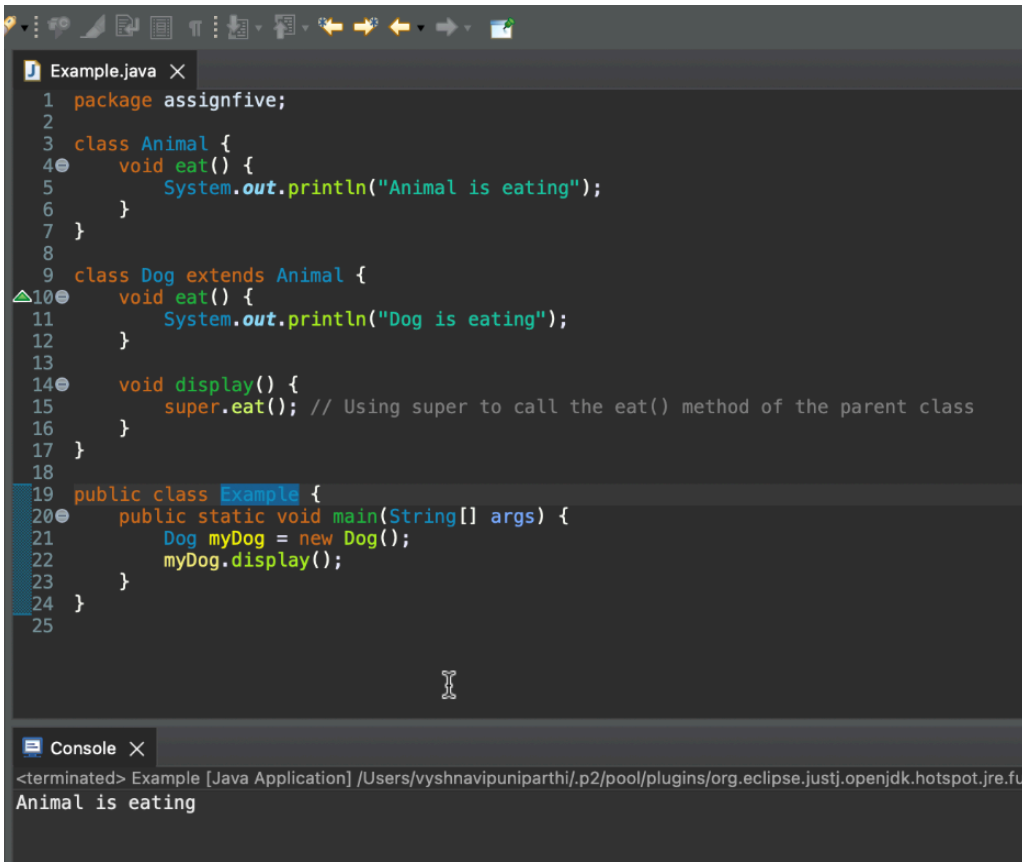
Vyshnavi Puniparthi

1. Explain in your own words:
a. Explain the difference between checked and unchecked exceptions in Java.

In Java, checked and unchecked exceptions are two types of exceptions that represent different error conditions. Checked exceptions are checked at compile time, that is, the compiler forces the programmer to handle them by catching an exception using a try-catch block or the "throws" keyword in a method signature. Unchecked objects typically extend the RuntimeException class.

b. What is the purpose of the "super" keyword in Java? Provide an example.
In Java, the "super" keyword is used to refer to an object in the immediate parent group. Typically this is used to invoke parent class methods, access parent class fields, or invoke the parent class constructor.

