Correction Lambda Built-in Functional Interfaces - 0

A - Only two options

Blank space of check(_____________ supplier) method needs to be filled such that check(...) method compiles, two statements invoking check(...) method compile and output is:

```
Document-Author
RFP-Author
```

'Document::new' is same as '() -> {return new Document();}' and 'RFP::new' is same as '() -> {return new RFP();}' and as these are the implementations of get method of Supplier interface, hence let's check all the options one by one:

1. Supplier:- Method parameter will be initialized as:

```
Supplier<Document> supplier = () -> {return new Document();};
```

or

```
Supplier<Document> supplier = () -> {return new RFP();};
```

No issues as get() method either returns Document or RFP instance.

2. Supplier<? extends Document>:- Method parameter will be initialize as:

```
Supplier<? extends Document> supplier = () -> {return new
        Document();};
```

or

```
Supplier<? extends Document> supplier = () -> {return new RFP();};
```

If target type is a wildcard-parameterized functional interface, then type inference of lambda expression is based on following from the JLS:

If T is a wildcard-parameterized functional interface type and the lambda expression is implicitly typed, then the ground target type is the non-wildcard parameterization of T.

Based on above JLS statement, ground target type of above lambda expression will be: Supplier and this means above expression is equivalent to:

```
Supplier<? extends Document> supplier = (Supplier<Document>)() ->
        {return new Document();};
```

or

```
Supplier<? extends Document> supplier = (Supplier<Document>)() ->
        {return new RFP();};
```

No issues as get() method either returns Document or RFP instance.

3. Supplier<? super Document>:- Method parameter will be initialize as:

```
Supplier<? super Document> supplier = () -> {return new
        Document();};
```

or

```
Supplier<? super Document> supplier = () -> {return new RFP();};
```

Based on above JLS statement, ground target type of above lambda expression will be: Supplier and this means above expression is equivalent to:

```
Supplier<? super Document> supplier = (Supplier<Document>)() ->
        {return new Document();};
```

or

```
Supplier<? super Document> supplier = (Supplier<Document>)() ->
        {return new RFP();};
```

There is no issue with lambda expressions (method invocation) but get() method would return an Object type and invoking printAuthor() method on Object type is not possible. Hence compilation error.

4. Supplier:- Method parameter will be initialized as:

```
Supplier<RFP> supplier = () -> {return new Document();};
```

or

```
Supplier<RFP> supplier = () -> {return new RFP();};
```

1st statement causes compilation failure.

5. Supplier<? extends RFP>:- Method parameter will be initialized as:

```
Supplier<? extends RFP> supplier = () -> {return new Document();};
```

or

```
Supplier<? extends RFP> supplier = () -> {return new RFP();};
```

Based on above JLS statement, ground target type of above lambda expression will be: Supplier and this means above expression is equivalent to:

```
Supplier<? extends RFP> supplier = (Supplier<RFP>)() -> {return
        new Document();};
```

or

```
Supplier<? extends RFP> supplier = (Supplier<RFP>)() -> {return
        new RFP();};
```

2nd statement compiles successfully but 1st statement fails to compile.

6. Supplier<? super RFP>:- Method parameter will be initialized as:

```
Supplier<? super RFP> supplier = () -> {return new Document();};
```

or

```
Supplier<? super RFP> supplier = () -> {return new RFP();};
```

Based on above JLS statement, ground target type of above lambda expression will be: Supplier and this means above expression is equivalent to:

```
Supplier<? super RFP> supplier = (Supplier<RFP>)() -> {return new
        Document();};
```

or

```
Supplier<? super RFP> supplier = (Supplier<RFP>)() -> {return new
        RFP();};
```

1st statement fails to compile. There is no issue with the lambda expression of 2nd statement (method invocation) but get() method would return an Object type and invoking printAuthor() method on Object type is not possible. Hence compilation error.

7. Supplier:- Doesn't compile as raw type's get() method returns Object type, so printAuthor() method can't be invoked.

Hence out of given 7 options, only two options 'Supplier' and 'Supplier<? extends Document>' will give the desired output.

