13 Java Streams

Problema 1

Given that a method named Double getPrice(String id) exists and may potentially return null, about which of the following options can you be certain that a run time exception will not be thrown

You had to select 2 options

- Optional<Double> price = Optional.of(getPrice("1111"));
    - Optional.of method throws NullPointerException if you try to create an Optional with a null value. If you expect the argument to be null, you should use Optional.ofNullable method, which returns an empty Optional if the argument is null.

- Optional<Double> price = Optional.ofNullable(getPrice("1111"));
  Double x = price.orElse(getPrice("2222"));

- Optional<Double> price = Optional.ofNullable(getPrice("1111"));
  Double y = price.orElseGet(()->getPrice("333"));
    - Optional's orElseGet method takes a java.util.function.Supplier function as an argument and invokes that function to get a value if the Optional itself is empty. Just like the orElse method, this method does not throw any exception even if the Supplier returns null. It does, however, throw a NullPointerException if the Optional is empty and the supplier function itself is null.

- Optional<Double> price = Optional.of(getPrice("1111"), 10.0);
    - This will not compile because Optional.of takes only one argument.

- Optional<Double> price = Optional.of(getPrice("1111"));
  Double z = price.orElseThrow(()->new RuntimeException("Bad Code"));
    - The orElseThrow method takes a Supplier function that returns an Exception. This method is useful when you want to throw a custom exception in case the Optional is empty.

Problema 2

What will the following code print?

```
        List<String> ls = Arrays.asList("Tom Cruise", "Tom Hart", "Tom Hanks", "Tom
Brady");
        Predicate<String> p = str->{
            System.out.println("Looking...");
            return str.indexOf("Tom") > -1;
        };
        boolean flag = ls.stream().filter(str->str.length()>8).allMatch(p);
```

You had to select 1 option

- It will not print anything.
- It will not print anything but will throw an exception at run time.
- `Looking...`
  `Looking...`
  `Looking...`
  `<exception stack trace>`
- `Looking...`
  `Looking...`
  `Looking...`
- `Looking...`
  `Looking...`
  `Looking...`
  `Looking...`
- `Looking…`

Explanation:

Remember that `filter` is an intermediate operation, which means it will not execute until a terminal operation is invoked on the stream.
`allMatch` is a short circuiting terminal operation. Thus, when `allMatch` is invoked, the `filter` method will be invoked and it will keep only those elements in the stream that satisfy the condition given in the filter i.e. the string must be longer than 8 characters. After this method is done, only three elements will be left in the stream.
When `allMatch` is invoked, the code in predicate will be executed for each element until it finds a mismatch. Thus, `Looking...` will be printed three times.

Problema 3

What can be inserted in the code below so that it will print 1 2 3?

```
class MyProcessor{
    Integer value;
    public MyProcessor(Integer value){
        this.value = value;
    }
    public void process(){
        System.out.println(value+" ");
    }
}
public class TestClass {

    public static void main(String[] args) {
        List<Integer> ls = Arrays.asList(1, 2, 3);

        INSERT CODE HERE
    }
}
```

You had to select 1 option

- `ls.stream()`
  `.map(x->MyProcessor::new(x))`
  `.forEach(MyProcessor::process);`
  - You cannot pass arguments to a constructor or method while referring to method/constructor reference. If the constructor or the method requires an argument, it is passed when the constructor or the method is actually invoked. For example, you can do this: names.stream() .map(x->{ Function<Integer, MyProcessor> f = MyProcessor::new; //referring to MyProcessor's constructor that takes one Integer argument. return f.apply(x); //passing the actual Integer argument. }) .forEach(MyProcessor::process);
- `ls.stream()`
- `.map(MyProcessor::new)`
- `.forEach(MyProcessor::process);`
  - Here, map method does have an implicit Integer object in the context, which is in fact the current element of the list. This element will be passed as an argument to the MyProcessor constructor. Similarly, forEach has a MyProcessor object in the context while invoking the process method. Since process is an instance method of MyProcessor, the MyProcessor instance that is available in the context will be used to invoke the process method.
- `ls.stream()`
  `.forEach((x)->process( MyProcessor::new ));`
  - This is syntactically illegal and will not compile.
- `ls.stream()`
  `.forEach(x->MyProcessor::new);`
  - This is syntactically illegal and will not compile. You could do this though: ls.stream().forEach(x->{ new MyProcessor(x).process(); });

Problema 4

Given:

```
public class Course{
    private String id;
    private String category;

    public Course(String id, String category){
        this.id = id; this.category = category;
    }

    public String toString(){
        return id+" "+category;
    }

    //accessors not shown
}
```

What will the following code print?

```
List<Course> s1 = Arrays.asList(
        new Course("OCAJP", "Java"),
        new Course("OCPJP", "Java"),
        new Course("C#", "C#"),
        new Course("OCEJPA", "Java")
);

s1.stream()
       .collect(Collectors.groupingBy(c->c.getCategory()))
       .forEach((m, n)->System.out.println(n));
```

You had to select 1 option

- `C# C#`
  `OCAJP Java, OCPJP Java, OCEJPA Java`

- `[C# C#]`
  `[OCAJP Java, OCPJP Java, OCEJPA Java]`
  - 1. Collectors.groupingBy(Function<? super T,? extends K> classifier) returns a Collector that groups elements of a Stream into multiple groups. Elements are grouped by the value returned by applying a classifier function on an element.
  - 2. It is important to understand that the return type of the collect method depends on the Collector that is passed as an argument. In this case, the return type would be Map<K, List<T>> because that is the type specified in the Collector returned by the groupingBy method.
  - 3. Java 8 has added a default forEach method in Map interface. This method takes a BiConsumer function object and applies this function to each key-value pair of the Map. In this case, m is the key and n is the value.

- 4. The given code provides a trivial lambda expression for BiConsumer that just prints the second parameter, which happens to be the value part of of the key-value pair of the Map.
- 5. The value is actually an object of type List<Course>, which is printed in the output. Since there are two groups, two lists are printed. First list has only one Course element and the second list has three.

