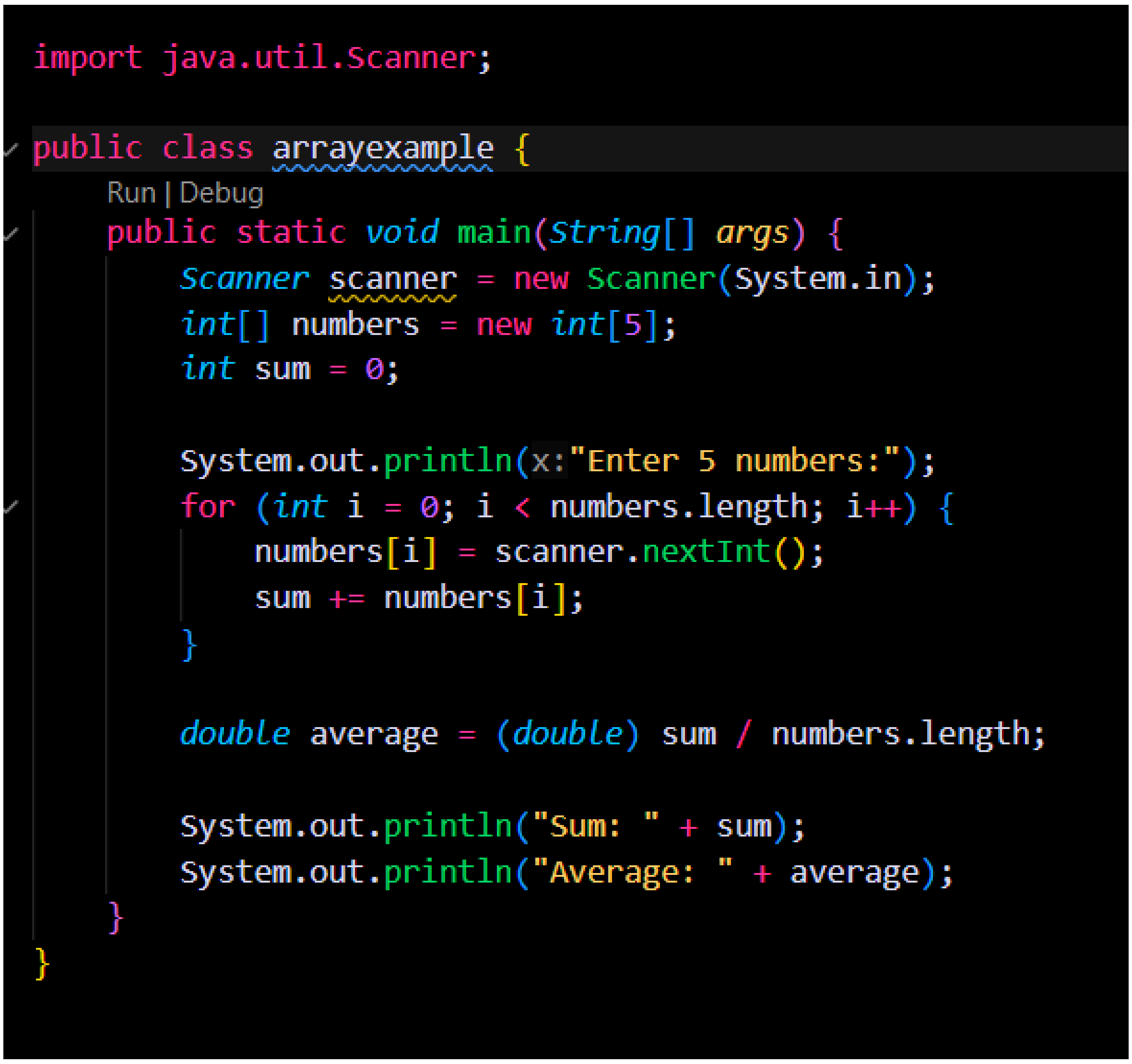
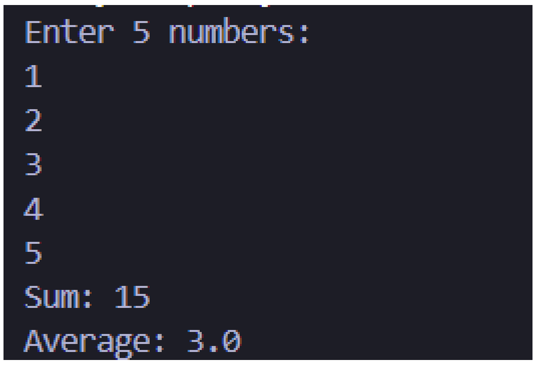
Q1)



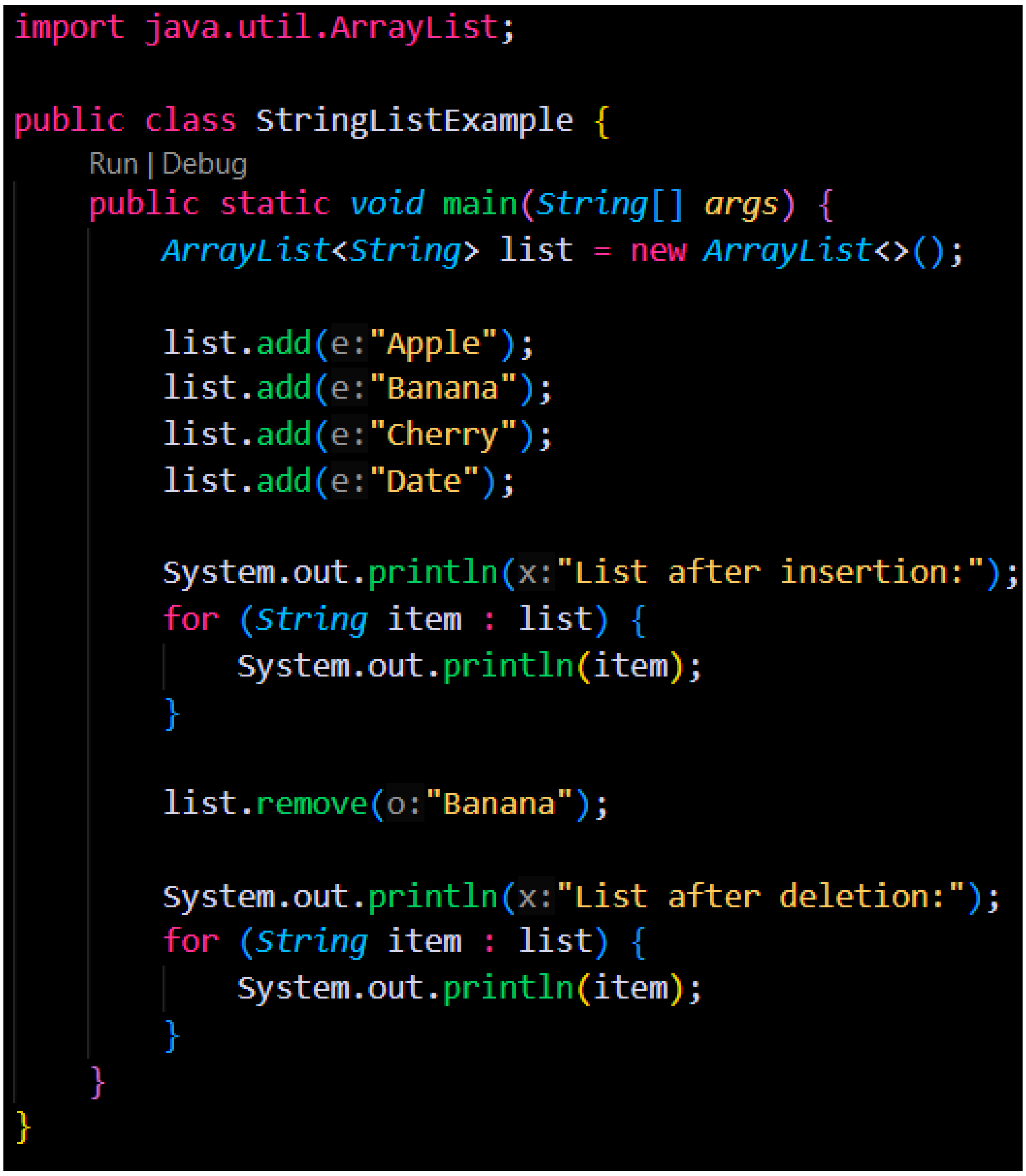
```
import java.util.Scanner;

public class arrayexample {
    Run | Debug
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int[] numbers = new int[5];
        int sum = 0;

        System.out.println(x:"Enter 5 numbers:");
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = scanner.nextInt();
            sum += numbers[i];
        }

        double average = (double) sum / numbers.length;

        System.out.println("Sum: " + sum);
        System.out.println("Average: " + average);
    }
}
```



