Activity-4

1. Write and test program to create a class known as "BankAccount" with methods called deposit() and withdraw(). Create a subclass called SavingsAccount that overrides the withdraw() method to prevent withdrawals if the account balance falls below one hundred using inheritance.

```
class BankAccount {
    protected double balance;
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: " + amount);
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
        } else {
            System.out.println("Invalid withdrawal amount or insufficient balance.");
        }
    }
    public void displayBalance() {
        System.out.println("Current Balance: " + balance);
    }
}

class SavingsAccount extends BankAccount {
    public SavingsAccount(double initialBalance) {
        super(initialBalance);
    }
    @Override
    public void withdraw(double amount) {
        if (balance - amount < 100) {
            System.out.println("Withdrawal denied! Minimum balance of 100 must
be maintained.");
        } else {
            super.withdraw(amount);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        SavingsAccount myAccount = new SavingsAccount(500);
        myAccount.displayBalance();
        myAccount.deposit(200);
        myAccount.displayBalance();
        myAccount.withdraw(550);
        myAccount.displayBalance();
        myAccount.withdraw(50);
        myAccount.displayBalance();
    }
}
```

Output:

```
Current Balance: 500.0
Deposited: 200.0
Current Balance: 700.0
Withdrawn: 550.0
Current Balance: 150.0
Withdrawn: 50.0
Current Balance: 100.0
```

Activity-5

5.Write and test program to create a base class BankAccount with methods deposit() and withdraw(). Create two subclasses SavingsAccount and CheckingAccount. Override the withdraw() method in each subclass to impose different withdrawal limits and fees using polymorphism.

```
// Base class
class BankAccount {
    protected double balance;
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: $" + amount);
        } else {
            System.out.println("Deposit amount must be positive.");
        }
    }
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: $" + amount);
        } else {
            System.out.println("Insufficient funds or invalid amount.");
        }
    }
    public double getBalance() {
        return balance;
    }
}
// SavingsAccount subclass with withdrawal limit
class SavingsAccount extends BankAccount {
    private static final double WITHDRAWAL_LIMIT = 1000; // Max withdrawal
limit
    public SavingsAccount(double initialBalance) {
        super(initialBalance);
    }
    @Override
    public void withdraw(double amount) {
        if (amount > WITHDRAWAL_LIMIT) {
            System.out.println("Withdrawal failed! Savings account has a limit of $"
+ WITHDRAWAL_LIMIT);
        } else if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Savings Account: Withdrawn $" + amount);
        } else {
            System.out.println("Invalid withdrawal amount or insufficient funds.");
        }
    }
}
// CheckingAccount subclass with transaction fee
class CheckingAccount extends BankAccount {
    private static final double TRANSACTION_FEE = 5.0; // Fixed fee per
withdrawal
    public CheckingAccount(double initialBalance) {
        super(initialBalance);
    }
    @Override
    public void withdraw(double amount) {
        double totalAmount = amount + TRANSACTION_FEE;
        if (amount > 0 && totalAmount <= balance) {
            balance -= totalAmount;
            System.out.println("Checking Account: Withdrawn $" + amount + " (Fee:
$" + TRANSACTION_FEE + ")");
        } else {
            System.out.println("Insufficient funds or invalid amount. Fee included.");
        }
    }
}

// Test the implementation
```

```
 class BankTest {
    public static void main(String[] args) {
        // Creating accounts
        BankAccount savings = new SavingsAccount(2000);
        BankAccount checking = new CheckingAccount(1500);
        // Testing deposit
        System.out.println("\nDepositing money:");
        savings.deposit(500);
        checking.deposit(300);
        // Testing withdrawals
        System.out.println("\nAttempting withdrawals:");
        savings.withdraw(1100); // Exceeds limit
        savings.withdraw(900);  // Valid
        checking.withdraw(1400); // Insufficient funds (after fee)
        checking.withdraw(1000); // Valid
        // Checking final balances
        System.out.println("\nFinal Balances:");
        System.out.println("Savings Account Balance: $" + savings.getBalance());
        System.out.println("Checking Account Balance: $" +
checking.getBalance());
    }
}
```

