Low Level Design: Blog

Introduction:

An online shopping system, similar to Amazon, is a web-based application that empowers customers to make purchases online. This dynamic system encompasses key functionalities, including product exploration, a shopping cart, secure checkout processes, order management, account administration, and real-time shipping status updates.

To ensure a successful implementation, the system must meet a range of functional and non-functional requirements, spanning performance, security, scalability, usability, reliability, compatibility, and maintainability. It's worth noting that the system can be constructed using various programming languages, with Java being a prominent choice for its versatility and robust ecosystem.

Functionalities of Online Shopping System:

I have added here a few functions for understand of the basic concepts of OOPs in Java. You can add more such functionalities as per your requirements.

1. To browse and search for products
2. To add items to a shopping cart
3. To add new Products
4. To ability to place an order
5. To make a payment
6. To view order history and status
7. To track the shipping method of an order

Entities and Relations in Online Shopping System:

Entities and relationships are fundamental concepts in system design, enabling the modeling of real-world objects and their interactions.

Entities, akin to real-world objects, are integral components we aim to replicate within the system.

On the other hand, relationships serve to define connections and interactions between these entities.

Imagine entities as nouns and relationships as verbs in the system's narrative. The primary objective behind modelling entities and relationships is to capture the essence of real-world objects and their dynamics while representing them in a comprehensible and maintainable manner.

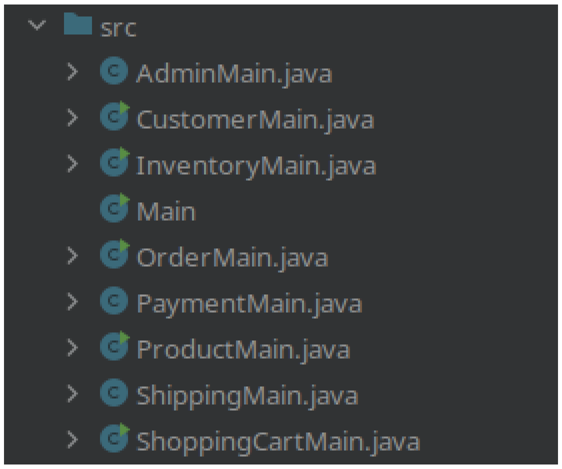
These entities and their relationships serve as the bedrock of the online shopping system. Crafting them with precision and care during the design and implementation phases is crucial to ensure the system's seamless functionality and scalability.

Code and Explanation In Java:

Getters and Setters:

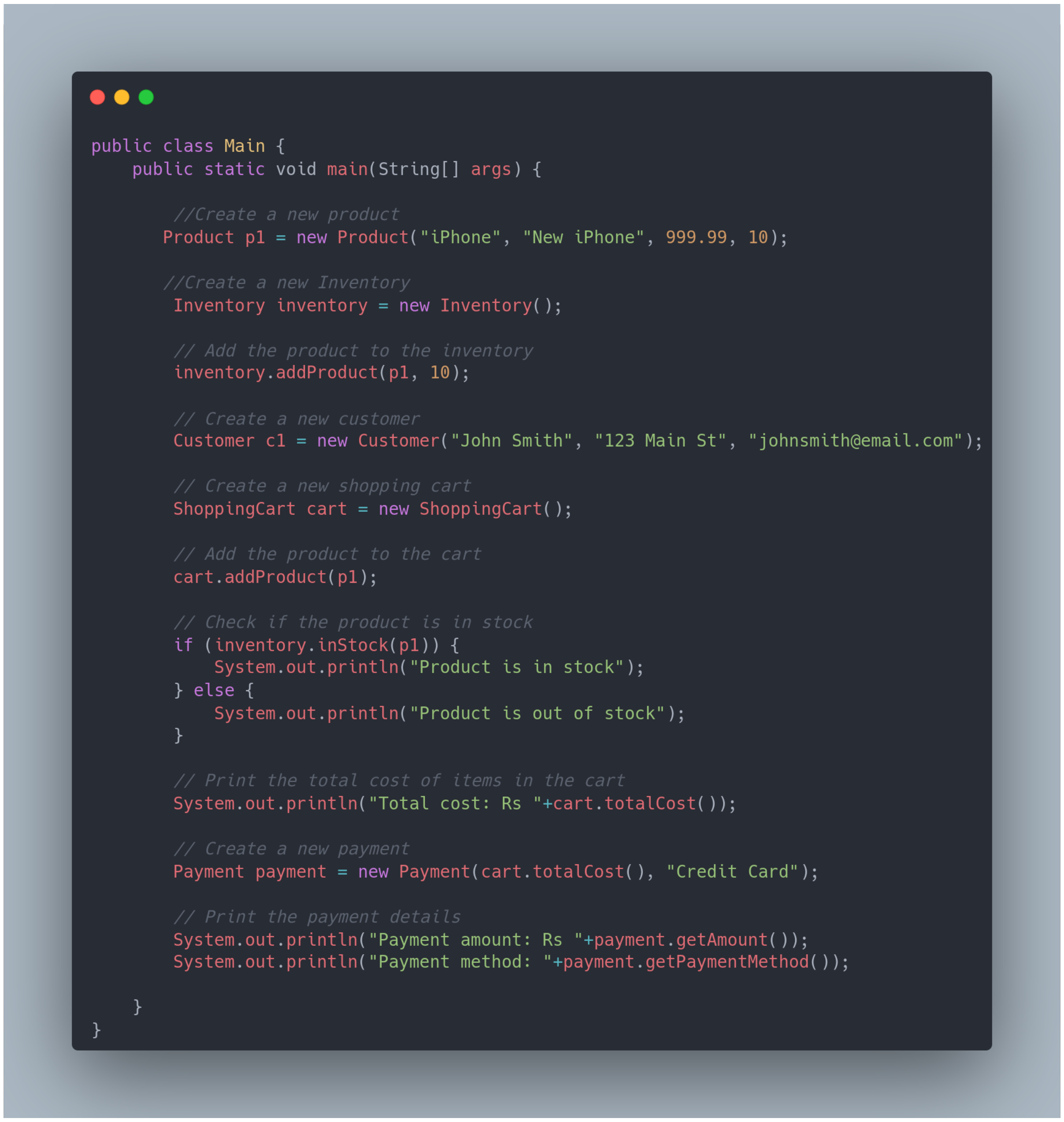
Getters and setters are methods in object-oriented programming used to access and modify the private fields (attributes) of a class, encapsulating the internal state of an object. Getters, often prefixed with "get," retrieve the values of these private fields, providing read-only access. Setters, usually prefixed with "set," allow for the controlled modification of these fields, ensuring data integrity and adhering to the principles of encapsulation. By using getters and setters, developers can enforce access control, validation, and maintainability, enhancing the robustness and security of their code while providing a standardized way to interact with object properties.

