**Java8 Notes by Vikash**

1. **Lambda Expression:**

Lambda Expression is used to provide the implementation of a Functional Interface. Java Lambda expression is treated as Function.

1. **Functional Interface:**

A Functional Interface in Java is an Interface that contains only one abstract method but can contains multiple default and static method is called functional interface.

1. **Consumer Functional Interface:**

Consumer<T> is an in-built Functional Interface introduced in Java 8. Consumer can be used in all contexts where an object needs to be consumed i.e taken as input and some operation is to be performed on the object without returning any result.

**void accept(T t);**

Example:

import java.util.function.Consumer;  
  
public class ConsumerExampleUsingOldMethod implements Consumer<Integer> {  
  
 @Override  
 public void accept(Integer integer){  
 System.*out*.println("The Number = "+integer);  
  
 }  
  
 public static void main(String[] args) {  
 ConsumerExampleUsingOldMethod consumerExampleUsingOldMethod = new ConsumerExampleUsingOldMethod();  
 consumerExampleUsingOldMethod.accept(10);  
  
 }  
}

public class ConsumerExampleUsingLambdaExpression {  
  
 public static void main(String[] args) {  
 *// So basically "(number) -> System.out.println("The Number = "+number)" returning Consumer object* Consumer<Integer> consumer = (number) -> System.*out*.println("The Number = "+number);  
  
 *// When we are passing this function from any other method like forEach then automatically this accept method will get called* consumer.accept(10);  
 }  
}

public class UseOfConsumer {  
  
 public static void main(String[] args) {  
  
 List<Integer> listOfInteger = Arrays.*asList*(1, 2, 3, 4, 5);  
  
 Consumer<Integer> consumer = (number) -> System.*out*.println("The Number = "+number);  
  
 listOfInteger.stream().forEach(consumer);  
 *// Or - In place of consumer we can direct write whole consumer function lambda expression in same foreach  
 //listOfInteger.stream().forEach((number) -> System.out.println("The Number = "+number));* }  
}

1. **Predicate Function Interface:**

This Functional Interface used for conditional check it has only one abstract method

**boolean test(T t);**

Example:

import java.util.function.Predicate;  
  
public class PredicateExampleUsingOldMethod implements Predicate<Integer> {  
  
 @Override  
 public boolean test(Integer integer){  
 *// Checking given number is Odd/Even* return integer % 2 == 0 ? Boolean.*TRUE* : Boolean.*FALSE*;  
 }  
  
 public static void main(String[] args) {  
 PredicateExampleUsingOldMethod predicateExampleUsingOldMethod = new PredicateExampleUsingOldMethod();  
  
 System.*out*.println(predicateExampleUsingOldMethod.test(10));  
 }  
}

public class PredicateExampleUsingLambdaExpression {  
  
 public static void main(String[] args) {  
 Predicate<Integer> evenOddPredicate = (integer) -> integer % 2 == 0 ? Boolean.*TRUE* : Boolean.*FALSE*;  
  
 System.*out*.println(evenOddPredicate.test(10));  
 }  
}

public class UseOfPredicate {  
 public static void main(String[] args) {  
  
 Predicate<Integer> evenOddPredicate = (integer) -> integer % 2 == 0 ? Boolean.*TRUE* : Boolean.*FALSE*;  
  
 List<Integer> listOfInteger = Arrays.*asList*(1, 2, 3, 4, 5);  
  
 listOfInteger.stream().filter(evenOddPredicate).forEach((t) -> System.*out*.println(t));  
 }  
}

1. **Supplier Function Interface:**

Supplier can be used in all contexts where there is no input but an outputs in expected. It has only one Method

**T get();**

Example:

import java.util.function.Supplier;  
  
public class SupplierExampleUsingOldMethod implements Supplier<String> {  
 @Override  
 public String get(){  
 return "This is Vikash";  
 }  
 public static void main(String[] args) {  
 SupplierExampleUsingOldMethod supplierExampleUsingOldMethod = new SupplierExampleUsingOldMethod();  
 System.*out*.println(supplierExampleUsingOldMethod.get());  
 }  
}

public class SupplierExampleUsingLambdaExpression {  
  
 public static void main(String[] args) {  
 *// Method 1* Supplier<String> supplier = () -> {  
 return "This is Vikash";  
 };  
  
 *// Method 2 - If we are writing then return key we can not write though return key word is optional* Supplier<String> supplier1 = () -> "This is Vikash";  
  
 System.*out*.println(supplier.get());  
  
  
 }  
}

public class UseOfSupplier {  
  
 public static void main(String[] args) {  
  
 List<String> list = new ArrayList<>();  
 list.add("Vikash");  
 list.add("Singh");  
  
 Supplier<String> supplier = () -> "This is first way";  
  
 *// If Kumar not found then print suplier msg* System.*out*.println(list.stream().filter( string -> string.equals("Kumar")).findFirst().orElseGet(supplier));  
  
 *//If found then pring what ever found* System.*out*.println(list.stream().filter( string -> string.equals("Vikash")).findFirst().orElseGet(supplier));  
  
 list.clear();  
 *// Since I have clear the list supplier msg will print and purpose of this code to show we can directly use supplier in here* System.*out*.println(list.stream().findAny().orElseGet(() -> "This is Second way"));  
 }  
}

1. **Stream in Java:**

* Stream API is use to process collection of Objects.
* A stream is sequence of objects that supports various methods which can be pipelined to produce the desired result.
* A stream is not a data structure instead it takes input from the Collections, Arrays, or I/O channels.
* Stream don’t change the original data structure; they only provide the result as per the pipelined methods.

**Why we need Stream?**

* Functional Programming
* Code Reduce
* Bulk Operations

1. **Sorting Using Java Stream:**

* **List Sorting:**

public class SortingWithPredefinedObjects {  
  
 public static void main(String[] args) {  
 List<Integer> list = Arrays.*asList*(10,40, 20, 30, 50);  
  
  
 *// Ascending order* System.*out*.println();  
 System.*out*.println("Ascending order");  
 list.stream().sorted().forEach(t -> System.*out*.print(t + " "));  
  
 *// Descending order* System.*out*.println();  
 System.*out*.println("Descending order");  
 list.stream().sorted(Comparator.*reverseOrder*()).forEach(t -> System.*out*.print(t + " "));  
  
 List<Integer> list2 = new ArrayList<>();  
 list2.add(2);  
 list2.add(50);  
 list2.add(30);  
 list2.add(null);  
  
 *// Descending order with Null as Last* System.*out*.println();  
 System.*out*.println("Descending order with Null as Last");  
 list2.stream().sorted(Comparator.*nullsLast*(Comparator.*reverseOrder*())).forEach(t -> System.*out*.print(t + " "));  
  
 }  
}

public class SortingWithCustomObjects {  
  
 public static void main(String[] args) {  
 List<Employee> employeeList = new ArrayList<>();  
 employeeList.add(new Employee(1, "Vikash", 30000.00));  
 employeeList.add(new Employee(2, "Sonu", 30000.00));  
 employeeList.add(new Employee(3, "Kumar", 10000.00));  
 employeeList.add(new Employee(4, "Singh", 50000.00));  
 employeeList.add(new Employee(5, "Vikaram", 50000.00));  
  
 *// Using Old Method of Java - Ascending* System.*out*.println("Using Old Method");  
 System.*out*.println("In Ascending of Java");  
 Collections.*sort*(employeeList, (object1, object2) -> object1.getSalary().compareTo(object2.getSalary()));  
 employeeList.stream().forEach(t -> System.*out*.println(t));  
  
 *// Using Old Method of Java - Descending* System.*out*.println();  
 System.*out*.println("In Descending of Java");  
 Collections.*sort*(employeeList, (object1, object2) -> - object1.getSalary().compareTo(object2.getSalary()));  
 employeeList.stream().forEach(t -> System.*out*.println(t));  
  
 *// By Using Stream API* System.*out*.println();  
 System.*out*.println("--------------------------------------------------");  
 System.*out*.println("Using Stream API - Ascending");  
 employeeList.stream().sorted(Comparator.*comparing*(Employee::getSalary)).forEach(t -> System.*out*.println(t));  
  
 System.*out*.println("Using Stream API - Descending");  
 employeeList.stream().sorted(Comparator.*comparing*(Employee::getSalary, Comparator.*reverseOrder*())).forEach(t -> System.*out*.println(t));  
  
