

Start coding or [generate](#) with AI.

✓ Week 1: Core Java + OOP

Goal: Learn Java basics and object-oriented programming.

Steps:

1. Install Java & IDE (IntelliJ or Eclipse)
2. Learn variables, data types, conditionals, loops
3. Understand classes, objects, methods, constructors

Task: Create a simple Student or Bank Management System in the console using OOP.

Tools:

- ChatGPT: "Java OOP example project"
- YouTube: "Java Full Course – Bro Code"

Start coding or [generate](#) with AI.

Step 1: Install Java & IDE (IntelliJ or Eclipse) - **Completed**

Start coding or [generate](#) with AI.

✓ Task

Create a simple Student or Bank Management System in the console using OOP based on the instructions in `"/content/📁📄 1-Month Full Stack Java Internship – Weekly Tasks.pdf"`.

✓ Design the classes

Subtask:

Identify the main classes needed (e.g., `Student`, `Course`, `Bank Account`, `Customer`) and their attributes and methods.

Reasoning: The subtask is to identify the main classes, attributes, and methods for either a Student Management System or a Bank Management System. A Bank Management System is a good choice for a simple OOP example. The core entities would be `BankAccount` and `Customer`. I will define the attributes and methods for these classes and document them as comments.

```
# Choosing to create a Bank Management System.

# Class: Customer
# Attributes:
# - customerId (String)
# - name (String)
# - address (String)
```

```

# - accounts (List of BankAccount objects)
# Methods:
# - addAccount(BankAccount account)
# - removeAccount(BankAccount account)
# - getAccounts()

# Class: BankAccount
# Attributes:
# - accountNumber (String)
# - balance (double)
# - owner (Customer object)
# Methods:
# - deposit(double amount)
# - withdraw(double amount)
# - getBalance()
# - getAccountNumber()
# - getOwner()

```

Implement the classes

Subtask:

Implement the Java code for each class, including constructors, getters, setters, and other relevant methods.

Reasoning: The subtask requires creating two Java files and implementing the defined classes within them, including constructors, getters, and specific methods. I will create the `Customer.java` and `BankAccount.java` files and add the basic class structure with attributes, constructors, and getter methods as the first part of the implementation.

```

%%writefile Customer.java
import java.util.ArrayList;
import java.util.List;

public class Customer {
    private String customerId;
    private String name;
    private String address;
    private List<BankAccount> accounts;

    public Customer(String customerId, String name, String address) {
        this.customerId = customerId;
        this.name = name;
        this.address = address;
        this.accounts = new ArrayList<>();
    }

    public String getCustomerId() {
        return customerId;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }
}

```

```
        public List<BankAccount> getAccounts() {
            return accounts;
        }
    }
}
```

Writing Customer.java

Reasoning: I have created the `Customer.java` file with basic structure. Now I need to create the `BankAccount.java` file with its basic structure including attributes, constructor, and getter methods as per the instructions.

```
%%writefile BankAccount.java
public class BankAccount {
    private String accountNumber;
    private double balance;
    private Customer owner;

    public BankAccount(String accountNumber, double balance, Customer owner) {
        this.accountNumber = accountNumber;
        this.balance = balance;
        this.owner = owner;
    }

    public String getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public Customer getOwner() {
        return owner;
    }
}
```

Writing BankAccount.java

Reasoning: I have created both `Customer.java` and `BankAccount.java` files with their basic structures. Now I need to implement the remaining methods for each class as described in the instructions: `addAccount` and `removeAccount` for `Customer`, and `deposit`, `withdraw`, and `getBalance` (although `getBalance` is already there as a getter) for `BankAccount`. I will add these methods to the existing files.

```
%%writefile Customer.java
import java.util.ArrayList;
import java.util.List;

public class Customer {
    private String customerId;
    private String name;
    private String address;
    private List<BankAccount> accounts;

    public Customer(String customerId, String name, String address) {
        this.customerId = customerId;
        this.name = name;
    }
}
```

```

        this.address = address;
        this.accounts = new ArrayList<>();
    }

    public String getCustomerId() {
        return customerId;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public List<BankAccount> getAccounts() {
        return accounts;
    }

    public void addAccount(BankAccount account) {
        if (account != null && !this.accounts.contains(account)) {
            this.accounts.add(account);
        }
    }

    public void removeAccount(BankAccount account) {
        if (account != null) {
            this.accounts.remove(account);
        }
    }
}

```

Overwriting Customer.java

Reasoning: I have updated the `Customer.java` file. Now I need to update the `BankAccount.java` file with the `deposit` and `withdraw` methods as per the instructions.

```

%%writefile BankAccount.java
public class BankAccount {
    private String accountNumber;
    private double balance;
    private Customer owner;

    public BankAccount(String accountNumber, double balance, Customer owner) {
        this.accountNumber = accountNumber;
        this.balance = balance;
        this.owner = owner;
    }

    public String getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public Customer getOwner() {
        return owner;
    }
}

```

```

    }

    public void deposit(double amount) {
        if (amount > 0) {
            this.balance += amount;
        } else {
            System.out.println("Deposit amount must be positive.");
        }
    }

    public boolean withdraw(double amount) {
        if (amount > 0 && amount <= this.balance) {
            this.balance -= amount;
            return true;
        } else if (amount > this.balance) {
            System.out.println("Insufficient funds.");
            return false;
        } else {
            System.out.println("Withdrawal amount must be positive.");
            return false;
        }
    }
}

```

Overwriting BankAccount.java

✓ Create the main application

Subtask:

Write the main Java class that will handle user interaction, create objects of the other classes, and implement the system's functionality (e.g., adding students, managing bank accounts).

