10 Java

Lambda Expressions & Functional Interfaces

Problema 1

Given :

```
//In Data.java
public class Data{
    int value;
    Data(int value){
        this.value = value;
    }
}

and the following code fragments:
public void printUsefulData(ArrayList<Data> dataList, Predicate<Data> p){
   for(Data d: dataList){
        if(p.test(d)) System.out.println(d.value);
   }
}

....
        ArrayList<Data> al = new ArrayList<Data>();
        al.add(new Data(1));al.add(new Data(2));al.add(new Data(3));

        //INSERT METHOD CALL HERE
```

Which of the following options can be inserted above so that it will print 3?

You had to select 2 options

- printUsefulData(al, (Data d)-> { return d.value>2; });
- printUsefulData(al, d-> d.value>2);
    - 1. Compiler already knows the parameter types, so Data can be omitted from the parameter list. 2. When there is only one parameter in the method, you can omit the brackets because the compiler can associate the -> sign with the parameter list without any ambiguity. 3. When all your method does is return the value of an expression, you can omit the curly braces, the return keyword, and the semi-colon from the method body part. Thus, instead of { return d.value>2; }, you can just write d.value>2
- printUsefulData(al, (d)-> return d.value>2;);
    - If you write return, the compiler assumes that you are writing the complete method body and so it expects the curly braces as well as the semi-colon.
- printUsefulData(al, Data d-> d.value>2);
    - If you write parameter type, the compiler assumes that you are writing the complete parameter list of the method and so it expects the brackets i.e. (Data d) instead of just Data d.
- printUsefulData(al, d -> d.value>2;   );
    - The semi-colon in the method body should not be there because the line of code is not enclosed within curly braces.

Problema 2

Given :

```
interface Process{
    public void process(int a, int b);
}

public class Data{
    int value;
    Data(int value){
        this.value = value;
    }
}

and the following code fragments:
public void processList(ArrayList<Data> dataList, Process p){
   for(Data d: dataList){
        p.process(d.value, d.value);
   }
}

....
        ArrayList<Data> al = new ArrayList<Data>();
        al.add(new Data(1));al.add(new Data(2));al.add(new Data(3));

        //INSERT METHOD CALL HERE
```

Which of the following options can be inserted above so that it will print 1 4 9?

You had to select 3 options

- `processList(al, a, b->System.out.println(a*b));`
  - Observe that without the brackets over a, b, it would imply that you are trying to pass 3 arguments to processList method - a, b, and b->System.out.println(a*b), which is incorrect. You actually want to pass only two arguments - a and the lambda expression. Therefore, whenever the method of a functional interface takes more than one parameter, you need to put the arguments within brackets.
  - If the method of a functional interface takes one parameter, you can omit the brackets. For example, x -> expression and (x) -> expression are equivalent.
  - If the method of a functional interface takes no parameter, you must write empty brackets. For example, ( ) -> expression
- `processList(al, (int a, int b)->System.out.println(a*b) );`
- `processList(al, (int a, int b)->System.out.println(a*b); );`
  - When your method body comprises only a single expression, you must omit the semi-colon.

- `processList(al, (a, b)->System.out.println(a*b));`
    - It is ok to omit the parameter types in case of a functional interface because the compiler can determine the type of the parameters by looking at the interface method.
- `processList(al, (a, b) ->{  System.out.println(a*b); } );`
    - If you enclose your method body within curly braces, you must write complete lines of code including the semi-colon. FYI, if the method is supposed to return a value, then you must include a return statement just like you do in a regular method if you are using the curly braces syntax.

Explanation:

There is a simple trick to identify invalid lambda constructs. When you write a lambda expression for a functional interface, you are essentially providing an implementation of the method declared in that interface but in a very concise manner. Therefore, the lambda expression code that you write must contain all the pieces of the regular method code except the ones that the compiler can easily figure out on its own such as the parameter types, return keyword, and brackets. So, in a lambda expression, just check that all the information is there and that the expression follows the basic syntax -

```
(parameter list) OR single_variable_without_type ->
    { regular lines of code } OR just_an_expression_without_semicolon
```

Problema 3

Given:

```
import java.util.*;
class Data{
    int value;
    public Data(int x){ this.value = x; }
    public String toString(){ return ""+value; }
}

class MyFilter {
  public boolean test(Data d){
     return d.value == 0;
  }
}

public class TestClass{

   public static void filterData(ArrayList<Data> dataList, MyFilter f){
      Iterator<Data> i = dataList.iterator();
      while(i.hasNext()){
           if(f.test(i.next())){
                i.remove();
           }
       }
   }

  public static void main(String[] args) {
        ArrayList<Data> al = new ArrayList<Data>();
        Data d = new Data(1); al.add(d);
        d = new Data(2); al.add(d);
        d = new Data(0); al.add(d);

        filterData(al, new MyFilter());  //1

        System.out.println(al);
    }
}
```

How can you use a lambda expression to achieve the same result?