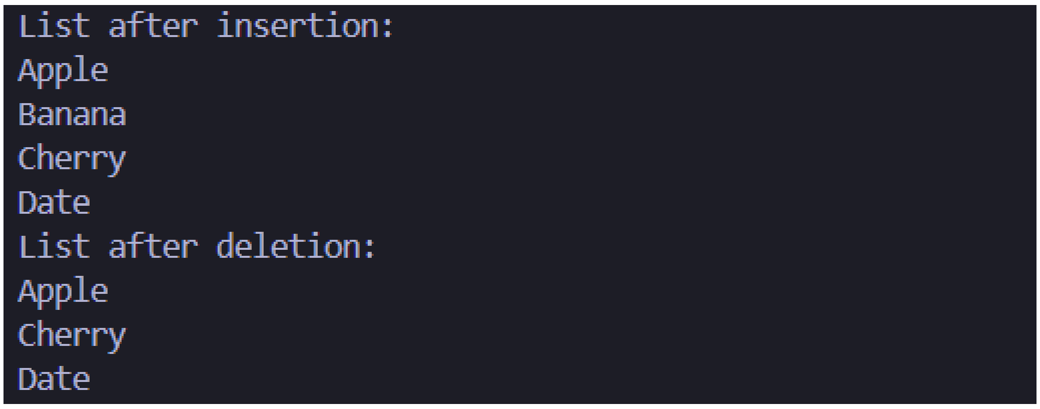
```
Enter 5 numbers:
1
2
3
4
5
Sum: 15
Average: 3.0
```

Q2)

Q3)

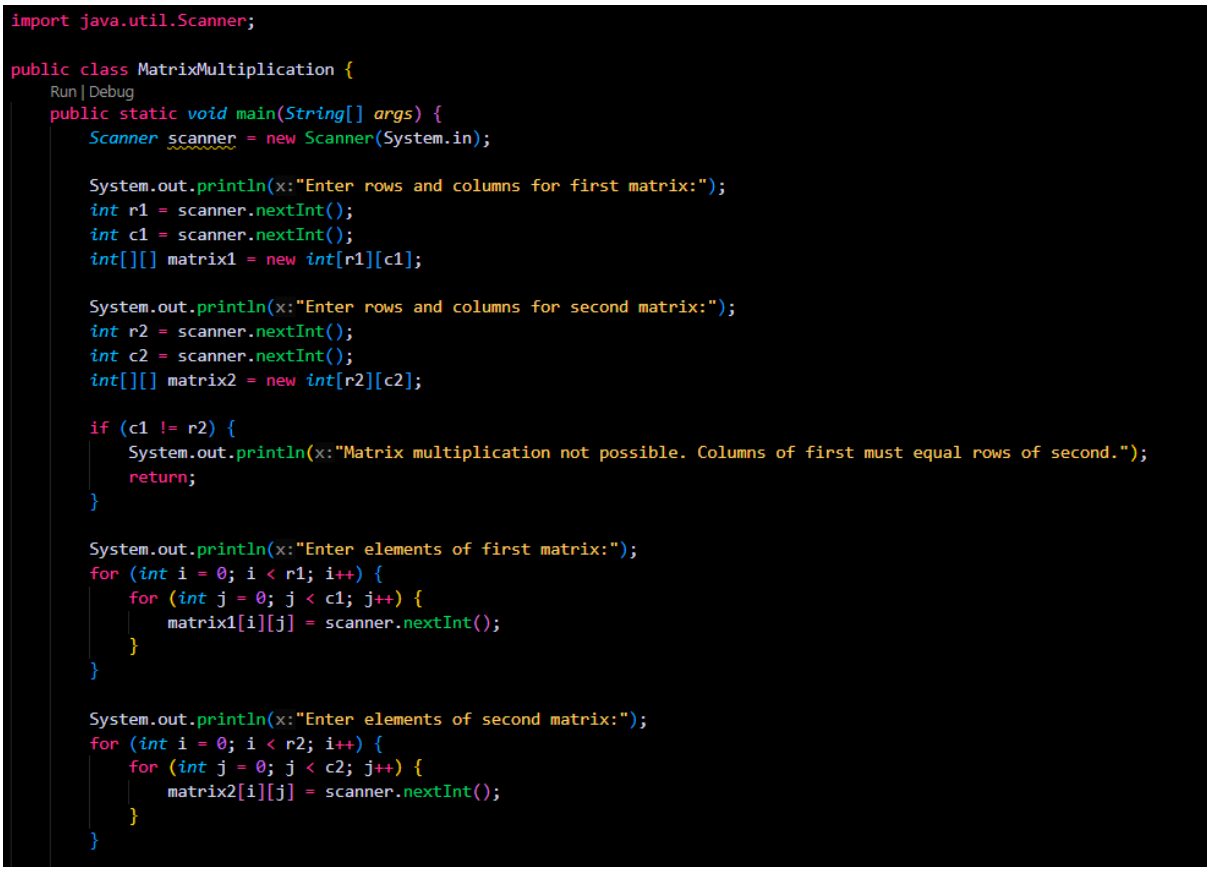


```java
import java.util.ArrayList;

public class StringListExample {
    Run | Debug
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();

        list.add(e:"Apple");
        list.add(e:"Banana");
        list.add(e:"Cherry");
        list.add(e:"Date");

        System.out.println(x:"List after insertion:");
        for (String item : list) {
            System.out.println(item);
        }

        list.remove(o:"Banana");

        System.out.println(x:"List after deletion:");
        for (String item : list) {
            System.out.println(item);
        }
    }
}
```

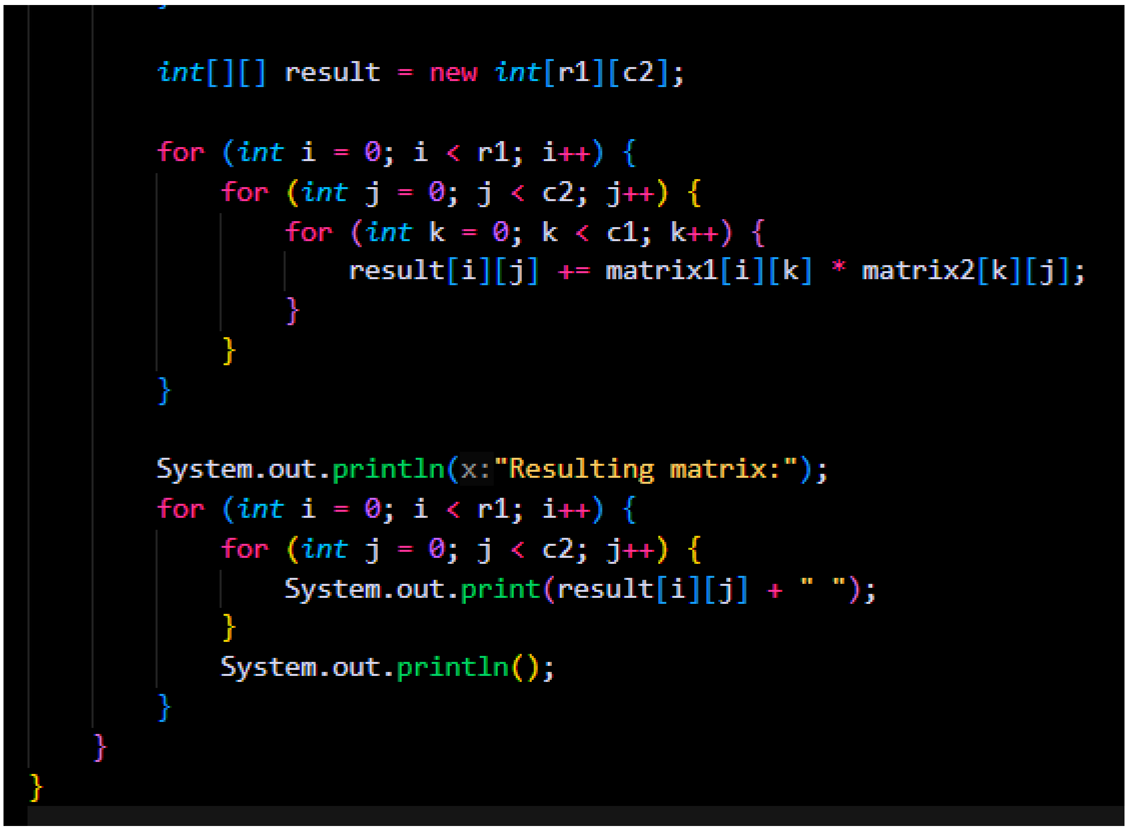


```
List after insertion:
Apple
Banana
Cherry
Date
List after deletion:
Apple
Cherry
Date
```

