Q1: Write a method intersect that accepts two sorted ArrayList of integers as parameters and returns a new list that contains only the elements that are found in both lists.

Example:
If lists named list1 and list2 initially store:
[1, **4**, 8, 9, **11**, 15, **17**, **28**, 41, **59**]
[**4**, 7, **11**, **17**, 19, 20, 23, **28**, 37, **59**, 81]

Then the call of intersect(list1, list2) returns the list:
[4, 11, 17, 28, 59]

Solution:

```
import java.util.*;
public class Intersection {
    public static void main(String[] args) {
        ArrayList<Integer> l1 = new ArrayList<Integer>();
        ArrayList<Integer> l2 = new ArrayList<Integer>();
        ArrayList<Integer> l3 = new ArrayList<Integer>();
        l1.add(1); l1.add(4); l1.add(8); l1.add(9); l1.add(11);
        l2.add(4); l2.add(7); l2.add(11);
        l3 = intersect(l1, l2);
        System.out.println(l3); }
    public static ArrayList<Integer> intersect(ArrayList<Integer> list1, ArrayList<Integer> list2) {
        int i = 0, j = 0;
        ArrayList<Integer> list3 = new ArrayList<Integer>();
        while (i < list1.size() && j < list2.size()) {
            if (list1.get(i) < list2.get(j)) {
                i++;
            } else if (list1.get(i) > list2.get(j)) {
                j++;
            } else {
                list3.add(list1.get(i));
                i++;
                j++;
            } }
        return list3;
    }
}
```

Q2: Create a comprehensive Java program that demonstrates the handling of different types of exceptions. The program should include the following functionalities:

1. **ArithmeticException**: Perform an arithmetic operation that leads to an illegal mathematical condition, such as division by zero. Ensure the program catches this exception and displays an appropriate error message.
2. **ArrayIndexOutOfBoundsException**: Create an array of integers with a fixed size. Attempt to access an index that is outside the bounds of the array. The program should catch this exception and provide a meaningful error message indicating the problem.
3. **NullPointerException**: Initialize a reference variable to null and attempt to invoke a method on this reference. Catch the resulting NullPointerException and display a message that explains the cause of the exception.
4. **NumberFormatException**: Prompt the user to enter a string. Attempt to parse this string into an integer using Integer.parseInt(). If the string does not contain a valid integer, catch the NumberFormatException and print a message to the user explaining the issue.
5. **InputMismatchException**: Create a scenario where the user is prompted to enter an integer, but instead inputs a string. Use Scanner to capture the input, and catch the InputMismatchException if the input does not match the expected data type. Display an informative message to the user.

Ensure that the program includes appropriate try-catch blocks for each of these exceptions. Additionally, after handling each exception, the program should continue executing and handle any subsequent exceptions without terminating abruptly.

Solution

```
import java.util.InputMismatchException;

import java.util.Scanner;

public class ExceptionHandlingDemo {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // 1. ArithmeticException example

        try {

            int result = 10 / 0; // This will cause ArithmeticException

        } catch (ArithmeticException e) {

            System.out.println("Caught an ArithmeticException: Division by zero is not allowed.");

        }

        // 2. ArrayIndexOutOfBoundsException example

        try {

            int[] numbers = {1, 2, 3};

            int num = numbers[5]; // This will cause ArrayIndexOutOfBoundsException

        } catch (ArrayIndexOutOfBoundsException e) {

            System.out.println("Caught an ArrayIndexOutOfBoundsException: Attempted to access an
index that is out of bounds.");

        }

        // 3. NullPointerException example

        try {

            String str = null;

            int length = str.length(); // This will cause NullPointerException

        } catch (NullPointerException e) {

            System.out.println("Caught a NullPointerException: Tried to access a method on a null
reference.");

        }
    // 4. NumberFormatException example
     try {
        System.out.print("Enter a number: ");
        String invalidNumber = scanner.nextLine();
        int number = Integer.parseInt(invalidNumber); // This may cause NumberFormatException
     } catch (NumberFormatException e) {
        System.out.println("Caught a NumberFormatException: Input string is not a valid integer.");
     }
     // 5. InputMismatchException example
     try {
        System.out.print("Enter an integer: ");
        int userInput = scanner.nextInt(); // This may cause InputMismatchException if input is not an
integer
     } catch (InputMismatchException e) {
        System.out.println("Caught an InputMismatchException: Expected an integer but got a
different data type.");
     } finally {
        scanner.close(); // Close the scanner resource
     }
  }
}
```

Q3: Consider the following Java program that is designed to copy the contents of one text file to another. The program should perform the following tasks:

1. **Prompt the user** to enter the name of the input file to be copied.
2. **Prompt the user** to enter the name of the output file where the content will be copied.
3. **Copy the content** of the input file to the output file using a method called copyFile.
4. **Handle any potential exceptions** that might occur during file operations, such as file not found or IO errors, using try-catch blocks.
5. **Ensure resources are properly closed** after the file operations are complete.

Solution

```
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;
public class FilesAndExceptions {
   public static void main(String[] args) {
      try {
         Scanner keyIn = new Scanner(System.in);
         System.out.print("Input file name: ");
         String fName = keyIn.nextLine();
         File inFile = new File(fName);
         Scanner fileIn = new Scanner(inFile);
         System.out.print("Output file name: ");
         fName = keyIn.nextLine();
         PrintWriter fileOut = new PrintWriter(fName);
         copyFile(fileIn, fileOut);
         fileIn.close();
         fileOut.close();
      } catch (IOException e) {
         System.out.println("Problem with file -- cannot copy: " + e.getMessage()); }}
   public static void copyFile(Scanner inF, PrintWriter outF) {
      while (inF.hasNextLine()) {
         String line = inF.nextLine();
         outF.println(line); }
      }
  }
```

Lab Work (10 Points): Create a Java program that manages a list of student grades. The program should allow the user to perform the following tasks:

1. **Add Grades to an ArrayList:** (3 points)
   - Create an ArrayList to store grades (as integers).
   - Allow the user to add grades to the list through the console input.
2. **Save Grades to a File:** (2 points)
   - Write a method to save the grades from the ArrayList to a text file named grades.txt.
3. **Read Grades from the File and Display Average Grade:** (3 points)
   - Read the grades from the grades.txt file and calculate the average grade.
   - Display the average grade on the console.
4. **Exception Handling:** (2 points)
   - Implement exception handling to manage potential issues such as file not found, incorrect input format when adding grades, or attempting to read from an empty file.