You had to select 1 option

- `Replace the line at //1 with:`
  `filterData(al, x -> x.value==0);`
    - This would not work because MyFilter is not a functional interface.
- Add `implements java.util.function.Predicate to MyFilter` definition and replace the line at `//1` with:
  `filterData(al, x -> x.value==0);`
    - This would not work because MyFilter would still not be a functional interface.
- Add `implements java.util.function.Predicate<Data> to MyFilter` definition and replace the line at `//1 with:`
  `filterData(al, x -> x.value==0);`
    - This would not work because MyFilter would still not be a functional interface.
- Remove `MyFilter` class altogether.
  Change type of `f from MyFilter to java.util.function.Predicate in filterData` method and replace the line at `//1` with:
  `filterData(al, x -> x.value==0);`
    - Predicate is a generified interface. So you need to type it to Data before you can use this lambda expression. Otherwise, the compiler will assume that the type of x is Object and since value is not a valid field in Object class, x.value will cause a the compilation to fail.
- Remove `MyFilter` class altogether.
  Change type of `f` from `MyFilter` to `java.util.function.Predicate<Data>` in `filterData` method and replace the line at `//1` with:
  `filterData(al, x -> x.value==0);`

// java.util.function.Predicate<Data> se tiene que agregar el tipo de Generics

Problema 4

Which of the following are correct about java.util.function.Predicate?

You had to select 1 option

- It is an interface that has only one abstract method (among other non-abstract methods) with the signature - public void test(T t);
- It is an interface that has only one abstract method (among other non-abstract methods) with the signature - public boolean test(T t);
- It is an abstract class that has only one abstract method (among other non-abstract methods) with the signature - public abstract void test(T t);
- It is an abstract class that has only one abstract method (among other non-abstract methods) with the signature - public abstract boolean test(T t);

Explanation
`java.util.function.Predicate` is one of the several functional interfaces that have been added to Java 8. This interface has exactly one abstract method named `test`, which takes any object as input and returns a boolean. This comes in very handy when you have a collection of objects and you want to go through each object of that collection and see if that object satisfies some criteria. For example, you may have a collection of `Employee` objects and, in one place of your application, you want to remove all such employees whose age is below 50, while in other place, you want to remove all such employees whose salary is above 100,000. In both the cases, you want to go through your collection of employees, and check each Employee object to determine if it fits the criteria. This can be implemented by writing an interface named `CheckEmployee` and having a method `check(Employee )` which would return `true` if the passed object satisfies the criteria. The following code fragments illustrate how it can be done -

```
//define the interface for creating criteria
interface CheckEmployee {
  boolean check(Employee e );
}

...

//write a method that filters Employees based on given criteria.
public void filterEmployees(ArrayList<Employee> dataList, CheckEmployee p){
   Iterator<Employee> i = dataList.iterator();
   while(i.hasNext()){
        if(p.check(i.next())){
             i.remove();
    }
   }
}
```

```
...

//create a specific criteria by defining a class that implements 
CheckEmployee
class MyCheckEmployee implements CheckEmployee{
   public boolean check(Employee e){
       return e.getSalary()>100000;
   }
};
...

//use the filter method with the specific criteria to filter the collection.
filterEmployees(employeeList, new MyCheckEmployee());
```

This is a very common requirement across applications. The purpose of Predicate interface (and other standard functional interfaces) is to eliminate the need for every application to write a customized interface. For example, you can do the same thing with the Predicate interface as follows -

```
public void filterEmployees(ArrayList<Employee> dataList, 
Predicate<Employee> p){
   Iterator<Employee> i = dataList.iterator();
   while(i.hasNext()){
        if(p.test(i.next())){
             i.remove();
    }
   }
}

...

// Instead of defining a MyPredicate class (like we did with 
MyCheckEmployee),
//we could also define and instantiate an anonymous inner class to reduce 
code clutter
Predicate<Employee> p = new Predicate<Employee>(){
  public boolean test(Employee e){
     return e.getSalary()>100000;
  }
};
...

filterEmployees(employeeList, p);
```

Note that both the interfaces (`CheckEmployee` and `Predicate`) can be used with lambda expressions in exactly the same way. Instead of creating an anonymous inner class that implements the CheckEmployee or Predicate interface, you could just do -

```
filterEmployees(employeeList, e->e.getSalary()>100000);
```

Problema 5

What can be inserted in the code below so that it will print true when run?

```
public class TestClass{

   public static boolean checkList(List list, Predicate<List> p){
      return p.test(list);
   }

   public static void main(String[] args) {
      boolean b = //WRITE CODE HERE
      System.out.println(b);
   }
}
```

You had to select 2 options

- `checkList(new ArrayList(), al -> al.isEmpty());`
    - The test method of Predicate returns a boolean. So all you need for your body part in your lambda expression is an expression that returns a boolean. isEmpty() is a valid method of ArrayList, which returns true if there are no elements in the list. Therefore, al.isEmpty() constitutes a valid body for the lambda expression in this case.
- `checkList(new ArrayList(), ArrayList al -> al.isEmpty());`
    - You need to put the parameter list of the lambda expression in brackets if you want to use the parameter type. For example, checkList(new ArrayList(), (List al) -> al.isEmpty()); Remember that specifying the parameter type is optional (as shown in option 1) because the compiler can figure out the parameter types by looking at the signature of the abstract method of any functional interface (here, Predicate's test method).
- `checkList(new ArrayList(), al -> return al.size() == 0);`
    - You need to put the body withing curly braces if you want to use the return keyword. For example, checkList(new ArrayList(), al -> { return al.size() == 0; });
- `checkList(new ArrayList(), al -> al.add("hello"));`
    - The add method of ArrayList returns a boolean. Further, it returns true if the list is altered because of the call to add. In this case, al.add("hello") indeed alters the list because a new element is added to the list.
- `checkList(new ArrayList(), (ArrayList al) -> al.isEmpty());`
    - Predicate is typed to List (not ArrayList) in the checkList method, therefore, the parameter type in the lambda expression must also be List. It cannot be ArrayList.