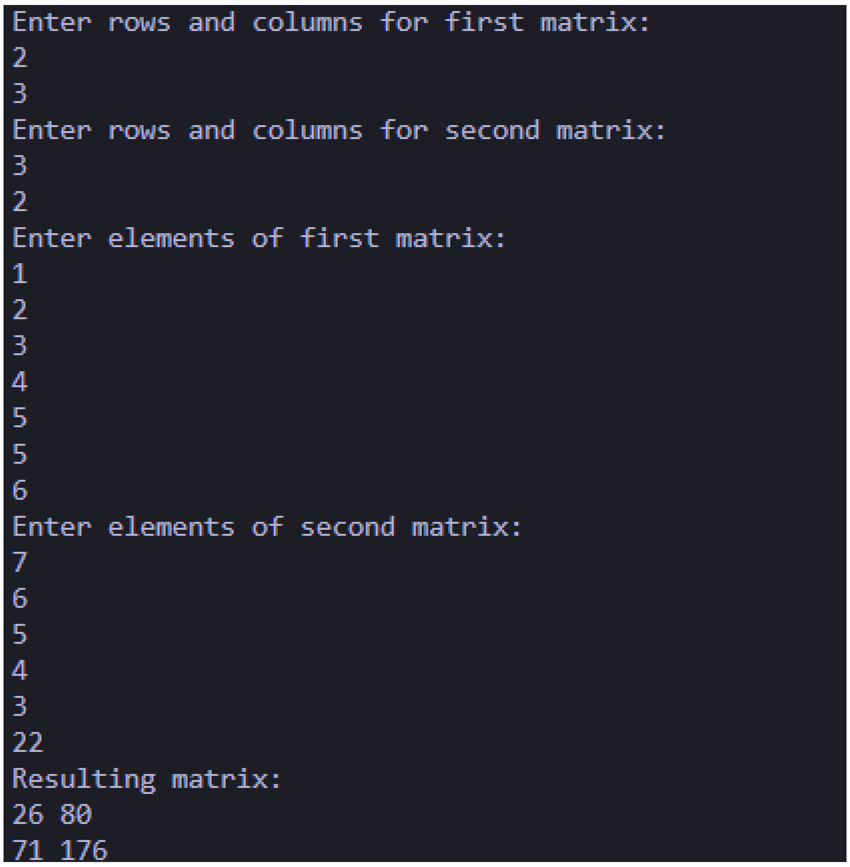
Q3)



```java
import java.util.Scanner;

public class MatrixMultiplication {
    Run | Debug
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println(x:"Enter rows and columns for first matrix:");
        int r1 = scanner.nextInt();
        int c1 = scanner.nextInt();
        int[][] matrix1 = new int[r1][c1];

        System.out.println(x:"Enter rows and columns for second matrix:");
        int r2 = scanner.nextInt();
        int c2 = scanner.nextInt();
        int[][] matrix2 = new int[r2][c2];

        if (c1 != r2) {
            System.out.println(x:"Matrix multiplication not possible. Columns of first must equal rows of second.");
            return;
        }

        System.out.println(x:"Enter elements of first matrix:");
        for (int i = 0; i < r1; i++) {
            for (int j = 0; j < c1; j++) {
                matrix1[i][j] = scanner.nextInt();
            }
        }

        System.out.println(x:"Enter elements of second matrix:");
        for (int i = 0; i < r2; i++) {
            for (int j = 0; j < c2; j++) {
                matrix2[i][j] = scanner.nextInt();
            }
        }
```



```java
        int[][] result = new int[r1][c2];

        for (int i = 0; i < r1; i++) {
            for (int j = 0; j < c2; j++) {
                for (int k = 0; k < c1; k++) {
                    result[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }

        System.out.println(x:"Resulting matrix:");
        for (int i = 0; i < r1; i++) {
            for (int j = 0; j < c2; j++) {
                System.out.print(result[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```



```
Enter rows and columns for first matrix:
2
3
Enter rows and columns for second matrix:
3
2
Enter elements of first matrix:
1
2
3
4
5
5
6
Enter elements of second matrix:
7
6
5
4
3
22
Resulting matrix:
26 80
71 176
```

Q4)

2-Dimensional Arrays
A 2-dimensional array is an array of arrays with fixed-length rows and columns. All rows have the same number of elements.

Syntax: int[][] matrix = new int[3][4];

Memory Allocation:

- A single contiguous block of memory is allocated.
- All elements are stored in row-major order (row by row).
- Rows have equal length and are stored sequentially in memory.

Jagged Arrays (Array of Arrays with Variable Lengths)
A jagged array is an array of arrays where each inner array (row) can have a different length.

Syntax: int[][] jagged = new int[3][];

jagged[0] = new int[2];

jagged[1] = new int[4];

jagged[2] = new int[3];

Memory Allocation:

- Memory is not contiguous.

- Each row is stored as a separate array.
- Rows can have different lengths and are stored independently.

Q5)

throw Keyword

- Used to explicitly throw an exception from a method or a block of code.
- It is followed by an instance of Throwable (usually an exception object).
- Used inside the method body.

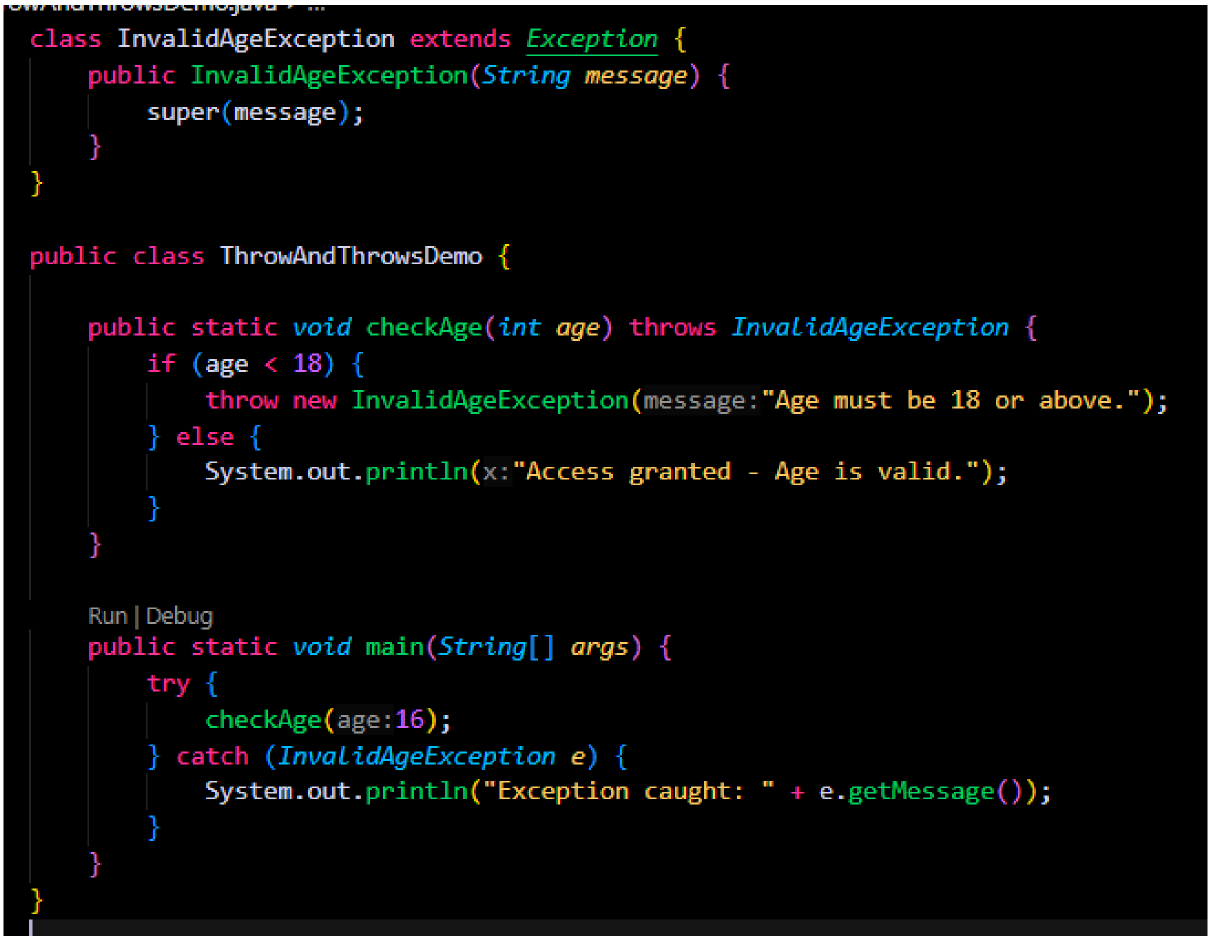
Syntax: throw new ExceptionType("Error Message");

throws Keyword

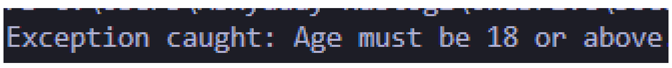
- Used in the method signature to declare that a method might throw one or more exceptions.
- Informs the caller of the method that it must handle or declare the exception.

Syntax: public void methodName() throws ExceptionType {}

Example:

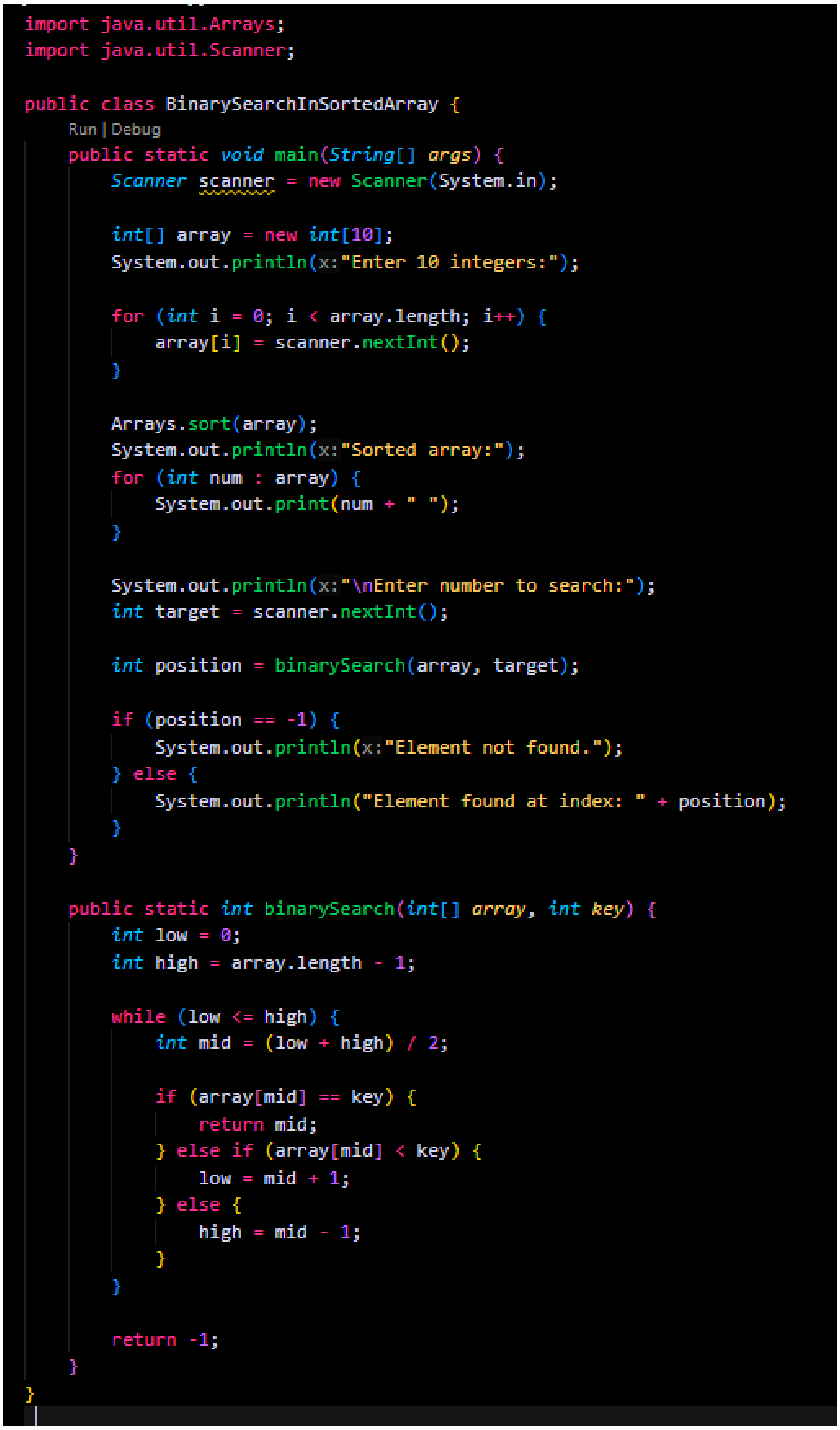


