Correction Java IO Fundamentals - 0

D - Runtime Exception

There is no compilation error in this code.

Optional class doesn’t implement Serializable interface, hence ‘oos.writeObject(optional);’ throws exception at runtime.

Correction Java IO Fundamentals - 1

C - All the pdf files in ‘Test’ directory and its sub-directories will be deleted successfully

File is used to represent both File and Directory.

deleteFiles(new File(“F:\Test”), “.pdf”); => A java file object referring to ‘F:\Test’ is created and on method call variable ‘dir’ refers to it.

dir.listFiles() method returns an File [], this contains the list of immediate files and directories.

NOTE: listFiles() method returns File[] and list() method returns String [].

In this case, array list will have 4 File objects: [{F:\Test\t1.pdf}, {F:\Test\t2.pdf},{F:\Test\A},{F:\Test}].

for loop iterates through each File object and if the File object is a Directory, then it recursively invokes deleteFiles method.

file.isDirectory() method checks if the file object is a File or Directory?

file.getName() returns the name (not the full path) of the file object (such as ‘t1.pdf’, ‘t2.pdf’, ‘A’ etc.).

file.delete() deletes the actual file from the disk.

Correction Java IO Fundamentals - 2

A - No

It is very interesting question and frequently asked in interviews as well.

Initially length of arr is 500000 and all the elements are initialized with 0.

There are 2 options:

1. ‘orig.png’ file’s size is way less than 500000, say 100000 bytes only. This means only 100000 array elements will be replaced and rest will remain 0.

fos.write(arr); => Writes whole array to the stream/file, which means along with 100000 data bytes 400000 extra bytes will also be written. In this case above code can’t produce exact copy.

2. ‘copy.png’ file’s size is not multiple of 500000, say 700000 bytes only.

1st invocation of ‘fis.read(arr)’ will populate all 500000 elements with data and ‘fos.write(arr);’ will correctly write first 500000 bytes to ‘copy.png’ file.

2nd invocation of ‘fis.read(arr)’ will replace 200000 elements with remaining data and leaving 300000 with the data read from previous read operation. This means ‘fos.write(arr);’ on 2nd invocation will write 300000 extra bytes. So for this case as well, above code cannot produce exact copy.

Solution is very simple. Use the overloaded version of write method.

res stores the number of bytes read into the arr. So to get exact copy replace line 12 with ‘fos.write(arr, 0, res)’.

Correction Java IO Fundamentals - 3

D - J

writeChars(String) is a convenient method to write multiple characters, behind the scenes it writes 4 characters as 8 bytes of data.

readChar() method returns 2 bytes of data, interpreted as char. Return type is char and not int.

‘J’ is displayed in the output.

Correction Java IO Fundamentals - 4

B - Above code will never throw NullPointerException.

Two cases are possible:

1. If JVM is associated with the Console (such as running above code from command prompt/window):

System.console() will return the Console object, optional.isPresent() will return true and System.out.println(optional.get()); will print the Console reference (such as java.io.Console@48140564).

2. If JVM is not associated with the Console (such as running above code from an IDE or online):

System.console() will return null and Optional.ofNullable(System.console()); returns an EMPTY Optional.

For EMPTY optional, optional.isPresent() returns false and System.out.println(optional.get()); will not get executed. So you may not get any output in this case.

In both the cases, optional reference will always refer to some Optional object and will never be null. NullPointerException can’t be thrown by calling methods on optional reference.

Correction Java IO Fundamentals - 5

A - F:\

File class has below 2 methods:

public File getParentFile() {…}

public String getParent() {…}

‘dir’ refers to File object for abstract path ‘F:\A\B’.

dir.getParentFile() => returns a File object for abstract path ‘F:\A’

dir.getParentFile().getParent() => returns a String object referring to ‘F:\’

Correction Java IO Fundamentals - 6

C - 3

Counter class implements Serializable, hence objects of Counter class can be serialized using ObjectOutputStream.

State of transient and static fields are not persisted.

While de-serializing, transient fields are initialized to default values (null for reference type and respective Zeros for primitive types) and static fields refer to current value.

In this case, count is static, so it is not persisted. On de-serializing, current value of count is used.

new Counter(); simply increments the variable count by 1.

System.out.println(Counter.getCount()); => Prints 3 on to the console.

