**Task-11 Java Exceptions and Error Handling**

**Qus-1 What Are the Four Access Modifiers Available in Java and What Is Their Significance in Terms of Class, Method, And Variable Accessibility?**

In Java, access modifiers are keywords that control the accessibility of classes, methods, and variables within a program. By specifying an access modifier, you can define the level of access that other classes or code within the same package or different packages have to the class, method, or variable. Understanding these access modifiers is crucial for designing well-structured and maintainable Java applications. In this blog post, we will explore the four access modifiers available in Java - public, protected, default (package-private), and private - and discuss their significance in terms of class, method, and variable accessibility.

1. Public:

The public access modifier is the most permissive access level. Classes, methods, and variables marked as public can be accessed from any other class or package. Here's a summary of its significance:

Class: A public class can be accessed from any other class or package. It serves as an entry point to the application and provides a clear interface for other components.

Method: A public method can be invoked from any other class or package. It exposes functionality to other parts of the program and facilitates code reuse.

Variable: A public variable can be accessed and modified from any other class or package. It provides direct access to the state of an object, which can be useful in certain scenarios but may also lead to tight coupling between components.

2. Protected:

The protected access modifier allows access to classes, methods, and variables within the same package or subclasses, even if they are in a different package. Here's how it impacts accessibility:

Class: A protected class is accessible within the same package or by subclasses, promoting encapsulation and inheritance.

Method: A protected method can be invoked within the same package or by subclasses, enabling subclass customization and method overriding.

Variable: A protected variable can be accessed and modified within the same package or by subclasses, providing controlled access to the state of an object.

3. Default (Package-Private):

When no access modifier is specified, the default access modifier is applied. Classes, methods, and variables with default access can only be accessed within the same package. Here's what it means for accessibility:

Class: A class with default access is accessible only within the same package, promoting encapsulation and package-level encapsulation.

Method: A method with default access can be invoked within the same package, facilitating communication between closely related classes.

Variable: A variable with default access can be accessed and modified within the same package, ensuring data integrity and controlled access.

4. Private:

The private access modifier is the most restrictive access level. Classes, methods, and variables marked as private can only be accessed within the same class. Here's its significance:

**Class:** A private class is accessible only within the same class, enforcing strong encapsulation and information hiding.

**Method:** A private method can only be invoked within the same class, encapsulating implementation details and promoting modular design.

**Variable:** A private variable can only be accessed and modified within the same class, protecting the internal state of an object from external interference.

In conclusion, access modifiers play a crucial role in controlling the accessibility and visibility of classes, methods, and variables in Java. By choosing the appropriate access level for each component, you can enforce encapsulation, promote code reuse, and enhance the maintainability and security of your Java applications. Understanding the significance of public, protected, default, and private access modifiers is essential for writing well-designed and robust Java code.

**Qus-2.what is difference between error and exception?**

In Java, both errors and exceptions are types of problems that can occur during the execution of a program, but they have distinct characteristics and implications. Understanding the difference between errors and exceptions is crucial for building robust and reliable Java applications. In this blog post, we'll explore the definitions, causes, handling, and implications of errors and exceptions in Java.

Errors:

Errors in Java represent serious problems that are typically beyond the control of the programmer or the Java Virtual Machine (JVM). They indicate critical issues that usually cannot be recovered from. Here are some key points about errors:

**Cause:** Errors are often caused by external factors such as hardware failures, system crashes, or resource exhaustion. They may also result from fundamental flaws in the JVM itself.

**Examples:** Common examples of errors in Java include OutOfMemoryError, StackOverflowError, and VirtualMachineError.

**Handling:** Errors are unchecked exceptions, meaning they are not meant to be caught or handled by the program. Attempting to catch or handle errors is generally not recommended because they typically indicate severe problems that require attention at the system level.

**Implications:** Errors indicate critical failures that may render the application unusable. They often require intervention at the system level, such as increasing memory allocation or fixing hardware issues.

