**Assignment: (Core Java)**

**12 : Exception Handling**

**Que.1 Types of Exceptions: Checked and Unchecked**

**Ans.1** An exception is an unexpected event that occurs during program execution, disrupting the normal flow of the program. In Java, exceptions are broadly categorized into two types: Checked Exceptions and Unchecked Exceptions. The primary distinction lies in how the Java compiler and the Java Virtual Machine (JVM) handle them.

**Checked Exceptions:** These are exceptions that the Java compiler checks at compile time. They represent conditions that a well-written application should anticipate and handle.

**Examples:**

* IOException → Input/output errors
* FileNotFoundException → File does not exist
* SQLException → Database errors
* ClassNotFoundException → Class not found

**Unchecked Exceptions:** These are exceptions that are not checked by the compiler at compile time. They typically represent programming errors or logical flaws within the code that should ideally be prevented during development.

**Examples**

* ArithmeticException → Division by zero
* NullPointerException → Accessing a null object
* ArrayIndexOutOfBoundsException → Array index invalid
* NumberFormatException → Invalid type conversion

**Que.2 try, catch, finally, throw, throws**

**Ans.2** In Java, try, catch, finally, throw, and throws are keywords used for exception handling, a mechanism to manage runtime errors and ensure program robustness.

**try:**

* Used to wrap code that might throw an exception.
* Must be followed by at least one catch or finally block.

try {

int result = 10 / 0; // may throw ArithmeticException

}

**catch**

* + - Used to **handle exceptions thrown in try block**.
    - Can have **multiple catch blocks** for different exceptions.

try {

int result = 10 / 0;

}

catch (ArithmeticException e) {

System.out.println("Cannot divide by zero!");

}

catch (Exception e) {

System.out.println("Some other exception occurred.");

}

**Finally**

* + - **Optional block executed always, whether exception occurs or not.**
    - **Typically used for cleanup (closing files, DB connections, scanners).**

try {

int result = 10 / 2;

}

catch (ArithmeticException e) {

System.out.println("Error!");

}

finally {

System.out.println("Finally block executed.");

}

**throw**

* Used to explicitly throw an exception.
* Can throw either built-in or custom exceptions.

int age = 15;

if (age < 18) {

throw new ArithmeticException("Age must be 18 or above!");

}

**throws**

* Used in a method declaration to declare exceptions that the method might throw.
* Tells the caller of the method that it should handle or propagate the exception.

public static void divide(int a, int b) throws ArithmeticException {

int result = a / b;

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

}

public static void main(String[] args) {

try {

divide(10, 0);

}

catch (ArithmeticException e) {

System.out.println("Cannot divide by zero!");

}

}

**Que.3 Custom Exception Classes**

**Ans.3** A custom exception is a user-defined exception created to handle specific application requirements.

* Extends either:
  + Exception → Checked exception (must be handled or declared)
  + RuntimeException → Unchecked exception (optional to handle)
* Helps make code more readable, maintainable, and meaningful.

// Checked Exception

class MyException extends Exception {

public MyException(String message) {

super(message); // Pass the message to Exception class

}

}

// Unchecked Exception

class MyRuntimeException extends RuntimeException {

public MyRuntimeException(String message) {

super(message);

}

}