- `[C#]`
  `[OCAJP OCPJP OCEJPA]`

- `C#`
  `OCAJP OCPJP OCEJPA`

Explanation:

This is a simple piece of code that illustrates how to group a stream of objects by any given criteria using `Stream`'s `collect` method. There are two important things to understand here:

1. `collect` method requires a `java.util.stream.Collector` object. `Collector` is actually an interface the details of which are not too important for the exam but are good for a clear understanding of how this works. JavaDoc API description explains this very clearly:
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

2. `java.util.stream.Collectors` is a utility class that lets you create various flavors of readymade `Collector` objects.

Problema 5

Which of the following method(s) of java.util.stream.Stream interface is/are used for reduction?

You had to select 2 options

- filter
  - filter(Predicate<? super T> predicate) method returns a stream consisting of the elements of this stream that match the given predicate. However, it is not a reduction operation because it does not combine the elements to produce one value.

- reduce
  - Stream has the following three reduce methods: Optional<T> reduce(BinaryOperator<T> accumulator) Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.

  - T reduce(T identity, BinaryOperator<T> accumulator) Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.

  - <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) Performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.

- sum
  - This is a valid reduction operation but it is not in java.util.stream.Stream interface. Methods to sum up the numbers in a stream are available in specialized Stream interfaces such an IntStream, LongStream, and DoubleStream.

- max
  - min and max are valid reduction operations. The Stream version of these methods take a Comparator as an argument, while the versions in specialized streams such as IntStream and DoubleStream do not take any argument.

- add
  - There is no such method in Stream interface.

Problema 6

Given:

```
String sentence1 = "Carpe diem. Seize the day, boys. Make your lives extraordinary.";
String sentence2 = "Frankly, my dear, I don't give a damn!";
String sentence3 = "Do I look like I give a damn?";
List<String> sentences = Arrays.asList(sentence1, sentence2, sentence3);
```

Which of the following options will create a stream containing all the words in the three sentences without repetition?

You had to select 1 option

```
Stream<String> strm = sentences.stream()
        .flatMap(str->Stream.of(str.split("[ ,.!?\r\n]")))
        .filter(s->s.length()>0)          .distinct();
```

1. The flatMap method replaces one element of a stream with elements of a new stream generated using the original element. Here, the original elements are the sentences. Each of these elements are replaced in the stream with the elements generated by applying str.split("[,.!?\r\n]", which basically converts the stream of sentences into a stream of words. 2. The stream is now filtered and only thoses elements where the length is greater than 0 are allowed to be in the stream. 3. Finally, distinct removes all the duplicates.

```
Stream<String> strm = sentences.stream()
        .map(str->Stream.of(str.split("[ ,.!?\r\n]")))
        .filter(s->s.length()>0)
        .distinct();
```

The map method is meant to replace one element of stream with another. While the flatMap method replaces one element of a stream with elements of a new stream generated using the original element. Therefore, map is not useful here.

```
Stream<String> strm = sentences.stream()
        .forEach(str->Stream.of(str.split("[ ,.!?\r\n]")))
        .filter(s->s.length()>0)
        .distinct();
```

forEach is a terminal operation. You can't chain methods after forEach.

```
Stream<String> strm = sentences.stream()
        .flatMap(str->Stream.of(str.split("[ ,.!?\r\n]")))
        .filter(s->s.length()>0)
        .merge();
```

There is no merge method in Stream.

```
Stream<String> strm = sentences.stream()
        .flatMap(str->str.split("[ ,.!?\r\n]"))
        .filter(s->s.length()>0)
        .distinct();
```

This will not compile because flatMap expects a Function object that returns a Stream. Here, you have a Function object that returns String[].

Problema 7

Given that Book is a valid class with appropriate constructor and getTitle and getPrice methods that return a String and a Double respectively, what can be inserted at //1 and //2 so that it will print the price of all the books having a title that starts with "A"?

```
List<Book> books = Arrays.asList(
        new Book("Atlas Shrugged", 10.0),
        new Book("Freedom at Midnight", 5.0),
        new Book("Gone with the wind", 5.0)
);

Map<String, Double> bookMap = //1 INSERT CODE HERE
//2 INSERT CODE HERE
bookMap.forEach(func);
```

You had to select 1 option

- `books.stream().collect(Collectors.toMap((b->b.getTitle()), b->b.getPrice()));`
  and
  ```
  BiConsumer<String, Double> func = (a, b)->{
              if(a.startsWith("A")){
                  System.out.println(b);
              }
          };
  ```
  - 1. The first line generates a Map<String, Double> from the List using Stream's collect method. The Collectors.toMap method uses two functions to get two values from each element of the stream. The value returned by the first function is used as a key and the value returned by the second function is used as a value to build the resulting Map.
  -
  - 2. The forEach method of a Map requires a BiConsumer. This function is invoked for each entry, that is each key-value pair, in the map. The first argument of this function is the key and the second is the value.

- ```
  books.stream().toMap((b->b.getTitle()), b->b.getPrice()));
  and
  BiConsumer<String, Double> func = (a, b)->{
              if(a.startsWith("A")){
                  System.out.println(b);
              }
          };
  ```
  - toMap is not a valid method in Stream.

- ```
  books.stream().toMap((b->b.getTitle()), b->b.getPrice()));
  and
  BiConsumer<Map.Entry> func = (b)->{
              if(b.getKey().startsWith("A")){
                  System.out.println(b.getValue());
              }
          };
  ```

