**7. Java IO+ NIO**

**Question 1:**

Which of the following are valid method implementations?

**public void outputText(PrintWriter pw, String text){**

**try{**

**pw.write(text);**

**}catch(IOException e){**

**System.out.println("exception in writing");**

**}**

**}**

This will not compile because the code in the try block does not throw any exception and therefore the catch block is unreachable.

**public void outputText(PrintWriter pw, String text){**

**pw.write(text);**

**if(pw.checkError()) System.out.println("exception in writing");**

**}**

Note that none of the PrintWriter's methods throw any I/O exceptions because they supress the errors in writing and set an internal flag for error status instead.

The checkError method returns true if there has been a problem in writing.

**public void outputText(PrintWriter pw, String text){**

**boolean flag = pw.write(text);**

**if(!flag) System.out.println("exception in writing");**

**}**

All the write and print methods of PrintWriter return void. Therefore, this code will not compile.

**public void outputText(PrintWriter pw, String text){**

**pw.printf(text).print("success");**

**}**

PrintWriter has printf(Locale l, String format, Object... args) and printf(String format, Object... args) methods that allow you to format the input before printing. These methods return the same PrintWriter object so that you can chain multiple calls as shown in this option.

**public void outputText(PrintWriter pw, String text){**

**pw.println(text).println("success");**

**}**

Although PrintWriter does have a println(String ) method, it returns void. Thus, you cannot chain anything to it. Note that the println methods of PrintWriter cause the data written in the stream to be flushed if automatic flushing is enabled. You can set the autoflush behaviour by using PrintWriter(OutputStream out, boolean autoFlush) constructor while creating a PrintWriter.

**Question 2:**

Which of the following statements regarding java.io.File are true?

**You can delete the actual file or directory represented by the File object using that object.**

File has an instance method public boolean delete() that can delete the file. It returns true if the file has actually been deleted, false otherwise.

**You cannot create a File object if a file or directory does not actually exist by that name.**

You can always create a File object. You can then call public boolean exists() to determine whether the actual file or directory exists or not.

**You can create files in any directory using File class's API.**

Using public boolean createNewFile() method.

**Once created there is no way to change the File object to make it represent a different file or directory.**

Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

**The same File object can be used to traverse between the directories.**

There is no way to do that. You must create a new File object for each directory that you want to traverse.

**Question 3:**

What will the following code print when run?

import java.nio.file.Path;

import java.nio.file.Paths;

public class PathTest {

static Path p1 = Paths.get("c:\\main\\project\\Starter.java");

public static String getData(){

String data = p1.getName(0).toString();

return data;

}

public static void main(String[] args) {

System.out.println(getData());

}

}

**IllegalArgumentException**

**ArrayIndexOutOfBoundsException**

**c:\**

**c:**

**main**

**Note:**

Remember the following 4 points about Path.getName() method :  
  
1. Indices for path names start from 0.  
2. Root (i.e. c:\) is not included in path names.  
3. \ is NOT a part of a path name.  
4. If you pass a negative index or a value greater than or equal to the number of elements, or this path has zero name elements, java.lang.IllegalArgumentException is thrown. It DOES NOT return null.  
  
Thus, for example, If your Path is "c:\\code\\java\\PathTest.java",  
  
p1.getRoot()  is c:\  ((For Unix based environments, the root is usually / ).  
p1.getName(0)  is code  
p1.getName(1)  is java  
p1.getName(2)  is PathTest.java  
p1.getName(3)  will cause IllegalArgumentException to be thrown.

**Question 4:**

Given:

Path p1 = Paths.get("c:\\a\\b\\c.java");

What will p1.getName(2).toString() return?

**a**

**b**

**c**

**c.java**

**Question 5:**

What will the following code print when run?

import java.nio.file.Path;

import java.nio.file.Paths;

public class PathTest {

static Path p1 = Paths.get("c:\\a\\b\\c");

public static String getValue(){

String x = p1.getName(1).toString();

String y = p1.subpath(1,2).toString();

return x+" : "+y;

}

public static void main(String[] args) {

System.out.println(getValue());

}

}

**\b : \b**

**b : b**

**b : b\c\**

**a : a\b**

**b : b\c**

**Note:**

Remember the following points about Path.subpath(int beginIndex, int endIndex)

1.Indexing starts from 0.

2. Root (i.e. c:\) is not considered as the beginning.

3. name at beginIndex is included but name at endIndex is not.

4. paths do not start or end with \.

Thus, if your path is "c:\\a\\b\\c",

subpath(1,1) will cause IllegalArgumentException to be thrown.

subpath(1,2) will correspond to b.

subpath(1,3) will correspond to b/c.

Remember the following 4 points about Path.getName() method :

1. Indices for path names start from 0.

2. Root (i.e. c:\) is not included in path names.

3. \ is NOT a part of a path name.

4. If you pass a negative index or a value greater than or equal to the number of elements, or this path has zero name elements, java.lang.IllegalArgumentException is thrown.

It DOES NOT return null. Thus, for example,

If your Path is "c:\\code\\java\\PathTest.java",

p1.getRoot()  is c:\  ((For Unix based environments, the root is usually / ).

p1.getName(0)  is code

p1.getName(1)  is java

p1.getName(2)  is PathTest.java

p1.getName(3)  will cause IllegalArgumentException to be thrown.

**Question 6:**

What will the following program print when compiled and run?

class Boo implements Serializable {

transient int ti = 10;

static int si = 20;

}

public class TestClass

{

public static void main(String[] args) throws Exception

{

Boo boo = new Boo();

boo.si++;

System.out.println(boo.ti+" "+boo.si);

var fos = new FileOutputStream("c:\\temp\\boo.ser");

var os = new ObjectOutputStream(fos);

os.writeObject(boo);

os.close();

var fis = new FileInputStream("c:\\temp\\boo.ser");

var is = new ObjectInputStream(fis);

boo = (Boo) is.readObject();

is.close();

System.out.println(boo.ti+" "+boo.si);

}

}

**It will not compile.**

**It will throw an exception at run time.**

**10 21**

**10 21**

**10 21**

**10 20**

**10 21**

**0 20**

**10 21**

**0 21**

**Note:**

Remember that transient fields and static fields are never serialized. Constructor, instance blocks, and field initialization of the class being deserialized are also not invoked. So, when boo is deserialized, the value of ti is set to 0.

The class Boo is loaded as soon as the code refers to the class (here, it happens at Boo boo = new Boo(); ), and so the static int si is initialized to the value given in the class code i.e. 20 and then it is incremented to 21 because of boo.si++;. This part has nothing to do with serialization. So when you deserialize an instance of Boo, Boo.si is not affected and is not reset to 20.

Therefore, if you run the program again with just the deserialization part, you will see that si is 20 and not 21.

**Question 7:**

Which of the following method implementations will write a boolean value to the underlying stream?

**public void usePrintWriter(PrintWriter pw){**

**boolean bval = true;**

**pw.writeBoolean(bval);**

**}**

PrintWriter does not have write<Primitive> methods such as writeInt, writeBoolean, WriteLong. It has overloaded print methods for writing various primitives.

**public void usePrintWriter(PrintWriter pw) throws IOException{**

**boolean bval = true;**

**pw.write(bval);**

**}**

PrintWriter does not have write(boolean ) method. It does have write(String), write(int ), write(char[] ) methods. It also has write(char[] buf, int off, int len) and write(String buf, int off, int len) methods that let you write a portion of the input buf.

**public void usePrintWriter(PrintWriter pw) throws IOException{**

**boolean bval = true;**

**pw.print(bval);**

**}**

Although the throws IOException clause is not required here, it is not invalid.

**public void usePrintWriter(PrintWriter pw) {**

**boolean bval = true;**

**pw.print(bval);**

**}**

**public void usePrintWriter(PrintWriter pw) {**

**boolean bval = true;**

**pw.println(bval);**

**}**

**Note:**

Remember that none of PrintWriter's print or write methods throw I/O exceptions (although some of its constructors may). This is unlike other streams, where you need to include exception handling (i.e. a try/catch or throws clause) when you use the stream.

**Question 8:**

Consider the following code:

//Assume appropriate imports

public class FileCopier {

public static void copy(String records1, String records2) throws IOException {

try (

InputStream is = new FileInputStream(records1);

OutputStream os = new FileOutputStream(records2);) {

var buffer = new byte[1024];

var bytesRead = 0;

while ((bytesRead = is.read(buffer)) != -1) {

os.write(buffer, 0, bytesRead);

}

} catch (FileNotFoundException | java.io.InvalidClassException e) {

e.printStackTrace();

}

}

public static void main(String[] args) throws Exception {

copy("c:\\temp\\test1.txt", "c:\\temp\\test2.txt");

}

}

Given that test1.txt exists but test2.txt does not exist, what will happen when the above program is compiled and run?

**The program will not compile.**

**The program will compile and run without any exception. test2.txt will be created automatically and contents of test1.txt will be copied to it.**

**The program will compile and run without any exception but test2.txt will not be created.**

**An exception will be thrown at run time if the size of test1.txt is not a multiple of 1024.**

The read method reads the bytes that are available even if the number of available bytes is less than the buffer size. The method returns the actual number of bytes read. It does not read more bytes than the size of the buffer. Hence, the need for a loop.

**Question 9:**

What will the following code print?

Path p1 = Paths.get("c:\\temp\\test.txt");

Path p2 = Paths.get("c:\\temp\\report.pdf");

System.out.println(p1.resolve(p2));

**..\report.pdf**

This output will be produced by p1.relativize(p2).

