employee maxSalEmp = empList.stream()
  .max(Comparator.comparing(Employee::getSalary))
  .orElseThrow(NoSuchElementException::new);
```

11. distinct

distinct() does not take any argument and returns the distinct elements in the stream, eliminating duplicates. It uses the *equals()* method of the elements to decide whether two elements are equal or not:

```
List<Integer> intList = Arrays.asList(2, 5, 3, 2, 4, 3);
List<Integer> distinctIntList = intList.stream()
                                        .distinct().collect(Collectors.toList());
```

12. allMatch, anyMatch, *and* noneMatch

These operations all take a predicate and return a boolean. Short-circuiting is applied and processing is stopped as soon as the answer is determined:

```
List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);

  boolean allEven = intList.stream().allMatch(i -> i % 2 == 0);
  boolean oneEven = intList.stream().anyMatch(i -> i % 2 == 0);
  boolean noneMultipleOfThree = intList.stream().noneMatch(i -> i % 3 == 0);
```

Note:-

allMatch() checks if the predicate is true for all the elements in the stream. Here, it returns *false* as soon as it encounters 5, which is not divisible by 2.

anyMatch() checks if the predicate is true for any one element in the stream. Here, again short-circuiting is applied and *true* is returned immediately after the first element.

noneMatch() checks if there are no elements matching the predicate. Here, it simply returns *false* as soon as it encounters 6, which is divisible by 3.

13. Java Stream Specializations

From what we discussed so far, *Stream* is a stream of object references. However, there are also the *IntStream*, *LongStream*, and *DoubleStream* – which are primitive specializations for *int*, *long* and *double* respectively. These are quite convenient when dealing with a lot of numerical primitives.

These specialized streams do not extend *Stream* but extend *BaseStream* on top of which *Stream* is also built.

As a consequence, not all operations supported by *Stream* are present in these stream implementations. For example, the standard *min()* and *max()* take a comparator, whereas the specialized streams do not.

Creation

The most common way of creating an *IntStream* is to call *mapToInt()* on an existing stream:

```
 Integer latestEmpId = empList.stream()
      .mapToInt(Employee::getId)
      .max()
      .orElseThrow(NoSuchElementException::new);
```

Here, we start with a *Stream<Employee>* and get an *IntStream* by supplying the *Employee::getId* to *mapToInt*. Finally, we call *max()* which returns the highest integer.

We can also use *IntStream.of()* for creating the *IntStream*:

```
IntStream.of(1, 2, 3);
```

or *IntStream.range()*:

```
IntStream.range(10, 20)
```

which creates *IntStream* of numbers 10 to 19.

One important distinction to note before we move on to the next topic:

```
Stream.of(1, 2, 3)
```

This returns a *Stream<Integer>* and not *IntStream*.

Similarly, using *map()* instead of *mapToInt()* returns a *Stream<Integer>* and not an *IntStream*.:

```
empList.stream().map(Employee::getId);
```

14.Specialized Operations

Specialized streams provide additional operations as compared to the standard *Stream* – which are quite convenient when dealing with numbers.

For example *sum(), average(), range()* etc:

```
Double avgSal = empList.stream()
  .mapToDouble(Employee::getSalary)
  .average()
  .orElseThrow(NoSuchElementException::new);
```

15.Reduction Operations

A reduction operation (also called as fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.

We already saw few reduction operations like *findFirst(), min()* and *max*().

Let's see the general-purpose *reduce()* operation in action.

reduce

The most common form of *reduce()* is:

T reduce(T identity, BinaryOperator<T> accumulator)
where *identity* is the starting value and *accumulator* is the binary operation we repeated apply.

For example:

```
Double sumSal = empList.stream()
  .map(Employee::getSalary)
  .reduce(0.0, Double::sum);
```

Here, we start with the initial value of 0 and repeated apply *Double::sum()* on elements of the stream. Effectively we've implemented the *DoubleStream.sum()* by applying *reduce()* on *Stream*.

15.Advanced *collect*

We already saw how we used *Collectors.toList()* to get the list out of the stream. Let's now see few more ways to collect elements from the stream.

joining

```
    String empNames = empList.stream()
      .map(Employee::getName)
      .collect(Collectors.joining(", "))
      .toString();

    assertEquals(empNames, "Jeff Bezos, Bill Gates, Mark Zuckerberg");
```

Collectors.joining() will insert the delimiter between the two *String* elements of the stream. It internally uses a *java.util.StringJoiner* to perform the joining operation.

toSet

We can also use *toSet()* to get a set out of stream elements:

```
@Test
public void whenCollectBySet_thenGetSet() {
    Set<String> empNames = empList.stream()
            .map(Employee::getName)
            .collect(Collectors.toSet());

    assertEquals(empNames.size(), 3);
}
```

toCollection

We can use *Collectors.toCollection()* to extract the elements into any other collection by passing in a *Supplier<Collection>*. We can also use a constructor reference for the *Supplier*:

```
@Test
public void whenToVectorCollection_thenGetVector() {
    Vector<String> empNames = empList.stream()
            .map(Employee::getName)
            .collect(Collectors.toCollection(Vector::new));

    assertEquals(empNames.size(), 3);
}
```

Here, an empty collection is created internally, and its *add()* method is called on each element of the stream.

partitioningBy

We can partition a stream into two – based on whether the elements satisfy certain criteria or not.

Let’s split our List of numerical data, into even and ods:

```
@Test
public void whenStreamPartition_thenGetMap() {
    List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);
    Map<Boolean, List<Integer>> isEven = intList.stream().collect(
      Collectors.partitioningBy(i -> i % 2 == 0));

    assertEquals(isEven.get(true).size(), 4);
    assertEquals(isEven.get(false).size(), 1);
}
```

Here, the stream is partitioned into a Map, with even and odds stored as true and false keys.

groupingBy

groupingBy() offers advanced partitioning – where we can partition the stream into more than just two groups.

It takes a classification function as its parameter. This classification function is applied to each element of the stream.

The value returned by the function is used as a key to the map that we get from the *groupingBy* collector:

```
@Test
public void whenStreamGroupingBy_thenGetMap() {
    Map<Character, List<Employee>> groupByAlphabet = empList.stream()
      .collect(Collectors.groupingBy(e -> new Character(e.getName().charAt(0))));

    assertEquals(groupByAlphabet.get('B').get(0).getName(), "Bill Gates");
    assertEquals(groupByAlphabet.get('J').get(0).getName(), "Jeff Bezos");
    assertEquals(groupByAlphabet.get('M').get(0).getName(), "Mark
Zuckerberg");
}
```

In this quick example, we grouped the employees based on the initial character of their first name.

mapping

groupingBy() discussed in the section above, groups elements of the stream with the use of a *Map*.

However, sometimes we might need to group data into a type other than the element type.

Here's how we can do that; we can use *mapping()* which can actually adapt the collector to a different type – using a mapping function:

```
@Test
public void whenStreamMapping_thenGetMap() {
   Map<Character, List<Integer>> idGroupedByAlphabet =
empList.stream().collect(
     Collectors.groupingBy(e -> new Character(e.getName().charAt(0)),
       Collectors.mapping(Employee::getId, Collectors.toList())));

   assertEquals(idGroupedByAlphabet.get('B').get(0), new Integer(2));
   assertEquals(idGroupedByAlphabet.get('J').get(0), new Integer(1));
   assertEquals(idGroupedByAlphabet.get('M').get(0), new Integer(3));
}
```

Here *mapping()* maps the stream element *Employee* into just the employee id – which is an *Integer* – using the *getId()* mapping function. These ids are still grouped based on the initial character of employee first name.

reducing

reducing() is similar to *reduce()* – which we explored before. It simply returns a collector which performs a reduction of its input elements:

```
@Test
public void whenStreamReducing_thenGetValue() {
   Double percentage = 10.0;
   Double salIncrOverhead = empList.stream().collect(Collectors.reducing(
      0.0, e -> e.getSalary() * percentage / 100, (s1, s2) -> s1 + s2));

   assertEquals(salIncrOverhead, 60000.0, 0);
}
```

Here *reducing()* gets the salary increment of each employee and returns the sum.

reducing() is most useful when used in a multi-level reduction, downstream of *groupingBy()* or *partitioningBy()*. To perform a simple reduction on a stream, use *reduce()* instead.

For example, let’s see how we can use *reducing()* with *groupingBy()*:

```
@Test
public void whenStreamGroupingAndReducing_thenGetMap() {
    Comparator<Employee> byNameLength =
Comparator.comparing(Employee::getName);

    Map<Character, Optional<Employee>> longestNameByAlphabet =
empList.stream().collect(
      Collectors.groupingBy(e -> new Character(e.getName().charAt(0)),
        Collectors.reducing(BinaryOperator.maxBy(byNameLength))));

    assertEquals(longestNameByAlphabet.get('B').get().getName(), "Bill Gates");
    assertEquals(longestNameByAlphabet.get('J').get().getName(), "Jeff Bezos");
    assertEquals(longestNameByAlphabet.get('M').get().getName(), "Mark
Zuckerberg");
}
```

Here we group the employees based on the initial character of their first name. Within each group, we find the employee with the longest name.

Parallel Streams

Using the support for parallel streams, we can perform stream operations in parallel without having to write any boilerplate code; we just have to designate the stream as parallel:

```
@Test
public void whenParallelStream_thenPerformOperationsInParallel() {
    Employee[] arrayOfEmps = {
      new Employee(1, "Jeff Bezos", 100000.0),
      new Employee(2, "Bill Gates", 200000.0),
      new Employee(3, "Mark Zuckerberg", 300000.0)
    };

    List<Employee> empList = Arrays.asList(arrayOfEmps);

    empList.stream().parallel().forEach(e -> e.salaryIncrement(10.0));

    assertThat(empList, contains(
      hasProperty("salary", equalTo(110000.0)),
      hasProperty("salary", equalTo(220000.0)),
      hasProperty("salary", equalTo(330000.0))
    ));
}
```

Here *salaryIncrement()* would get executed in parallel on multiple elements of the stream, by simply adding the *parallel()* syntax.

This functionality can, of course, be tuned and configured further, if you need more control over the performance characteristics of the operation.

As is the case with writing multi-threaded code, we need to be aware of few things while using parallel streams:

1. We need to ensure that the code is thread-safe. Special care needs to be taken if the operations performed in parallel modifies shared data.
2. We should not use parallel streams if the order in which operations are performed or the order returned in the output stream matters. For example operations like *findFirst()* may generate the different result in case of parallel streams.
3. Also, we should ensure that it is worth making the code execute in parallel. Understanding the performance characteristics of the operation in particular, but also of the system as a whole – is naturally very important here.

File Operations

Let's see how we could use the stream in file operations.

File Write Operation

```
@Test
public void whenStreamToFile_thenGetFile() throws IOException {
    String[] words = {
      "hello",
      "refer",
      "world",
      "level"
    };

    try (PrintWriter pw = new PrintWriter(
      Files.newBufferedWriter(Paths.get(fileName)))) {
        Stream.of(words).forEach(pw::println);
    }
}
```

Here we use *forEach()* to write each element of the stream into the file by calling *PrintWriter.println()*.

File Read Operation

```
private List<String> getPalindrome(Stream<String> stream, int length) {
    return stream.filter(s -> s.length() == length)
      .filter(s -> s.compareToIgnoreCase(
        new StringBuilder(s).reverse().toString()) == 0)
      .collect(Collectors.toList());
}

@Test
public void whenFileToStream_thenGetStream() throws IOException {
    List<String> str = getPalindrome(Files.lines(Paths.get(fileName)), 5);
    assertThat(str, contains("refer", "level"));
}
```

Here *Files.lines()* returns the lines from the file as a *Stream* which is consumed by the *getPalindrome()* for further processing.

getPalindrome() works on the stream, completely unaware of how the stream was generated. This also increases code reusability and simplifies unit testing.

5. Sort number in ascending order using Stream api.