Examples of "super" keyword usage:



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

Generics in Java enable you to create classes, interfaces, and methods that work with parameterized types and provide compile-time type safety. They allow you to specify what type of objects a collection or class can contain or manipulate without having to explicitly dispose of them. Generics are useful for improving code reuse and maintainability, because they allow you to create flexible and type-safe code that can work with different types of data.

```
// Without generics
class Box {
    private Object value;

    public void setValue(Object value) {
        this.value = value;
    }

    public Object getValue() {
        return value;
    }
}

// With generics
class GenericBox<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

In the example, the GenericBox class with generics ensures type safety by allowing the type of data it holds at instantiation to be defined, while a non-generic Box class can contain objects of any type but without a type security

2. Explain in your own words

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

SOLID is an acronym that stands for five design principles that aim to make software design more logical, simple and maintainable. These principles are:

Single Responsibility Principle (SRP): A class can have only one reason to change, i.e. it can have only one responsibility or role.

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

Liskov Substitution Principle (LSP): Subtypes should be substitutable for their base type. This principle ensures that replacing objects in the superclass with objects in its subclass cannot affect the validity of the program.

Interface isolation principle (ISP): Subscribers are forced to rely on unused interfaces. Instead of having one big interface, break it down into smaller specific interfaces.

Dependency Inversion Principle (DIP): A higher-level module should not depend on a lower-level module; Both must be based on abstractions. Abstractions should not be based on details; Details may depend on abstractions. This principle encourages loose coupling between modules.

These principles are used to generate maintainable and scalable code in a Java framework. Example:

SRP: Each class should only have one responsibility. For example, the class that handles database operations may not even be able to handle user authentication.
OCP: Instead of changing existing laws, you expand them. For example, you can use interfaces and abstract classes to define a contract and provide functionality independently, allowing for easy extension without modifying existing code
LSP: Subtypes can potentially be substituted for their base type without changing according to the configuration. For example, smaller classes must obey the conventions defined by their larger class, so that they can use them interchangeably.
ISP: Interconnections should be structured and focused on specific related services. Consumers should not resort to unused channels. This ensures that the function class only needs to assign functions to the required methods.
DIP: Upstream modules must not be directly dependent on downstream modules; Instead, both must be based on abstractions. This forces decoupling and simplifies testing and maintenance.

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

Lambda Expressions: Introduced in Java 8, lambda expressions are a shortcut to represent anonymous functions. They provide a way to carry actions as arguments to a channel. Lambda expressions improve the readability and conciseness of the code by reducing the boilerplate code associated with anonymous classes.

Functional Interfaces: Functional Communication is an abstract form of communication. They act as a convention for lambda expressions or styles. Functional connectivity enables the operating system model to work in Java by allowing services to be treated as first-class citizens.

Lambda expressions and function interfaces simplify programming models:

Function structures are like high-level functions, which accept new functions as arguments or return functions as results to optimize them.
Providing a concise vocabulary for representing services, especially when working with collections, streams, and parallel systems.

Example:

```
// Functional Interface
@FunctionalInterface
interface MyFunction {
    int apply(int a, int b);
}