**temp\report.pdf**

**report.pdf**

**c:\temp\report.pdf**

When the argument to resolve starts with the root (such as c: or, on \*nix, a /), the result is same as the argument.

**Note:**

Please go through the following description of Path.resolve() method as given in JavaDoc API:

public Path resolve(Path other)

Resolve the given path against this path.

If the other parameter is an absolute path then this method trivially returns other. If other is an empty path then this method trivially returns this path. Otherwise this method considers this path to be a directory and resolves the given path against this path. In the simplest case, the given path does not have a root component, in which case this method joins the given path to this path and returns a resulting path that ends with the given path. Where the given path has a root component then resolution is highly implementation dependent and therefore unspecified.

Parameters:

other - the path to resolve against this path

Returns:

the resulting path

**Question 10:**

Given that c:\temp\pathtest is a directory that contains several directories. Each sub directory contains several files but there is exactly one regular file named test.txt within the whole directory structure.

Which of the given options can be inserted in the code below so that it will print complete path of test.txt?

try{

Stream<Path> s = null;

INSERT CODE HERE

s.forEach(System.out::println);

}catch(IOException ioe){

ioe.printStackTrace();

}

**s = Files.list(Paths.get("c:\\temp\\pathtest\\\*\*\\test.txt"));**

This will throw an exception at run time. For example:

java.nio.file.InvalidPathException: Illegal char <\*> at index 17: c:\temp\pathtest\\*\*\test.txt

**s = Files.walk(Paths.get("c:\\temp\\pathtest"), "test.txt");**

It will not compile because the Files class does not have a walk method that takes a string as the second parameter.

It does have two other walk methods:

public static Stream<Path> walk(Path start, FileVisitOption... options) throws IOException Return a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.

public static Stream<Path> walk(Path start, int maxDepth, FileVisitOption... options) throws IOException Return a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.

**s = Files.find(Paths.get("c:\\temp\\pathtest"), Integer.MAX\_VALUE, (p, a)->p.endsWith("test.txt")&& a.isRegularFile());**

public static Stream<Path> find(Path start, int maxDepth, BiPredicate<Path,BasicFileAttributes> matcher, FileVisitOption... options) throws IOException

Return a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.

**s = Files.find(Paths.get("test.txt"));**

This will not compile because of incorrect arguments to find method.

public static Stream<Path> find(Path start, int maxDepth, BiPredicate<Path,BasicFileAttributes> matcher, FileVisitOption... options) throws IOException

Return a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.

**s = Files.list(Paths.get("c:\\temp\\pathtest"), (p, a)->p.endsWith(".txt")&&a.isRegularFile());**

It will not compile because list method does not filter the paths based on a criteria. It is not recursive either.

public static Stream<Path> list(Path dir) throws IOException Return a lazily populated Stream, the elements of which are the entries in the directory. The listing is not recursive.

**s = Files.walk(Paths.get("c:\\temp\\pathtest"), (p, a)->p.endsWith(".txt")&&a.isRegularFile());**

It will not compile because you cannot pass a function to test each path while walking through the tree using the walk method.

Files class has the following two walk methods:

public static Stream<Path> walk(Path start, FileVisitOption... options) throws IOException Return a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.

public static Stream<Path> walk(Path start, int maxDepth, FileVisitOption... options) throws IOException

Return a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.

**Question 11:**

Given:

public class TestClass

{

public static void main(String[] args) {

String s = "/usr/home1/test.txt";

String d = "/usr/home2/test.txt";

//INSERT CODE HERE

}catch(Exception e){ }

}

}

Which of the following code fragments can be inserted in the above code so that the file test.txt can be moved to the destination location even if a file by the same name already exists at the destination location?

**try{**

**Files.move(Paths.get(s), Paths.get(d), StandardCopyOption.CREATE\_NEW);**

**Files.delete(Paths.get(s));**

CREATE\_NEW is not a valid field in StandardCopyOption. Only the following three are valid:

ATOMIC\_MOVE: Move the file as an atomic file system operation.

COPY\_ATTRIBUTES: Copy attributes to the new file.

REPLACE\_EXISTING: Replace an existing file if it exists.

**try{**

**Files.move(Paths.get(s), Paths.get(d), StandardCopyOption.REPLACE\_EXISTING);**

**try(FileChannel in = new FileInputStream(s).getChannel();**

**FileChannel out = new FileOutputStream(d).getChannel()){**

**in.transferTo(0, in.size(), out);**

**}catch(Exception e){ }**

This code is actually valid but it will only copy the file contents. It will not move the file.

Note that FileChannel is not on the exam but some candidates have reported seeing an option like this.

**try{**

**Files.move(Paths.get(s), Paths.get(d));**

By default, Files.move attempts to move the file to the target file, failing if the target file exists except if the source and target are the same file, in which case this method has no effect.

Therefore, this is a valid method call but it will throw an exception if the file already exists at the destination.

**Question 12:**

Given that the file test.txt contains :

12345678

What will the following code print when compiled and run?

public static void main(String[] args) throws Exception{

try(var fis = new FileInputStream("c:\\temp\\test.txt");

var isr = new InputStreamReader(fis)){

while(isr.ready()){

isr.skip(1);

int i = isr.read();

char c = (char) i;

System.out.print(c);

}

}

}

**It will not compile.**

There is no problem with the code.

**It will throw an exeception when run.**

**It will print just 2**

**It will run without any exception but will not print anything.**

**It will print 2468**

The ready method just checks if there are more bytes available to read.

The skip method skips the given number of characters i.e. it basically moves the file pointer one character ahead.

The read method reads one character.

Overall, the code simply skips one character after reading each character. Therefore, it prints 2468.

**Question 13:**

Which of the following methods are available in java.io.Console?

**readPassword**

char[] readPassword()

Reads a password or passphrase from the console with echoing disabled

char[] readPassword(String fmt, Object... args)

Provides a formatted prompt, then reads a password or passphrase from the console with echoing disabled.

**reader**

java.io.Reader reader()

Retrieves the unique Reader object associated with this console.

**writer**

java.io.PrintWriter writer()

Retrieves the unique PrintWriter object associated with this console.

**readLine**

String readLine()

Reads a single line of text from the console.

String readLine(String fmt, Object... args)

Provides a formatted prompt, then reads a single line of text from the console.

**read**

There is no such method. To read a char, you have to first get a Reader and then call read() on it.

**getPassword**

**format**

Console format(String fmt, Object... args)

Writes a formatted string to this console's output stream using the specified format string and arguments.

Notice that it returns the same Console object again. Thus, you can chain the format calls. For example,

con.format("%dth visitor logged in.", n).format("Id is %d.", id).format("Name is %s", name);

**Question 14:**

What will the following code print when run?

import java.nio.file.Path;

import java.nio.file.Paths;

public class PathTest {

static Path p1 = Paths.get("c:\\finance\\data\\reports\\daily\\pnl.txt");

public static void main(String[] args) {

System.out.println(p1.subpath(0, 2));

}

}

**finance\data**

**finance\data\**

**\finance\data\reports**

**c:\finance\data**

**c:\finance**

**Note:**

Remember the following points about Path.subpath(int beginIndex, int endIndex)  
1. Indexing starts from 0.  
2. Root (i.e. c:\) is not considered as the beginning.  
3. name at beginIndex is included but name at endIndex is not.  
4. paths do not start or end with \.  
  
Thus, in case of "c:\\finance\\data\\reports\\daily\\pnl.txt", name at 0 is finance and name at 2 is reports. However, since the name at endIndex is excluded, subpath(0, 2) will correspond to finance\data.  
  
  
The following is the API description for this method:  
  
public Path subpath(int beginIndex, int endIndex)  
Returns a relative Path that is a subsequence of the name elements of this path.  
The beginIndex and endIndex parameters specify the subsequence of name elements. The name that is closest to the root in the directory hierarchy has index 0. The name that is farthest from the root has index count-1. The returned Path object has the name elements that begin at beginIndex and extend to the element at index endIndex-1.  
  
Parameters:  
beginIndex - the index of the first element, inclusive  
endIndex - the index of the last element, exclusive  
Returns:  
a new Path object that is a subsequence of the name elements in this Path  
Throws:  
IllegalArgumentException - if beginIndex is negative, or greater than or equal to the number of elements. If endIndex is less than or equal to beginIndex, or larger than the number of elements.

**Question 15:**

Consider the directory structure and its contents shown in the figure.

(c:\temp is a directory that contains two text files - test1.txt and text2.txt)

What should be inserted at //Line 10 in the following code so that it will write "hello" to text2.txt?

public static void writeData() throws Exception{

var p1 = Paths.get("c:\\temp\\test1.txt");

var p2 = //LINE 10 - INSERT CODE HERE

var bw = new BufferedWriter(new FileWriter(p2.toFile()));

bw.write("hello");

bw.close();

}

**p1.resolve("text2.txt");**

This will return "c:\temp\test1.txt\text2.txt", which is not what you want. The following is  how resolve(String other) works:

If the other parameter is an absolute path then this method trivially returns other. If other is an empty path then this method trivially returns this path. Otherwise this method considers this path to be a directory and resolves the given path against this path. In the simplest case, the given path does not have a root component, in which case this method joins the given path to this path and returns a resulting path that ends with the given path. Where the given path has a root component then resolution is highly implementation dependent and therefore unspecified.

**p1.relativize("c:\\temp\\text2.txt");**

This is wrong for two reasons –

1. relativize method does not take a String as an argument. It takes a Path object.

2. relativize method is meant to convert an absolute path into a relative path. But here, we want to convert a relative path to an absolute path. For example,

p2 = p1.relativize(Paths.get("c:\\temp\\text2.txt")); will produce "..\text2.txt".