Correction Lambda Built-in Functional Interfaces - 1

B - 55

andThen is the default method defined in Consumer interface, so it is invoked on consumer reference variable.

Value passed in the argument of accept method is passed to both the consumer objects. So, for understanding purpose Line 7 can be split into: consumer.accept(5); consumer.accept(5); So it prints ‘55’ on to the console.

Check the code of andThen method defined in Consumer interface to understand it better.

Correction Lambda Built-in Functional Interfaces - 2

D - Compilation error

Target type of lambda expression should be a functional interface.

StringConsumer is not a Functional Interface as it just specifies one default method, abstract method is not available.

Statement ‘StringConsumer consumer = s -> System.out.println(s.toLowerCase());’ causes compilation error.

Correction Lambda Built-in Functional Interfaces - 3

E - {1=ONE, 2=two}

Though reference variable of NavigableMap is used but putIfAbsent method is from Map interface. It is a default method added in JDK 8.0.

BiConsumer<T, U> : void accept(T t, U u);

Lambda expression corresponding to ‘map::putIfAbsent;’ is ‘(i, s) -> map.putIfAbsent(i, s)’

This is the case of “Reference to an Instance Method of a Particular Object”.

TreeMap sorts the data on the basis of natural order of keys.

consumer.accept(1, null); => {1=null}.

consumer.accept(2, “two”); => {1=null, 2=two}.

consumer.accept(1, “ONE”); => {1=ONE, 2=two}. putIfAbsent method replaces null value with the new value.

consumer.accept(2, “TWO”); => {1=ONE, 2=two}. As value is available against ‘2’, hence value is not replaced.

Correction Lambda Built-in Functional Interfaces - 4

B -

```
ab
bab
bb
baba
aba
```

startsWith is an instance method in String class. ‘String::startsWith’ corresponds to ‘(str1, str2) -> str1.startsWith(str2);’

Given predicate means return true for any text starting with uppercase ‘A’.

But inside for loop predicate.negate() means, return true for any text not starting with uppercase ‘A’.

Correction Lambda Built-in Functional Interfaces - 5

A - true

BiPredicate<T, U> : boolean test(T t, U u);

BiPredicate interface accepts 2 type parameters and these parameters (T,U) are passed to test method, which returns primitive boolean.

In this case, ‘BiPredicate<String, String> predicate’ means test method will have declaration: ‘boolean test(String s1, String d2)’.

‘String::equalsIgnoreCase’ is equivalent to ‘(s1, s2) -> s1.equalsIgnoreCase(s2)’

This is an example of “Reference to an Instance Method of an Arbitrary Object of a Particular Type” and not “method reference to static method”.

“JaVa”.equalsIgnoreCase(“Java”) => returns true.

Correction Lambda Built-in Functional Interfaces - 6

C - 1

‘text::indexOf’ is equivalent to lambda expression ‘search -> text.indexOf(search)’.

ToIntFunction has method: int applyAsInt(T value);.

In this case T is of String type.

func.applyAsInt(“a”) will invoke text.indexOf(“a”), which returns the index of first occurrence of “a”. In the given text, it is 1.

Correction Lambda Built-in Functional Interfaces - 7

A - map.forEach(consumer);

In JDK 1.8, Map interface had added the default method: “default void forEach(BiConsumer)”.

This method performs the BiConsumer action (passed as an argument) for each entry in the Map. From the given options, ‘map.forEach(consumer);’ is the only valid option.

There is no method with the name ‘forEachOrdered’ in Map interface.

Correction Lambda Built-in Functional Interfaces - 8

D-

```
dr
mr
miss
mrs
```

list is of List type, so list.forEach(…) method can accept argument of Consumer type.

interface StringConsumer extends Consumer, which means instances of StringConsumer will also be instances of Consumer.

Note: StringConsumer is not a Functional Interface as it just specifies one default method.