Problema 6

Given :

```
//In Data.java
public class Data{
    int value;
    Data(int value){
        this.value = value;
    }
    public String toString(){ return ""+value; }
}

and the following code fragments:
public  void filterData(ArrayList<Data> dataList, Predicate<Data> p){
   Iterator<Data> i = dataList.iterator();
   while(i.hasNext()){
        if(p.test(i.next())){
             i.remove();
    }
   }
}
....
        Data d = new Data(1); al.add(d);
        d = new Data(2); al.add(d);
        d = new Data(3); al.add(d);

        //INSERT METHOD CALL HERE
       System.out.println(al);
```

Which of the following options can be inserted above so that it will print [1, 3]?

You had to select 1 option

- `filterData(al, d -> d.value%2 == 0 );`
    - Syntactically, this lambda expression is correct. However, remember that a lambda expression does not create a new scope for variables. Therefore, you cannot reuse the variable names that have already been used to define new variables in your argument list . Here, observe that the variable d is already defined so your argument list cannot use d as a variable name. It would be like defining the same variable twice in the same scope.
- `filterData(al, (Data x) -> x.value%2 == 0 );`
    - When all your method does is return the value of an expression, you can omit the curly braces, the return keyword, and the semi-colon from the method body part. Thus, instead of { return x.value%2 == 0; }, you can just write x.value%2 == 0
- `filterData(al, (Data y) -> y.value%2  );`

    - java.util.function.Predicate interface has one method named test and this method returns a boolean. Therefore, the body of the lambda expression that satisfies this method must return a boolean. Here, y.value%2 is an int and not a boolean.
- `filterData(al, d -> return d.value%2 );`
    - This is invalid because of three reasons -
    - 1. You cannot use d as the name for your parameter as explained in option 1.
    - 2. If you write return statement in your method body, you must enclose it within curly braces and include the semi-colon.
    - 3. To satisfy the Predicate interface, your lambda expression must return a boolean not an int as explained in option 3.

Problema 7

Which of the following statements are correct regarding a functional interface?

You had to select 1 option

- It has exactly one method and it must be abstract.
- It has exactly one method and it may or may not be abstract.
- It must have exactly one abstract method and may have other default or static methods.
- It must have exactly one static method and may have other default or abstract methods.

Explanation

A functional interface is an interface that contains exactly one abstract method. It may contain zero or more default methods and/or static methods. Because a functional interface contains exactly one abstract method, you can omit the name of that method when you implement it using a lambda expression. For example, consider the following interface -

```
interface Predicate<T> {
    boolean test(T t);
}
```

The purpose of this interface is to provide a method that operates on an object of class T and return a boolean.

You could have a method that takes an instance of class that implements this interface defined like this -

```
public void printImportantData(ArrayList<Data> dataList, Predicate<Data> p){
   for(Data d: dataList){
      if(p.test(d)) System.out.println(d);
   }
}
```

where Data class could be as simple as `public class Data{ public int value; }`

Problema 8

Given:
```
interface Runner {
  public void run();
}
```

Which of the following is/are valid lambda expression(s) that capture(s) the above interface?

You had to select 2 options

- `-> System.out.println("running...");`
- `void -> System.out.println("running...")`
- `() -> System.out.println("running...")`
    - A lambda body must be either a single expression or a block.
    - `System.out.println("running...")` is not an expression because it doesn't return anything. So, it doesn't really satisfy the above requirement. However, it is an exception to the rule. A semicolon is not required when you use a method call returning void as the body of the lamda expression even though the method call is not an expression.
- `() -> { System.out.println("running..."); return; }`
- `(void) -> System.out.println("running...")`
- `-> System.out.println("running...")`

Explanation

`Runner` is a valid functional interface because it has exactly one abstract method.
Since this method does not take any parameter, the parameter list part of the lambda expression must be `()`. Further, since it does not return anything, the body part should ideally be such that it does not return anything either. Thus, you can either use a method call that returns void or some code enclosed within { and } that does not return anything. In this case, however, since there is only one interface with one method, it is ok even if the body of the lambda expression returns a value as illustrated by the following code:

Problema 9

What will the following code print when compiled and run?

```
public class Data{
    int value;
    Data(int value){
        this.value = value;
    }
    public String toString(){ return ""+value; }

    public static void main(String[] args) {
        Data[] dataArr = new Data[]{ new Data(1), new Data(2),
              new Data(3), new Data(4) };

        List<Data> dataList = Arrays.asList(dataArr); //1

        for(Data element :  dataList){

            dataList.removeIf( (Data d) -> { return d.value%2 ==  0; } );  //2

            System.out.println("Removed "+d+", "); //3
        }
   }
}
```

You had to select 1 option

- Removed 2, Removed 4
- It will print Removed 2 and then an exception stack trace.
- It will print an exception stack trace.
- It will not compile due to //1
- It will not compile due to //2
- It will not compile due to //3
    - Line at //3 will cause compilation failure because the lambda variable d is available only within the lambda body. It is not available at //3.

Explanation:

1. Observe that the loop is completely unnecessary. The `Collection's removeIf` method takes a `Predicate` and removes all elements of the `List` for which the `Predicate` returns `true`. You need not put it in a loop to check for each element.

2. If you remove //3, the code will compile but will throw a `java.lang.UnsupportedOperationException` at run time. Since the list is backed by an array and the size of the array cannot change, you cannot add or remove elements from a List backed by an array. You may, however, change the elements of the List using `list.set(index, element);` method.