Correction Java IO Fundamentals - 7

A - dir.getParentFile().getParentFile()

D - dir.getParentFile().getParent( )

File class has below 2 methods:

public File getParentFile() {…}

public String getParent() {…}

toString() method of File class prints the abstract path of file object on to the console.

dir.getParentFile() => returns a File object for abstract path ‘F:\A’.

dir.getParentFile().getParentFile() => returns a File object for abstract path ‘F:\’, when this file object is passed to println method, its toString() method is invoked and ‘F:\’ gets printed on to the console.

dir.getParentFile().getParent() => Returns a String object containing “F:\”.

dir.getParent() returns String and String class doesn’t contain getParentFile() and getParent() methods.

Correction Java IO Fundamentals - 8

C - 2 will not be printed on to the console.

InputStreamReader class has a constructor ‘public InputStreamReader(InputStream in)’, hence System.in, which is an instance of InputStream can be passed to it.

br.read() reads a single character, so it reads user input as ‘2’ and not 2. ASCII code for character ‘2’ is not 2. In fact, it is 50.

So, 50 will be shown in the output.

OCP exam doesn’t expect you to know exact corresponding ASCII codes but you should know that read() method returns single character (16-bit: 0 to 65535) but as return type is int hence it returns the ASCII value in this case.

If you want to retrieve the user input then use below code:

```
String s = br.readLine();
int i = Integer.parseInt(s); //In case you want to convert to int.
System.out.println(i);
```

Correction Java IO Fundamentals - 9

A - Class Test compiles and executes fine and no output is displayed on to the console

new PrintWriter(“F:\test.txt”) creates a blank file ‘F:\test.txt’.

new PrintWriter(“F:\test.txt”) can throw FileNotFoundException at runtime and given catch handler can access FileNotFoundException as it extends IOException.

public methods of PrintWriter don’t throw IOException. In case of IOException, internal flag variable, ‘trouble’ is set to true.

bw.write(1); is invoked after bw.close() and hence nothing is written to the file.

Correction Java IO Fundamentals - 10

B -

```
Console console = System.console();
char [] pwd = console.readPassword("Enter Password: ");
System.out.println(new String(pwd));
```

new Console(…) causes compilation error as it doesn't have matching constructor. It's no-argument constructor is private.

Correct way to get Console instance is by System.console();

readPassword method is overloaded:

char [] readPassword() {…}

char [] readPassword(String fmtString, Object… args) {…}

console.readPassword("Enter Password:"); => compiles and executes successfully as there are no format specifier in 1st argument, so it is OK to provide just one argument to this method.

Return type of readPassword(…) method is char [] and not String.

Correction Java IO Fundamentals - 11

A - Line 11 causes Runtime exception

As variable ‘bw’ is created outside of try-with-resources block, hence it can easily be accessed in finally block.

There is no issue with ‘try(BufferedWriter writer = bw)’ as well.

NOTE: writer and bw are referring to the same object.

Just before finishing the try-with-resources block, Java runtime invokes writer.close() method, which closes the stream referred by writer object.

But as writer and bw are referring to the same object, hence when bw.flush(); is invoked in finally block, it throws IOException.

Correction Java IO Fundamentals - 12

A - On execution, IOException is printed on to the console

Once the close() method is called on BufferedWriter, same stream cannot be used for writing.

bw.newLine(); tries to append a newline character to the closed stream and hence IOException is thrown.

catch handler is provided for IOException. Control goes inside and prints ‘IOException’ on to the console.

Correction Java IO Fundamentals - 13

A - Runtime Exception

As Student class doesn’t implement Serializable or Externalizable, hence ‘oos.writeObject(stud);’ throws java.io.NotSerializableException.

To persist Student objects using ObjectInputStream, Student class should implement Serializable.

Correction Java IO Fundamentals - 14

A - There is a chance of resource leak

main method declares to throw IOException, hence given code compiles successfully.

Even though, bos.close(); method is invoked but it is not invoked in finally block. If statements before bos.close(); throw any exception, then bos.close() will not execute and this will result in resource leak.

To avoid resource leak, either use try-with-resources statement or provide try-finally block.

