ST33 Assessment - 14

Basic Questions:

Handling Array Index Out of Bounds

- Write a program that initializes an array of integers with a fixed size. Access an index that is out of the array bounds and handle the resulting exception with an appropriate error message.

Division by Zero

- Create a simple calculator program that takes two integers as input and performs division. Handle the **`ArithmeticException`** that occurs when the divisor is zero and display an appropriate message.

Number Format Exception

- Write a program that takes a string input from the user and converts it to an integer. Handle the **`NumberFormatException`** if the input string is not a valid integer.

File Reading Exception

- Write a program that reads the content of a file. Handle the `FileNotFoundException` and `IOException` that might occur during file operations. Ensure that the file is properly closed in a `finally` block.

Null Pointer Exception

- Create a program that initializes a string variable to `null`. Attempt to call a method on this string and handle the `NullPointerException` with an appropriate message.

Intermediary Questions:

Custom User-Defined Exception

- Define a custom exception called `InvalidAgeException`. Write a program that takes an age as input and throws this exception if the age is less than 18.

Method Overloading with Exceptions

- Write a class with multiple overloaded methods for processing user data. Demonstrate

Intermediary Tasks on Exceptions:

Task-1 (Banking Context)

You are required to create two custom exceptions: `InsufficientFundsException` and `InvalidAccountException`. These exceptions should be used to handle specific error conditions in a banking application that simulates basic operations like withdrawal and deposit.

Steps to Follow:

1. **Define the Custom Exceptions:**
    - **InsufficientFundsException:** Create a new class named `InsufficientFundsException` that extends the `Exception` class.
        - Implement a constructor that accepts a string message and passes it to the superclass constructor.
    - **InvalidAccountException:** Create another class named `InvalidAccountException` that extends the `Exception` class.
        - Implement a constructor that accepts a string message and passes it to the superclass constructor.
2. **Create the Banking Operations Program:**
    - **BankAccount Class:**
        - Create a class named `BankAccount` with the following properties:
            - `String accountNumber`
            - `double balance`
        - Implement the following methods:
            - `deposit(double amount)`: Adds the specified amount to the account balance. Ensure that the deposit amount is positive.
            - `withdraw(double amount)`: Deducts the specified amount from the account balance. Ensure that the withdrawal amount is positive and that the account has sufficient funds. If not, throw an `InsufficientFundsException`.
            - `validateAccount(String accountNumber)`: Checks if the given account number matches the account's number. If not, throw an `InvalidAccountException`.
        - Ensure proper encapsulation by providing getters and setters for the properties.
3. **Handle the Exceptions in the Main Program:**
    - **BankApplication Class:**
        - Create a `main` method that simulates user interactions with the bank account.
        - Prompt the user to enter an account number, deposit amount, and withdrawal amount.
        - Instantiate a `BankAccount` object and use the provided inputs to perform deposit and withdrawal operations.
        - Use `try-catch` blocks to handle `InsufficientFundsException` and `InvalidAccountException`.
        - Display appropriate error messages in the catch blocks to inform the user of invalid operations.

4. **Program Requirements:**
    - **Custom Exception Classes:** Must define `InsufficientFundsException` and `InvalidAccountException` with constructors that accept messages.
    - **BankAccount Class:** Must implement methods to handle deposit, withdrawal, and account validation, throwing exceptions for invalid operations.
    - **Main Method:** Must handle user input, call the bank account methods, and properly handle exceptions to guide the user towards valid operations.

Output Format (for your idea):

```
Enter account number: 123456

Enter deposit amount: 1000

Deposit successful. New balance: 1000.0



Enter withdrawal amount: 1500

InsufficientFundsException: Withdrawal amount exceeds
available balance.



Enter account number to validate: 654321

InvalidAccountException: Account number is invalid.
```

Task-2 (E-Commerce Context)

You are required to create two custom exceptions: `ProductNotFoundException` and `OrderFailedException`. These exceptions should be used to handle specific error conditions in a simulated e-commerce application that deals with product searches and order processing.