Problema 10

What can be inserted in the code below so that it will print true when run?

```
public class TestClass{

   public static boolean checkList(List list, Predicate<List> p){
      return p.test(list);
   }

   public static void main(String[] args) {
      boolean b = //WRITE CODE HERE
      System.out.println(b);
   }
}
```

You had to select 2 options

`checkList(new ArrayList(), al -> al.isEmpty());`

The test method of Predicate returns a boolean. So all you need for your body part in your lambda expression is an expression that returns a boolean. isEmpty() is a valid method of ArrayList, which returns true if there are no elements in the list. Therefore, al.isEmpty() constitutes a valid body for the lambda expression in this case.

`checkList(new ArrayList(), ArrayList al -> al.isEmpty());`

You need to put the parameter list of the lambda expression in brackets if you want to use the parameter type. For example, checkList(new ArrayList(), (List al) -> al.isEmpty()); Remember that specifying the parameter type is optional (as shown in option 1) because the compiler can figure out the parameter types by looking at the signature of the abstract method of any functional interface (here, Predicate's test method).

`checkList(new ArrayList(), al -> return al.size() == 0);`

You need to put the body withing curly braces if you want to use the return keyword. For example, checkList(new ArrayList(), al -> { return al.size() == 0; });

`checkList(new ArrayList(), al -> al.add("hello"));`

The add method of ArrayList returns a boolean. Further, it returns true if the list is altered because of the call to add. In this case, al.add("hello") indeed alters the list because a new element is added to the list.

`checkList(new ArrayList(), (ArrayList al) -> al.isEmpty());`

Predicate is typed to List (not ArrayList) in the checkList method, therefore, the parameter type in the lambda expression must also be List. It cannot be ArrayList.

Problema 11 - OCP

Identify the correct statements about the following code:

```
import java.util.*;
import java.util.function.*
class Account {
    private String id;
    public Account(String id){ this.id = id; }
    //accessors not shown
}
public class BankAccount extends Account{
    private double balance;
    public BankAccount(String id, double balance){ super(id); this.balance =
balance;}

    //accessors not shown

    public static void main(String[] args) {
        Map<String, Account> myAccts = new HashMap<>();
        myAccts.put("111", new Account("111"));
        myAccts.put("222", new BankAccount("111", 200.0));

        BiFunction<String, Account, Account> bif =
  (a1, a2)-> a2 instanceof BankAccount?new BankAccount(a1, 300.0):new
Account(a1); //1

        myAccts.computeIfPresent("222", bif);//2
        BankAccount ba = (BankAccount) myAccts.get("222");
        System.out.println(ba.getBalance());
    }
}
```

You had to select 1 option

- It will not compile due to code at //1.
- It will not compile due to code at //2.
- It will print 200.0
- It will print 300.0
    - Since myAccts map does contain a key "222", computeIfPresent method will execute the function and replace the existing value associated with the given key in the map with the new value returned by the function. The given function returns a new BankAccount object with a balance of 300.0.

Problema 12 - OCP

Given:

```
HashMap<Integer, String> hm = new HashMap<>();
hm.put(1, "a"); hm.put(2, "b"); hm.put(3, "c");
```

Which of the following statements will print the keys as well as values contained in the map?

You had to select 1 option

- `hm.forEach((key, entry)->System.out.println(entry));`
  - This will compile and run fine but will only print the values and not the keys.
- `hm.forEach(System.out.println("%d %s"));`
  - This will not compile because Map's forEach method requires BiConsumer object. A BiConsumer takes exactly two parameters but println method takes only one and therefore cannot be used to implement BiConsumer.
- `hm.forEach((key, value)->System.out.printf("%d %s ", key, value));`
- `hm.forEach(System.out::println);`
  - This will not compile because Map's forEach method requires BiConsumer object. A BiConsumer takes exactly two parameters but println method takes only one and therefore cannot be used to implement BiConsumer.
- `hm.forEach(entry->System.out.println(entry));`
  - This will not compile for the same reason as above. This lambda expression does not capture BiConsumer interface.

Explanation:
Prácticamente es como en Python teniendo unos placeholders y después declarando en orden a cual se refieren. %d double que es el ‘key’ y %s que es de tipo String y apunta al ‘value’

Problema 13 - OCP

What will the following code print when compiled and run?

```
List<StringBuilder> messages = Arrays.asList(new StringBuilder(), new
StringBuilder());
messages.stream().forEach(s->s.append("helloworld"));
messages.forEach(s->{
    s.insert(5,",");
    System.out.println(s);
});
```

You had to select 1 option

- It will not print anything.
- It will not compile.
- It will throw IllegalStateException at runtime.
- `hello,world`
  `hello,world`
    - StringBuilder is mutable. It has several overloaded insert methods (one for each data type such as boolean, char, int, and String) that insert the given object into a given position.

    - public StringBuilder insert(int offset, String str)
    - Inserts the string into this character sequence.
    - The characters of the String argument are inserted, in order, into this sequence at the indicated offset, moving up any characters originally above that position and increasing the length of this sequence by the length of the argument. If str is null, then the four characters "null" are inserted into this sequence.

    - The character at index k in the new character sequence is equal to:
    - the character at index k in the old character sequence, if k is less than offset
    - the character at index k-offset in the argument str, if k is not less than offset but is less than offset+str.length()
    - the character at index k-str.length() in the old character sequence, if k is not less than offset+str.length()
    - The offset argument must be greater than or equal to 0, and less than or equal to the length of this sequence.
    - Parameters: offset -
      the offset.
      str - a string.
    - Returns:
      a reference to this object.
- `helloworld`
- `helloworld`

Problema 14 - OCP

Given:

```
List<Integer> ls = Arrays.asList(3,4,6,9,2,5,7);
System.out.println(ls.stream().reduce(Integer.MIN_VALUE, (a, b)->a>b?a:b)); //1
System.out.println(ls.stream().max(Integer::max).get()); //2
System.out.println(ls.stream().max(Integer::compare).get()); //3
System.out.println(ls.stream().max((a, b)->a>b?a:b)); //4
```

Which of the above statements will print 9?