1) Classes Structure:



```
src
  AdminMain.java
  CustomerMain.java
  InventoryMain.java
  Main
  OrderMain.java
  PaymentMain.java
  ProductMain.java
  ShippingMain.java
  ShoppingCartMain.java
```

2) Main.java File



```
public class Main {
    public static void main(String[] args) {

        //Create a new product
       Product p1 = new Product("iPhone", "New iPhone", 999.99, 10);

       //Create a new Inventory
        Inventory inventory = new Inventory();

        // Add the product to the inventory
        inventory.addProduct(p1, 10);

        // Create a new customer
        Customer c1 = new Customer("John Smith", "123 Main St", "johnsmith@email.com");

        // Create a new shopping cart
        ShoppingCart cart = new ShoppingCart();

        // Add the product to the cart
        cart.addProduct(p1);

        // Check if the product is in stock
        if (inventory.inStock(p1)) {
            System.out.println("Product is in stock");
        } else {
            System.out.println("Product is out of stock");
        }

        // Print the total cost of items in the cart
        System.out.println("Total cost: Rs "+cart.totalCost());

        // Create a new payment
        Payment payment = new Payment(cart.totalCost(), "Credit Card");

        // Print the payment details
        System.out.println("Payment amount: Rs "+payment.getAmount());
        System.out.println("Payment method: "+payment.getPaymentMethod());

    }
}
```

1. **Product Creation:**

- `Product p1 = new Product("iPhone", "New iPhone", 999.99, 10);`
  - This code creates a new product named "iPhone" with a description, price of Rs 999.99, and an initial stock level of 10.

2. **Inventory Creation:**

- `Inventory inventory = new Inventory();`

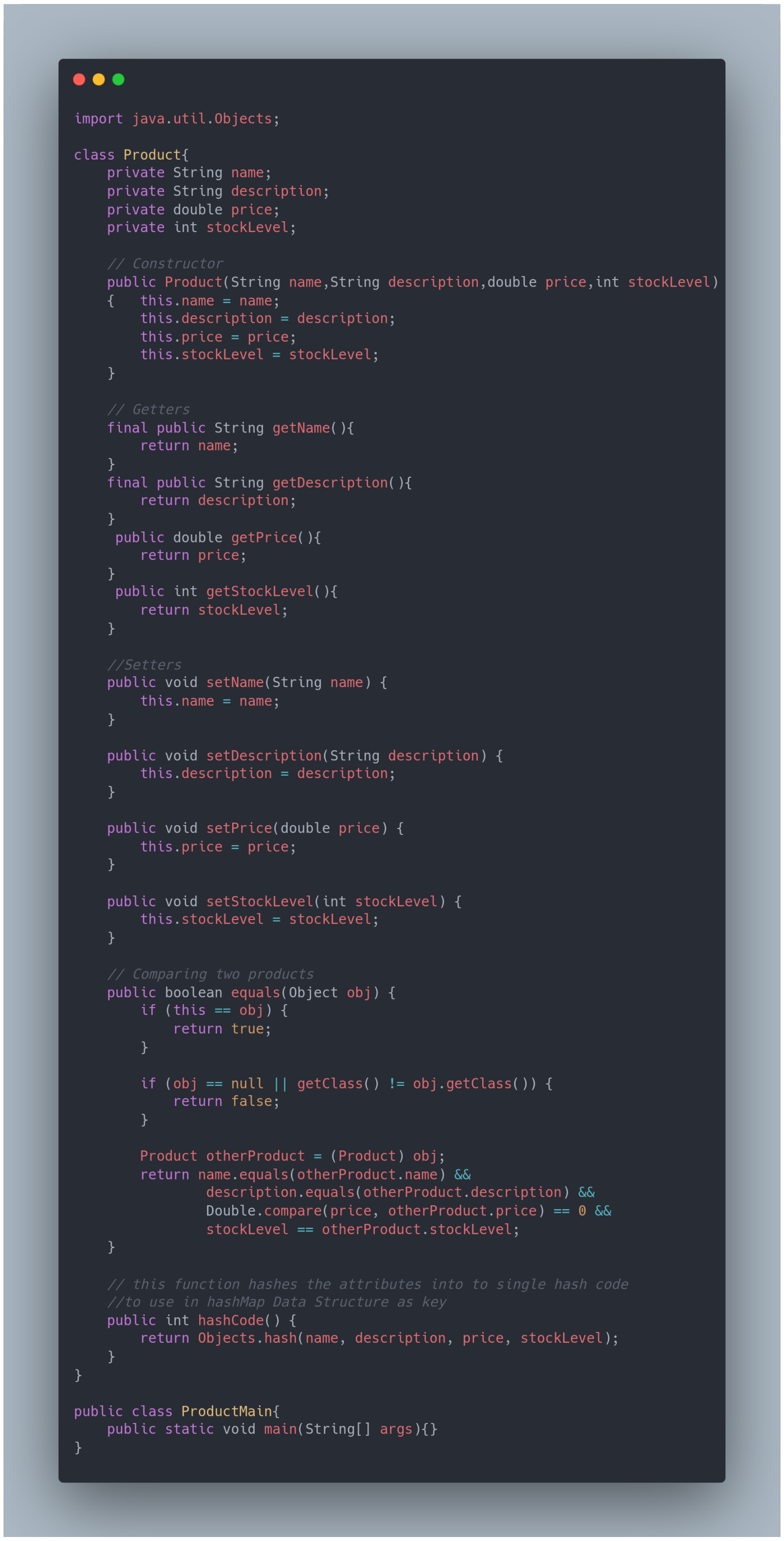
    - An inventory object is created, representing a store's product inventory.

