Aneri Moradia

Java Fundamentals - 5

Vellunta Technologies

Java Batch 33

Manager: Shiva Reddy

Lead: Deepak Nathi

March 11th, 2024

**Question 1:**

**a.**

Checked exceptions in Java are those that are checked at compile time. This means that if a piece of code within a method throws a checked exception, the method must either handle the exception using a try-catch block or specify that it throws the exception using the throws keyword in its method signature. Checked exceptions include both fully checked exceptions, where all child classes are also checked (like IOException and InterruptedException), and partially checked exceptions, where some child classes are unchecked (like Exception). For example, in the given Java program, the compiler detects that the main() method uses FileReader(), readLine(), and close() methods, all of which throw checked exceptions (FileNotFoundException and IOException). Since these exceptions are not handled or declared in the main() method, the program does not compile.

Unchecked exceptions, on the other hand, are not checked at compile time in Java. These exceptions, which include subclasses of Error and RuntimeException, are not enforced by the compiler to be handled or declared. It is up to the programmer to handle or specify these exceptions if needed. In the provided Java program, the ArithmeticException thrown at runtime is an example of an unchecked exception. The compiler allows the program to compile because unchecked exceptions are not required to be handled or declared. However, when the program is executed and encounters the division by zero (int z = y / x;), it throws an ArithmeticException, which is a runtime exception.

**b.**

The super keyword in Java is used to refer to the parent class when working with objects, particularly in the context of inheritance and polymorphism. There are several uses of the super keyword:

Calling a superclass constructor: When a subclass is created, its constructor must call the constructor of its parent class. This is done using the super() keyword, which calls the constructor of the parent class.

Calling a superclass method: A subclass can call a method defined in its parent class using the super keyword. This is useful when the subclass wants to invoke the parent class’s implementation of the method in addition to its own.

Accessing a superclass field: A subclass can access a field defined in its parent class using the super keyword. This is useful when the subclass wants to reference the parent class’s version of a field.

It's important to note the following rules and restrictions related to the super keyword:

super must be the first statement in a constructor: When calling a superclass constructor, the super() statement must be the first statement in the constructor of the subclass.

super cannot be used in a static context: The super keyword cannot be used in a static context, such as in a static method or a static variable initializer.

Here's an example illustrating the use of the super keyword:

class Vehicle {

int maxSpeed = 120;

}

class Car extends Vehicle {

int maxSpeed = 180;

void display() {

System.out.println("Maximum Speed: " + super.maxSpeed);

}

}

class Test {

public static void main(String[] args) {

Car small = new Car();

small.display();

}

}

In this example, the Car class extends the Vehicle class. The display() method in the Car class uses the super.maxSpeed expression to access the maxSpeed field from the parent Vehicle class, demonstrating the use of the super keyword to access a superclass field. When the display() method is called in the main() method, it prints the maximum speed of the Vehicle class, which is 120.

**c.**

Generics in Java refer to parameterized types, allowing types such as `Integer`, `String`, or user-defined types to be parameters to methods, classes, and interfaces. This enables the creation of classes that can work with different data types. Entities such as classes, interfaces, or methods that operate on parameterized types are termed generic entities.

Generics are essential in Java for two main reasons:

**Code Reuse:** With generics, a method, class, or interface can be written once and used with any type, enhancing code reusability. This means that developers can create generic algorithms or data structures that are not tied to a specific type, making their code more flexible and adaptable.

**Type Safety:** Generics provide type safety, which means that errors are detected at compile time rather than at runtime. This ensures that potential type-related issues are identified during the development phase, preventing runtime exceptions and making the code more robust. For instance, consider the example Java program provided, where an `ArrayList` is created without specifying its type. While the compiler allows adding elements of different types (`String` and `Integer`), it results in a runtime exception when attempting to retrieve an element, demonstrating the lack of type safety without generics.

// Creating an ArrayList without any type specified

ArrayList al = new ArrayList();

al.add("Sachin");

al.add("Rahul");

al.add(10); // Compiler allows this

String s1 = (String)al.get(0);

String s2 = (String)al.get(1);

// Causes Runtime Exception

String s3 = (String)al.get(2);

In this example, the lack of generics allows the programmer to add an `Integer` object (`10`) to the `ArrayList`, which is intended to store `String` objects. This error is not detected by the compiler but results in a runtime exception when attempting to retrieve the element as a `String`. Generics help prevent such errors by enforcing type safety at compile time.

**Question 2:**

**a.**

In software development, Object-Oriented Design (OOD) is essential for creating code that is flexible, scalable, maintainable, and reusable. One crucial aspect of good OOD is adhering to the SOLID principles, introduced by Robert C. Martin (Uncle Bob), which serve as a coding standard. The SOLID principles comprise five key principles:

**1.** **Single Responsibility Principle (SRP):** This principle emphasizes that a class should have only one reason to change, meaning it should have a single responsibility or purpose. When implementing software, different team members handle various tasks, such as front-end design, testing, or backend development. Similarly, each class should focus on a single job to avoid complex and lengthy code. It is essential to use layers in your application and break down large, monolithic classes into smaller, more focused ones.

**2. Open/Closed Principle:** The Open/Closed Principle states that software entities, such as classes or modules, should be open for extension but closed for modification. This means that you should be able to extend a class's behavior without altering its existing code. For instance, if developer A releases a library or framework, developer B can add new features or modify behavior by extending the existing class rather than directly modifying it. This separation ensures better stability, maintainability, and minimizes changes in the existing codebase.