Reference variable ‘consumer’ refers to an instance of anonymous subclass of StringConsumer and overrides accept(String) method.

list.forEach(consumer); => Prints list elements in lower case as overriding version of anonymous inner class is used.

Correction Lambda Built-in Functional Interfaces - 9

C - Compilation error

BiFunction<T, U, R> : R apply(T t, U u);

BiFunction interface accepts 3 type parameters, first 2 parameters (T,U) are passed to apply method and 3rd type parameter is the return type of apply method.

In this case, ‘BiFunction<Integer, Integer, Character>’ means apply method will have declaration: ‘Character apply(Integer d1, Integer d2)’.

Lambda expression should accept 2 Integer type parameters and must return Character object.

Lambda expression is:

‘(i, j) -> i + j’, i + j returns an int type and int cannot be implicitly casted to Character and this causes compilation error.

Correction Lambda Built-in Functional Interfaces - 10

D - [North, East, West, South]

replaceAll(UnaryOperator operator) is the default method added in List interface.

interface UnaryOperator extends Function<T, T>.

As List is of String type, this means operator must be of UnaryOperator type only. Its accept method should have signature: String apply(String s);

Lambda expression ‘s -> s.substring(0,1).toUpperCase().concat(s.substring(1))’ is correctly defined for apply method.

The lambda expression is applied for all the elements of the list. Let’s check it for first element “north”.

“north”.substring(0,1) => “n”,

“n”.toUpperCase() => “N”,

“N”.concat(“north”.substring(1)) => “N”.concat(“orth”) => “North”.

Hence, the output is: [North, East, West, South]

Correction Lambda Built-in Functional Interfaces - 11

A -

```
{Jack, 8000.0}
{Lucy, 14000.0}
```

There are 3 primitive interfaces corresponding to Predicate interface:

DoublePredicate : boolean test(double value);

IntPredicate : boolean test(int value);

LongPredicate : boolean test(long value);

Lambda expression ‘d -> d < 10000’ in this case can be easily assigned to DoublePredicate target type. ‘d -> d < 10000’ returns true if value is less than 10000.

But note: negate() method is used, which means a new DoublePredicate is returned which is just opposite of the given DoublePredicate. After negation, the resultant predicate is ‘d -> d >= 10000’.

Salary is incremented only for Lucy and not Jack.

Correction Lambda Built-in Functional Interfaces - 12

F - Supplier

Date date = obj.get(); means get() method of the interface is invoked. get() method is declared in Supplier interface. All options of Function interface are incorrect.

Supplier interface’s declaration is: public interface Supplier { T get(); }.

Note: No parameters are specified in get() method, this means no-argument constructor of Date class is invoked by Date::new.

Supplier can replace /*INSERT*/.

If you use raw type, Supplier or parameterized type Supplier, then obj.get() method would return Object type.

So Date date = obj.get(); will have to be converted to Date date = (Date)obj.get(); but you are allowed to replace /*INSERT*/ only, hence Supplier and Supplier are incorrect options.

Correction Lambda Built-in Functional Interfaces - 13

B - JamesGosling

BiFunction<String, String, String> interface’s apply method signature will be: String apply(String str1, String str2).

‘String::concat’ is equivalent to LAMBDA expression ‘(str1, str2) -> str1.concat(str2)’. concat method returns new String object after concatenation.

Hence, func.apply(“James”, “Gosling”) returns ‘JamesGosling’.

Correction Lambda Built-in Functional Interfaces - 14

B - Function fog = f.andThen(g);

compose & andThen are default methods defined in Function interface.

Starting with JDK 8, interfaces can define default and static methods.

g.compose(f); means first apply f and then g. Same result is achieved by f.andThen(g); => first apply f and then g.

f.apply(10) = 10 + 10 = 20 and g.apply(20) = 20 * 20 = 400.

Correction Lambda Built-in Functional Interfaces - 15

B - [-10, -100, -1000]

replaceAll(UnaryOperator operator) is the default method added in List interface.

interface UnaryOperator extends Function<T, T>.