3. **Adding Product to Inventory:**
   - `inventory.addProduct(p1, 10);`
     - The product created earlier (iPhone) is added to the inventory with an initial stock quantity of 10.
4. **Customer Creation:**
   - `Customer c1 = new Customer("John Smith", "123 Main St", "johnsmith@email.com");`
     - A customer object is created with the name "John Smith," an address, and an email address.
5. **Shopping Cart Creation:**
   - `ShoppingCart cart = new ShoppingCart();`
     - A shopping cart object is created to hold selected products.
6. **Adding Product to Cart:**
   - `cart.addProduct(p1);`
     - The iPhone product is added to the customer's shopping cart.
7. **Checking Product Stock:**
   - `if (inventory.inStock(p1)) { ... }`
     - This code checks if the product (iPhone) is in stock within the inventory. If it is, it prints "Product is in stock"; otherwise, it prints "Product is out of stock."
8. **Calculating Total Cart Cost:**
   - `System.out.println("Total cost: Rs " + cart.totalCost());`
     - The code calculates and prints the total cost of the items in the shopping cart.
9. **Payment Creation:**
   - `Payment payment = new Payment(cart.totalCost(), "Credit Card");`
     - A payment object is created, representing a payment transaction for the total cart cost using the payment method "Credit Card."

10. **Printing Payment Details:**

- `System.out.println("Payment amount: Rs " + payment.getAmount());`
- `System.out.println("Payment method: " + payment.getPaymentMethod());`
- The code prints the payment amount and the chosen payment method.

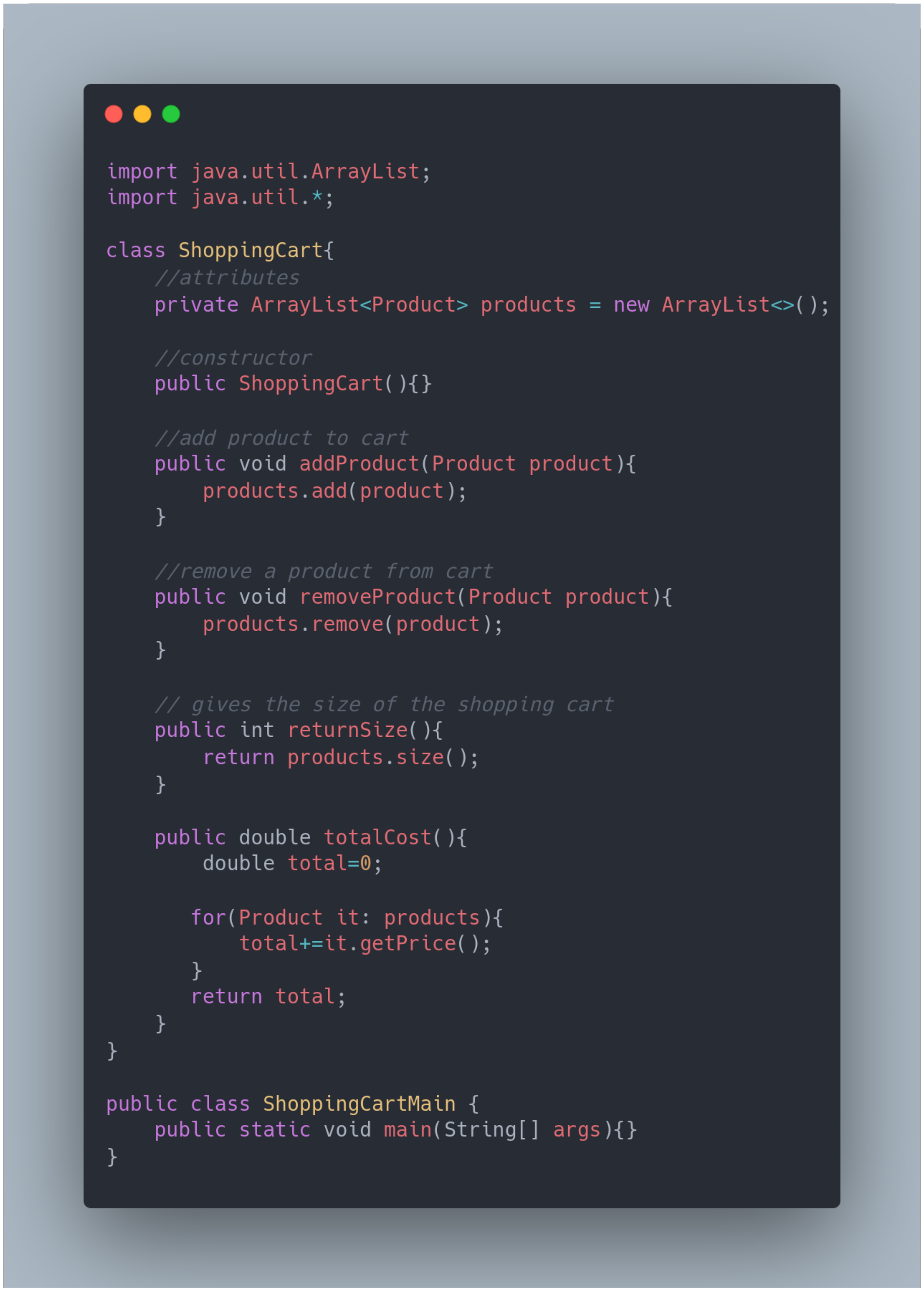
3) ProductMain.java File



```java
import java.util.Objects;

class Product{
    private String name;
    private String description;
    private double price;
    private int stockLevel;

    // Constructor
    public Product(String name,String description,double price,int stockLevel)
    {   this.name = name;
        this.description = description;
        this.price = price;
        this.stockLevel = stockLevel;
    }

    // Getters
    final public String getName(){
        return name;
    }
    final public String getDescription(){
        return description;
    }
     public double getPrice(){
        return price;
    }
     public int getStockLevel(){
        return stockLevel;
    }

    //Setters
    public void setName(String name) {
        this.name = name;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public void setStockLevel(int stockLevel) {
        this.stockLevel = stockLevel;
    }

    // Comparing two products
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }

        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Product otherProduct = (Product) obj;
        return name.equals(otherProduct.name) &&
                description.equals(otherProduct.description) &&
                Double.compare(price, otherProduct.price) == 0 &&
                stockLevel == otherProduct.stockLevel;
    }

    // this function hashes the attributes into to single hash code
    //to use in hashMap Data Structure as key
    public int hashCode() {
        return Objects.hash(name, description, price, stockLevel);
    }
}

public class ProductMain{
    public static void main(String[] args){}
}
```

1. **Product Class**:
    - The `Product` class represents a product with attributes like name, description, price, and stock level.
    - It includes constructors to initialize these attributes when creating a product object.
2. **Equals Method**:
    - The `equals` method overrides the default `equals` method to compare two `Product` objects based on their attributes.
    - It checks if the compared objects are the same, their classes match, and if their attributes (name, description, price, stock level) are equal.
3. **Hash Code Method**:
    - The `hashCode` method computes a hash code based on the product's attributes (name, description, price, stock level). It uses the `Objects.hash` method to create a unique hash code, useful for data structures like `HashMap` where the product can be used as a key.
4. **ProductMain Class**:
    - The `ProductMain` class serves as the entry point but currently contains no specific functionality.

