JAVA DEVELOPER INTERNSHIP BY ELEVATE LABS

Interview Questions:

1.Difference between FileReader and BufferedReader?

2.What is try-with-resources?

3.How to handle IOException?

4.What are checked and unchecked exceptions?

5.How does file writing work in Java?

6.What is the difference between append and overwrite mode?

7.What is exception propagation?

8.How to log exceptions?

9.What is a stack trace?

10.When to use finally block?

1. Difference between FileReader and BufferedReader?

The key difference is **speed**.

- **FileReader** reads a file one character at a time.
- **BufferedReader** is a wrapper that reads large chunks of the file into memory (a "buffer") and serves data from there, making it much faster.

Real-Life Example: Imagine reading a long book. FileReader is like reading one single letter at a time (V-e-r-y s-l-o-w). BufferedReader is like reading whole sentences or paragraphs at a time, which is how people normally read.

You almost always wrap a FileReader in a BufferedReader for better performance.

Code Example:

Java

```
// BufferedReader wraps FileReader for efficient reading
try (BufferedReader reader = new BufferedReader(new FileReader("notes.txt"))) {
   String line;
   // .readLine() is a helpful method from BufferedReader
   while ((line = reader.readLine()) != null) {
      System.out.println(line);
   }
} catch (IOException e) {
   System.err.println("Error reading file: " + e.getMessage());
}
```

2. What is try-with-resources?

It's a modern way to handle resources (like files) that **automatically closes them** for you. This prevents "resource leaks," which can cause bugs. Before this, you had to manually close resources in a finally block.

Real-Life Example: It’s like a self-closing door. You walk through it (try the code), and it automatically shuts behind you. The old way was like having to remember to turn around and manually close the door (finally) every time you went through.

Code Example:

Java

```
// The resource (reader) is declared in the parentheses and is closed automatically.
try (BufferedReader reader = new BufferedReader(new FileReader("notes.txt"))) {
    System.out.println(reader.readLine());
} catch (IOException e) {
    System.err.println("An error occurred: " + e.getMessage());
}
// No need for a 'finally' block to close the reader!
```

3. How to handle IOException?

IOException is an error that can happen during any Input/Output operation (like reading or writing files). You handle it using a **try-catch** block.

1. **try**: You put the risky file-handling code in the try block.
2. **catch**: If an IOException occurs, the program immediately jumps to the catch block, where you can handle the error gracefully instead of letting the app crash.

Real-Life Example: It's like having a safety net when juggling. You perform your act on the main stage (try). If you drop a ball (IOException), you have a net (catch) to safely stop it from causing a bigger problem.

Code Example:

Java

```
try {
    // Attempt to read from a file that might not exist
    FileReader reader = new FileReader("a_file_that_might_be_missing.txt");
    // ... more code
    reader.close();
} catch (IOException e) {
    // This code runs if the file is not found or another I/O error occurs
    System.out.println("Something went wrong! Could not access the file.");
    System.err.println("Error details: " + e.getMessage());
```

4. What are checked and unchecked exceptions?

This is about when Java forces you to deal with errors.

- **Checked Exceptions**: The compiler *checks* that you've handled these. They are for predictable problems outside your program's control (e.g., a file is missing). You *must* use a try-catch block for them.
- **Unchecked Exceptions**: The compiler *doesn't* check for these. They are usually caused by programming mistakes (e.g., dividing by zero or using a null variable).

Real-Life Example:

- A **checked exception** (IOException) is like planning an outdoor picnic. You *check* the weather forecast because rain is a predictable problem you must plan for (bring an umbrella).
- An **unchecked exception** (NullPointerException) is like tripping on the sidewalk while walking. It's a personal mistake; there's no rule forcing you to plan for it beforehand.

5. How does file writing work in Java?

It involves creating a stream of data from your program to a file on your computer's storage.

1. **Open a Stream**: Create a FileWriter object with the path to your file.
2. **Write Data**: Use the .write() method to send text to the stream.
3. **Close the Stream**: You must close the writer using .close() (or try-with-resources). This step is crucial because it ensures all your data is actually written from memory to the file.

Real-Life Example: It's like writing a letter and mailing it.

1. You get an envelope and address it (FileWriter).
2. You write the letter and put it inside (.write()).
3. You seal the envelope and put it in the mailbox (.close()). If you forget this last step, the letter never gets sent!