**p1.resolveSibling("text2.txt");**

**p1.relativize(Paths.get("text2.txt"));**

This will thrown an IllegalArgumentException saying 'other' is different type of Path. This is because a relative path cannot be constructed if only one of the paths is an absolute path. Here, p1 is an absolute path (because it starts with a root) and p2 is a relative path.

You already have the absolute path to test1.txt in p1. Further, it is given that test1.txt is in the same directory as text2.txt i.e. both the files are siblings.  
  
To open test2.txt, you need to determine the absolute path for text2.txt using the absolute path for test1.txt. In other words, you are trying to get the absolute path for a file that exists in the same directory as the original file. The method resolveSibling is meant exactly for this purpose.    
  
This will set p2 to c:\temp\text2.txt, which can then be used to create File object.  
  
You should go through the following JavaDoc API description for resolveSibling method.  
  
public Path resolveSibling(String other) or public Path resolveSibling(Path other) :-  
  
Resolves the given path against this path's parent path. This is useful where a file name needs to be replaced with another file name. For example, suppose that the name separator is "/" and a path represents "dir1/dir2/foo", then invoking this method with the Path "bar" will result in the Path "dir1/dir2/bar". If this path does not have a parent path, or other is absolute, then this method returns other. If other is an empty path then this method returns this path's parent, or where this path doesn't have a parent, the empty path.

**Question 16:**

Given:

var bfr = new BufferedReader(new FileReader("c:\\temp\\pathtest\\a.java"));

var bfw = new BufferedWriter(new FileWriter("c:\\temp\\pathtest\\b.java"));

String line = null;

while( (line=bfr.readLine()) != null ){

bfw.append(line);

}

//INSERT CODE HERE

bfw.close();

Which of the following lines is required to be inserted in the code above so that content in b.java will be overwritten with the content in a.java?

**bfr.close();**

**bfw.close();**

**bfr.flush();**

**bfw.flush();**

**None of the above.**

1. BufferedWriter's append method = works same as the write(String) method. It doesn't really append the data to the end of the existing content. It overwrites the existing content.  
2. The flush method flushes the stream and makes sure any data that is in the stream but is not written to the file yet, is written to the file. It does not close the stream. A call to flush is useful when you want to write the contents to the file but don't want to close the writer yet.  
3. The close method flushes the stream and makes sure that all data is actually written to the file.  
  
Since the given code includes a call to close(), there is no need for a call to flush.

**Question 17:**

What will the following code fragment print?

Path p1 = Paths.get("photos\\..\\beaches\\.\\calangute\\a.txt");

Path p2 = p1.normalize();

Path p3 = p1.relativize(p2);

Path p4 = p2.relativize(p1);

System.out.println(

p1.getNameCount()+" "+p2.getNameCount()+" "+

p3.getNameCount()+" "+p4.getNameCount());

**6 4 10 10**

**7 4 11 10**

**7 3 8 9**

**6 3 1 1**

1. p1 has 6 components and so p1.getNameCount() will return 6.

2. normalize applies all the .. and . contained in the path to the path. Therefore, p2 contains beaches\calangute\a.txt, that is 3 components.

3. p3 contains an an empty path because both paths are equal after normalization.

4. p4 contains an an empty path because both paths are equal after normalization.

NOTE: Implementation for relativize method has changed in Java 11. It normalizes the paths before relativizing. That is why, for Java 11, it prints 6 3 1 1, while on Java 8, it prints 6 3 9 9.

**Note:**

You need to understand how relativize works for the purpose of the exam. The basic idea of relativize is to determine a path, which, when applied to the original path will give you the path that was passed. For example, "a/b" relativize "c/d" is "../../c/d" because if you are in directory b, you have to go two steps back and then one step forward to c and another step forward to d to be in d. However, "a/c" relativize "a/b" is "../b" because you have to go only one step back to a and then one step forward to b.  
  
Please go through the following description of relativize() method, which explains how it works in more detail.  
  
public Path relativize(Path other)  
Constructs a relative path between this path and a given path.  
Relativization is the inverse of resolution. This method attempts to construct a relative path that when resolved against this path, yields a path that locates the same file as the given path. For example, on UNIX, if this path is "/a/b" and the given path is "/a/b/c/d" then the resulting relative path would be "c/d". Where this path and the given path do not have a root component, then a relative path can be constructed. A relative path cannot be constructed if only one of the paths have a root component. Where both paths have a root component then it is implementation dependent if a relative path can be constructed. If this path and the given path are equal then an empty path is returned.  
  
For any two normalized paths p and q, where q does not have a root component,  
  
p.relativize(p .resolve(q)).equals(q)  
When symbolic links are supported, then whether the resulting path, when resolved against this path, yields a path that can be used to locate the same file as other is implementation dependent. For example, if this path is "/a/b" and the given path is "/a/x" then the resulting relative path may be "../x". If "b" is a symbolic link then is implementation dependent if "a/b/../x" would locate the same file as "/a/x".  
  
Parameters:  
other - the path to relativize against this path  
Returns:  
the resulting relative path, or an empty path if both paths are equal

**Question 18:**

Given:

import java.io.\*;

class TestClass{

public static void main(String[] args) throws Exception{

try(var bfr = new BufferedReader(new InputStreamReader(System.in))){

System.out.println("Enter Number:");

var s = bfr.readLine();

System.out.println("Your Number is : "+s);

}catch(Exception e){

e.printStackTrace();

}

}

}

What will be the output if the above code is executed using the following command:

java TestClass 123

**Enter Number: (Program is stuck after printing the above)**

**Enter Number: Your Number is:123**

**Enter Number: Your Number is:TestClass 123**

**An exception stack trace will be printed.**

**Note:**

The program expects you to type something on the console when you try to read from the standard input stream. The input is considered complete when you type the enter key. Until you type the enter key, the program will keep buffering whatever you type on the console.

**Question 19:**

What will the following code fragment print?

Path p1 = Paths.get("photos/goa");

Path p2 = Paths.get("/index.html");

Path p3 = p1.relativize(p2);

System.out.println(p3);

**..\index.html**

**\index.html**

**\photos\index.html**

**\photos\goa\index.html**

**java.lang.IllegalArgumentException will be thrown**

Note that if one path has a root (for example, if a path starts with a // or c:) and the other does not, relativize cannot work and it will throw an IllegalArgumentException.

**Question 20:**

Given the following code:

var raf = new RandomAccessFile("c:\\temp\\test.txt", "rwd");

raf.writeChars("hello");

raf.close();

Which of the following statements are correct?

(Assume that the code has appropriate security permissions.)

**If the file test.txt does not exist, an attempt will be made to create it.**

**If the file test.txt does not exist, an exception will be thrown.**

**If the file test.txt exists, an exception will be thrown.**

**If the file test.txt exists, it will be overwritten and all the existing data will be lost.**

Only the initial 5 characters (i.e. 10 bytes) of the file will be overwritten. Any existing data beyond the first 10 bytes will be left untouched.

**If the file test.txt exists, the given characters will be appended to the end of the existing data.**

When you open the file, the file pointer is at the first position. So the given characters will be written at the beginning of the file.

The permitted values for the access mode and their meanings are:  
  
"r": Open for reading only. Invoking any of the write methods of the resulting object will cause an IOException to be thrown.  
"rw": Open for reading and writing. If the file does not already exist then an attempt will be made to create it.  
"rws": Open for reading and writing, as with "rw", and also require that every update to the file's content or metadata be written synchronously to the underlying storage device.  
"rwd": Open for reading and writing, as with "rw", and also require that every update to the file's content be written synchronously to the underlying storage device.

**Question 21:**

Consider the following code :

String id = c.readLine("%s", "Enter UserId:"); //1

System.out.println("userid is " + id); //2

String pwd = c.readPassword("%s", "Enter Password :"); //3

System.out.println("password is " + pwd); //4

Assuming that c is a valid reference to java.io.Console and that a user types jack as userid and jj123 as password, what will be the output on the console?

**Enter UserId:jack userid is jack**

**Enter Password : password is jj123**

**Enter UserId:jack userid is jack**

**Enter Password :\*\*\*\*\* password is jj123**

**Enter UserId:jack userid is jack**

**Enter Password : password is \*\*\*\***

**Enter UserId:jack userid is jack**

**Enter Password : password is jj123**

**It will not compile.**

Note that the return type of readPassword is char[] and not a String. So it will not compile.

**Note:**

If you replace String pwd to char[] pwd, you will get the following output:

Enter UserId:jack

userid is jack

Enter Password :

password is [C@1fb8ee3

Observe that password is not echoed while the user is typing and char[] is an object, which is printed out as [C@1fb8ee3.

**Question 22:**

The following are complete contents of ConsoleTest.java:

import java.io.Console;

public class ConsoleTest {

public static void main(String[] args) {

var c = System.console(); //1

char[] line = c.readPassword("Please enter your pwd:"); //2

System.out.println("Pwd is "+new String(line));

}

}

What will happen when it is compiled and run from the command line?

**It will print whatever password is entered by the user.**

**It will print a garbled version of whatever password is entered by the user.**

**It will throw an exception at run time.**

**It will not compile because it does not incorporate exception handling for the readPassword method.**

Remember that the readLine and readPassword methods of Console do not declare any checked exceptions. Nor does System.console(). Therefore, calls to these methods need not be wrapped in a try block or declared in the throws clause of the calling method.

readLine and readPassword methods may throw IOError if there is a problem in reading from the console. System.console simply returns null if a Console is not available to the program.