```
package com.abc.sort;

import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

public class SortingUsingStream {
        public static List<Integer> number(){
                return List.of(11,2,3,45,23,47,23,12,54,21,24,27);
        }

        public static void main(String[] args) {
                List<Integer> numbers = number();

                //ascending order
                List<Integer> sort = numbers.stream()
                                                        .sorted()
                                                        .collect(Collectors.toList());
                System.out.println(sort);
```

```
                    //descending order
                    List<Integer> reversesort = numbers.stream()
                                                        .sorted(Collections.reverseOrder())
                                                        .collect(Collectors.toList());
                    System.out.println(reversesort);

                    //array in descending order
                      Integer[] array = numbers.stream()
                              .sorted(Collections.reverseOrder())
                              .toArray(Integer[]::new);
                    System.out.println(array[0]);
          }

  }
```

Output:-

```
[2, 3, 11, 12, 21, 23, 23, 24, 27, 45, 47, 54]
[54, 47, 45, 27, 24, 23, 23, 21, 12, 11, 3, 2]
54
```

6. Find the frequency of each character using stream api.

```
package com.abc.frequencyofeachchar;

import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;

public class FrequencyOfEachCharUsingStream {

          public static void main(String[] args) {
                    String str = "aaaabddegkshfhaajjjjjzzzzz";


//1st approach to count frequency of char using map function
Map<Character, Integer> frequency1 = str.chars()
                    .mapToObj(c -> (char) c)
                    .collect(Collectors.toMap(Function.identity(),c -> 1,Integer::sum));
                    System.out.println(frequency1);

//2st approach to count frequency of char using grouping function
Map<Character, Integer> frequency2 = str.chars().mapToObj(c -> (char) c)
                                        .collect(Collectors.groupingBy(Function.identity(),
Collectors.summingInt(c -> 1)));

                    /this is to get highest repeated char if frequency same then based on ascii
                    int max = 0;
```

```
            char ch = ' ';
            for (Map.Entry<Character, Integer> entry : frequency2.entrySet()) {
                System.out.println(entry.getKey() + " ==> " + entry.getValue());
                if (entry.getValue() >= max) {
                    if ((int) entry.getKey() > (int)ch) {
                        max = entry.getValue();
                    ch = entry.getKey();
                    }
                }
            }
            System.out.println(ch + " ===> " + max);
            System.out.println(getNumChar(str));

        }

        //3rd approach without using any map
        public static char getNumChar(String s) {
            char[] c = s.toCharArray();
            String alphabet = "abcdefghijklmnopqrstuvwxyz";
            int[] countArray = new int[26];
            for (char x : c) {
                for (int i = 0; i < alphabet.length(); i++) {
                    if (alphabet.charAt(i) == x) {
                        countArray[i]++;
                    }
                }
            }

    HashMap<Integer, Character> countList = new HashMap<Integer, Character>();

            for (int i = 0; i < 26; i++) {
                countList.put(countArray[i], alphabet.charAt(i));
            }
            java.util.Arrays.sort(countArray);
            int max = countArray[25];
            return countList.get(max);
        }



    }
```
Output:-

```
{a=6, b=1, s=1, d=2, e=1, f=1, g=1, h=2, z=5, j=5, k=1}
a ==> 6
b ==> 1
s ==> 1
d ==> 2
```

```
e ==> 1
f ==> 1
g ==> 1
h ==> 2
z ==> 5
j ==> 5
k ==> 1
a ===> 6
a
```

7. Removing duplicate character using stream api.

```
package com.abc.frequencyofeachchar;

import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class RemoveDuplicateCharacterUsingStream {

        public static void main(String[] args) {
                String str = "aaabddegkshfhjhhhd";
                Set<Character> collect = str.chars()
                                            .mapToObj(c -> (char) c)
                                            .collect(Collectors.toSet());
        System.out.println(collect.toString());

        //2nd approach using distinct method
        List<Character> collect1 = str.chars()
                                   .mapToObj(c -> (char) c)
                                   .distinct()
                                   .collect(Collectors.toList());
        System.out.println(collect1);
        }
}
Output:-
[a, b, s, d, e, f, g, h, j, k]
[a, b, d, e, g, k, s, h, f, j]
```

8.Remove duplicate numbers and find min and max in list using stream api.

```
List<Integer> numbersList = new ArrayList<>(Arrays.asList(1,2,1,34,45,5,7,
1, 2, 3, 3, 3, 4, 5, 6, 6,9, 6, 7, 8));
List<Integer> collect = numbersList.stream()
                                   .sorted()
                                   .distinct()
                                   .collect(Collectors.toList());
System.out.println(collect);
System.out.println(collect.get(0)+" "+collect.get(collect.size()-1));
```

Output:-

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 34, 45]

1 45
```