 System.*out*.println();  
 System.*out*.println("----------------------Null handle----------------------------");  
 System.*out*.println("Using Stream API with Null at last - Descending");  
 employeeList.add(new Employee(4, "Test", null));  
 *// If we will not add Comparator.nullsLast(Comparator.reverseOrder()) then we will get null pointer exception* employeeList.stream().sorted(Comparator.*comparing*(Employee::getSalary, Comparator.*nullsLast*(Comparator.*reverseOrder*()))).forEach(t -> System.*out*.println(t));  
  
 System.*out*.println();  
 System.*out*.println("----------------------Then compare----------------------------");  
 System.*out*.println("Using Stream API with Then compare and null at last - Descending");  
 employeeList.add(new Employee(4, "Test", null));  
 *// If we will not add Comparator.nullsLast(Comparator.reverseOrder()) then we will get null pointer exception* employeeList.stream().sorted(Comparator.*comparing*(Employee::getSalary, Comparator.*nullsLast*(Comparator.*reverseOrder*())).thenComparing(Employee::getEmployeeName, Comparator.*nullsLast*(Comparator.*reverseOrder*()))).forEach(t -> System.*out*.println(t));  
 }  
 }

}  
  
public class Employee{  
 private Integer employeeId;  
 private String employeeName;  
 private Double salary;  
  
 public Employee(Integer employeeId, String employeeName, Double salary) {  
 this.employeeId = employeeId;  
 this.employeeName = employeeName;  
 this.salary = salary;  
 }  
  
 public Integer getEmployeeId() {  
 return employeeId;  
 }  
  
 public void setEmployeeId(Integer employeeId) {  
 this.employeeId = employeeId;  
 }  
  
 public String getEmployeeName() {  
 return employeeName;  
 }  
  
 public void setEmployeeName(String employeeName) {  
 this.employeeName = employeeName;  
 }  
  
 public Double getSalary() {  
 return salary;  
 }  
  
 public void setSalary(Double salary) {  
 this.salary = salary;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "employeeId=" + employeeId +  
 ", employeeName='" + employeeName + '\'' +  
 ", salary=" + salary +  
 '}';  
 }  
}

* **Map Sorting:**

public class MapSortingWithPredefinedObjects {

public static void main(String[] args) {  
 Map<String, Integer> map = new HashMap<>();  
 map.put("eight", 8);  
 map.put("four", 4);  
 map.put("ten", 10);  
 map.put("two", 2);  
 map.put("six", 6);  
  
 *// Sort by Key - Sorting Map by Using old approach* System.*out*.println("-----------Sort by Key - Sorting Map by Using old approach-----------");  
 List<Map.Entry<String, Integer>> entries = new ArrayList<>(map.entrySet());  
 Collections.*sort*(entries, new Comparator<Map.Entry<String, Integer>>() {  
 @Override  
 public int compare(Map.Entry<String, Integer> o1, Map.Entry<String, Integer> o2) {  
 return o1.getKey().compareTo(o2.getKey());  
 }  
 });  
  
 for(Map.Entry<String, Integer> entry : entries){  
 System.*out*.println(entry.getKey()+" : "+entry.getValue());  
 }  
  
 System.*out*.println("-----------Sort by Value - Sorting Map by Using old approach-----------");  
 Collections.*sort*(entries, new Comparator<Map.Entry<String, Integer>>() {  
 @Override  
 public int compare(Map.Entry<String, Integer> o1, Map.Entry<String, Integer> o2) {  
 return o1.getValue().compareTo(o2.getValue());  
 }  
 });  
  
 for(Map.Entry<String, Integer> entry : entries){  
 System.*out*.println(entry.getKey()+" : "+entry.getValue());  
 }  
  
 System.*out*.println("-----------Sort by Value Ascending- Sorting Map by Using Lambda Expression-----------");  
 Collections.*sort*(entries, (object1, object2) -> object1.getValue().compareTo(object2.getValue()));  
  
 for(Map.Entry<String, Integer> entry : entries){  
 System.*out*.println(entry.getKey()+" : "+entry.getValue());  
 }  
 *// ------------------- Lambda Expression --------------------------------------------------------------------* System.*out*.println("-----------Sort by Value Descending - Sorting Map by Using Lambda Expression-----------");  
 Collections.*sort*(entries, (object1, object2) -> - object1.getValue().compareTo(object2.getValue()));  
  
 for(Map.Entry<String, Integer> entry : entries){  
 System.*out*.println(entry.getKey()+" : "+entry.getValue());  
 }  
  
 */\* ------------------- Using Java8 Stream API -----------------------------------------------------------\*/* System.*out*.println("---------------------Using Java8 Stream API - Ascending By Key---------------------");  
 map.entrySet().stream().sorted(Map.Entry.*comparingByKey*()).forEach(t -> System.*out*.println(t));  
  
 System.*out*.println("---------------------Using Java8 Stream API - Ascending By Value---------------------");  
 map.entrySet().stream().sorted(Map.Entry.*comparingByValue*()).forEach(t -> System.*out*.println(t));  
  
 System.*out*.println("---------------------Using Java8 Stream API - Descending By Value---------------------");  
 // **VVI**  **map.entrySet().stream().sorted(Map.Entry.*comparingByValue*(Comparator.*reverseOrder*())).forEach(t -> System.*out*.println(t));**  
  
 System.*out*.println("---------------------Using Java8 Stream API - Descending By Value null at last---------------------");  
 map.put("three", null);  
 // **VVI**  **map.entrySet().stream().sorted(Map.Entry.*comparingByValue*(Comparator.*nullsLast*(Comparator.*reverseOrder*()))).forEach(t -> System.*out*.println(t));**  
 }  
}

public class MapSortingWithCustomObjects {  
  
 public static void main(String[] args) {  
 Map<Employee, Integer> map = new HashMap<>();  
 map.put(new Employee(1, "Vikash", 50000.0), 8);  
 map.put(new Employee(2, "Singh", 20000.0), 4);  
 map.put(new Employee(3, "Kumar", 30000.0), 10);  
 map.put(new Employee(4, "Sonu", 60000.0), 2);  
 map.put(new Employee(5, "Sachin", 80000.0), 6);  
  
 *// Ascending* System.*out*.println("------------Ascending by Key Salary------------");  
 map.entrySet().stream().sorted(Map.Entry.*comparingByKey*(Comparator.*comparing*(Employee::getSalary))).forEach(t -> System.*out*.println(t));  
  
 *// Descending* System.*out*.println();  
 System.*out*.println("------------Descending by Key Salary------------");  
 map.entrySet().stream().sorted(Map.Entry.*comparingByKey*(Comparator.*comparing*(Employee::getSalary, Comparator.*reverseOrder*()))).forEach(t -> System.*out*.println(t));  
 *// Descending* System.*out*.println();  
 System.*out*.println("------------Descending by Key Salary null at Last------------");  
 map.put(new Employee(6, "Test", null), 6);  
 map.entrySet().stream().sorted(Map.Entry.*comparingByKey*(Comparator.*comparing*(Employee::getSalary, Comparator.*nullsLast*(Comparator.*reverseOrder*())))).forEach(t -> System.*out*.println(t));  
 }  
}

1. **Map() vs flatMap():**

* Java Stream API provides map() and flatMap() methods. 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**

**map() : (Means Data Transformation)**

* map() takes Stream<T> as input and return Stream<R>
* Stream<R> map(Stream<R> input){}

Syntax: <R> Stream<R> map(Function<? Super T, ? extends R> mapper);

* It’s mapper function produces single value for each input value; hence it is also called One-To-One mapping.

**flatMap() : (Means Map + Flattering)**

* FlatMap() takes Stream<Stream<T>> (also called Stream of Stream) as input and returns Stream<R>
* Stream<R> flatMap(Stream<Stream<T>> input) {}
* Syntax: <R> Stream<R> flatMap(Function<? Super T, ? extends Stream<? extends R>> mapper);
* It’s mapper function produces multiple value for each input value hence it is also called One-To-Many mapping

**What is Data Transformation & Flattering?**

**Data Transformation:**

Converting Stream.of(a, b, c, d) into A, B, C, D. Here converting lower to upper case called mapping and return it as another stream called data transformation and the entire process is called transforming the data.

