Question1:

Checked vs Unchecked Exceptions in Java

a. **Checked Exceptions:** These are the exceptions that a method is expected to catch and handle. They extend Java. lang. Exceptions are checked at compile-time. If a method throws a checked exception, it must declare it using the throws keyword or handle it using a try-catch block. An example is IOException.
 Unchecked Exceptions: These are exceptions that occur due to programming errors, such as logic errors or improper use of an API. They extend Java. lang. Runtime Exception and are not checked at compile-time but at runtime. Examples include NullPointerException, ArrayIndexOutOfBoundsException, etc.

b. The "super" keyword in Java The super keyword in Java is a reference variable that is used to refer to the immediate parent class object. It can be used to invoke the parent class methods or constructors. For example:

```
Java
class Parent {
   void display () {
      System. out.println("Parent display ()");
   }
}

 class Child extends Parent {

   void display () {
      super. Display (); // Calls Parent's display method
      System. out. println ("Child display()");
   }
}
```

c. **Generics in Java** Generics in Java are a language feature that allows you to define classes, interfaces, and methods with type parameters. Generics improve type safety because they enforce compile-time type checking, reducing runtime errors due to incorrect types. They also improve code reusability because you can write a method, class, or interface that can be used with different types. For example, List<T> can be used as List<String>, List<Integer>, etc., with the same underlying logic.

Question 2:

a. SOLID Principles in Java Programming: The SOLID principles are a set of software design principles that promote code readability, maintainability, and scalability. They are:
Single Responsibility Principle (SRP): This principle states that every Java class must perform a single functionality. This makes the code easier to understand and test and reduces coupling.
Open-Closed Principle (OCP): According to this principle, entities (methods, classes, modules, etc.) should be open for extension but closed for modification. This means that you should be able to add new functionality without changing the existing code.
Liskov Substitution Principle (LSP): This principle ensures that a subclass can substitute its superclass without affecting the correctness of the program.
Interface Segregation Principle (ISP): This principle suggests that clients should not be forced to depend on interfaces they do not use.
Dependency Inversion Principle (DIP): This principle states that high-level modules should not depend on low-level modules. Both should depend on abstractions.

b. Lambda Expressions and Functional Interfaces in Java Lambda expressions in Java are a new feature introduced in Java 8. They provide a clear and concise way to represent one method interface using an expression. Lambda expressions also improve the Collection libraries making it easier to iterate through, filter, and extract data from a Collection.
A Functional Interface is an interface with only one abstract method. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. The java. util. function package contains many built-in functional interfaces that you can use.
c. Design Patterns in Java Development Design patterns are reusable solutions to common problems in software design. They represent best practices and are templates that can be applied to multiple real-world problems.
There are three types of design patterns:
Creational Patterns: These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. Examples include Factory Method, Abstract Factory, Builder, Prototype, and Singleton.
Structural Patterns: These patterns deal with object composition and typically identify simple ways to realize relationships between different objects. Examples include Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy.
Behavioral Patterns: These patterns are specifically concerned with communication between objects. Examples include Observer, State, Strategy, Template Method, and Visitor.
Design patterns provide standard terminology and are specific to a particular scenario. They allow developers to communicate using well-known, well-understood names for software interactions.