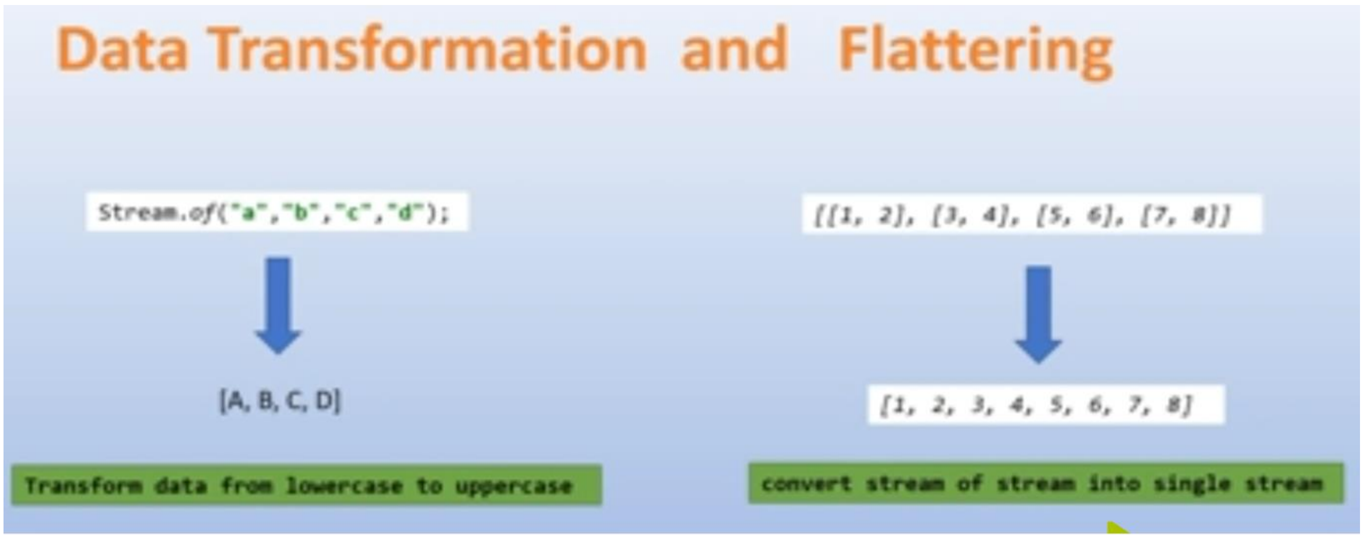
9. Demonstrate difference between map and flat map.

Java 8 stream API provides map() and flatMap() method. Both these methods are intermediate methods and returns another stream as part of the output.

• map() method used for transformation

• flatMap() used for transformation & flattering

• flatMap() → map() + flattering


Data Transformation and Flattering
Stream.of("a","b","c","d");
[A, B, C, D]
Transform data from lowercase to uppercase
[[1, 2], [3, 4], [5, 6], [7, 8]]
[1, 2, 3, 4, 5, 6, 7, 8]
convert stream of stream into single stream

Real time examples

Let's assume that we have a list of Employes and we need to get the list of cities where they have worked in past years.

Employee.java:-

```
package com.abc.entity;

import java.util.List;
@Data
public class Employee {
      private Integer id;
      private String name;
      private List<String> city;
      }
```

EmployeeDao.java:-

```
package com.abc.entity;

import java.util.List;
```

```
public class EmployeeDao {

	public static List<Employee> getData(){
Employee employee1 = new Employee(1,"Rustam",List.of("Delhi","Banglore","Chennai"));
Employee employee2 = new Employee(2,"Saddam",List.of("Pune","Banglore","Kolkata"));
Employee employee3 = new Employee(3,"Atul",List.of("Delhi","Banglore","Noyda"));
		return List.of(employee1,employee2,employee3);
	}
}
```

MapFlatMap.java:-

```
package com.abc.entity;

import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class Map_FlatMap_Controller {

	public static void main(String[] args) {
		List<Employee> data = EmployeeDao.getData();
		System.out.println(data);

		// uses of map
		List<Integer> id = data.stream()
						.map(Employee::getId)
						.collect(Collectors.toList());
		System.out.println(id);

		// uses of flatMap
		Set<List<String>> city = data.stream()
							.map(Employee::getCity)
							.collect(Collectors.toSet());
	   System.out.println(city); // stream of stream we have to flat it.
		// with flatMap
		Set<String> cities = data.stream()
				.flatMap(e -> e.getCity().stream())
				.collect(Collectors.toSet());
		System.out.println(cities);
	}
}
```