**It will not compile because it does not incorporate exception handling for System.console() method.**

**Question 23:**

Consider the following code:

import java.io.\*;

public class Test

{

public static void main(String[] args) throws Exception

{

var fw = new FileWriter("text.txt");

// fw.write("hello"); //1

fw.close();

}

}

Which of the following statements are correct?

**It will throw an exception if  text.txt does not exist.**

**It will create text.txt file in the filesystem if it does not exist.**

**It will not throw an exception if text.txt does not exist and it will not create a file either because nothing is being written to the file.**

It will create an empty file.

**It will throw an exception if //1 is uncommented and if text.txt does not exist.**

If the file does not exist, it will be created and data will be written to it.

**It will throw an exception if text.txt already exists.**

If the file already exists, it will be overwritten with a new file. To append to the existing file, the following constructor should be used. public FileWriter(String fileName,  boolean append)

**Question 24:**

Consider the following code:

public static boolean isValid(Path p){

return p.startsWith("temp") && p.endsWith("clients.dat");

}

public static void writeData() {

var p1 = Paths.get("\\temp\\records");

var p2 = p1.resolve("clients.dat");

System.out.println(p2+" "+isValid(p2));

}

What will be printed when the method writeData() is executed?

**\temp\records\clients false**

**temp\records\clients.dat false**

**\temp\records\clients.dat false**

**temp\records\clients.dat true**

**clients.dat false**

**\clients.dat false**

**Note:**

p2 will be set to \temp\records\clients.dat. Since it starts with \temp and not with temp, the method isValid will return false.

**Question 25:**

Assume that the following directory exists:

c:\a\b\c

A File object is created as follows:

var f = new File("c:\\a\\b\\c\\d\\e");

Given that directories d and e do not exist under c, which of the following statements are correct?

**The given line of code will throw an exception at run time.**

A file or directory need not exist to create a File object.

**f.mkdir(); will create directory d under c and directory e under d.**

mkdir() can only create the last component of a path. It cannot create a directory structure. For that you must use mkdirs().

**f.mkdirs(); will create directory d under c and directory e under d.**

public boolean mkdirs() Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories. Returns: true if and only if the directory was created, along with all necessary parent directories; false otherwise

**f.getParentFile() will return a File Object representing c:\a\b\c\d**

getParent() returns a String and getParentFile() returns a File object.

**None of these.**

**Question 26:**

Consider the following code:

//Assume appropriate imports

public class FileCopier {

public static void copy(String records1, String records2) throws IOException {

try (

InputStream is = new FileInputStream(records1);

OutputStream os = new FileOutputStream(records2);) {

var buffer = new byte[1024];

var bytesRead = 0;

while ((bytesRead = is.read(buffer)) != -1) {

os.write(buffer, 0, bytesRead);

}

} catch (FileNotFoundException | java.io.InvalidClassException e) {

e.printStackTrace();

}

}

public static void main(String[] args) throws Exception {

copy("c:\\temp\\test1.txt", "c:\\temp\\test2.txt");

}

}

Given that test1.txt exists but test2.txt does not exist, what will happen when the above program is compiled and run?

**The program will not compile.**

**The program will compile and run without any exception. test2.txt will be created automatically and contents of test1.txt will be copied to it.**

**The program will compile and run without any exception but test2.txt will not be created.**

**An exception will be thrown at run time if the size of test1.txt is not a multiple of 1024.**

The read method reads the bytes that are available even if the number of available bytes is less than the buffer size. The method returns the actual number of bytes read. It does not read more bytes than the size of the buffer. Hence, the need for a loop.

**Question 27:**

What will the following code print when run?

import java.nio.file.Path;

import java.nio.file.Paths;

public class PathTest {

static Path p1 = Paths.get("c:\\main\\project\\Starter.java");

public static String getRoot(){

String root = p1.getRoot().toString();

return root;

}

public static void main(String[] args) {

System.out.println(getRoot());

}

}

**\**

**c:**

**c:\**

**It will print an empty string.**

Path getRoot() Returns the root component of this path as a Path object, or null if this path does not have a root component.

Path getName(int index)

Returns a name element of this path as a Path object. The index parameter is the index of the name element to return. The element that is closest to the root in the directory hierarchy has index 0. The element that is farthest from the root has index count-1.

Parameters: index - the index of the element

**Question 28:**

Consider the following code:

import java.io.\*;

public class TestClass {

public static void main(String[] args) throws Exception {

var f = new File("x"); //1

var bfr1 = new BufferedReader(new FileReader(f)); //2

var bfr2 = new BufferedReader( bfr1 ); //3

var pw = new PrintWriter(new FileReader(f)); //4

}

}

Select the correct statements about the above program.

**// 1 will throw an exception at runtime if a file named "x" does not exist.**

You can always create a File object whether or not an actual file or directory by that name exists.

**//2 and //3 will compile without any error.**

A BufferedReader can wrap any Reader. Both FileReader and BufferedReader are Readers so both //2 and //3 are valid.

**//4 will compile without any error.**

A Reader can't be converted into a Writer or vice-versa. In other words, there is no way you can chain a Reader and a Writer together.

**The complete program will compile without any error.**

None of these.

**Question 29:**

Given:

public static void createFile(String name) throws Exception{

try (

OutputStream os = new FileOutputStream(name); ) {

//INSERT CODE HERE

//flush and close the streams that are open

}

}

Which of the following combinations of the lines of code and their outcome when inserted above, are correct?

**var pw = new PrintWriter(os);**

**pw.write(1);**

**Size of the file depends on default character encoding.**

PrintWriter's write method writes a single character to the file. The size in bytes of a character depends on the default character encoding of the underlying platform.

For example, if the encoding is UTF-8, only 1 byte will be written and the size of the file will be 1 byte.

**os.write(99);**

**A file of size 1 byte will be created.**

Note that the write(int b) method of various streams based classes such as FileOutputStream take an int parameter but write only the low 8 bits (i.e. 1 byte) of that integer.

DataOutputStream provides methods such as writeInt, writeChar, and writeDouble, for writing complete value of the primitives to a file. So if you want to write an integer to the file, you should use writeInt(1) in which case a file of size 4 bytes will be created. You can read back the stored primitives using methods such as DataInputSream.readInt(). (Note: DataInput/DataOutputStream is not mentioned explicitly in the exam objectives.)

**var bos = new BufferedOutputStream(os); var pw = new PrintWriter(bos); pw.print(99); A file of size 1 byte will be created.**

PrintWriter's print(int) method actually writes the string produced by String.valueOf(int). This string is translated into bytes according to the platform's default character encoding, and these bytes are written in using the write(int) method. Therefore, in this case, if the default character encoding is UTF-8, 2 bytes will be written.

**os.writeInt(99); A file of size 4 bytes will be created.**

OutputStream does not provide methods for writing primitives. It writes bytes only. Therefore, this will not compile.

**var pw = new PrintWriter(os); pw.writeInt(1); A file of size 4 bytes will be created.**

PrintWriter does not provide explicit methods for writing primitives (i.e. writeInt, writeBoolean, etc.). It has overloaded print methods that take various primitives (i.e. print(int), print(boolean), print(long), and print(char) as arguments). Therefore, this will not compile.

**Question 30:**

Consider the following classes:

class Boo {

public Boo(){ System.out.println("In Boo"); }

}

class BooBoo extends Boo {

public BooBoo(){ System.out.println("In BooBoo"); }

}

class Moo extends BooBoo implements Serializable {

int moo = 10; { System.out.println("moo set to 10"); }

public Moo(){ System.out.println("In Moo"); }

}

First, the following code was executed and the file moo1.ser was created successfully:

Moo moo = new Moo();

moo.moo = 20;

FileOutputStream fos = new FileOutputStream("c:\\temp\\moo1.ser");

ObjectOutputStream os = new ObjectOutputStream(fos);

os.writeObject(moo);

os.close();

Next, the following code was executed.

FileInputStream fis = new FileInputStream("c:\\temp\\moo1.ser");

ObjectInputStream is = new ObjectInputStream(fis);

Moo moo = (Moo) is.readObject();

is.close();

System.out.println(moo.moo);

Which of the following will be a part of the output of the second piece of code?

**In Boo**

**In BooBoo**

**In Moo**

**10**

**20**

**moo set to 10**

**Note:**

During deserialization, the constructor of the class (or any static or instance blocks) is not executed. However, if the super class does not implement Serializable, its constructor is called. So here, BooBoo and Boo are not Serializable. So, their constructor is invoked.

**Question 31:**

Given:

public static void reader(String fileName1) throws Exception{

try (var fr = new FileReader(fileName1);) {

int charRead = 0;

while ((charRead = fr.read()) != -1) {

System.out.println("Read char " + charRead);

}

}

}

What can be done to the above code to make it read Strings instead of chars?

**Chain fr to a StringReader and use its readString method.**

While StringReader is a valid class but it creates a Reader out of a String. It does not read Strings from a Reader. For example: StringReader sr = new StringReader("some long string");

**Use fr.readString instead of fr.read.**

FileReader doesn't provide higher level methods for reading Strings.

**Chain fr to a BufferedReader use its readLine method**

A Reader can be chained to a BufferedReader to read Strings. BufferedReader has readLine method that returns a String.

**Chain fr to a DataReader and use its readLine method.**

There is no such class as DataReader. There is a java.io.DataInputStream though,  which lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a java.io.DataOutputStream to write data that can later be read by a java.io.DataInputStream.

**Question 32:**

What can be inserted in the following code at //1 so that it will print the number of lines present in the given file?

public void countLines(String filePath) throws Exception{

//1

System.out.println(ref.count());

}

**Stream<String> ref = new BufferedReader(new FileReader("c:\\temp\\test.txt")).lines();**

The lines() method has been added to java.io.BufferedReader in Java 1.8. It returns Stream<String>.