Exceptions:

Exceptions in Java represent exceptional conditions that occur during the execution of a program and disrupt the normal flow of control. They can occur due to various reasons such as invalid input, network issues, file I/O errors, or arithmetic errors. Here are some key points about exceptions:

Cause: Exceptions can be caused by both external factors and programming errors. They may result from unexpected conditions that are beyond the control of the programmer, such as network failures or invalid user input.

Examples: Exceptions in Java are classified into two main types: checked exceptions and unchecked exceptions. Checked exceptions, such as IOException and SQLException, are checked at compile time and must be caught or declared. Unchecked exceptions, such as NullPointerException and ArrayIndexOutOfBoundsException, are not checked at compile time and do not require catching or declaring.

Handling: Exceptions can be caught and handled using try-catch blocks or propagated up the call stack using the throws keyword. Handling exceptions allows the program to gracefully recover from exceptional conditions and continue its execution or provide appropriate error messages to the user.

Implications: Exceptions disrupt the normal flow of control but can often be recovered from by handling them appropriately. Proper exception handling is essential for ensuring the robustness and reliability of Java applications.

Conclusion:

In summary, err ors and exceptions in Java are both types of problems that can occur during the execution of a program, but they have different characteristics and handling requirements. Errors represent critical failures that are typically beyond the control of the programmer or the JVM and are not meant to be caught or handled. Exceptions, on the other hand, represent exceptional conditions that disrupt the normal flow of control but can often be recovered from by properly handling them. Understanding the difference between errors and exceptions is essential for building robust and reliable Java applications.

**Qus-3 What is the Difference between checked exceptions and un checked exceptions?**

In Java, exceptions play a critical role in handling unexpected conditions that may arise during the execution of a program. Two main categories of exceptions exist: checked exceptions and unchecked exceptions. Understanding the difference between these two types of exceptions is crucial for writing robust and reliable Java code. In this blog post, we will delve into the distinctions between checked and unchecked exceptions, including their definitions, characteristics, handling requirements, and common use cases.

**Checked Exceptions:**

Checked exceptions, also known as compile-time exceptions, are exceptions that must be either caught and handled or declared in the method signature using the **throws** keyword. These exceptions are checked by the compiler at compile-time to ensure that they are properly handled. Here are some key points about checked exceptions:

**Unchecked Exceptions:**

Unchecked exceptions, also known as runtime exceptions, are exceptions that are not checked by the compiler at compile-time. These exceptions can occur at runtime and are not required to be caught or declared by the programmer. Here are some key points about unchecked exceptions:

|  |  |
| --- | --- |
| **Checked exceptions** | **Unchecked exceptions** |
| **Definition**: Checked exceptions represent expected conditions that a program may encounter during its execution, such as file I/O errors, database connectivity issues, or network failures. | **Definition**: Unchecked exceptions represent programming errors or unexpected conditions that are beyond the control of the programmer, such as null pointer dereferences, array index out of bounds, or arithmetic errors. |
| **Handling Requirements**: Checked exceptions must be either caught and handled using try-catch blocks or declared in the method signature using the **throws** keyword. If a method throws a checked exception, the caller of the method must handle the exception or propagate it up the call stack. | **Handling Requirements**: Unchecked exceptions are not required to be caught or declared by the programmer. While it is possible to catch and handle unchecked exceptions, it is not mandatory. Unchecked exceptions can propagate up the call stack until they are caught or result in the termination of the program. |
| **Examples**: Common examples of checked exceptions include **IOException**, **SQLException**, and **ClassNotFoundException**. | **Examples**: Common examples of unchecked exceptions include **NullPointerException**, **ArrayIndexOutOfBoundsException**, **ArithmeticException**, and **IllegalArgumentException**. |

**Conclusion:**

In summary, checked and unchecked exceptions in Java serve different purposes and have distinct characteristics and handling requirements. Checked exceptions are checked by the compiler at compile-time and must be either caught and handled or declared in the method signature. They are typically used for recoverable conditions where the program can take corrective action. Unchecked exceptions, on the other hand, are not checked by the compiler and are not required to be caught or declared. They represent programming errors or unexpected conditions that may arise at runtime. Understanding the difference between checked and unchecked exceptions is essential for writing robust and reliable Java code that handles exceptions effectively.