public class Example {
    public static void main(String[] args) {
        // Lambda Expression
        MyFunction add = (a, b) -> a + b;
        System.out.println(add.apply(3, 5)); // Output: 8
    }
}
```

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

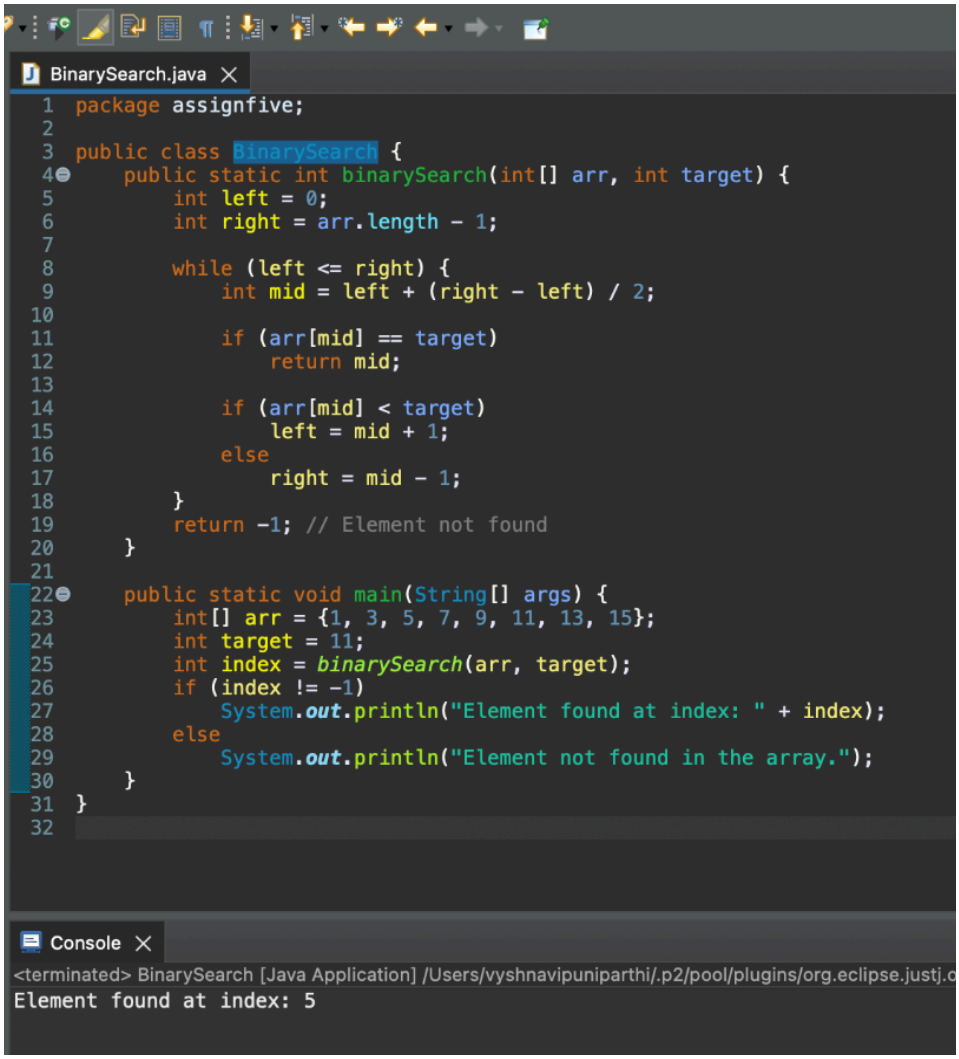
Design patterns are reusable solutions to common problems in software development. They provide templates for dealing with recurring system issues and promote best practices for building maintainable and scalable software systems. Design processes are important in Java development because:

Incorporate best practices: Design includes proven solutions to common design problems. Developers are provided with shared vocabulary to effectively communicate solutions.
Encourage code reuse: Designs allow developers to reuse solutions to common problems instead of reinventing the wheel. This saves time and reduces the possibility of error.
Increase solvability: Design helps develop code that provides clear and well-defined solutions to common problems that are easy to understand and maintain.
Scalability improvements: By following design processes, developers can create software systems that are scalable and easy to modify.

Some of the commonly used concepts in Java are:

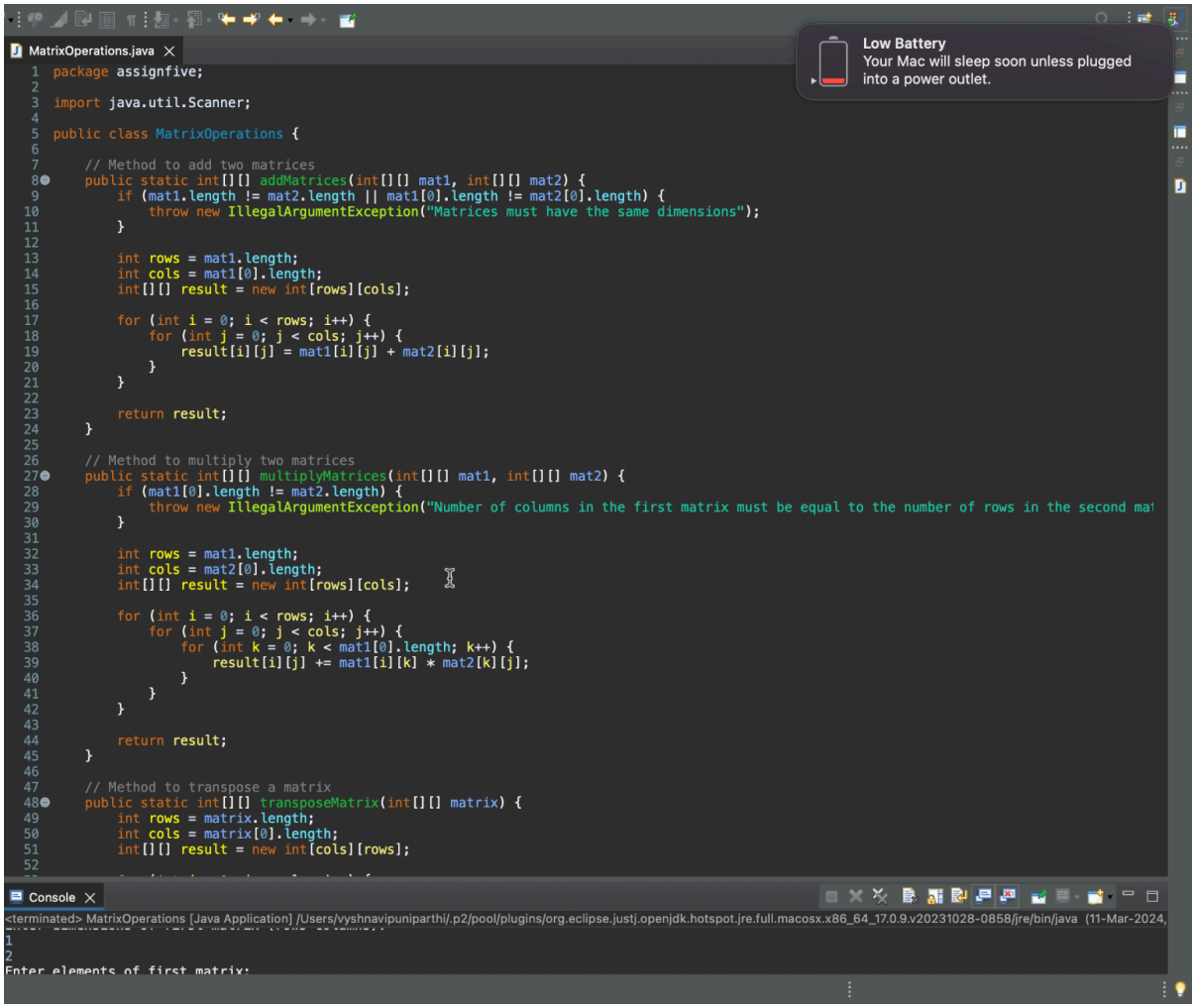
Singleton model
Factory sample
Abstract workspace model
Builder examples
adapter model
Supervisor examples
planning of operations
Decorator designs
Each of these models addresses a specific design problem and provides a systematic approach to solving common problems in software development.

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



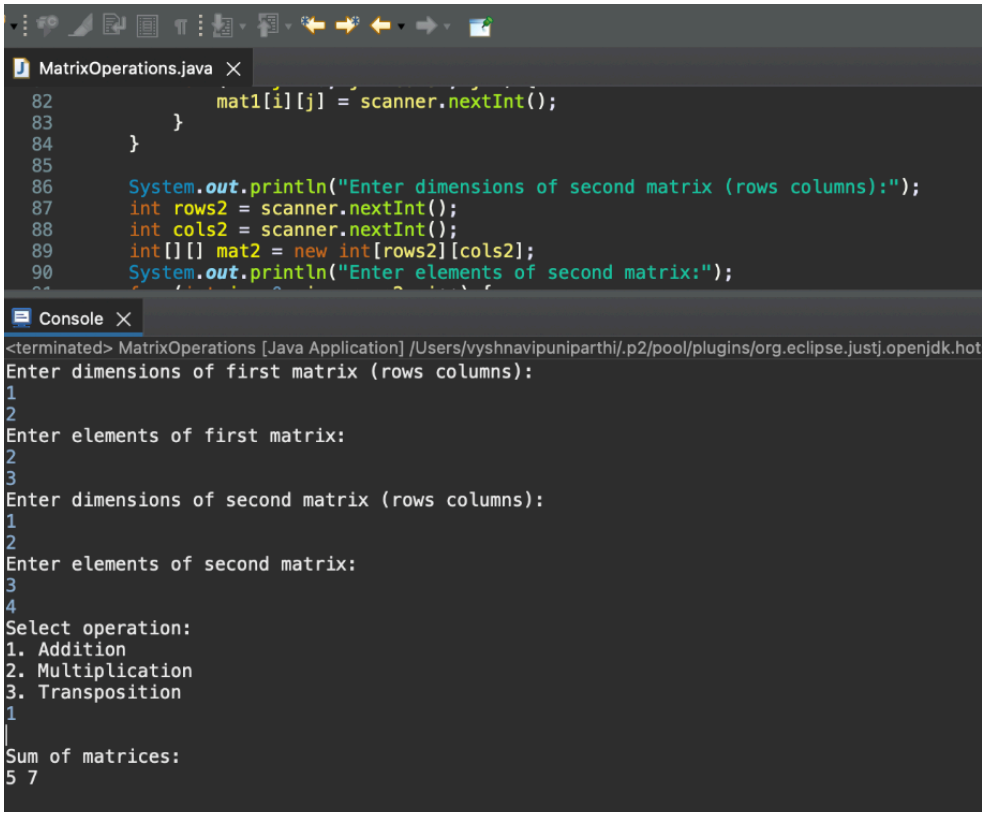
This Java program uses a binary search algorithm, which searches for a target element in a well-ordered array by repeatedly dividing the search interval in half The BinarySearch method takes an array and a target element as input and returns the target element of an array of the index occurs if it is found, or if If no element is present, algorithm -1 contains two left and right pointers representing the beginning