- 1. toMap is not a valid method in Stream. 2. BiConsumer requires two generic types and two arguments.

- ```
  books.stream().collect(Collectors.toMap((b->b.getTitle()), b->b.getPrice()));
  and
  Consumer<Map.Entry<String, Double>> func = (e)->{
              if(e.getKey().startsWith("A")){
                  System.out.println(e.getValue());
              }
      };
  ```
  - The implementation of Consumer is technically correct. However, the forEach method requires a BiConsumer.
  ```

Problema 8

Which of the following method(s) of java.util.stream.Stream interface is/are used for reduction?
You had to select 2 options

- collect
- count
    - long count() Returns the count of elements in this stream. This is a special case of a reduction and is equivalent to: return mapToLong(e -> 1L).sum(); This is a terminal operation.
- distinct
    - Stream<T> distinct() Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream. For ordered streams, the selection of distinct elements is stable (for duplicated elements, the element appearing first in the encounter order is preserved.) For unordered streams, no stability guarantees are made. This is a stateful intermediate operation.
- findAny
    - This is a valid terminal operation but is not considered a reduction operation.
- findFirst
    - This is a valid terminal operation but is not considered a reduction operation.

Explanation

As per https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html :
A reduction operation (also called a fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation, such as finding the sum or maximum of a set of numbers, or accumulating elements into a list. The streams classes have multiple forms of general reduction operations, called `reduce()` and `collect()`, as well as multiple specialized reduction forms such as `sum()`, `max()`, or `count()`.

For example, you can combine elements of a stream to express complicated queries such as "Calculate the sum of all sales in a given list of transactions" or "Find the highest sales transaction in a list of transactions" using reduction operations. They are called as such because such queries combine all the elements in the stream repeatedly to produce a single value such as a Double. In other words, a stream is reduced to a value by applying these operations.

Note that `filter(Predicate<? super T> predicate)` is not a reduction operation because it does not combine all the elements and produce one value.

Problema 9

Which of the given option when inserted in the code below will make the code print true?

```
List<String> values = Arrays.asList("Alpha A", "Alpha B", "Alpha C");
//INSERT CODE HERE
System.out.println(flag);
```

You had to select 1 option

- boolean flag = values.stream().allMatch(str->str.equals("Alpha"));
- boolean flag = values.stream().findFirst().get().equals("Alpha");
- boolean flag = values.stream().findAny().get().equals("Alpha");
- boolean flag = values.stream().anyMatch(str->str.equals("Alpha"));
- None of the above.
    - Predicate's test in options 1 and 4 and the call to equals in options 2 and 3 always return false, whichever element we apply them to. If we replaced equals() with contains() or startsWith(), they would all be correct.

Problema 10

Given:

```
public class Book {
private int id;
private String title;
private String genre;
private String author;
private double price;

//constructors and accessors not shown
}
```

Assuming that books is a List of Book objects, what can be inserted in the code below at DECLARATION and EXPRESSION so that it will classify the books by genre and then also by author?

```
DECLARATION classified = null;
classified = books.stream().collect(Collectors.groupingBy(
        EXPRESSION
));
System.out.println(classified);
```

You had to select 1 option

- Map<String, Map<String, List<Book>>>
  and
  x->x.getGenre(), x->x.getAuthor()

- Map<String, Map<String, Book>>
  and
  x->x.getGenre(), x->x.getAuthor()

- Map<String, Map<String, List<Book>>>
  and
  Book::getGenre, Collectors.groupingBy(Book::getAuthor)

- Map<String, Map<String, List<Book>>>
  and
  Book::getGenre().groupingBy(Book::getAuthor)

- Map<String, Map<String, List<Book>>>
  and
  Book::getGenre, Collectors.mapping(Book::getAuthor, Collectors.toList())

Problema 11

What will be the result of compilation and execution of the following code ?

```
IntStream is1 = IntStream.range(0, 5); //1
OptionalDouble x = is1.average(); //2
System.out.println(x); //3
```

You had to select 1 option

- It will print 2.5
- It will print 2.0
- It will print OptionalDouble[2.0]
- It will print OptionalDouble[3.0]
- It will print 2.0 if line at //2 is replaced with double x = is1.average();
- It will print 2.5 if line at //2 is replaced with double x = is1.average();

Explanation:

There are three things that you need to watch out for here:

1. The range method includes the starting number but not the ending number. Thus, `range(0, 5)`, will give you numbers `0, 1, 2, 3,` and `4`.
(The `rangeClosed` method includes the ending number also.)

2. The `average` method of all numeric streams (i.e. `IntStream, LongStream,` and `DoubleStream`) returns an `OptionalDouble` and not a `double`. It never returns a `null`. (If there are no elements in the stream, it returns `OptionalDouble.empty` but not 0).
Note that this is unlike the `sum` method which always returns a primitive value of the same type as the type of the stream (i.e. int, long, or double).
In this case, the average of the given 5 numbers is 2, so it returns an `OptionalDouble` containing `2.0`.

3. `OptionalDouble`'s `toString` method returns a `String` of the form OptionalDouble[<double value>]. Therefore, the given code prints `OptionalDouble[2.0]`.

Problema 12

Given:

```
List<String> l1 = Arrays.asList("a", "b");
List<String> l2 = Arrays.asList("1", "2");
```

Which of the following lines of code will print the following output?

```
a
b
1
2
```

You had to select 1 option

- `Stream.of(l1, l2).forEach((x)->System.out.println(x));`
  - This will print :
    ```
    [a, b]
    [1, 2]
    ```

- `Stream.of(l1, l2).flatMap((x)->Stream.of(x)).forEach((x)->System.out.println(x));`
  - This will print :
    ```
    [a, b]
    [1, 2]
    ```

- `Stream.of(l1, l2).flatMap((x)->x.stream()).forEach((x)->System.out.println(x));`
  - The objective of flatMap is to take each element of the current stream and replace that element with elements contained in the stream returned by the Function that is passed as an argument to flatMap. It is perfect for the requirement of this question. You have a stream that contains Lists. So you need a Function object that converts a List into a Stream of elements. Now, List does have a method named stream() that does just that. It generates a stream of its elements. Therefore, the lambda expression x->x.stream() can be used here to create the Function object.

- `Stream.of(l1, l2).flatMap((x)->x.iterator()).forEach((x)->System.out.println(x));`
  - This will not compile because the lambda expression (x)->x.iterator() will return an Iterator but we need a Stream.

Problema 13

What will the following code print?