Top of Form

Top of Form

Top of Form

**Qus-4 write a java program that reads user input for two integers and perform divisions. Handle the exception that is thrown when the second number is zero, and display an error message to the user?**

import java.util.Scanner;

public class DivisionProgram {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the first integer: ");

int num1 = scanner.nextInt();

System.out.print("Enter the second integer: ");

int num2 = scanner.nextInt();

try {

int result = num1 / num2;

System.out.println("Result of division: " + result);

} catch (ArithmeticException e) {

System.out.println("Error: Division by zero is not allowed.");

}

scanner.close();

}

**}**

**Qus-5 write the code of ArrayIndexOutOfBoundsException?**

public class ArrayIndexOutOfBoundsExceptionExample {

public static void main(String[] args) {

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

// Trying to access an index beyond the array size

try {

int index = 10;

int value = numbers[index];

System.out.println("Value at index " + index + ": " + value);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Error: Array index out of bounds!");

}

}

**write the code of StringIndexOutOfBoundsException?**

public class StringIndexOutOfBoundsExceptionExample {

public static void main(String[] args) {

String str = "Hello";

// Trying to access a character at an index beyond the string length

try {

char ch = str.charAt(10);

System.out.println("Character at index 10: " + ch);

} catch (StringIndexOutOfBoundsException e) {

System.out.println("Error: String index out of bounds!");

}

}

}

**Qus-6 Building a login System using Exception handling**

package exceptions;

import java.util.Scanner;

public class LoginException extends Exception {

//Java treats custom exception as checked exception category

//To set the title/message for the custom exception

public LoginException(String expceptionMessage){

super(expceptionMessage);

}

public LoginException(){

}

public static void login() throws LoginException{

LoginException loginException = new LoginException("Invalid User Entry");

System.out.println("Please enter your user name ");

Scanner scanner = new Scanner(System.in);

String teamName = scanner.nextLine();

if(teamName.equals("CSK")){

System.out.println("Welcome to the Den ");

}

else{

//throw new LoginException("Invalid User Entry");

throw loginException;

}

}

public static void main(String []args) throws LoginException {

try {

login();

throw new LoginException("Invalid User Entry2");

} catch (LoginException e) {

e.printStackTrace();

System.out.println(e.getMessage());

System.out.println(e.getClass());

}

}

}

**Qus-7 InvalidAgeException using Exception Handling**

package exceptions;

import java.util.Scanner;

public class InvalidAgeException extends Exception {

public InvalidAgeException (String expceptionMessage){

super(expceptionMessage);

}

public InvalidAgeException (){

}

public static void login() throws InvalidAgeException {

InvalidAgeException invalidageException = new invalidageException ("User Age Above 18 is invalid ");

System.out.println("Please enter your age ");

Scanner scanner = new Scanner(System.in);

String teamName = scanner.nextLine();

if(teamName.equals("UserAge")){

System.out.println("Please enter to the next level ");

}

else{

//throw new LoginException("Invalid User Age");

throw invalidageException;

}

}

public static void main(String []args) throws invalidageException {

try {

invalidage ();

throw new InvalidAgeException ("User Age Above 18 is invalid 2 ");

} catch (InvalidAgeException e) {

e.printStackTrace();

System.out.println(e.getMessage());

System.out.println(e.getClass());

}

}

}

Qus-8 FileNotFound using Exception handling

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class FileReadingExample {

public static void main(String[] args) {

try {

File file = new File("input.txt");

Scanner scanner = new Scanner(file);

while (scanner.hasNextLine()) {

String line = scanner.nextLine();

System.out.println(line);

}

scanner.close();

} catch (FileNotFoundException e) {

// Handle FileNotFoundException

System.out.println("Error: File not found!");

e.printStackTrace();

}

}

}