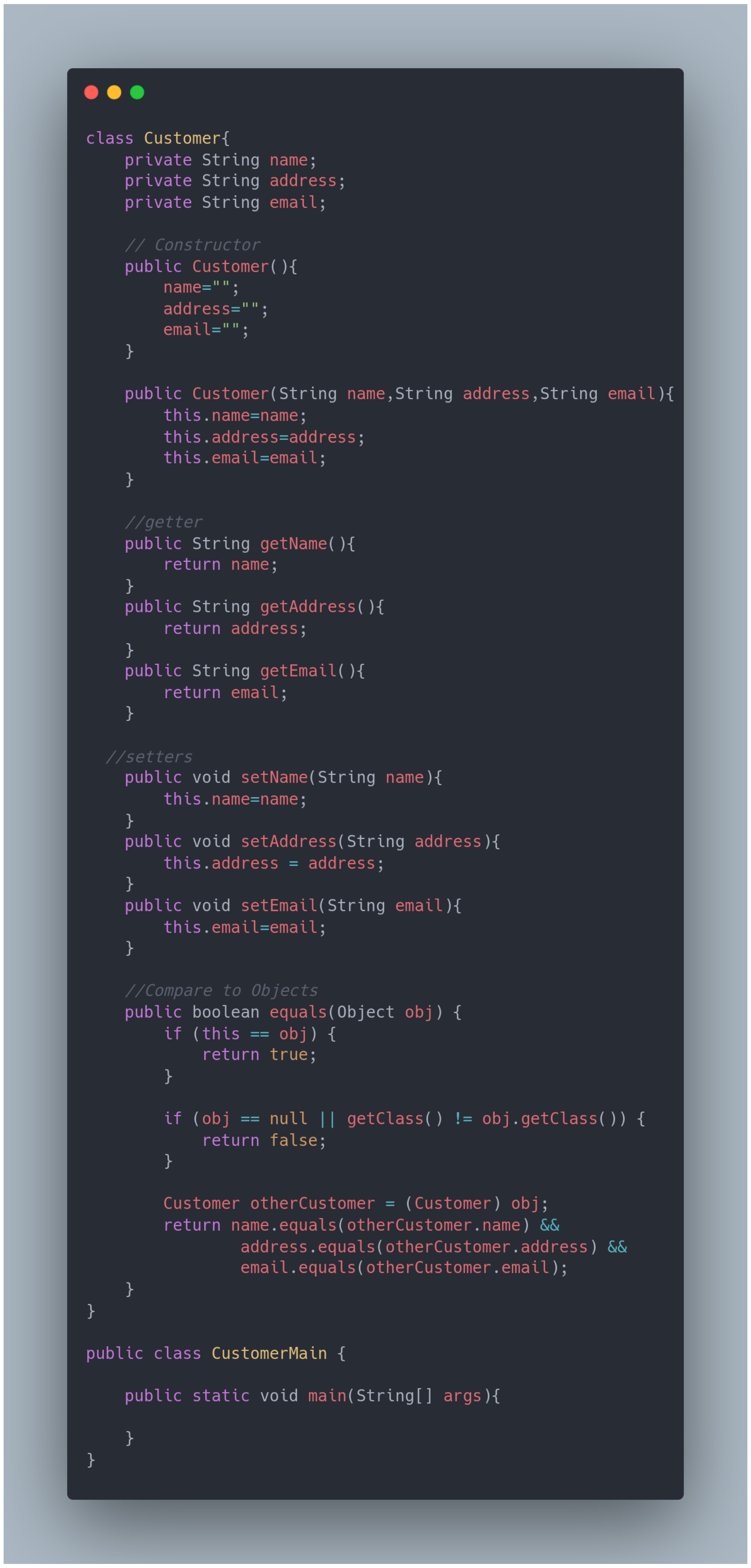
4)ShoppingCartMain.java File



```java
import java.util.ArrayList;
import java.util.*;

class ShoppingCart{
    //attributes
    private ArrayList<Product> products = new ArrayList<>();

    //constructor
    public ShoppingCart(){}

    //add product to cart
    public void addProduct(Product product){
        products.add(product);
    }

    //remove a product from cart
    public void removeProduct(Product product){
        products.remove(product);
    }

    // gives the size of the shopping cart
    public int returnSize(){
        return products.size();
    }

    public double totalCost(){
        double total=0;

       for(Product it: products){
           total+=it.getPrice();
       }
       return total;
    }
}

public class ShoppingCartMain {
    public static void main(String[] args){}
}
```

1. **ShoppingCart Class**:
    - The `ShoppingCart` class represents a shopping cart that can hold products.
    - It uses an `ArrayList` to store a collection of `Product` objects.
2. **Add and Remove Products**:
    - The `addProduct` method allows you to add a `Product` object to the shopping cart by appending it to the `products` list.
    - The `removeProduct` method enables you to remove a specified `Product` from the cart by removing it from the `products` list.
3. **Return Cart Size**:
    - The `returnSize` method returns the number of products in the cart, which is equivalent to the size of the `products` list.
4. **Calculate Total Cost**:
    - The `totalCost` method calculates the total cost of all the products in the cart.
    - It iterates through the `products` list, summing up the prices of each product to compute the total cost.

