

## Assignment No:-36

Name:-Suryawanshi Sangramsingh Sambhaji

Batch: - Delta - DCA (Java) 2024     Date:-26/6/2024

**Q1. WAP to create the custom exception to throw the message if user is not eligible for blood donation, message will be you're not eligible for blood donation.**

```
package Assignment37;

import java.util.Scanner;

class NotEligibleBloodException extends Exception{
    NotEligibleBloodException(String msg){
        System.out.println(msg);
    }
}

public class UserdefinedExceptionBlood {
    void method() throws NotEligibleBloodException{
        Scanner s=new Scanner(System.in);
        System.out.println("Enter the age:");
        int age=s.nextInt();
        System.out.println("Enter the weight:");
        int weight=s.nextInt();
        if(age>=18 && weight>=60){
            System.out.println("Eligible for blood donate");
        }else {
            throw new NotEligibleBloodException("Exception..");
        }
    }

    public static void main(String[] args) {
        UserdefinedExceptionBlood obj=new UserdefinedExceptionBlood();
        try {
            obj.method();
        }catch(NotEligibleBloodException e) {
            System.out.println(e);
        }
    }
}
```

Output:

```
Console X
<terminated> UserdefinedExceptionBlood [Java Application] C:\Program
Enter the age:
21
Enter the waight:
57
Exception..
Assignment37.NotEligibleBloodException
```

Q2. Write a Java program that simulates a bank account. Create a custom exception class called "InsufficientBalanceException" and raise it whenever a withdrawal is attempted with an amount higher than the account balance. Handle the custom exception and display an error message

```
à package Assignment37;

import java.util.Scanner;
class InsufficientBalanceException extends Exception{
    InsufficientBalanceException(String msg){
        System.out.println(msg);
    }
}
public class banckBalance {
    void method() throws InsufficientBalanceException{
        Scanner s=new Scanner(System.in);
        System.out.println("Enter the Deposit amount:");
        int n1=s.nextInt();
        System.out.println("You want to withdraw your Amount:(Yes/No)");
        String str=s.next();
        if(str.equals("yes")) {
            System.out.println("Enter Amount:");
            int n2=s.nextInt();
            if(n2>n1) {
                throw new InsufficientBalanceException("InsufficientBalance..");
            } else {
                System.out.println("Withdraw Successfull!!!");
            }
        }
    }
}
```

```

    }
}

}

public static void main(String[] args) {
    banckBalance obj=new banckBalance();
    try {
        obj.method();
    } catch (InsufficientBalanceException e) {
        System.out.println(e);
    }
}
}

```

Output:

```

Console X
<terminated> banckBalance [Java Application] C:\Program Files\Java\jdk-20\bin\java.exe
Enter the Deposit amount:
10000
You want to withdraw your Amount: (Yes/No)
yes
Enter Amount:
11000
InsufficientBalance..
Assignment37.InsufficientBalanceException

```

**Q3 Write a Java program that calculates the area of a rectangle.**

The program should take the length and width of the rectangle as input from the user. If either the length

or width entered by the user is negative, throw a custom exception called NegativeDimensionException.

The NegativeDimensionException should be a checked exception that includes an appropriate error message.

```

package Assignment37;

import java.util.Scanner;
class NegativeDimensionException extends Exception{
    NegativeDimensionException(String msg){
        System.out.println(msg);
    }
}

public class lentghErrorException {
    void method() throws NegativeDimensionException{
        Scanner s=new Scanner(System.in);
    }
}

```

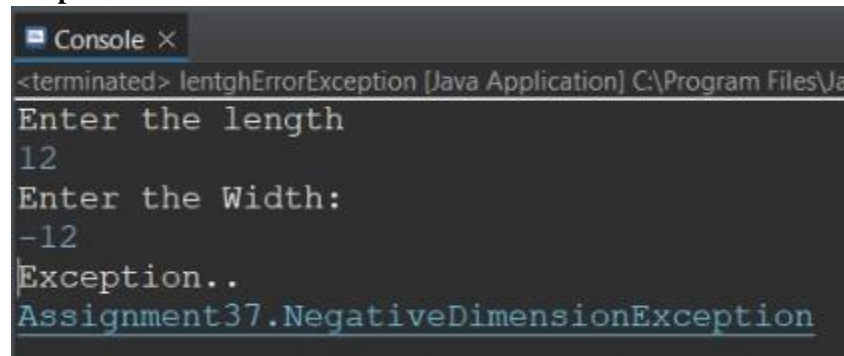
```

        System.out.println("Enter the length");
        int n1=s.nextInt();
        System.out.println("Enter the Width:");
        int n2=s.nextInt();
        if(n1<0 || n2<0) {
            throw new NegativeDimensionException("Exception..");
        }else {
            System.out.println(n1*n2);
        }
    }
}

public static void main(String[] args) {
    lentghErrorException obj=new lentghErrorException();
    try {
        obj.method();
    }catch(NegativeDimensionException e) {
        System.out.println(e);
    }
}
}
}