Reasoning: Create the main Java class `BankManagementSystem.java` with a `main` method to instantiate objects, associate accounts with a customer, and demonstrate deposit/withdrawal functionality.

```

%%writefile BankManagementSystem.java
public class BankManagementSystem {
    public static void main(String[] args) {
        // 3. Inside the main method, instantiate a Customer object and one or more BankAccount object
        Customer customer = new Customer("C123", "Alice Smith", "123 Main St");
        BankAccount savingsAccount = new BankAccount("SA456", 1000.0, customer);
        BankAccount checkingAccount = new BankAccount("CA789", 500.0, customer);

        // 4. Associate the created BankAccount objects with the Customer object using the addAccount
        customer.addAccount(savingsAccount);
        customer.addAccount(checkingAccount);

        // 5. Include basic functionality to demonstrate the system's operation, such as depositing to
        System.out.println("--- Initial Balances ---");
        System.out.println(customer.getName() + "'s Savings Account (" + savingsAccount.getAccountNumb
        System.out.println(customer.getName() + "'s Checking Account (" + checkingAccount.getAccountNu
        System.out.println("-----");

        // Demonstrate deposit
        System.out.println("\nDepositing $200 into Savings Account...");
        savingsAccount.deposit(200.0);
        System.out.println("New Savings Account Balance: $" + savingsAccount.getBalance());
    }
}

```

```

        // Demonstrate withdrawal
        System.out.println("\nWithdrawing $100 from Checking Account...");
        if (checkingAccount.withdraw(100.0)) {
            System.out.println("New Checking Account Balance: $" + checkingAccount.getBalance());
        }

        System.out.println("\nWithdrawing $1000 from Checking Account (Insufficient Funds)...");
        if (checkingAccount.withdraw(1000.0)) {
            System.out.println("New Checking Account Balance: $" + checkingAccount.getBalance());
        }

        // Print final balances
        System.out.println("\n--- Final Balances ---");
        for (BankAccount account : customer.getAccounts()) {
            System.out.println(customer.getName() + "'s Account (" + account.getAccountNumber() + "):");
        }
        System.out.println("-----");
    }
}

```

Writing BankManagementSystem.java

Reasoning: Compile and run the `BankManagementSystem.java` file to verify the implementation and demonstrate the system's functionality.

```

!javac Customer.java BankAccount.java BankManagementSystem.java
!java BankManagementSystem

```

```

--- Initial Balances ---
Alice Smith's Savings Account (SA456): $1000.0
Alice Smith's Checking Account (CA789): $500.0
-----

Depositing $200 into Savings Account...
New Savings Account Balance: $1200.0

Withdrawing $100 from Checking Account...
New Checking Account Balance: $400.0

Withdrawing $1000 from Checking Account (Insufficient Funds)...
Insufficient funds.

--- Final Balances ---
Alice Smith's Account (SA456): $1200.0
Alice Smith's Account (CA789): $400.0
-----

```

Implement the user interface

Subtask:

Create a console-based interface for users to interact with the system (e.g., using `Scanner` for input and `System.out.println` for output).

Reasoning: Modify the `BankManagementSystem.java` file to include a console-based interface with a menu, user input handling using `Scanner`, and a loop to process user choices, implementing the requested operations and input validation.

```

%%writefile BankManagementSystem.java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class BankManagementSystem {
    private static List<Customer> customers = new ArrayList<>();
    private static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        System.out.println("Welcome to the Bank Management System!");

        while (true) {
            printMenu();
            int choice = getUserChoice();

            switch (choice) {
                case 1:
                    createCustomer();
                    break;
                case 2:
                    createAccount();
                    break;
                case 3:
                    deposit();
                    break;
                case 4:
                    withdraw();
                    break;
                case 5:
                    viewAccounts();
                    break;
                case 6:
                    System.out.println("Exiting Bank Management System. Goodbye!");
                    scanner.close();
                    System.exit(0);
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }

    private static void printMenu() {
        System.out.println("\n--- Menu ---");
        System.out.println("1. Create Customer");
        System.out.println("2. Create Account");
        System.out.println("3. Deposit");
        System.out.println("4. Withdraw");
        System.out.println("5. View Accounts");
        System.out.println("6. Exit");
        System.out.print("Enter your choice: ");
    }

    private static int getUserChoice() {
        while (!scanner.hasNextInt()) {
            System.out.println("Invalid input. Please enter a number.");
            scanner.next(); // consume the invalid input
            System.out.print("Enter your choice: ");
        }
        int choice = scanner.nextInt();
        scanner.nextLine(); // consume the newline character
    }
}

```

```

        return choice;
    }

    private static void createCustomer() {
        System.out.println("\n--- Create Customer ---");
        System.out.print("Enter Customer ID: ");
        String customerId = scanner.nextLine();
        System.out.print("Enter Customer Name: ");
        String name = scanner.nextLine();
        System.out.print("Enter Customer Address: ");
        String address = scanner.nextLine();

        Customer newCustomer = new Customer(customerId, name, address);
        customers.add(newCustomer);
        System.out.println("Customer created successfully!");
    }

    private static void createAccount() {
        System.out.println("\n--- Create Account ---");
        System.out.print("Enter Customer ID for the account: ");
        String customerId = scanner.nextLine();

        Customer customer = findCustomerById(customerId);

        if (customer != null) {
            System.out.print("Enter Account Number: ");
            String accountNumber = scanner.nextLine();
            System.out.print("Enter Initial Balance: ");
            while (!scanner.hasNextDouble()) {
                System.out.println("Invalid input. Please enter a valid number for the balance.");
                scanner.next(); // consume the invalid input
                System.out.print("Enter Initial Balance: ");
            }
            double initialBalance = scanner.nextDouble();
            scanner.nextLine(); // consume the newline

            BankAccount newAccount = new BankAccount(accountNumber, initialBalance, customer);
            customer.addAccount(newAccount);
            System.out.println("Account created successfully for " + customer.getName());
        } else {
            System.out.println("Customer not found with ID: " + customerId);
        }
    }

    private static void deposit() {
        System.out.println("\n--- Deposit ---");
        System.out.print("Enter Account Number: ");
        String accountNumber = scanner.nextLine();

        BankAccount account = findAccountByNumber(accountNumber);

        if (account != null) {
            System.out.print("Enter amount to deposit: ");
            while (!scanner.hasNextDouble()) {
                System.out.println("Invalid input. Please enter a valid number for the amount.");
                scanner.next(); // consume the invalid input
                System.out.print("Enter amount to deposit: ");
            }
            double amount = scanner.nextDouble();
            scanner.nextLine(); // consume the newline

            account.deposit(amount);
        }
    }

