Name: Mariam Ibrahim

Date: November 11, 2023

Milestone 3: Inventory Manager and Shopping Cart

Code: https://onlinegdb.com/yVA0JNgqX

Analyze and model out, using UML class diagrams, an Inventory Manager that supports the following features:
Initialization of the Store inventory (should be invoked when the Store Front starts up).
Removing a Salable Product from Store inventory (should be invoked when a Salable Product is purchased).
Adding a Salable Product to Store inventory (should be invoked when a Salable Product purchase is canceled).
Return the entire inventory. Integration of this class with the Store Front application.

Store Class:

- Represents the store that contains an inventory.
- Has a private attribute inventory which is a List of SalableProducts.
- Includes a method getInventory() to retrieve the entire inventory.

InventoryManager Class:

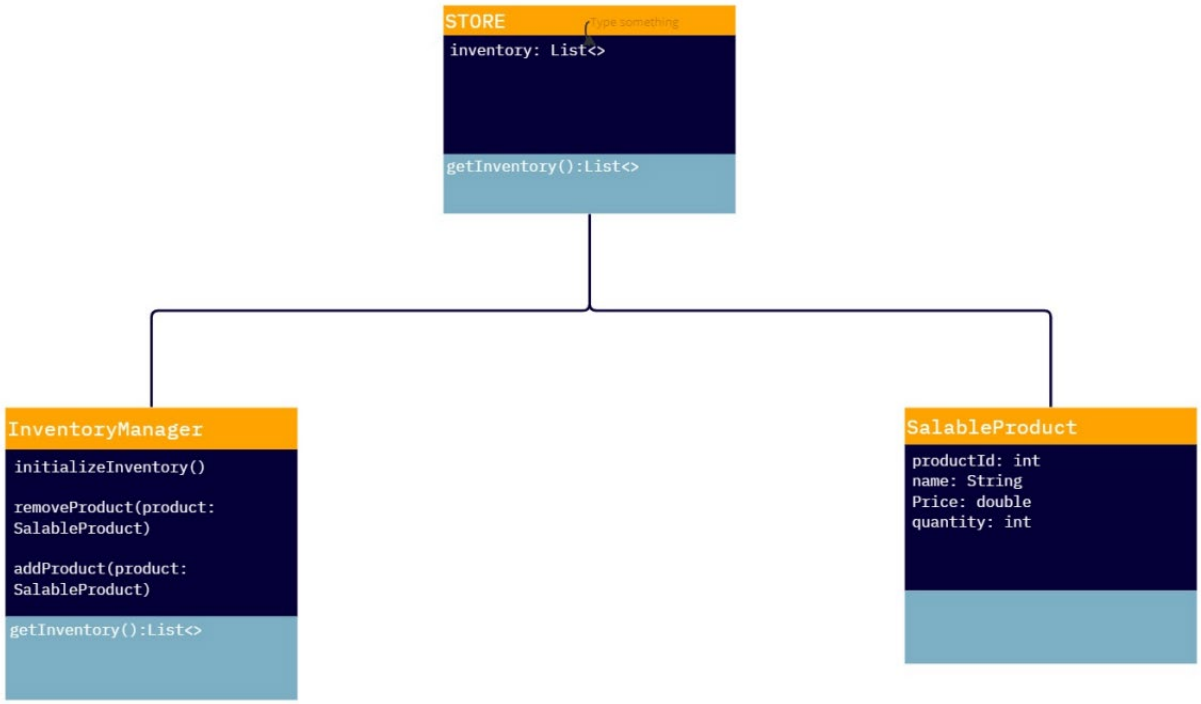
- Manages the inventory of the store.
- Contains methods:
    - **initializeInventory():** Initializes the store inventory. This method should be invoked when the store front starts up.
    - **removeProduct(product: SalableProduct):** Removes a salable product from the inventory when it is purchased.
    - **addProduct(product: SalableProduct):** Adds a salable product to the inventory when a purchase is canceled.
    - **getInventory():** List: Returns the entire inventory.

SalableProduct Class:

- Represents a salable product with attributes like productId, name, price, and quantity.

Integration with Store Front:

- The integration with the Store Front application would involve creating an instance of the InventoryManager within the Store Front application and invoking its methods as needed. For example, calling initializeInventory() when the store front starts up, and calling removeProduct() or addProduct() when a purchase is made or canceled.