```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class ThrowAndThrowsDemo {

    public static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException(message:"Age must be 18 or above.");
        } else {
            System.out.println(x:"Access granted - Age is valid.");
        }
    }

    Run | Debug
    public static void main(String[] args) {
        try {
            checkAge(age:16);
        } catch (InvalidAgeException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

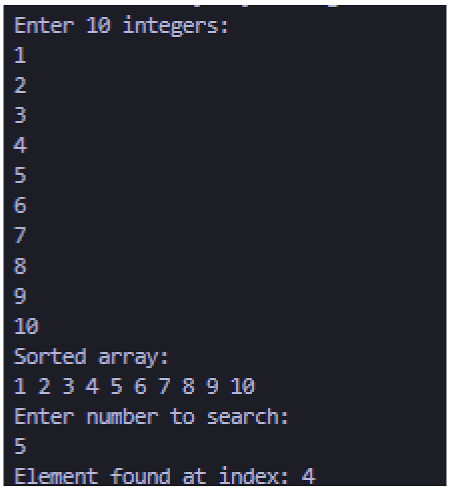


```
Exception caught: Age must be 18 or above
```

Q6)

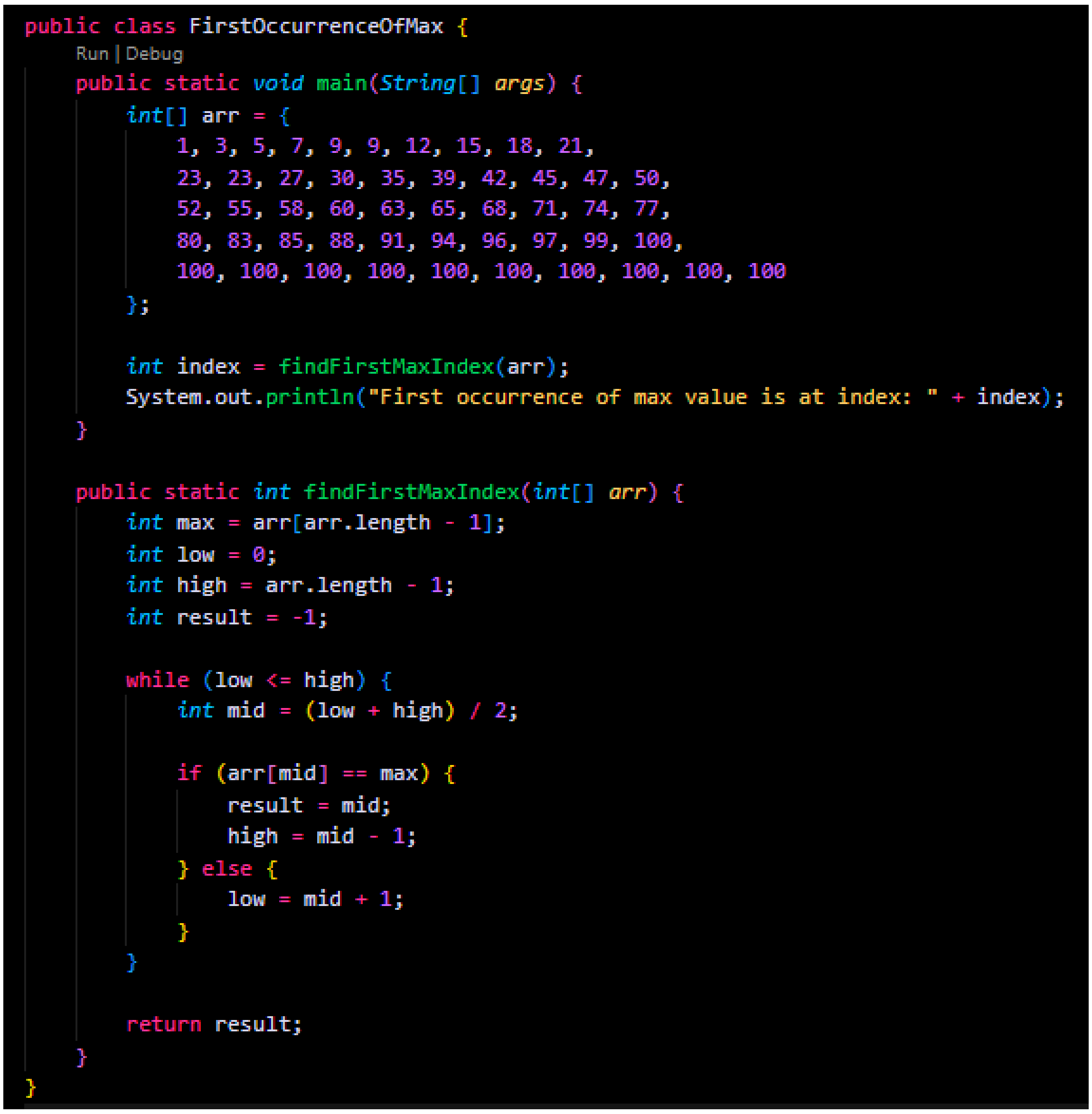


```java
import java.util.Arrays;
import java.util.Scanner;

public class BinarySearchInSortedArray {
    Run | Debug
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int[] array = new int[10];
        System.out.println(x:"Enter 10 integers:");

        for (int i = 0; i < array.length; i++) {
            array[i] = scanner.nextInt();
        }

        Arrays.sort(array);
        System.out.println(x:"Sorted array:");
        for (int num : array) {
            System.out.print(num + " ");
        }

        System.out.println(x:"\nEnter number to search:");
        int target = scanner.nextInt();

        int position = binarySearch(array, target);

        if (position == -1) {
            System.out.println(x:"Element not found.");
        } else {
            System.out.println("Element found at index: " + position);
        }
    }

    public static int binarySearch(int[] array, int key) {
        int low = 0;
        int high = array.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;

            if (array[mid] == key) {
                return mid;
            } else if (array[mid] < key) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }

        return -1;
    }
}
```

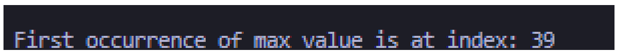


```
Enter 10 integers:
1
2
3
4
5
6
7
8
9
10
Sorted array:
1 2 3 4 5 6 7 8 9 10
Enter number to search:
5
Element found at index: 4
```

Q7)



```java
public class FirstOccurrenceOfMax {
    Run | Debug
    public static void main(String[] args) {
        int[] arr = {
            1, 3, 5, 7, 9, 9, 12, 15, 18, 21,
            23, 23, 27, 30, 35, 39, 42, 45, 47, 50,
            52, 55, 58, 60, 63, 65, 68, 71, 74, 77,
            80, 83, 85, 88, 91, 94, 96, 97, 99, 100,
            100, 100, 100, 100, 100, 100, 100, 100, 100, 100
        };

        int index = findFirstMaxIndex(arr);
        System.out.println("First occurrence of max value is at index: " + index);
    }

    public static int findFirstMaxIndex(int[] arr) {
        int max = arr[arr.length - 1];
        int low = 0;
        int high = arr.length - 1;
        int result = -1;

        while (low <= high) {
            int mid = (low + high) / 2;

            if (arr[mid] == max) {
                result = mid;
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }

        return result;
    }
}
```

