Task No 11

Question No 1

In Java, there are four access modifiers that control the accessibility of classes, methods, and variables. These modifiers determine which parts of your code can access certain members of a class. These are the four access modifiers and their significance:

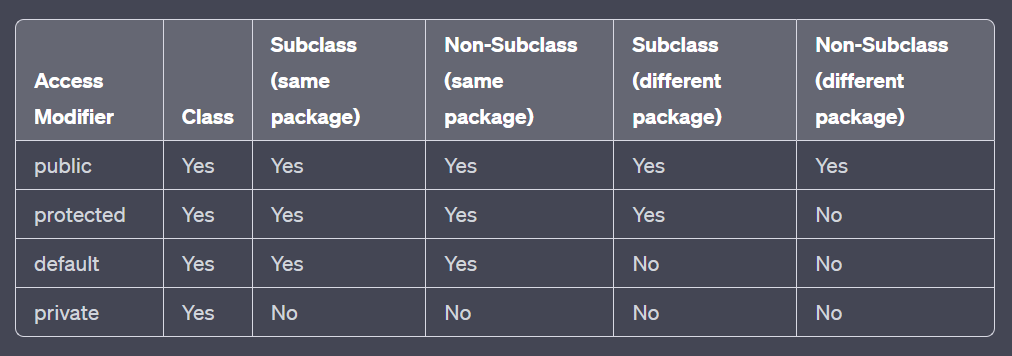
public: The public access modifier makes a class, method, or variable accessible from anywhere in the program. It has the widest accessibility. When a class is declared as public, it can be accessed from any other class in the same or different packages. Similarly, when a method or variable is declared as public, it can be accessed from any class.

protected: The protected access modifier allows access within the same class, subclasses (even if they are in different packages), and other classes in the same package. It is typically used to provide controlled access to specific members.

default : If no access modifier is specified, the default access modifier is applied. This makes the class, method, or variable accessible only within the same package. It restricts access from classes in different packages.

private: The private access modifier restricts the accessibility to within the same class. It is commonly used to encapsulate and hide the internal details of a class from other classes, providing data hiding and security.

Summary of the accessibility of the members based on their access modifiers:



Question No 2

In Java, both exceptions and errors are subclasses of the Throwable class, but they serve different purposes and have distinct characteristics:

Exceptions:

Exceptions in Java are used to indicate exceptional conditions that can occur during the execution of a program but can be handled and recovered from. Exceptions are further divided into two categories:

a. Checked Exceptions: These are the exceptions that are checked at compile-time. Methods that may throw checked exceptions must declare them in their method signature using the throws keyword, or they must handle the exceptions using try-catch blocks. Examples of checked exceptions include IOException, SQLException, etc.

b. Unchecked Exceptions (RuntimeExceptions): These exceptions are not checked at compile-time, and the compiler doesn't enforce the handling of these exceptions. Unchecked exceptions usually indicate programming errors or unexpected conditions that the program may or may not recover from. Examples include NullPointerException, ArrayIndexOutOfBoundsException, etc.

Errors:

Errors, on the other hand, indicate serious problems that are usually beyond the control of the application. They are typically caused by external factors or system-level issues and are not meant to be caught or handled by the application. Errors usually result from issues that might be irrecoverable, such as running out of system resources or hardware failures.

Common examples of errors include OutOfMemoryError, StackOverflowError, etc. Since errors are usually severe and uncontrollable, it's generally not recommended to try to catch or handle them. Instead, it's best to focus on preventing them through good programming practices and proper system configuration.

In summary, exceptions are used to handle exceptional conditions that can be recovered from, while errors represent serious and often unrecoverable problems in the application or the system. Handling exceptions is an essential part of writing robust Java code, whereas dealing with errors is generally out of the scope of regular application-level error handling.

Question No 3

In Java, exceptions are categorized into two main types: checked exceptions and unchecked exceptions (also known as runtime exceptions). The key difference between the two lies in whether they are checked at compile-time or not:

Checked Exceptions:

Checked exceptions are exceptions that are checked by the compiler at compile-time. When a method can throw a checked exception, the compiler enforces that the exception must be either caught using a try-catch block or declared to be thrown in the method signature using the throws keyword. This ensures that the calling code is aware of the potential exceptions that can be thrown and takes appropriate action to handle them.

Unchecked Exceptions (Runtime Exceptions):

Unchecked exceptions, also known as runtime exceptions, are not checked at compile-time. The compiler does not enforce that these exceptions be caught or declared to be thrown. Unchecked exceptions are typically caused by programming errors or unexpected conditions that the program may or may not recover from. These exceptions are used for situations that are not easily recoverable and may indicate a bug or logic flaw in the program. Common examples include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.

Checked exceptions are used for exceptional conditions that a method or code block can anticipate and recover from. Common examples include file I/O errors (IOException), database access errors (SQLException), and network-related issues.

In summary, checked exceptions are checked at compile-time and must be handled or declared in the method signature, while unchecked exceptions (runtime exceptions) are not checked at compile-time and are often indicative of programming errors or unexpected conditions. It is generally recommended to use checked exceptions for recoverable exceptional conditions and unchecked exceptions for cases where recovery is not possible or unlikely.