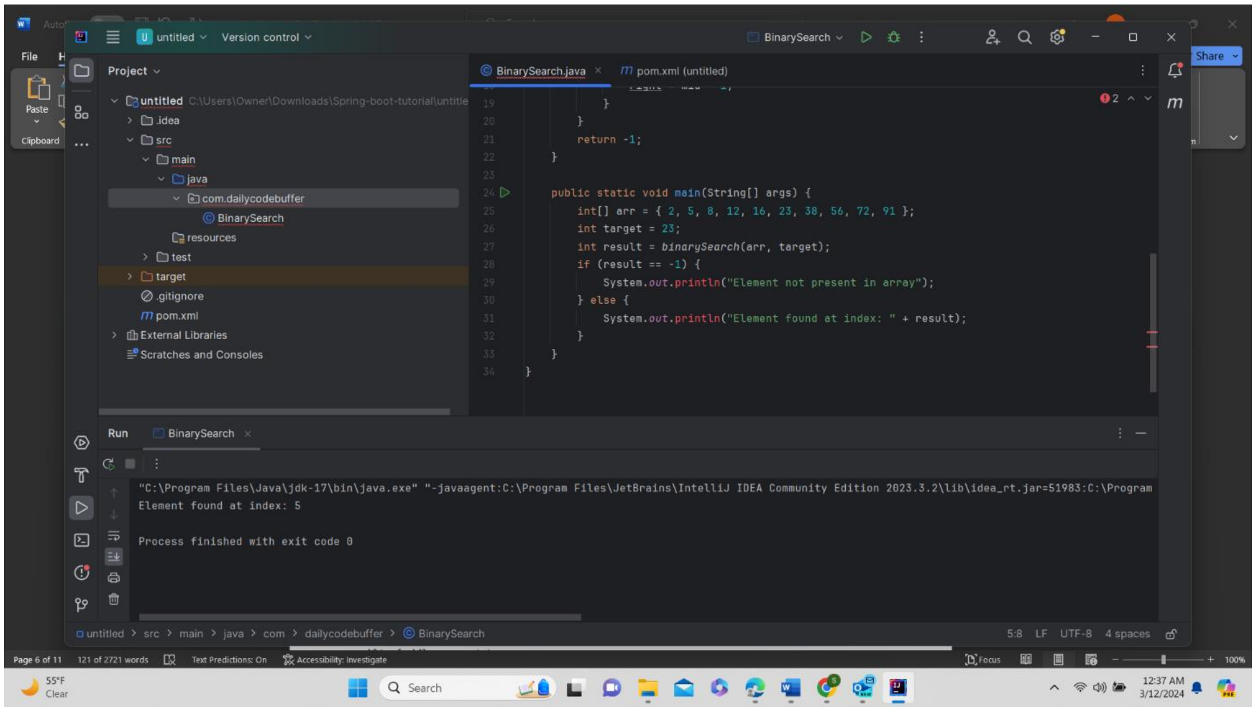
Question 3:

```java
package com.dailycodebuffer;

//TIP To <b>Run</b> code, press <shortcut actionId="Run"/> or
// click the <icon src="AllIcons.Actions.Execute"/> icon in the gutter.
public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int left = 0;
        int right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) {
                return mid;
            }
            if (arr[mid] < target) {
                left = mid + 1;
            }
            else {
                right = mid - 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = { 2, 5, 8, 12, 16, 23, 38, 56, 72, 91 };
        int target = 23;
        int result = binarySearch(arr, target);
        if (result == -1) {
            System.out.println("Element not present in array");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }
}
```

Output:



```java
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = { 2, 5, 8, 12, 16, 23, 38, 56, 72, 91 };
        int target = 23;
        int result = binarySearch(arr, target);
        if (result == -1) {
            System.out.println("Element not present in array");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }
}
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2023.3.2\lib\idea_rt.jar=51983:C:\Program
Element found at index: 5

Process finished with exit code 0
```

This is a Java program that implements a binary search algorithm to find a target element in an integer array. Here's a breakdown of the code:

1. The package declaration package com.dailycodebuffer; specifies the package for this program.

2. The BinarySearch class contains the binary search algorithm.

3. The binarySearch() method takes an integer array and an integer target as input, and returns the index of the target element if it is found, or -1 if it is not found.

4. The method starts by initializing left to 0 and right to the length of the array minus 1. This sets up the bounds of the search.

5. The method then enters a while loop that continues as long as the left is less than or equal to the right.

6. Inside the loop, the method calculates the midpoint of the search range (mid) and checks whether the target element is equal to the element at the midpoint.

7. If the target element is found, the method returns the index mid.

8. If the target element is less than the element at the midpoint, the method updates left to mid + 1 and repeats the loop.

9. If the target element is greater than the element at the midpoint, the method updates right to mid-1 and repeats the loop.