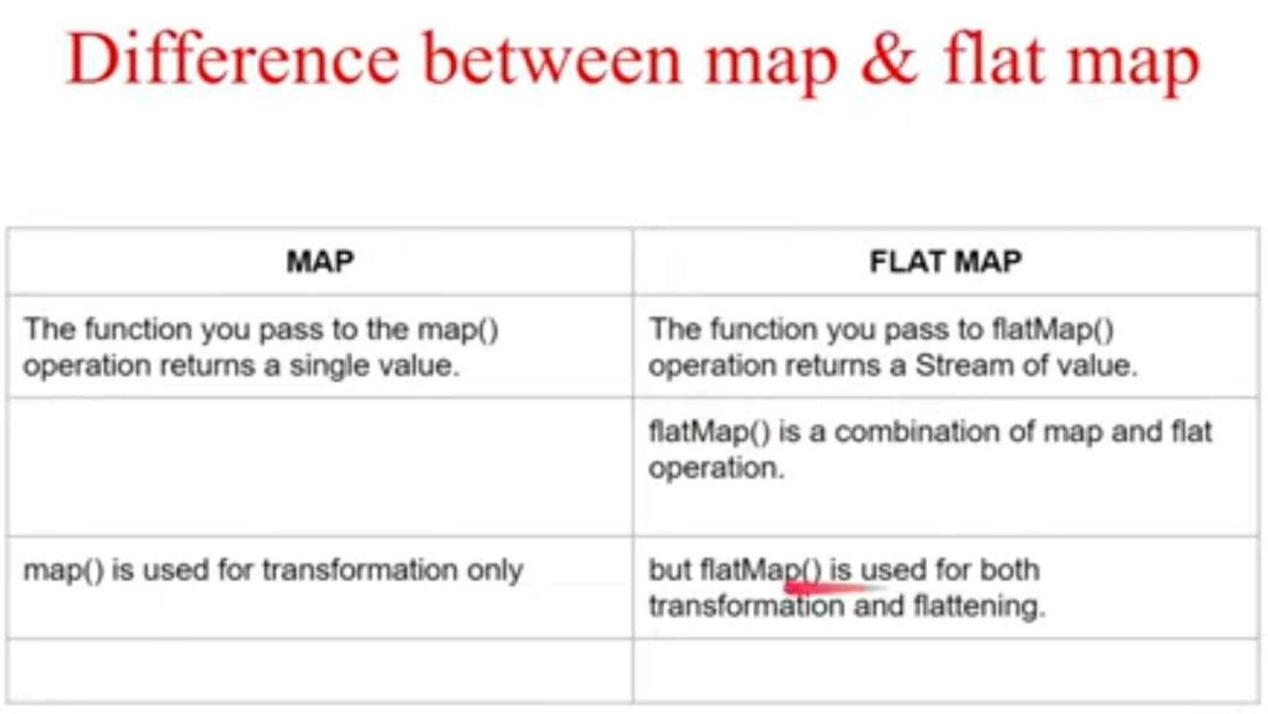
Output:-

```
[
Employee [id=1, name=Rustam, city=[Delhi, Banglore, Chennai]],
Employee [id=2, name=Saddam, city=[Pune, Banglore, Kolkata]],
Employee [id=3, name=Atul, city=[Delhi, Banglore, Noyda]]
]

[1, 2, 3]

[[Delhi, Banglore, Noyda], [Delhi, Banglore, Chennai], [Pune, Banglore,
Kolkata]]

[Delhi, Banglore, Chennai, Pune, Kolkata, Noyda]
```



Difference between map & flat map

MAP	FLAT MAP
The function you pass to the map() operation returns a single value.	The function you pass to flatMap() operation returns a Stream of value.
	flatMap() is a combination of map and flat operation.
map() is used for transformation only	but flatMap() is used for both transformation and flattening.

10. What is immutable class . How to write customize immutable class .

Immutable class in java means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable.

Following are the requirements:

- The class must be declared as final so that child classes can’t be inherited.
- Data members in the class must be declared **private and final** so that direct access is not allowed and can’t change the value of it after object creation.
- A parameterized constructor should initialize all the fields performing a deep copy so that data members can’t be modified with an object reference.

- Deep Copy of objects should be performed in the getter methods to return a copy rather than returning the actual object reference)
- Setter not be there in immutable class.

Ex:-

Depertment.java:-

```
package com.abc.immutable;

public class Depertment {

	private String name;
	private String location;

	public Depertment() {
		}

	public Depertment(String name,String location) {
		this.name=name;
		this.location=location;
	}

	//using copy constructor
	public Depertment(Depertment depertment) {
		this(depertment.getName(),depertment.getLocation());
	}

	public String getName() {
		return name;
	}

	public String getLocation() {
		return location;
	}

	public void setLocation(String location) {
		this.location = location;
	}

	public void setName(String name) {
		this.name = name;
	}

	@Override
	public String toString() {
	return "Depertment [name=" + name + ", location=" + location + "]";
	}

}
```

ImmutableEmployee.java:-

```
package com.abc.immutable;


final public class ImmutableEmployee {
        private final Integer id;
        private final String name;
        private final Depertment dept;

        public ImmutableEmployee() {
                this.id = null;
                this.name = "";
                this.dept = new Depertment();
        }

        public ImmutableEmployee(Integer id, String name,Depertment dept) {
                super();
                this.id = id;
                this.name = name;
                this.dept = dept;
        }
        public Integer getId() {
                return id;
        }

        public String getName() {
                return name;
        }

        public Depertment getDept() {
//              return dept;  don't send actual object reference
//              return new Depertment(dept.getName(),dept.getLocation());
                return new Depertment(this.dept);
        }

        @Override
        public String toString() {
return "ImmutableEmployee [id=" + id + ", name=" + name + ", dept=" + dept
+ "]";
        }

}
```

Main.java:-

```
package com.abc.immutable;

public class Main {

        public static void main(String[] args) {
                Depertment depertment = new Depertment();
                depertment.setName("tech");
                depertment.setLocation("banglore");
ImmutableEmployee employee = new ImmutableEmployee(10,"rustam",depertment);
                System.out.println(employee);
//              employee.setName("Ali");

                Depertment dept = employee.getDept();
                System.out.println(dept);
                System.err.println(depertment==dept);
                dept.setName("HR"); //even we trying to change its not change
                dept.setLocation("kolkata");
                System.err.println(dept);
                System.out.println(employee);
        }
    }
```

Output:-

```
ImmutableEmployee [id=10, name=rustam, dept=Depertment [name=tech,
location=banglore]]
false
Depertment [name=HR, location=kolkata]
Depertment [name=tech, location=banglore]
ImmutableEmployee [id=10, name=rustam, dept=Depertment [name=tech,
location=banglore]]
```

11. What is Singleton. What are the ways to break singleton pattren. How to write a code for single pattren .

Singleton is a design pattern that ensures that a class can only have one object.

To create a singleton class, a class must implement the following properties:

- Create a **private constructor** of the class to restrict object creation outside of the class.
- Create a **private static attribute** of the class type that refers to the single object.
- Create a **public static method** that allows us to create and access the object we created. Inside the method, we will create a condition that restricts us from creating more than one object.