Analyze and model out, using UML class diagrams, a Shopping Cart that supports the following features:
Initialization of the Shopping Cart (should be invoked when the Store Front starts up).
Adding a Salable Product to the Shopping Cart (should be invoked when a Salable Product is purchased).
Removing a Salable Product from the Shopping Cart (should be invoked when a Salable Product purchase is canceled).
Return the contents of the Shopping Cart.
Empty the contents of the Shopping Cart.
Integration of this class with the Store Front application.

ShoppingCart Class:

- Represents the shopping cart that contains a list of cart items.
- Has a private attribute cartItems, which is a List of SalableProducts.
- Includes methods:
    - **addProduct(product: SalableProduct**): Adds a salable product to the shopping cart when it is purchased.
    - **removeProduct(product: SalableProduct):** Removes a salable product from the shopping cart when a purchase is canceled.
    - **getCartContents(): List<SalableProduct>:** Returns the contents of the shopping cart.
    - **emptyCart():** Empties the contents of the shopping cart.
    - **initializeCart():** Initializes the shopping cart. This method should be invoked when the store front starts up.

StoreFrontApplication Class:

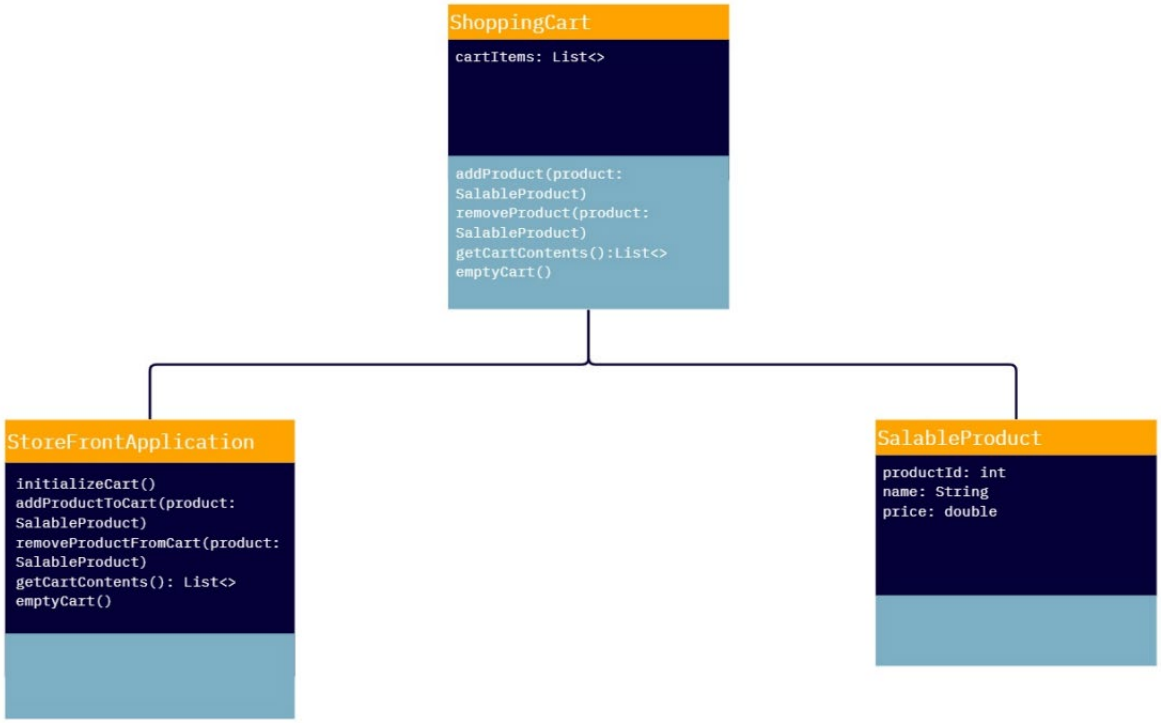
- Represents the Store Front application.
- Integrates with the ShoppingCart class.

SalableProduct Class:

- Represents a salable product with attributes like productId, name, and price.