10. If the loop completes without finding the target element, the method returns -1.

11. The main() method creates an integer array and sets the target element to 23.

12. It then calls the binarySearch() method with the array and target element as input and prints a message indicating whether the element was found or not.

Question 4.

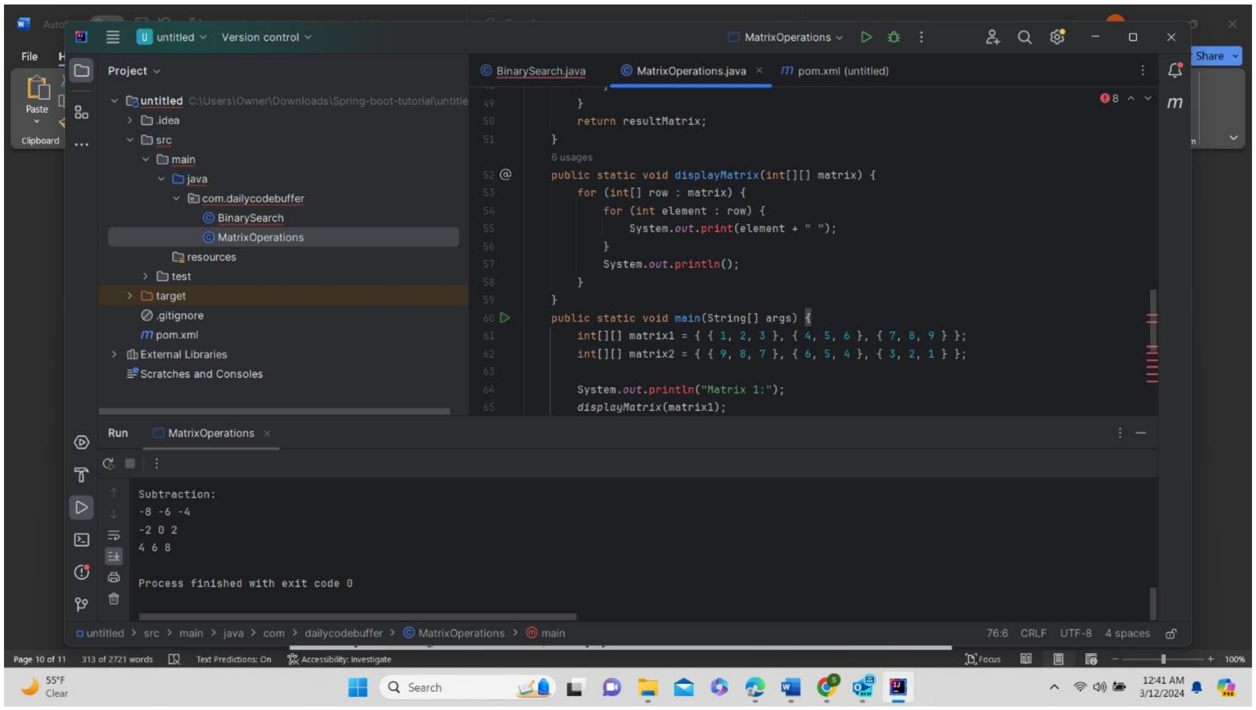
```java
package com.dailycodebuffer;

public class MatrixOperations {
    public static int[][] addMatrices(int[][] matrix1, int[][] matrix2) {
        int rows = matrix1.length;
        int columns = matrix1[0].length;
        int[][] resultMatrix = new int[rows][columns];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                resultMatrix[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        }
        return resultMatrix;
    }
    public static int[][] multiplyMatrices(int[][] matrix1, int[][] matrix2)
{
        int rowsMatrix1 = matrix1.length;
        int columnsMatrix1 = matrix1[0].length;
        int columnsMatrix2 = matrix2[0].length;
        int[][] productMatrix = new int[rowsMatrix1][columnsMatrix2];
        for (int i = 0; i < rowsMatrix1; i++) {
            for (int j = 0; j < columnsMatrix2; j++) {
                for (int k = 0; k < columnsMatrix1; k++) {
                    productMatrix[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }
        return productMatrix;
    }
    public static int[][] transposeMatrix(int[][] matrix) {
        int rows = matrix.length;
        int columns = matrix[0].length;
        int[][] transpose = new int[columns][rows];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                transpose[j][i] = matrix[i][j];
            }
        }
        return transpose;
    }
    public static int[][] subtractMatrices(int[][] matrix1, int[][] matrix2)
{
        int rows = matrix1.length;
        int columns = matrix1[0].length;
        int[][] resultMatrix = new int[rows][columns];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                resultMatrix[i][j] = matrix1[i][j] - matrix2[i][j];
            }
        }
        return resultMatrix;
    }
    public static void displayMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int element : row) {
                System.out.print(element + " ");
            }
            System.out.println();
        }
    }
    public static void main(String[] args) {
        int[][] matrix1 = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
        int[][] matrix2 = { { 9, 8, 7 }, { 6, 5, 4 }, { 3, 2, 1 } };

        System.out.println("Matrix 1:");
        displayMatrix(matrix1);
        System.out.println("\nMatrix 2:");
        displayMatrix(matrix2);
        System.out.println("\nAddition:");
        displayMatrix(addMatrices(matrix1, matrix2));
        System.out.println("\nMultiplication:");
        displayMatrix(multiplyMatrices(matrix1, matrix2));
        System.out.println("\nTranspose of Matrix 1:");
        displayMatrix(transposeMatrix(matrix1));
        System.out.println("\nSubtraction:");
        displayMatrix(subtractMatrices(matrix1, matrix2));
    }
}
```

