POLYMORPHISM-METHOD OVERLOADING AND OVERRIDING

Q1. Design a class named **Person** and its two subclasses named **Student** and **Employee**. Make **Faculty** and **Staff** subclasses of **Employee**. A person has a name, address, phone number, and e-mail address. A student has a class status (freshman, sophomore, junior, or senior). Define the status as a constant. An employee has an office, salary, and date hired. A faculty member has office hours and a rank. A staff member has a title. Override the **toString** method in each class to display the class name and the person's name.

Write a test program that creates a **Person**, **Student**, **Employee**, **Faculty**, and **Staff**, and invokes their **toString()** methods.

Q2. Design a simulation of a **Multi-Mode Payment System** that supports different payment methods: CreditCard, DebitCard, UPI, and NetBanking.

Each payment method has:

- A unique **transaction fee logic**.
- A way to **authorize** a payment.
- Custom **initialization messages** using **static** and **instance initializer blocks**.

Requirements:

1. Create a **base class** PaymentMethod with:
    - A double amount field.
    - A constructor to set the amount.
    - A method double calculateFee() — to be overridden.
    - A method void authorize() — to be overridden.
2. Create **4 subclasses**:
    - CreditCard
    - DebitCard
    - UPI
    - NetBanking

    Each subclass should:
    - Override calculateFee() and authorize() with its own logic.
    - Include a **static block** to print "Class [ClassName] loaded".
    - Include an **instance initializer block** to print "Instance of [ClassName] created".
3. Create a PaymentProcessor class:
    - Has a method void process(PaymentMethod method) that:
        - Calls authorize() and calculateFee() polymorphically.
    - In the main method:
        - Randomly choose one of the four payment types.
        - Create an instance (with random amount ₹500 to ₹5000).
        - Upcast to PaymentMethod.
        - Call process()

Q3. Design a **Custom Calculator** that demonstrates the full depth of **method overloading**

You must overload the calculate() method for different operations and data types. The program should highlight how Java resolves overloaded methods based on:

- Number of arguments
- Argument types
- Type promotion
- Varargs
- Ambiguity in overloads

Requirements:

1. Create a class CustomCalculator with **at least 6 overloaded** versions of a method named calculate():
    - int calculate(int a, int b) – returns sum
    - double calculate(double a, double b) – returns product
    - long calculate(long a, int b) – returns difference
    - float calculate(float a, float b, float c) – returns average
    - int calculate(int... values) – uses **varargs** to return total
    - void calculate(short a, short b) – just prints "Short version called"
2. In the main() method of a separate class:
    - Call all the overloaded versions of calculate() with appropriate arguments.
    - Intentionally call the method with **values like calculate(10, 10)** and observe which version is called.
    - Call calculate(10L, 10) and calculate(10, 10L) – and **explain the results**.
    - Attempt to call calculate(10, 10) when both int and short versions are available, and observe **ambiguity**.
    - Resolve ambiguity explicitly using **type casting**.