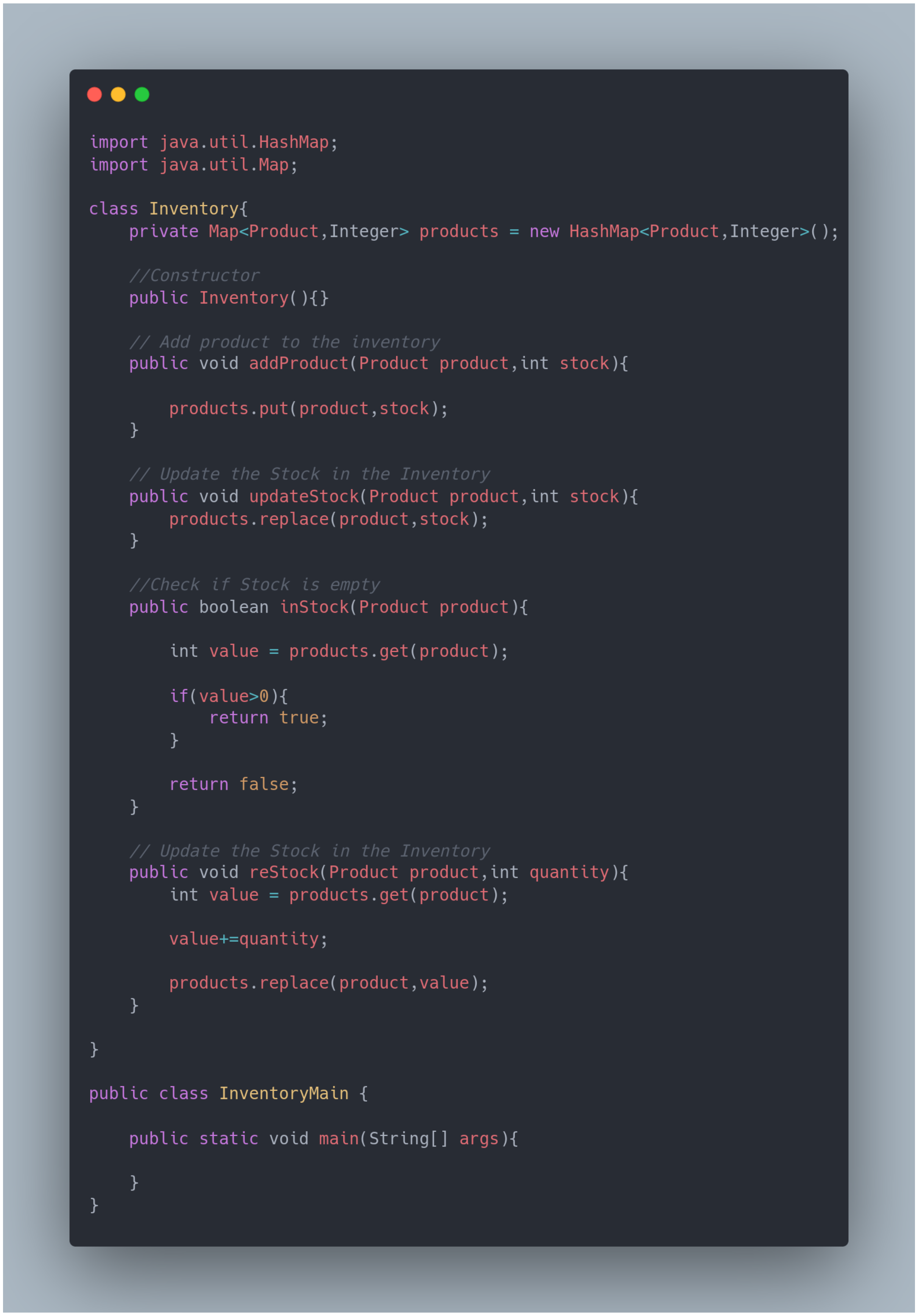
5)CustomerMain.java



```java
class Customer{
    private String name;
    private String address;
    private String email;

    // Constructor
    public Customer(){
        name="";
        address="";
        email="";
    }

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

    //getter
    public String getName(){
        return name;
    }
    public String getAddress(){
        return address;
    }
    public String getEmail(){
        return email;
    }

  //setters
    public void setName(String name){
        this.name=name;
    }
    public void setAddress(String address){
        this.address = address;
    }
    public void setEmail(String email){
        this.email=email;
    }

    //Compare to Objects
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }

        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Customer otherCustomer = (Customer) obj;
        return name.equals(otherCustomer.name) &&
                address.equals(otherCustomer.address) &&
                email.equals(otherCustomer.email);
    }
}

public class CustomerMain {

    public static void main(String[] args){

    }
}
```

1. **Customer Class**:
    - The `Customer` class represents a customer with attributes such as name, address, and email.
    - It includes constructors, getters, setters, and an `equals` method for comparing customer objects.
2. **Equals Method**:
    - The `equals` method overrides the default `equals` method to compare two `Customer` objects based on their attributes (name, address, email).
    - It checks if the compared objects are the same, their classes match, and if their attributes are equal.
3. **CustomerMain Class**:
    - The `CustomerMain` class serves as the entry point for the program.
    - It is currently empty and can be used to create and manipulate customer objects.

6) InventoryMain.java



```java
import java.util.HashMap;
import java.util.Map;

class Inventory{
    private Map<Product,Integer> products = new HashMap<Product,Integer>();

    //Constructor
    public Inventory(){}

    // Add product to the inventory
    public void addProduct(Product product,int stock){

        products.put(product,stock);
    }

    // Update the Stock in the Inventory
    public void updateStock(Product product,int stock){
        products.replace(product,stock);
    }

    //Check if Stock is empty
    public boolean inStock(Product product){

        int value = products.get(product);

        if(value>0){
            return true;
        }

        return false;
    }

    // Update the Stock in the Inventory
    public void reStock(Product product,int quantity){
        int value = products.get(product);

        value+=quantity;

        products.replace(product,value);
    }

}

public class InventoryMain {

    public static void main(String[] args){

    }
}
```