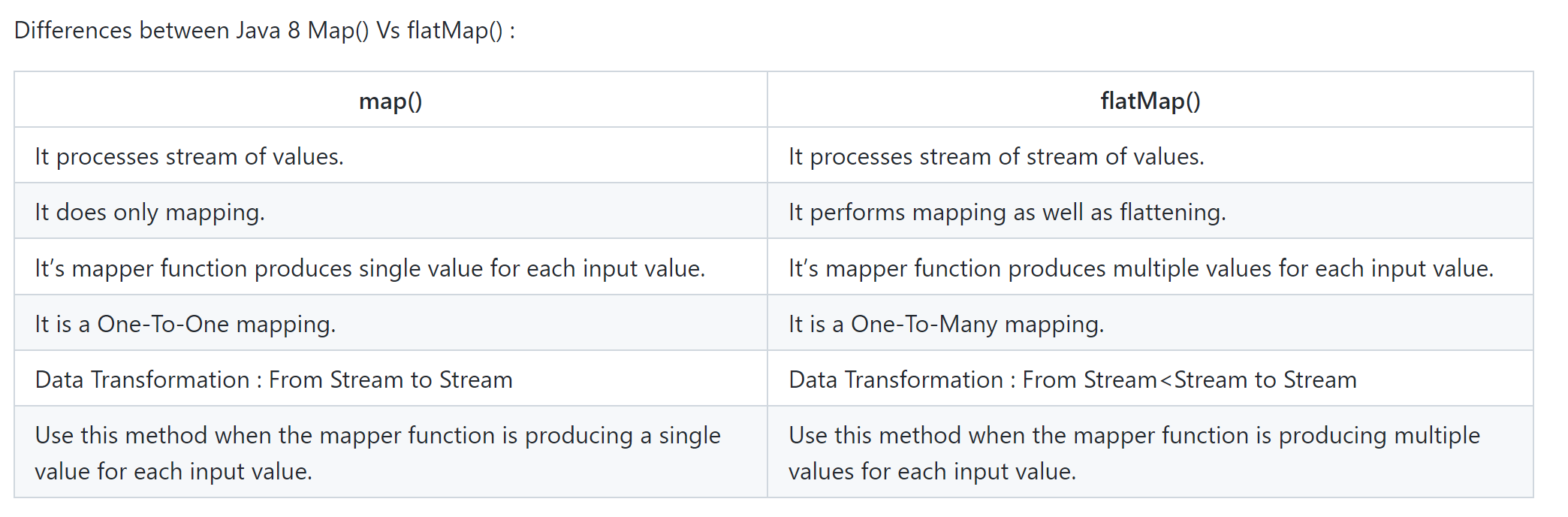
**Flattering:**

Let’s assume we have data like [[1, 2], [3, 4], [4, 5], [5, 6], [7, 8]] i.e Once stream contains another Stream now, I want to combine all the sub stream into a single stream this conversion of stream called as Flattering.

public class MapVsFlatMap {  
  
 public static void main(String[] args) {  
  
 List<Customer> customers = Stream.*of*(  
 new Customer(101, "john", "john@gmail.com", Arrays.*asList*("397937955", "21654725")),  
 new Customer(102, "smith", "smith@gmail.com", Arrays.*asList*("89563865", "2487238947")),  
 new Customer(103, "peter", "peter@gmail.com", Arrays.*asList*("38946328654", "3286487236")),  
 new Customer(104, "kely", "kely@gmail.com", Arrays.*asList*("389246829364", "948609467"))  
 ).collect(Collectors.*toList*());  
  
 *//List<Customer> convert List<String> -> Data Transformation  
 //mapping : customer -> customer.getEmail()  
 //customer -> customer.getEmail() one to one mapping because one customer has only one email in this scenario* System.*out*.println("------------Normal map() example------------");  
 List<String> emails = customers.stream().map(customer -> customer.getEmail()).collect(Collectors.*toList*());  
 System.*out*.println(emails);  
  
 *//customer -> customer.getPhoneNumbers() ->> one to many mapping* System.*out*.println("------------Getting Phone number by using map() which will return List of List------------");  
 **List<List<String>> phoneNumbers = customers.stream().map(customer -> customer.getPhoneNumbers()).collect(Collectors.*toList*());  
 System.*out*.println(phoneNumbers);**  
  
 *// So if we want to merge all the list in a single list then we can use flatMapFunction  
 //List<Customer> convert List<String> -> Data Transformation  
 //mapping : customer -> phone Numbers  
 //customer -> customer.getPhoneNumbers() ->> one to many mapping because in this scenario we have multiple phone numbers* **List<String> phones = customers.stream().flatMap(customer -> customer.getPhoneNumbers().stream()).collect(Collectors.*toList*());  
 System.*out*.println(phones);**  
 }  
}

Customer.class

public class Customer {  
  
 private int id;  
 private String name;  
 private String email;  
 private List<String> phoneNumbers;  
  
 public Customer() {  
 }  
  
public Customer(int id, String name, String email, List<String> phoneNumbers) {  
 this.id = id;  
 this.name = name;  
 this.email = email;  
 this.phoneNumbers = phoneNumbers;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 public List<String> getPhoneNumbers() {  
 return phoneNumbers;  
 }  
  
 public void setPhoneNumbers(List<String> phoneNumbers) {  
 this.phoneNumbers = phoneNumbers;  
 }  
  
 @Override  
 public String toString() {  
 return "Customer{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", email='" + email + '\'' +  
 ", phoneNumbers=" + phoneNumbers +  
 '}';  
 }  
}



1. **Optional:**

Optional is final class which having lot of methods to deal with normal Java object. It is a container object may or may not contains non-null value.

Java provides 3 methods to create Optional object.

1. empty() – It will create empty optional object
2. of() – It create option object of type give but if passing object is null then it will throw null pointer exception, So it is not recommended to use this method util we are sure that there will be no null value for that object.
3. ofNullable() – **It is recommended to use this method to get Optional object in place to of() method because first it checking if object is null then it will return empty optional object and if not null then it will return option object of passed type**.

Note: .orElse(String) this method takes string and orElseGet(Suplier) and orElseThorw() – this method takes supplier so make sure you are passing or proper lambda from these methods like () -> new Exception(“Object not found”), since supplier have only one abstract method “T get()” i.e its takes no argument but returning one value.

public class OptionDemo {  
  
 public static void main(String[] args) throws Exception {  
  
 Employee employee = new Employee(1, null, 4000.0);  
 Employee employeeVikash = new Employee(2, "Vikash", 4000.0);  
  
 *// This method will return null since name is null  
 /\*Optional<String> employeeOptional = Optional.of(employee.getEmployeeName());  
 System.out.println(employeeOptional);\*/  
  
 // If null then print Optional.empty* Optional<String> employeeOptional1 = Optional.*ofNullable*(employee.getEmployeeName());  
 System.*out*.println(employeeOptional1);  
  
 *// In this if null then returning some msg* Optional<String> employeeOptional2 = Optional.*ofNullable*(employee.getEmployeeName());  
 System.*out*.println(employeeOptional2.orElse("No name present"));  
  
 *// We can use other stream api method optional provide us liberty that if we want we  
 // can perform some operation or else return some default value* Optional<String> employeeOptional3 = Optional.*ofNullable*(employee.getEmployeeName());  
 System.*out*.println(employeeOptional3.map(String::toUpperCase).orElse("No name present"));  
  
 *// If some present then changing to Upper case and no value then returning default value* Optional<String> employeeOptional4 = Optional.*ofNullable*(employeeVikash.getEmployeeName());  
 System.*out*.println(employeeOptional4.map(String::toUpperCase).orElse("No name present"));  
 }  
}

1. **Map() and reduce():**

* Map and reduce is a functional programming model it serves our two purpose

Map – Transforming Data

Reduce – Aggerating Data (Combining elements of stream and produces a single value)

* Ex: Stream: [2, 4, 6, 9, 1, 3, 7] Sum of numbers present in stream.
* Reduce() – Combine stream of int and produce the sum result.