```



```

        System.out.println("Deposit successful. New balance: $" + account.getBalance());
    } else {
        System.out.println("Account not found with number: " + accountNumber);
    }
}

private static void withdraw() {
    System.out.println("\n--- Withdraw ---");
    System.out.print("Enter Account Number: ");
    String accountNumber = scanner.nextLine();

    BankAccount account = findAccountByNumber(accountNumber);

    if (account != null) {
        System.out.print("Enter amount to withdraw: ");
        while (!scanner.hasNextDouble()) {
            System.out.println("Invalid input. Please enter a valid number for the amount.");
            scanner.next(); // consume the invalid input
            System.out.print("Enter amount to withdraw: ");
        }
        double amount = scanner.nextDouble();
        scanner.nextLine(); // consume the newline

        if (account.withdraw(amount)) {
            System.out.println("Withdrawal successful. New balance: $" + account.getBalance());
        }
    } else {
        System.out.println("Account not found with number: " + accountNumber);
    }
}

private static void viewAccounts() {
    System.out.println("\n--- View Accounts ---");
    System.out.print("Enter Customer ID: ");
    String customerId = scanner.nextLine();

    Customer customer = findCustomerById(customerId);

    if (customer != null) {
        List<BankAccount> accounts = customer.getAccounts();
        if (accounts.isEmpty()) {
            System.out.println(customer.getName() + " has no accounts.");
        } else {
            System.out.println(customer.getName() + "'s Accounts:");
            for (BankAccount account : accounts) {
                System.out.println("    Account Number: " + account.getAccountNumber() + ", Balance:");
            }
        }
    } else {
        System.out.println("Customer not found with ID: " + customerId);
    }
}

private static Customer findCustomerById(String customerId) {
    for (Customer customer : customers) {
        if (customer.getCustomerId().equals(customerId)) {
            return customer;
        }
    }
    return null;
}

```

```

        private static BankAccount findAccountByNumber(String accountNumber) {
            for (Customer customer : customers) {
                for (BankAccount account : customer.getAccounts()) {
                    if (account.getAccountNumber().equals(accountNumber)) {
                        return account;
                    }
                }
            }
            return null;
        }
    }
}

```

Overwriting BankManagementSystem.java

Reasoning: Compile and run the updated BankManagementSystem.java file to test the console-based interface.

```

!javac Customer.java BankAccount.java BankManagementSystem.java
!java BankManagementSystem

```

```

--- Create Customer ---
Enter Customer ID: 8970
Enter Customer Name: Nitish Gowda G
Enter Customer Address: Tumkur
Customer created successfully!

--- Menu ---
1. Create Customer
2. Create Account

```

Enter amount to withdraw: 5000

```
Enter amount to withdraw: 5000
Withdrawal successful. New balance: $5000.0

--- Menu ---
1. Create Customer
2. Create Account
3. Deposit
4. Withdraw
5. View Accounts
6. Exit
Enter your choice: 6
Exiting Bank Management System. Goodbye!
```

✓ Task

Test the Bank Management System code.

✓ Create a test class

Subtask:

Create a new Java class specifically for testing the `Customer` and `BankAccount` classes.

Reasoning: Create a new Java file named `BankSystemTest.java` and add the basic class structure with a `main` method as requested by the instructions.

```
%writefile BankSystemTest.java
public class BankSystemTest {
    public static void main(String[] args) {
        // Test code will go here
    }
}
```

Writing BankSystemTest.java

✓ Set up testing framework

Subtask:

If you want to use a testing framework like JUnit, you'll need to set it up. For this simple system, you can write basic tests without a framework.

Reasoning: Add comments to the `main` method in `BankSystemTest.java` to indicate where tests for the `Customer` and `BankAccount` classes will be placed as per the instructions.

```
%%writefile BankSystemTest.java
public class BankSystemTest {
    public static void main(String[] args) {
        // --- Test Customer Class ---

        // --- Test BankAccount Class ---
    }
}
```

Overwriting BankSystemTest.java

✓ Write test methods for `customer`

Subtask:

Write test methods for `customer`

Reasoning: Implement the test cases for the `Customer` class within the `main` method of `BankSystemTest.java`, including instantiating a customer, testing getters, adding accounts, testing `addAccount` with edge cases, removing an account, and testing `removeAccount` with edge cases.

```
%%writefile BankSystemTest.java
import java.util.List;

public class BankSystemTest {
    public static void main(String[] args) {
        // --- Test Customer Class ---
        System.out.println("--- Testing Customer Class ---");

        // 2. Instantiate a Customer object and print results of getter methods
        Customer customer = new Customer("C123", "Alice Smith", "123 Main St");
        System.out.println("Customer ID: " + customer.getCustomerId());
        System.out.println("Customer Name: " + customer.getName());
        System.out.println("Customer Address: " + customer.getAddress());
        System.out.println("Customer Accounts (initial): " + customer.getAccounts());

        // 3. Create one or more BankAccount objects.
        BankAccount account1 = new BankAccount("A001", 1000.0, customer);
        BankAccount account2 = new BankAccount("A002", 500.0, customer);

        // 4. Call addAccount method and print messages
        System.out.println("\nAdding account1...");
        customer.addAccount(account1);
        System.out.println("Accounts after adding account1: " + customer.getAccounts().size());

        System.out.println("Adding account2...");
        customer.addAccount(account2);
        System.out.println("Accounts after adding account2: " + customer.getAccounts().size());

        // 5. Print list of accounts after adding
        System.out.println("\nAccounts after adding:");
        List<BankAccount> accounts = customer.getAccounts();
        for (BankAccount acc : accounts) {
            System.out.println("  Account Number: " + acc.getAccountNumber() + ", Balance: $" + acc.ge

        // 6. Test addAccount with null and duplicate
        System.out.println("\nAttempting to add null account...");
```

```

customer.addAccount(null);
System.out.println("Accounts after attempting to add null: " + customer.getAccounts().size());

System.out.println("Attempting to add duplicate account1...");
customer.addAccount(account1);
System.out.println("Accounts after attempting to add duplicate: " + customer.getAccounts().size());

// 7. Call removeAccount and print message
System.out.println("\nRemoving account1...");
customer.removeAccount(account1);
System.out.println("Accounts after removing account1: " + customer.getAccounts().size());

// 8. Print list of accounts after removing
System.out.println("\nAccounts after removing account1:");
accounts = customer.getAccounts();
for (BankAccount acc : accounts) {
    System.out.println("  Account Number: " + acc.getAccountNumber() + ", Balance: $" + acc.getBalance());
}

// 9. Test removeAccount with null and non-existent
System.out.println("\nAttempting to remove null account...");
customer.removeAccount(null);
System.out.println("Accounts after attempting to remove null: " + customer.getAccounts().size());

BankAccount nonExistentAccount = new BankAccount("A999", 0.0, customer);
System.out.println("Attempting to remove non-existent account (A999)...");
customer.removeAccount(nonExistentAccount);
System.out.println("Accounts after attempting to remove non-existent: " + customer.getAccounts().size());

System.out.println("\n--- Finished Testing Customer Class ---");

// --- Test BankAccount Class ---
System.out.println("\n--- Testing BankAccount Class (Placeholder) ---");
// Test code for BankAccount will be added later
System.out.println("--- Finished Testing BankAccount Class (Placeholder) ---");
}
}

```

Overwriting BankSystemTest.java

Reasoning: Compile and run the `BankSystemTest.java` file to execute the test cases for the `Customer` class and observe the output to verify the expected behavior of the methods.

```

!javac Customer.java BankAccount.java BankSystemTest.java
!java BankSystemTest

```

```

--- Testing Customer Class ---
Customer ID: C123
Customer Name: Alice Smith
Customer Address: 123 Main St
Customer Accounts (initial): []

Adding account1...
Accounts after adding account1: 1
Adding account2...
Accounts after adding account2: 2

Accounts after adding:
  Account Number: A001, Balance: $1000.0
  Account Number: A002, Balance: $500.0

```

```

Attempting to add null account...
Accounts after attempting to add null: 2
Attempting to add duplicate account1...
Accounts after attempting to add duplicate: 2

Removing account1...
Accounts after removing account1: 1

Accounts after removing account1:
    Account Number: A002, Balance: $500.0

Attempting to remove null account...
Accounts after attempting to remove null: 1
Attempting to remove non-existent account (A999)...
Accounts after attempting to remove non-existent: 1

--- Finished Testing Customer Class ---

--- Testing BankAccount Class (Placeholder) ---
--- Finished Testing BankAccount Class (Placeholder) ---

```

✓ Write test methods for `bankaccount`

Subtask:

Write test methods for `bankaccount`

Reasoning: The subtask is to write test methods for the `BankAccount` class. I will modify the `BankSystemTest.java` file to include the tests for `BankAccount` as outlined in the instructions, which involve instantiating a `BankAccount`, testing deposit with positive and negative amounts, and testing withdraw with valid, insufficient, and negative amounts.

```

%%writefile BankSystemTest.java
import java.util.List;

public class BankSystemTest {
    public static void main(String[] args) {
        // --- Test Customer Class ---
        System.out.println("--- Testing Customer Class ---");

        // 2. Instantiate a Customer object and print results of getter methods
        Customer customer = new Customer("C123", "Alice Smith", "123 Main St");
        System.out.println("Customer ID: " + customer.getCustomerId());
        System.out.println("Customer Name: " + customer.getName());
        System.out.println("Customer Address: " + customer.getAddress());
        System.out.println("Customer Accounts (initial): " + customer.getAccounts());

        // 3. Create one or more BankAccount objects.
        BankAccount account1 = new BankAccount("A001", 1000.0, customer);
        BankAccount account2 = new BankAccount("A002", 500.0, customer);

        // 4. Call addAccount method and print messages
        System.out.println("\nAdding account1...");
        customer.addAccount(account1);
        System.out.println("Accounts after adding account1: " + customer.getAccounts().size());

        System.out.println("Adding account2...");
        customer.addAccount(account2);
    }
}