```

**Output:**



```

Console x
<terminated> lentghErrorException [Java Application] C:\Program Files\Ja
Enter the length
12
Enter the Width:
-12
Exception..
Assignment37.NegativeDimensionException

```

**Q4 Create a user authentication system with a method that checks if a user's credentials (e.g., username and password) are valid. If the credentials are incorrect, throw a custom exception AuthenticationException with a message indicating authentication failure.**

```

package UserdefinedException;

import java.util.Arrays;
import java.util.Scanner;

class AuthenticationLockoutException extends Exception{
    AuthenticationLockoutException(String msg){

```

```

        System.out.println(msg);
    }
}

public class UserLockLoginAttempt {
    void method() throws AuthenticationLockoutException{
        int Attempt=3;
        while(Attempt!=0) {
            Scanner s=new Scanner(System.in);
            System.out.println("*****USER LOGIN*****");
            System.out.println("UserName-->");
            String str=s.next();
            System.out.println("PassWord-->");
            int pass=s.nextInt();
            if(str.equals("om") && pass==1234 && Attempt!=0) {
                System.out.println("Login Successfull!!!");
                Attempt=0;
            }else {
                Attempt--;
                System.out.println("You Have Only "+Attempt+" Attempt");
                System.out.println("*****\n");
                if(Attempt==0) {
                    throw new AuthenticationLockoutException("Attempt zero!!!");
                }
            }
        }
    }

    public static void main(String[] args) {
        UserLockLoginAttempt obj=new UserLockLoginAttempt();
        try {
            obj.method();
        } catch (AuthenticationLockoutException e) {
            System.out.println(e);
        }
    }
}

```

**Output:**

```
Console x
<terminated> UserLockLoginAttempt [Java Application] C:\Program Files\Java\jdk-20\bin\javaw
*****USER LOGIN*****
UserName-->
aditya
PassWord-->
1234
You Have Only 2 Attempt
*****

*****USER LOGIN*****
UserName-->
sangram
PassWord-->
1234
You Have Only 1 Attempt
*****

*****USER LOGIN*****
UserName-->
harsh
PassWord-->
1234
You Have Only 0 Attempt
*****

Attempt zero!!!
UserdefinedException.AuthenticationLockoutException
```

#### Question 5:

Write a Java program that asks the user to input two numbers. Handle the possible exception that may occur if the user enters a non-numeric value, such as a string, instead of a number.

```
package UserdefinedException;
```

```
import java.util.Scanner;
```

```
public class CheckInputvalue {
```

```
    public static void main(String[] args) {
        try {
            Scanner s=new Scanner(System.in);
            System.out.println("Enter the Input Number:");
            int n1=s.nextInt();
            System.out.println("Enter the Input Number:");
            int n2=s.nextInt();
            System.out.println(n1+" "+n2);
        }catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Output:

```
Console X
<terminated> CheckInputvalue [Java Application] C:\Program
Enter the Input Number:
12
Enter the Input Number:
str
java.util.InputMismatchException
```

#### Question 6:

Implement a method called calculateFactorial that takes an integer as a parameter and calculates its factorial. Handle the exception that may occur if the input number is negative or exceeds a predefined limit.

