1

a. In Java, exceptions can be categorized into two types: checked exceptions and unchecked exceptions.

Checked Exceptions: These are exceptions that are checked at compile time. It means the compiler forces you to handle these exceptions. If a method throws a checked exception, then the caller must handle that exception using a try-catch block or declare it in its throws clause. Examples of checked exceptions in Java include IOException, SQLException, ClassNotFoundException, etc.

Unchecked Exceptions: Also known as runtime exceptions, these exceptions are not checked at compile time. It means the compiler doesn’t force you to handle them explicitly. Unchecked exceptions occur during runtime and are usually indicative of programming errors or unexpected conditions. Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, etc.

b. The super keyword in Java is used to refer to the superclass (parent class) of the current object or to call the superclass constructor.

When used to call the superclass constructor, super() must be the first statement in the subclass constructor. It allows you to call a constructor of the superclass explicitly. This is often used when the subclass needs to initialize the superclass’s state before continuing with its own initialization.

c. Generics in Java provide a way to create classes, interfaces, and methods that operate with type parameters. They allow you to specify the type of objects that a container class can contain or that a method can operate on. Generics improve type safety and code reusability by enabling compile-time type checking and eliminating the need for explicit type casting.

Benefits of generics in Java:

Type Safety: With generics, the compiler can detect type mismatches at compile time, which helps catch errors early in the development process rather than at runtime.

Code Reusability: Generics enable you to write classes and methods that can work with any type, improving code reusability. You can create generic classes and methods that are not tied to specific data types.

Elimination of Type Casting: Generics eliminate the need for explicit type casting because the compiler ensures type compatibility. This leads to cleaner and more readable code.

2

a. SOLID Design Principles: SOLID is an acronym that represents a set of five design principles for writing maintainable, scalable, and understandable code:

Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should have only one responsibility. This principle encourages creating smaller, focused classes that are easier to understand, maintain, and test.

Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This principle promotes designing code in a way that allows adding new functionality without altering existing code, usually achieved through inheritance, composition, or interfaces.

Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types without affecting the correctness of the program. This principle ensures that derived classes can be used interchangeably with their base classes, maintaining correctness and avoiding unexpected behavior.

Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use. This principle suggests breaking down interfaces into smaller, more specific ones so that clients only need to depend on the interfaces that are relevant to them, reducing coupling and preventing interface pollution.

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; details should depend on abstractions. This principle encourages decoupling between components by depending on abstractions rather than concrete implementations, facilitating flexibility and easier maintenance.

These principles can be applied to Java programming by following practices such as creating cohesive classes with single responsibilities, using interfaces to define contracts, favoring composition over inheritance, and designing classes to be open for extension and closed for modification.

b. Lambda Expressions and Functional Interfaces: Lambda expressions introduce a concise way to represent anonymous functions in Java. They are essentially a shorthand for defining small, single-method implementations of functional interfaces. A functional interface is an interface that contains only one abstract method and can be used as the basis for lambda expressions.

Lambda expressions and functional interfaces facilitate functional programming paradigms in Java by enabling:

Concise Syntax: Lambda expressions provide a more compact syntax for writing code, especially for implementing functional interfaces.

Functional Composition: They allow for the composition of functions, enabling developers to combine multiple functions into a single operation.

Parallelism and Concurrency: Lambda expressions can be used effectively with Java’s streams API to enable parallelism and concurrency, allowing for more efficient processing of large datasets.

Flexibility: They provide a way to pass behavior as a parameter to methods, enhancing flexibility and enabling higher-order functions.

c. Design Patterns: Design patterns are reusable solutions to commonly occurring problems in software design. They provide templates for solving design issues and promoting best practices in code organization, maintainability, and scalability. Design patterns are important in Java development because they:

Promote Reusability: Design patterns encapsulate solutions to common problems, making them reusable across different projects and scenarios.

Improve Maintainability: By following established design patterns, developers can create code that is easier to understand, maintain, and extend.

Facilitate Communication: Design patterns provide a common language and framework for discussing and documenting solutions to design problems, enhancing communication among developers.

Encourage Best Practices: Design patterns embody best practices and principles of object-oriented design, promoting good coding practices and architectural decisions.

Examples of 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, providing a flexible alternative to subclassing for extending functionality.

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

These patterns, among others, provide solutions to common problems encountered in software design and development, helping to create more robust, maintainable, and scalable Java applications.

3

```
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 found at index " + result);
        else
            System.out.println("Element not found in the array");
    }
}
```

4

```
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 displayMatrix(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:");
        displayMatrix(matrix1);

        System.out.println("\nMatrix 2:");
        displayMatrix(matrix2);

        System.out.println("\nMatrix Addition:");
        int[][] additionResult = addMatrices(matrix1, matrix2);
        displayMatrix(additionResult);

        System.out.println("\nMatrix Multiplication:");
        int[][] multiplicationResult = multiplyMatrices(matrix1, matrix2);
        displayMatrix(multiplicationResult);

        System.out.println("\nTranspose of Matrix 1:");
        int[][] transposeResult = transposeMatrix(matrix1);
        displayMatrix(transposeResult);
    }
}
```

5

```
public class Main
{
 public static void main(String[] args) {
     int[] array={4, 3, 2, 4, 2};
     int result=array[0];
  for(int i=1;i<array.length;i++){
      result = result ^ array[i];
  }
  System.out.println("Unique Number is "+result);
 }
}
```