**Stream<String> ref = new BufferedReader(new FileReader(Paths.get("c:\\temp\\test.txt"))).lines();**

FileReader does not have a constructor that takes a Path object. It has the following three constructors: FileReader(File file) Creates a new FileReader, given the File to read from. FileReader(FileDescriptor fd) Creates a new FileReader, given the FileDescriptor to read from. FileReader(String fileName) Creates a new FileReader, given the name of the file to read from.

**Stream<String> ref = new BufferedReader(new FileReader("c:\\temp\\test.txt")).readLines();**

There is no readLines method in BufferedReader. There is a readLine method but it returns only one line.

**Stream<String> ref = Files.lines("c:\\temp\\test.txt");**

Files.lines(Path path) method expects a Path object as an argument (not a String). The following would have been valid: Stream<String> ref = Files.lines(Paths.get("c:\\temp\\test.txt"));

**Question 33:**

Given the following code (assume appropriate imports):

public class IOTest {

public static void main(String[] args) {

var myfile = Paths.get("test.txt");

try(var bfr = Files.newBufferedReader(myfile, Charset.forName("US-ASCII") )){

String line = null;

while( (line = bfr.readLine()) != null){

System.out.println(line);

}

}catch(Exception e){

System.out.println(e);

}

}

}

What will be printed when this code is run if test.txt doesn't exist?

**java.io.FileNotFoundException: test.txt**

**java.nio.file.FileNotFoundException: test.txt**

**java.nio.file.NoSuchFileException: test.txt**

This exception will be thrown when the program tries to create a BufferedReader to read the file specified by the Path object.

**java.nio.file.InvalidPathException : test.txt**

This exception is thrown when the argument passed while creating Path object is invalid. For example, "c:c:test.txt". In the given code, the path string is valid, so this exception will not be thrown. The existence of the file is not checked at the time of creation of Path object.

**Note:**

Note that java.io.FileNotFoundException may be thrown by FileInputStream, FileOutputStream, and RandomAccessFile constructors if the file by the given name does not exist.

**Question 34:**

In which of the following cases can the Console object be acquired?

**When the JVM is started from an interactive command line with explicitly redirecting the standard input and output streams to Console.**

**When the JVM is started from an interactive command line without redirecting the standard input and output streams.**

**When the JVM is started in the background with the standard input and output streams directed to Console.**

**When the JVM is started in the background without redirecting the standard input and output streams.**

**Note:**

Whether a virtual machine has a console is dependent upon the underlying platform and also upon the manner in which the virtual machine is invoked. If the virtual machine is started from an interactive command line without redirecting the standard input and output streams then its console will exist and will typically be connected to the keyboard and display from which the virtual machine was launched. If the virtual machine is started automatically, for example by a background job scheduler, then it will typically not have a console.

If this virtual machine has a console then it is represented by a unique instance of this class which can be obtained by invoking the System.console() method. If no console device is available then an invocation of that method will return null.

**Question 35:**

What will the following code print?

Path p1 = Paths.get("c:\\temp\\test.txt");

Path p2 = Paths.get("report.pdf");

System.out.println(p1.resolve(p2));

**..\report.pdf**

**c:\temp\test.txt\report.pdf**

If the argument is a relative path (i.e. if it doesn't start with a root), the argument is simply appended to the path to produce the result.

**c:\temp\report.pdf**

**It will throw an exception.**

**Note:**

Please go through the following description of Path.resolve() method as given in JavaDoc API:

public Path resolve(Path other)

Resolve the given path against this path.

If the other parameter is an absolute path then this method trivially returns other. If other is an empty path then this method trivially returns this path. Otherwise this method considers this path to be a directory and resolves the given path against this path. In the simplest case, the given path does not have a root component, in which case this method joins the given path to this path and returns a resulting path that ends with the given path. Where the given path has a root component then resolution is highly implementation dependent and therefore unspecified.

Parameters:

other - the path to resolve against this path

Returns:

the resulting path

**Question 36:**

As a part of an application, you have serialized and stored some objects of a class in the database. At another place in the same application, you deserialize those objects.

After a few months you determine that you need to add one new String field in the class.

Which of the following statements are correct regarding the above described situation?

**The objects serialized earlier cannot be deserialized to the updated class objects.**

**Objects serialized earlier can be deserialized to the updated class objects by adding a new serialVersionUID field with a value of 0 to the updated class.**

**No special change is necessary in the updated class. Objects serialized earlier will be deserialized to the updated class objects but the newly added field will be null.**

**Old serialized objects can be deserialized only if the original class had explicitly defined a serialVersionUID field and if the updated class maintains the same value for that field.**

Every class that implements Serializable should explicitly define a static final serialVersionUID field of type long. For example, public static final long serialVersionUID = 1; If you make changes to the class and if you still want old objects to be successfully deserialized into the updated class objects, you should keep the same value for serialVersionUID. However, this is not a must.

**It is possible to deserialize the older objects into the update class objects even if the original class did not explicitly define the serialVersionUID field.**

If a class that implements Serializable does not explicitly define serialVersionUID field, the compiler automatically adds this field. It assignes this field a value that is computed based on the attributes of the class such as the fields and the implemented interfaces. It is possible to determine this value by various means. For example, by using the serialver tool provided by the JDK: serialver MyClass Or by using ObjectInputStream.readClassDescriptor() method. Once you get this number, you can assign the same number to serialVersionUID in your updated class.

**Question 37:**

Given that the file test.txt is accessible and contains multiple lines, which of the following code fragments will correctly print all the lines from the file?

**Stream<String> lines = Files.find(Paths.get("test.txt"));**

**lines.forEach(System.out::println);**

Files.find method returns a stream of Path objects. The following is the correct signature for the find method. public static Stream<Path> find(Path start, int maxDepth, BiPredicate<Path,BasicFileAttributes> matcher, FileVisitOption... options) throws IOException Return a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.

**BufferedReader bfr = new BufferedReader(new FileReader("test.txt"));**

**System.out.println(bfr.readLines());**

There is no readLines method in BufferedReader. There is a readLine method but it returns only one line. Here are a couple of correct ways to print lines using a BufferedReader:

while(bfr.ready()){

System.out.println(bfr.readLine());

}

String line = null;

while( (line = bfr.readLine()) != null){

System.out.println(line);

}

**Stream<String> lines = Files.list(Paths.get("test.txt"));**

**lines.forEach(x->System.out.println(x));**

The usage of Files.list method is correct but it returns a stream of Path objects for files contained in a directory. Unlike the Files.find method, the list method doesn't search for files. It just returns all the files in a given directory. If you change the code to Stream<Path> lines = Files.list(Paths.get("c:\\temp\\test.txt"));, it will compile but will throw java.nio.file.NotDirectoryException because test.txt is not a directory.

**Stream<String> lines = Files.lines(Paths.get("test.txt"));**

**lines.forEach(System.out::println);**

**Stream<String> lines = Files.lines(Paths.get("test.txt"), Charset.defaultCharset());**

**lines.forEach(s -> System.out.println(s));**

**Question 38:**

What will the following code fragment print?

Path p1 = Paths.get("c:\\personal\\.\\photos\\..\\readme.txt");

Path p2 = p1.normalize();

System.out.println(p2);

**readme.txt**

**c:\personal\photos\readme.txt**

c:\personal\readme.txt

Notice that . is always redundant and is removed by itself, while .. and the preceding directory cancel each other out because .. means parent directory. For example, a/b/.. is same as a.

**c:\photos\readme.txt**

**Note:**

The following is the complete JavaDoc API description of this method:

public Path normalize()

Returns a path that is this path with redundant name elements eliminated.

The precise definition of this method is implementation dependent but in general it derives from this path, a path that does not contain redundant name elements. In many file systems, the "." and ".." are special names used to indicate the current directory and parent directory. In such file systems all occurrences of "." are considered redundant. If a ".." is preceded by a non-".." name then both names are considered redundant (the process to identify such names is repeated until is it no longer applicable).

This method does not access the file system; the path may not locate a file that exists. Eliminating ".." and a preceding name from a path may result in the path that locates a different file than the original path. This can arise when the preceding name is a symbolic link.

Returns:

the resulting path or this path if it does not contain redundant name elements; an empty path is returned if this path does have a root component and all name elements are redundant

**Question 39:**

Given:

String INPUT\_FILE = "c:\\temp\\src\\foo.bar\\module-info.java";

Assuming the file exists, which of the following options will print the contents of the file?

**Files.lines(INPUT\_FILE).forEach(System.out::println);**

**Stream<String> lines = Files.lines(Paths.get(INPUT\_FILE));**

**lines.forEach(System.out::println);**

**Stream<String> lines = Files.readAllLines(Paths.get(INPUT\_FILE));**

**lines.forEach(System.out::println);**

Files.readAllLines returns List<String>.

**List<String> lines = Files.readAllLines(Paths.get(INPUT\_FILE));**

**lines.forEach(System.out::println);**

**List<String> lines = Files.lines(Paths.get(INPUT\_FILE));**

**lines.forEach(System.out::println);**

Files.lines returns Stream<String>.

**String[] stra = Files.readLines(Paths.get(INPUT\_FILE));**

**for(String s: stra) System.out.println(s);**

There is a Files.readString(Path ) method that reads the whole file into a String. But there is no method that returns a String[].

