OOP
Lab 7.

🎯 Objectives

- Creating new classes using inheritance
- Polymorphism
  - Method Overriding
  - Polymorphic collection
- Aggregation vs. Inheritance

Create a Project/Module with 2 packages:

- `lab7_1`
- `lab7_2`

Each exercise has its own package and Main class (main method).

Exercise 1.

Structure:

- `lab7_1`
  - `BankAccount`
  - `SavingsAccount, CheckingAccount`
  - `Customer, Bank`
  - `Main`

a. `BankAccount` class

- Copy the `BankAccount` class from the lab6_1 package.
- In this exercise you have to modify the `BankAccount` class in order to prepare for extension (inheritance). Modify the visibility of the following members:
  Constructor: `public → protected`
  Attributes `balance, accountNumber: private → protected`
- Create the classes `SavingsAccount` and `CheckingAccount` (see Fig. 1). Both classes extend the `BankAccount` class. The `addInterest` method in the `SavingsAccount` class adds to the balance the interest (interestRate * balance). The `CheckingAccount` class has an `overdraftLimit` attribute. Overdraft limit is basically the money value permitted by the bank which can be withdrawn additional to the credit bank balance.

- Test the `CheckingAccount` and the `SavingsAccount` classes!

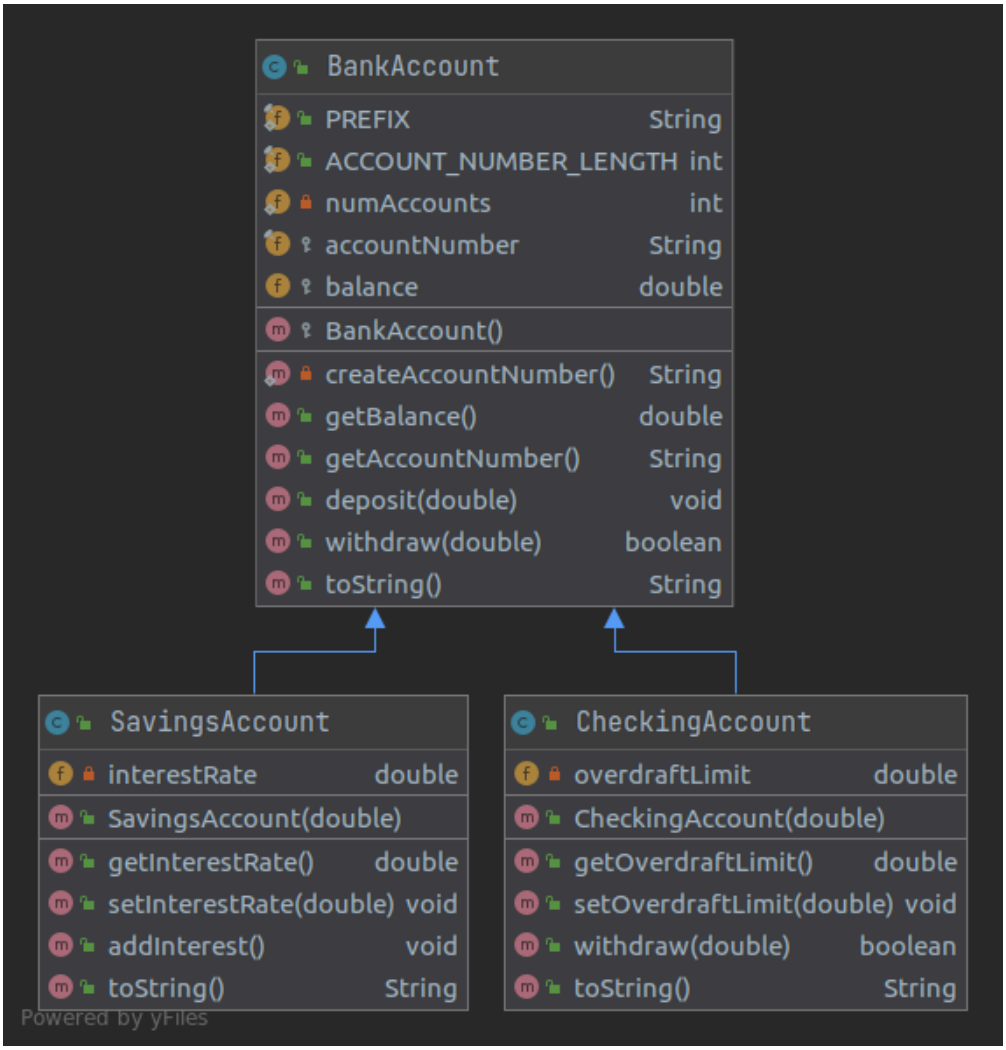

Fig. 1. Class diagram

b. Customer and Bank classes

- Copy the `Customer` and `Bank` classes from `lab6_1` to `lab7_1`. No modifications are required for these classes. Please note that the Customer class aggregates different types of bank accounts, therefore the `accounts` ArrayList will contain references to

different types of accounts (`SavingsAccount` and `CheckingAccount`): **polymorphism**.

```
private ArrayList<BankAccount> accounts = new ArrayList<>();
```

c. Main class

- Create a bank with the name OTP.
- Add two customers to the bank.
- Add two accounts to each customer, one `SavingsAccount` and one `CheckingAccount`.
- Deposit some amount of money in each of the accounts.
- Print the customers to the standard output. In this case, all customer data should be printed, followed by the detailed account information.
- In the case of `SavingsAccount, call` the `addInterest` method.
- Print the customers to the standard output. In this case, all customer data should be printed, followed by the detailed account information.
- Withdraw from each account an arbitrary amount of money
- Print the customers to the standard output. In this case, all customer data should be printed, followed by the detailed account information.