Correction Java IO Fundamentals - 15

D - attack

reader.ready() => Returns true if input buffer is not empty or if bytes are available to be read from the underlying byte stream, otherwise false.

reader.skip(long n) => This method is inherited from Reader class and skips the passed n characters.

Let's check how the file contents are processed. Initially available stream: "sdaleteftdeagncedk".

reader.ready() => returns true.

reader.skip(1); => skips 's'. Available Stream: "daleteftdeagncedk".

reader.skip(1); => skips 'd'. Available Stream: "aleteftdeagncedk".

reader.read(); => Reads 'a' and prints it. Available Stream: "leteftdeagncedk".

reader.ready() => returns true.

reader.skip(1); => skips 'l'. Available Stream: "eteftdeagncedk".

reader.skip(1); => skips 'e'. Available Stream: "teftdeagncedk".

reader.read(); => Reads 't' and prints it. Available Stream: "eftdeagncedk".

And so on, until whole stream is exhausted.

'attack' will be printed on to the console.

Correction Java IO Fundamentals - 16

C -

```
A
B
C
```

Format specifier, ‘n’ is for newline.

NOTE: System.out.printf(…) is same as System.out.format(…).

Correction Java IO Fundamentals - 17

D - 0

Initially F:\ is blank.

new File(“F:\f1.txt”); => It just creates a Java object containing abstract path ‘F:.txt’. NO physical file is created on the disc.

Constructors of FileWriter and PrintWriter can create a new file but not directory.

new FileWriter(“F:\dir\f2.txt”); => Throws IOException as ‘dir’ is a folder, which doesn’t exist. This constructor can’t create folders/directories.

new PrintWriter(“F:\f3.txt”); would have created ‘f3.txt’ on the disk but as previous statement threw IOException, hence program ends abruptly after printing the stack trace.

Correction Java IO Fundamentals - 18

C - 20 +10

In format string, format specifier are just replaced.

2$ means 2nd argument, which is 20 and 1$ means 1st argument, which is 10.

Hence ‘System.out.printf(“%2*d*+d”, 10, 20);’ prints ‘20 + 10’ on to the console.

NOTE: System.out.printf(…) is same as System.out.format(…).

Correction Java IO Fundamentals - 19

C - err.log file will be created and it will contain following texts: ONE

new PrintStream(“F:\err.log”) => This will create a new file ‘err.log’ under F: and will create a PrintStream instance.

System.setOut(new PrintStream(“F:\err.log”)); => Sets the out PrintStream to passed object.

System.out.println(“ONE”); => Writes ‘ONE’ to ‘err.log’ file.

System.out.println(1 / 0); => Throws ArithmeticException, which is caught by the catch handler.

System.err.println(“TWO”); => Prints ‘TWO’ on to the console, default err Stream. err PrintStream was not changed using System.setErr(PrintStream) method.

So, ‘err.log’ file contains ‘ONE’ only.

Correction Java IO Fundamentals - 20

E - null : 20

Student class implements Serializable, hence objects of Student class can be serialized using ObjectOutputStream.

State of transient and static fields are not persisted.

While de-serializing, transient fields are initialized to default values (null for reference type and respective Zeros for primitive types) and static fields refer to current value.

In this case, name is transient, so it is not persisted. On de-serializing, null is printed for name and 20 is printed for age.

Correction Java IO Fundamentals - 21

B - It compiles fine but can cause NullPointerException at runtime

C - It waits indefinitely for the user input after displaying the text: What's your name?

This code compiles successfully.

Code inside main method doesn't throw IOException or its subtype but main method is free to declare any exception in its throws clause.

Even though program is executed from command line but System.console() may return null, in case no console available for the underlying OS.

In that case, console.readLine(…) will cause NullPointerException at runtime.

readLine method is a blocking method, it waits for the user action. It waits until user presses Enter key or terminates the program.

Correction Java IO Fundamentals - 22

B -

```
true
false
false
```

Given File methods return boolean and don't throw any checked exception at runtime.

dirs -> referring to File object for abstract path: F:\A\B\C.

Initially F: is blank.

System.out.println(dirs.mkdirs()); => Creates all the directories A, B, C as per abstract path and returns true.

dir -> referring to File object for abstract path: F:\A

System.out.println(dir.mkdir()); => returns false as F:\A directory exists.

System.out.println(dir.delete()); => returns false as F:\A is not empty directory, it contains directory 'B'.