and end of the search interval, respectively. It then divides the network two times and compares the central object in the array with the target object. If the middle element is equal to the value, the index is returned. If the central feature is smaller than the target, the search takes place in the right half of the interval, otherwise in the left half. This process repeats until the element is found or the search interval becomes empty.

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




MatrixOperations.java
}
// Method to transpose a matrix
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 < cols; i++) {
for (int j = 0; j < rows; j++) {
result[i][j] = matrix[j][i];
}
}
return result;
}
// Method to print a matrix
public static void printMatrix(int[][] matrix) {
for (int i = 0; i < matrix.length; i++) {
for (int j = 0; j < matrix[0].length; j++) {
System.out.print(matrix[i][j] + " ");
}
System.out.println();
}
}
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.println("Enter dimensions of first matrix (rows columns):");
int rows1 = scanner.nextInt();
int cols1 = scanner.nextInt();
int[][] mat1 = new int[rows1][cols1];
System.out.println("Enter elements of first matrix:");
for (int i = 0; i < rows1; i++) {
for (int j = 0; j < cols1; j++) {
mat1[i][j] = scanner.nextInt();
}
}
System.out.println("Enter dimensions of second matrix (rows columns):");
int rows2 = scanner.nextInt();
int cols2 = scanner.nextInt();
int[][] mat2 = new int[rows2][cols2];
System.out.println("Enter elements of second matrix:");
for (int i = 0; i < rows2; i++) {
for (int j = 0; j < cols2; j++) {
mat2[i][j] = scanner.nextInt();
}
}
Console
<terminated> MatrixOperations [Java Application] /Users/vyshnavipuniparthi/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jr
1
2
Enter elements of first matrix:


MatrixOperations.java
mat1[i][j] = scanner.nextInt();
}
}
System.out.println("Enter dimensions of second matrix (rows columns):");
int rows2 = scanner.nextInt();
int cols2 = scanner.nextInt();
int[][] mat2 = new int[rows2][cols2];
System.out.println("Enter elements of second matrix:");
for (int i = 0; i < rows2; i++) {