As List is of Integer type, this means operator must be of UnaryOperator type only. Its accept method should have signature: Integer apply(Integer s);

Lambda expression ‘i -> -i++’ is correctly defined for apply method.

Unary post increment(++) operator has higher precedence than unary minus(-) operator.

i -> -i++ = i -> -(i++). NO COMPILATION ERROR here.

Post increment operator is applied after the value is used, which means in this case list elements are replaced with -10, -100 and -1000.

Correction Lambda Built-in Functional Interfaces - 16

C - BooleanSupplier

There is only BooleanSupplier available in JDK 1.8.

Correction Lambda Built-in Functional Interfaces - 17

D - true

BiPredicate<T, U> : boolean test(T t, U u);

BiPredicate interface accepts 2 type parameters and these parameters (T,U) are passed to test method, which returns primitive boolean.

In this case, ‘BiPredicate<String, String>’ means test method will have declaration: ‘boolean test(String s1, String s2)’.

‘String::contains’ is equivalent to ‘(str1, str2) -> str1.contains(str2)’

This is an example of “Reference to an Instance Method of an Arbitrary Object of a Particular Type” and not “method reference to static method”, predicate declaration is correct.

BiFunction<T, U, R> : R apply(T t, U u);

BiFunction interface accepts 3 type parameters, first 2 parameters (T,U) are passed to apply method and 3rd type parameter is the return type of apply method.

In this case, ‘BiFunction<String, String, Boolean>’ means apply method will have declaration: ‘Boolean apply(String str1, String str2)’. Given lambda expression ‘(str1, str2) -> { return predicate.test(str1, str2) ? true : false; };’ is the correct implementation of BiFunction<String, String, Boolean> interface.

Please note, the lambda expressions makes use of predicate declared above.

func.apply(“Tomato”, “at”) returns true as “Tomato” contains “at”, hence true is printed on to the console.

Also note that usage of ternary operator (?:) is not needed here, you can simply write ‘return predicate.test(str1, str2);’ as well.

Correction Lambda Built-in Functional Interfaces - 18

E - >World! olleH<

Syntax is correct and without any errors. Methods are chained from left to right.

new StringBuilder(” olleH”) => ” olleH”

” olleH”.reverse() => “Hello”

“Hello”.append(“!dlroW”) => “Hello !dlroW”

“Hello !dlroW”.reverse() => “World! olleH”

Correction Lambda Built-in Functional Interfaces - 19

D - 10

andThen is the default method defined in Consumer interface, so it is invoked on consumer reference variable.

Value passed in the argument of accept method is passed to both the consumer objects.

So, for understanding purpose Line 10 can be split into: add.accept(10); print.accept(10);

add.accept(10) is executed first and it increments the count variable by 10, so count becomes 11.

Then print.accept(10); method prints 10 on to the console.

Check the code of andThen method defined in Consumer interface to understand it better.

Correction Lambda Built-in Functional Interfaces - 20

C - Compilation error

replaceAll(UnaryOperator operator) is the default method added in List interface.

interface UnaryOperator extends Function<T, T>.

As List is of Integer type, this means operator must be of UnaryOperator type only.

'list.replaceAll(operator);' causes compilation error as operator is of UnaryOperator type.

Correction Lambda Built-in Functional Interfaces - 21

A - 55 60 65 70 75

There are 3 primitive streams especially to handle primitive data: DoubleStream, IntStream and LongStream.

There are some important methods available in Stream class:

Stream filter(Predicate<? super T> predicate);

Stream map(Function<? super T, ? extends R> mapper);

void forEach(Consumer<? super T> action);

public static Stream generate(Supplier s){…}

Corresponding primitive streams have similar methods working with primitive version of Functional interfaces:

Double Stream:

DoubleStream filter(DoublePredicate predicate);

DoubleStream map(DoubleUnaryOperator mapper); [Operator is similar to Function]

void forEach(DoubleConsumer action);

public static DoubleStream generate(DoubleSupplier s){…}

IntStream:

IntStream filter(IntPredicate predicate);

IntStream map(IntUnaryOperator mapper); [Operator is similar to Function]

void forEach(IntConsumer action);

public static IntStream generate(IntSupplier s) {…}

LongStream:

LongStream filter(LongPredicate predicate);

LongStream map(LongUnaryOperator mapper); [Operator is similar to Function] void forEach(LongConsumer action);

public static LongStream generate(LongSupplier s) {…}

For exams, you will have to remember some of the important methods and their signature.