Steps to Follow:

1. **Define the Custom Exceptions:**
    - **ProductNotFoundException:** Create a new class named `ProductNotFoundException` that extends the `Exception` class.
        - Implement a constructor that accepts a string message and passes it to the superclass constructor.
    - **OrderFailedException:** Create another class named `OrderFailedException` that extends the `Exception` class.
        - Implement a constructor that accepts a string message and passes it to the superclass constructor.

2. **Create the E-Commerce Application:**
    - **Product Interface:**
        - Define an interface named `Product` with the following method:
            - `String getProductDetails()`: Returns the details of the product.
    - **Concrete Product Class:**
        - Create a class named `ConcreteProduct` that implements the `Product` interface.
        - Implement the `getProductDetails` method to return product details (e.g., name, price, and description).
    - **ProductCatalog Class:**
        - Create a class named `ProductCatalog` with the following methods:
            - `addProduct(Product product)`: Adds a product to the catalog.
            - `Product findProduct(String productName) throws ProductNotFoundException`: Searches for a product by name. If the product is not found, throws `ProductNotFoundException`.
    - **OrderProcessor Interface:**
        - Define an interface named `OrderProcessor` with the following method:
            - `void processOrder(Product product) throws OrderFailedException`: Processes an order for the given product. Throws `OrderFailedException` if the order cannot be processed.
    - **ConcreteOrderProcessor Class:**
        - Create a class named `ConcreteOrderProcessor` that implements the `OrderProcessor` interface.
        - Implement the `processOrder` method to simulate order processing. If the order fails (e.g., due to inventory issues), throw an `OrderFailedException`.
3. **Handle the Exceptions in the Main Program:**
    - **ECommerceApplication Class:**
        - Create a `main` method that simulates user interactions with the e-commerce application.
        - Create and populate a `ProductCatalog` with several products.
        - Prompt the user to enter a product name to search for and an order quantity.
        - Use the `ProductCatalog` to find the product. If the product is not found, handle the `ProductNotFoundException`.
        - If the product is found, use the `OrderProcessor` to process the order. Handle the `OrderFailedException` if the order cannot be processed.
        - Use `try-catch` blocks to handle `ProductNotFoundException` and `OrderFailedException`.
        - Display appropriate error messages in the catch blocks to inform the user of invalid operations.
        - Ensure the program continues running after handling exceptions.
4. **Program Requirements:**
    - **Custom Exception Classes:** Must define `ProductNotFoundException` and `OrderFailedException` with constructors that accept messages.

- **Product Interface and Implementation:** Must define a `Product` interface and a concrete class implementing this interface.
- **ProductCatalog Class:** Must implement methods to add and find products, throwing exceptions for invalid searches.
- **OrderProcessor Interface and Implementation:** Must define an `OrderProcessor` interface and a concrete class implementing this interface, throwing exceptions for order failures.
- **Main Method:** Must handle user input, interact with the product catalog and order processor, and properly handle exceptions to guide the user towards valid operations.

Example Output:

```
Product Catalog:
1. Laptop - $999.99
2. Smartphone - $499.99
3. Headphones - $199.99

Enter product name to search: Tablet
ProductNotFoundException: Product 'Tablet' not found.

Enter product name to search: Laptop
Enter order quantity: 2
Order processed successfully for 2 units of Laptop.

Enter product name to search: Smartphone
Enter order quantity: 5
OrderFailedException: Order failed for product 'Smartphone' due to
insufficient inventory.
```

Additional Notes:

- Ensure that the program handles invalid input types gracefully by wrapping input parsing in a try-catch block for `NumberFormatException`.
- Provide clear and user-friendly error messages to guide the user towards valid operations and inputs.
- Make sure the program allows multiple operations without exiting after a single transaction.

By following these steps, you will gain practical experience in creating custom exceptions, simulating e-commerce operations, validating input, and handling exceptions in Java using classes and interfaces.