for (int j = 0; j < cols2; j++) {
mat2[i][j] = scanner.nextInt();
}
}
System.out.println("Select operation: ");
System.out.println("1. Addition");
System.out.println("2. Multiplication");
System.out.println("3. Transposition");
int choice = scanner.nextInt();
switch (choice) {
case 1:
try {
int[][] sum = addMatrices(mat1, mat2);
System.out.println("\nSum of matrices:");
printMatrix(sum);
} catch (IllegalArgumentException e) {
System.out.println("Error: " + e.getMessage());
}
break;
case 2:
try {
int[][] product = multiplyMatrices(mat1, mat2);
System.out.println("\nProduct of matrices:");
printMatrix(product);
} catch (IllegalArgumentException e) {
System.out.println("Error: " + e.getMessage());
}
break;
case 3:
System.out.println("\nTranspose of first matrix:");
int[][] transposed = transposeMatrix(mat1);
printMatrix(transposed);
break;
default:
System.out.println("Invalid choice");
}
scanner.close();
}
Console
<terminated> MatrixOperations [Java Application] /Users/vyshnavipuniparthi/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.ful
1
2
Enter elements of first matrix:



This Java program allows users to insert the dimensions and elements of two matrices and choose the operation (addition, multiplication, or transformation) to perform. Ensures data integrity by validating assumptions prior to execution, providing clear error messages when needed. Leveraging user input and method calls, the program demonstrates matrix manipulation functionality, promoting interactive and dynamic matrix functionality in a console-based environment

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

This function first prompts the user to input the number of items in the array and then allows the user to input the items in the array. The findUniqueInteger method is then called, which uses a HashMap to count every number of occurrences in the array. Finally, it goes back through the HashMap to find a unique integer (a number whose count is 1) and returns it. If no unique integer is found, the method returns -1. The main method displays the results to the user.

UniqueIntegerFinder.java