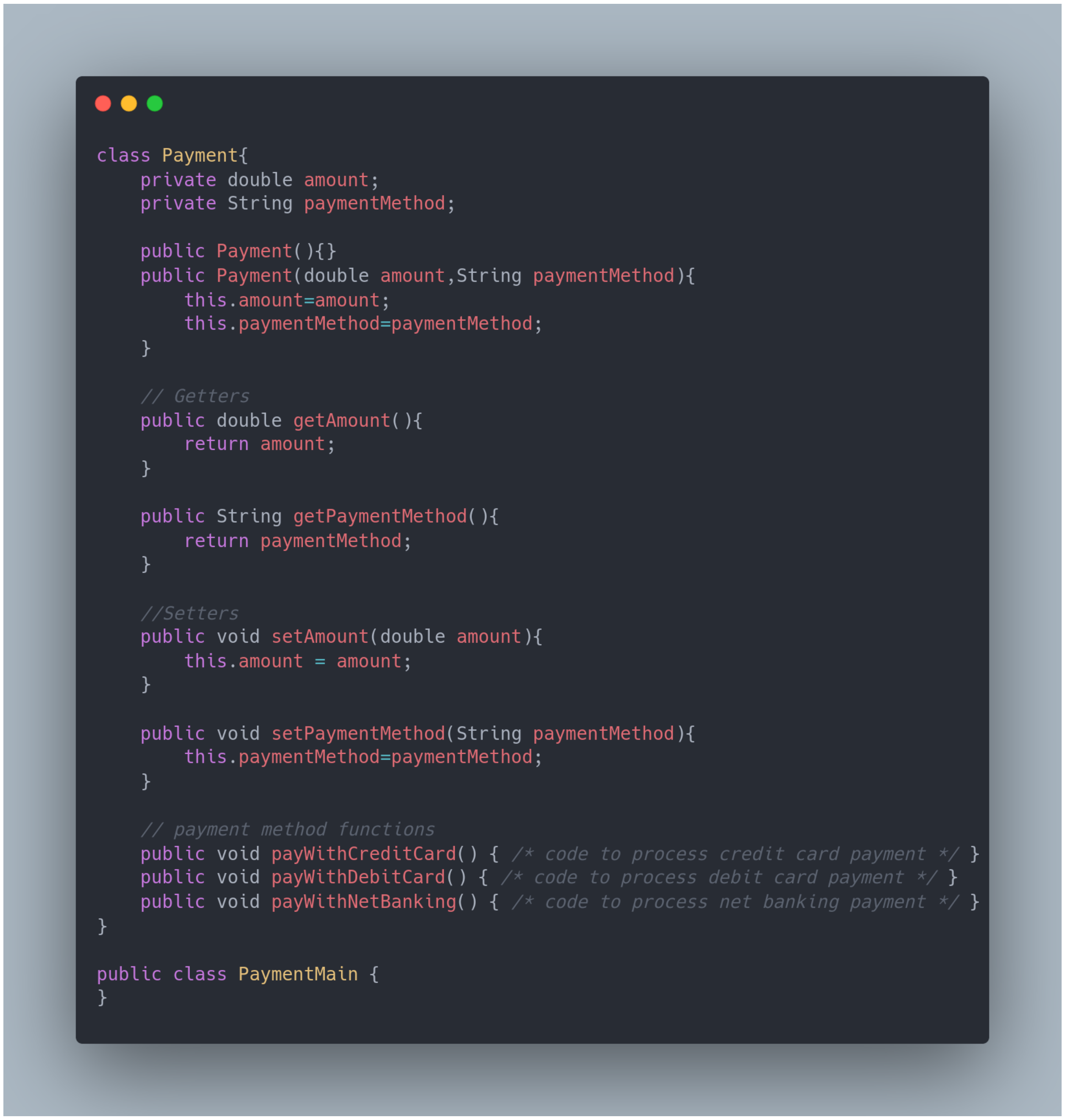
1. **Inventory Class**:
    - The `Inventory` class represents an inventory system that tracks products and their stock levels using a `Map` where `Product` objects are keys, and integers represent the stock levels.
    - It provides methods for adding products to the inventory, updating stock levels, checking if a product is in stock, and restocking products.
2. **Add Product to Inventory**:
    - The `addProduct` method allows you to add a `Product` to the inventory with an initial stock level. It puts the product into the `Map` with the specified stock level.
3. **Update Stock**:
    - The `updateStock` method updates the stock level of a specific product in the inventory. It uses the `replace` method of the `Map` to update the stock level associated with the product.
4. **Check if Product is in Stock**:
    - The `inStock` method checks if a product is in stock by looking up its stock level in the `Map`. If the stock level is greater than zero, it returns `true`; otherwise, it returns `false`.
5. **Restock Product**:
    - The `reStock` method allows you to increase the stock level of a product in the inventory by a specified quantity. It retrieves the current stock level from the `Map`, adds the quantity, and then updates the stock level in the `Map`.
6. **InventoryMain Class**:
    - The `InventoryMain` class serves as the entry point for the program but is currently empty and can be used for testing the inventory functionality.

7) PaymentMain.java File

1. **Payment Class**:

- The `Payment` class represents a payment transaction, including attributes for the payment amount and payment method.
- It provides constructors, getters, setters, and methods for different payment methods (e.g., credit card, debit card, net banking).

2. **Payment Method Functions**:
    - The class includes placeholder methods for different payment methods (`payWithCreditCard`, `payWithDebitCard`, `payWithNetBanking`). These methods are meant to contain the code for processing payments using the respective methods.
3. **PaymentMain Class**:
    - The `PaymentMain` class serves as the entry point for the program but is currently empty. You can use this class to test payment-related functionality.



```java
class Payment{
    private double amount;
    private String paymentMethod;

    public Payment(){}
    public Payment(double amount,String paymentMethod){
        this.amount=amount;
        this.paymentMethod=paymentMethod;
    }

    // Getters
    public double getAmount(){
        return amount;
    }

    public String getPaymentMethod(){
        return paymentMethod;
    }

    //Setters
    public void setAmount(double amount){
        this.amount = amount;
    }

    public void setPaymentMethod(String paymentMethod){
        this.paymentMethod=paymentMethod;
    }

    // payment method functions
    public void payWithCreditCard() { /* code to process credit card payment */ }
    public void payWithDebitCard() { /* code to process debit card payment */ }
    public void payWithNetBanking() { /* code to process net banking payment */ }
}

public class PaymentMain {
}
```