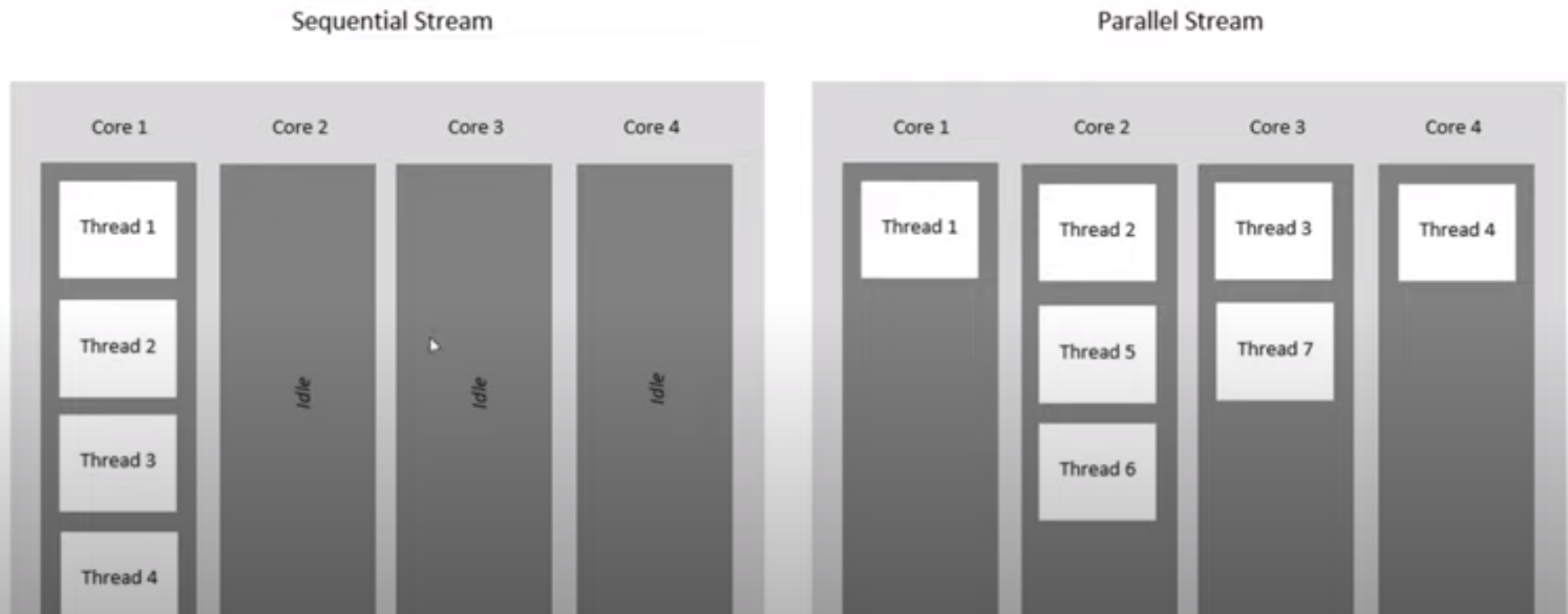
Reduce:

* T reduce (T identity, BinaryOperator<T> accumulator);
* Identity is initial value for type T
* Accumulator is function for combining two values
* Integer sumResult = Stream.of(2, 4, 6, 9, 1, 3, 7).**reduce(0, (a, b) -> a+b)**;

**In the above example Identity 0 which is nothing but initial value and Accumulator: (a,b) -> a+b is function.**

public class MapReduceDemo {  
  
 public static void main(String[] args) {  
  
 List<Integer> numbers = Arrays.*asList*(3, 7, 8, 1, 5, 9);  
  
 List<String> words = Arrays.*asList*("corejava", "spring", "hibernate");  
  
 int sum1 = numbers.stream().mapToInt(i -> i).sum();  
 System.*out*.println(sum1);  
  
 Integer reduceSum = numbers.stream().reduce(0, (a, b) -> a + b);  
 System.*out*.println(reduceSum);  
  
 Optional<Integer> reduceSumWithMethodReference = numbers.stream().reduce(Integer::*sum*);  
 System.*out*.println(reduceSumWithMethodReference.get());  
  
 Integer mulResult = numbers.stream().reduce(1, (a, b) -> a \* b);  
 System.*out*.println(mulResult);  
  
 Integer maxvalue = numbers.stream().reduce(0, (a, b) -> a > b ? a : b);  
 System.*out*.println(maxvalue);  
  
 Integer maxvalueWithMethodReference = numbers.stream().reduce(Integer::*max*).get();  
 System.*out*.println(maxvalueWithMethodReference);  
  
  
 String longestString = words.stream()  
 .reduce((word1, word2) -> word1.length() > word2.length() ? word1 : word2)  
 .get();  
 System.*out*.println(longestString);  
  
 List<Employee> employeeList = Stream.*of*(new Employee(101,"john","A",60000),  
 new Employee(109,"peter","B",30000),  
 new Employee(102,"mak","A",80000),  
 new Employee(103,"kim","A",90000),  
 new Employee(104,"json","C",15000))  
 .collect(Collectors.*toList*());  
  
 *//get employee whose grade A  
 //get salary* double avgSalary = employeeList.stream()  
 .filter(employee -> employee.getGrade().equalsIgnoreCase("A"))  
 .map(employee -> employee.getSalary())  
 .mapToDouble(i -> i)  
 .average().getAsDouble();  
  
 System.*out*.println(avgSalary);  
  
 double sumSalary = employeeList.stream()  
 .filter(employee -> employee.getGrade().equalsIgnoreCase("A"))  
 .map(employee -> employee.getSalary())  
 .mapToDouble(i -> i)  
 .sum();  
 System.*out*.println(sumSalary);  
 }  
}

1. **Parallel Streams:**

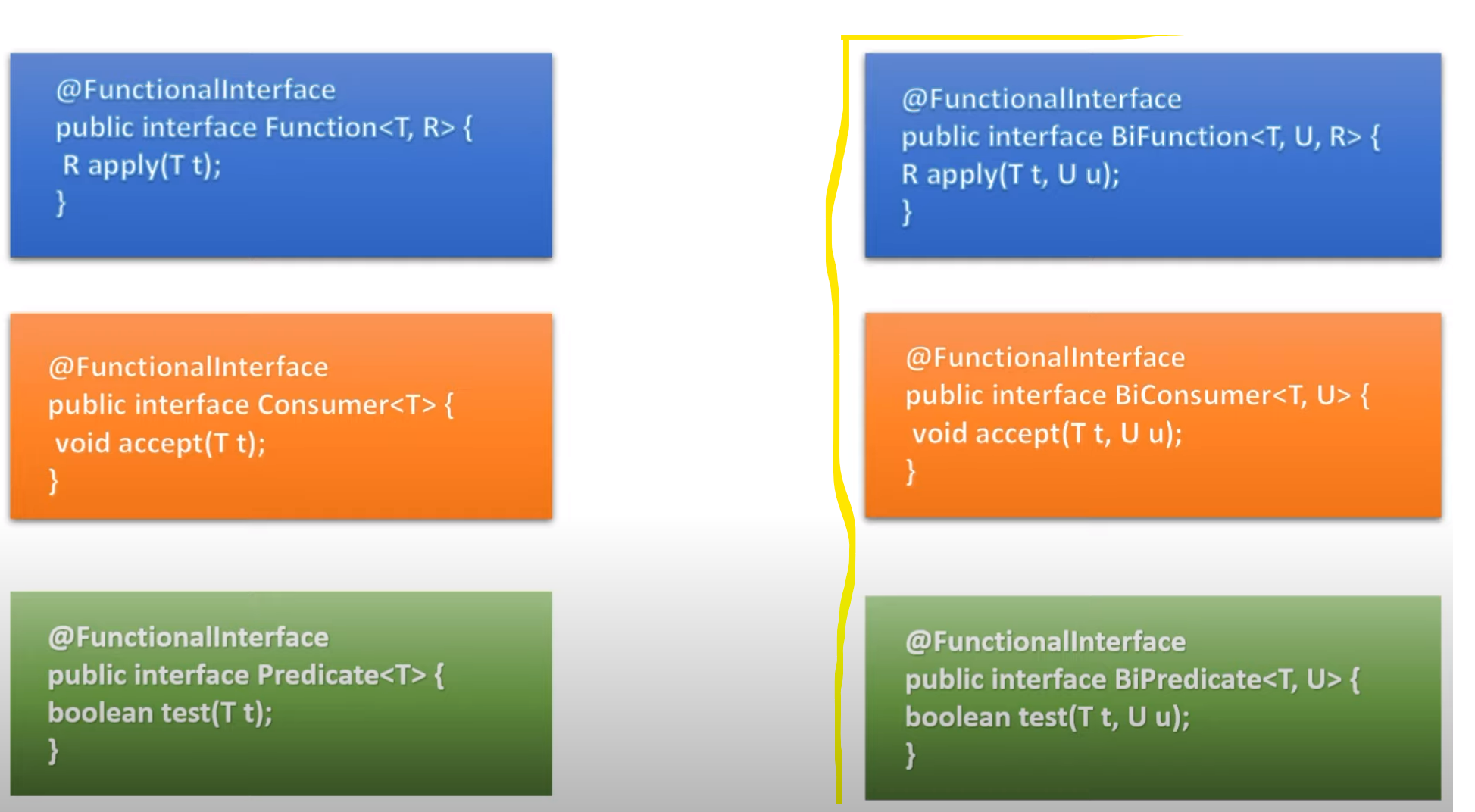
* Java Parallel Streams is a feature of Java 8, It meant for utilizing multiple cores of the processor
* Normally any Java code has one stream of processing, where it is executed sequentially. Where by using parallel streams, we can divide the code into multiple streams that are executed in parallel on separate cores and the final result is the combination of the individual outcomes
* Then Order of execution, however is not under control. 