Output:

```
Depositing money:
Deposited: $500.0
Deposited: $300.0
Attempting withdrawals:
Withdrawal failed! Savings account has a limit of $1000.0
```

```
Savings Account: Withdrawn $900.0
Checking Account: Withdrawn $1400.0 (Fee: $5.0)
Insufficient funds or invalid amount. Fee included.
Final Balances:
Savings Account Balance: $1600.0
Checking Account Balance: $395.0
```

Activity-6

1. Write and test code to check if given year is a leap year or not.

```
import java.util.Scanner;
public class LeapYearChecker {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Taking user input for the year
        System.out.print("Enter a year: ");
        int year = scanner.nextInt();

        // Checking leap year condition
        if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {
            System.out.println(year + " is a Leap Year.");
        } else {
            System.out.println(year + " is not a Leap Year.");
        }
        scanner.close();
    }
}
```

Output:

```
Enter a year: 2000
2000 is a Leap Year.
Enter a year: 2001
2001 is not a Leap Year.
```

2. Write and test code to display multiplication table for a given number

```
import java.util.Scanner;
public class MultiplicationTable {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Ask user for a number
        System.out.print("Enter a number: ");
        int number = scanner.nextInt();
        // Display multiplication table
        System.out.println("Multiplication Table for " + number + ":");
        for (int i = 1; i <= 10; i++) {
            System.out.println(number + " x " + i + " = " + (number * i));
        }
        scanner.close();
    }
}
```

Output:

```
Enter a number: 2
Multiplication Table for 2:
2 x 1 = 2
2 x 2 = 4
```

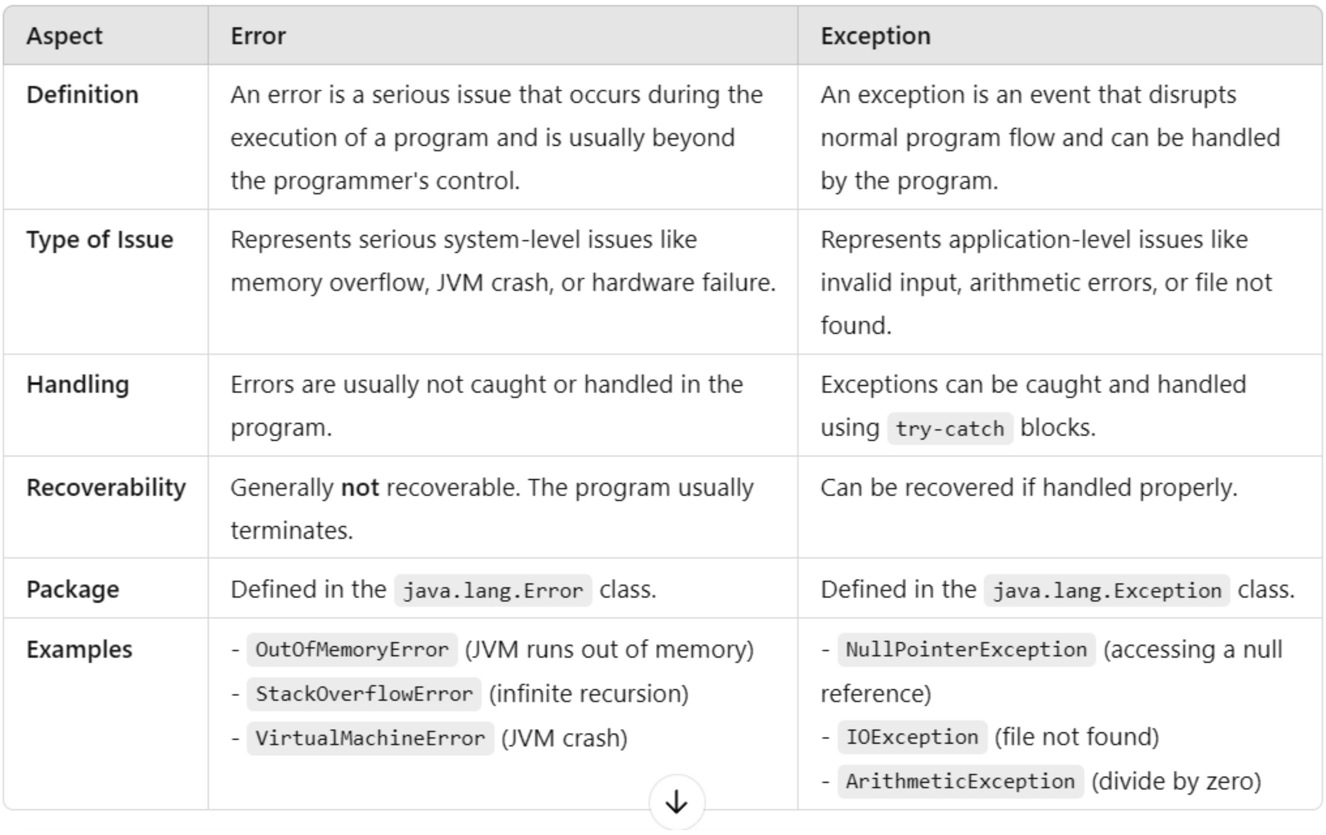
```
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

3. Write and test code to display pyramid and diamond pattern as shown.