LongStream.rangeClosed(51,75) => [51,52,53,…,75]. Both start and end are inclusive.

filter(l -> l % 5 == 0) => [55,60,65,70,75]. filter method accepts LongPredicate and filters the data divisible by 5.

forEach(l -> System.out.print(l + " “)) => Prints the stream data on to the console.

Correction Lambda Built-in Functional Interfaces - 22

C - Function

It is always handy to remember the names and methods of four important built-in functional interfaces:

Supplier : T get();

Function<T, R> : R apply(T t);

Consumer : void accept(T t);

Predicate : boolean test(T t);

Rest of the built-in functional interfaces are either similar to or dependent upon these four interfaces.

Clearly, interface Function can be used instead or defining Generator interface.

Correction Lambda Built-in Functional Interfaces - 23

A - java.util.function

All the built-in functional interfaces are defined inside java.util.function package.

Correction Lambda Built-in Functional Interfaces - 24

D -

```
prevention
prehistoric
```

BiFunction<T, U, R> : R apply(T t, U u);

BiFunction interface accepts 3 type parameters, first 2 parameters (T,U) are passed to apply method and 3rd type parameter is the return type of apply method.

In this case, ‘BiFunction<String, String, String>’ means apply method will have declaration: ‘String apply(String str1, String str2)’. Given lambda expression ‘(str1, str2) -> { return (str1 + str2); };’ is the correct implementation of BiFunction<String, String, String> interface. It simply concatenates the passed strings.

BiPredicate<T, U> : boolean test(T t, U u);

BiPredicate interface accepts 2 type parameters and these parameters (T,U) are passed to test method, which returns primitive boolean.

In this case, ‘BiPredicate<String, String>’ means test method will have declaration: ‘boolean test(String s1, String s2)’. Given lambada expression ‘(str1, str2) -> { return func.apply(str1, str2).length() > 10; };’ is correct implementation of BiPredicate<String, String>. Also note, lambda expression for BiPredicate uses BiFunction. This predicate returns true if combined length of passed strings is greater than 10.

For-each loop simply iterates over the String array elements and prints the string after pre-pending it with “pre” in case the combined length of result string is greater than 10. “prehistoric” has 11 characters and “presentation” has 12 characters and hence these are displayed in the output.

Correction Lambda Built-in Functional Interfaces - 25

A -

```
aba
Abab
```

“or” and “and” method of Predicate interface works just like short-circuit || and && operators.

p1.or(p2) will return {“A”, “ab”, “Aa”, “aba”, “Abab”} and after that and method will retrieve strings of length greater than or equal to 3, this means you would get {“aba”, “Abab”} as the final result.

Correction Lambda Built-in Functional Interfaces - 26

A - happy!

Consumer interface has void accept(T) method, which means in this case, Consumer interface has void accept(String) method.

Given lambda expression accepts String argument and does some operation.

First String is converted to StringBuilder object to use the reverse method. new StringBuilder("!yppahnu").reverse().toString() returns "unhappy!" and "unhappy!".substring(2) returns "happy!", which is printed by System.out.println method.

Correction Lambda Built-in Functional Interfaces - 27

B - java

String::new is the constructor reference for String(char []) constructor and obj.apply(new char[] {‘j’, ‘a’, ‘v’, ‘a’}); would call the constructor at runtime, converting char [] to String. Variable s refers “java”.

If you have issues in understanding method reference syntax, then try to write the corresponding lambda expression first.

For example, in Line 5, I have to write a lambda expression which accepts char [] and returns String object. It can be written as:

Function<char [], String> obj2 = arr -> new String(arr); It is bit easier to understand this syntax.