You had to select 1 option

- 1 and 4
- 2 and 3
- [✓ 1 and 3]
- 2, 3, and 4
- All of them.
- None of them.

Explanation
The code will print:

```
9
3
9
Optional[3]
```

You need to understand the following points to answer this question:

1. The reduce method needs a `BinaryOperator`. This interface is meant to consume two arguments and produce one output. It is applied repeatedly on the elements in the stream until only one element is left. The first argument is used to provide an initial value to start the process. (If you don't pass this argument, a different reduce method will be invoked and that returns an Optional object.)

2. The `Stream.max` method requires a `Comparator`. All you need to implement this interface using a lambda expression is a reference to any method that takes two arguments and returns an int. The name of the method doesn't matter. That is why it is possible to pass the reference of `Integer's max` method as an argument to `Stream's max` method. However, `Integer.max` works very differently from `Integer.compare`. The `max` method returns the maximum of two numbers while the `compare` method returns a difference between two numbers. Therefore, when you pass `Integer::max` to `Stream's max`, you will not get the correct maximum element from the stream. That is why //2 will compile but will not work correctly.

//4 is basically same as //2. It will not work correctly for the same reason.

Problema 15 - OCP

What will the following code print when compiled and run?

```
BiPredicate<String, Integer> bip = (s, i)-> s.length()>i;  //1
BiFunction<String, Integer, String> bif = (s, i)-> {  //2
   if(bip.test(s, i)){ //3
      return s.substring(0, i);
   }
   else return s;
};
String str = bif.apply("hello world", 5); //4
System.out.println(str);
```

You had to select 1 option

- hello
    - There is no problem with the code.
- compilation failure at //1
- compilation failure at //2
- compilation failure at //3

Explanation

The given code illustrates the usage of BiPredicate and BiFunction interfaces. Remember that a predicate is used to test some condition (that means it must return a boolean), while a function is meant to consume one thing (or two, in case of Bi) and return another.

Problema 16 - OCP

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")
);

Reader r = b->{
    System.out.println("Reading book "+b.getTitle());
};
books.forEach(x->r.read(x));
```

What would be a valid definition of Reader for the above code to compile and run without any error or exception?
You had to select 2 options

-
  ```
  abstract class Reader{
     abstract void read(Book b);
  }
  ```

-
  ```
  abstract class Reader{
      void read(Book b);
  }
  ```
  - This is invalid code because read method doesn't have a body and is not declared abstract either.

-
  ```
  interface Reader{
      void read(Book b);
      default void unread(Book b){      }
  }
  ```
  - Since the given code uses lambda expression, Reader must be a functional interface (i.e. an interface with exactly one abstract method). The interface can have other default and/or static method.

-
  ```
  interface Reader{
      default void read(Book b){ }
      void unread(Book b);
  }
  ```
  - This is a valid functional interface and so the code will work fine. However, observe that the lambda expression in the code will capture the unread method (not the read method, because read method is not abstract). Therefore, r.read() will cause the read method defined in this interface to be invoked instead of the code implemented by the lambda expression.

-
  ```
  interface Reader{
      default void read(Book b){ System.out.println("Default read");};
  }
  ```
  - This interface code itself is valid but it is not a valid functional interface because it has no abstract method. Therefore, the code for lambda expression will not compile.

Problema 17 - OCP

What can be inserted in the following code so that it will print 12?

```
public class TestClass{
    interface Doer{
        String doIt(int x, String y);
   }
   public static void main(String[] args) {

        INSERT CODE HERE

        System.out.println(d.doIt(2, "12345"));
    }
}
```

You had to select 2 options

- `int a; String b;`
- `Doer d = (a,  b)->b.substring(0, a);`
    - This will not compile because a and b are already defined. Remember that the parameter specification of a lambda expression does not create a new scope. The parameter names in the lambda expression exist in the same scope as the method or the block in which the lambda expression is created.
- `int a = 0; String b = "";`
- `Doer d = (a,  b)->b.substring(0, a);`
    - Same as above.
- `Doer d = (a, b)->b.substring(0, a);`
- `Doer d = (int a, String b)->b.substring(0, a);`
- `Doer d = a, b->b.substring(0, a);`
    - You must enclose the parameters within parenthesis when the lambda expression has more than one parameter.

Explanation

Interface Doer has exactly one abstract method and therefore it is a valid functional interface. Both - `(a, b)->b.substring(0, a);` and `(int a, String  b)->b.substring(0, a);` are valid lambda expressions that capture this interface.

Think of the expression as if you are writing a method for a class that implements this interface:

```
class SomeDoer implements Doer{
  public  String doIt(int a, String b){
     return b.substring(0, a);
  }
}
```

Problema 18 - OCP

Given:

```
List<String> strList = Arrays.asList("a", "aa", "aaa");
Function<String, Integer> f = x->x.length();
Consumer<Integer> c = x->System.out.print("Len:"+x+" ");
strList.stream().map(f).forEach(c);
```

What will it print when compiled and run?

You had to select 1 option

- A compilation error will occur.
  - There is no problem with the code.
- Len:a Len:aa Len:aaa
- Len:1 Len:2 Len:3
  - The function f accepts a String and returns its length. The call to map(f), uses this function to replace each element of the stream with an Integer. The call to forEach(c) uses function c to print each element.
- It will compile and run fine but will not print anything.

Problema 19 - OCP

Given:

```
List<Double> dList = Arrays.asList(10.0, 12.0);
dList.stream().forEach(x->{ x = x+10; });
dList.stream().forEach(d->System.out.println(d));
```

What will it print when compiled and run?

You had to select 1 option

- A compilation error will occur.
  - There is no problem with the code.
- `10.0`
  `12.0`
  - Remember that the variables are passed by value. Therefore, when a new Double object is assigned to x by the statement x = x + 10; , the original element in the list is not changed. Therefore, the first call to forEach does not change the elements in the original list on which the stream is based. Thus, when you call dlist.stream() the second time, a new stream is created and it has the same elements as the list i.e. 10.0 and 12.0.
  - If it were a Stream of mutable objects such as StringBuilders and if you append something to the elements in the first forEach, that would change the original StringBuilder objects contained in the list. In that case, the second forEach will actually print the updated values. For example, the following code will indeed print `ab` and `aab`.

  -
    ```
    List<StringBuilder> dList = Arrays.asList(new StringBuilder("a"), new
                                 StringBuilder("aa"));
    dList.stream().forEach(x->x.append("b"));
    dList.stream().forEach(x->System.out.println(x));
    ```
- `20.0`
  `22.0`
- It will compile but throw an exception at run time.

Problema 20 - OCP

Given:

```
List<Double> dList = Arrays.asList(10.0, 12.0);
DoubleFunction df = x->x+10;
dList.stream().forEach(df);
dList.stream().forEach(d->System.out.println(d));
```

What will it print when compiled and run?

You had to select 1 option

- A compilation error will occur.
  - forEach method expects a Consumer as an argument. Not a Function.

- `10.0`
- `12.0`

- `20.0`
- `22.0`

- It will compile but throw an exception at run time.

Problema 21 - OCP

Which of the following interface definitions can be implemented using Lambda expressions?

```
interface A{
   static void m1(){};
}

interface AA extends A{
   void m2();
}

interface AAA extends AA{
   void m3();
}

interface B {
   default void m1(){   }
}
interface BB extends B {
  static void m2(){ };
}
interface C extends BB{
    void m3();
}
```

You had to select 2 options

- A
  - It has no abstract method.
- AA
  - It has one abstract method.
- AAA
  - It has two abstract methods.
- B
  - It has no abstract method.
- BB
  - It has no abstract method.
- C
  - It has one abstract method.

To take advantage of lambda expressions, an interface must be a "functional" interface, which basically means that the interface must have exactly one abstract method. A lambda expression essentially provides the implementation for that abstract method.

It does not matter whether the abstract method is declared in this interface or a super interface. There is no constraint on the parameter types or on the return type. The interface may have other default or static methods as well but those are not relevant. All that is required is that it must have exactly one abstract method.

Problema 22 - OCP

What will the following code print when compiled and run?

```
public class TestClass{

    public static int operate(IntUnaryOperator iuo){
        return iuo.applyAsInt(5);
    }

    public static void main(String[] args) {

        IntFunction<IntUnaryOperator> fo = a->b->a-b;  //1   es mandar b->20-b
               //funcion para mandar un IntUnaryOperator en el método
        int x = operate(fo.apply(20)); //2
                                //primero resuelve este IntFunction
        System.out.println(x);
    }
}
```

You had to select 1 option

- Compilation error at //1
- Compilation error at //2
- [✓ 15]
- -15
- 20
- Exception at run time

Explanation

The lambda expression `a->b->a-b` looks complicated but it is actually simple if you group it like this:
`a->(b->a-b);`

1. `IntFunction` is a functional interface that takes an int and returns whatever it is typed to. Here, the `IntFunction` is typed to `IntUnaryOperator`. Therefore, `IntFunction<IntUnaryOperator>` will take in an `int` and return `IntUnaryOperator`.
The general form of a lambda expression that captures `IntFunction` would be `x->line of code that returns the correct return type i.e.` IntUnaryOperator`, in this case;` where `x` is an int.

Now, if you look at `fo = a->b->a-b;`, `a` is the argument variable of type int (because it is being used to capture IntFunction) and `b->a-b` forms the method body of the `IntFunction`. `b->a-b` must somehow return an `IntUnaryOperator` for `fo = a->b->a-b;` to work.

Problema 23 - OCP

Identify the correct statements about the following code:-

```
List<Integer> values = Arrays.asList(2, 4, 6, 9); //1
Predicate<Integer> check = (Integer i) -> {
    System.out.println("Checking");
    return i == 4; //2
};
Predicate even = (Integer i)-> i%2==0;  //3
values.stream().filter(check).filter(even).count(); //4
```

You had to select 1 option

- It will not compile because of code at //1.
- It will not compile because of code at //2.
- It will not compile because of code at //3.
    - Observe the lambda expression used to instantiate the Predicate is using Integer as the type of the variable. To make this work, the declaration part must be typed to Integer. Like this:
    - `Predicate<Integer> even = (Integer i)-> i%2==0;  //3`
      Another option is to used Object type like this:
      `Predicate even = (Object i)-> ((Integer)i)%2==0;`
      or
      `Predicate even = i -> ((Integer)i)%2==0;`
- It will not compile because of code at //4.
- It will compile fine and print Checking four times.
- It will compile fine and print Checking three times.
- It will compile fine and print Checking one time.
- It will compile fine but will print nothing.