**Question 40:**

Given the following code fragment:

var raf = new RandomAccessFile("c:\\temp\\test.txt", "rwd");

//INSERT CODE HERE

raf.close();

var dis = new DataInputStream(new FileInputStream("c:\\temp\\test.txt"));

String value = dis.readUTF();

System.out.print(value);

dis.close();

Which of the following options can be inserted in the above code so that it will print hello world?

**raf.writeString("hello world");**

There is no writeString method in RandomAccessFile.

**raf.writeChars("hello world");**

Although writeChars(String ) is a valid method in RandomAccessFile, it is not suitable here because you want to read the contents in UTF format later. writeChars will write the String is default encoding and if you try to read it as UTF, it will throw an exception while reading.

**raf.writeUTF("hello world");**

Remember that RandomAccessFile implements DataInput as well as DataOutput interfaces. Therefore, in this case, you can use raf as an instance of DataOutput and call its writeUTF(String) method.

raf.writeData("hello world");

There is no writeData method in RandomAccessFile.

**Question 41:**

Consider the following code:

public static void findFiles() throws Exception{

Path dir = Paths.get("c:\\temp");

//INSERT CODE HERE

for(Path p : ds){

System.out.println(p);

}

}

catch(Exception e){

e.printStackTrace();

}

}

What should be inserted in the above code so that it will print all the files with extension gif and jpeg?

**try{ DirectoryStream<Path> ds = Files.newDirectoryStream(dir, "\*.[gif,jpeg]");**

**try{ DirectoryStream<Path> ds = Files.newDirectoryStream(dir, "\*.{gif,jpeg}");**

**try{ DirectoryStream<Path> ds = Files.newDirectoryStream(dir, "\*.gif,\*.jpeg");**

**try{ DirectoryStream<Path> ds = Files.newDirectoryStream(dir, "gif,jpeg");**

**Note:**

Files.newDirectoryStream(Path dir, String globPattern)

Opens a directory, returning a DirectoryStream to iterate over the entries in the directory. The elements returned by the directory stream's iterator are of type Path, each one representing an entry in the directory. The Path objects are obtained as if by resolving the name of the directory entry against dir. The entries returned by the iterator are filtered by matching the String representation of their file names against the given globbing pattern.

There are questions in the exam that expect you to know basic syntax for the glob pattern. The following description from JavaDoc API is sufficient :-

String representation of the path is matched using a limited pattern language that resembles regular expressions but with a simpler syntax.

For example:

\*.java : Matches a path that represents a file name ending in .java

\*.\* : Matches file names containing a dot

\*.{java,class} : Matches file names ending with .java or .class

foo.? : Matches file names starting with foo. and a single character extension

/home/\*/\* : Matches /home/gus/data on UNIX platforms

/home/\*\* : Matches /home/gus and /home/gus/data on UNIX platforms

C:\\\* : Matches C:\foo and C:\bar on the Windows platform (note that the backslash is escaped; as a string literal in the Java Language the pattern would be "C:\\\\\*")

The following rules are used to interpret glob patterns:

The \* character matches zero or more characters of a name component without crossing directory boundaries.

The \*\* characters matches zero or more characters crossing directory boundaries.

The ? character matches exactly one character of a name component.

The backslash character (\) is used to escape characters that would otherwise be interpreted as special characters. The expression \\ matches a single backslash and "\{" matches a left brace for example.

The [ ] characters are a bracket expression that match a single character of a name component out of a set of characters. For example, [abc] matches "a", "b", or "c". The hyphen (-) may be used to specify a range so [a-z] specifies a range that matches from "a" to "z" (inclusive). These forms can be mixed so [abce-g] matches "a", "b", "c", "e", "f" or "g". If the character after the [ is a ! then it is used for negation so [!a-c] matches any character except "a", "b", or "c".

Within a bracket expression the \*, ? and \ characters match themselves. The (-) character matches itself if it is the first character within the brackets, or the first character after the ! if negating.

The { } characters are a group of subpatterns, where the group matches if any subpattern in the group matches. The "," character is used to separate the subpatterns. Groups cannot be nested.

Leading period/dot characters in file name are treated as regular characters in match operations. For example, the "\*" glob pattern matches file name ".login". The Files.isHidden method may be used to test whether a file is considered hidden.

All other characters match themselves in an implementation dependent manner. This includes characters representing any name-separators.

The matching of root components is highly implementation-dependent and is not specified.

**Question 42:**

Consider the following class:

public class Student implements Serializable{

public static final long serialVersionUID = 1;

public String name;

public String grade;

public String toString(){ return "["+name+", "+grade+"]"; }

}

An object of this class was created as follows and was serialized to a file c:\temp\bob.ser:

Student s = new Student();

s.name = "bob";

s.grade = "10";

After some time the Student class was changed as follows:

public class Student implements Serializable{

public static final long serialVersionUID = 1;

public String id="S111";

public String name;

public String grade;

public int age=15;

public String toString(){ return "["+id+", "+name+", "+grade+", "+age+"]"; }

}

Now, the serialized file is read back as follows:

FileInputStream fis = new FileInputStream("c:\\temp\\bob.ser");

ObjectInputStream is = new ObjectInputStream(fis);

s = (Student) is.readObject();

is.close();

System.out.println("Loaded "+s);

What will it print?

**It will throw an exception while deserializing the file.**

**Loaded [null, bob, 10, 0]**

Since the serialVersionUID of the serialized class and the new class are same, the file will be deserialized without any issue. The new fields will be initialized to their Java defaults (because constructors and initializers are not invoked during deserialization). So the values for id and and age will remain null and 0 respectively.

**Loaded [S111, bob, 10, 15]**

**It will have unpredicable values for id and age.**

**Note:**

1. When a file is deserialized into an object, the class's constructor and instance initializers are not called. So the fields for which no value is available in the serialized file, are initialized to their default values (i.e. number fields to 0, boolean to false, and references to null). Note that the fields of a non-serializable super class are initialized by the instance initializers and the no-args constructor of that super class.  
  
2. serialVersionUID denotes the version number of the class. If you don't specify serialVersionUID for a class that implements Serializable, Java compiler automatically adds this field. It computes a value based on the attributes of the class such as the fields and interfaces, and assigns that value to serialVersionUID.  
  
It is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an InvalidClassException.  
  
3. If the serialVersionUID for the serialized object and the actual class is same then the deserialization completes successfully but any additional fields that are not present in the serialized file are initialized to their default value (as per point 1. above).  Any fields that are missing in the class but are present in the serialized file are ignored.

**Question 43:**

Consider the following code:

class Bond

{

String ticker; double coupon; java.time.LocalDate maturity;

}

class Portfolio implements Serializable

{

String accountName;

Bond[] bonds;

}

public class TestClass {

public static void main(String[] args) throws Exception{

Portfolio portfolio = // get portfolio somehow

// serialize portfolio

}

}

Which of the following approaches can be taken independent of each other so that a Portfolio object can be serialized while preserving the state of the Bond objects contained in Portfolio?

**It can be serialized as it is without any modification.**

**Just have Bond class implement Serializable.**

**Just make 'bonds' field in Portfolio transient.**

Making it transient will leave 'bonds' unserialized. So, its state will be lost when Portfolio is serialized.

**Change the type of bonds from Bond[] to ArrayList<Bond> bonds;**

It will not have any bearing on the fact that Bond is not Serializable.

**Make bonds array transient in Portfolio and implement readObject(ObjectInputStream os)  and writeObject(ObjectOutputStream os)  methods to read and write the state of Bond objects explicitly.**

**Note:**

If for any reason, you want to serialize Portfolio objects without making Bond class Serializable, you can customize the serialization of a Portfolio class by implementing readObject and writeObject methods as shown below:

class Bond // does not implement Serializable

{

String ticker = "bac"; double coupon = 8.3; java.time.LocalDate maturity = new Date();

}

class Portfolio implements Serializable

{

String accountName;

transient Bond[] bonds = new Bond[]{ }; // must be transient because Bond class does not implement Serializable

private void writeObject(ObjectOutputStream os) throws Exception{

os.defaultWriteObject();

os.writeInt(bonds.length);

//write the state of bond objects

for(int i=0; i<bonds.length; i++) {

os.writeObject(bonds[i].ticker);

os.writeDouble(bonds[i].coupon);

os.writeObject(bonds[i].maturity);

}

}

private void readObject(ObjectInputStream os) throws Exception{

os.defaultReadObject();

int n = os.readInt();

this.bonds = new Bond[n];

//read the state of bond objects.

for(int i=0; i<bonds.length; i++) {

bonds[i] = new Bond();

bonds[i].ticker = (String) os.readObject();

bonds[i].coupon = os.readDouble();

bonds[i].maturity = (java.util.Date) os.readObject();

}

}

}

**Question 44:**

Given:

public static void copy1(Path p1, Path p2) throws Exception {

Files.copy(p1, p2, StandardCopyOption.REPLACE\_EXISTING);

}

Identify correct statements.

**An exception will be thrown at runtime if p2 is a symbolic link.**

**If p2 is a symbolic link, then that link, and not the target of that link, will be replaced .**

**If p1 is a symbolic link, then the final target of the link is copied to p2.**

**An exception will be thrown at runtime if either of p1 or p2 is a symbolic link.**

**Result is OS dependent if either of p1 or p2 is a symbolic link.**

**Note:**

public static Path copy(Path source, Path target, CopyOption... options) throws IOException  
  
Copy a file to a target file.  
This method copies a file to the target file with the options parameter specifying how the copy is performed. By default, the copy fails if the target file already exists or is a symbolic link, except if the source and target are the same file, in which case the method completes without copying the file. File attributes are not required to be copied to the target file. If symbolic links are supported, and the file is a symbolic link, then the final target of the link is copied. If the file is a directory then it creates an empty directory in the target location (entries in the directory are not copied). This method can be used with the walkFileTree method to copy a directory and all entries in the directory, or an entire file-tree where required.  
  