```
import java.util.Optional;
public class NewClass {
    public static Optional<String> getGrade(int marks){
        Optional<String> grade = Optional.empty();
        if(marks>50){
            grade = Optional.of("PASS");
        }
        else {
            grade.of("FAIL");
        }
        return grade;
    }
    public static void main(String[] args) {
        Optional<String> grade1 = getGrade(50);
        Optional<String> grade2 = getGrade(55);
        System.out.println(grade1.orElse("UNKNOWN"));
        if(grade2.isPresent()){
            grade2.ifPresent(x->System.out.println(x));
        }else{
            System.out.println(grade2.orElse("Empty"));
        }
    }
}
```

You had to select 1 option

- UNKNOWN
  PASS
- Optional[UNKNOWN]
  PASS
- Optional[UNKNOWN]
  Optional[PASS]
- FAIL
  PASS
- Optional[FAIL]
  OPTIONAL[PASS]

Explanation

You should go through the following article about java.util.Optional:
http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html

Here are a few important things you need to know about Optional class:

1. Optional has a static method named `of(T t)` that returns an Optional object containing the value passed as argument. It will throw `NullPointerException` if you pass `null`. If you want to avoid `NullPointerException`, you should use `Optional.ofNullable(T t)` method. This will return `Optional.empty` if you pass `null`.

2. You cannot change the contents of Optional object after creation. Optional does not have a set method. Therefore, `grade.of`, although technically correct, will not actually change the Optional object referred to by `grade`. It will return a new Optional object containing the passed argument.

3. The `orElse` method returns the actual object contained inside the Optional or the argument passed to this method if the Optional is empty. It does not return an Optional object. Therefore, `print(grade1.orElse("UNKNOWN"))` will print `UNKNOWN` and not `Optional[UNKNOWN]`.

4. `isPresent()` returns true if the Optional contains a value, false otherwise.

5. `ifPresent(Consumer )` executes the Consumer object with the value if the Optional contains a value. Not that it is the value contained in the Optional that is passed to the Consumer and not the Optional itself.

Problema 14

Assuming that Book has appropriate constructor and accessor methods, what will the following code print?

```
List<Book> books = Arrays.asList(
        new Book("Freedom at Midnight", 5.0),
        new Book("Gone with the wind", 5.0),
        new Book("Midnight Cowboy", 15.0)
);

books.stream()
          .filter(b->b.getTitle().startsWith("F"))
          .forEach(b->b.setPrice(10.0));
books.stream()
          .forEach(b->System.out.println(b.getTitle()+":"+b.getPrice()));
```

You had to select 1 option

- `Freedom at Midnight:5.0`
  `Gone with the wind:5.0`
  `Midnight Cowboy:15.0`

- `Freedom at Midnight:5.0`
  `Gone with the wind:10.0`
  `Midnight Cowboy:10.0`

- `Freedom at Midnight:10.0`
  `Gone with the wind:5.0`
  `Midnight Cowboy:15.0`

- `Freedom at Midnight:10.0`

- `Exception at run time`

Explanation:

This code illustrates how you can go through a list of elements, filter the elements based on a criteria, and call methods on each of the elements in the filtered list.

filter method removes all the elements for which the given condition (i.e. `b.getTitle().startsWith("F")`) returns false from the stream. These elements are not removed from the underlying list but only from the stream. Therefore, when you create a stream from the list again, it will have all the elements from the list. Since the setPrice operation changes the Book object contained in the list, the updated value is shown the second time when you go through the list.

Problema 15

What will the following code print when compiled and run?

```
List<Integer> ls = Arrays.asList(1, 2, 3);
ls.stream().forEach(System.out::print)
   .map(a->a*2)
   .forEach(System.out::print);
```

You had to select 1 option

- 123
- 246
- Compilation failure
    - Remember that forEach is a terminal operation. It returns void. This means that you cannot chain methods after calling forEach.
- exception at run time

Problema 16

Given:

```
Stream<String> names = Stream.of("Sarah Adams", "Suzy Pinnell", "Paul Basgall");
Stream<String> firstNames = //INSERT CODE HERE
```

Which of the following options will correctly assign a stream of just first names to firstNames?

You had to select 1 option

- `names.map(e->e.split(" ")[0]);`
    - public <R> Stream<R> map(Function<? super T,? extends R> mapper) Returns a stream consisting of the results of applying the given function to the elements of this stream.

- `names.reduce(e->e.split(" ")[0]);`
    - Stream has several flavors of reduce methods. These methods are used to reduce the stream into one value using a BiFunction or BinaryOperator. It does not return a Stream. Note: BinaryOperator extends BiFunction. BinaryOperator is just a specialization of BiFunctionfor situations where types of the parameters and the return value are same.

- `names.filter(e->e.split(" ")[0]);`
    - filter method expects a Predicate object as argument. It is meant to filter the elements of the stream based on the given Predicate. In other words, filtering a stream implies that you are reducing the number of elements in the resulting stream. You are not modifying the elements.

- `names.forEach(e->e.split(" ")[0]);`
    - forEach method expects a Consumer object as argument. It is meant to process each element but doesn't return anything.

Problema 17

Which of the following elements is/are a must in a stream pipeline?

You had to select 2 options

- a source
- an intermediate operation
- a terminal operation
- a reduction operation
- a method reference
- a lambda expression

Explanation:
A pipeline contains the following components:
A source: This could be a collection, an array, a generator function, or an I/O channel. In this example, the source is the collection roster.
Zero or more intermediate operations: An intermediate operation, such as filter, produces a new stream.
A terminal operation.

Problema 18

What will the following code print when compiled and run?

```
class Names{
    private List<String> list;
    public List<String> getList() {
        return list;
    }
    public void setList(List<String> list) {
        this.list = list;
    }
    public void printNames(){
        System.out.println(getList());
    }

    public static void main(String[] args) {
       List<String> list = Arrays.asList(
        "Bob Hope",
        "Bob Dole",
        "Bob Brown"
       );

       Names n = new Names();
       n.setList(list.stream().collect(Collectors.toList()));
       n.getList().forEach(Names::printNames);
   }

}
```

You had to select 1 option

- `[Bob Hope, Bob Dole, Bob Brown]`
- `Bob HopeBob DoleBob Brown`
- `[null, null, null]`
- Compilation error
    - Observe that the method printNames does not take any argument. But the argument for forEach method requires a method that takes an argument. The forEach method basically invokes the passed method and gives that method an element of the list as an argument. The following is how you can visualise the working of the given code: Iterator it = list.iterator(); while(it.hasNext()){ Names.printNames(it.next()); //This will not compile because printNames method does not take any argument. } To make it work, the line n.getList().forEach(Names::printNames); should be changed to simply: n.printNames();
- An exception will be thrown at run time.

Problema 19

Given:

```
List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13, 17); //1
Stream<Integer> primeStream = primes.stream(); //2

Predicate<Integer> test1 = k->k<10; //3
long count1 = primeStream.filter(test1).count();//4

Predicate<Integer> test2 = k->k>10; //5
long count2 = primeStream.filter(test2).count(); //6

System.out.println(count1+" "+count2); //7
```

Identify correct statements.

You had to select 1 option

- `It will print 4 3 if line at //6 is replaced with:`
  `long count2 = primeStream.reset().filter(test2).count(); //6`
  - There is no reset method in Stream interface. Once a terminal operation is invoked on a Stream, it is considered closed. You cannot do anything with it. You have to create a new Stream.

- `It will print 4 3 if types of count1 and count2 are changed to int.`
  - Stream.count() returns long. So the types of count1 and count2 are fine.

- `It will print 4 3 if line //4 and //6 are replaced with:`
  `long count1 = IntStream.of(primes).filter(test1).count();//4`
  `and`
  `long count2 = IntStream.of(primes).filter(test2).count();//6`
  - This change will fail to compile because IntStream.of does not take a List as argument. It takes either an int or a varargs parameter of type int. Further, the filter method of an IntStream takes IntPredicate as an argument, while test1 and test2 are of type Predicate<Integer>.

- `It will print 34 if lines 4 to 7 are replaced with:`
- `primeStream.collect(Collectors.partitioningBy(test1, Collectors.counting()))`
- `.values().forEach(System.out::print);`

Explanation:

The only problem with the given code is that a stream cannot be reused once a terminal operation has been invoked on it. Therefore, line 6 will throw `java.lang.IllegalStateException: stream has already been operated upon or closed`.

Problema 20

What will the following code print?

```
List<Integer> ls = Arrays.asList(1, 2, 3);
Function<Integer, Integer> func = a->a*a; //1
ls.stream().map(func).peek(System.out::print); //2
```

You had to select 1 option

- Compilation error at //1
- Compilation error at //2
- 149
- 123
- It will compile and run fine but will not print anything.
    - Remember that none of the "intermediate" operations on a stream are executed until a "terminal" operation is called on that stream. In the given code, there is no call to a terminal operation. map and peek are intermediate operations. Therefore, the map function will not be invoked for any of the elements.
    - If you append a terminal operation such as count(), [for example, ls.stream().map(func).peek(System.out::print).count();], it will print 149.
    - Note that you can invoke at most one terminal operation on a stream and that too at the end.

Problema 21

Given:

```
List<Integer> ls = Arrays.asList(11, 11, 22, 33, 33, 55, 66);
```

Which of the following expressions will return true?
You had to select 2 options

- `ls.stream().anyMatch(44);`
  - This will not compile because anyMatch requires a Predicate object as an argument, not an int.

- `ls.stream().anyMatch(11);`
  - This will not compile because anyMatch requires a Predicate object as an argument, not an int.

- `ls.stream().distinct().anyMatch(x->x==11);`
  - anyMatch(Predicate<? super T> predicate) returns whether any elements of this stream match the provided predicate. May not evaluate the predicate on all elements if not necessary for determining the result. If the stream is empty then false is returned and the predicate is not evaluated. This is a short-circuiting terminal operation.

- `ls.stream().distinct().allMatch(11);`
  - This will not compile because allMatch requires a Predicate object as an argument, not an int. You could do ls.stream().distinct().allMatch(x->x==11); but it will return false because for allMatch to return true, the given predicate must return true for each element of the stream.

- `ls.stream().noneMatch(x->x%11>0);`
  - noneMatch returns true only if none of the elements in the stream satisfy the given Predicate. Here, all the elements are divisible by 11 and x%11 will be 0 for each element. Therefore, the given Predicate will return false for every element, causing noneMatch to return true.

Problema 22

What will the following code print when compiled and run?

```
//imports not shown
class Movie{
    static enum Genre  {DRAMA, THRILLER, HORROR, ACTION };
    private Genre genre;
    private String name;
    private char rating = 'R';
    Movie(String name, Genre genre, char rating){
        this.name = name; this.genre = genre; this.rating = rating;
    }
    //accessors not shown
}
public class FilteringStuff {
    public static void main(String[] args) {
        List<Movie> movies = Arrays.asList(
                new Movie("Titanic", Movie.Genre.DRAMA, 'U'),
                new Movie("Psycho", Movie.Genre.THRILLER, 'U'),
                new Movie("Oldboy", Movie.Genre.THRILLER, 'R'),
                new Movie("Shining", Movie.Genre.HORROR, 'U')
                );

        movies.stream()
                .filter(mov->mov.getRating()=='R')
                .peek(mov->System.out.println(mov.getName()))
                .map(mov->mov.getName());
    }

}
```

You had to select 1 option

- It will not print anything.
- Titanic
  Psycho
  Oldboy
  Oldboy
  Shining
- Oldboy
  Oldboy
- Oldboy
- It will throw an exception at run time.

Explanation:

To answer this question, you need to know two things - distinction between "intermediate" and "terminal" operations and which operations of Stream are "intermediate" operations.

A Stream supports several operations and these operations are divided into intermediate and terminal operations. The distinction between an intermediate operation and a termination operation is that an intermediate operation is lazy while a terminal operation is not. When you invoke an intermediate operation on a stream, the operation is not executed immediately. It is executed only when a terminal operation is invoked on that stream. In a way, an intermediate operation is memorized and is recalled as soon as a terminal operation is invoked. You can chain multiple intermediate operations and none of them will do anything until you invoke a terminal operation, at which time, all of the intermediate operations that you invoked earlier will be invoked along with the terminal operation.

You should read more about this here:
http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#StreamOps

It is easy to identify which operations are intermediate and which are terminal. All intermediate operations return Stream (that means, they can be chained), while terminal operations don't.

In the given questions, filter, peek, and map are intermediate operations. Since the code does not invoke any terminal operation on the stream, the calls to these intermediate method do nothing. Therefore, no output is produced.

If you add a call to `forEach(System.out::println)` (which is a terminal operation) at the end, all the intermediate operations will be performed and you will see the following output -

```
Oldboy
Oldboy
```

Problema 23

What will the following code print when compiled and run?