Problema 24 - OCP

An existing application contains the following code:

```
interface AmountValidator{
    public boolean checkAmount(double value);
}
public class Account {
    public void updateBalance(double bal ){

        boolean isOK = new AmountValidator(){
            public boolean checkAmount(double val){
                return val >= 0.0;
            }
        }.checkAmount(bal);

        //other irrelevant code
    }
}
```

Which interface from java.util.function package should be used to refactor this code?

You had to select 1 option

- Consumer
  - Represents an operation that accepts a single input argument and returns no result.

- Function
  - Represents a function that accepts one argument and produces a result.

- Predicate
  - Represents a predicate (boolean-valued function) of one argument. This interface is used when you want to check for some condition. For example, the given code can be easily refactored as follows -

    ```
    public void updateBalance(double bal ){
            Predicate<Double> p = val -> val>=0.0;
                        //use lamba expression to create a Predicate
            boolean isOK = p.test(bal);
            //other irrelevant code
     }
    ```
  - There is no need to create the AmountValidator interface and the anonymous class.

- Supplier
  - Represents a supplier of results.

Problema 25 - OCP

What will the following code print?

```
List<Integer> str = Arrays.asList(1,2, 3, 4 );
str.stream().filter(x->{
    System.out.print(x+" ");
    return x>2;
});
```

You had to select 1 option

- 1 2 3 4
- 1 2 3 4 4
- 4
- It will not print anything.

Remember that filter is an intermediate operation. It will not be executed until you invoke a terminal operation such as count or forEach on the stream.

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.

Problema 26 - OCP

What can be inserted into the following code at //1 so that it will print "Oldboy"?

```
//imports not shown
class Movie{
    static enum Genre  {DRAMA, THRILLER, HORROR, ACTION };
    private Genre genre;
    private String name;
    Movie(String name, Genre genre){
        this.name = name; this.genre = genre;
    }
    //accessors not shown
}
public class FilteringStuff {
    public static void main(String[] args) {
        List<Movie> movies = Arrays.asList(
                new Movie("On the Waterfront", Movie.Genre.DRAMA),
                new Movie("Psycho", Movie.Genre.THRILLER),
                new Movie("Oldboy", Movie.Genre.THRILLER),
                new Movie("Shining", Movie.Genre.HORROR)
                );
        Predicate<Movie> horror = mov->mov.getGenre() == Movie.Genre.THRILLER;
        Predicate<Movie> name = mov->mov.getName().startsWith("O");
        //1 INSERT CODE HERE
        .forEach(mov->System.out.println(mov.getName()));
    }

}
```

You had to select 1 option

- movies.stream().filter(horror, name)
  - filter method takes only one argument of type Predicate.
- movies.filter({horror, name})
- movies.stream().filter({horror, name})
- movies.stream().filter(horror).filter(name)
  - This is the correct way to chain multiple filters to a Stream.
- movies.filter(horror).filter(name)

Explanation

The given code illustrates how you can use functional interfaces, lambda expressions, and streams to filter a List of objects based on multiple criteria.
First, you need to convert the List into a `Stream`. List has `stream()` method that returns a `Stream`. Stream support `filter(Predicate )` method, which filters the elements in the stream. Only those elements for which Predicate's `test()` method returns `true`, are left remaining in the stream. The return type of filter method is `Stream`, so you can chain multiple filter methods one after another.

Problema 27 - OCP Effective Final

What changes, when applied independent of each other, will enable the following code to compile?

```
//assume appropriate import statements
class TestClass{
   public double process(double payment, int rate)
   {
      double defaultrate = 0.10;        //1
                               //Requiere no cambiar de valor para ser effective final
                               // No tiene nada que ver la 'clase local o anidada'
                               // Si son atributos locales se requiere
                               // Si fuera atributo de clase no hay problema
      if(rate>10) defaultrate = rate;  //2
      class Implement{
         public int apply(double data){
             Function<Integer, Integer> f = x->x+(int)(x*defaultrate);  //3
             return f.apply((int)data); //4
         }
      }
      Implement i = new Implement();
      return i.apply(payment);
   }
}
```

You had to select 2 options

- Change //1 to final double defaultrate = 0.10;
  - If you make it final, //2 will not compile.

- Remove code at //2.
  - A local variable needs to be final or effectively final to be accessed from an inner class or lambda expression. If you remove //2, that means defaultrate never changes and is therefore effectively final.

- `Replace lines at //3 and //4 with:`
  ```
  BiFunction<Integer, Double, Integer> f = (m, n)->m+(int)(n*m);
  return f.apply((int)data, defaultrate);
  ```
  - This will still not solve the problem with using non-final variable from an inner class. f.apply((int)data, defaultrate); will still not compile because default rate is not final (or effectively final).

- Change //3 to:
  `Function<Integer, Integer> f = x->x+(int)(x*rate);`

Problema 28 - OCP

Which of the following statements is/are true about java.util.function.IntFunction?

- It represents a function that returns an int primitive.
    - It takes int primitive as an argument. It can be parameterized to return any thing. For example, IntFunction<String> f = x->""+x; returns a String.