Integration with Store Front:

- The integration with the Store Front application involves creating an instance of the ShoppingCart within the Store Front application and invoking its methods as needed. For example, calling initializeCart() when the store front starts up, and calling addProduct(), removeProduct(), getCartContents(), or emptyCart() based on user actions.



Update the Salable Product UML class diagrams to support the following new features:
Update all Weapon classes so that they implement the comparable interface. Comparison should be based on the name of the item and follow alphabetical ordering rules that ignore case.

Comparable Interface:

- Added a Comparable interface with a single method compareTo(other: Comparable): int. This interface will be implemented by the SalableProduct class and its subclasses to provide a natural ordering based on the name of the item.

SalableProduct Class:

- Now implements the Comparable interface.
- Includes the compareTo(other: Comparable): int method.

AbstractWeapon Class:

- Extends SalableProduct and implements the Comparable interface.
- Includes the compareTo(other: Comparable): int method.

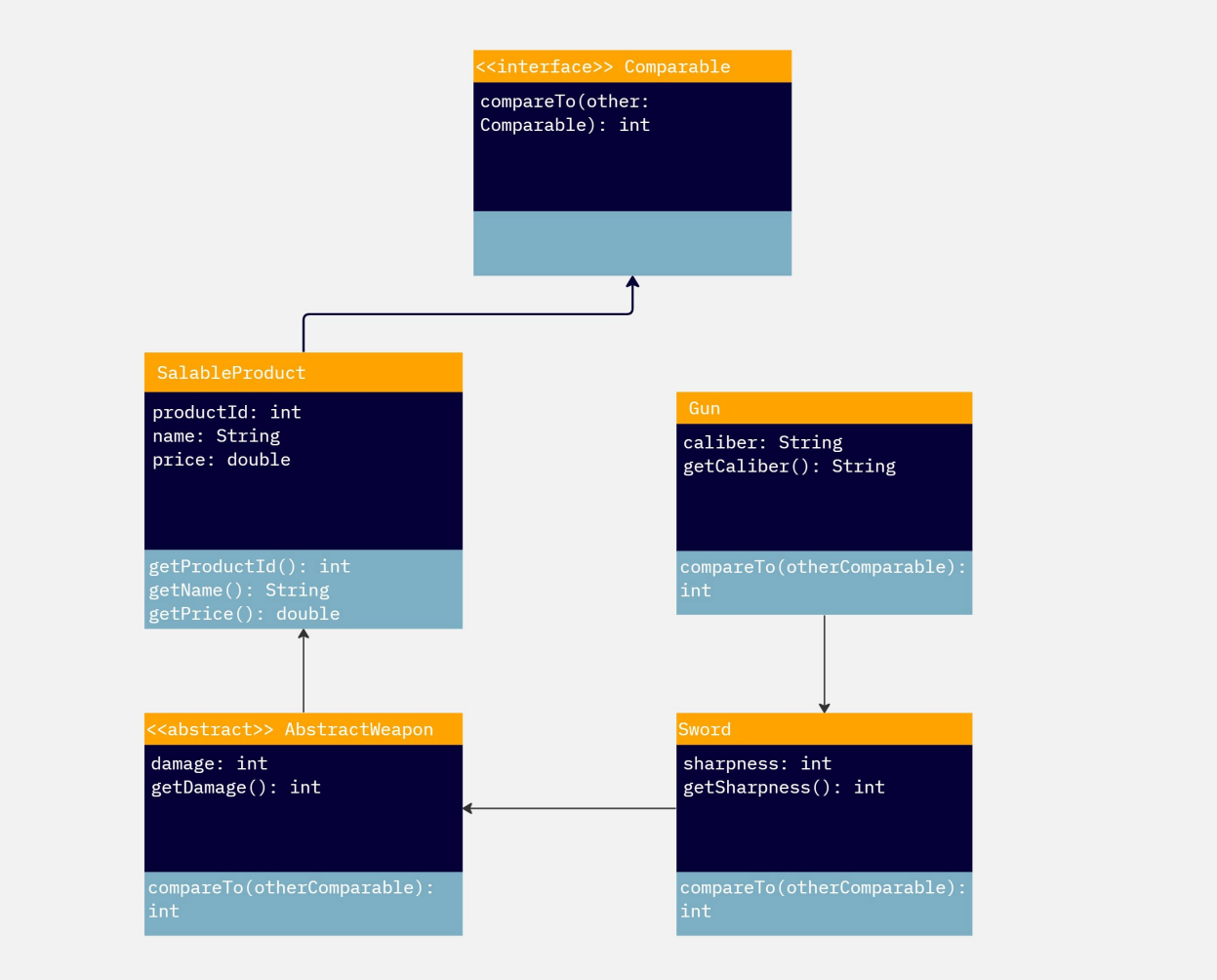
Sword Class:

- Extends AbstractWeapon and also implements the Comparable interface.
- Includes the compareTo(other: Comparable): int method.