```
package UserdefinedException;

import java.util.Scanner;

class MaxLimitException extends Exception{
    MaxLimitException(String msg){
        System.out.println(msg);
    }
}

class NegativeException extends Exception{
    NegativeException(String msg){
        System.out.println(msg);
    }
}

public class Factorial {
    void method() throws MaxLimitException, NegativeException{
        Scanner s=new Scanner(System.in);
        System.out.println("Input Number:");
        int n1=s.nextInt();
        System.out.println("Input Limit:");
        int n2=s.nextInt();
        int fact=1;
        for(int i=1;i<=n1;i++) {
            fact*=i;
        }
        if( n1>n2) {
            throw new MaxLimitException("Exception Max Number....");
        }else if(n1<0){
            throw new NegativeException("Exception Negative Number....");
        }else {
```

```
        System.out.println("Factoreal:"+fact);
    }
}

public static void main(String[] args) {
    Factoreal obj=new Factoreal();
    for(int i=1;i<=3;i++) {
        try {
            obj.method();
        } catch(MaxLimitException e) {
            System.out.println(e);
        } catch(NegativeException e) {
            System.out.println(e);
        }
    }
}
}
```

**Output:**



```
Console X
<terminated> Factoreal [Java Application] C:\Program Files\Java\jdk-20\
Input Number:
5
Input Limit:
6
Factoreal:120
Input Number:
5
Input Limit:
4
Exception Max Number....
UserdefinedException.MaxLimitException
Input Number:
-5
Input Limit:
6
Exception Negative Number....
UserdefinedException.NegativeException
```

### Exception Handling:

#### 1. What is exception handling in Java, and why is it important?

Exception handling in Java is a mechanism to handle runtime errors, allowing a program to continue execution or gracefully terminate. It involves using a set of keywords and classes to manage errors and exceptional conditions that occur during the execution of a program

#### 2. Differentiate between checked and unchecked exceptions in Java.

### Checked Exceptions:

**Definition:** Checked exceptions are exceptions that are checked at compile-time. These are subclasses of Exception (excluding RuntimeException and its subclasses).

**Examples:** IOException, SQLException, ClassNotFoundException

### Unchecked Exceptions:

**Definition:** Unchecked exceptions are exceptions that are not checked at compile-time. These are subclasses of RuntimeException.

**Examples:** NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException

**3. Explain the purpose of the try, catch, and finally blocks in exception handling.**

**Try Block**

**Purpose:** The try block is used to wrap the code that might throw an exception. This block defines a scope for catching exceptions.

**Catch Block**

**Purpose:** The catch block is used to handle the exception that occurs in the try block. You can have multiple catch blocks to handle different types of exceptions.

**Finally Block**

**Purpose:** The finally block is used to execute important code such as cleanup code, which should run regardless of whether an exception was thrown or not. This block is always executed after the try and catch blocks, except in the case of a system exit or fatal error

**4. How is the "throw" keyword used in Java exception handling?**

In Java, the throw keyword is used to explicitly throw an exception. This can be either a built-in exception or a custom exception. By using the throw keyword, you can create an instance of an exception and throw it to signal that an error has occurred. This allows you to handle exceptional situations more flexibly and in a controlled manne

**5. Discuss the concept of custom exceptions and when to use them.**

Custom exceptions in Java are user-defined exceptions that extend the Exception class (or its subclasses) to create more specific and meaningful error conditions within your application. They allow you to create exceptions that are tailored to your particular application logic and use cases

**6. What is the significance of the "throws" clause in a method signature?**

The throws clause in a method signature is used to declare that a method can throw one or more exceptions. This clause informs the caller of the method about the types of exceptions that might be thrown, making it possible for the caller to handle these exceptions appropriately.

**7. Explain the difference between "throw" and "throws" in Java.**

**throw :**

- **Purpose:** The throw keyword is used to explicitly throw an exception from within a method or a block of code.
- **Usage:** It is used when you want to signal an exceptional condition or error manually. You create an instance of an exception (either built-in or custom) and throw it.
- **Context:** It is used inside the method body.

#### **throws**

- **Purpose:** The throws keyword is used in the method signature to declare that a method can throw one or more exceptions. This informs the caller of the method about the potential exceptions that need to be handled.
- **Usage:** It is used when a method is capable of throwing exceptions, and you want to specify those exceptions in the method's signature.
- **Context:** It is used in the method declaration.

#### **8. Discuss the role of the "finally" block and its execution in exception handling.**

à The finally block in Java plays a crucial role in exception handling by providing a way to execute code that must run regardless of whether an exception is thrown or not. It is typically used for cleanup activities, such as closing resources (files, database connections, sockets, etc.) to ensure that these resources are released properly even if an error occurs.

#### **9. How does the "try-with-resources" statement enhance exception handling in Java?**

The "try-with-resources" statement in Java enhances exception handling by simplifying the management and cleanup of resources that require closing, such as files, database connections, sockets, etc. It was introduced in Java 7 as part of the language's ongoing effort to improve resource management and exception handling. Here's how it enhances exception handling