```

```

System.out.println("Accounts after adding account2: " + customer.getAccounts().size());

// 5. Print list of accounts after adding
System.out.println("\nAccounts after adding:");
List<BankAccount> accounts = customer.getAccounts();
for (BankAccount acc : accounts) {
    System.out.println("  Account Number: " + acc.getAccountNumber() + ", Balance: $" + acc.ge
}

// 6. Test addAccount with null and duplicate
System.out.println("\nAttempting to add null account...");
customer.addAccount(null);
System.out.println("Accounts after attempting to add null: " + customer.getAccounts().size());

System.out.println("Attempting to add duplicate account1...");
customer.addAccount(account1);
System.out.println("Accounts after attempting to add duplicate: " + customer.getAccounts().siz

// 7. Call removeAccount and print message
System.out.println("\nRemoving account1...");
customer.removeAccount(account1);
System.out.println("Accounts after removing account1: " + customer.getAccounts().size());

// 8. Print list of accounts after removing
System.out.println("\nAccounts after removing account1:");
accounts = customer.getAccounts();
for (BankAccount acc : accounts) {
    System.out.println("  Account Number: " + acc.getAccountNumber() + ", Balance: $" + acc.ge
}

// 9. Test removeAccount with null and non-existent
System.out.println("\nAttempting to remove null account...");
customer.removeAccount(null);
System.out.println("Accounts after attempting to remove null: " + customer.getAccounts().size(

BankAccount nonExistentAccount = new BankAccount("A999", 0.0, customer);
System.out.println("Attempting to remove non-existent account (A999)...");
customer.removeAccount(nonExistentAccount);
System.out.println("Accounts after attempting to remove non-existent: " + customer.getAccounts

System.out.println("\n--- Finished Testing Customer Class ---");

// --- Test BankAccount Class ---
System.out.println("\n--- Testing BankAccount Class ---");

// 2. Instantiate a BankAccount object with an initial balance.
BankAccount testAccount = new BankAccount("B001", 500.0, customer);
System.out.println("Instantiated BankAccount with initial balance.");

// 3. Print the initial balance using the getBalance method.
System.out.println("Initial Balance: $" + testAccount.getBalance());

// 4. Call the deposit method with a positive amount and print the new balance.
double depositAmount = 100.0;
System.out.println("\nAttempting to deposit $" + depositAmount);
testAccount.deposit(depositAmount);
System.out.println("Balance after depositing $" + depositAmount + ": $" + testAccount.getBalan

// 5. Call the deposit method with a negative amount and observe the output (it should print a
double negativeDepositAmount = -50.0;

```

```

        System.out.println("\nAttempting to deposit $" + negativeDepositAmount + " (expected error)");
        testAccount.deposit(negativeDepositAmount);
        System.out.println("Balance after attempting to deposit $" + negativeDepositAmount + ": $" + t

// 6. Call the withdraw method with a valid amount (less than or equal to the current balance)
double validWithdrawAmount = 200.0;
System.out.println("\nAttempting to withdraw $" + validWithdrawAmount);
boolean withdrawSuccess = testAccount.withdraw(validWithdrawAmount);
System.out.println("Withdrawal successful: " + withdrawSuccess);
System.out.println("Balance after withdrawing $" + validWithdrawAmount + ": $" + testAccount.g

// 7. Call the withdraw method with an amount greater than the current balance and check the r
double excessiveWithdrawAmount = 1000.0;
System.out.println("\nAttempting to withdraw $" + excessiveWithdrawAmount + " (expected insuff
boolean excessiveWithdrawSuccess = testAccount.withdraw(excessiveWithdrawAmount);
System.out.println("Withdrawal successful: " + excessiveWithdrawSuccess); // Should be false
System.out.println("Balance after attempting to withdraw $" + excessiveWithdrawAmount + ": $"

// 8. Call the withdraw method with a negative amount and observe the output (it should print
double negativeWithdrawAmount = -50.0;
System.out.println("\nAttempting to withdraw $" + negativeWithdrawAmount + " (expected error)"
boolean negativeWithdrawSuccess = testAccount.withdraw(negativeWithdrawAmount);
System.out.println("Withdrawal successful: " + negativeWithdrawSuccess); // Should be false
System.out.println("Balance after attempting to withdraw $" + negativeWithdrawAmount + ": $" +

        System.out.println("\n--- Finished Testing BankAccount Class ---");
    }
}

```

Overwriting BankSystemTest.java

Reasoning: I have updated the `BankSystemTest.java` file with the test methods for the `BankAccount` class. Now I need to compile and run this file to execute the tests and verify the functionality of the `BankAccount` class.

```

!javac Customer.java BankAccount.java BankSystemTest.java
!java BankSystemTest

```

```

Customer ID: C123
Customer Name: Alice Smith
Customer Address: 123 Main St
Customer Accounts (initial): []

```

```

Adding account1...
Accounts after adding account1: 1
Adding account2...
Accounts after adding account2: 2

```

```

Accounts after adding:
  Account Number: A001, Balance: $1000.0
  Account Number: A002, Balance: $500.0

```

```

Attempting to add null account...
Accounts after attempting to add null: 2
Attempting to add duplicate account1...
Accounts after attempting to add duplicate: 2

```

```

Removing account1...
Accounts after removing account1: 1

```

Account Number: A002, Balance: \$500.0


```

Account Number: A002, Balance: $500.0

Attempting to remove null account...
Accounts after attempting to remove null: 1
Attempting to remove non-existent account (A999)...
Accounts after attempting to remove non-existent: 1

--- Finished Testing Customer Class ---

--- Testing BankAccount Class ---
Instantiated BankAccount with initial balance.
Initial Balance: $500.0

Attempting to deposit $100.0
Balance after depositing $100.0: $600.0

Attempting to deposit $-50.0 (expected error)
Deposit amount must be positive.
Balance after attempting to deposit $-50.0: $600.0

Attempting to withdraw $200.0
Withdrawal successful: true
Balance after withdrawing $200.0: $400.0

Attempting to withdraw $1000.0 (expected insufficient funds)
Insufficient funds.
Withdrawal successful: false
Balance after attempting to withdraw $1000.0: $400.0

Attempting to withdraw $-50.0 (expected error)
Withdrawal amount must be positive.
Withdrawal successful: false
Balance after attempting to withdraw $-50.0: $400.0

--- Finished Testing BankAccount Class ---

```

✓ Run the tests

Subtask:

Compile and run the test class to execute the test methods and check for failures.

