Description of the Application:

The **MauriBankLoanCalculator** is a Java-based console application that helps users calculate the monthly payments for different types of loans: **Home Loan**, **Car Loan**, and **Personal Loan**. The application collects user inputs such as age, loan amount, and loan term, and based on these details, calculates the monthly payment using the specific interest rate associated with each loan type. The results are then displayed in an ASCII art table for better readability.

Object-Oriented Programming (OOP) Features in the Application:

1. Abstraction:

- **Definition**: Abstraction allows you to hide the implementation details of specific functionality while exposing only the necessary information to the user.
- **Application in the Code**: The `Loan` class is an abstract class that provides the basic structure for different types of loans. It contains attributes and methods that are common to all loan types (such as `loanAmount`, `loanTerm`, and `calculateMonthlyPayment()`), but the specific details, like setting the interest rate and the loan type, are abstracted and left to the subclasses (e.g., `HomeLoan`, `CarLoan`, and `PersonalLoan`).

```java
public abstract class Loan {
    protected double loanAmount;
    protected int loanTerm; // in years
    protected double interestRate;
    public Loan(double loanAmount, int loanTerm) {
        this.loanAmount = loanAmount;
        this.loanTerm = loanTerm;
        setInterestRate(); // Interest rate is set by the bank for each loan type
    }

    // Abstract method to be implemented by subclasses
    protected abstract void setInterestRate();
    public abstract String getLoanType();
}
```

2. Encapsulation:

- **Definition**: Encapsulation restricts direct access to some of an object's components, which is a way of hiding the internal state and only exposing methods that modify or access the state.
- **Application in the Code**: The fields `loanAmount`, `loanTerm`, and `interestRate` are declared as `protected` in the `Loan` class. The users of the class cannot directly access or modify these fields. Instead, they interact with the class through public methods like `getInterestRate()` and `calculateMonthlyPayment()`, which control how the internal data is accessed and modified.

```
protected double loanAmount;
protected int loanTerm;
protected double interestRate;

public double getInterestRate() {
    return interestRate;
}
```

3. Inheritance:

- **Definition**: Inheritance allows a class (called a child or subclass) to inherit fields and methods from another class (called a parent or superclass).
- **Application in the Code**: The classes `HomeLoan`, `CarLoan`, and `PersonalLoan` all inherit from the abstract `Loan` class. They reuse the common functionality provided by the `Loan` class (such as calculating monthly payments) and override specific behaviors like setting the interest rate and defining the loan type.
- This approach reduces code duplication since all loans share common features like loan amount and term but differ in their specific characteristics (e.g., interest rates).

```java
class HomeLoan extends Loan {
    public HomeLoan(double loanAmount, int loanTerm) {
        super(loanAmount, loanTerm);
    }
    @Override
    protected void setInterestRate() {
        this.interestRate = 5.0; // Bank-defined interest rate for home loan
    }
    @Override
    public String getLoanType() {
        return "Home Loan";
    }
}
```

4. Polymorphism:

- **Definition**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. The specific method to be executed is determined at runtime based on the object type.
- **Application in the Code**: The `Loan` object is polymorphic, meaning it can reference objects of any subclass (e.g., `HomeLoan`, `CarLoan`, or `PersonalLoan`). The specific implementation of `setInterestRate()` and `getLoanType()` is chosen at runtime based on the type of loan object created.
- For example, when the user selects a loan type, the system dynamically creates the corresponding loan object (either `HomeLoan`, `CarLoan`, or `PersonalLoan`) and calculates the monthly payment using the overridden methods of that specific subclass.

```
Loan loan = null;
switch (loanChoice) {
    case 1:
        loan = new HomeLoan(loanAmount, loanTerm);
        break;
    case 2:
        loan = new CarLoan(loanAmount, loanTerm);
        break;
    case 3:
        loan = new PersonalLoan(loanAmount, loanTerm);
        break;
}
```

5. Method Overriding:

- **Definition**: Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its parent class.
- **Application in the Code**: The `setInterestRate()` and `getLoanType()` methods are overridden in each subclass (`HomeLoan`, `CarLoan`, and `PersonalLoan`). Each subclass provides its own implementation of these methods based on the specific type of loan.

```
@Override
protected void setInterestRate() {
    this.interestRate = 5.0; // Bank-defined interest rate for home loan
}

@Override
public String getLoanType() {
    return "Home Loan";
}
```

6. Error Handling:

- **Definition**: Error handling allows the program to handle unexpected or invalid input without crashing, providing meaningful feedback to the user.
- **Application in the Code**: The program uses `try-catch` blocks to handle

exceptions like `InputMismatchException` (for invalid data types) and `IllegalArgumentException` (for invalid loan terms or loan amounts). This ensures the program can gracefully handle errors and give feedback to the user when incorrect input is provided.

```java
try {
    // Input and processing code
} catch (InputMismatchException e) {
    System.out.println("Invalid input. Please enter the correct data type.");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

Summary of OOP Principles Applied:

1. **Abstraction**: The abstract class `Loan` provides a general structure for different loan types, leaving specific details to the subclasses.
2. **Encapsulation**: The class restricts direct access to its data and exposes it via public methods.
3. **Inheritance**: `HomeLoan`, `CarLoan`, and `PersonalLoan` inherit from `Loan` to reuse the common loan logic.
4. **Polymorphism**: A `Loan` object can reference any loan subclass, and method calls are resolved based on the actual object type at runtime.
5. **Method Overriding**: Subclasses provide their own implementation of certain methods like `setInterestRate()` and `getLoanType()`.
6. **Error Handling**: The program handles invalid inputs and exceptions gracefully, ensuring robust and user-friendly behavior.

These OOP features help to create a clean, modular, and maintainable codebase that can easily be extended or modified in the future.