Syntax:-

```
class SingletonExample {

   // private field that refers to the object
   private static SingletonExample singleObject;

   private SingletonExample() {
      // constructor of the SingletonExample class
   }

   public static SingletonExample getInstance() {
      // write code that allows us to create only one object
      // access the object as per our need
   }
}
```

Usage:-

Singletons can be used while working with databases. To make Connection .

There are two forms of singleton design patterns, which are:

Early Instantiation: The object creation takes place at the load time.

Lazy Instantiation: The object creation is done according to the requirement.

2.There are mainly 3 concepts that can break the singleton property of a class.

- **Through reflection Api.**
- **Through Serailzation and Deserailzation.**
- **Through Cloning.**
- **Through MultiThreading.**

Way To create SingleTon Object:-

LazySingleTon.java:-

```
import java.io.Serializable;

/**
Lazy initialization mean application will create instance when it is
requested.
However, this can be used when you have non-thread-safe application.
If used in multi threading it might break,Why?
because your getinstance method if invoked by two thread at same time
then!!!!
 *
When to use?
Non thread safe and creating common resource like db connection.
 *
 *
 */
public class LazySingleton implements Serializable {
    private static LazySingleton instance = null;   //lazy loading

    private LazySingleton() {

    }

    public static LazySingleton getInstance() {
        if(instance == null){
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

EagerSingleTon.java:-

```
/**
This is eager initialization concept where as soon as JVM start the object
will be created irrespective whether it got accessed by any code in
application or not.
  When to use :
One possible usage can be let say your application has some static cache
which is required to be loaded.
  Drawback :
As mention consumes resource even if application does not use it.
 */


public class EagerSingleton {
    private static final EagerSingleton instance = new EagerSingleton();
    private EagerSingleton() {
        // Do your init work here
    }
    public static  EagerSingleton getInstance() {
        return instance;
    }
}
```

Way to handle to avoid to break SingleTon

MultiThreadSingleTon.java:-

```
/**
As mentioned in Lazy Initialization if our get instance method invoked by
two thread at the same time then there are chances that two objects
created, and we violate singleton pattern.
 * To avoid we have two choices :
 * 1. Create getinstance synchronized so that only one instance can invoke
      that method.
 *  However, disadvantage is lets say
there are 3 thread t1 is inside getinstance and t2,t3 waiting. Now t2 will
get into method and simply return instance created by t1 and t3 still
waiting. So it had lead to unnecessary of locks.
 *
 * 2. To avoid we have synchronized block which we will implement here.
 */
public class MultithreadSingleton {
    private static MultithreadSingleton instance = null;
    private MultithreadSingleton() {

    }

    public static MultithreadSingleton getInstance() {
        // Question arise why we have two null check here.
        // Reason for first null check is same as explained in method level
           synchronization why create lock if our object is already
           created.
        if(instance == null) {
            // Our method is static, so we have class level locking here
            synchronized (MultithreadSingleton.class) {
                //Reason for second null check is lets say two thread are
                  come inside first null at same time
// One call has taken lock and proceeds for creating object first time. Now
once lock is released for t1 t2 should not create object because its
already created and that's why we have second null check.
                if(instance == null) {
                    instance = new MultithreadSingleton();
                }
            }
        }
        return instance;
    }
}
```

SerializableSingleTon.java:-

```
import java.io.Serializable;

/**
Let's say your singleton has implemented serialization. Now what will
happen if you serialize object and deserialize.
 * During deserialization it will create the new object every time if we go
   in traditional way.
 * To resolve it add readResolve method which will ensure that during
   deserialize we return same instance.
 *
 * Check Main class for violation example
 */
public class SerializableSingleton implements Serializable {
    private static  SerializableSingleton instance = null;
    private SerializableSingleton() {

    }

    public static SerializableSingleton getInstance() {
        if(instance == null) {
            instance = new SerializableSingleton();
        }
        return instance;
    }

    /**
This is the key method which is responsible during deserialization process
This method get invoked, and we are simply returning already created object
     * @return
     */
    protected Object readResolve() {
        return instance;
    }
}
```

EnumSingleTon.java:-

```
/**
 * Here we described creating singleton using enum but why?
 * Let's say in Lazy Init method you access constructor by reflection
(Reason being you can access private constructor using reflection!!)
 * and create the object which eventually creates the problem by having
multiple instances.
 *
ENUM's constructor gets invoked by JVM not by User who is using so it is
safe to use.
Another advantage of using enum is , we don't need to worry about threads
as it is thread safe.
It also solved the problem of Serialization as JVM takes care to return
same object.
 *
 */
public enum EnumSingleton {
    INSTANCE;
    public void doSomething() {
        System.out.println("Cool");
    }
}
```

Main.java:-

```
import java.io.*;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

//Before jumping into pattern just explain what is lazy loading and eager loading
// Mainly this class used to show violations using serializable and reflection.
public class Main {
    public static void main(String[] args) throws IOException,
ClassNotFoundException, InvocationTargetException, InstantiationException,
IllegalAccessException {

        exampleSerialization();
       exampleReflection();

    }
    private static void exampleSerialization() throws IOException,
ClassNotFoundException {

        LazySingleton lazySingleton = LazySingleton.getInstance();
        LazySingleton lazySingleton2 = LazySingleton.getInstance();  //it
will return same object ref
System.out.println("Object 1 :" + lazySingleton.hashCode()+" Object 2 :" +
lazySingleton2.hashCode());
        // due to serialization and deserialization object get changed
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(new
FileOutputStream("object.obj"));
        objectOutputStream.writeObject(lazySingleton);
        objectOutputStream.close();
        ObjectInputStream objectInputStream = new ObjectInputStream(new
FileInputStream("object.obj"));
        LazySingleton deserializedLazy = (LazySingleton)
objectInputStream.readObject();
        objectInputStream.close();
        System.out.println("Object 1 :" + lazySingleton.hashCode());
        System.out.println("Object 2 :" + deserializedLazy.hashCode());