```
public class PatternDisplay {
    public static void main(String[] args) {
        int n = 5; // Number of rows
        // Pyramid Pattern
        for (int i = 1; i <= n; i++) {
            for (int j = i; j < n; j++) {
                System.out.print("   "); // Printing spaces
            }
            for (int j = 1; j <= (2 * i - 1); j++) {
                System.out.print("*  "); // Printing stars
            }
            System.out.println();
        }
        System.out.println(); // Space between patterns
        // Diamond Pattern
        for (int i = 1; i <= n; i++) {
            for (int j = i; j < n; j++) {
                System.out.print("   "); // Printing spaces
            }
            for (int j = 1; j <= (2 * i - 1); j++) {
                System.out.print("*  "); // Printing stars
            }
            System.out.println();
        }
        for (int i = n - 1; i >= 1; i--) {
            for (int j = n; j > i; j--) {
                System.out.print("   "); // Printing spaces
            }
            for (int j = 1; j <= (2 * i - 1); j++) {
                System.out.print("*  "); // Printing stars
            }
            System.out.println();
        }
    }
}
```

Activity-7

1.Differentiate error and exception



Aspect	Error	Exception
Definition	An error is a serious issue that occurs during the execution of a program and is usually beyond the programmer's control.	An exception is an event that disrupts normal program flow and can be handled by the program.
Type of Issue	Represents serious system-level issues like memory overflow, JVM crash, or hardware failure.	Represents application-level issues like invalid input, arithmetic errors, or file not found.
Handling	Errors are usually not caught or handled in the program.	Exceptions can be caught and handled using `try-catch` blocks.
Recoverability	Generally **not** recoverable. The program usually terminates.	Can be recovered if handled properly.
Package	Defined in the `java.lang.Error` class.	Defined in the `java.lang.Exception` class.
Examples	- `OutOfMemoryError` (JVM runs out of memory) - `StackOverflowError` (infinite recursion) - `VirtualMachineError` (JVM crash)	- `NullPointerException` (accessing a null reference) - `IOException` (file not found) - `ArithmeticException` (divide by zero)

2. Identify and document system exceptions

1. OutOfMemoryError

- **Description:** Thrown when the Java Virtual Machine (JVM) runs out of memory and the garbage collector is unable to reclaim enough memory.
- **Example:** When your program tries to allocate too much memory for an object or array, resulting in a memory leak.
- **How to Handle:** Although you cannot "catch" this error in a traditional sense, ensuring efficient memory usage and avoiding memory leaks can mitigate it.

```
try {
    int[] largeArray = new int[Integer.MAX_VALUE];  // may cause OutOfMemoryError
} catch (OutOfMemoryError e) {
    System.out.println("Out of memory!");
}
```

2. StackOverflowError

- **Description:** Thrown when a stack overflow occurs, typically due to excessive recursion or deep nesting of method calls.

- **Example:** A method calls itself too many times without a proper base case.
- **How to Handle:** You can try optimizing recursion or use an iterative approach where applicable.