**3. Liskov’s Substitution Principle (LSP):** Named after Barbara Liskov, this principle asserts that derived or child classes must be substitutable for their base or parent classes without altering the desirable properties of the program. In other words, any subclass should be usable in place of its superclass without causing unexpected behavior. For example, if a class represents a rectangle, its subclass representing a square should inherit its properties without changing the behavior expected from a rectangle.

**4. Interface Segregation Principle (ISP):** Unlike the previous principles which focus on classes, ISP applies to interfaces. It suggests that clients should not be forced to implement interfaces that are irrelevant to them. Instead, interfaces should be tailored to specific client needs, avoiding fat or bloated interfaces. For example, in a restaurant scenario, if a customer is vegetarian, they should only receive a menu card with vegetarian items, not a comprehensive menu including non-vegetarian items.

**5. Dependency Inversion Principle (DIP):** DIP emphasizes decoupling dependencies between high-level and low-level modules or classes. High-level modules should not depend on low-level ones; both should depend on abstractions. This principle encourages loose coupling by ensuring that changes in one class do not affect another. For example, just as a TV remote depends on a battery but isn't tied to a specific brand, classes should depend on abstractions rather than concrete implementations to achieve flexibility and maintainability.

**b.**

Lambda expressions and functional interfaces in Java are features introduced in Java 8 to support functional programming paradigms.

A functional interface is an interface that contains only one abstract method. This means it has a single functionality to exhibit. Java 8 onwards, lambda expressions can be used to represent instances of functional interfaces. These interfaces can also contain any number of default methods. Examples of functional interfaces include `Runnable`, `ActionListener`, and `Comparable`.

Functional interfaces are also known as Single Abstract Method (SAM) interfaces. They ensure that they include precisely one abstract method, although they can contain any number of default and static methods. They are marked with an annotation `@FunctionalInterface` to indicate that they are intended for use with lambda expressions.

Lambda expressions, on the other hand, are short blocks of code that can accept input parameters and return a value. They are essentially a concise way to express instances of functional interfaces. Lambda expressions enable treating functionality as a method argument or code as data, similar to lambda functions in other programming languages. They allow functions to be created without belonging to any class and can be passed around as if they were objects, enabling execution on demand.

In summary, lambda expressions and functional interfaces in Java facilitate functional programming by providing a way to express concise blocks of code (lambda expressions) and by defining interfaces (functional interfaces) that represent single functionality, making the code more readable, clean, and straightforward. They enable treating functions as first-class citizens, allowing for a more functional approach to programming in Java.

**c.**

Design patterns in Java are essential techniques or templates used by developers to solve common problems in software design effectively. They encapsulate proven solutions and best practices, making it easier for programmers to create maintainable, flexible, and understandable code. These patterns are not specific implementations but rather general strategies or models for addressing recurring challenges in software development.

One commonly used design pattern in Java is the Singleton pattern. This pattern ensures that a class has only one instance and provides a global point of access to that instance. This is useful in scenarios where only one instance of a class is needed throughout the application, such as a configuration manager or a logger.

Another example is the Factory pattern, which provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This promotes loose coupling and simplifies object creation, especially in complex systems where different types of objects need to be created based on specific conditions or configurations.

Additionally, the Observer pattern is widely used in Java to establish a one-to-many dependency between objects. In this pattern, an object, known as the subject, maintains a list of its dependents, known as observers, and notifies them of any changes in its state. This is useful in scenarios where multiple objects need to be notified and updated when the state of another object changes, such as in event-driven architectures or graphical user interfaces.

These are just a few examples of the many design patterns available in Java. By leveraging these patterns, developers can improve code quality, maintainability, and scalability while adhering to established best practices in software design.

**Question 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 not present in array");

} else {

System.out.println("Element found at index: " + result);

}

}

}

In this program:

* We define a `binarySearch` method that takes an array of integers `arr` and the target integer `target` to search for.
* Inside the `binarySearch` method, we initialize `left` to 0 and `right` to the last index of the array.
* We then enter a while loop that continues until `left` is less than or equal to `right`.
* Inside the loop, we calculate the `mid` index using `(left + right) / 2`.
* We compare the element at `arr[mid]` with the `target`. If they are equal, we return `mid`.
* If the `target` is greater than the element at `arr[mid]`, we update `left` to `mid + 1` to search in the right half of the array.
* If the `target` is smaller, we update `right` to `mid - 1` to search in the left half of the array.
* If the `target` is not found in the array, we return -1.
* In the `main` method, we define an array `arr` and a target integer `target`, and then call the `binarySearch` method to find the index of the target element in the array. We then print the result.

**Question 4:**

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));

}

}

In this program:

* We define methods `addMatrices`, `multiplyMatrices`, `transposeMatrix`, and `subtractMatrices` to perform addition, multiplication, transposition, and subtraction operations on matrices, respectively.
* Each method takes two matrices as input parameters and returns the resulting matrix after performing the respective operation.
* The `displayMatrix` method is used to print the elements of a matrix.
* In the `main` method, we define two sample matrices `matrix1` and `matrix2`, perform various matrix operations using the defined methods, and display the results.

**Question 5:**

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);

}

}

In this program:

* The `findUniqueInteger` method takes an array of integers `nums` as input.
* It initializes `uniqueInteger` to 0.
* It iterates through each element of the array and performs XOR operation with `uniqueInteger`. Since XOR of a number with itself is 0, and XOR of a number with 0 is the number itself, all the pairs cancel out each other, leaving the unique number.
* Finally, it returns the `uniqueInteger`.
* In the `main` method, we provide an example array `nums`, call the `findUniqueInteger` method to find the unique integer, and then print the result.