Code Example:

Java

```
try (FileWriter writer = new FileWriter("myNote.txt")) {
    writer.write("Hello, this is a new note.\n");
    writer.write("This is the second line.");
    System.out.println("Successfully wrote to the file.");
} catch (IOException e) {
    System.err.println("An error occurred while writing: " + e.getMessage());
}
```

6. What is the difference between append and overwrite mode?

This determines what happens when you write to a file that already exists.

- **Overwrite Mode (Default)**: Deletes all the old content in the file and replaces it with the new content.
- **Append Mode**: Keeps all the old content and adds the new content to the very end of the file.

Real-Life Example:

- **Overwrite** is like erasing a whiteboard completely before writing a new message.
- **Append** is like adding a new item to the bottom of a grocery list without erasing what's already there.

You enable append mode by adding true when creating the FileWriter.

Code Example:

Java

```
// The 'true' flag enables append mode
try (FileWriter writer = new FileWriter("shoppingList.txt", true)) {
    writer.write("Milk\n"); // This will be added to the end of the file
} catch (IOException e) {
    System.err.println("Error appending to file: " + e.getMessage());
}
```

7. What is exception propagation?

If a method causes an error (an exception) but doesn't handle it with try-catch, it **"throws" the exception up to the method that called it**. This "bubbling up" continues up the chain of calls (the "call stack") until some method catches it. If it reaches the top (main method) without being caught, the program crashes.

Real-Life Example: Imagine a game of hot potato. One person (methodC) gets the hot potato (the error) but can't handle it, so they toss it to the person who threw it to them (methodB). methodB also can't handle it and throws it to methodA. This continues until someone catches it or it's dropped, ending the game (the program crashes).

Code Example:

Java

```
public class PropagationExample {
    public static void main(String[] args) {
        try {
            startProcess();
        } catch (Exception e) {
            // The exception is finally caught here!
            System.out.println("Error was caught in the main method.");
        }
    }
static void startProcess() {
        // This method doesn't handle the error, so it propagates it up.
        performCalculation();
    }
static void performCalculation() {
        // This is where the error originates.
        int result = 10 / 0; // Throws ArithmeticException
    }  }
```

8. How to log exceptions?

Logging is the professional way to record errors. Instead of just printing an error to the console (e.printStackTrace()), which is messy, you use a **logging framework** (like SLF4J or Log4j).

Real-Life Example: e.printStackTrace() is like shouting "I have a problem!" in a crowded room. Logging is like writing the problem down neatly in a captain's logbook, noting the time, the location, and what went wrong, so you can analyze it properly later.

Logging lets you control:

- **Where** logs go (to a file, a server, etc.).
- **What level** of detail you record (ERROR, INFO, DEBUG).
- The **format** of the log message (e.g., adding a timestamp).

Code Example (Conceptual):

Java

```
// Import a logger at the top of your class
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


public class MyClass {
   private static final Logger logger = LoggerFactory.getLogger(MyClass.class);


   public void myMethod() {
      try {
         // Risky code here
      } catch (Exception e) {
         // Log the error with a clear message and the exception details
         logger.error("Failed to process user data.", e);
      }
   }  }
```

9. What is a stack trace?

A **stack trace** is the error report that Java prints when an uncaught exception crashes the program. It's a map that shows you exactly where the error happened and how the program got there.

You read it from the **top down**:

- The **top line** is the most important: it tells you the exact error and the file and line number where it occurred.
- The lines below show the "call stack"—the sequence of method calls that led to the error.

Real-Life Example: It's like a trail of footprints leading back from a crime scene. The footprints at the scene (top line) show you where the problem is. The trail leading away shows you the exact path the program took to get there.

Example Stack Trace:

java.lang.NullPointerException: Cannot invoke "String.length()" because "name" is null

    at com.example.User.printNameLength(User.java:25) <-- The error happened here

    at com.example.App.processUser(App.java:15)   <-- This method called the one above

    at com.example.App.main(App.java:10)   <-- And this method called that one

10. When to use finally block?

A **finally** block contains code that **always runs** after a try-catch statement, no matter what happens—whether an exception was thrown or not.

Its classic use case was to clean up resources (like closing files). However, try-with-resources is now the preferred way for that. You should still use finally for other cleanup actions that must happen no matter what.

Real-Life Example: It's like the "turn off the lights" rule when leaving a room. Whether your visit to the room was successful (try) or you had a problem (catch), you *always* turn off the lights (finally) before you leave.

Code Example:

Java

```
try {
   System.out.println("Accessing a resource.");
} catch (Exception e) {
   System.out.println("An error occurred.");
} finally {
   // This code always executes.
   System.out.println("Cleanup complete. Resource has been released.");
}
```