The options parameter may include any of the following:  
  
REPLACE\_EXISTING:  If the target file exists, then the target file is replaced if it is not a non-empty directory. If the target file exists and is a symbolic link, then the symbolic link itself, not the target of the link, is replaced.  
  
COPY\_ATTRIBUTES:  Attempts to copy the file attributes associated with this file to the target file. The exact file attributes that are copied is platform and file system dependent and therefore unspecified. Minimally, the last-modified-time is copied to the target file if supported by both the source and target file stores. Copying of file timestamps may result in precision loss.  
  
NOFOLLOW\_LINKS:  Symbolic links are not followed. If the file is a symbolic link, then the symbolic link itself, not the target of the link, is copied. It is implementation specific if file attributes can be copied to the new link. In other words, the COPY\_ATTRIBUTES option may be ignored when copying a symbolic link.  
An implementation of this interface may support additional implementation specific options.  
  
Copying a file is not an atomic operation. If an IOException is thrown, then it is possible that the target file is incomplete or some of its file attributes have not been copied from the source file. When the REPLACE\_EXISTING option is specified and the target file exists, then the target file is replaced. The check for the existence of the file and the creation of the new file may not be atomic with respect to other file system activities.

**Question 45:**

A programmer is writing a small component that processes a file line by line. The following is the code :

public class LineByLineProcessor {

public void processLines(String fullFilePath) throws Exception

{

// declare and initialize "handle" here

String str = null;

while( (str = handle.readLine()) != null)

{

System.out.println("Processing line : "+str);

}

handle.close();

}

}

Which of the given options will declare and initialize handle appropriately?

**Reader handle = new FileReader(fullFilePath);**

Reader does not have high level methods such as readLine().

**BufferedReader handle = new BufferedReader(fullFilePath);**

A BufferedReader can only be created using a Reader such as FileReader. It cannot directly operate on a file.

**BufferedReader handle = new BufferedReader(new File(fullFilePath));**

A BufferedReader can only be created using a Reader such as FileReader. It cannot directly operate on a file.

**BufferedReader handle = new BufferedReader(new FileReader(fullFilePath));**

This is a correct way to create a BufferedReader.

**BufferedReader handle = new BufferedReader(new FileReader( new File(fullFilePath)));**

FileReader(String) and FileReader(File), both are valid ways to create a FileReader.

**Note:**

A Reader such as a FileReader provides only low level operations such as reading a single character or array of characters. It does not understand the notion of "lines".

BufferedReader "decorates" Reader to provide higher level method readLine() by buffering characters. It is an efficient way of reading characters, character arrays, and lines.

The same relationship exists between FileWriter and BufferedWriter but for writing.

**Question 46:**

You have a file named customers.dat in c:\company\records directory. You want to copy all the lines in this file to another file named clients.dat in the same directory and you have the following code to do it:

public static void writeData() {

Path p1 = Paths.get("c:\\company\\records\\customers.dat");

//LINE 20 - INSERT CODE HERE

try (

var br = new BufferedReader(new FileReader(p1.toFile()));

var bw = new BufferedWriter(new FileWriter(p2.toFile()))) {

String line = null;

while ((line = br.readLine()) != null) {

bw.write(line);

bw.newLine();

}

}catch(Exception e){

e.printStackTrace();

}

}

Which of the following options can be inserted independent of each other at //LINE 20 to make it work?

Assume that the current directory for the program when it runs is c:\code.

**Path p2 = p1.resolveSibling("\\clients.dat");**

This will set p2 to c:\clients.dat, which is not what you want.

**Path p2 = p1.resolveSibling("clients.dat");**

You already have the absolute path to customers.dat in p1. Further, it is given that you want to copy the data to a file in the same directory i.e. both the files - old and new, are siblings. So, to open clients.dat, you need to determine the absolute path for clients.dat using the absolute path for customers.dat. In other words, you are trying to get the absolute path for a file that exists in the same directory as the original file. The method resolveSibling is meant exactly for this purpose.

**Path p2 = p1.relativize("clients.dat");**

**Path p2 = Paths.get("c:", p1.subpath(0, 2).toString(), "clients.dat");**

This is very straight forward. You should go through the JavaDoc API description for Path.subpath and Paths.get methods.

**Path p2 = Paths.get("c:", p1.subpath(1, 2).toString(), "clients.dat");**

p1.subpath(1, 2) will return "records", so the full path will be "c:\records\clients.dat", which is not what you want.

**Question 47:**

What will the following code print?

Path p1 = Paths.get("\\photos\\vacation");

Path p2 = Paths.get("\\yellowstone");

System.out.println(p1.resolve(p2)+" "+p1.relativize(p2));

**yellowstone  ..\..\yellowstone**

**\yellowstone  ..\..\yellowstone**

1. Since the argument to resolve starts with \\, the result will be the same as the argument. If the argument doesn't start with a \ and it doesn't start with a root such as c:, the output is the result on appending the argument to the path on which the method is invoked. 2. To arrive at \\yellowstone from \\photos\\vacation, you have to first go two directories up and then down to yellowstone. Therefore, p1.relativize(p2) will be ..\..\yellowstone

**\yellowstone  \yellowstone**

**\yellowstone  Yellowstone**

**Note:**

Please go through the following description of Path.resolve() method as given in JavaDoc API:

public Path resolve(Path other)

Resolve the given path against this path.

If the other parameter is an absolute path then this method trivially returns other. If other is an empty path then this method trivially returns this path. Otherwise this method considers this path to be a directory and resolves the given path against this path. In the simplest case, the given path does not have a root component, in which case this method joins the given path to this path and returns a resulting path that ends with the given path. Where the given path has a root component then resolution is highly implementation dependent and therefore unspecified.

Parameters:

other - the path to resolve against this path

Returns:

the resulting path

**Question 48:**

What will the following code fragment print?

Path p1 = Paths.get("\\personal\\readme.txt");

Path p2 = Paths.get("\\index.html");

Path p3 = p1.relativize(p2);

System.out.println(p3);

**\index.html**

Observe what happens when you append this path to p1:

 \personal\readme.txt + \index.html =>\personal\readme.txt\index.html

This is not same as \index.html

**\personal\index.html**

Observe what happens when you append this path to p1:

\personal\readme.txt + \personal\index.html =>\personal\readme.txt\\personal\index.html

This is not same as \index.html

**personal\index.html**

Observe what happens when you append this path to p1:

\personal\readme.txt + personal\index.html =>\personal\readme.txt\personal\index.html

This is not same as \index.html

**..\..\index.html**

Observe that if you append this path to p1, you will get p2. Therefore, this is the right answer.

p1 + ..\..\index.html =>\personal\readme.txt + ..\..\index.html =>\personal + ..\index.html =>\index.html

A ".." implies parent folder, therefore imagine that you are taking off one ".." from the right side of the plus sign and removing the last name of the path on the left side of the plus sign.

For example, .. appended to personal makes it personal\.., which cancels out.

**Question 49:**

Consider the following code:

var s = "hello";

byte i = 100;

var fos = new FileOutputStream("c:\\temp\\data.bin");

var dos = new DataOutputStream(fos);

//WRITE s to file

//WRITE i to file

dos.flush(); dos.close(); fos.close();

var dis = new DataInputStream(new FileInputStream("c:\\temp\\data.bin"));

//READ s from file

//READ i from file

Which methods should be used to write and read s and i to/from the data.bin file?

**writeString, writeByte and readString, readByte**

write/readString are not a valid methods in Data[Output|Input]Stream class. If you need to write and read Strings, you should use writeUTF and readUTF.

**writeString, writeInt and readString, readInt**

**writeChars, writeByte and readChars, readByte**

There is a writeChars(String ) method in DataOutputStream but there is no readChars in DataInputStream

**writeUTF, writeByte and readUTF, readByte**

**writeUTF, writeInt and readUTF, readInt**

While this will work (a byte can always be put in an int) but if you want to read/write a byte, you should used read/writeByte method. int will take more space in the file than a byte.

Since the question says, "which methods should be used", this is an incorrect option. Had it asked "can" instead of "should", this option would have been correct.

**Question 50:**

Given:

Path p = Paths.get("c:\\temp\\out");

try{

var b = Files.deleteIfExists(p);

System.out.println(b);

}catch(Exception e){

e.printStackTrace();

}

Identify correct statements.

**It will print "c:\temp\out" if the file referred to by p is not deleted for any reason.**

The Files.deleteIfExists method returns a boolean - true if the file is deleted and false otherwise.

**It will print an exception stack trace if p refers to a directory instead of a file.**

It can delete a directory also. However, a java.nio.file.DirectoryNotEmptyException will be thrown if the directory is not empty.

**It will print an exception stack trace if p refers to an empty file or a non-empty directory.**

Size of the file is immaterial. It can delete an empty file also. But not a non-empty directory.

**It will print an exception stack trace the file referred to by p cannot be deleted due to lack of appropriate file permissions.**

It will not throw any exception in that case. It will just return false.

**It will print true if p refers to an empty directory.**

public static boolean deleteIfExists(Path path) throws IOException

Deletes a file if it exists.

As with the delete(Path) method, an implementation may need to examine the file to determine if the file is a directory. Consequently this method may not be atomic with respect to other file system operations. If the file is a symbolic link, then the symbolic link itself, not the final target of the link, is deleted.