Gun Class:

- Extends AbstractWeapon and implements the Comparable interface.
- Includes the compareTo(other: Comparable): int method.

Now, instances of SalableProduct, AbstractWeapon, Sword, and Gun can be compared based on their names using the compareTo method, following alphabetical ordering rules that ignore case.



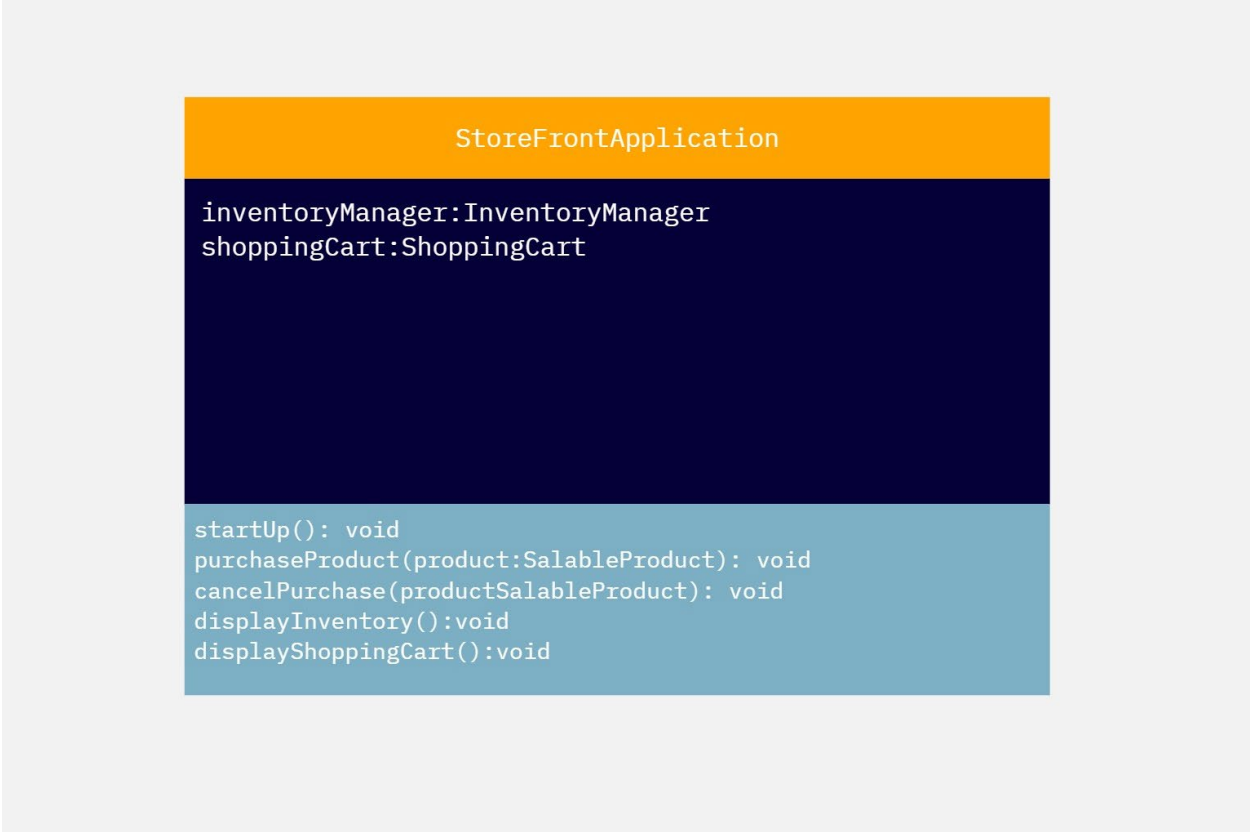
Update the Store Front UML class diagrams to support the following new features: Integration of the Inventory Manager. Integration of the Shopping Cart.