Reasoning: Compile and run the test class to execute the test methods and check for failures.

```

!javac Customer.java BankAccount.java BankSystemTest.java
!java BankSystemTest

```

```

Customer ID: C123
Customer Name: Alice Smith
Customer Address: 123 Main St
Customer Accounts (initial): []

Adding account1...
Accounts after adding account1: 1
Adding account2...
Accounts after adding account2: 2

Accounts after adding:
  Account Number: A001, Balance: $1000.0
  Account Number: A002, Balance: $500.0

Attempting to add null account...
Accounts after attempting to add null: 2

```

```

Accounts after attempting to add duplicate: 2

Removing account1...
Accounts after removing account1: 1

Accounts after removing account1:
  Account Number: A002, Balance: $500.0

Attempting to remove null account...
Accounts after attempting to remove null: 1
Attempting to remove non-existent account (A999)...
Accounts after attempting to remove non-existent: 1

--- Finished Testing Customer Class ---

--- Testing BankAccount Class ---
Instantiated BankAccount with initial balance.
Initial Balance: $500.0

Attempting to deposit $100.0
Balance after depositing $100.0: $600.0

Attempting to deposit $-50.0 (expected error)
Deposit amount must be positive.
Balance after attempting to deposit $-50.0: $600.0

Attempting to withdraw $200.0
Withdrawal successful: true
Balance after withdrawing $200.0: $400.0

Attempting to withdraw $1000.0 (expected insufficient funds)
Insufficient funds.
Withdrawal successful: false
Balance after attempting to withdraw $1000.0: $400.0

Attempting to withdraw $-50.0 (expected error)
Withdrawal amount must be positive.
Withdrawal successful: false
Balance after attempting to withdraw $-50.0: $400.0

--- Finished Testing BankAccount Class ---

```

Analyze test results

Subtask:

Analyze the output of the test execution to confirm that all tests passed and that the `Customer` and `BankAccount` classes are functioning as expected based on the implemented test cases.

✓ Refine tests

Subtask:

Add more test cases to cover different scenarios and edge cases.

Reasoning: The subtask is to add more test cases to the `BankSystemTest.java` file as per the instructions, covering additional scenarios and edge cases for both the `Customer` and `BankAccount` classes and including assertions or print statements for verification. I will modify the existing `BankSystemTest.java` file to include these new tests.

```

%%writefile BankSystemTest.java
import java.util.List;
import java.util.ArrayList; // Import ArrayList

public class BankSystemTest {
    public static void main(String[] args) {
        // --- Test Customer Class ---
        System.out.println("--- Testing Customer Class ---");

        // 2. Instantiate a Customer object and print results of getter methods
        Customer customer = new Customer("C123", "Alice Smith", "123 Main St");
        System.out.println("Customer ID: " + customer.getCustomerId());
        System.out.println("Customer Name: " + customer.getName());
        System.out.println("Customer Address: " + customer.getAddress());
        System.out.println("Customer Accounts (initial): " + customer.getAccounts());

        // 3. Create one or more BankAccount objects.
        BankAccount account1 = new BankAccount("A001", 1000.0, customer);
        BankAccount account2 = new BankAccount("A002", 500.0, customer);

        // 4. Call addAccount method and print messages
        System.out.println("\nAdding account1...");
        customer.addAccount(account1);
        System.out.println("Accounts after adding account1: " + customer.getAccounts().size());

        System.out.println("Adding account2...");
        customer.addAccount(account2);
        System.out.println("Accounts after adding account2: " + customer.getAccounts().size());

        // 5. Print list of accounts after adding
        System.out.println("\nAccounts after adding:");
        List<BankAccount> accounts = customer.getAccounts();
        for (BankAccount acc : accounts) {
            System.out.println("  Account Number: " + acc.getAccountNumber() + ", Balance: $" + acc.ge
        }

        // 6. Test addAccount with null and duplicate
        System.out.println("\nAttempting to add null account...");
        customer.addAccount(null);
        System.out.println("Accounts after attempting to add null: " + customer.getAccounts().size());

        System.out.println("Attempting to add duplicate account1...");
        customer.addAccount(account1);
        System.out.println("Accounts after attempting to add duplicate: " + customer.getAccounts().siz

        // --- New Customer Tests ---

        // Getting accounts from a customer with no accounts.
        System.out.println("\n--- New Customer Tests ---");
        Customer newCustomer = new Customer("C456", "Bob Johnson", "456 Oak Ave");
        System.out.println("Customer " + newCustomer.getName() + "'s accounts (initial): " + newCustom
        if (newCustomer.getAccounts().isEmpty()) {
            System.out.println("Test Passed: Customer with no accounts returns an empty list.");
        } else {
            System.out.println("Test Failed: Customer with no accounts did not return an empty list.")
        }

        // Attempting to remove the same account multiple times.
        System.out.println("\nAttempting to remove account2 multiple times...");
        int initialAccountCount = customer.getAccounts().size();
    }
}