**Note: Parallel Stream uses ForkJoinPool for multithreading.**

public class ParallelStreamDemo {  
 public static void main(String[] args) {  
 long start = 0;  
 long end = 0;  
  
 *// Printing 1 to 100 using sequential stream* start = System.*currentTimeMillis*();  
 IntStream.*range*(1, 100).forEach(x -> {  
 System.*out*.print(x + " ");  
 });  
 end = System.*currentTimeMillis*();  
  
 System.*out*.println();  
 System.*out*.println("Total time taken to print 1 to 100 by sequential stream = "+(end - start));  
  
 *// Printing 1 to 100 using parallel stream* start = System.*currentTimeMillis*();  
 System.*out*.println();  
 IntStream.*range*(1, 100).parallel().forEach(x -> {  
 System.*out*.print(x + " ");  
 });  
 end = System.*currentTimeMillis*();  
 System.*out*.println();  
 System.*out*.println("Total time taken to print 1 to 100 by parallel stream = "+(end - start));  
  
 *// ======================= Thread name and printing values =======================  
 // Printing 1 to 100 using sequential stream with Thread Name* start = System.*currentTimeMillis*();  
 System.*out*.println("======================= Thread name and printing values =======================");  
 System.*out*.println();  
 IntStream.*range*(1, 100).forEach(x -> {  
 System.*out*.println("Thread "+Thread.*currentThread*().getName()+" : "+x);  
 });  
 end = System.*currentTimeMillis*();  
 System.*out*.println("Total time taken to print 1 to 100 by sequential stream with thread= "+(end - start));  
  
 *// Printing 1 to 100 using Parallel stream with Thread Name* start = System.*currentTimeMillis*();  
 System.*out*.println();  
 IntStream.*range*(1, 100).parallel().forEach(x -> {  
 System.*out*.println("Thread "+Thread.*currentThread*().getName()+" : "+x);  
 });  
 end = System.*currentTimeMillis*();  
 System.*out*.println("Total time taken to print 1 to 100 by parallel stream with thread = "+(end - start));  
  
 *// ======================= Working with Employee Objects =======================* System.*out*.println("Working with Employee Objects");  
  
 *// Creating 1000 objects of Employee and finding average salary* start = System.*currentTimeMillis*();  
 List<Employee> employeeList = *getEmployeeList*();  
 employeeList.stream().map(employee -> employee.getSalary()).mapToDouble(i -> i).average();  
 end = System.*currentTimeMillis*();  
 System.*out*.println("Finding average salary of employee by using sequential stream "+(end - start));  
  
 start = System.*currentTimeMillis*();  
 List<Employee> employeeList2 = *getEmployeeList*();  
 employeeList2.parallelStream().map(employee -> employee.getSalary()).mapToDouble(i -> i).average();  
 end = System.*currentTimeMillis*();  
 System.*out*.println("Finding average salary of employee by using parallel stream "+(end - start));  
  
  
  
 }  
  
 public static List<Employee> getEmployeeList(){  
 List<Employee> employeeList = new ArrayList<>();  
 for(int i = 0; i<= 1000; i++){  
 employeeList.add(new Employee(i, "Employee\_name"+i, new Random().nextDouble()));  
 }  
 return employeeList;  
 }  
}

1. **BiFunctional Interface/ BiConsumer/ BiPredicate:**

**It is same as Function the only difference is that it takes 2 inputs:**

****

**BiFunction Program:**

public class BiFunctionDemoByOldApproach implements BiFunction<List<Integer>, List<Integer>, List<Integer>> {  
  
 @Override  
 public List<Integer> apply(List<Integer> list1, List<Integer> list2){  
 return Stream.*of*(list1, list2).flatMap(List::stream).distinct().collect(Collectors.*toList*());  
 }  
  
 public static void main(String[] args) {  
 List<Integer> list1 = Stream.*of*(1, 3, 4, 6, 7, 9, 19).collect(Collectors.*toList*());  
 List<Integer> list2 = Stream.*of*(11, 3, 43, 6, 7, 19).collect(Collectors.*toList*());  
  
 BiFunctionDemoByOldApproach biFunctionDemoByOldApproach = new BiFunctionDemoByOldApproach();  
 List<Integer> list3 = biFunctionDemoByOldApproach.apply(list1, list2);  
 System.*out*.println(list3);  
 }  
}

public class BiFunctionDemoByLambdaExpression {  
  
 public static void main(String[] args) {  
  
 *// Output - [1, 3, 4, 6, 7, 9, 19, 11, 43]* List<Integer> listWithData1 = Stream.*of*(1, 3, 4, 6, 7, 9, 19).collect(Collectors.*toList*());  
 List<Integer> listWithData2 = Stream.*of*(11, 3, 43, 6, 7, 19).collect(Collectors.*toList*());  
  
 BiFunction<List<Integer>, List<Integer>, List<Integer>> biFunction = (list1, list2) -> Stream.*of*(list1, list2).flatMap(List::stream).distinct().collect(Collectors.*toList*());  
  
 List<Integer> listAfterMerge = biFunction.apply(listWithData1, listWithData2);  
 System.*out*.println(listAfterMerge);  
  
 }  
}

public class ApplicationOfBiFunction {  
  
 public static void main(String[] args) {  
  
 Map<String, Integer> map=new HashMap<>();  
 map.put("basant",5000);  
 map.put("santosh",15000);  
 map.put("javed",12000);  
  
 System.*out*.println("Before: "+map);  
 BiFunction<String, Integer, Integer> salaryIncremental = (key, value) -> value + 500;  
 map.replaceAll(salaryIncremental);  
  
 System.*out*.println("After: "+map);  
  
 *// We can directly use lambda expression inside replaceAll method* System.*out*.println("Before adding 1000: "+map);  
 map.replaceAll((key, value) -> value + 1000);  
 System.*out*.println("After adding 1000: "+map);  
 }  
}

public class ApplicationOfBiFunction2 {  
  
 public static void main(String[] args) {  
  
 List<Integer> listWithData1 = Stream.*of*(1, 3, 4, 6, 7, 9, 19).collect(Collectors.*toList*());  
 List<Integer> listWithData2 = Stream.*of*(11, 3, 43, 6, 7, 19).collect(Collectors.*toList*());  
  
 BiFunction<List<Integer>, List<Integer>, List<Integer>> biFunction = (list1, list2) -> Stream.*of*(list1, list2).flatMap(List::stream).distinct().collect(Collectors.*toList*());  
  
 *// This is function to use andThen method of BiFunction* Function<List<Integer>, List<Integer>> sortFunction = (rowList) -> rowList.stream().sorted().collect(Collectors.*toList*());  
  
 List<Integer> listAfterMergeUnsorted = biFunction.apply(listWithData1, listWithData2);  
 System.*out*.println("Un-Sorted = "+ listAfterMergeUnsorted);  
  
 List<Integer> listAfterMergeSorted = biFunction.andThen(sortFunction).apply(listWithData1, listWithData2);  
 System.*out*.println("Sorted = "+ listAfterMergeSorted);  
  
