Mouath Sami Masalmah

No: 1220179

Date: 16/12/2023

“Solution for Pre\_Lap 5”

Dr: Fadi khalil

Q1:  
Checked exceptions are exceptions that must be either caught or declared by the method using the "throws" keyword. Unchecked exceptions, on the other hand, do not require explicit handling or declaration.

Q2:

To declare an exception in Java, you use the "throws" clause in the method signature. For example:

**public void myMethod() throws SomeException {**

**// method code**

**}**

Here, ‘**SomeException’** is the declared exception, indicating that the method may throw this specific exception, and the calling code needs to handle it or declare it as well.

Q3:

Yes, you can declare multiple exceptions in a method header by separating them with commas. For example:

**public void myMethod() throws ExceptionType1, ExceptionType2 {**

**// method code**

**}**

In this case, ‘**myMethod’** is declared to throw both ‘**ExceptionType1’** and ‘**ExceptionType2’**, and the calling code needs to handle or declare these exceptions accordingly.

Q4:  
In Java, you can throw an exception using the **throw** keyword, followed by an instance of the exception class. For example:

**public void myMethod() {**

**if (someCondition) {**

**throw new MyException("This is a custom exception message");**

**} // rest of the method code**

**}**

In this example, if the ‘**someCondition’** is true, the ‘**MyException’** is thrown with a custom message. The calling code or a higher-level exception handler needs to handle this exception appropriately.

Q5:  
The keyword ‘**throw’** is used to explicitly throw an exception in Java. It is followed by an instance of an exception class.

The keyword ‘**throws’** is used in a method signature to declare that the method may throw one or more types of exceptions. It specifies the exceptions that the method might throw, and the calling code must handle or declare these exceptions.

Q6:

To define a custom exception class in Java:

1. Create a new class that extends either ‘**Exception’** for checked exceptions or ‘**RuntimeException’** for unchecked exceptions.
2. Optionally, add constructors to initialize the exception with a message or other parameters.

Example:

**public class MyException extends Exception {**

**public MyException() {**

**super();**

**}**

**public MyException(String message) {**

**super(message);**

**}**

**}**

Now, you can use ‘**MyException’** in your code by throwing or catching it as needed.

Q7:  
Text I/O deals with character data and typically involves reading or writing data as human-readable text. It handles characters, encodings, and newline characters.

Binary I/O deals with raw binary data, treating it as a sequence of bytes. It doesn't interpret the data and is used for non-text files, such as images or executables.

In text I/O, newline characters may be automatically converted based on the platform, while binary I/O preserves the exact byte values. Text I/O is more human-readable but may not be suitable for all data types. Binary I/O is more versatile but not human-readable.

Q8:

Always closing streams is crucial to release system resources and prevent resource leaks. It ensures proper cleanup and avoids potential issues like file locks or data corruption.

To close streams in Java, you use the **close()** method. It's typically called in a **finally** block to ensure execution, even if an exception occurs. Example:

**BufferedReader reader = null;**

**try {**

**reader = new BufferedReader(new FileReader("example.txt"));**

**// read or write operations**

**} catch (IOException e) {**

**// handle exception**

**} finally {**

**try {**

**if (reader != null) { reader.close();**

**} } catch (IOException e) {**

**// handle exception**

**} }**

Java 7 introduced the try-with-resources statement, which automatically closes resources like streams when they go out of scope:

**try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))) {**

**// read or write operations**

**} catch (IOException e) {**

**// handle exception**

**}**

Q9:

To check the end of a file in an ‘**InputStream’** (including ‘**FileInputStream’** and ‘**DataInputStream’**), you can use the ‘**read’** method, which returns **-1** when the end of the stream is reached. Example:

**FileInputStream fileInputStream = null;**

**try {**

**fileInputStream = new FileInputStream("example.txt");**

**int data;**

**while ((data = fileInputStream.read()) != -1) {**

**// process the data**

**} } catch (IOException e) {**

**// handle exception**

**} finally {**

**try {**

**if (fileInputStream != null) {**

**fileInputStream.close();**

**} } catch (IOException e) {**

**// handle exception } }**

Remember to close the stream appropriately in a **finally** block or, preferably, use try-with-resources.

Top of Form