```

```

customer.removeAccount(account2);
System.out.println("Accounts after first removal of account2: " + customer.getAccounts().size());
customer.removeAccount(account2); // Attempt removal again
System.out.println("Accounts after second removal of account2: " + customer.getAccounts().size());
if (customer.getAccounts().size() == initialAccountCount - 1) {
    System.out.println("Test Passed: Removing the same account multiple times only removes it");
} else {
    System.out.println("Test Failed: Removing the same account multiple times had unexpected");
}

// Adding and then immediately removing an account.
System.out.println("\nAdding and then immediately removing account3...");
BankAccount account3 = new BankAccount("A003", 2000.0, customer);
customer.addAccount(account3);
System.out.println("Accounts after adding account3: " + customer.getAccounts().size());
customer.removeAccount(account3);
System.out.println("Accounts after removing account3: " + customer.getAccounts().size());
if (!customer.getAccounts().contains(account3) && customer.getAccounts().size() == initialAccountCount) {
    System.out.println("Test Passed: Adding and immediately removing an account works correct");
} else {
    System.out.println("Test Failed: Adding and immediately removing an account had unexpected");
}

// 7. Call removeAccount and print message (Original test, leaving for context)
System.out.println("\nRemoving account1 (Original test)...");
customer.removeAccount(account1);
System.out.println("Accounts after removing account1: " + customer.getAccounts().size());

// 8. Print list of accounts after removing (Original test, leaving for context)
System.out.println("\nAccounts after removing account1:");
accounts = customer.getAccounts();
for (BankAccount acc : accounts) {
    System.out.println("  Account Number: " + acc.getAccountNumber() + ", Balance: $" + acc.getBalance());
}

// 9. Test removeAccount with null and non-existent (Original test, leaving for context)
System.out.println("\nAttempting to remove null account (Original test)...");
customer.removeAccount(null);
System.out.println("Accounts after attempting to remove null: " + customer.getAccounts().size());

BankAccount nonExistentAccount = new BankAccount("A999", 0.0, customer);
System.out.println("Attempting to remove non-existent account (A999 - Original test)...");
customer.removeAccount(nonExistentAccount);
System.out.println("Accounts after attempting to remove non-existent: " + customer.getAccounts().size());

System.out.println("\n--- Finished Testing Customer Class ---");

// --- Test BankAccount Class ---
System.out.println("\n--- Testing BankAccount Class ---");

// 2. Instantiate a BankAccount object with an initial balance.
BankAccount testAccount = new BankAccount("B001", 500.0, customer);
System.out.println("Instantiated BankAccount with initial balance.");

// 3. Print the initial balance using the getBalance method.
System.out.println("Initial Balance: $" + testAccount.getBalance());

// 4. Call the deposit method with a positive amount and print the new balance.

```

```

double depositAmount = 100.0;
System.out.println("\nAttempting to deposit $" + depositAmount);
testAccount.deposit(depositAmount);
System.out.println("Balance after depositing $" + depositAmount + ": $" + testAccount.getBalance());

// 5. Call the deposit method with a negative amount and observe the output (it should print a
double negativeDepositAmount = -50.0;
System.out.println("\nAttempting to deposit $" + negativeDepositAmount + " (expected error)");
testAccount.deposit(negativeDepositAmount);
System.out.println("Balance after attempting to deposit $" + negativeDepositAmount + ": $" + testAccount.getBalance());

// 6. Call the withdraw method with a valid amount (less than or equal to the current balance)
double validWithdrawAmount = 200.0;
System.out.println("\nAttempting to withdraw $" + validWithdrawAmount);
boolean withdrawSuccess = testAccount.withdraw(validWithdrawAmount);
System.out.println("Withdrawal successful: " + withdrawSuccess);
System.out.println("Balance after withdrawing $" + validWithdrawAmount + ": $" + testAccount.getBalance());

// 7. Call the withdraw method with an amount greater than the current balance and check the result
double excessiveWithdrawAmount = 1000.0;
System.out.println("\nAttempting to withdraw $" + excessiveWithdrawAmount + " (expected insufficient funds error)");
boolean excessiveWithdrawSuccess = testAccount.withdraw(excessiveWithdrawAmount);
System.out.println("Withdrawal successful: " + excessiveWithdrawSuccess); // Should be false
System.out.println("Balance after attempting to withdraw $" + excessiveWithdrawAmount + ": $" + testAccount.getBalance());

// 8. Call the withdraw method with a negative amount and observe the output (it should print an error)
double negativeWithdrawAmount = -50.0;
System.out.println("\nAttempting to withdraw $" + negativeWithdrawAmount + " (expected error)");
boolean negativeWithdrawSuccess = testAccount.withdraw(negativeWithdrawAmount);
System.out.println("Withdrawal successful: " + negativeWithdrawSuccess); // Should be false
System.out.println("Balance after attempting to withdraw $" + negativeWithdrawAmount + ": $" + testAccount.getBalance());

// --- New BankAccount Tests ---

// Depositing zero amount.
System.out.println("\n--- New BankAccount Tests ---");
double zeroDeposit = 0.0;
double balanceBeforeZeroDeposit = testAccount.getBalance();
System.out.println("Attempting to deposit $" + zeroDeposit);
testAccount.deposit(zeroDeposit);
System.out.println("Balance after depositing $" + zeroDeposit + ": $" + testAccount.getBalance());
if (testAccount.getBalance() == balanceBeforeZeroDeposit) {
    System.out.println("Test Passed: Depositing zero amount does not change balance.");
} else {
    System.out.println("Test Failed: Depositing zero amount changed balance.");
}

// Withdrawing zero amount.
double zeroWithdraw = 0.0;
double balanceBeforeZeroWithdraw = testAccount.getBalance();
System.out.println("\nAttempting to withdraw $" + zeroWithdraw);
boolean zeroWithdrawSuccess = testAccount.withdraw(zeroWithdraw);
System.out.println("Withdrawal successful: " + zeroWithdrawSuccess);
System.out.println("Balance after withdrawing $" + zeroWithdraw + ": $" + testAccount.getBalance());
if (testAccount.getBalance() == balanceBeforeZeroWithdraw && !zeroWithdrawSuccess) {
    System.out.println("Test Passed: Withdrawing zero amount does not change balance and return false");
} else {
    System.out.println("Test Failed: Withdrawing zero amount had unexpected behavior.");
}

// Creating a BankAccount with an initial balance of zero or a negative value