 }  
}

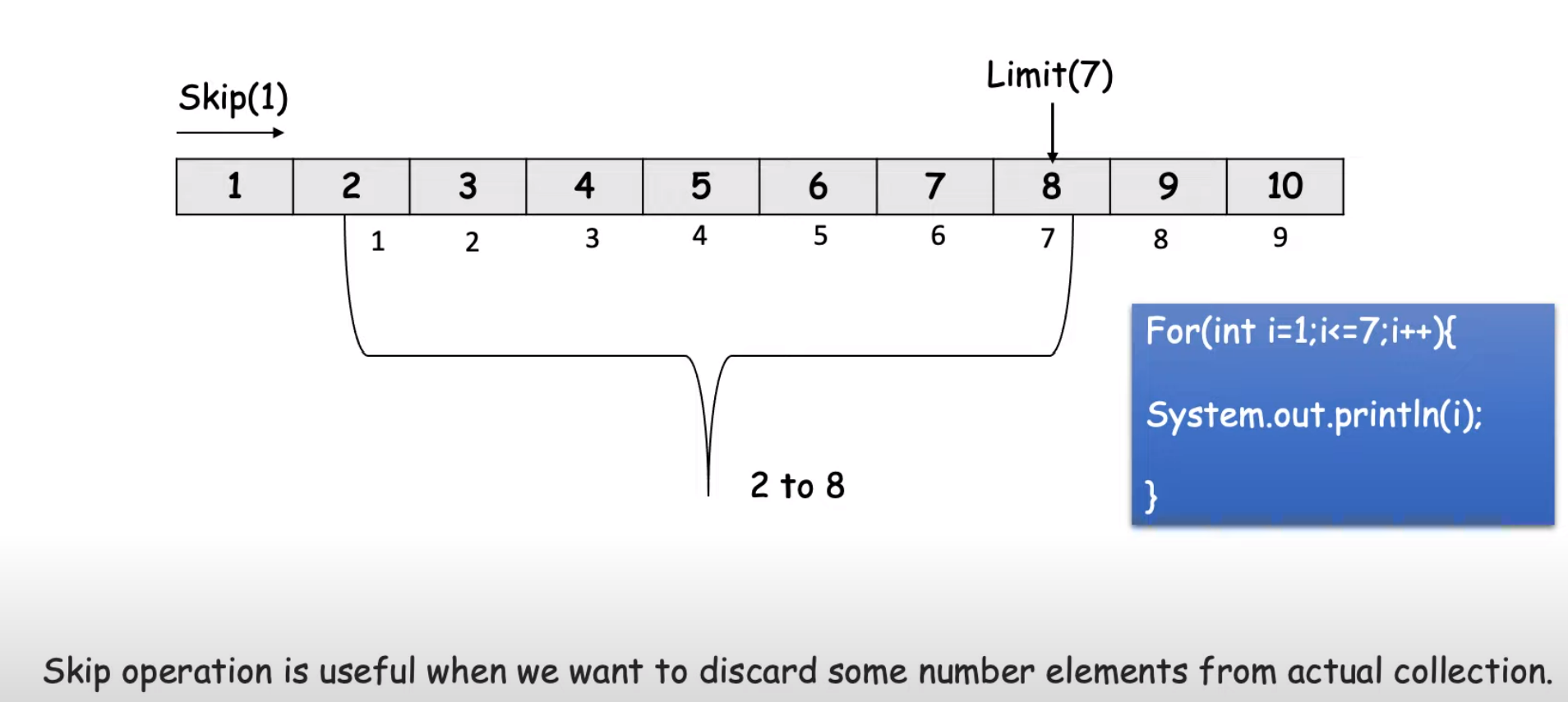
**BiCosumer:**

public class BiConsumerDemoByOldApproach implements BiConsumer<String,Integer> {  
  
 @Override  
 public void accept(String i1, Integer i2) {  
 System.*out*.println("input 1 "+i1 +": input 2 "+i2);  
 }  
  
 public static void main(String[] args) {  
 BiConsumer<String,Integer> biConsumer=new BiConsumerDemoByOldApproach();  
 biConsumer.accept("javatechie",53000);  
 }  
}

public class BiConsumerDemoByLambdaExpression {  
  
 public static void main(String[] args) {  
 BiConsumer<String,Integer> biConsumer = (firstValue, secondValue) -> {  
 System.*out*.println("input 1 "+ firstValue +": input 2 "+ secondValue);  
 };  
  
 biConsumer.accept("test", 10);  
 }  
}

public class BiConsumerApplication {  
  
 public static void main(String[] args) {  
 Map<String, Integer> map=new HashMap<>();  
 map.put("basant",5000);  
 map.put("santosh",15000);  
 map.put("javed",12000);  
  
 map.forEach((k,v)-> System.*out*.println(k+","+v));  
 }  
}

1. **Skip() & limit(): (VVI asked in interview)**



public class SkipAndLimitDemo {  
  
 public static void main(String[] args) {  
 List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
  
 *// Print 2 to 8* list.stream().skip(1).limit(7).forEach(t -> System.*out*.print(t+" "));  
  
 *// Print 3 to 9* System.*out*.println();  
 list.stream().skip(2).limit(7).forEach(t -> System.*out*.print(t+" "));  
 }  
}

public class SkipAndLimitApplication {  
  
 public static void main(String[] args) throws IOException {  
  
 List<String> fileData = Files.*readAllLines*(Paths.*get*("data.txt"));  
  
 *// Read and print all the data of file* fileData.forEach(System.*out*::println);  
  
 *// Now Remove the header of the file* System.*out*.println("\n=============After Removing of header of the File=============");  
 fileData.stream().skip(1).forEach(System.*out*::println);  
  
 *// Now Remove the header of the file* System.*out*.println("\n=============After Removing of header and footer of the File=============");  
 fileData.stream().skip(1).limit(fileData.size()-2).forEach(System.*out*::println);  
 }  
}

File data: (Add this data.txt in direct project folder)

1. id    name       qty     price
2. 101   Mobile      1      50000
3. 102   watch       2      4000
4. 103   TV          1      75000
5. 104   HeadSet     3      15000
6. 105   Books       1      1
7. 8      100000

**CompletableFuture**

1. **What is CompletableFuture?**

CompletableFuture: A new era of asynchronous programming

* Using Asynchronous programming you can write non-blocking code where concurrently you can run N no of task in separate thread without blocking main thread.
* When the task is complete it notifies to the main thread (whether the task was complete or failed)

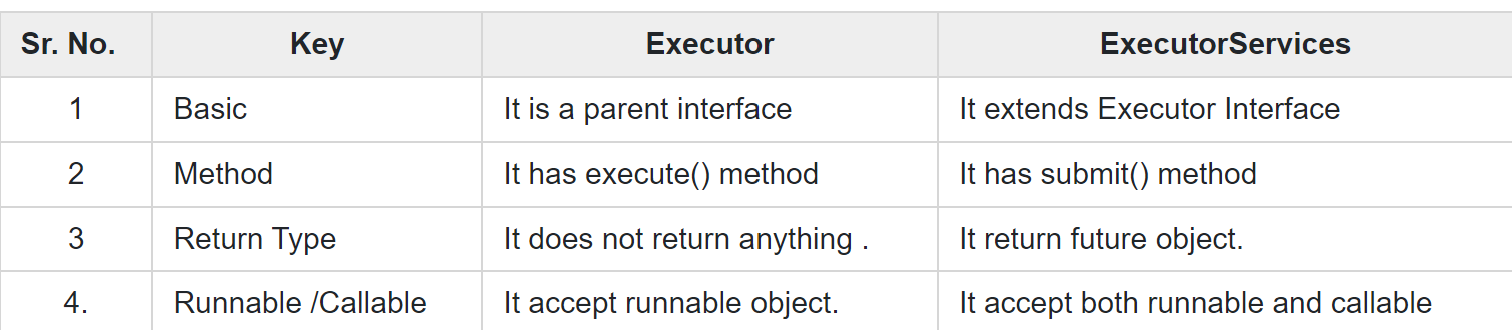
1. **What is CompletableFuture?**

There are different ways to implements asynchronous programming in Java, like **Futures**, **ExecutorService**, CallBack interfaces, Thread Pools etc.