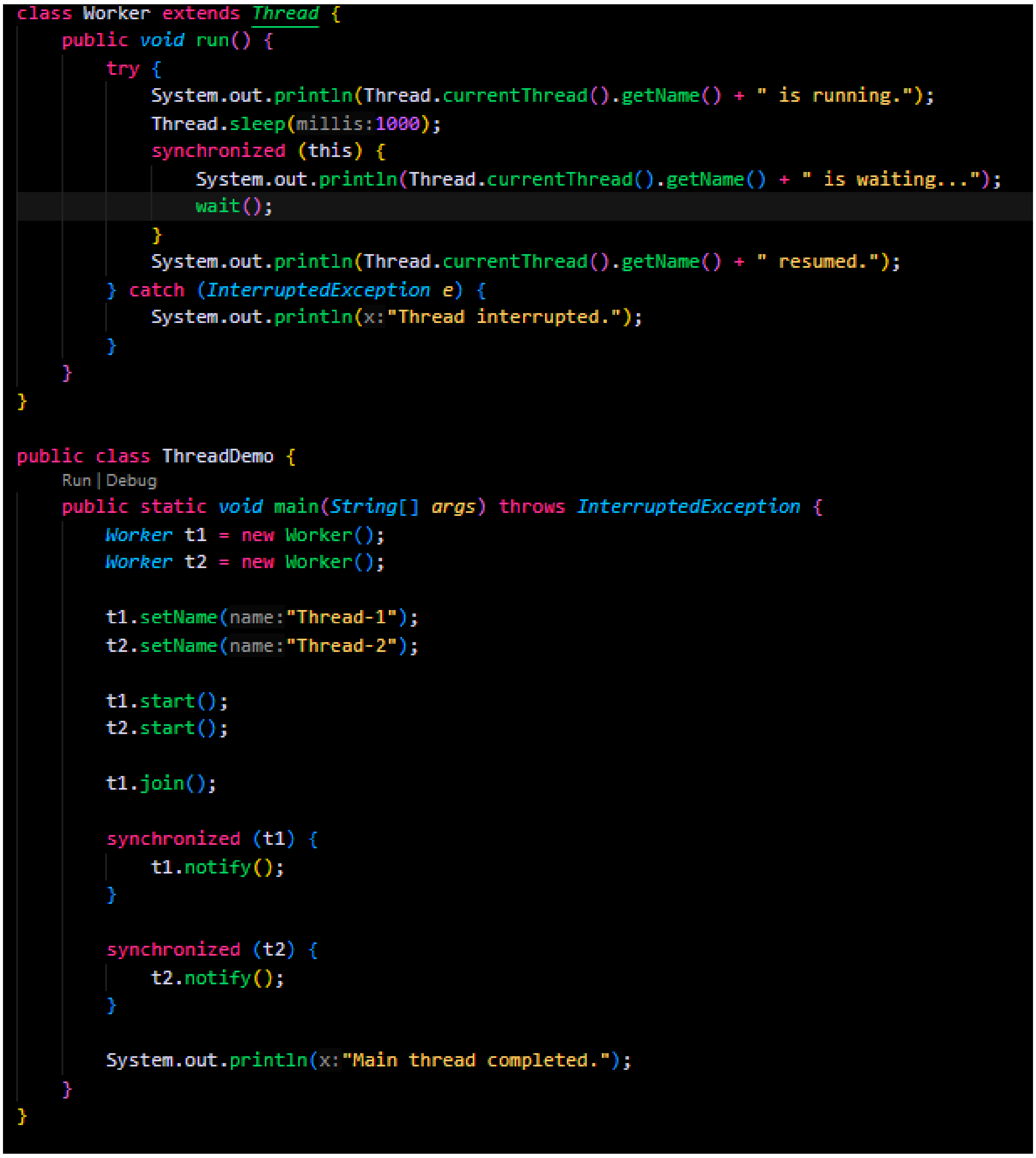


```
First occurrence of max value is at index: 39
```

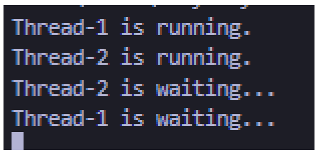
Q8)

Multithreading allows concurrent execution of two or more parts of a program to maximize CPU utilization.

Example:



```java
class Worker extends Thread {
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " is running.");
            Thread.sleep(millis:1000);
            synchronized (this) {
                System.out.println(Thread.currentThread().getName() + " is waiting...");
                wait();
            }
            System.out.println(Thread.currentThread().getName() + " resumed.");
        } catch (InterruptedException e) {
            System.out.println(x:"Thread interrupted.");
        }
    }
}

public class ThreadDemo {
    Run | Debug
    public static void main(String[] args) throws InterruptedException {
        Worker t1 = new Worker();
        Worker t2 = new Worker();

        t1.setName(name:"Thread-1");
        t2.setName(name:"Thread-2");

        t1.start();
        t2.start();

        t1.join();

        synchronized (t1) {
            t1.notify();
        }

        synchronized (t2) {
            t2.notify();
        }

        System.out.println(x:"Main thread completed.");
    }
}
```



```
Thread-1 is running.
Thread-2 is running.
Thread-2 is waiting...
Thread-1 is waiting...
```

Q9)

The Java Collection Framework (JCF) is a set of classes and interfaces that implement commonly reusable data structures. It provides various interfaces like List, Set, Queue, and their implementations such as ArrayList, HashSet, LinkedList, Stack, etc.

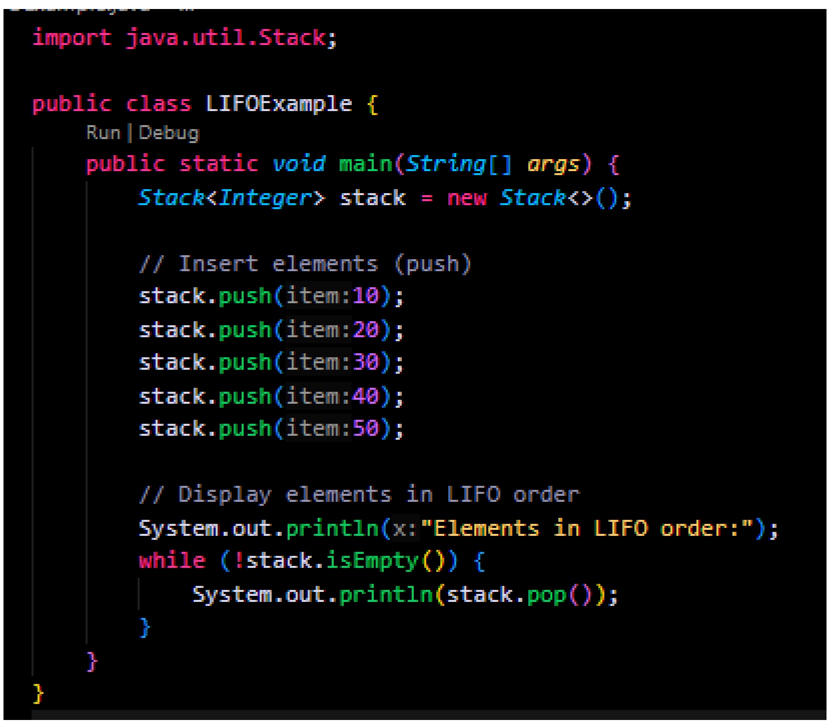
Each collection class is designed for specific types of operations:

- List: Ordered collection (e.g., ArrayList, LinkedList)
- Set: Unordered, no duplicates (e.g., HashSet, TreeSet)
- Queue: FIFO behavior (e.g., LinkedList, PriorityQueue)
- Deque/Stack: LIFO behavior (e.g., Stack, ArrayDeque)

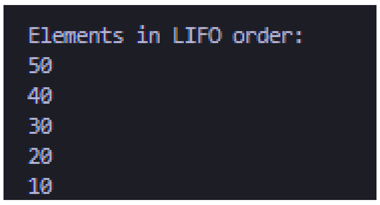
To implement Last-In-First-Out (LIFO) behavior, the most appropriate collection class is Stack. It extends Vector and provides methods like:

- push() – to insert an element
- pop() – to remove the top element
- peek() – to view the top element without removing it

Example:



```java
import java.util.Stack;

public class LIFOExample {
    Run | Debug
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        // Insert elements (push)
        stack.push(item:10);
        stack.push(item:20);
        stack.push(item:30);
        stack.push(item:40);
        stack.push(item:50);

        // Display elements in LIFO order
        System.out.println(x:"Elements in LIFO order:");
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```