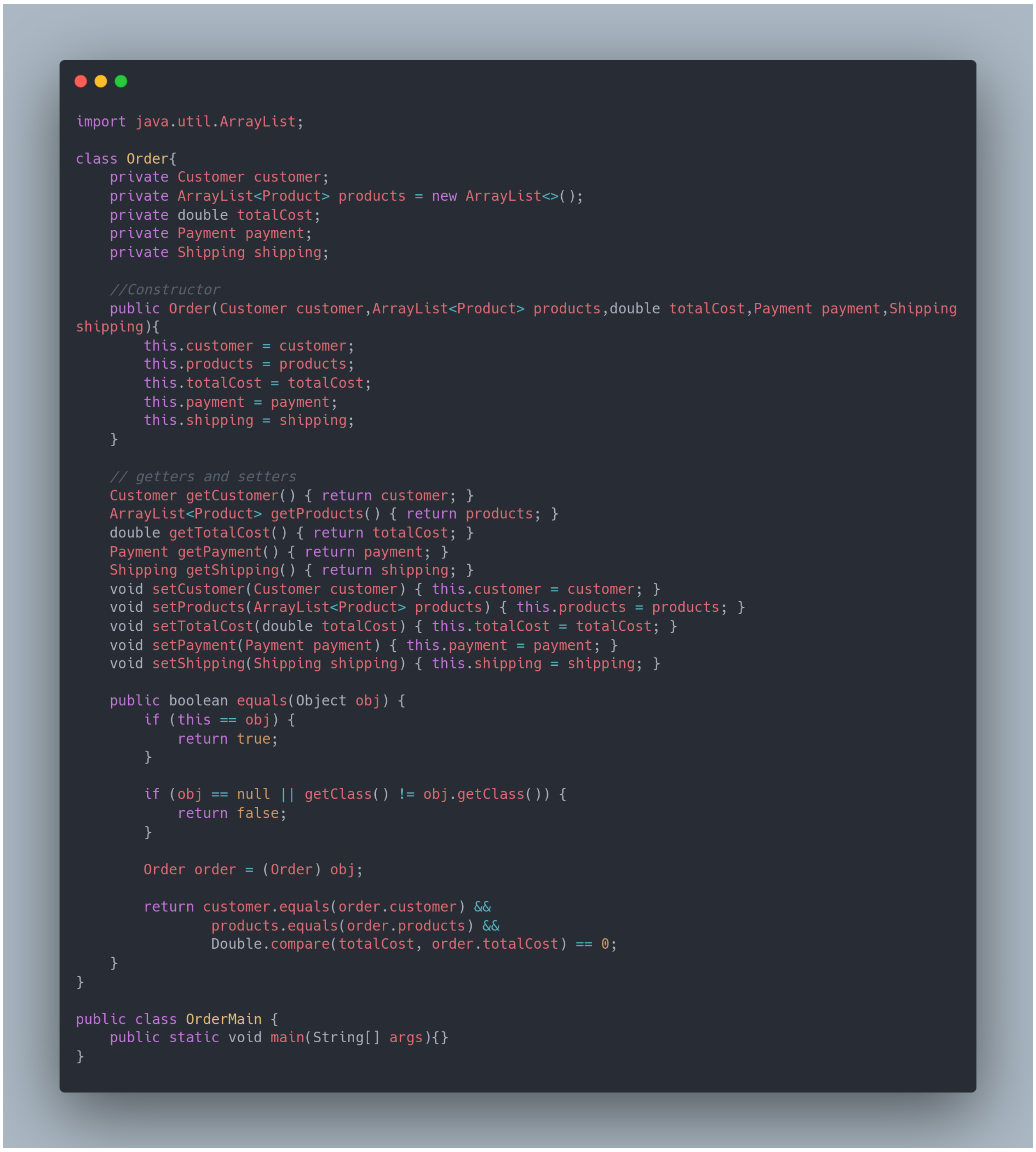
8)ShippingMain.java

1. **Shipping Class**:
    - The `Shipping` class represents shipping information, including attributes for the shipping method and shipping cost.

- It provides constructors, getters, and setters for these attributes.

2. **ShippingMain Class**:
    - The `ShippingMain` class serves as the entry point for the program but is currently empty. You can use this class to test shipping-related functionality.

