Question 1

After reading the blog that overviews error handling in Java, analyse the following code:

```
public class Program {
    public static void main(String[] args) {
        BankAccount acc1 = new BankAccount("AB123", 100.0);
        BankAccount acc2 = new BankAccount("CD456", 0.0);
        boolean depositResult = acc1.deposit(200.8);
        if (depositResult == true) {
            System.out.println("Money successfully added on account " + acc1.getNumber());
        } else {
            System.out.println("Deposit failed on account " + acc1.getNumber());
        }
        if (acc2.deposit(130.5) == true) {
            System.out.println("Money successfully added on account " + acc2.getNumber());
        } else {
            System.out.println("Deposit failed on account " + acc2.getNumber());
        }
        double withdrawAmount = -200.35;
        if (acc1.withdraw(withdrawAmount)) {
            System.out.println("Successful withdrawal! Balance on " + acc1.getNumber() + ": "
+ acc1.getBalance());
        } else {
            System.out.println("Withdrawal failed on " + acc1.getNumber());
        }
        if (acc2.withdraw(1000.50)) {
            System.out.println("Successful withdrawal! Balance on " + acc2.getNumber() + ": "
+ acc2.getBalance());
        } else {
            System.out.println("Withdrawal failed on " + acc2.getNumber());
        }
    }
}
```

Identify the type of *error-handling technique* used when implementing the `BankAccount` class in the blog. Reflect on the advantages and disadvantages of this approach and explain them in no more than 100 words. Add your answer to this Padlet and compare it with the answers provided by other students.

Question 2

Now, read the blog on exceptions. Then, analyse the following implementation of the `BankAccount` class and compare it with the one used in the previous question:

```
public class BankAccount {
    private String number;
    private double balance;

    public BankAccount(String num, double bal) {
        if (bal < 0)
            throw new IllegalArgumentException("The initial balance on " + num + " cannot be
negative");
        number = num;
        balance = bal;
    }

    public void deposit(double amount) {
        if (amount < 0)
            throw new IllegalArgumentException("The amount to deposit on " + number + " cannot
be negative");
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount < 0)
            throw new IllegalArgumentException("The amount to withdraw from " + number + "
cannot be negative");
        else if (amount > balance)
            throw new IllegalStateException("Not enough money on account " + number + ",
balance: " + balance);
        balance -= amount;
    }

    public double getBalance() {
        return balance;
    }

    public String getNumber() {
        return number;
    }
}
```

Once you have identified the difference, write a class to perform operations on bank account objects. In the main method, first, create a new `BankAccount` object `account1` with the `number` "AB123" and an initial `balance` of £100.0 (the initial balance is hardcoded and not read as input; hence, it is always correct).
Then, allow the user to provide input to the program to perform at least one deposit and one withdrawal operation. Ensure the input is read and converted using `Double.parseDouble(scanner.nextLine())`.

Use exceptions as the error-handling strategy and:

1. Write a `try-catch` block to handle `IllegalArgumentException` and `IllegalStateException` according to where they can be generated in the code of the `BankAccount` class—display specific error messages for each type of exception.
2. Test your program by trying to input the value `"three"` (literally) instead of a number. What happens? Look at the examples in the blog and add some code to the main to handle this scenario.
3. Include a `finally` block to ensure that a message indicating the latest `balance` is displayed regardless of whether an exception occurs or not.
4. Use a *loop* to allow the user to retry inserting the input after an error occurred. The program should loop continuously until a valid input is provided.