```
public class StackOverflowExample {
    public void recursiveMethod() {
        recursiveMethod();  // Infinite recursion
    }
}
```

3. VirtualMachineError

- **Description:** This is the superclass of errors that are related to the virtual machine (JVM). Common subtypes include OutOfMemoryError and StackOverflowError.
- **Example:** An issue in the JVM that prevents normal operation.
- **How to Handle:** Generally, these errors indicate a severe failure in the JVM, and the application is unlikely to recover from them.

4. UnknownError

- **Description:** Thrown when an unknown error occurs in the JVM.
- **Example:** This is an extreme case and may happen due to internal JVM errors.
- **How to Handle:** It's usually an unrecoverable error, and the program may need to be terminated.

5. AssertionError

- **Description:** Thrown when an assertion fails. Assertions are used to test assumptions in your code.
- **Example:** When you use the assert keyword and the condition evaluates to false.
- **How to Handle:** You can catch it if needed, but generally, assertions are used to indicate programming errors during development.

```
assert x > 0 : "x must be positive";  // AssertionError is thrown if x <= 0
```

6. LinkageError

- **Description:** Thrown when a class or interface is found to be incompatible with another class that has already been loaded.
- **Example:** This can happen when classes are loaded dynamically at runtime, and they are incompatible with the class definitions that already exist in the JVM.
- **How to Handle:** Ensure that the classes are compatible and there are no conflicts in the classpath.

7. ClassFormatError

- **Description:** Thrown when the JVM encounters a class file that does not have a valid format.
- **Example:** Corrupted class files or incompatible class versions can trigger this error.
- **How to Handle:** Ensure that your compiled class files are valid and the correct versions of libraries are used.

8. OutOfSyncError

- **Description:** Thrown when there is a mismatch in synchronization of objects in the JVM.
- **Example:** A situation where two threads are trying to access synchronized blocks incorrectly.
- **How to Handle:** Carefully manage synchronization between threads to avoid concurrency issues

Activity-8

1. Prepare a report on how custom exceptions are implemented in java program with example

Introduction

In Java, exceptions are events that disrupt the normal flow of execution of a program. Custom exceptions are user-defined exceptions that extend the built-in Exception class or its subclasses. They allow developers to handle specific error conditions more effectively by providing meaningful error messages tailored to the application.

This report will discuss how custom exceptions are implemented in Java programs, highlighting their usage and providing an example.

Steps to Create Custom Exceptions

1. **Define a Custom Exception Class**:
   - To create a custom exception, you need to define a new class that extends the Exception class or its subclass like RuntimeException.
   - You can include constructors in your custom exception class to pass custom error messages.
2. **Throwing the Custom Exception**:
   - The throw keyword is used to explicitly throw an instance of the custom exception when a specific condition is met in the program.
3. **Catching the Custom Exception**:
   - You can handle custom exceptions using try-catch blocks, just like handling standard exceptions.

Eg:

```
// Custom exception class
class InsufficientBalanceException extends Exception {
    // Constructor with a custom error message
    public InsufficientBalanceException(String message) {
        super(message);
    }
}
// Main class
public class BankAccount {
    private double balance;
    public BankAccount(double balance) {
        this.balance = balance;
    }
    // Method to withdraw money, throws a custom exception if balance is
insufficient
    public void withdraw(double amount) throws InsufficientBalanceException {
        if (amount > balance) {
            throw new InsufficientBalanceException("Insufficient balance.
Withdrawal amount: " + amount + ", Available balance: " + balance);
        }
        balance -= amount;
        System.out.println("Withdrawal successful! New balance: " + balance);
    }
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000.00);
        try {
            // Attempting to withdraw more than the available balance
            account.withdraw(1500.00);
        } catch (InsufficientBalanceException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Output:

```
Error: Insufficient balance. Withdrawal amount: 1500.0, Available balance:
1000.0
```

Advantages of Custom Exceptions:

- **Better Error Handling**: Custom exceptions allow you to provide more meaningful and context-specific error messages.