#### **9. Explain the order of execution of catch blocks when an exception occurs.**

When an exception occurs in Java and is thrown from within a try block, the order of execution of catch blocks is determined by the type hierarchy of the exceptions and their corresponding catch blocks. Here's how the order of execution of catch blocks is determined

#### **11. Discuss the purpose of the multi-catch block introduced in Java 7.**

The multi-catch block introduced in Java 7 enhances the language's exception handling capabilities by allowing a single catch block to handle multiple types of exceptions simultaneously. This feature simplifies and improves the readability of code where different exceptions are handled in the same way, reducing redundancy and making the exception handling more concise.

#### **12. What is the role of the "Exception" class in the Java exception hierarchy?**

In Java, the Exception class plays a central role in the exception hierarchy as it serves as the base class for checked exceptions, which are exceptions that must be either caught or declared in the method signature using the throws clause. Here's an overview of the role and significance of the Exception class.

### **13. How does the "finally" block handle exceptions thrown in the "try" block?**

The finally block in Java plays a crucial role in exception handling by providing a mechanism to execute cleanup code, regardless of whether an exception is thrown or not in the preceding try block. Here's how the finally block handles exceptions thrown in the try block.

### **14. Explain the concept of exception propagation in Java.**

Exception propagation in Java refers to the mechanism by which an exception is passed from one method to another up the call stack until it is handled or the program terminates if it remains unhandled. This process ensures that exceptions are not only detected but also appropriately managed or reported within the application. Here's a detailed explanation of how exception propagation works in Java

### **15. Discuss the importance of stack trace in debugging exceptions.**

#### **· Identifying the Source:**

- The stack trace pinpoints the exact location in the code where an exception occurred. It includes the names of methods called, along with the corresponding line numbers and class names.
- This information helps developers quickly locate the source of the exception and understand the context in which it occurred.

#### **· Understanding Method Call Hierarchy:**

- By examining the stack trace, developers can trace the sequence of method calls leading up to the exception. This helps in understanding the flow of execution and identifying any unexpected or erroneous method invocations.

#### **· Diagnosing Root Cause:**

- Exception stack traces often reveal the root cause of the exception, such as null references, out-of-bounds accesses, or logic errors. This diagnostic information is crucial for resolving the underlying issue causing the exception

### **16. How does the "assert" statement contribute to exception handling?**

The assert statement in Java is primarily used for debugging purposes and does not directly contribute to exception handling in the traditional sense. However, it serves a related role in enhancing code reliability and detecting logical errors early in development. Here's how the assert statement works and its relationship to exception handling.

### **17. Explain the difference between "Error" and "Exception" in Java.**

**Error:** Errors represent abnormal conditions that are typically beyond the control of the application and usually cannot be anticipated or recovered from. They often indicate serious problems that may cause the JVM or the application itself to terminate.

- Examples: OutOfMemoryError, StackOverflowError, AssertionError.

**Exception:** Exceptions represent conditions that a well-written application should anticipate and handle. They are often recoverable and allow the program to continue execution under controlled conditions.

- Examples: IOException, NullPointerException, NumberFormatException.

#### **18. Discuss the role of the "try" block without any catch or finally blocks.**

→ In Java, a try block serves a specific role in exception handling, even when it is not accompanied by catch or finally blocks. Understanding the use of a try block without catch or finally involves scenarios where you might want to attempt a risky operation without handling exceptions locally or providing cleanup operations.

#### **19. How can you create a user-defined exception in Java?**

→ In Java, creating user-defined exceptions involves extending either the Exception class or one of its subclasses to define custom exception types tailored to specific application needs. Here's a step-by-step guide on how to create and use user-defined exceptions

#### **20. Explain the purpose of the "getMessage()" method in the Throwable class.**

##### **→ Retrieving Error Information:**

- The getMessage() method is used to retrieve a detailed error message associated with an instance of a Throwable object, which includes instances of both Exception and Error.

##### **→ Providing Contextual Information:**

- When an exception is thrown (whether it's a built-in Java exception or a custom exception), developers can provide an informative message that describes the reason for the exception.
- This message can include details about what went wrong, what conditions led to the error, or any other relevant contextual information.

##### **→ Diagnostic Information:**

- The error message returned by getMessage() is often used for diagnostic purposes. It helps developers understand the nature and cause of the exception quickly, especially when debugging code.