Question No 4

import java.util.Scanner;  
  
public class DivisionProgram {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 try {  
 System.*out*.print("Enter the first integer: ");  
 int numerator = scanner.nextInt();  
  
 System.*out*.print("Enter the second integer: ");  
 int denominator = scanner.nextInt();  
  
 int result = *divideNumbers*(numerator, denominator);  
 System.*out*.println("Result of division: " + result);  
 } catch (ArithmeticException e) {  
 System.*err*.println("Error: Cannot divide by zero.");  
 } catch (Exception e) {  
 System.*err*.println("Error: Invalid input or other exception occurred.");  
 } finally {  
 scanner.close();  
 }  
 }  
  
 public static int divideNumbers(int numerator, int denominator) {  
 if (denominator == 0) {  
 throw new ArithmeticException("Divisor cannot be zero.");  
 }  
 return numerator / denominator;  
 }  
}

Question No 5

public class ArrayIndexOutOfBoundsExample {  
 public static void main(String[] args) {  
 int[] numbers = {1, 2, 3};  
  
 try {  
 // Trying to access the element at index 3, which is out of bounds.  
 int element = numbers[3];  
 } catch (ArrayIndexOutOfBoundsException e) {  
 System.*err*.println("Error: ArrayIndexOutOfBoundsException occurred.");  
 }  
 }  
}

public class StringIndexOutOfBoundsExample {  
 public static void main(String[] args) {  
 String text = "Hello, World!";  
  
 try {  
 // Trying to access the character at index 15, which is out of bounds.  
 char character = text.charAt(15);  
 } catch (StringIndexOutOfBoundsException e) {  
 System.*err*.println("Error: StringIndexOutOfBoundsException occurred.");  
 }  
 }  
}

Question No 6

import java.util.Scanner;  
  
public class LoginSystem {  
 private static final String *CORRECT\_PASSWORD* = "BetweenTwoCoconutsTrees";  
  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 System.*out*.print("Enter your username: ");  
 String username = scanner.nextLine();  
  
 System.*out*.print("Enter your password: ");  
 String password = scanner.nextLine();  
  
 try {  
 *checkPassword*(username, password);  
 System.*out*.println("Login successful! Welcome, " + username + ".");  
 } catch (IncorrectPasswordException e) {  
 System.*err*.println("Error: Incorrect password entered.");  
 } catch (UserNotFoundException e) {  
 System.*err*.println("Error: User not found.");  
 } finally {  
 scanner.close();  
 }  
 }  
  
 public static void checkPassword(String username, String password) throws IncorrectPasswordException, UserNotFoundException {  
   
 if (!username.equals("Vinayak")) {  
 throw new UserNotFoundException();  
 }  
  
 // Check if the password matches the correct password  
 if (!password.equals(*CORRECT\_PASSWORD*)) {  
 throw new IncorrectPasswordException();  
 }  
 }  
}  
  
class IncorrectPasswordException extends Exception {  
 // Custom exception class for incorrect password  
}  
  
class UserNotFoundException extends Exception {  
 // Custom exception class for user not found  
}

Question No 7

import java.util.Scanner;  
  
public class AgeValidation {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 try {  
 System.*out*.print("Enter your age: ");  
 int age = scanner.nextInt();  
  
 *validateAge*(age);  
  
 System.*out*.println("Age is valid. Welcome!");  
 } catch (InvalidAgeException e) {  
 System.*err*.println("Error: " + e.getMessage());  
 } catch (Exception e) {  
 System.*err*.println("Error: Invalid input or other exception occurred.");  
 } finally {  
 scanner.close();  
 }  
 }  
  
 public static void validateAge(int age) throws InvalidAgeException {  
 if (age < 18) {  
 throw new InvalidAgeException("Age must be at least 18 years.");  
 }  
 }  
}  
  
class InvalidAgeException extends Exception {  
 public InvalidAgeException(String message) {  
 super(message);  
 }  
}

Question No 8

import java.io.File;  
import java.io.FileNotFoundException;  
import java.util.Scanner;  
  
public class ReadFileExample {  
 public static void main(String[] args) {  
 try {  
 // Replace "example.txt" with the name of your file  
 String filename = "example.txt";  
 *readFileData*(filename);  
 } catch (FileNotFoundException e) {  
 System.*err*.println("Error: File not found. " + e.getMessage());  
 } catch (Exception e) {  
 System.*err*.println("Error: Something went wrong. " + e.getMessage());  
 }  
 }  
  
 public static void readFileData(String filename) throws FileNotFoundException {  
 File file = new File(filename);  
  
 if (!file.exists()) {  
 throw new FileNotFoundException("File '" + filename + "' not found.");  
 }  
  
 try (Scanner scanner = new Scanner(file)) {  
 // Read and process data from the file  
 while (scanner.hasNextLine()) {  
 String line = scanner.nextLine();  
 // Process the line as needed  
 System.*out*.println(line);  
 }  
 }  
 }  
}