```
Elements in LIFO order:
50
40
30
20
10
```

Q10)

Byte-oriented streams work with raw binary data, such as image files, audio files, or any data that is not text-based. These streams read and write data in terms of bytes (8 bits). They are appropriate for handling non-text files.

Key Points:

- Work with raw binary data.
- Data is processed in terms of bytes (8 bits).
- Used for handling any kind of data (images, audio, etc.).
- Can process text files as well but not suitable for character encoding.

Commonly Used Byte-Oriented Stream Classes:

1. InputStream (abstract class)
    - Used for reading byte data.
    - Example: FileInputStream, BufferedInputStream
2. OutputStream (abstract class)
    - Used for writing byte data.
    - Example: FileOutputStream, BufferedOutputStream

Character-oriented streams are specifically designed to handle text data, using Unicode encoding. These streams automatically handle the character encoding/decoding, making them more suitable for working with text files in Java.

Key Points:

- Work with character data, reading and writing 16-bit Unicode characters.
- Use character encoding to correctly interpret text files.
- Preferred for reading and writing text files.
- Ensures that text is correctly interpreted regardless of the platform.

Commonly Used Character-Oriented Stream Classes:

1. Reader (abstract class)
    - Used for reading character data.
    - Example: FileReader, BufferedReader
2. Writer (abstract class)
    - Used for writing character data.
    - Example: FileWriter, BufferedWriter

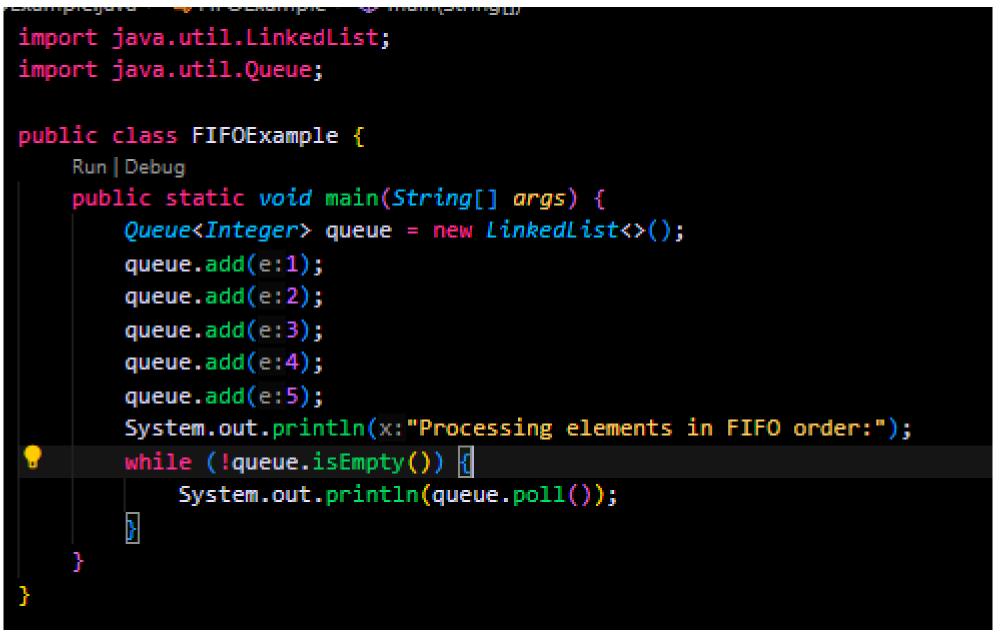
Q11)

The **Java Collection Framework** provides a set of interfaces and classes to handle collections of objects. It includes various types of collections like **List**, **Set**, **Queue**, and **Deque**.

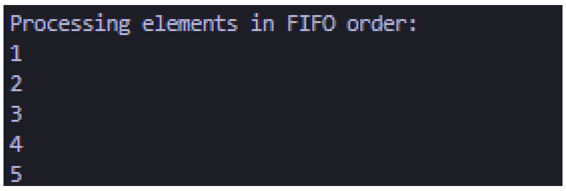
- **Queue** is the collection type most suited for **FIFO (First-In-First-Out)** behavior.
- In a **FIFO** structure, the first element added is the first one to be removed. It is commonly used in scenarios like **message processing** or **task scheduling**.

For implementing FIFO behavior, the most appropriate class from the **Java Collection Framework** is **LinkedList** which implements the **Queue** interface.

Example:



```java
import java.util.LinkedList;
import java.util.Queue;

public class FIFOExample {
    Run | Debug
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(e:1);
        queue.add(e:2);
        queue.add(e:3);
        queue.add(e:4);
        queue.add(e:5);
        System.out.println(x:"Processing elements in FIFO order:");
        while (!queue.isEmpty()) {
            System.out.println(queue.poll());
        }
    }
}
```



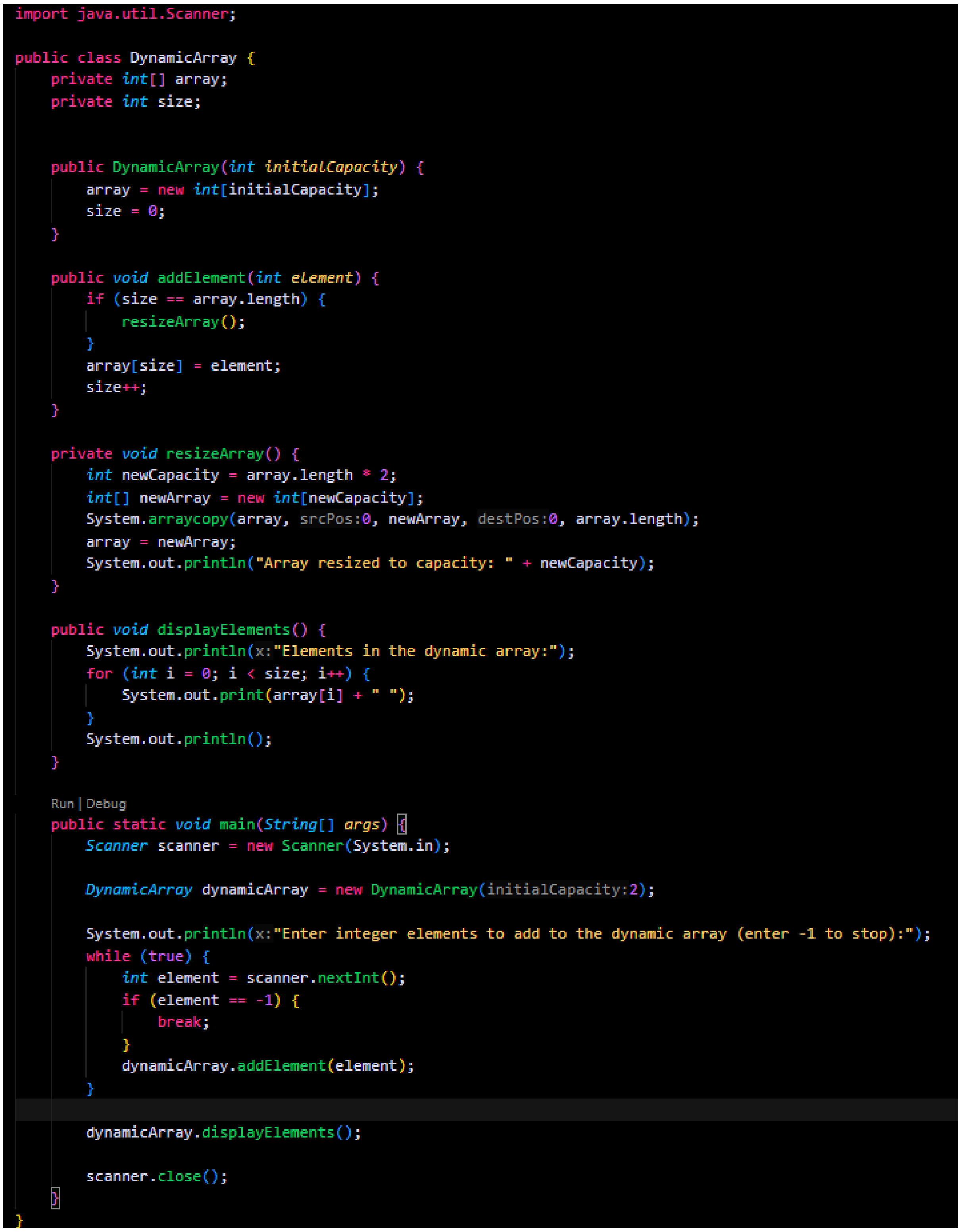
```
Processing elements in FIFO order:
1
2
3
4
5
```

Q12)

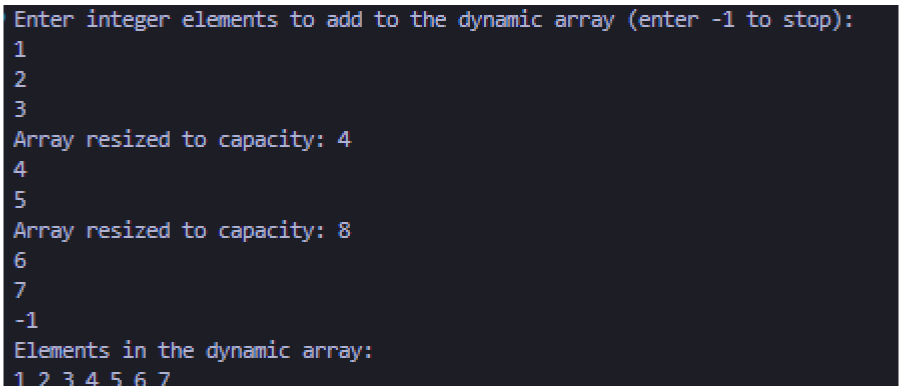
In Java, arrays are fixed in size once they are created, meaning you cannot change their size during runtime. However, by using a **dynamic array**, we can resize the array when it reaches its capacity. This can be achieved by creating a new, larger array and copying the elements from the old array to the new one. A common approach is to double the array size when it's full.

The Java **ArrayList** class is an example of a built-in dynamic array implementation, but here we'll implement a simple dynamic array manually for better understanding.

Example:

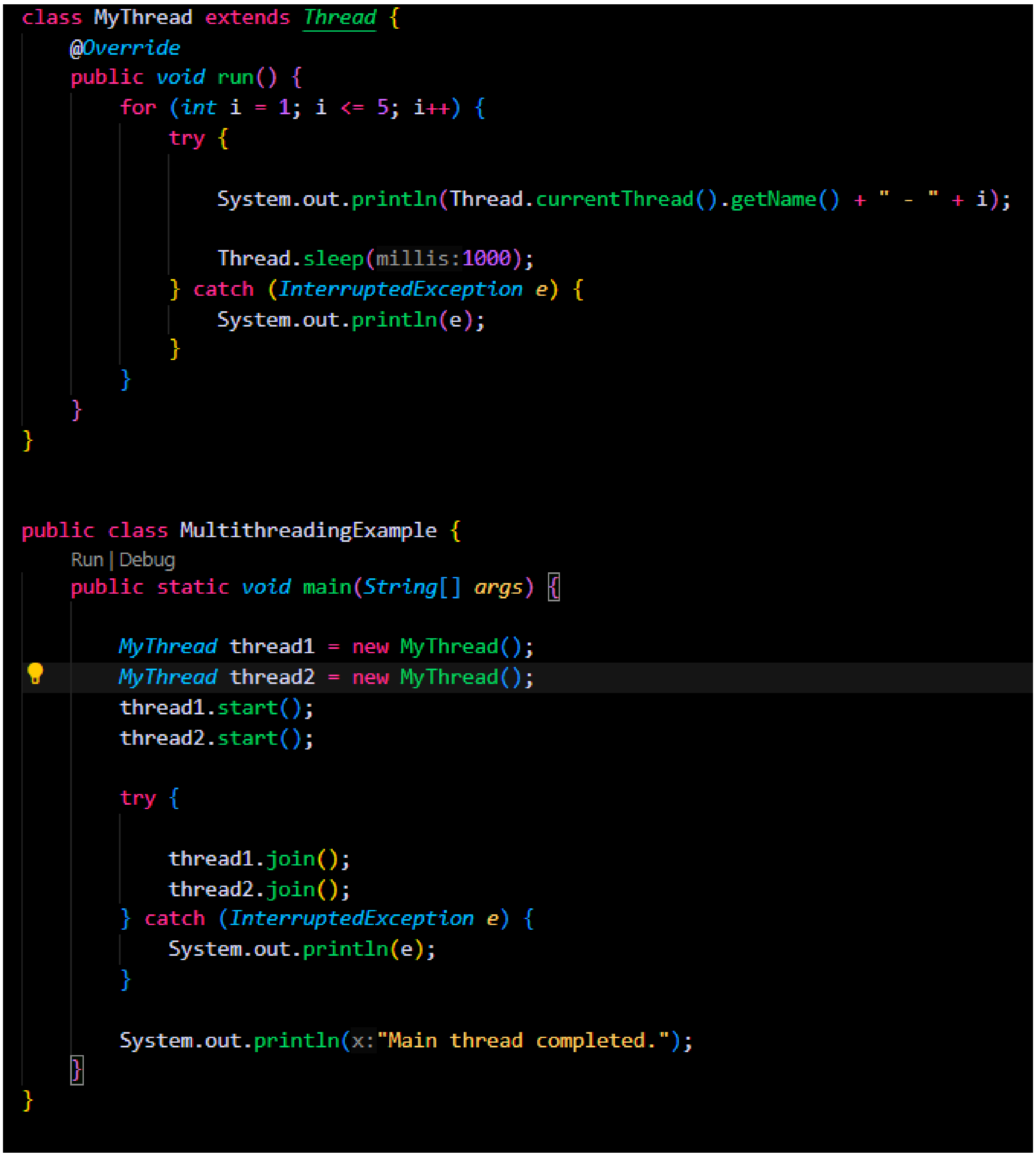