Output:



```java
        }
        return resultMatrix;
    }
    6 usages
    public static void displayMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int element : row) {
                System.out.print(element + " ");
            }
            System.out.println();
        }
    }
    public static void main(String[] args) {
        int[][] matrix1 = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
        int[][] matrix2 = { { 9, 8, 7 }, { 6, 5, 4 }, { 3, 2, 1 } };

        System.out.println("Matrix 1:");
        displayMatrix(matrix1);
```

```
Subtraction:
-8 -6 -4
-2 0 2
4 6 8

Process finished with exit code 0
```

Explanation:

This code defines a class called MatrixOperations in the package com.dailycodebuffer. It contains several methods for performing operations on matrices, including

1. addMatrices(int[][] matrix1, int[][] matrix2): Adds two matrices element-wise.

2. multiplyMatrices(int[][] matrix1, int[][] matrix2): Multiplies two matrices.

3. transposeMatrix(int[][] matrix): Transposes a matrix (swaps rows and columns).

4. subtractMatrices(int[][] matrix1, int[][] matrix2): Subtracts one matrix from another element-wise.

5. displayMatrix(int[][] matrix): Prints a matrix to the console.

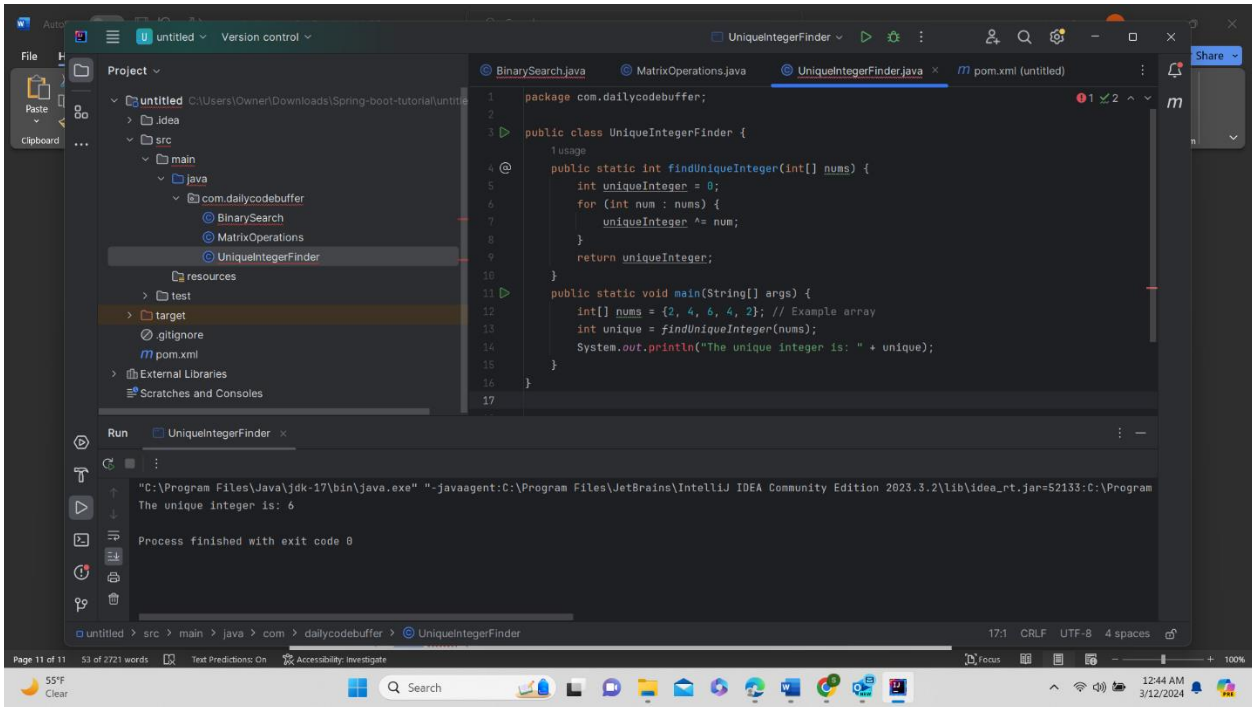
The main() method creates two example matrices matrix1 and matrix2, and then calls each of the above methods in turn, printing the result of each operation to the console. Note that these methods assume that the matrices are rectangular (i.e., have the same number of rows and columns). If the matrices are not rectangular, the methods will throw an IllegalArgumentException.

Question 5.

```java
package com.dailycodebuffer;

public class UniqueIntegerFinder {
    public static int findUniqueInteger(int[] nums) {
        int uniqueInteger = 0;
        for (int num : nums) {
            uniqueInteger ^= num;
        }
        return uniqueInteger;
    }
    public static void main(String[] args) {