```java
package assignfive;

import java.util.HashMap;

public class UniqueIntegerFinder {

    public static int findUniqueInteger(int[] nums) {
        Map<Integer, Integer> countMap = new HashMap<>();

        // Count occurrences of each number
        for (int num : nums) {
            countMap.put(num, countMap.getOrDefault(num, 0) + 1);
        }

        // Find the unique integer
        for (Map.Entry<Integer, Integer> entry : countMap.entrySet()) {
            if (entry.getValue() == 1) {
                return entry.getKey();
            }
        }

        // If no unique integer found, return -1
        return -1;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of elements in the array: ");
        int n = scanner.nextInt();

        int[] nums = new int[n];
        System.out.println("Enter the elements of the array:");
        for (int i = 0; i < n; i++) {
            nums[i] = scanner.nextInt();
        }

        int uniqueInteger = findUniqueInteger(nums);
        if (uniqueInteger != -1) {
            System.out.println("The unique integer is: " + uniqueInteger);
        } else {
            System.out.println("No unique integer found in the array.");
        }
        scanner.close();
    }
}
```

Console

```
<terminated> UniqueIntegerFinder [Java Application] /Users/vyshnavipuniparthi/.p2/pool/plugins/org.eclipse.justj.oper
Enter the number of elements in the array: 5
Enter the elements of the array:
4
3
2
4
2
The unique integer is: 3
```