1. Explain in your own words:

a. Explain the difference between checked and unchecked exceptions in Java.

b. What is the purpose of the "super" keyword in Java? Provide an example.

c. What are generics in Java? How do they improve type safety and code reusability?

a. In Java, checked and unchecked exceptions differ in how the compiler handles them. Checked exceptions must be explicitly caught or declared in the method signature using the throws keyword. These exceptions represent conditions that a programmer can anticipate and handle, such as file not found or database connection issues. Unchecked exceptions, however, do not require handling at compile-time. They typically represent programming errors, such as null pointer exceptions or array index out-of-bounds. While it's not mandatory to handle unchecked exceptions, it's good practice to do so to prevent unexpected termination of the program.

b. The "super" keyword in Java refers to the immediate parent class of a subclass. It is commonly used to access superclass methods, constructors, and instance variables from the subclass. One of the primary purposes of "super" is to differentiate between the superclass and subclass members with the same name.

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void sound() {
        super.sound(); // Calling superclass method
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();
```

```
    }
}
```

the Dog class extends the Animal class. The sound() method in Dog class uses the "super" keyword to call the sound() method of the Animal class before adding its own behavior.

c. Generics in Java allow you to create classes, interfaces, and methods that operate on specified types. They provide type safety by enabling you to specify the type of objects that a collection or class can contain or operate on at compile time. Generics also enhance code reusability by allowing you to write code that can work with different data types without sacrificing type safety.

For example, consider a generic class Box:

```
public class Box<T> {
    private T content;

    public void setContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
```

With this generic class, you can create instances of Box that can hold any type of object:

```
Box<Integer> integerBox = new Box<>();
integerBox.setContent(10);
int intValue = integerBox.getContent(); // No casting needed

Box<String> stringBox = new Box<>();
stringBox.setContent("Hello");
String stringValue = stringBox.getContent(); // No casting needed
```

Generics allow you to create reusable code that maintains type safety at compile time, thereby reducing the risk of runtime errors and enhancing the flexibility and maintainability of your code.

2. Explain in your own words

a. Explain the principles of SOLID design and how they apply to Java programming.

b.What are lambda expressions and functional interfaces in Java? How do they facilitate functional programming paradigms?

c.What are design patterns, and why are they important in Java development? Provide examples of commonly used design patterns in Java.

The principles of SOLID design are a set of five design principles intended to make software designs more understandable, flexible, and maintainable. These principles are:

Single Responsibility Principle (SRP): A class should have only one reason to change. This means that a class should have only one responsibility or purpose, and if there are multiple reasons for it to change, it should be divided into separate classes.

Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that you should be able to extend the behavior of a module without modifying its source code.

Liskov Substitution Principle (LSP): Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. In Java, this means that subclasses should be able to override methods from the superclass without changing their behavior.

Interface Segregation Principle (ISP): Clients should not be forced to depend on methods they do not use. This principle suggests that interfaces should be specific to the needs of the clients that use them, rather than being too general.

Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details; rather, details should depend on abstractions. This principle promotes loose coupling between classes by ensuring that high-level modules depend on abstractions rather than concrete implementations.

These principles apply to Java programming by guiding developers to write modular, flexible, and maintainable code that is easier to understand and extend.

b. Lambda expressions in Java provide a concise way to represent anonymous functions or blocks of code. They enable functional programming paradigms by allowing functions to be passed as arguments to other functions, stored in variables, or returned from other functions. Lambda expressions are particularly useful when working with collections and streams in Java, as they allow for more expressive and concise code**.**

Functional interfaces in Java are interfaces that have exactly one abstract method. They serve as the basis for lambda expressions, as lambda expressions can be used to provide implementations for the abstract method of a functional interface. Functional interfaces are annotated with @FunctionalInterface to indicate that they are intended for use with lambda expressions.

Together, lambda expressions and functional interfaces facilitate functional programming paradigms in Java by allowing developers to write more concise and expressive code, making it easier to work with collections, streams, and asynchronous programming.

c. Design patterns are reusable solutions to common problems encountered in software design and development. They provide a structured approach to solving design problems and help to improve code quality, maintainability, and scalability. Design patterns are important in Java development because they encapsulate best practices and proven solutions to recurring design problems, making it easier for developers to write high-quality and maintainable code.

Some commonly used design patterns in Java include:

Singleton Pattern: Ensures that a class has only one instance and provides a global point of access to that instance.

Factory Pattern: Defines an interface for creating objects but allows subclasses to alter the type of objects that will be created.

Observer Pattern: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Decorator Pattern: Allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class.

Strategy Pattern: Defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

These are just a few examples of the many design patterns that can be applied in Java development to solve common design problems effectively.

3. Write a Java program to implement a binary search algorithm to find the index of a given element in a sorted array.