Exercise 2.

In this exercise, you will implement a stack using both the **aggregation** and the **inheritance** relationships and see why the former is preferred over the latter.

A. AGGREGATION

Create a class for the stack data structure. Use an `ArrayList<Object>` for storing the stack's items. In this case, you will use the aggregation relationship (each stack contains an ArrayList). Your stack will have a fixed `capacity`.
Implement the following methods (see Fig. 2):

- constructor: initializes the stack's capacity
- `push(Object): void`
- `pop(): void`
- `top(): Object`
- `isEmpty(): boolean`
- `isFull(): boolean`

In case of empty stack, `top()` will return `null`.

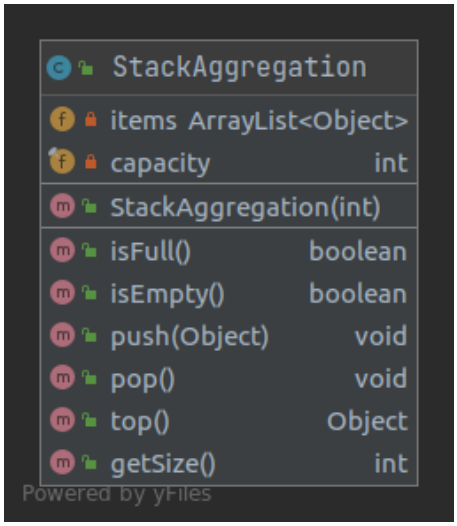

Fig. 2 StackAggregation class

Main method: test your class for two data types: `Integer` and `Character`.

B. INHERITANCE

Create a class for the stack data structure. In this case, use the inheritance for creating your class. Your stack class will extend the `ArrayList<Object>` class (**inheritance**).

Implement the class according to the class diagram shown in Fig. 3. This class will have the same interface (public methods) as the `StackAggregation` class.

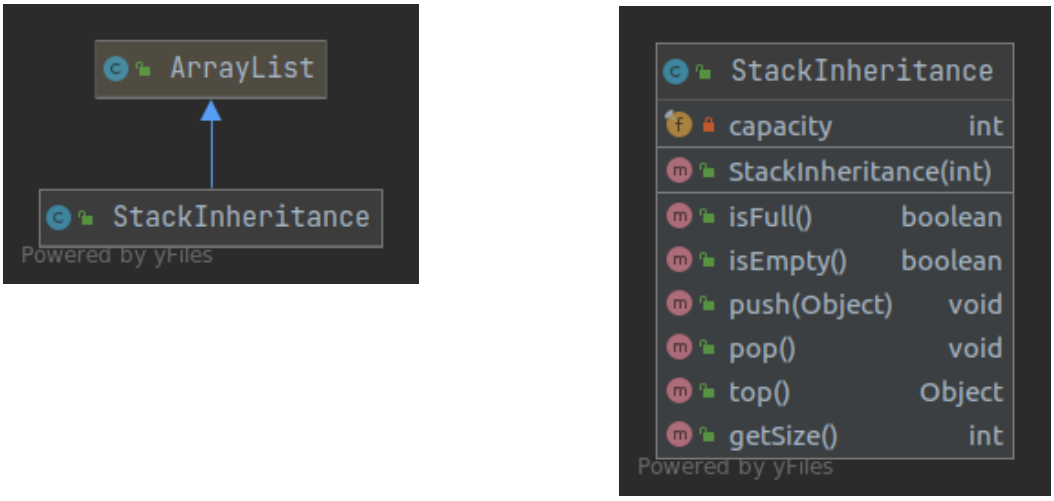

Fig. 3 StackInheritance class

Test your class!

```
StackAggregation stack1 = new StackAggregation( 5 );
for( int i=0; i<10; ++i ){
   // boxing: int --> Integer
   stack1.push( i );
}
System.out.print("StackAggregation : ");
while( !stack1.isEmpty() ){
   System.out.print( stack1.top() + " ");
   stack1.pop();
}
System.out.println();

StackInheritance stack2 = new StackInheritance( 5 );
for( int i=0; i<10; ++i ){
   stack2.push( i );
}
stack2.remove( 1 );
System.out.print("StackInheritance : ");
```

```
while( !stack2.isEmpty() ){
   System.out.print( stack2.top() + " ");
   stack2.pop();
}
System.out.println();
```

Why does the following code snippet execute correctly?

```
stack2.remove( 1 );
```

Does this implementation follow the LIFO (Last In First Out) principle of stack?