```

```

System.out.println("\nTesting initial balance of zero...");
BankAccount zeroBalanceAccount = new BankAccount("B002", 0.0, customer);
System.out.println("Initial balance of B002: $" + zeroBalanceAccount.getBalance());
if (zeroBalanceAccount.getBalance() == 0.0) {
    System.out.println("Test Passed: Initial balance of zero is set correctly.");
} else {
    System.out.println("Test Failed: Initial balance of zero was not set correctly.");
}

System.out.println("\nTesting initial balance of negative value...");
BankAccount negativeBalanceAccount = new BankAccount("B003", -100.0, customer);
System.out.println("Initial balance of B003: $" + negativeBalanceAccount.getBalance());
// Assuming the constructor doesn't modify negative initial balance, check if it's the same.
if (negativeBalanceAccount.getBalance() == -100.0) {
    System.out.println("Test Passed: Initial balance of negative value is set correctly (as n
} else {
    System.out.println("Test Failed: Initial balance of negative value was not set correctly.
}

// Performing multiple deposits and withdrawals in sequence.
System.out.println("\nPerforming multiple deposits and withdrawals in sequence...");
BankAccount sequenceAccount = new BankAccount("B004", 100.0, customer);
System.out.println("Initial balance of B004: $" + sequenceAccount.getBalance());

sequenceAccount.deposit(50.0); // Balance should be 150.0
System.out.println("Balance after deposit 1: $" + sequenceAccount.getBalance());
sequenceAccount.withdraw(30.0); // Balance should be 120.0
System.out.println("Balance after withdrawal 1: $" + sequenceAccount.getBalance());
sequenceAccount.deposit(200.0); // Balance should be 320.0
System.out.println("Balance after deposit 2: $" + sequenceAccount.getBalance());
sequenceAccount.withdraw(150.0); // Balance should be 170.0
System.out.println("Balance after withdrawal 2: $" + sequenceAccount.getBalance());

if (sequenceAccount.getBalance() == 170.0) {
    System.out.println("Test Passed: Multiple deposits and withdrawals sequence resulted in c
} else {
    System.out.println("Test Failed: Multiple deposits and withdrawals sequence resulted in i
}

System.out.println("\n--- Finished Testing BankAccount Class ---");
}
}

```

Overwriting BankSystemTest.java

Reasoning: I have updated the `BankSystemTest.java` file with the new test cases. Now I need to compile and run the test class to execute the tests and check if the code behaves as expected for the added scenarios and edge cases.

```

!javac Customer.java BankAccount.java BankSystemTest.java
!java BankSystemTest

```

```

Attempting to deposit $100.0
Balance after depositing $100.0: $600.0

Attempting to deposit $-50.0 (expected error)
Deposit amount must be positive.
Balance after attempting to deposit $-50.0: $600.0

Attempting to withdraw $200.0
Withdrawal successful: true
Balance after withdrawing $200.0: $400.0

Attempting to withdraw $1000.0 (expected insufficient funds)
Insufficient funds.
Withdrawal successful: false
Balance after attempting to withdraw $1000.0: $400.0

Attempting to withdraw $-50.0 (expected error)
Withdrawal amount must be positive.
Withdrawal successful: false
Balance after attempting to withdraw $-50.0: $400.0

--- New BankAccount Tests ---
Attempting to deposit $0.0
Deposit amount must be positive.
Balance after depositing $0.0: $400.0
Test Passed: Depositing zero amount does not change balance.

Attempting to withdraw $0.0
Withdrawal amount must be positive.
Withdrawal successful: false
Balance after withdrawing $0.0: $400.0
Test Passed: Withdrawing zero amount does not change balance and returns false.

Testing initial balance of zero...
Initial balance of B002: $0.0
Test Passed: Initial balance of zero is set correctly.

Testing initial balance of negative value...
Initial balance of B003: $-100.0
Test Passed: Initial balance of negative value is set correctly (as negative).

Performing multiple deposits and withdrawals in sequence...
Initial balance of B004: $100.0
Balance after deposit 1: $150.0
Balance after withdrawal 1: $120.0
Balance after deposit 2: $320.0
Balance after withdrawal 2: $170.0
Test Passed: Multiple deposits and withdrawals sequence resulted in correct final balance.

--- Finished Testing BankAccount Class ---

```

Summary:

Data Analysis Key Findings

- A `BankSystemTest.java` file was created to test the `Customer` and `BankAccount` classes.
- Initial tests for the `Customer` class verified the constructor, getter methods, `addAccount`, and `removeAccount` methods, including handling `null` and duplicate/non-existent accounts.
- Initial tests for the `BankAccount` class verified the constructor, `deposit`, and `withdraw` methods, including handling positive/negative deposits and valid/excessive/negative withdrawals.

- Refined tests for the `Customer` class confirmed that getting accounts from a customer with no accounts returns an empty list, removing the same account multiple times only removes it once, and adding then immediately removing an account works correctly.
- Refined tests for the `BankAccount` class confirmed that depositing/withdrawing zero amounts does not change the balance, initial balances of zero or negative values are set correctly, and multiple deposits and withdrawals in sequence result in the correct final balance.
- All tests compiled and ran successfully, indicating that the `Customer` and `BankAccount` classes function as expected based on the implemented test cases.

Insights or Next Steps

- Consider implementing constructor validation for `BankAccount` to prevent negative initial balances if that is not desired behavior.
- Explore using a testing framework like JUnit to structure tests more formally and provide richer reporting.