```
Stream<Integer> values = IntStream.rangeClosed(10, 15).boxed(); //1
Object obj = values.collect(Collectors.partitioningBy(x->x%2==0)); //2
System.out.println(obj);
```

You had to select 1 option

- Compilation error at //1
- Compilation error at //2
- `{[11, 13, 15], [10, 12, 14]}`
- `[[11, 13, 15], [10, 12, 14]]`
- `{false=[11, 13, 15], true=[10, 12, 14]}`

Explanation:

This code illustrates the use of `Collectors.partitioningBy` method. This method takes a `Predicate` and returns `Collector` that distributes the elements of the stream into two groups - one containing elements for which the `Predicate` returns `true`, and another containing elements for which the `Predicate` returns `false`. The return type is a `Map` containing two keys - `true` and `false` and the values are Lists of the elements.

`IntStream.rangeClosed(10, 15)` creates an IntStream of int primitives containing elements 10, 11, 12, 13, 14, and 15 (Observe that 15 is included). `IntStream` does not support the various collect methods supported by a regular `Stream` of objects. But it does support a `boxed()` method that returns a `Stream<Integer>` containing `Integer` objects.

Problema 24

Given:

```
public class Book {
   private String title;
   private String genre;
   public Book(String title, String genre){
        this.title = title; this.genre = genre;
   }

   //accessors not shown
}

and the following code:

List<Book> books = Arrays.asList(
        new Book("Gone with the wind", "Fiction"),
        new Book("Bourne Ultimatum", "Thriller"),
        new Book("The Client", "Thriller")
);

List<String> genreList = new ArrayList<>();
//INSERT CODE HERE
System.out.println(genreList);
```

Which of the following options will correctly make genreList refer to a List containing the genres of the books present in books List?

You had to select 4 options

- `books.stream().map(Book::getGenre).forEach(s->genreList.add(s));`
    - 1. map(Book::getGenre) will replace each element of the Stream with the value returned by calling getGenre method on that element. Thus, the stream will now contain genre values. 2. The forEach method expects a Consumer instance, which is correctly captured by the lambda expression s->genreList.add(s). This expression adds each element to genreList.
- `genreList = books.stream().map(Book::getGenre).collect(Collectors.toList());`
    - 1. map(Book::getGenre) will replace each element of the Stream with the value returned by calling getGenre method on that element. Thus, the stream will now contain genre values. 2. The Collector returned by Collectors.toList() will cause the collect method to return a List of elements in the stream. This list is assigned to genreList.
- `books.stream().map(Book::getGenre).collect(Collectors.toList(genreList));`
    - Collectors.toList doesn't take any argument.
- `books.stream().map(Book::getGenre).forEach(genreList::add);`
- `books.stream().map(b->b.getGenre()).forEach(genreList::add);`
- `books.stream().flatMap(b->b.getGenre()).forEach(g->genreList.add(g));`
    - flatMap is used when each element of a given stream can itself generate a Stream of objects. The purpose of this method is to extract the elements of each of those individual streams and return a stream that contains all those elements.

Problema 25

What will the following code print when compiled and run?

```
IntStream is1 = IntStream.range(1, 3);
IntStream is2 = IntStream.rangeClosed(1, 3);
IntStream is3 = IntStream.concat(is1, is2);
Object val = is3.boxed().collect(Collectors.groupingBy(k->k)).get(3);
System.out.println(val);
```

You had to select 1 option

- `[1]`
- `[3]`
- `[3, 3]`
- `[2]`
- `[1, 2]`

Explanation:

There are several things going on here:

1. `IntStream.range` returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1. Therefore, `is1` contains 1, 2.
2. `IntStream.rangeClosed` returns a sequential ordered IntStream from startInclusive (inclusive) to endInclusive (inclusive) by an incremental step of 1. Therefore, `is2` contains 1, 2, 3.
3. `IntStream.concat` returns a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. Therefore, `is3` contains 1, 2, 1, 2, 3.
4. `is3` is a stream of primitive ints. `is3.boxed()` returns a new `Stream` containing `Integer` objects instead of primitives. This allows the use various flavors of `collect` method available in non-primitive streams. [`IntStream` does have one collect method but it does not take a `Collector` as argument.]
5. `Collectors.groupingBy(k->k)` creates a `Collector` that groups the elements of the stream by a key returned by the function `k->k`, which is nothing but the value in the stream itself. Therefore, it will group the elements into a `Map<Integer, List<Integer>>` containing: `{1=[1, 1], 2=[2, 2], 3=[3]}`
6. Finally, `get(3)` will return `[3]`.

Problema 26

Given:

```
public class Student {

    public static enum Grade{ A, B , C, D, F}

    private String name;
    private Grade grade;
    public Student(String name, Grade grade){
        this.name = name;
        this.grade = grade;
    }
    public String toString(){
        return name+":"+grade;
    }
    //getters and setters not shown
}

What can be inserted in the code below so that it will print:
{C=[S3], A=[S1, S2]}


List<Student> ls = Arrays.asList(new Student("S1", Student.Grade.A), new Student("S2",
Student.Grade.A), new Student("S3", Student.Grade.C));
//INSERT CODE HERE
System.out.println(grouping);
```

You had to select 1 option