StoreFrontApplication Class:

- Added two private attributes:
  - **inventoryManager:** An instance of the InventoryManager class to manage the store inventory.
  - **shoppingCart:** An instance of the ShoppingCart class to handle shopping cart operations.

- Updated the startUp(), purchaseProduct(product: SalableProduct), and cancelPurchase(product: SalableProduct) methods to interact with the InventoryManager and ShoppingCart as needed. Added two new methods:
    - **displayInventory():** Displays the store inventory, which would involve interacting with the InventoryManager.
    - **displayShoppingCart():** Displays the contents of the shopping cart, which would involve interacting with the ShoppingCart.

This diagram shows how the StoreFrontApplication class integrates with the InventoryManager and ShoppingCart to manage store operations. Depending on your implementation, you may need additional methods or details to fully capture the interactions between these components



Update the flow chart of the logic of a Game User interacting with the Store Front and with the internal Inventory Manager and Shopping Cart.

Game User Interaction:

- The process begins with the game user interacting with the store.

Store Front:

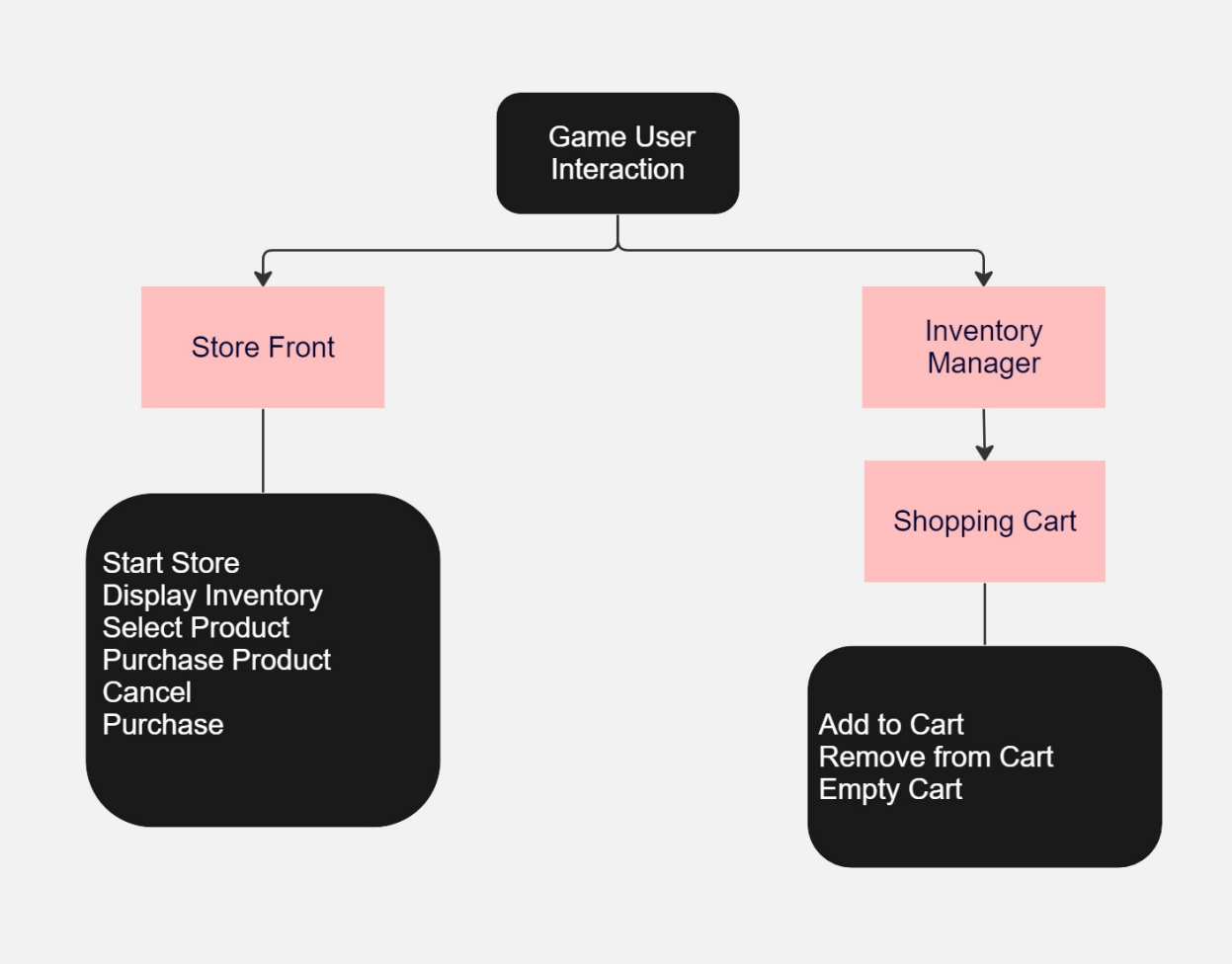
- The user starts the store, and the inventory is displayed.
- The user can select a product to purchase or cancel a purchase.
- The cart contents can be displayed.

