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?

Ans)

1. In Java, checked exceptions and unchecked exceptions differ in how the compiler enforces handling or declaration of exceptions. Checked exceptions must either be caught using a try-catch block or declared in the method's signature using the "throws" keyword. Unchecked exceptions, on the other hand, do not require explicit handling or declaration. They are usually subclasses of Runtime Exception. The main distinction lies in the fact that the compiler checks for checked exceptions during compile-time, ensuring that they are handled appropriately, while unchecked exceptions are not subjected to such strict compile-time checking.
2. The "super" keyword in Java is used to refer to the immediate parent class of the current class. It is primarily utilized to access the members of the superclass, including methods, fields, and constructors. One common usage of the "super" keyword is in constructor chaining, where it allows a subclass constructor to call a superclass constructor.
3. Generics in Java allow developers to create classes, interfaces, and methods that operate with parameterized types. They provide a way to specify or enforce the type of objects that a collection, class, or method can work with, thereby enhancing type safety and code reusability. Generics enable the creation of classes and methods that are independent of the specific data types they manipulate, thus promoting more generalized and reusable code. By using generics, developers can avoid type casting and catch type errors at compile-time rather than at runtime, which helps in detecting errors early in the development process and making the code more robust.

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.

Ans)

1. SOLID design principles are a set of five design principles that promote maintainable, flexible, and scalable software design. They are:

\* Single Responsibility Principle (SRP): A class should have only one reason to change.

\* Open-Closed Principle (OCP): A class should be open to extension but closed to modification.

\* Liskov Substitution Principle (LSP): Derived classes should be substitutable for their base classes.

\* Interface Segregation Principle (ISP): A client should not be forced to depend on interfaces it does not use.

\* Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules, but both should depend on abstractions.

These principles help developers write clean, modular, and maintainable code in Java.

1. Lambda expressions are concise, anonymous functions that can be defined inline. Functional interfaces are interfaces with a single abstract method (SAM). Together, they enable functional programming paradigms like map, filter, and reduce. Java 8's lambda expressions and functional interfaces simplify code and improve readability.
2. Design patterns are reusable solutions to common software design problems. They provide a standard approach to solving specific design issues, making code more predictable and maintainable. Examples of commonly used design patterns in Java include:

\* Singleton: Ensures a class has only one instance.

\* Factory Method: Provides a way to create objects without specifying the exact class type.

\* Observer: Enables objects to be notified of changes to other objects.

\* Decorator: Adds functionality to an object without changing its external interface.

\* Command: Encapsulates a request or action as an object.

Using design patterns helps developers write more robust, flexible, and maintainable code

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

Ans)

public class BinarySearch {

public static int binarySearch(int[] array, int target) {

int left = 0;

int right = array.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (array[mid] == target)

return mid;

if (array[mid] < target)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

public static void main(String[] args) {

int[] sortedArray = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};

int target = 13;

int result = binarySearch(sortedArray, 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.

Ans)

public class MatrixOperations {

public static int[][] addMatrices(int[][] matrix1, int[][] matrix2) {

int rows = matrix1.length;

int cols = matrix1[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] = matrix1[i][j] + matrix2[i][j];

} }

return result;

}

public static int[][] multiplyMatrices(int[][] matrix1, int[][] matrix2) {

int rows1 = matrix1.length;

int cols1 = matrix1[0].length;

int cols2 = matrix2[0].length;

int[][] result = new int[rows1][cols2];

for (int i = 0; i < rows1; i++) {

for (int j = 0; j < cols2; j++) {

for (int k = 0; k < cols1; k++) {

result[i][j] += matrix1[i][k] \* matrix2[k][j];

}

}

}

return result;

}

public static int[][] transposeMatrix(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;

}

public static void printMatrix(int[][] matrix) {

int rows = matrix.length;

int cols = matrix[0].length;

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

System.out.print(matrix[i][j] + " ");

}

System.out.println();

}

}

public static void main(String[] args) {

int[][] matrix1 = {{1, 2, 3}, {4, 5, 6}};

int[][] matrix2 = {{7, 8, 9}, {10, 11, 12}};

System.out.println("Matrix 1:");

printMatrix(matrix1);

System.out.println("Matrix 2:");

printMatrix(matrix2);

int[][] sum = addMatrices(matrix1, matrix2);

System.out.println("Sum of matrices:");

printMatrix(sum);

int[][] product = multiplyMatrices(matrix1, matrix2);

System.out.println("Product of matrices:");

printMatrix(product);

System.out.println("Transpose of matrix 1:");

int[][] transpose = transposeMatrix(matrix1);

printMatrix(transpose);

}

}

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

Ans)

import java.util.HashMap;

import java.util.Map;

public class UniqueIntegerFinder {

public static int findUniqueInteger(int[] nums) {

Map<Integer, Integer> countMap = new HashMap<>();

for (int num : nums) {

countMap.put(num, countMap.getOrDefault(num, 0) + 1);

}

for (Map.Entry<Integer, Integer> entry : countMap.entrySet()) {

if (entry.getValue() == 1) {

return entry.getKey();

}

}

throw new IllegalArgumentException("No unique integer found");

}

public static void main(String[] args) {

int[] nums = {4, 3, 2, 4, 2};

int uniqueInteger = findUniqueInteger(nums);

System.out.println("Unique integer: " + uniqueInteger);

}

}