Supermarket Management Simulator

Team Information

Member1:

- **Name**: شهد محمد محمد حسن
- **Section**: 2

Member2:

- **Name**: منةالله حربي الشربيني رمضان
- **Section**: 5

What is the Supermarket Management Simulator?

- A Java-based interactive simulation game where players manage a virtual supermarket.
- Players make decisions to grow their business by expanding the supermarket, hiring employees, and restocking products.

Features of the Simulator

1. **Starting Resources**:
   - Players begin with a balance of **$1000**.
2. **Key Actions**:
   - **Expand Supermarket**: Increase capacity for growth but at a cost.
   - **Hire Employees**: Choose from various employees, balancing efficiency and salary.
   - **Restock and Sell Products**: Manage inventory to calculate profits strategically.
3. **Realistic Constraints**:
   - Expenses include rent, electricity, salaries, and restocking costs.
   - Players must stay within the budget to avoid bankruptcy.
4. **Product Options**:
   - Includes categories like Dairy, Snacks, Frozen Meat, and Soda, each with a fixed per-box cost.

Gameplay Mechanics

- **Main Menu Options**:
  1. **Expand the Supermarket** for $100: Increases capacity and overhead costs.
  2. **Hire Employees**: Improves efficiency based on the selected employee.
  3. **Restock Goods**: Input restocking quantities for each product category and calculate profits.
  4. **Exit the Simulator**: Ends the game.
- **Profit Calculation Formula**:

  Profit = Restocking Cost * Efficiency * 1.3 - Expenses
- **Game End**:
  - Players can exit anytime, or they may continue growing their business indefinitely.

Employees

Employee	Efficiency	Salary
Employee A	90%	$50
Employee B	85%	$40
Employee C	95%	$60

Decision Factor:

- Higher efficiency results in better profits but incurs higher salaries.

Product Options

Product	Cost Per Box ($)
Dairy	70
Snacks	50
Frozen Meat	150
Soda	40

Code Design

Programming Language:

- Java

Structure Overview:

1. **Initialization Block**:
    - Sets up the starting balance, expenses, and product/employee objects.
2. **Main Menu**:
    - Displays available actions and processes user choices in a loop.
3. **Action Blocks**:
    - Each menu option corresponds to a logical block for expanding, hiring, restocking, and exiting.
4. **Employee and Product Classes**:
    - Encapsulate employee attributes (efficiency, salary) and product details (name, cost).

Initialization Block

Code Snippet:

```
double money = 1000;
double rent = 30;
double electricity = 15;
double restockingCapacity = 1000;
Employee hiredEmployee = null;
Employee emp1 = new Employee("Employee A", 0.90, 50);
Employee emp2 = new Employee("Employee B", 0.85, 40);
Employee emp3 = new Employee("Employee C", 0.95, 60);
Product dairy = new Product("Dairy", 70);
Product snacks = new Product("Snacks", 50);
Product frozenMeat = new Product("Frozen Meat", 150);
Product soda = new Product("Soda", 40);
```

Explanation:

- Initializes game variables (balance, rent, electricity, etc.).
- Creates employee and product objects with specific attributes.

Main Menu Block

Code Snippet:

```
while (true) {
    System.out.println("\nYour current balance: $" + mone
y);
    System.out.println("1. Expand Supermarket");
    System.out.println("2. Hire Employee");
    System.out.println("3. Restock and Calculate Profit");
    System.out.println("4. Exit");
    int choice = sc.nextInt();

    switch (choice) {
        case 1:
            // Expand supermarket logic here
            break;
```

```
            case 2:
                // Hire employee logic here
                break;
            case 3:
                // Restock and profit calculation logic here
                break;
            case 4:
                // Exit logic here
                return;
            default:
                System.out.println("Invalid choice. Try agai
n.");
        }
    }
```

Explanation:

- Provides an interactive interface for the player to choose actions.
- Uses a loop to keep the game running until the player exits.

Employee Class Block

Code Snippet:

```
class Employee {
    String name;
    double efficiency;
    double salary;

    Employee(String name, double efficiency, double salary)
{
        this.name = name;
        this.efficiency = efficiency;
        this.salary = salary;
    }
}
```

Explanation:

- Defines attributes for employees: name, efficiency, and salary.

- Provides a structured way to handle employee data.

Product Class Block

Code Snippet:

```
class Product {
    String name;
    double cost;

    Product(String name, double cost) {
        this.name = name;
        this.cost = cost;
    }
}
```

Explanation:

- Encapsulates product details, such as name and cost.
- Allows easy addition of new products if needed.

Restock and Calculate Profit Block

Code Snippet:

```
if (choice == 3) {
    double totalGoodsCost = 0;
    for (Product product : products) {
        System.out.print("Enter the number of boxes for " +
product.name + " (Cost per box: $" + product.cost + "): ");
        int boxes = sc.nextInt();
        totalGoodsCost += product.cost * boxes;
    }

    if (totalGoodsCost > restockingCapacity || totalGoodsCo
st > money) {
        System.out.println("Restocking cost exceeds limit
s.");
    } else {
```

```
        double income = totalGoodsCost * hiredEmployee.effi
ciency * 1.3;
        double expenses = rent + electricity + totalGoodsCo
st + hiredEmployee.salary;
        double profit = income - expenses;
        money += profit;
        System.out.printf("Income: $%.2f, Expenses: $%.2f,
Profit: $%.2f\n", income, expenses, profit);
    }
}
```

Explanation:

- Calculates profit based on restocking costs, efficiency, and fixed expenses.
- Updates player balance with profit.

Example Gameplay

Scenario:

- Starting Balance: $1000
- Action: Hire Employee A
- Restocking: 10 boxes of Dairy ($70 each)
- Calculation:
  - Income: $819
  - Expenses: $715 (Rent: $30, Electricity: $15, Restocking: $700, Salary: $50)
  - **Profit**: $104
- Updated Balance: $1104