//but here not changed because of override readResolve method because
  during deserialized jvm will automatic call this method
         SerializableSingleton serializableSingleton =
      SerializableSingleton.getInstance();
        ObjectOutputStream objectOutputStream2 = new ObjectOutputStream(new
FileOutputStream("object1.obj"));
        objectOutputStream2.writeObject(serializableSingleton);
        objectOutputStream2.close();
        ObjectInputStream objectInputStream2 = new ObjectInputStream(new
FileInputStream("object1.obj"));
        SerializableSingleton deserializedInstance =
(SerializableSingleton) objectInputStream2.readObject();
        objectInputStream2.close();
        System.out.println("SerializableSingleton Object 1 :" +
serializableSingleton.hashCode());
        System.out.println(" SerializableSingleton Object 2 :" +
deserializedInstance.hashCode());
    }

    private static void exampleReflection() throws
InvocationTargetException, InstantiationException, IllegalAccessException {
       Constructor[] constructors =
LazySingleton.class.getDeclaredConstructors();
       //Knowing only one constructor taking it using index
        Constructor constructor = constructors[0];
        constructor.setAccessible(true);
  LazySingleton lazySingleton = (LazySingleton) constructor.newInstance();
        LazySingleton instance = LazySingleton.getInstance();
        System.out.println("Reflected hashcode singleton
:"+lazySingleton.hashCode());
        System.out.println("Singleton instance : "+ instance.hashCode());

        //Solution to this is go by enum
        EnumSingleton.INSTANCE.doSomething();
    }
}
```

Output:-

```
Object 1 :1365202186 Object 2 :1365202186
Object 1 :1365202186
Object 2 :1597462040
SerializableSingleton Object 1 :1576861390
SerializableSingleton Object 2 :1576861390
Reflected hashcode singleton :600746945
Singleton instance : 1365202186
Cool
```

12. What is java refelction api.

Java Reflection is a *process of examining or modifying the run time behavior of a class at run time.*

Pros and Cons of Reflection

Pros: Inspection of interfaces, classes, methods, and fields during runtime is possible using reflection, even without using their names during the compile time. It is also possible to call methods, instantiate a clear or to set the value of fields using reflection. It helps in the creation of Visual Development Environments and class browsers which provides aid to the developers to write the correct code.

Cons: Using reflection, one can break the principles of encapsulation. It is possible to access the private methods and fields of a class using reflection. Thus, reflection may leak important data to the outside world, which is dangerous. For example, if one access the private members of a class and sets null value to it, then the other user of the same class can get the NullReferenceException, and this behaviour is not expected.

13. What is concurrent hash map and how it works internally

a. Traditional Collection Object like ArrayList,LinkedList,HashSet,HashMap is accessed by Multiple Threads simultaneously and there may be a chance of "DataIncosistencyProblem". Since they are accessed by multiple threads simultaneously they are not "ThreadSafe".

b. Already Existed Collection Objects like a Vector,Hashtable ,synchronizedList() ,synchronizedSet(),synchronizedMap() are "ThreadSafe", but performance is very low as the locking mechanism is not good.

c. In traditional collection if one thread iterates and other tries to modify **structural** change then ConcurrentModificationException is thrown.

Becoz of the above mentioned problem,these collection objects are not suitable for "MultiThreadingEnvironment".To resolve this problem SUNMS introduced a new Set of Collection called "ConcurrentCollection" in JDK1.5.

ConcurrentCollection classes

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

1. ConcurrentHashMap
2. CopyOnWriteArrayList
3. CopyOnWriteArraySet

ConcurrentHashMap

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

- **Underlying datastructure** => Hashtable
- **locking mechanism** => not on the entire object, it is on part of object(bucket level)
- **Read Operation** => Mulitple threads can read no lock required
- **Update Operation** => will be decided by bucket level/concurrency level.
- **DefaultConcurrency level** => 16(can be changed as per user requirement).
- **null** => not allowed as both key and value
- **ConcurrentModification** => yes it is possible,no Excpetion

ConcurrentCollection

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

```
  Map(I)
    |
    |extends
    |
ConcurrentMap(I)
    |
    |implements
    |
ConcurrentHashMap(C) => No ConcurrentModificationException
       putIfAbsent(k,v)
       remove(k,v)
       replace(k,v,v)
```

Types of Constructor

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

ConcurrentHashMap hmp=ConcurrentHashMap();
ConcurrentHashMap hmp=ConcurrentHashMap(int capacity);
ConcurrentHashMap hmp=ConcurrentHashMap(int capacity,int fillratio);
ConcurrentHashMap hmp=ConcurrentHashMap(int capacity,int fillratio ,int concurrencylevel);
ConcurrentHashMap hmp=ConcurrentHashMap(Map m);

Ex:-
```
package com.fullcreative.staticprogram;

import java.util.HashMap;
import java.util.Iterator;

public class ConcurrentHashMapExample {
     public static void main(String[] args) {
          HashMap<Integer, Integer> map = new HashMap<>();
          map.put(1, 1);
          map.put(2, 2);
          map.put(3, 3);
          Iterator<Integer> it = map.keySet().iterator();
          while (it.hasNext()) {
               Integer key = it.next();
               if (key.equals(2)) {
                    map.put(4, 4);  //structure got changed
               }
          System.out.println("Map Value:" + map.get(key));
          }
     }
}
```
Output:-
```
Map Value:1
Map Value:2
Exception in thread "main" java.util.ConcurrentModificationException
	at 
java.base/java.util.HashMap$HashIterator.nextNode(HashMap.java:1597)
	at java.base/java.util.HashMap$KeyIterator.next(HashMap.java:1620)
	at 
com.fullcreative.staticprogram.ConcurrentHashMapExample.main(ConcurrentHash
MapExample.java:14)
```

Note:-
```
if (key.equals(2)) {
map.put(4, 4);
//if you give existing key then structure not get changed and it
will not show error like map.put(2,7); here 2 is exits so its just
update the value. structure will not change  so no conncurrent
modifiaction show .
in case of remove also its show conncurrent modifiaction Exception.
}
```

The limitation of Traditional Collection is overcomed in ConcurrentCollections

1. Concurrent collection is ThreadSafe.
2. Performance of ConurrentCollection is better than NormalCollection becoz of different locking mechansim.
3. In case of ConcurrentCollection simultaneously read and update operation can be performed so their is no "ConcurrentModificationException".

Using ConcurrentHashMap :-