If the file is a directory then the directory must be empty. In some implementations a directory has entries for special files or links that are created when the directory is created. In such implementations a directory is considered empty when only the special entries exist.

On some operating systems it may not be possible to remove a file when it is open and in use by this Java virtual machine or other programs.

Parameters:

path - the path to the file to delete

Returns: true if the file was deleted by this method; false if the file could not be deleted because it did not exist

Throws:

DirectoryNotEmptyException - if the file is a directory and could not otherwise be deleted because the directory is not empty (optional specific exception)

IOException - if an I/O error occurs

SecurityException - In the case of the default provider, and a security manager is installed, the SecurityManager.checkDelete(String) method is invoked to check delete access to the file.

**Question 51:**

Given the code fragment:

class Classes implements Serializable{

String id;

}

class Person{

String name;

transient String address;

}

class Student extends Person implements Serializable{

String studentNo;

Classes classes=new Classes();

}

Which fields are serialized in a Student object?

A. studentNo and classes

B. studentNo and name

C. studentNo, classes and name

D. studentNo, classes, name, and address

Answer: A

**Question 52:**

Given:

Path p1=Paths.get("/scratch/exam/topsecret/answers");

Path p2=Paths.get("/scratch/exam/answers/temp.txt");

Path p3=Paths.get("/scratch/answers/topsecret");

Which two statements print ..\..\..\answers\topsecret? (Choose two.)

A. System.out.print(p3.relativize(p1));

B. System.out.print(p2.relativize(p3));

C. System.out.print(p1.relativize(p3));

D. System.out.print(p3.relativize(p2));

E. System.out.print(p1.relativize(p2));

F. System.out.print(p2.relativize(p1));

Answer: B,C

**Question 53:**

Given the code fragment:

public class Main {

public static void main(String[] args) {

try {

Path path=Paths.get("/u01/work");

// line 1

System.out.println(attributes.isDirectory());

} catch (IOException e) {

e.printStackTrace();

}

You want to examine whether path is a directory. Which code inserted on line 1 will accomplish this?

A. BasicFileAttributes attributes = Files isDirectory (path);

B. BasicFileAttributes attributes =Files.getAttribute (path, ‘’insdirectory’’);

C. BasicFileAttributes attributes = Files.readAttributes(path, BasicFileAttributes.class);

D. BasicFileAttributes attributes = Files.readAttributes (path, FileAttributes.class);

Answer: C

**Question 54:**

Given:

class MyPersistenceDate{

String str;

private void methodA(){

System.out.println(“methodA”);

}

}

You want to implement the java. io.Serializable interface to the MypersistenceData class.

Which method should be overriden?

A. The readExternal and writeExternal method

B. The readExternal method

C. The writeExternal method

D. nothing

Answer: D

**Question 55:**

Given:

public class Main {

private String[] strings = { "ABCDEFGHIJKLMNOPQRSTUVWXYZ", "abcdefghijklmnopqrstuvwxyz", "0123456789" };

public void write(String filename) {

// line 1

for (String str : strings) {

ByteBuffer buffer = ByteBuffer.wrap(str.getBytes());

fileChannel.write(buffer);

}

} catch (IOException e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

Main test=new Main();

test.write(“file\_to\_path”);

}

}

You want to obtain the Filechannel object on line 1. Which code fragment will accomplish this?

A.

try(FileChannel fileChannel=Channels.newChannel(new FileOutputStream(filename))){

B.

try (FileChannel fileChannel = new FileOutputStream(filename).getChannel()) {

C.

try (FileChannel fileChannel = new FileOutputStream(new FileChannel(filename))) {

D.

try(FileChannel fileChannel=new FileChannel(new FileOutputStream(filename))){

Answer: B

**Question 56:**

Given the code fragment:

Path currentFile=Paths.get("/scratch/exam/temp.txt");

Path outputFile=Paths.get("/scratch/exam/new.txt");

Path directory=Paths.get("/scratch/");

Files.copy(currentFile, outputFile);

Files.copy(outputFile, directory);

Files.delete(outputFile);

The /scratch/exam/temp.txt file exists. The /scratch/exam/new.txt and /scratch/new.txt files do not exist.

What is the result?

A. /scratch/exam/new.txt and /scratch/new.txt are deleted.

B. The program throws a FileAlreadyExistsException.

C. The program throws a NoSuchFileException.

D. A copy of /scratch/exam/new.txt exists in the /scratch directory and /scratch/exam/new.txt is deleted.

Answer: C

**Question 57:**

Given:

public class Main {

public static void main(String[] args) {

try {

Path path=Paths.get("/u01/work/filestore.txt");

boolean result=Files.deleteIfExists(path);

if(result) System.out.println(path+" is deleted.");

else System.out.println(path+" is not deleted.");

} catch (IOException e) {

System.out.println("Exception");

}

}

}

Assume the file on path does not exist. What is the result?

A. The compilation fails.

B. /u01/work/filestore.txt is not deleted.

C. Exception

D. /u01/work/filestore.txt is deleted.

Answer: B

**Question 58:**

Given:

public class Main {

public static void main(String[] args) {

try(BufferedReader in=new BufferedReader(new InputStreamReader(System.in))){

System.out.print("Input: ");

String input=in.readLine();

System.out.print("Echo: "+input);

}catch (IOException e) {

e.printStackTrace();

}

}

}

And the command: java Main Helloworld What is the result ?

A. Input: Echo:

B. Input: Helloworld Echo: Helloworld

C. Input:Then block until any input comes from System.in.

D. Input:Echo: Helloworld

E. A NullPointerException is thrown at run time.

Answer: C

**Question 59:**

Given:

public static void main(String[] args){

try(Reader reader1=new FileReader(“File1.txt”);

Reader reader2=new FileReader(“File2.txt”);

Reader reader3=new FileReader(“File3.txt”)){

}catch(IOException ex){

Logger.getLogger(Main.class.getName()).log(Level.SEVERE,null,ex);

}

// line 1

System.out.println(“Done”);

}

When run and all three files exist, what is the state of each reader on Line 1?

A. All three readers are still open.

B. All three readers have been closed.

C. The compilation fails.

D. Only reader1 has been closed.

Answer: B

**Question 60:**

Given:

String originalPath = “data\\projects\\a-project\\..\\..\\another-project”;

Path path = Paths.get(originalPath);

System.out.print(path.normalize());

What is the result?

A. data\another-project

B. data\projects\a-project\another-project

C. data\\projects\\a-project\\..\\..\\another-project

D. data\projects\a-project\..\..\another-project

Answer: A

**Question 61:**

Given:

public class SerializedMessage implements Serializable{

String message;

LocalDatetime createdTime;

transient LocalDateTime updatedDateTime;

SerializedMessage(String message){

this.message=message;

this.createdTime=LocalDateTime.now();

}

private void readObject(ObjectInputStream in){

try{

in.defaultReadObject();

this.updatedTime=LocalDateTime.now();

}catch(IOException | ClassNotFoundException e){

e.printStackTrace();

}

}

}

When is the readObject method called?

A. before this object is deserialized

B. after this object is deserialized

C. before this object Is serialized

D. The method is never called.

E. after this object is serialized

Answer: B

**Question 62:**

Given the code fragment:

Path source = Paths.get(“/repo/a/a.txt”);

Path destination = Paths.get(“/repo”);

Files.move(source, destination); // line 1

Files.delete (source); // line 2

Assuming the source file and destination folder exist, what Is the result?

A. A java.nio.file.FileAlreadyExistsException is thrown on line 1.

B. A java.nio.file.NoSuchFileException is thrown on line 2.

C. A copy of /repo/a/a.txt is moved to the /repo directory and /repo/a/a.txt is deleted.

D. a.txt is renamed repo.

Answer: A

**Question 63:**

Given:

public class Main {

public static void main(String[] args) {

try(BufferedReader br=new BufferedReader(new java.io.InputStreamReader(System.in))){

String input=br.readLine();

System.out.println("Input String was: "+input);

}catch (IOException e) {

e.printStackTrace();

}

}

}

Which is true?

A. System.out is the standard output stream

B. The stream is open only when System.out is called.

C. System.in cannot reassign the other stream.

D. System.out is an instance of java.io.OutputStream by default.

E. System.in is the standard input stream

F. The stream is already open.

Answer: ADE

**Question 64:**

Given:

try {

// line 1

lines.map(l->l.toUpperCase())

.forEach(line->{

try {

Files.write(Paths.get("files/copyabc.txt"),

line.getBytes(),StandardOpenOption.CREATE);

} catch (IOException e) {

e.printStackTrace();

}

});

} catch (Exception e) {

e.printStackTrace();

}

You want to obtain the Stream object on reading the file. Which code inserted on line 1 will accomplish this?

A. var lines = Files.lines(Paths.get(INPUT\_FILE\_NAME));

B. Stream lines = Files.readAllLines(Paths.get(INPUT\_FILE\_NAME));

C. var lines = Files.readAllLines(Paths.get(INPUT\_FILE\_NAME));

D. Stream<String> lines = Files.lines(INPUT\_FILE\_NAME);

Answer: A

**Question 65:**

Given:

public class Main {

public static void checkConfiguration(String fileName) {

File file=new File(fileName);

if(!file.exists()) {

throw new Error("Fatal Error: Configuration File, "+fileName+", is missing.");

}

}

public static void main(String[] args) {

checkConfiguration("App.config");

System.out.println("Configuration is OK");

}

}

If file "App.config" is not found, what is the result?

A. Configuration is OK

B. The compilation fails.

C. Exception in thread "main" java.lang.Error:Fatal Error: Configuration File, App.config, is missing.

D. nothing

Answer: C