Correction Lambda Built-in Functional Interfaces - 28

A - 54321

f1.compose(f2) means first apply f2 and then f1.

f2.apply(“12345”) returns “54321” and then f1.apply(“54321”) returns 54321

Correction Lambda Built-in Functional Interfaces - 29

D -

```
*
**
***
```

Lambda expression for Predicate is: s -> s.length() > 3. This means return true if passed string’s length is > 3.

pr1.negate() means return true if passed string’s length is <= 3.

So first three array elements are printed.

Correction Lambda Built-in Functional Interfaces - 30

D - AB CD

BiFunction<String, String, String> interface’s apply method signature will be: String apply(String str1, String str2).

NOTE: Lambda expression’s body is: ‘s2.concat(s1)’ and not ‘s1.concat(s2)’. trim() method trims leading and trailing white spaces and not the white spaces in between.

func.apply(” CD”, ” AB”)

= ” AB”.concat(” CD”).trim();

= ” AB CD”.trim(); = “AB CD”

Correction Lambda Built-in Functional Interfaces - 31

E - 35.0

Though the lambda expression with 2 arrows seems confusing but it is correct syntax. To understand, Line n1 can be re-written as:

```
DoubleFunction<DoubleUnaryOperator> func = (m) -> {
    return (n) -> {
        return m + n;
    };
};
```

And corresponding anonymous class syntax is:

```
DoubleFunction<DoubleUnaryOperator> func = new
        DoubleFunction<DoubleUnaryOperator>() {
    @Override
    public DoubleUnaryOperator apply(double m) {
        DoubleUnaryOperator duo = new DoubleUnaryOperator() {
            @Override
            public double applyAsDouble(double n) {
                return m + n;
            }
        };
        return duo;
    }
};
```

So, there is no issue with Line n1. Let’s check Line n2.

‘func.apply(11)’ returns an instance of DoubleUnaryOperator, in which applyAsDouble(double) method has below implementation:

```
DoubleUnaryOperator duo = new DoubleUnaryOperator() {
    @Override
    public double applyAsDouble(double n) {
        return 11 + n;
    }
};
```

And hence, when applyAsDouble(24) is invoked on above instance of DoubleUnaryOperator, then it returns the result of 11 + 24, which is 35 and as return type is double, so 35.0 is returned.

Correction Lambda Built-in Functional Interfaces - 32

B -

```
Consumer : accept
Function : apply
Supplier : get
Predicate : test
```

It is always handy to remember the names and methods of four important built-in functional interfaces:

Supplier : T get();

Function<T, R> : R apply(T t);

Consumer : void accept(T t);

Predicate : boolean test(T t);

Rest of the built-in functional interfaces are either similar to or dependent upon these four interfaces.

Correction Lambda Built-in Functional Interfaces - 33

B - Compilation error at Line n2

interface UnaryOperator extends Function<T, T>, so its apply function has the signature: T apply(T).

In this case, UnaryOperator is used and hence apply method will have the signature: String apply(String).

Lambda expression ‘s -> s.toString().toUpperCase()’ is the correct implementation of ‘String apply(String)’ method and hence there is no issue with Line n1.

But at Line n2, argument passed to apply method is of StringBuilder type and not String type and hence Line n2 causes compilation error.

Correction Lambda Built-in Functional Interfaces - 34

A - p -> true

B - p -> p.length() < 10

C - p -> !false

D - p -> p.length() >= 1

p -> true means test method returns true for the passed String.

p -> !false means test method returns true for the passed String.

p -> p.length() >= 1 means test method returns true if passed String’s length is greater than or equal to 1 and this is true for all the array elements.

p -> p.length() < 10 means test method returns true if passed String’s length is less than 10 and this is true for all the array elements.

Correction Lambda Built-in Functional Interfaces - 35

A - Compilation error

IntConsumer has single abstract method, ‘void accept(int value);’.

accept(int) method doesn’t return anything.