```
package com.fullcreative.staticprogram;

import java.util.Iterator;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {
	public static void main(String[] args) {
	ConcurrentHashMap <Integer, Integer> map = new ConcurrentHashMap<>();
		map.put(1, 1);
		map.put(2, 2);
		map.put(3, 3);
		Iterator<Integer> it = map.keySet().iterator();
		while (it.hasNext()) {
			Integer key = it.next();
			System.out.println("Map Value:" + map.get(key));
			if (key.equals(2)) {
				map.put(4, 4);
			}
		}
	}
}
```

Output:-

```
Map Value:1
Map Value:2
Map Value:3
Map Value:4
```

Here Structure got change length of map now become 4.

It has 3 methods

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

public Object putIfAbsent(Object key,Object value);

//put the entry into the Map,only if key is not present. if key already exists then it wont replace the value associated with new key.

public boolean remove(Object key, Object value);

//remove would delete a particular entry only if key and value is matched, if value is not matched with a key then that particular entry will not be remove.

public boolean replace(Object key, Object value,Object value);

//it will replace the value associated with a key only if both key and value is matched otherwise it wont replace.

Ex:-

```
import java.util.concurrent.*;

class ConcurrentHashMapDemo {

	public static void main(String[] args)
```

```
      {
ConcurrentHashMap<Integer, String> m= new ConcurrentHashMap<>();

            m.put(100, "Hello");
            m.put(101, "Geeks");
            m.put(102, "Geeks");

            // Here we cant add Hello because 101 key
            // is already present in ConcurrentHashMap object
            m.putIfAbsent(101, "Hello");

            // We can remove entry because 101 key
            // is associated with For value
            m.remove(101, "Geeks");

            // Now we can add Hello
            m.putIfAbsent(103, "Hello");

            // We cant replace Hello with For
            m.replace(101, "Hello", "For");
            System.out.println(m);
      }
}
```

Output:-

```
{100=Hello, 102=Geeks, 103=Hello}
```

Difference b/w HashMap and ConcurrentHashMap?

HashMap => Not Thread safe.
ConcurrentHashMap => Thread safe because of differnt locking mechanism(bucket level lock)

HashMap => one thread is iterating and at the same time modification is not possible it would result in "ConcurrentModificationException".
ConcurrentHashMap => one thread is iterating and at the same time modification is possibleit won't result in "ConcurrentModif icationException".

HashMap => Performance is low becoz of more waiting time.
ConcurrentHashMap => Performance is high becoz of less waiting time.

HashMap => Iterator would result in "FailFast".
ConcurrentHashMap => Iterator is of "FailSafe".

HashMap => 1.2v
ConcurrentHashMap => 1.5v

Difference b/w ConcurrentHashMap,synchronizedMap and Hashtable?

ConcurrentHashMap	=> Thread safety by getting lock at the bucket level
SynchronizedMap	=> Thread safety by appling lock at object level
Hashtable	=> Thread safety by appling lock at object level

ConcurrentHashMap	=>Since lock is at bucket level so no ConcurrentModificationException.
SynchronizedMap	=>Since lock is Object level so ConcurrentModificationException.
Hashtable	=>Since lock is at Object level so ConcurrentModificationException.

ConcurrentHashMap	=>Iterator is "FailSafe".
SynchronizedMap	=>Iterator is "FailFast".
Hashtable	=>Iterator is "FailFast".

ConcurrentHashMap	=> Read operation is performed without lock,only for update operation lock is required only at the bucket level (concurrency level).
SynchronizedMap	=> Every read and write operation required lock.
Hashtable	=> Every read and write operation required lock.

ConcurrentHashMap	=>null is not allowed for both key and value.
SynchronizedMap	=>null is allowed for both key and value.
Hashtable	=>null is not allowed for both key and value.

ConcurrentHashMap	=>JDK1.5
SynchronizedMap	=>JDK1.2
Hashtable	=>JDK1.0

14.Difference b/w Serialization and Externalization

Serialization

1. It is meant for default Serialization
2. Here every thing takes care by JVM and programmer doesn't have any control.
3. Here total object will be saved always and it is not possible to save part of the object.
4. Serialization is the best choice if we want to save total object to the file.
5. relatively performence is low.
6. Serializable interface doesn't contain any method.
7. It is a marker interface.
8. Serializable class not required to contains public no-arg constructor.
9. transient keyword play role in serialization

Externalization

1. It is meant for Customized Serialization
2. Here every thing takes care by programmer and JVM does not have any control.
3. Here based on our requirement we can save either total object or part of the object.
4. Externalization is the best choice if we want to save part of the object.
5. relatively performence is high
6. Externalizable interface contains 2 methods :
   1.writeExternal()
   2. readExternal()
7. It is not a marker interface.
8. Externalizable class should compulsory contains public no-arg constructor otherwise we will get RuntimeException saying "InvalidClassException"
9. transient keyword don't play any role in Externalization.