Disadvantage of Future:

* 1. It can not manually completed
  2. Multiple Futures cannot be chained together
  3. We can not combine multiple Futures together
  4. No Proper Exception Handling Mechanism

1. **Executor vs ExecutorService:**



1. **Executor framework built in thread pools:**

* Fixed Thread Pool
* Cached Thread Pool
* Scheduled Thread Pool
* Single Thread Executor

1. **Difference between the submit() and execute() method in Java Concurrency?**

1) The submit() can accept both Runnable and Callable tasks but execute() can only accept the Runnable task.

2) The submit() method is declared in the ExecutorService interface while the execute() method is declared in the Executor interface.

3) The return type of submit() method is a Future object but the return type of execute() method is void.

import java.util.Arrays;  
import java.util.List;  
import java.util.concurrent.\*;  
  
public class WhyNotFuture {  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
  
 ExecutorService executorService = Executors.*newFixedThreadPool*(10);  
 Future<List<Integer>> submit = executorService.submit(() -> {  
 return Arrays.*asList*(1, 2, 3, 4);  
 });  
  
 List<Integer> list = submit.get();  
  
 list.stream().forEach(System.*out*::println);  
  
 *// We do not have any feature to combine, manually/forcefully complete the task we cannot chain multiple futures and no exception handling* }  
 public static void delay(int min){  
 try{  
 TimeUnit.*MINUTES*.sleep(min);  
 }catch (InterruptedException e){  
 e.printStackTrace();  
 }  
 }  
}

public class CompletableFutureDemo1 {  
  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
 *// How to create object and get and manually complete the execution* CompletableFuture<String> completableFuture = new CompletableFuture<>();  
 *// Here we are blocking the thread to get the result* completableFuture.get();  
 *// if get is taking more time than we can complete the thread by  
 // using compete() method as below* completableFuture.complete("Return some dummy data...");  
 }  
}

1. **What is runAsync() and supplyAsync():**

If we want to run some background task asynchronous and **do not want to return** anything from that task then use CompletableFuture.runAsync() method. It takes a Runnable object and returns CompletableFuture<Void>.

1. CompletableFuture.runAsync(Runnable) – (In case of Runnable this method get the thread from ForkJoinPool)
2. CompletableFuture.runAsync(Runnable, Executor) – (In that case thread will be taken from executor)

If we want to run some background task asynchronously and want to return anything from that task, we should use CompletableFuture.supplyAsync(). It takes a Supplier<T> and returns CompletableFuture<T> where T is the type of the value obtained by calling the given supplier

1. CompletableFuture.supplyAsync(Supllier<T>)
2. CompletableFuture.supplyAsync(Supllier<T>, Executor)

public class **RunAsyncDemo** {  
  
 public Void saveEmployees(File jsonFile) throws ExecutionException, InterruptedException {  
 ObjectMapper mapper = new ObjectMapper();  
 CompletableFuture<Void> runAsyncFuture = CompletableFuture.*runAsync*(new Runnable() {  
  
 @Override  
 public void run() {  
 try {  
 List<Employee> employees = mapper.readValue(jsonFile, new TypeReference<Object>() {  
 });  
 *//write logic t save list of employee to database  
 //repository.saveAll(employees);* System.*out*.println("Thread : " + Thread.*currentThread*().getName());  
 System.*out*.println(employees);  
 System.*out*.println(employees.size());  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 return runAsyncFuture.get();  
 }  
  
 public Void saveEmployeesByLambda(File jsonFile) throws ExecutionException, InterruptedException {  
 ObjectMapper mapper = new ObjectMapper();  
 CompletableFuture<Void> runAsyncFuture = CompletableFuture.*runAsync*(() -> {  
  
 try {  
 List<Employee> employees = mapper.readValue(jsonFile, new TypeReference<Object>() {  
 });  
 *//write logic t save list of employee to database  
 //repository.saveAll(employees);* System.*out*.println("Thread : " + Thread.*currentThread*().getName());  
 System.*out*.println(employees);  
 System.*out*.println(employees.size());  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 });  
  
 return runAsyncFuture.get();  
 }  
  
 public Void saveEmployeesWithCustomExecutor(File jsonFile) throws ExecutionException, InterruptedException {  
 ObjectMapper mapper = new ObjectMapper();  
 Executor executor = Executors.*newFixedThreadPool*(5);  
 CompletableFuture<Void> runAsyncFuture = CompletableFuture.*runAsync*(  
 () -> {  
 try {  
 List<Employee> employees = mapper  
 .readValue(jsonFile, new TypeReference<List<Employee>>() {  
 });  
 *//write logic t save list of employee to database  
 //repository.saveAll(employees);* System.*out*.println("Thread : " + Thread.*currentThread*().getName());  
 System.*out*.println(employees.size());  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 },executor);  
  
 return runAsyncFuture.get();  
 }  
  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
 RunAsyncDemo runAsyncDemo = new RunAsyncDemo();  
 *// runAsyncDemo.saveEmployees(new File("employees.json"));  
 // runAsyncDemo.saveEmployeesByLambda(new File("employees.json"));* runAsyncDemo.saveEmployeesWithCustomExecutor(new File("employees.json"));  
 }  
}

public class **SupplyAsyncDemo** {  
  
 public List<Employee> getEmployees() throws ExecutionException, InterruptedException {  
 CompletableFuture<List<Employee>> listCompletableFuture = CompletableFuture  
 .*supplyAsync*(() -> {  
 System.*out*.println("Executed by : " + Thread.*currentThread*().getName());  
 return *fetchEmployees*();  
 });  
 return listCompletableFuture.get();  
 }  
  
 public List<Employee> getEmployeesWithExecutor() throws ExecutionException, InterruptedException {  
 Executor executor = Executors.*newCachedThreadPool*();  
 CompletableFuture<List<Employee>> listCompletableFuture = CompletableFuture  
 .*supplyAsync*(() -> {  
 System.*out*.println("Executed by : " + Thread.*currentThread*().getName());  
 return *fetchEmployees*();  
 },executor);  
 return listCompletableFuture.get();  
 }  
  
 public static List<Employee> fetchEmployees() {  
 ObjectMapper mapper = new ObjectMapper();  
 try {  
 return mapper.readValue(new File("employees.json"), new TypeReference<List<Employee>>() {  
 });  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return null;  
 }  
  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
 SupplyAsyncDemo supplyAsyncDemo = new SupplyAsyncDemo();  
 List<Employee> employees = supplyAsyncDemo.getEmployees();  
 employees.stream().forEach(System.*out*::println);  
 }  
}

public class Employee {  
 private String employeeId;  
 private String firstName;  
 private String lastName;  
 private String email;  
 private String gender;  
 private String newJoiner;  
 private String learningPending;  
 private int salary;  
 private int rating;  
  
 public String getEmployeeId() {  
 return employeeId;  
 }  
  
 public void setEmployeeId(String employeeId) {  
 this.employeeId = employeeId;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public void setFirstName(String firstName) {  
 this.firstName = firstName;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 public String getGender() {  
 return gender;  
 }  
  
 public void setGender(String gender) {  
 this.gender = gender;  
 }  
  
 public String getNewJoiner() {  
 return newJoiner;  
 }  
  
 public void setNewJoiner(String newJoiner) {  
 this.newJoiner = newJoiner;  
 }  
  
 public String getLearningPending() {  
 return learningPending;  
 }  
  
 public void setLearningPending(String learningPending) {  
 this.learningPending = learningPending;  
 }  
  
 public int getSalary() {  
 return salary;  
 }  
  
 public void setSalary(int salary) {  
 this.salary = salary;  
 }  
  
 public int getRating() {  
 return rating;  
 }  
  
 public void setRating(int rating) {  
 this.rating = rating;  
 }  
}

1. **thenApply(Function)/thenApplyAsync(Function), thenAccept(Consumer)/ thenAcceptAsync(Consumer), thenRun(Runnable)/thenRunAsync(Runnable)**