```java
import java.util.Scanner;

public class DynamicArray {
    private int[] array;
    private int size;


    public DynamicArray(int initialCapacity) {
        array = new int[initialCapacity];
        size = 0;
    }

    public void addElement(int element) {
        if (size == array.length) {
            resizeArray();
        }
        array[size] = element;
        size++;
    }

    private void resizeArray() {
        int newCapacity = array.length * 2;
        int[] newArray = new int[newCapacity];
        System.arraycopy(array, srcPos:0, newArray, destPos:0, array.length);
        array = newArray;
        System.out.println("Array resized to capacity: " + newCapacity);
    }

    public void displayElements() {
        System.out.println(x:"Elements in the dynamic array:");
        for (int i = 0; i < size; i++) {
            System.out.print(array[i] + " ");
        }
        System.out.println();
    }

    Run | Debug
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        DynamicArray dynamicArray = new DynamicArray(initialCapacity:2);

        System.out.println(x:"Enter integer elements to add to the dynamic array (enter -1 to stop):");
        while (true) {
            int element = scanner.nextInt();
            if (element == -1) {
                break;
            }
            dynamicArray.addElement(element);
        }

        dynamicArray.displayElements();

        scanner.close();
    }
}
```

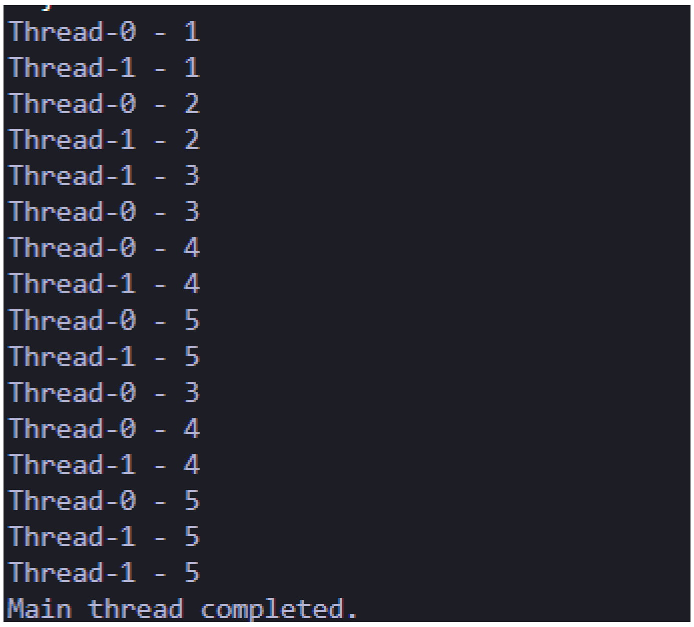


```
Enter integer elements to add to the dynamic array (enter -1 to stop):
1
2
3
Array resized to capacity: 4
4
5
Array resized to capacity: 8
6
7
-1
Elements in the dynamic array:
1 2 3 4 5 6 7
```

Q13)

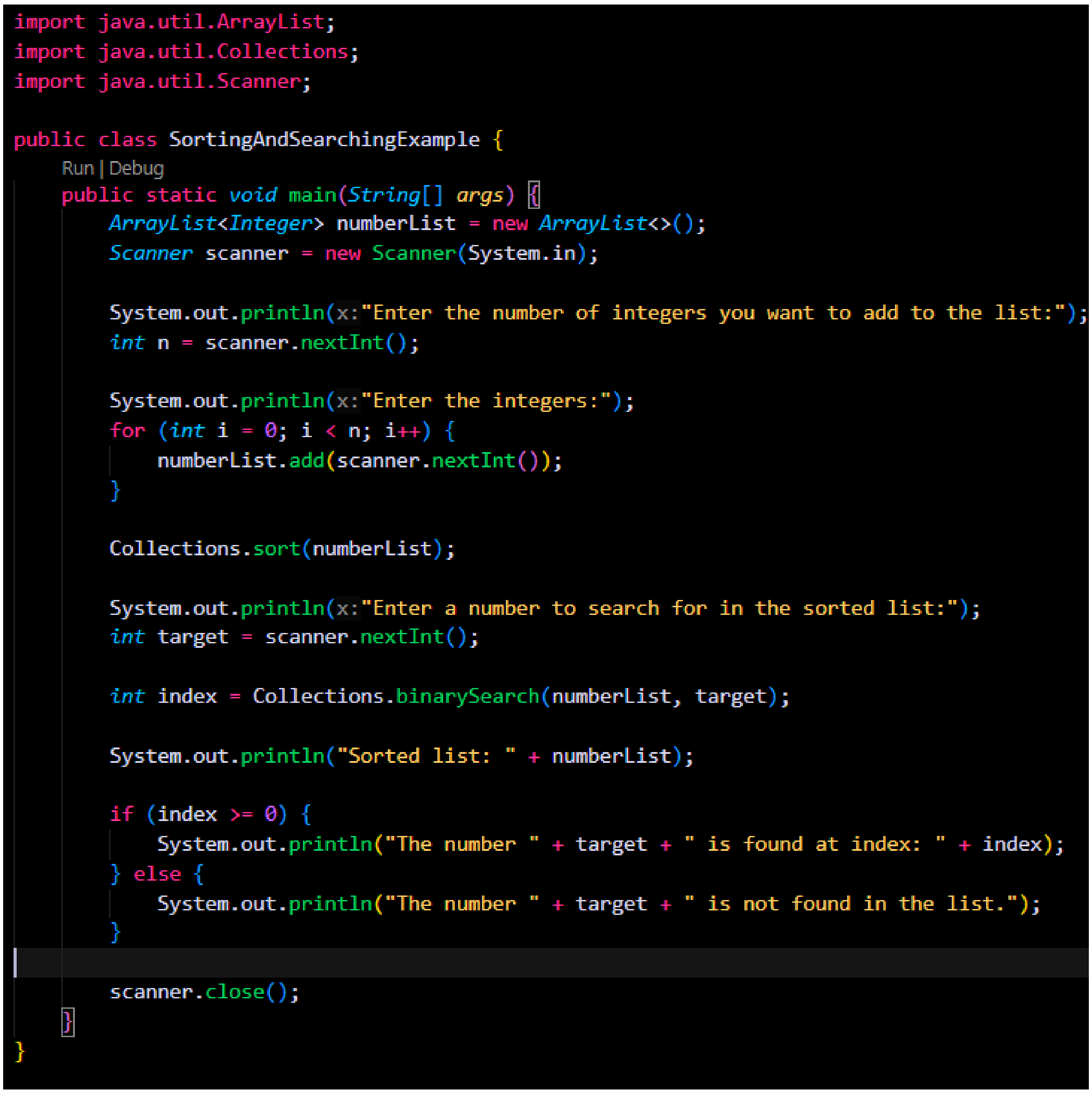