- It avoids additional cost associated with auto-boxing/unboxing.
    - Remember that primitive and object versions of data types (i.e. int and Integer, double and Double, etc.) are not really compatible with each other in java. They are made compatible through the extra step of auto-boxing/unboxing. Thus, if you have a stream of primitive ints and if you try to use the object versions of Stream and Function (i.e. Stream<Integer> and Function<Integer, Integer>, you will incur the cost of boxing and unboxing the elements. To eliminate this problem, the function package contains primitive specialized versions of streams as well as functional interfaces. For example, instead of using Stream<Integer>, you should use IntStream. You can now process each element of the stream using IntFunction. This will avoid auto-boxing/unboxing altogether. Thus, whenever you want to process streams of primitive elements, you should use the primitive specialized streams (i.e. IntStream, LongStream, and DoubleStream) and primitive specialized functional interfaces (i.e. IntFunction, IntConsumer, IntSupplier etc.) to achieve better performance.

- It extends java.util.function.Function
    - None of the primitive specialized functional interfaces (such as IntFunction, DoubleFunction, or IntConsumer) extend the non-primitive functional interfaces (i.e. Function, Consumer, and so on).

- It cannot be parameterized as it is meant to deal only with int primitives.
    - The parameter type cannot be parameterized but return type can be as shown in explanation to option 1.

Explanation
java.util.function package contains int, double, and long (but no float) versions of all the functional interfaces. For example, there is an IntFunction, a DoubleFunction, and a LongFunction, which are int, double, and long, versions of Function.

Problema 29 - OCP

Given that Book is a valid class with appropriate constructor and getPrice and getTitle methods that returns a double and a String respectively, consider the following code:

```
List<List<Book>> books = Arrays.asList(
        Arrays.asList(
                new Book("Windmills of the Gods", 7.0),
                new Book("Tell me your dreams",9.0) ),
        Arrays.asList(
                new Book("There is a hippy on the highway", 5.0),
                new Book("Easy come easy go", 5.0)) );

double d = books.stream()
        INSERT CODE HERE //1
        INSERT CODE HERE //2
        .sum();
System.out.println(d);
```

What can be inserted in the above code so that it will print 26.0?

You had to select 1 option

- `.flatMap(bs)`
  and
  `.mapToDouble(book.getPrice())`

- `.flatMap(bs.stream())`
  and
  `.mapToDouble(book.getPrice())`

- `.flatMap(bs->bs.stream())`
  and
  `.map(book->book.getPrice()).toDouble()`

- `.flatMap(bs.stream())`
  and
  `.map(book->book.getPrice()).toDouble()`

- `.flatMap(bs->bs.stream())`
  and
  `.mapToDouble(book->book.getPrice())`

Explanation

The `flatMap` method expects a `Function` that will take an element and create a `Stream` out of it. It then joins each of those streams (one stream for each element in the original stream) to return a big combined stream of elements. `bs->bs.stream()` correctly captures such a `Function`.
The `mapToDouble` method expects a `ToDoubleFunction` object that will take an argument and return a `double`. It then returns a `DoubleStream` containing `double` primitives.

`DoubleStream` has method sum that simply returns the sum of all the elements.
None of the other options will compile.

Problema 30 - OCP

What will the following code print when compiled and run?

```
BinaryOperator<String> bo = (s1, s2) -> s1.concat(s2);
List<String> names  = new ArrayList<>();
names.add("Bill"); names.add("George"); names.add("Obama");
String finalvalue = names.stream().reduce("Hello : ", bo);
System.out.println(finalvalue);
```

You had to select 1 option

- Hello : BillGeorgeObama
- Hello : BillHello : GeorgeHello : Obama
- BillGeorgeObamaHello :
- BillGeorgeObama
- It will throw an exception at run time.

Explanation

This question tests you on Lambda expressions, Functional interfaces, and Stream API.

1. The lambda expression `(s1, s2) -> s1.concat(s2);` is quite straight forward. It implements the functional interface BinaryOperator, which has one abstract method `apply(T, T)`. Here, T is typed to String and the method body simply returns the concatenated String.

2. The `reduce` method of a Stream is mean to reduce a stream into just one value. It combines two elements of a stream at a time using a given `BinaryOperator`, and replaces those two elements in the stream with the return value of the apply method of BinaryOperator. It keeps on doing this reduction until there is just one value left.

There are three versions of reduce :

`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.

The second version is used in this question. This version ensures that there is always a resulting value unlike the first version, which may return an empty Optional.

Problema 31 - OCP

Which of the following interface definitions can use Lambda expressions?

You had to select 1 option

-
  ```
  interface A{
  }
  ```

-
  ```
  interface A{
      default void m(){};
  }
  ```

-
  ```
  interface A{
      void m(){};
  }
  ```
    - This is an invalid interface definition. m() must either be declared as default or must not have a body.

-
  ```
  interface A{
      default void m1(){};
      void m2();
  }
  ```

-
  ```
  interface A{
      void m1();
      void m2();
  }
  ```

Explanation

To take advantage of lambda expressions, an interface must be a "functional" interface, which basically means that the interface must have exactly one abstract method. A lambda expression essentially provides the implementation for that abstract method.

It does not matter whether the abstract method is declared in this interface or a super interface. There is no constraint on the parameter types or on the return type. The interface may have other default or static methods as well but those are not relevant. All that is required is that it must have exactly one abstract method.

Anexo

compute()

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

Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). For example, to either create or append a String msg to a value mapping:

```
map.compute(key, (k, v) -> (v == null) ? msg : v.concat(msg))
```

computeIfPresent()

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

If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
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 specified value is to be associated remappingFunction - the function to compute a value
Returns:
the new value associated with the specified key, or null if none.

00 Java Basics

01 Java Working with Data Types

02 Java Using Operators & Decision Constructs