        int[] nums = {2, 4, 6, 4, 2}; // Example array
        int unique = findUniqueInteger(nums);
        System.out.println("The unique integer is: " + unique);
    }
}
```

Output:



```java
package com.dailycodebuffer;

public class UniqueIntegerFinder {
    public static int findUniqueInteger(int[] nums) {
        int uniqueInteger = 0;
        for (int num : nums) {
            uniqueInteger ^= num;
        }
        return uniqueInteger;
    }
    public static void main(String[] args) {
        int[] nums = {2, 4, 6, 4, 2}; // Example array
        int unique = findUniqueInteger(nums);
        System.out.println("The unique integer is: " + unique);
    }
}
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2023.3.2\lib\idea_rt.jar=52133:C:\Program
The unique integer is: 6

Process finished with exit code 0
```

This code defines a class called UniqueIntegerFinder in the package com.dailycodebuffer. It has two methods:

1. find unique integer (int[] nums): This method takes an integer array as input and returns a single unique integer that appears only once in the array.

2. main (String[] args): This is the entry point of the program, which demonstrates how to use the findUniqueInteger() method.

Here's how the findUniqueInteger () method works:

1. It initializes a variable unique integer to 0.

2. It iterates through each integer num in the input array nums.

3. For each integer, it uses the bitwise XOR operator (^) to combine a unique integer with num. This has the effect of setting a bit to 1 if either unique integer or num has a 1 in that bit position, but not both.

4. After iterating through all integers, the unique integer will have a 1 in only one-bit position, corresponding to the unique integer.

5. The method returns a unique integer.

The main() method simply creates an example array num with duplicates, calls to find unique integer () on it, and prints the resulting unique integer to the console.