Inventory Manager:

- The Inventory Manager initializes the store inventory.
- It manages the addition and removal of products from the store inventory.

Shopping Cart:

- The Shopping Cart handles adding products to the cart, removing them, and emptying the cart. The cart contents can be displayed



- Implement the code for all UML class designs.

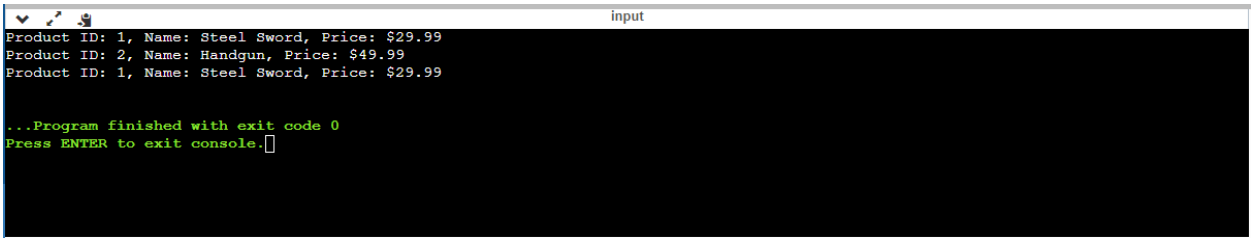
StoreFrontApplicati...

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

// Comparable interface for comparing SalableProducts based on name
interface Comparable<T> {
    int compareTo(T other);
}

// SalableProduct class
class SalableProduct implements Comparable<SalableProduct> {
    private int productId;
    private String name;
    private double price;

    public SalableProduct(int productId, String name, double price) {
        this.productId = productId;
        this.name = name;
        this.price = price;
    }

    public int getProductId() {
        return productId;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }

    // Implementing compareTo based on name
    @Override
    public int compareTo(SalableProduct other) {
        return this.name.compareToIgnoreCase(other.getName());
    }
}

// AbstractWeapon class extending SalableProduct
abstract class AbstractWeapon extends SalableProduct {
    private int damage;

    public AbstractWeapon(int productId, String name, double price, int damage) {
        super(productId, name, price);
        this.damage = damage;
    }

    public int getDamage() {
        return damage;
    }
}

// Sword class extending AbstractWeapon
class Sword extends AbstractWeapon {
    private int sharpness;

    public Sword(int productId, String name, double price, int damage, int sharpness) {
        super(productId, name, price, damage);
        this.sharpness = sharpness;
    }
```

```
    public int getSharpness() {
        return sharpness;
    }
}

// Gun class extending AbstractWeapon
class Gun extends AbstractWeapon {
    private String caliber;

    public Gun(int productId, String name, double price, int damage, String caliber) {
        super(productId, name, price, damage);
        this.caliber = caliber;
    }

    public String getCaliber() {
        return caliber;
    }
}

// InventoryManager class
class InventoryManager {
    private List<SalableProduct> inventory;

    public InventoryManager() {
        this.inventory = new ArrayList<>();
    }

    public void initializeInventory() {
        // Initialize the inventory (populate with initial products)
        // For simplicity, we'll add a few products manually
        inventory.add(new Sword(1, "Steel Sword", 29.99, 15, 5));
        inventory.add(new Gun(2, "Handgun", 49.99, 10, "9mm"));
    }

    public void removeProduct(SalableProduct product) {
        inventory.remove(product);
    }

    public void addProduct(SalableProduct product) {
        inventory.add(product);
    }

    public List<SalableProduct> getInventory() {
        return Collections.unmodifiableList(inventory);
    }
}