Lambda expression ‘i -> i * i * i’ returns an int value and hence given lambda expression causes compilation failure.

Correction Lambda Built-in Functional Interfaces - 36

B - import java.util.Comparator;

java.util.Comparator interface is available with Java since JDK 1.2.

So, even though it is a functional interface but Java guys didn’t move it to java.util.function package.

Had Comparator interface moved to java.util.function package, then millions of lines of existing Java codes would have broken. That’s why package of all the existing functional interface was not changed.

Correction Lambda Built-in Functional Interfaces - 37

E - -1

BiFunction<T, U, R> : R apply(T t, U u);

BiFunction interface accepts 3 type parameters, first 2 parameters (T,U) are passed to apply method and 3rd type parameter is the return type of apply method.

In this case, ‘BiFunction<Double, Double, Integer>’ means apply method will have declaration: ‘Integer apply(Double d1, Double d2)’.

Lambda expression should accept 2 Double type parameters and must return Integer object. Lambda expression is:

(d1, d2) -> d1.compareTo(d2); and corresponding method reference syntax is: ‘Double::compareTo’.

This is an example of “Reference to an Instance Method of an Arbitrary Object of a Particular Type” and not “method reference to static method”.

If d1 < d2, then -1 is returned and if d1 > d2, then 1 is returned.

Correction Lambda Built-in Functional Interfaces - 38

D - -50

Though the lambda expression with 2 arrows seems confusing but it is correct syntax. To understand, Line n1 can be re-written as:

```java
LongFunction<LongUnaryOperator> func = (a) -> {
    return (b) -> {
        return b - a;
    };
};
```

And corresponding anonymous class syntax is:

```java
LongFunction<LongUnaryOperator> func = new
        LongFunction<LongUnaryOperator>() {
    @Override
    public LongUnaryOperator apply(long a) {
        LongUnaryOperator operator = new LongUnaryOperator() {
            @Override
            public long applyAsLong(long b) {
                return b - a;
            }
        };
        return operator;
    }
};
```

So, there is no issue with Line n1. Let's check Line n2.

'func.apply(100)' returns an instance of LongUnaryOperator, in which applyAsLong(long) method has below implementation:

```java
LongUnaryOperator operator = new LongUnaryOperator() {
    @Override
    public long applyAsLong(long b) {
        return b - 100;
    }
};
```

When calc(LongUnaryOperator op, long val) is invoked using calc(func.apply(100), 50), op refers to above LongUnaryOperator instance and val is 50.

op.applyAsLong(50); returns 50 - 100, which is -50.

Correction Lambda Built-in Functional Interfaces - 39

C - 13579

In the boolean expression (predicate.test(i)): i is of primitive int type but auto-boxing feature converts it to Integer wrapper type.

for loops works for the numbers from 1 to 10. test(Integer) method of Predicate returns true if passed number is an odd number, so given loop prints only odd numbers.

Correction Lambda Built-in Functional Interfaces - 40

C - Password: Or@cle!

interface UnaryOperator extends Function<T, T>, so its apply function has the signature: T apply(T).

In this case, UnaryOperator is used and hence apply method will have the signature: String apply(String).

Lambda expression ‘s -> s.replace(’a’, ‘@’)’ is the correct implementation of ‘String apply(String)’ method and hence there is no issue with Line n1.

Lambda expression ‘s -> str.concat(s)’ is also the correct implementation of ‘String apply(String)’ method. Don’t get confused with final modifier being used for ‘str’ reference, it is safe to invoke methods on final reference variable but yes you can’t assign another String object to final reference variable. By invoking str.concat(s) a new String object is returned. So, no there is no issue with Line n2 as well.

Let’s solve Line n3:

System.out.println(“Password:” + opr1.apply(opr2.apply(“!”)));

System.out.println(“Password:” + opr1.apply(“Oracle!”)); //opr2.apply(“!”) returns “Oracle!”

System.out.println(“Password:” + “Or@cle!”); //opr1.apply(“Oracle!”) returns “Or@cle!”

Hence, output is: Password: Or@cle!