Correction Java IO Fundamentals - 23

A - No

System.getProperty(“path.separator”) returns the path-separator, semicolon(;) on Windows system and colon(:) on Linux system.

So, above code will definitely not create directory named, ‘A’. On windows system, it created directory with name ‘;A’.

You can get file separator by using below code:

System.getProperty(“file.separator”) OR File.separator.

Correction Java IO Fundamentals - 24

A -

```
File file = new File("F:\A\B\C");
file.mkdirs();
```

createNewDirectory() and createNewDirectories() are not available in java.io.File class.

createNewFile() is for creating new file but not directory.

mkdir() is for creating just 1 directory, in this case abstract path is: 'F:\A\B\C', for mkdir() to create directory, 'F:\A\B\' path must exist.

As per question F: doesn't contain any directories, hence mkdir() doesn't create any directory and returns false.

mkdirs() creates all the directories (if not available), specified in the given abstract path and returns true.

Correction Java IO Fundamentals - 25

C - 10 a

‘<’ within format string is used to reuse the argument matched by the previous format specifier.

d is for Integer and x is for hexadecimal. Hexadecimal value for 10 is a.

So,

```
console.format("%d %<x", 10);
```

prints ‘10 a’ on to the console.

NOTE: console.format(…) and console.printf(…) methods are same. In fact, printf(…) method invokes format(…) method.

Correction Java IO Fundamentals - 26

A - Runtime Exception

Class Student implements Serializable but it's super class Person is not Serializable.

While de-serializing of Serializable class, constructor of that class is not invoked.

But if Parent class is not Serializable, then to construct the Parent class object, a no-argument constructor in Parent class is needed. This no-argument constructor initializes the properties to their default values.

As Person class doesn't have no-argument constructor, hence 'ois.readObject();' throws runtime exception.

Correction Java IO Fundamentals - 27

C - 2

Initially F:\ is blank.

new File(“F:\f1.txt”); => It just creates a Java object containing abstract path ‘F:.txt’. NO physical file is created on the disc.

new FileWriter(“F:\f2.txt”); => It creates ‘f2.txt’ under F:.

new PrintWriter(“F:\f3.txt”); => It creates ‘f3.txt’ under F:.

NOTE: Constructors of FileWriter and PrintWriter can create a new file but not directory.

Correction Java IO Fundamentals - 28

A - Runtime Exception

console.format(“%d %x”, 10); => There are 2 format specifiers (%d and %x) in the format string but only one argument (10) is passed.

MissingFormatArgumentException is thrown at runtime.

NOTE: console.format(…) and console.printf(…) methods are same.

In fact, printf(…) method invokes format(…) method.

Correction Java IO Fundamentals - 29

C - None of the other options

java.io.Serializable is a marker interface and hence classes which implement this interface are not required to implement any methods for serialization to work.

Correction Java IO Fundamentals - 30

D - null, 0, Computer Science

Class Student implements Serializable but it's super class Person is not Serializable.

While de-serializing of Serializable class, constructor of that class is not invoked.

But if Parent class is not Serializable, then to construct the Parent class object, a no-argument constructor in Parent class is needed. This no-argument constructor initializes the properties to their default values.

Person class has no-argument constructor. So while de-serialization name and age are initialized to their default values: null and 0 respectively.

course refers to "Computer Science" as it belongs to Serializable class, Student.

In the output, you get 'null, 0, Computer Science'.

Correction Java IO Fundamentals - 31

A - John : 20

Student class implements Serializable, hence objects of Student class can be serialized using ObjectOutputStream.

setName() and setAge() are called after executing ‘oos.writeObject(stud);’ hence these are not reflected in the serialized data.

On de-serializing, persisted Student data is printed on to the console.

Correction Java IO Fundamentals - 32

D - 20 + 10

In format string, format specifier are just replaced.

2$ means 2nd argument, which is 20 and 1$ means 1st argument, which is 10.

Hence ‘System.out.printf(“%2*d*+d”, 10, 20);’ prints ‘20 + 10’ on to the console.

Having more arguments than the format specifiers is OK, extra arguments are ignored but having less number of arguments than format specifiers throws MissingFormatArgumentException at runtime.

NOTE: System.out.printf(…) is same as System.out.format(…).