9)OrderMain.java



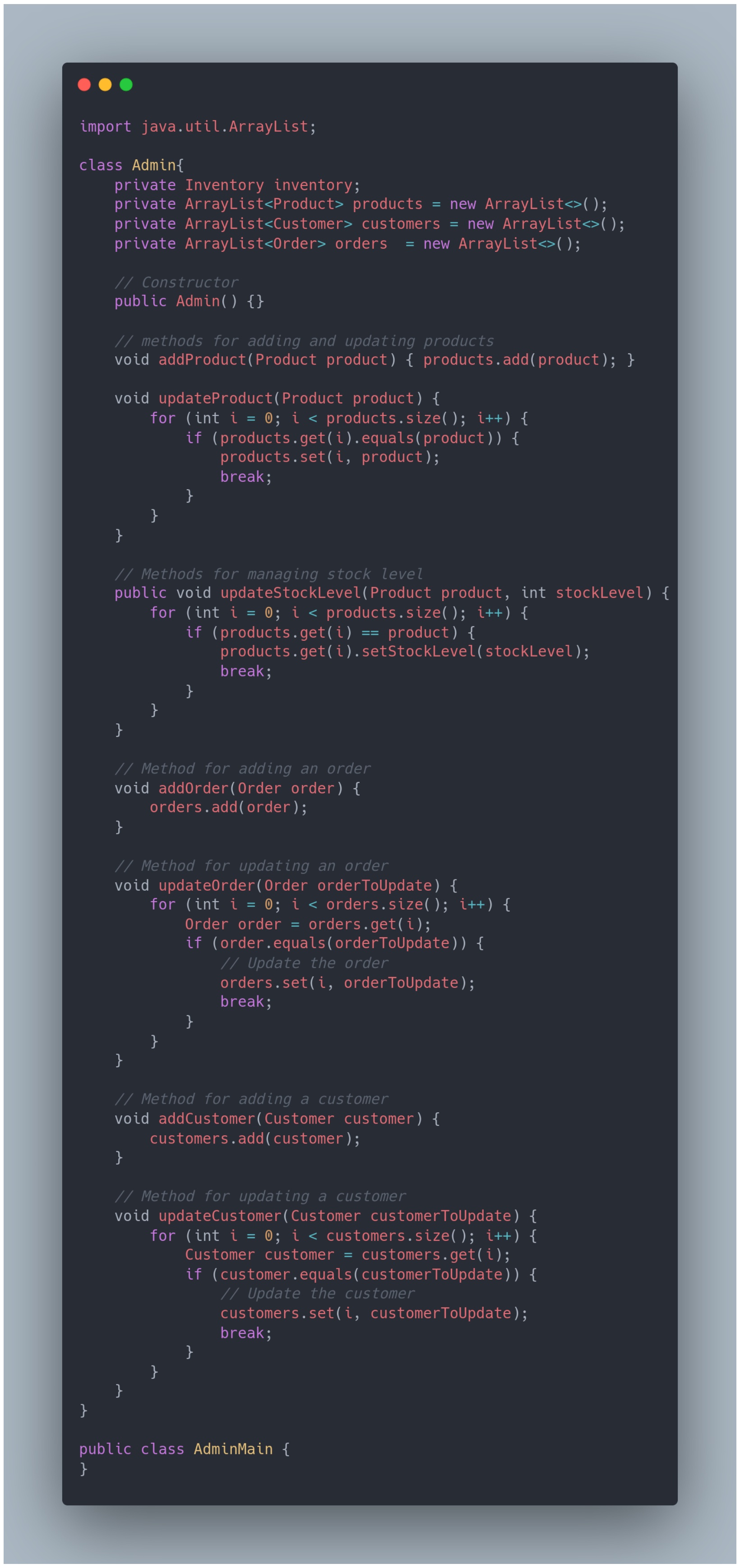
```java
import java.util.ArrayList;

class Order{
    private Customer customer;
    private ArrayList<Product> products = new ArrayList<>();
    private double totalCost;
    private Payment payment;
    private Shipping shipping;

    //Constructor
    public Order(Customer customer,ArrayList<Product> products,double totalCost,Payment payment,Shipping
shipping){
        this.customer = customer;
        this.products = products;
        this.totalCost = totalCost;
        this.payment = payment;
        this.shipping = shipping;
    }

    // getters and setters
    Customer getCustomer() { return customer; }
    ArrayList<Product> getProducts() { return products; }
    double getTotalCost() { return totalCost; }
    Payment getPayment() { return payment; }
    Shipping getShipping() { return shipping; }
    void setCustomer(Customer customer) { this.customer = customer; }
    void setProducts(ArrayList<Product> products) { this.products = products; }
    void setTotalCost(double totalCost) { this.totalCost = totalCost; }
    void setPayment(Payment payment) { this.payment = payment; }
    void setShipping(Shipping shipping) { this.shipping = shipping; }

    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }

        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Order order = (Order) obj;

        return customer.equals(order.customer) &&
                products.equals(order.products) &&
                Double.compare(totalCost, order.totalCost) == 0;
    }
}

public class OrderMain {
    public static void main(String[] args){}
}
```

1. **Order Class**:

   - The `Order` class represents an order placed by a customer in an online shopping system. It contains attributes for customer information, products in the order, total cost, payment information, and shipping information.
   - It provides constructors, getters, setters, and an `equals` method.

2. **Equals Method**:
    - The `equals` method is overridden to compare two `Order` objects for equality. It checks if the customer, products, and total cost of two orders are the same.
3. **OrderMain Class**:
    - The `OrderMain` class serves as the entry point for the program but is currently empty. You can use this class to test order-related functionality.

10)AdminMain.java



```java
import java.util.ArrayList;

class Admin{
    private Inventory inventory;
    private ArrayList<Product> products = new ArrayList<>();
    private ArrayList<Customer> customers = new ArrayList<>();
    private ArrayList<Order> orders  = new ArrayList<>();

    // Constructor
    public Admin() {}

    // methods for adding and updating products
    void addProduct(Product product) { products.add(product); }

    void updateProduct(Product product) {
        for (int i = 0; i < products.size(); i++) {
            if (products.get(i).equals(product)) {
                products.set(i, product);
                break;
            }
        }
    }

    // Methods for managing stock level
    public void updateStockLevel(Product product, int stockLevel) {
        for (int i = 0; i < products.size(); i++) {
            if (products.get(i) == product) {
                products.get(i).setStockLevel(stockLevel);
                break;
            }
        }
    }

    // Method for adding an order
    void addOrder(Order order) {
        orders.add(order);
    }

    // Method for updating an order
    void updateOrder(Order orderToUpdate) {
        for (int i = 0; i < orders.size(); i++) {
            Order order = orders.get(i);
            if (order.equals(orderToUpdate)) {
                // Update the order
                orders.set(i, orderToUpdate);
                break;
            }
        }
    }

    // Method for adding a customer
    void addCustomer(Customer customer) {
        customers.add(customer);
    }

    // Method for updating a customer
    void updateCustomer(Customer customerToUpdate) {
        for (int i = 0; i < customers.size(); i++) {
            Customer customer = customers.get(i);
            if (customer.equals(customerToUpdate)) {
                // Update the customer
                customers.set(i, customerToUpdate);
                break;
            }
        }
    }
}

public class AdminMain {
}
```

1. **Admin Class**:
    - The `Admin` class represents an administrator's interface for managing an inventory system in an online shopping application. It contains attributes for inventory, lists of products, customers, and orders.
2. **Methods for Adding and Updating Products**:
    - `void addProduct(Product product)`: This method allows adding a new product to the list of products in the inventory.
    - `void updateProduct(Product product)`: It updates an existing product in the list of products by comparing and replacing it.
3. **Methods for Managing Stock Level**:
    - `public void updateStockLevel(Product product, int stockLevel)`: This method updates the stock level of a specific product in the inventory. It finds the matching product by reference and updates its stock level.
4. **Methods for Managing Orders**:
    - `void addOrder(Order order)`: Allows adding a new order to the list of orders.
    - `void updateOrder(Order orderToUpdate)`: Updates an existing order in the list of orders by comparing and replacing it.
5. **Methods for Managing Customers**:
    - `void addCustomer(Customer customer)`: Adds a new customer to the list of customers.
    - `void updateCustomer(Customer customerToUpdate)`: Updates an existing customer in the list of customers by comparing and replacing it.
6. **AdminMain Class**:
    - The `AdminMain` class serves as the entry point for the program but is currently empty. You can use this class to initiate admin-related functionality.