- **Cleaner Code**: By using custom exceptions, your code becomes cleaner and easier to understand since each exception can be linked to a specific error case.
- **Customizable Logic**: Custom exceptions allow developers to define custom logic for error scenarios, which is not possible with standard exceptions.

Conclusion

Custom exceptions in Java provide a powerful way to handle specific error conditions in your application. By extending the built-in Exception class, you can create exceptions that are more relevant to your program's needs, making error handling more intuitive and maintainable. The example above demonstrates how to define, throw, and catch a custom exception, ensuring robust error handling in Java applications.

Activity-9

1. Prepare a report on how to connect java with MySQL

Introduction

Connecting Java to MySQL allows developers to create dynamic and interactive applications that can manage, retrieve, and modify data stored in MySQL databases. This report provides the steps and example code to establish a connection between Java applications and MySQL.

Prerequisites

1. **Java Development Kit (JDK)** - Ensure that the JDK is installed on your system. You can download it from Oracle's website.
2. **MySQL Server** - Install MySQL on your system. You can download it from MySQL's official site.
3. **MySQL JDBC Driver (Connector/J)** - The JDBC driver allows Java to communicate with MySQL databases. You can download it from here. Alternatively, you can add it via Maven dependency in your project.
4. **IDE** - Use an Integrated Development Environment (IDE) like IntelliJ IDEA, Eclipse, or NetBeans for easier development.

Step 1: Install MySQL and Create Database

1. **Install MySQL** - Download and install MySQL if you haven't already done so.
2. **Create Database -** You need to create a database in MySQL to interact with Java. Example SQL command to create a database:

```
CREATE DATABASE sampleDB;
USE sampleDB;
```

3.**Create a Table** - After creating a database, create a sample table.

```
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50),
    email VARCHAR(50)
);
```

Step 2: Add MySQL JDBC Driver to Java Project

- If you're using Maven, you can add the dependency for MySQL JDBC driver in the pom.xml:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.25</version>  <!-- Check for the latest version -->
</dependency>
```

- If you're not using Maven, you need to download the JAR file from MySQL's website and add it to the classpath of your project.

Step 3: Java Code to Connect to MySQL

The following steps detail how to write a simple Java program to connect to a MySQL database.

1. **Import Required Classes**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
```

2.Establish a Connection

You can use DriverManager.getConnection() to connect to the database.

```
public class MySQLConnection {
    public static void main(String[] args) {
        // Database credentials
        String url = "jdbc:mysql://localhost:3306/sampleDB";  // URL of the
database
        String username = "root";  // MySQL username
        String password = "your_password";  // MySQL password
        try {
            // Establish connection
            Connection connection = DriverManager.getConnection(url, username,
password);
            System.out.println("Connection established successfully!");
            // Create statement
            Statement stmt = connection.createStatement();

            // Example query to fetch data from the users table
            ResultSet rs = stmt.executeQuery("SELECT * FROM users");
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                String email = rs.getString("email");
                System.out.println("ID: " + id + ", Name: " + name + ", Email: " +
email);
            }
            // Close connection
            rs.close();
            stmt.close();
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Step 4: Run the Java Program

- Compile and run the Java program from your IDE or using the command line.
- If the connection is successful, the data from the users table will be printed to the console.

Step 5: Handle Exceptions and Errors

It's essential to handle SQL exceptions properly using try-catch blocks. Common exceptions include:

- SQLException for database errors.
- ClassNotFoundException when the JDBC driver class cannot be found.

You can use e.printStackTrace() for debugging, or log exceptions for better management in production code.

Step 6: Close the Connection

It's crucial to close the database resources after usage. Failing to do so can lead to memory leaks and performance issues.

```
// Close ResultSet, Statement, and Connection in the finally block
finally {
    try {
        if (rs != null) rs.close();
        if (stmt != null) stmt.close();
        if (connection != null) connection.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Conclusion

Connecting Java with MySQL involves the following key steps:

- Installing MySQL and creating a database.
- Adding the JDBC driver to your Java project.
- Writing Java code to establish a connection to the database.
- Executing queries and handling the results.