```
public class BinarySearch {

   // Binary search function
```

```
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;

        // Check if target is present at mid
        if (arr[mid] == target)
            return mid;

        // If target is greater, ignore left half
        if (arr[mid] < target)
            left = mid + 1;
        // If target is smaller, ignore right half
        else
            right = mid - 1;
    }

    // If target is not present in the array
    return -1;
}

// Main method to test the binary search function
public static void main(String[] args) {
    int[] arr = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int target = 11;
    int result = binarySearch(arr, target);
    if (result != -1)
        System.out.println("Element found at index: " + result);
    else
```

```
            System.out.println("Element not found in the array");
        }
    }
```

4. Implement Java methods to perform operations on matrices such as addition, multiplication, transposition, etc.

```
public class MatrixOperations {

    // Method to add two matrices
    public static int[][] add(int[][] A, int[][] B) {
        int rows = A.length;
        int cols = A[0].length;
        int[][] result = new int[rows][cols];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = A[i][j] + B[i][j];
            }
        }
        return result;
    }

    // Method to multiply two matrices
    public static int[][] multiply(int[][] A, int[][] B) {
        int rowsA = A.length;
        int colsA = A[0].length;
        int colsB = B[0].length;
        int[][] result = new int[rowsA][colsB];

        for (int i = 0; i < rowsA; i++) {
```

```
        for (int j = 0; j < colsB; j++) {
            for (int k = 0; k < colsA; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}

// Method to transpose a matrix
public static int[][] transpose(int[][] matrix) {
    int rows = matrix.length;
    int cols = matrix[0].length;
    int[][] result = new int[cols][rows];

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[j][i] = matrix[i][j];
        }
    }
    return result;
}

// Method to display a matrix
public static void displayMatrix(int[][] matrix) {
    for (int[] row : matrix) {
        for (int element : row) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
```

```
}

// Main method to test matrix operations
public static void main(String[] args) {
    int[][] A = {
        {1, 2, 3},
        {4, 5, 6}
    };
    int[][] B = {
        {7, 8, 9},
        {10, 11, 12}
    };

    System.out.println("Matrix A:");
    displayMatrix(A);

    System.out.println("\nMatrix B:");
    displayMatrix(B);

    System.out.println("\nMatrix Addition (A + B):");
    int[][] additionResult = add(A, B);
    displayMatrix(additionResult);

    System.out.println("\nMatrix Multiplication (A * B):");
    int[][] multiplicationResult = multiply(A, B);
    displayMatrix(multiplicationResult);

    System.out.println("\nTranspose of Matrix A:");
    int[][] transposedA = transpose(A);
    displayMatrix(transposedA);
}
```

}

5. You are given an array of integers where each element appears twice except for one. Write a Java method to find and return the unique integer.

Example:

Input: [4, 3, 2, 4, 2]

Output: 3

```
import java.util.HashSet;

public class UniqueIntegerFinder {

    public static int findUniqueInteger(int[] nums) {
        HashSet<Integer> set = new HashSet<>();

        for (int num : nums) {
            if (!set.remove(num)) {
                set.add(num);
            }
        }

        // At this point, set should contain only the unique integer
        // There should be exactly one element in the set
        // So, we retrieve and return that element
        return set.iterator().next();
    }

    public static void main(String[] args) {
        int[] nums = {4, 3, 2, 4, 2};
        int uniqueInteger = findUniqueInteger(nums);
```

```
        System.out.println("The unique integer is: " + uniqueInteger);
    }
}
```