- ```
  Map<Student.Grade, List<Student>> grouping = ls.stream().collect(
            Collectors.groupingBy(Student::getGrade),
            Collectors.groupingBy(Student::getName, Collectors.toList())));
  ```
  - Invalid arguments to the collect method.
  - Remember that Stream has only two overloaded collect methods - one that takes a Collector as an argument and another one that takes a Supplier, BiConsumer, and BiConsumer. In this option, it is trying to pass two Collectors to the collect method. Therefore, it will not compile.
  - 1. public <R,A> R collect(Collector<? super T,A,R> collector) Performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to Stream.collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.
  - 2.public <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) Performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result.
  ```

- ```
  Map<Student.Grade, List<String>> grouping = ls.stream().collect(
          Collectors.groupingBy(Student::getGrade,
              Collectors.groupingBy(Student::getName, Collectors.toList())));
  ```
  - The right hand side of = is actually okay from a compilation perspective. It is trying to group the elements of the stream by Grade and then it is again trying to groups the elements of each grade by name. So technically, the return type of this expression would be: Map<Student.Grade, Map<String, List<Student>>> grouping = ...
  - Even if you change the left hand side declaration as described, it will only print {C={S3=[S3:C]}, A={S1=[S1:A], S2=[S2:A]}}, which is not what is required by the question.

- ```
  Map<Student.Grade, List<String>> grouping = ls.stream().collect(
      Collectors.groupingBy(Student::getGrade,
      Collectors.mapping(Student::getName, Collectors.toList())));
  ```
  - This code illustrates how to cascade Collectors. Here, you are first grouping the elements by Grade and then collecting each element of a particular grade into a list after mapping it to a String. This will produce the required output.

- ```
  Map<Student.Grade, List<String>> grouping = ls.stream().collect(
                  Collectors.groupingBy(Student::getGrade,
                  Collectors.mapping(Student::getName)));
  ```
  - Collectors.mapping method requires two arguments - the first argument must be a Function that maps one element type into another (here, you are mapping Student to String, which is good), and the second argument must be an appropriate Collector in which you can hold the result (here, this argument is missing). Therefore, it will not compile.
  ```

Problema 27

Identify the correct statements about Java Stream API.
You had to select 1 option

- Streams are reusable.
    - They are not. Once you process a Stream, you cannot do anything with it (except call close on it). For example, the following code will throw "java.lang.IllegalStateException: stream has already been operated upon or closed": List<Integer> ls = Arrays.asList(1, 2, 3); Stream stream = ls.stream(); long count = stream.count(); //returns 3 count = stream.count(); //throws exception

- Elements of a stream are mutable.
    - A stream has no bearing on the type of the elements that it contains. Whether the elements are mutable or not depends on the class of the objects.

- Streams support aggregate operations.
    - Streams support several aggregate operations such as forEach, count, average, and sum. You should go through this link to learn more: https://docs.oracle.com/javase/tutorial/collections/streams/

- All Stream operations are lazy.
    - Only non-terminal operations such as filter, peek, map, distinct, sorted, and skip are lazy. All terminal operations such as forEach, reduce, collect, count, min, max, allMatch, findAny are eager.

Problema 28

Given:

```
class MyProcessor{
    public void process(){
        System.out.println("Processing ");
    }
}
public class TestClass {

    public static void main(String[] args) {
        process(()->new MyProcessor()); //1 REPLACE THIS LINE OF CODE
    }

    public static void process(Supplier<MyProcessor> s){
        s.get().process();
    }
}
```

Which of the following options correctly replaces the line marked //1 with code that uses method reference?

You had to select 1 option

- TestClass.process(MyProcessor::new);
    - This is the correct syntax for referring to a constructor.

- TestClass.process(new::MyProcessor);
- TestClass.process(MyProcessor::new());
- TestClass.process(new::MyProcessor());
- TestClass.process(MyProcessor()::new);

Explanation

Observe that the TestClass's static method process takes `Supplier<MyProcessor>` as an argument. The lambda expression //()->new MyProcessor()// creates a Supplier with the implementation of its functional method named get that just returns a new MyProcessor object. If it makes it easier to understand, you can think of this lambda expression in terms of an anonymous class like this:

```
new Supplier(){
   public MyProcessor get(){
       return new MyProcessor(); //the body of a lambda expression becomes
the body of the functional method of the functional interface.
   }
}
```

The expression `new MyProcessor()` in the lambda expression can be replaced by `MyProcessor::new`.

Problema 29

What will be the result of compilation and execution of the following code ?

```
IntStream is1 = IntStream.range(0, 5); //1
OptionalDouble x = is1.average(); //2
System.out.println(x); //3
```

You had to select 1 option

- It will print 2.5
- It will print 2.0
- It will print OptionalDouble[2.0]
- It will print OptionalDouble[3.0]
- It will print 2.0 if line at //2 is replaced with double x = is1.average();
- It will print 2.5 if line at //2 is replaced with double x = is1.average();

Explanation

There are three things that you need to watch out for here:

1. The range method includes the starting number but not the ending number. Thus, `range(0, 5)`, will give you numbers `0, 1, 2, 3,` and `4`.
(The `rangeClosed` method includes the ending number also.)

2. The `average` method of all numeric streams (i.e. `IntStream, LongStream,` and `DoubleStream`) returns an `OptionalDouble` and not a `double`. It never returns a `null`. (If there are no elements in the stream, it returns `OptionalDouble.empty` but not 0).
Note that this is unlike the `sum` method which always returns a primitive value of the same type as the type of the stream (i.e. int, long, or double).
In this case, the average of the given 5 numbers is 2, so it returns an `OptionalDouble` containing `2.0`.

3. `OptionalDouble`'s `toString` method returns a `String` of the form OptionalDouble[<double value>]. Therefore, the given code prints `OptionalDouble[2.0]`.

Problema 30

Given:

```
class Course{
    private String id;
    private String name;

    public Course(String id, String name){
        this.id = id; this.name = name;
    }
    //accessors not shows
}
```

What will the following code print?

```
List<Course> cList = Arrays.asList(
        new Course("803", "OCAJP 7"),
        new Course("808", "OCAJP 8"),
        new Course("809", "OCPJP 8")
);

cList.stream().filter(c->c.getName().indexOf("8")>-1)
        .map(c->c.getId())
        .collect(Collectors.joining("1Z0-"));
cList.stream().forEach(c->System.out.println(c.getId()));
```

You had to select 1 option

- 803
  808
  809

- 803
  1Z0-808
  1Z0-809

- 1Z0-808
  1Z0-809

- It will throw an exception at run time.

- It will not compile.

Explanation:

There are multiple flavors of `Collectors.joining` method and all of them are meant to join `CharSequence`s and return the combined `String`. For example, if you have a List of Strings, you could join all the elements into one long `String` using the `Collectors` returned by these methods. You should check their JavaDoc API description for details.

The given code ostensibly tries to apply `"1Z0-"` as prefix to the `id` value of each course that has a name containing `"8"`, but that is not what the code actually does. It first filters the stream (removing the elements that don't satisfy the condition, which means the stream now contains only two `Course` objects), then replaces each `Course` element with a String element (containing just the id, which means the stream now has two Strings `"808"` and `"809"`), and then joins each of the elements with `"1Z0-"` as delimiter to return `"8081Z0-809"`. However, this return value is lost because it is not assigned to any thing.

The next line creates a new stream using the original List of `Course` objects and prints the `id` for each Course. This prints the three ids values `803, 808,` and `809.`

Problema 31

Given:

```
List<Integer> ls = Arrays.asList(1, 2, 3);
```

Which of the following options will compute the sum of all Integers in the list correctly?

You had to select 2 options

- `double sum = ls.stream().sum();`
  - There no sum method in Stream. There is one in IntStream and DoubleStream.

- `double sum = ls.stream().reduce(0, (a, b)->a+b);`
  - The reduce method performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.

- `double sum = ls.stream().mapToInt(x->x).sum();`

- ```
  double sum = 0;
  ls.stream().forEach(a->{ sum=sum+a; });
  ```
  - This code is almost correct but for the fact that only final or effective final local variables can be used in a lambda expression. Here, the code is trying to use sum and sum is not final.
  - Effectively final means that even though it is not declared as final, it is not assigned any value anywhere else after the first assignment. The compiler determines that this variable never changes and considers it as final.

- `double sum = 0; ls.stream().peek(x->{sum=sum+x;}).forEach(y->{});`
  - This has the same problem as above. sum is not final or effectively final.
  ```

Problema 32

Given:

```
Stream<Integer> strm1 = Stream.of(2, 3, 5, 7, 11, 13, 17, 19); //1
Stream<Integer> strm2 = strm1.filter(i->{ return i>5 && i<15; });  //2
strm2.forEach(System.out::print); //3
```

Which of the following options can be used to replace line at //2 and still print the same elements of the stream?

You had to select 1 option

- `Stream<Integer> strm2 = strm1.filter(i>5).filter(i<15);`
    - This will not compile because i>5 and i<15 are invalid lambda expressions.

- `Stream<Integer> strm2 = strm1.parallel().filter(i->i>5).filter(i->i<15).sequential();`
    - Stream pipelines may execute either sequentially or in parallel. This execution mode is a property of the stream. Streams are created with an initial choice of sequential or parallel execution. (For example, Collection.stream() creates a sequential stream, and Collection.parallelStream() creates a parallel one.) This choice of execution mode may be modified by the BaseStream.sequential() or BaseStream.parallel() methods. It is not documented by Oracle exactly what happens when you change the stream execution mode multiple times in a pipeline. It is not clear whether it is the last change that matters or whether operations invoked after calling () parallel can be executed in parallel and operations invoked after calling sequential() will be executed sequentially.

- ```
  Stream<Integer> strm2 = strm1.collect(
                Collectors.partitioningBy(i->{ return i>5 && i<15; })
             ).get("true").stream();
  ```
    - This is almost correct but for the fact that the keys in the Map produced by partitioningBy Collector are Boolean and not String. Therefore, get("true") will return null. It should have been get(true).

- `Stream<Integer> strm2 = strm1.map(i-> i>5?i<15?i:null:null);`
    - It will print nullnullnull71113nullnull. This is not what we want.
  ```

Problema 33

Given:

```
class Item{
    private int id;
    private String name;
    public Item(int id, String name){
        this.id = id;
        this.name = name;
    }
    public Integer 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 toString(){
      return name;
    }
}


public class Test {
    public static void main(String[] args) {
        List<Item> l = Arrays.asList(
                new Item(1, "Screw"),
                new Item(2, "Nail"),
                new Item(3, "Bolt")
        );

       l.stream()
//INSERT CODE HERE
      .forEach(System.out::print);
    }
}
```

Which of the following options can be inserted in the above code independent of each other, so that the code will print BoltNailScrew?

You had to select 2 options

- `.sorted((a, b)->a.getId().compareTo(b.getId()))`
  - This option creates a Comparator using a lambda expression that compares two Item objects for their id attribute. Syntactically, this option is correct but we need to sort by name instead of id.

- `.sorted(Comparator.comparing(a->a.getName())).map((i)->i.getName())`
  - 1. This option uses Comparator's comparing method that accepts a function that extracts a Comparable sort key, and returns a Comparator that compares by that sort key. Note that this is helpful only if the type of the object returned by the function implements Comparable. Here, it returns a String, which does implement Comparable and so it is ok.
  - 2. Although the map part is not required because Item class overrides the toString method to print the name anyway, it is valid.

- `.map((i)->i.getName())`
  - Just mapping the Items to their names will not help because we need to sort the elements as well.

- `.map((i)->i.getName()).sorted()`
  - 1. The call to map converts the stream of Items to a stream of Strings. 2. The call to sorted() sorts the stream of String by their natural order, which is what we want here.

Problema 34

What will the following code print?

```
Map<String, Integer> map1 = new HashMap<>();
map1.put("a", 1);
map1.put("b", 1);
map1.merge("b", 1, (i1, i2)->i1+i2);
map1.merge("c", 3, (i1, i2)->i1+i2);
System.out.println(map1);
```

You had to select 1 option

- {a=1, b=2, c=3}
- {a=1, b=1, c=3}
- {a=1, b=2}
- A NullPointerException will be thrown at run time.

Explanation:

The JavaDoc API description explains exactly how the merge method works. You should go through it as it is important for the exam.

```
public V merge(K key, V value, BiFunction<? super V,? super V,? extends V>
remappingFunction)
```

If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null. This method may be of use when combining multiple mapped values for a key. For example, to either create or append a String msg to a value mapping:
map.merge(key, msg, String::concat)

If the function returns null the mapping is removed. If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.

Parameters:
key - key with which the resulting value is to be associated value - the non-null value to be merged with the existing value associated with the key or, if no existing value or a null value is associated with the key, to be associated with the key remappingFunction - the function to recompute a value if present

Returns:
the new value associated with the specified key, or null if no value is associated with the key

Throws:
UnsupportedOperationException - if the put operation is not supported by this map (optional)
ClassCastException - if the class of the specified key or value prevents it from being stored in this map (optional)
NullPointerException - if the specified key is null and this map does not support null keys or the value or remappingFunction is null

Problema 35