public class EmployeeReminderService {  
  
 public CompletableFuture<Void> sendNewJoinerNotCompletedReminder(){  
  
 CompletableFuture<Void> voidCompletableFuture = CompletableFuture.*supplyAsync*(() -> {  
 System.*out*.println("fetchEmployee : " + Thread.*currentThread*().getName());  
 return *fetchEmployees*();  
 }).thenApplyAsync((employees) -> {  
 System.*out*.println("filter new joiner employee : " + Thread.*currentThread*().getName());  
 return employees.stream()  
 .filter(employee -> "TRUE".equals(employee.getNewJoiner()))  
 .collect(Collectors.*toList*());  
 }).thenApplyAsync((employees) -> {  
 System.*out*.println("filter training not complete employee : " + Thread.*currentThread*().getName());  
 return employees.stream()  
 .filter(employee -> "TRUE".equals(employee.getLearningPending()))  
 .collect(Collectors.*toList*());  
 }).thenApplyAsync((employees) -> {  
 System.*out*.println("get emails : " + Thread.*currentThread*().getName());  
 return employees.stream().map(Employee::getEmail).collect(Collectors.*toList*());  
 }).thenAcceptAsync((emails) -> {  
 System.*out*.println("send email : " + Thread.*currentThread*().getName());  
 emails.forEach(email -> sendEmail(email));  
 });  
 return voidCompletableFuture;  
 }  
  
 public CompletableFuture<Void> sendNewJoinerNotCompletedReminderWithoutAsync(){  
  
 CompletableFuture<Void> voidCompletableFuture = CompletableFuture.*supplyAsync*(() -> {  
 System.*out*.println("fetchEmployee : " + Thread.*currentThread*().getName());  
 return *fetchEmployees*();  
 }).thenApply((employees) -> {  
 System.*out*.println("filter new joiner employee : " + Thread.*currentThread*().getName());  
 return employees.stream()  
 .filter(employee -> "TRUE".equals(employee.getNewJoiner()))  
 .collect(Collectors.*toList*());  
 }).thenApply((employees) -> {  
 System.*out*.println("filter training not complete employee : " + Thread.*currentThread*().getName());  
 return employees.stream()  
 .filter(employee -> "TRUE".equals(employee.getLearningPending()))  
 .collect(Collectors.*toList*());  
 }).thenApply((employees) -> {  
 System.*out*.println("get emails : " + Thread.*currentThread*().getName());  
 return employees.stream().map(Employee::getEmail).collect(Collectors.*toList*());  
 }).thenAccept((emails) -> {  
 System.*out*.println("send email : " + Thread.*currentThread*().getName());  
 emails.forEach(email -> sendEmail(email));  
 });  
 return voidCompletableFuture;  
 }  
  
 public void sendEmail(String email){  
 System.*out*.println("Email sending to email + "+email);  
 }  
  
  
 public static List<Employee> fetchEmployees() {  
 ObjectMapper mapper = new ObjectMapper();  
 try {  
 List<Employee> employees = mapper.readValue(new File("employees.json"), new TypeReference<List<Employee>>() {});  
 System.*out*.println(employees);  
 return mapper.readValue(new File("employees.json"), new TypeReference<List<Employee>>() {});  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return null;  
 }  
  
 public static void main(String[] args) {  
 EmployeeReminderService employeeReminderService = new EmployeeReminderService();  
 employeeReminderService.sendNewJoinerNotCompletedReminder();  
 employeeReminderService.sendNewJoinerNotCompletedReminderWithoutAsync();  
 }  
}

Java8 Interview Questions:

1. What are all features of Java 8 you have used?

* Functional Interface (Include default and static methods)
* Lambda Expression
* Stream
* CompaletableFuture
* Java Date Time API

1. What is Functional Interface?
2. Can you tell few Functional Interface which is already there before Java 8?
3. Can your write one Functional Interface?
4. Can we extends Function Interface from another interface? (Yes, but that will no more FI)
5. What are all Functional Interface Introduced in Java 8?

* Function
* Predicate
* Consumer
* Supplier
* BiFunction
* BiConsumer
* BiPredicate

1. What is Lambda Expression?
2. What are Advantage and Disadvantage of using Lambda Expression?

Advantage:

* Avoid writing anonymous impl
* It saves Lot of code writing burden
* The code is directly readable without interpretation

Disadvantage:

* Hard to use without an IDE
* Complex to debug

1. What is Stream API?

Stream API introduce in Java 8 and it is used to process collections of Objects with functional style of coding using lambda expression.

1. What is Stream in Java 8?

A Stream is sequence of Object that supports various methods which can be pipelined to produce the desired result.

The Feature of Java Stream are:

* A Stream is not a data structure instead its take input from the Collection, Array or I/O channel.
* Stream do not change the original data structure they only provide the result as per the pipelined method.

1. What is method reference in Java 8?

Method reference is short hand Notation of a Lambda expression to call a method.

1. Spell few stream methods you used in your project?

* Filter
* Foreach
* Sorted
* Map
* flatMap
* reduce
* groupingBy(VVI)
* Count
* Collects

1. When to use map and flatMat?
2. WAP using stream to find frequency of each character in given string? - VVI

public class FrequencyOfEachChar {  
 public static void main(String[] args) {  
 String str = "Vikashh";  
 Map<String, Long> countOfEachCharacter = **Arrays.*stream*(str.split("")).collect(Collectors.*groupingBy*(Function.*identity*(), Collectors.*counting*()));**  
 System.*out*.println(countOfEachCharacter);  
 }  
}

1. Assume you have list of employees in various department write a program to highest paid employee from each department?

public class HighestSalaryInEachDept {  
 public static void main(String[] args) {  
  
 List<EmployeeInterviewProgramTest> employeeList =  
 Arrays.*asList*(  
 new EmployeeInterviewProgramTest(1, "Vikash", "IT", 40000.0),  
 new EmployeeInterviewProgramTest(2, "Singh", "SOFT", 80000.0),  
 new EmployeeInterviewProgramTest(3, "Kumar", "BUSINESS", 20000.0),  
 new EmployeeInterviewProgramTest(4, "Sonu", "SECURITY", 90000.0),  
 new EmployeeInterviewProgramTest(5, "Sachin", "IT", 70000.0),  
 new EmployeeInterviewProgramTest(6, "Akash", "SECURITY", 10000.0),  
 new EmployeeInterviewProgramTest(7, "Soumya", "SOFT", 45000.0),  
 new EmployeeInterviewProgramTest(8, "Nikhil", "BUSINESS", 85000.0));  
  
 Map<String, List<EmployeeInterviewProgramTest>> employeeFormEachDepart = employeeList.stream().collect(Collectors.*groupingBy*(EmployeeInterviewProgramTest::getDepartment));  
 System.*out*.println(employeeFormEachDepart);  
  
 *// Finding Highest Salary from each department* Comparator<EmployeeInterviewProgramTest> comparingHighestPaidSalary = Comparator.*comparing*(EmployeeInterviewProgramTest::getSalary);  
 Map<String, Optional<EmployeeInterviewProgramTest>> employeeFormEachDepartWithHighestPaidSalary = employeeList.stream()  
 .collect(Collectors.*groupingBy*(EmployeeInterviewProgramTest::getDepartment, Collectors.*reducing*(BinaryOperator.*maxBy*(comparingHighestPaidSalary))));  
 System.*out*.println(employeeFormEachDepartWithHighestPaidSalary);  
 }  
}

1. Stream vs Parallel Stream in Java 8?

Both are used to process group of object however Stream execute in a single core of machine with sequential flow whereas parallel Stream uses multiple core of machine with parallel flow.

1. What is CompatableFuture?
2. How to Decide ThreadPool Size?

CPU Intensive task

IO intensive task

1. Write a program to print Odd and Even number 2 thread.
2. public class OddEvenBy2Thread {  
     
    private static Object *object* = new Object();  
     
    private static IntPredicate *evenCondition* = e -> e%2 == 0;  
    private static IntPredicate *oddCondition* = e -> e%2 != 0;  
     
    public static void main(String[] args) throws InterruptedException {  
     
    *// Odd number printer* CompletableFuture.*runAsync*(() -> OddEvenBy2Thread.*printNumber*(*oddCondition*));  
     
    *// Even number printer* CompletableFuture.*runAsync*(() -> OddEvenBy2Thread.*printNumber*(*evenCondition*));  
     
    Thread.*sleep*(1000);  
    }  
     
    public static void printNumber(IntPredicate condition){  
    IntStream.*rangeClosed*(1, 10).filter(condition).forEach(OddEvenBy2Thread::*execute*);  
    }  
     
    public static void execute(int num){  
    synchronized (*object*){  
    try{  
    System.*out*.println(Thread.*currentThread*().getName()+" : "+num);  
    *object*.notify();  
    *object*.wait();  
    }catch (InterruptedException e){  
    e.printStackTrace();  
    }  
    }  
     
    }  
   }