```java
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {

                System.out.println(Thread.currentThread().getName() + " - " + i);

                Thread.sleep(millis:1000);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}


public class MultithreadingExample {
    Run | Debug
    public static void main(String[] args) {

        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();
        thread1.start();
        thread2.start();

        try {

            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        System.out.println(x:"Main thread completed.");
    }
}
```

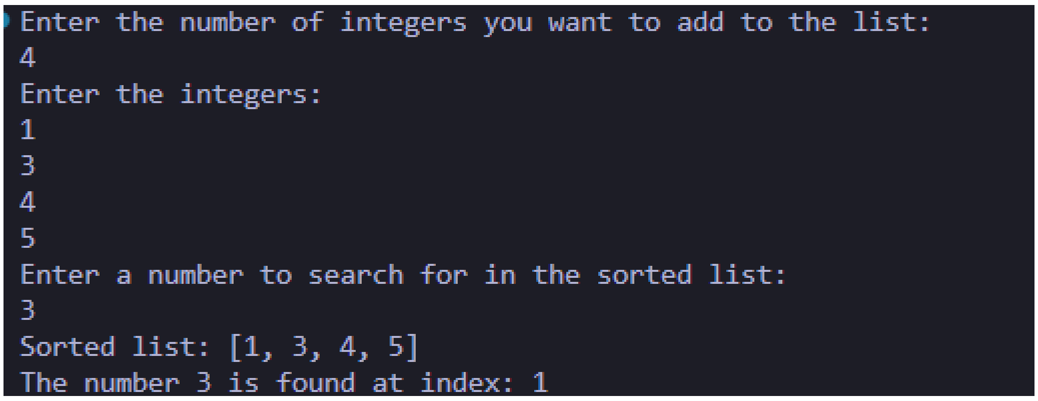

```
Thread-0 - 1
Thread-1 - 1
Thread-0 - 2
Thread-1 - 2
Thread-1 - 3
Thread-0 - 3
Thread-0 - 4
Thread-1 - 4
Thread-0 - 5
Thread-1 - 5
Thread-0 - 3
Thread-0 - 4
Thread-1 - 4
Thread-0 - 5
Thread-1 - 5
Thread-1 - 5
Main thread completed.
```

Q14)

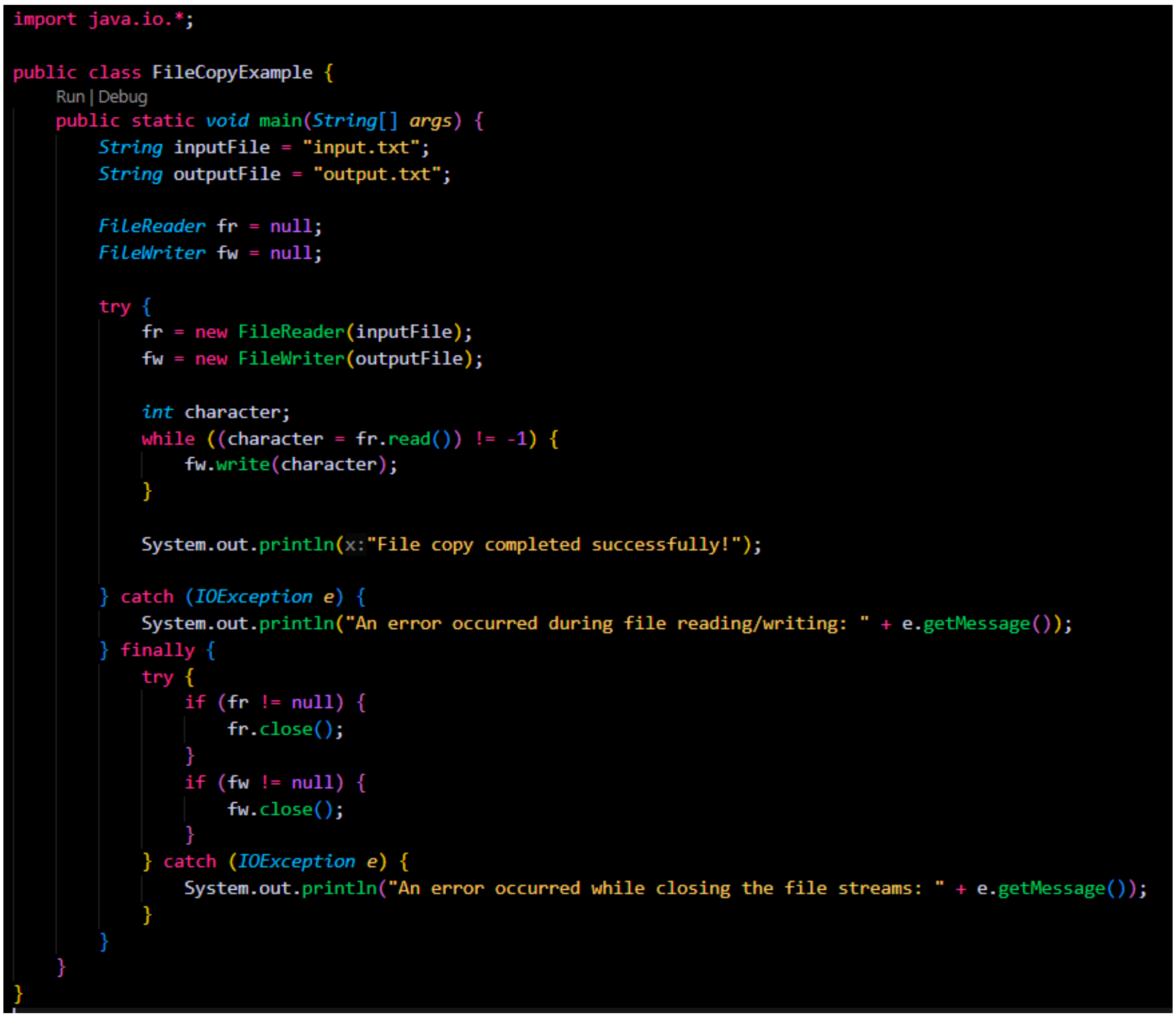
Q15)



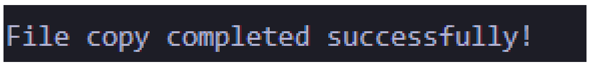
```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;

public class SortingAndSearchingExample {
    Run | Debug
    public static void main(String[] args) {
        ArrayList<Integer> numberList = new ArrayList<>();
        Scanner scanner = new Scanner(System.in);

        System.out.println(x:"Enter the number of integers you want to add to the list:");
        int n = scanner.nextInt();

        System.out.println(x:"Enter the integers:");
        for (int i = 0; i < n; i++) {
            numberList.add(scanner.nextInt());
        }

        Collections.sort(numberList);

        System.out.println(x:"Enter a number to search for in the sorted list:");
        int target = scanner.nextInt();

        int index = Collections.binarySearch(numberList, target);

        System.out.println("Sorted list: " + numberList);

        if (index >= 0) {
            System.out.println("The number " + target + " is found at index: " + index);
        } else {
            System.out.println("The number " + target + " is not found in the list.");
        }

        scanner.close();
    }
}
```



```
Enter the number of integers you want to add to the list:
4
Enter the integers:
1
3
4
5
Enter a number to search for in the sorted list:
3
Sorted list: [1, 3, 4, 5]
The number 3 is found at index: 1
```



```
import java.io.*;

public class FileCopyExample {
    Run | Debug
    public static void main(String[] args) {
        String inputFile = "input.txt";
        String outputFile = "output.txt";

        FileReader fr = null;
        FileWriter fw = null;

        try {
            fr = new FileReader(inputFile);
            fw = new FileWriter(outputFile);

            int character;
            while ((character = fr.read()) != -1) {
                fw.write(character);
            }

            System.out.println(x:"File copy completed successfully!");

        } catch (IOException e) {
            System.out.println("An error occurred during file reading/writing: " + e.getMessage());
        } finally {
            try {
                if (fr != null) {
                    fr.close();
                }
                if (fw != null) {
                    fw.close();
                }
            } catch (IOException e) {
                System.out.println("An error occurred while closing the file streams: " + e.getMessage());
            }
        }
    }
}
```



```
File copy completed successfully!
```