// ShoppingCart class
class ShoppingCart {
    private List<SalableProduct> cartItems;

    public ShoppingCart() {
        this.cartItems = new ArrayList<>();
    }

    public void addProduct(SalableProduct product) {
        cartItems.add(product);
    }

    public void removeProduct(SalableProduct product) {
        cartItems.remove(product);
    }

    public List<SalableProduct> getCartContents() {
        return Collections.unmodifiableList(cartItems);
    }
```

```
    public void emptyCart() {
        cartItems.clear();
    }
}

// StoreFrontApplication class
public class StoreFrontApplication {
    private InventoryManager inventoryManager;
    private ShoppingCart shoppingCart;

    public StoreFrontApplication() {
        this.inventoryManager = new InventoryManager();
        this.shoppingCart = new ShoppingCart();
    }

    public void startUp() {
        // Initialize the inventory when the store starts up
        inventoryManager.initializeInventory();
    }

    public void displayInventory() {
        // Display the store inventory
        List<SalableProduct> inventory = inventoryManager.getInventory();
        for (SalableProduct product : inventory) {
            System.out.println("Product ID: " + product.getProductId() +
                    ", Name: " + product.getName() +
                    ", Price: $" + product.getPrice());
        }
    }

    public void purchaseProduct(SalableProduct product) {
        // Add the product to the shopping cart and remove it from the inventory
        shoppingCart.addProduct(product);
        inventoryManager.removeProduct(product);
    }

    public void cancelPurchase(SalableProduct product) {
        // Remove the product from the shopping cart and add it back to the inventory
        shoppingCart.removeProduct(product);
        inventoryManager.addProduct(product);
    }

    public void displayShoppingCart() {
        // Display the contents of the shopping cart
        List<SalableProduct> cartContents = shoppingCart.getCartContents();
        for (SalableProduct product : cartContents) {
            System.out.println("Product ID: " + product.getProductId() +
                    ", Name: " + product.getName() +
                    ", Price: $" + product.getPrice());
        }
    }

    public static void main(String[] args) {
        // Example usage
        StoreFrontApplication storeFront = new StoreFrontApplication();
        storeFront.startUp();
        storeFront.displayInventory();

        // Simulate a user purchasing a product
        SalableProduct productToPurchase = storeFront.inventoryManager.getInventory().get(0);
        storeFront.purchaseProduct(productToPurchase);
        storeFront.displayShoppingCart();

        // Simulate a user canceling a purchase
        SalableProduct productToCancel = storeFront.shoppingCart.getCartContents().get(0);
        storeFront.cancelPurchase(productToCancel);
        storeFront.displayShoppingCart();
    }
}
```

Code Result:



What was challenging?
What did you learn?
If you had more time, how would you improve on the project
How can you use what you learned on the job.

Challenges:
Creating a comprehensive UML class diagram and implementing it in code requires careful consideration of relationships, responsibilities, and interactions between classes. Implementing the Comparable interface and ensuring proper sorting based on the name attribute can be a nuanced task, especially when dealing with complex class hierarchies. Integrating the InventoryManager and ShoppingCart into the StoreFrontApplication involves managing dependencies and ensuring smooth communication between different parts of the system.

Learnings:
Creating UML class diagrams helps reinforce understanding of design patterns, class relationships, and abstraction. Applying UML to a real-world scenario helps understand the practical challenges and considerations in software design.

Improvements with More Time:
Enhance error handling mechanisms, such as checking for null objects, validating inputs, and handling potential exceptions. Integrate logging mechanisms for better debugging and monitoring. User Interface: If applicable, develop a user interface for a more user-friendly experience.

Applying Learning on the Job:
Design Thinking: Understand the importance of thoughtful design in creating scalable and maintainable software systems. Experience with class diagrams and code implementation helps in collaborating with team members on complex projects. Developing the project enhances problem-solving skills, especially when it comes to debugging and improving code efficiency.